FORCESPRO

FORCESPRO

Version 6.4.0

FORCESPRO User Manual

Embotech AG

Giessereistrasse 18

CH-8005 Zürich

www.embotech.com

info@embotech.com

Contents

1 Introduction **1**
1.1 Troubleshooting and support . . . 2
1.2 Licensing . . . 2
1.3 Citing FORCESPRO . . . 2
1.4 Product Life Cycle . . . 3
1.5 Release Notes . . . 4
1.6 Version history of manual . . . 17

2 License Variants **19**
2.1 Variant Summary . . . 19
2.2 Variant S . . . 20
2.3 Variant M . . . 20
2.4 Variant L . . . 20

3 Installation **23**
3.1 Obtaining FORCESPRO . . . 23
3.2 Installation of the MATLAB Client . . . 23
3.3 Installation of the Python Client . . . 25

4 Backward Compatibility **31**
4.1 Determining Client Version . . . 31
4.2 Changes from Version 6.4.0 . . . 32
4.3 Changes from Version 6.3.0 . . . 32
4.4 Changes from Version 6.2.0 . . . 32
4.5 Changes from Version 6.1.0 . . . 32
4.6 Changes from Version 6.0.0 . . . 33
4.7 Changes from Version 5.0.1 . . . 33
4.8 Changes from Version 5.0.0 . . . 34
4.9 Changes from Version 4.3.0 . . . 34
4.10 Changes from Version 4.2.0 . . . 34
4.11 Changes from Version 4.1.0 . . . 35

5 Y2F Interface **37**
5.1 Installing Y2F . . . 37
5.2 Generating a solver . . . 38
5.3 Calling the solver . . . 38
5.4 Solver info . . . 38
5.5 Performance . . . 39
5.6 Examples . . . 40

6 MathWorks Linear MPC Plugin **41**
6.1 Different types of solvers . . . 42
6.2 Different algorithms . . . 43
6.3 Generating a QP solver from an MPC object . . . 44
6.4 Solving a QP from MPC online data . . . 48
6.5 Using the FORCESPRO MPC Simulink block . . . 48
6.6 Deploy to dSPACE MicroAutoBox II using the FORCESPRO MPC Simulink block 51

6.7 Examples . . . 54

7 MathWorks Nonlinear MPC Plugin 73
7.1 Introduction . . . 73
7.2 The SQP Fast algorithm for nlmpc . . . 74
7.3 Defining a nonlinear model . . . 75
7.4 Generating an NLP solver . . . 76
7.5 Simulation in MATLAB and Simulink . . . 82
7.6 Code generation in MATLAB and Simulink . . . 82
7.7 Examples . . . 82

8 Low-level interface 95
8.1 Supported problem class . . . 96
8.2 Multistage struct . . . 96
8.3 Dimensions . . . 97
8.4 Cost function . . . 97
8.5 Equality constraints . . . 97
8.6 Lower and upper bounds . . . 98
8.7 Polytopic constraints . . . 99
8.8 Quadratic constraints . . . 99
8.9 Binary constraints . . . 100
8.10 Declaring parameters . . . 100
8.11 Declaring Solver Outputs . . . 101
8.12 Generating the solver . . . 102
8.13 Calling the generated low-level solver . . . 102
8.14 Debugging a formulation . . . 103
8.15 The QP_FAST algorithm . . . 103
8.16 Condensing (automatic state elimination) . . . 104

9 High-level Interface 107
9.1 Supported problems . . . 108
9.2 Expressing the optimization problem in code . . . 110
9.3 Generating a solver . . . 120
9.4 Calling the solver . . . 121
9.5 External function evaluations in C . . . 124
9.6 Calling the nonlinear functions from MATLAB or Python . . . 129
9.7 Mixed-integer nonlinear solver . . . 131
9.8 Sequential quadratic programming algorithm . . . 135
9.9 Differences between the MATLAB and the Python client . . . 138
9.10 Examples . . . 139

10 Simulating your custom controller in Simulink® 141
10.1 The S-Function interface . . . 141
10.2 The Coder interface . . . 141
10.3 S-Function vs Coder interface . . . 142
10.4 FORCESPRO Simulink® blocks . . . 142

11 Examples 149
11.1 How to . . . 149
11.2 Y2F interface: Basic example . . . 165
11.3 Y2F interface: Trajectory Optimization for Quadrotor Flight . . . 172
11.4 Low-level interface: Active Suspension Control . . . 176
11.5 Low-level interface: Robust estimation (Kalman filter) . . . 179
11.6 Low-level interface: Spacecraft Rendezvous (MATLAB & Python) . . . 187
11.7 Low-level interface: DC/DC converter . . . 191
11.8 High-level interface: Basic example . . . 198
11.9 High-level interface: Obstacle avoidance (MATLAB & Python) . . . 202
11.10 High-level interface: Indoor localization (MATLAB & Python) . . . 212
11.11 High-level interface: Path tracking example (MATLAB) . . . 216

11.12 High-level interface: Legacy path tracking example (MATLAB & Python) 227
11.13 High-level interface: Rate Constraints 240
11.14 High-level interface: Soft Constraints 245
11.15 Controlling a crane using a FORCESPRO NLP solver 249
11.16 Real-time SQP Solver: Robotic Arm Manipulator (MATLAB & Python) 253
11.17 Controlling a DC motor using a FORCESPRO SQP solver 260
11.18 Mixed-integer nonlinear solver: F8 Crusader aircraft 265
11.19 High-level interface: Optimal EV charging and speed profile example (MATLAB & PYTHON) 273
11.20 High-level interface: Extended optimal EV charging and speed profile example using a 2D motor efficiency map (MATLAB & PYTHON) 299

12 Parametric problems **331**
12.1 Defining parameters 331
12.2 Example 332
12.3 Parametric Quadratic Constraints 333
12.4 Diagonal Hessians 333
12.5 Sparse Parameters 334
12.6 Special Parameters 334
12.7 Python: Column vs Row Major Storage Format 334

13 Calling a solver from C or C++ **337**
13.1 Generating a solver library 337
13.2 C interface 338
13.3 Compiling solver executable 340
13.4 Memory allocations 341

14 Code Deployment **345**
14.1 Main Targets 345
14.2 dSPACE deployment through Simulink Coder 348
14.3 dSPACE deployment through ConfigurationDesk 366
14.4 Speedgoat 386
14.5 Speedgoat QNX 404

15 Multicore parallelization **419**
15.1 Internal parallelism 419
15.2 External parallelism 420
15.3 Combining external and internal parallelism 421

16 Licensing **423**
16.1 Machine Identification 423
16.2 Static License 424
16.3 License Files 424
16.4 Floating Licenses 425

17 Autotuner **429**
17.1 Autotuner Options 430
17.2 Collecting Tuning Data 430
17.3 Validation 431

18 Solver Options **433**
18.1 Default options 434
18.2 General options 435
18.3 Low-level interface options 454
18.4 High-level interface options 462

19 Exitflags **475**
19.1 Exitflags and quality of the result 475
19.2 Mixed integer Nonlinear Programming exitflags 478

20 Modelling Utilities **479**
20.1 Interpolations 1D (e.g. splines) . . . 479
20.2 Interpolations 2D (B-splines) . . . 481
20.3 Smooth Approximations . . . 490

21 Dumping Problem Formulation and Data **493**
21.1 Why to use the dump tool? . . . 493
21.2 How to use the dump tool? . . . 494

22 Webcompilers **507**
22.1 Webcompilers in FORCESPRO . . . 507
22.2 Webcompilers versioning . . . 507
22.3 Ensuring use of webcompiler version . . . 508

23 Frequently asked questions **509**
23.1 Features of FORCESPRO . . . 509
23.2 Setting up a FORCESPRO problem formulation . . . 510
23.3 Issues during code generation . . . 511
23.4 Issues when running a generated solver . . . 513
23.5 Issues when using the Simulink interface . . . 515
23.6 Issues during code deployment . . . 516
23.7 Other topics . . . 516

Bibliography **519**

Chapter 1

Introduction

- *Troubleshooting and support*
- *Licensing*
- *Citing FORCESPRO*
- *Product Life Cycle*
- *Release Notes*
- *Version history of manual*

This is a user manual for FORCESPRO, a commercial tool for generating highly customized optimization solvers that can be deployed on all embedded computers. FORCESPRO is intended to be used in situations were the same optimization problem has to be solved many times, possibly in real-time, with varying data, i.e. there is sufficient time in the design stage for generating a customized solution for the problem you want to solve.

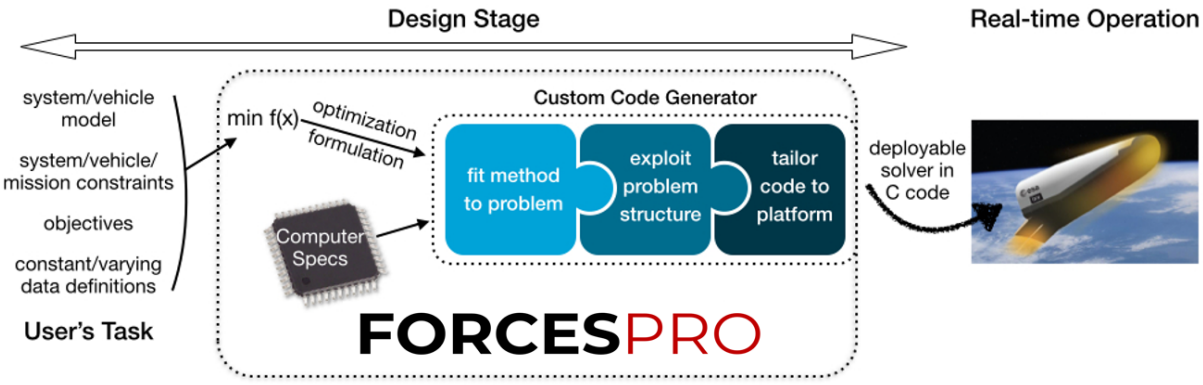


Figure 1.1: Overview of FORCESPRO.

The code generation engine in FORCESPRO extracts the structure in your optimization problem and automatically synthesizes a custom optimization solver. The resulting C code can only solve one optimization problem (with certain data changing), hence it is typically many times more efficient and smaller code size than general-purpose optimization solvers. The generated C code is also library-free and uses no dynamic memory allocation making it suitable for safe deployment on real-time autonomous systems.

This document will show you how to input your optimization problem description for code generation in FORCESPRO. It is important to point out that FORCESPRO is not a tool for transforming a problem specification into an optimization problem description. This responsibility lies with the user.

1.1 Troubleshooting and support

FORCESPRO typically returns meaningful error messages when code generation errors occur due to invalid user inputs. When encountering other errors please consult our documentation which is included in the FORCESPRO client and is also available on all FORCESPRO servers. In case you cannot find a solution to your problem please submit a bug report to support@embotech.com.

Much effort has gone into making this interface easy to use. We welcome all your suggestions for further improving the usability of the tool. Requests for special functionality for your particular problem will also be considered by our development team. For all requests and feedback please contact support@embotech.com.

1.2 Licensing

1.2.1 Commercial licensing

FORCESPRO licenses are available through a subscription model. There are four types of licenses, as seen below:

- **Engineer License**: For generating FORCESPRO solvers. Charged per engineer computer.
- **Software Testing License (SiL/CI)**: For running FORCESPRO solvers on a desktop PC or a server for simulation and (automated) testing. No physical system is controlled. Charged per platform running the solver.
- **Floating License**: For running FORCESPRO solvers on servers or virtualized environments (such as Docker containers) without permanently mapping the license to a hardware system. Charged per number of platforms able to concurrently run the solver. Currently available only on Linux x86/x86_64.
- **Hardware Testing License (HiL/Field Testing)**: For controlling a physical system (i.e. the target platform may also be an ECU or a rapid prototyping platform). Charged per platform running the solver.

For more information regarding licensing please check on our website or contact sales@embotech.com.

FORCESPRO licenses are available in variants S, M and L. For more information please check the section *License Variants*

1.2.2 Academic licensing

Users at degree granting institutions can have access to the **Engineer License** version of FORCESPRO free of charge provided they are not doing research for an industrial partner. **Software Testing** and **Hardware Testing** licenses are also available at highly reduced rates.

1.3 Citing FORCESPRO

If you use FORCESPRO in published scientific work, please cite the following two papers:

```
@misc{FORCESPro,
      Author       = "Embotech AG",
      Howpublished = "{{\tt https://forces.embotech.com/}}",
```

(continues on next page)

(continued from previous page)

```
      Title        = "FORCESPRO",
      Year         = "2014--2023"
}
```

```
@article{FORCESNLP,
  Author       = "A. Zanelli and A. Domahidi and J. Jerez and M. Morari",
  Title        = "FORCES NLP: an efficient implementation of interior-point...
                  methods for multistage nonlinear nonconvex programs",
  Journal      = "International Journal of Control",
  Year         = "2017",
  Pages        = "1-17"
  }
```

1.4 Product Life Cycle

A new major or minor version of FORCESPRO is released every quarter, with patch releases in between. These new versions contain new functionalities and improvements in terms of speed and robustness.

In order to be able to add novel features and improve existing ones at a high pace, FORCESPRO uses continuous deployment as development policy. That also implies that we have to ask users to update their clients if they want to benefit from the latest version of FORCESPRO. In rare cases, this may also mean breaking backward compatibility (see also section *Backward Compatibility*) requiring users to either make the necessary changes in their own code or to stick with an older version (and the corresponding server for code generation).

We guarantee that the codegen servers of any new version of FORCESPRO will be kept available for at least one year, starting from their respective release dates. Table 1.1 lists all release dates since version 1.7.0 along with the actual or planned date for the corresponding code generation server to go offline. In case you need an older version of FORCESPRO to be available beyond the scheduled offline date, please contact support@embotech.com so we can work out a solution for you.

Table 1.1: Release Dates and Codegen Server Availabilities

FORCESPRO Version	Release Date	Actual or *Planned* Date Server goes Offline
6.4.0	2025-07-16	*2026-12-31*
6.3.0	2023-09-20	*2026-06-30*
6.2.0	2023-06-15	*2025-12-31*
6.1.0	2023-04-04	*2025-12-31*
6.0.1	2022-12-14	*2025-12-31*
6.0.0	2022-07-13	*2025-21-31*
5.1.0	2021-12-09	*2025-09-30*
5.0.1	2021-10-21	*2025-09-30*
5.0.0	2021-09-09	*2025-09-30*
4.4.0	2021-06-16	2023-06-15
4.3.1	2021-06-01	2022-09-08
4.3.0	2021-05-18	2022-09-08
4.2.1	2021-03-23	2022-09-08
4.2.0	2021-02-11	2022-09-08
4.1.1	2020-12-09	2022-09-08
4.1.0	2020-11-04	2022-09-08
4.0.0	2020-09-22	2021-10-15
3.1.0	2020-07-15	2021-07-15
3.0.1	2020-05-26	2021-05-31
3.0.0	2020-04-09	2021-03-15
2.0.0	2019-12-17	2020-12-15
1.9.1	2019-10-18	2020-10-01
1.9.0	2019-09-05	2020-05-15
1.8.0	2019-06-13	2020-04-01
1.7.0	2019-03-08	2019-10-01

1.5 Release Notes

1.5.1 New features in FORCESPRO 6.4.0

- Added AD tool support for CasADi 3.6.3 and 3.6.7
- Added support for Python versions 3.10, 3.11, 3.12 and 3.13
- Made plugin for the MathWorks Model Predictive Control Toolbox (TM) work with R2024a, R2024b and R2025a

1.5.2 Improvements in FORCESPRO 6.4.0

- Enabled parametric linesearch also for `SQP_NLP` general
- Introduced codeoptions `nlp.parametric_barrier_loqo/monotone` to allow making all barrier parameters related to those two strategies real-time parameters
- Introduced codeoptions `sqp_nlp.parametric_reg_hessian` to set Hessian regularization being passed as real-time parameter
- Added missing `blckMatrices` option to Python client
- Added support for sparse stacked parameters
- Consistent defaults for missing run-time parameters in low-level interface
- Normalize line endings of generated C code (unix format)

- Added client support to return a solver object to call a solver generated with the low-level interface
- Added `forcespro_version.py` script to Python client
- Deprecated support for Python versions 2.7 and 3.4.
- Deprecated WSDL from server and clients

1.5.3 Bug fixes in FORCESPRO 6.4.0

- Bugfix in monotonicity check for the 2D-spline knots
- Fixes related to incorrect indexing when using compact code
- Index fixes for varying number of variables per stage
- Fix in linear algebra function used for `settings.init=2` and sparse `ineq.p.A` matrix
- Fixed AVX issue with stacked parameters
- Prevent generation of unused C variables in happening in specific situations
- Various bugfixes in symbolic dump tool in Python client
- Bugfix for loading symbolic functions of last stage in Python client
- Fixed number of arguments for LSobjective in Python client
- Fixed local compilation of low-level in Python client
- Fixed compatibility issue with setting network certificates in MATLAB R2016a

1.5.4 New features in FORCESPRO 6.3.0

- Introduced condensing (i.e. automatic elimination of states) for convex solvers to improve performance for problems with short control horizon

1.5.5 Improvements in FORCESPRO 6.3.0

- Added solver functions for (de)serializing C params with Python client
- Allowing SQP solver to be generated if linear inequalities only depend on real-time params
- Added client example illustrating how to setup a standalone solver in C++
- Allowed Matlab code generation for ForcesMin/Max
- Made plugin for the MathWorks Model Predictive Control Toolbox (TM) work with R2023b

1.5.6 Bug fixes in FORCESPRO 6.3.0

- Robustified Matlab client check for free variables
- Fixed compatibility issue for MATLAB R2016b
- Fixed missing initialization of `info.res_eq`
- Fixes in info struct field types for coder interface
- Fixed issue when projecting to double-sided polytopic constraints in first-order methods

- Fixed selection of certificates for server connections and printed messages

1.5.7 New features in FORCESPRO 6.2.0

- Added support for adaptive and LTV MPC to the plugin for the MathWorks Model Predictive Control Toolbox (TM)

1.5.8 Improvements in FORCESPRO 6.2.0

- Extended support to provide stage-wise BFGS initialization
- Made available webcompiler hash to allow enforcing a specific configuration
- Use default C interface for Y2F and added coder interface support

1.5.9 Bug fixes in FORCESPRO 6.2.0

- Made all function and file names unique for SQP solver, chainrule integrators and memory interface to avoid issues when combining multiple solvers
- Fixed a couple of Python/MATLAB incompatibilities in symbolic dumps
- Fixed `SQP_NLP` compilation failure when only either lower or upper bounds are specified
- Fixed state initialization issue in Simulink block for SQP_NLP
- Deprecated `ADMMfactorize` option and enabled factorization by default
- Fixed compatibility issue for MATLAB R2018b

1.5.10 New features in FORCESPRO 6.1.0

- Added support for CasADi MX expressions to MATLAB and Python client (with few limitations)
- Added functionality to use 2D splines inside symbolic problem formulations to both MATLAB and Python client

1.5.11 Improvements in FORCESPRO 6.1.0

- Change default dump tool from legacy dumps to symbolic dumps in MATLAB client
- Made plugin for the MathWorks Model Predictive Control Toolbox (TM) work with R2023a
- Added platform support for AARCH-Cortex-A53
- Improvements to the Simulink Coder interface, e.g. added support for solvers generated via the Python client
- Introduced dedicated exitflags to handle bad (negative) return values of external callbacks
- Added utility script `stages2qcqp` to Python client to convert FORCESPRO low-level formulations into standard (QC)QPs

1.5.12 Bug fixes in FORCESPRO 6.1.0

- Fixed a couple of compatibility issues in Python client
- Ensured consistency of parametric and hardcoded tolerances for `PDIP_NLP`
- Fix for Hessian regularization in `SQP_NLP`
- Fixed splitting of ComparisonOutputs and ComparisonObjectives for QP Fast and SQP Fast tuning
- Fixed issues with parameter indices for MINLP and binary QPs
- Removed FMA operations when AVX has been disabled
- Fixes for initial state with Python client formulation

1.5.13 Improvements in FORCESPRO 6.0.1

- Added Coder interface for solvers (MATLAB and Simulink)
- Made plugin for The MathWorks Model Predictive Control Toolbox (TM) work with R2022b
- Floating point functions now follow precision accuracy
- Added strict size check for E matrix in Python client
- Changed Python default codeoptions to be consistent with MATLAB ones

1.5.14 Bug fixes in FORCESPRO 6.0.1

- Fixed Python convexity check for case of continuous dynamics
- Fixed objective function computation when tuning `QP_FAST` solver
- Fixed bugs related to saturateFloats, optimization options and certain linear algebra operations
- Fixed code generator issue when using the `compact_code` feature
- Fixed multi-threaded code for `parallel>0` and `compact_code`

1.5.15 New features in FORCESPRO 6.0.0

- Introduced solvemethod `QP_FAST` as well as SQP Fast algorithm to MATLAB client and plugin for The MathWorks Model Predictive Control Toolbox (TM); including automatic tuning and validation functionality
- Made SQP algorithm run with floattype `'float'`
- Adapted C interface implementing thread-safe memory layout for all solvers (except those including binary or integer variables)
- Added C library functions returning required solver memory
- Enabled possibility to combine both internal and external parallelism
- Added support for code generation for the Infineon AURIX(TM) platform
- Removed legacy Simulink GUI from MATLAB client

1.5.16 Improvements in FORCESPRO 6.0.0

- Added real-time parameter `parametric_iterations` to adjust maximum number of iterations during runtime
- Added real-time parameter `solver_exit` to early-terminate solver
- Passing inner QP exitflag and iterations to info struct of `SQP_NLP`
- Added new client function to get default codeoptions
- Allow for diagonal Hessians when stacking parameters
- Made plugin for The MathWorks Model Predictive Control Toolbox (TM) work with R2022a
- Renamed `casadi2forces` interface to `adtool2forces` and changed type to int to return exitcode
- Added functionality to configure server certificate authentication for web requests of FORCESPRO client
- Added solver ID as part of the FORCESPRO solver interfaces
- Added new client example illustrating time-optimal EV charging and operation in both MATLAB and Python client

1.5.17 Bug fixes in FORCESPRO 6.0.0

- Fixed initialization when using `nlp.BarrStrat = "monotone"`
- Fixed bugs related to stacked parameters
- Fixed two bugs in low-level linear algebra functions
- Addressed MISRA C issues in SQP solver
- Added solvername prefix to `sparse2fullcopy` function

1.5.18 New features in FORCESPRO 5.1.0

- Added Floating License Proxy packages for floating license connections

1.5.19 Improvements in FORCESPRO 5.1.0

- Added more advanced path tracking example to MATLAB client
- Made plugin for The MathWorks Model Predictive Control Toolbox (TM) work with R2021b
- Made MATLAB client compatible with R2014b

1.5.20 Bug fixes in FORCESPRO 5.1.0

- Robustified array conversions for Interpolation in Python client
- Bugfix in SQP solver to use correct tolerances in underlying QP solver
- Fixed issue in SQP log when showing solvetime
- Small bugfix in loading of symbolic dumps in MATLAB

1.5.21 New features in FORCESPRO 5.0.1

- Added MEX interfaces and Python interfaces for calling nonlinear callback functions stand-alone on user side

1.5.22 Improvements in FORCESPRO 5.0.1

- Re-enabled generation of ADMM solvers via Python client
- Extended code generation floating license options; improved floating licensing's performance and robustness
- Added files for requirements in Python client and made suds package optional

1.5.23 Bug fixes in FORCESPRO 5.0.1

- Bugfix in callback evaluations function of SQP solver
- Bugfix for detection of affine inequalities in SQP solver in Python client
- Fixed MISRA C issues when using SQP solver or using float callbacks with chainrule integrators
- Fixed issue with loading old dumps in Python

1.5.24 New features in FORCESPRO 5.0.0

- Added improved symmetric indefinite linear solver and iterative refinement
- Added CasADi 3.5.5 support also to Python client and made it default AD tool in both clients
- Added support for Python 3.9

1.5.25 Improvements in FORCESPRO 5.0.0

- Added Y2F example for optimizing trajectory of quadrotor flight
- Added support for server connection via RestAPI
- Added option to export lower triangular BFGS

1.5.26 Bug fixes in FORCESPRO 5.0.0

- Minor bugfixes in low-level and high-level interface solver interface
- Fix to handle scalar input arguments to interpolations in Python client
- Added fixes for size one parameters
- Improved adherence to C90 standard and MISRA C rules
- Bugfix concerning nonlinear inequalities detection in Python client

1.5.27 New features in FORCESPRO 4.4.0

- Implemented linear subsystem exploitation for explicit chainrule integrator RK4
- Implemented chainrule variant for integrator IRK2
- Added support for dSPACE SCALEXIO and dSPACE MicroLabBox
- Added functionality to use interpolations (such as splines) inside symbolic problem formulations to both MATLAB and Python client

1.5.28 Improvements in FORCESPRO 4.4.0

- Added Python variant of ForcesMin/ForcesMax, wrapped into new `modelling` subpackage

1.5.29 Bug fixes in FORCESPRO 4.4.0

- Fixed minor issues with return flags of SQP solver, e.g. in case of license error

1.5.30 Improvements in FORCESPRO 4.3.1

- Added support for server connection via proxy in Python client

1.5.31 Bug fixes in FORCESPRO 4.3.1

- Fixed bug for code option `threadSafeExpert` causing two static variables
- Added missing code option `nlp.max_num_threads` to Python client

1.5.32 New features in FORCESPRO 4.3.0

- Added support to formulate and solve multistage nonlinear MPC problems with The MathWorks Model Predictive Control Toolbox (TM)
- Added code option `threadSafeExpert` to give users full control over memory allocation when running multiple instances of an `PDIP_NLP` or `PDIP` solver
- Added support for CasADi 3.5.5 in the MATLAB client

1.5.33 Improvements in FORCESPRO 4.3.0

- Made CasADi 3.5.1 default AD tool also in the MATLAB client
- Added support for single (stacked) solution vector for solvers `PDIP_NLP` and `SQP_NLP`
- Added more thorough check for identical stages in MATLAB client along with new code option `nlp.strictCheckDistinctStages`
- Let `FORCESversion` also return planned offline date of client version
- Added new client examples demonstrating how to formulate problems comprising soft and rate constraints using the high-level interface

1.5.34 Bug fixes in FORCESPRO 4.3.0

- Fixed freeing of DLLs in Python Client
- Fixed issue with code option `noVariableElimination` is used along with linear solver `normal_eqs` in MATLAB client

1.5.35 Improvements in FORCESPRO 4.2.1

- Added msgpack support for MacOS
- Added separate optlevel options for host and target
- Improved robustness of client connection to the codegen server
- Added support for custom parameters in Python client

1.5.36 Bug fixes in FORCESPRO 4.2.1

- Bugfix in QP solver caused by code optimization for source in src_target folder
- Fixed bug in ADMM method

1.5.37 New features in FORCESPRO 4.2.0

- Added support for dumping of problem formulation from C
- Added support for NI cRIO platforms
- Created Simulink fingerprinters for platforms with Simulink Model deployment
- Added Speedgoat (for MATLAB R2020b and later) example for The MathWorks Model Predictive Control Toolbox (TM)
- Added support for single precision callbacks, i.e. mixed-precision NLP solution

1.5.38 Improvements in FORCESPRO 4.2.0

- Changed to new server communication in MATLAB client for improved safety and connection stability
- Reenabled chainrule integrators as default integrator when using continuous dynamics, and fixed performance issues
- Extended Speedgoat support to more MATLAB/Simulink Real-Time releases

1.5.39 Improvements in FORCESPRO 4.1.1

- Improved hashing of sparse linear algebra routines
- Fixed MATLAB network communication, replaced deprecated SOAP methods with new ones, enabled with `legacyNetworkConnections = 0`
- Parallel BFGS updates with `compact_code = 0` (default)
- Prevent MATLAB client to overwrite user script if solver has same filename
- Throw proper exceptions when block structure is not detected in sparse parametric equalities in low-level interface

- Parallel callbacks evaluation with `compact_code = 1` and `parallel >= 1`
- Reverted to legacy integrators in default behavior
- Added code generation compatibility with MATLAB 2016a
- Various fixes and updates in the API of the Python dump tool

1.5.40 Bug fixes in FORCESPRO 4.1.1

- Fixed FORCESconfigureClient to work on all OS
- Fixed codegen failure with compact_code related to nonlinear inequalities
- Disabled compact_code when initial equality constraint not eliminated (D0)
- Restored functionality to collect variable declarations at beginning of CasADi callbacks (only if `c90` code option is set)
- Bug fix for copy of scaler parameters with `compact_code = 1`

1.5.41 New features in FORCESPRO 4.1.0

- Code-generated explicit integrators and sensitivity with chain rule and variational differential equation
- Python dump tool and compatible MATLAB dump tool
- Option for adding a single external callback instead of adding all callbacks externally
- Added solver and webcompiler support for speedgoat (for MATLAB R2020b and later)

1.5.42 Improvements in FORCESPRO 4.1.0

- Scalar parameters are not treated as arrays anymore for compatibility with MATLAB Coder. To enable the previous behavior set code option `size_one_param_as_array = 1`
- Separated CasADi and Symbolic math toolbox callbacks to have more control over dynamics callbacks.
- Introduce code option `separateCasadiFiles` which when set to 1 ensures old callback file structure (separate model files).
- Replaced old obstacle avoidance client examples for python and MATLAB by new interactive ones
- Added path tracking example for the python and the MATLAB client

1.5.43 Bug fixes in FORCESPRO 4.1.0

- Fixed Simulink and standalone Python interface using scalar parameters
- Fixed some openmp issues and added number of threads as runtime parameter

1.5.44 New features in FORCESPRO 4.0.0

- Support for FORCESPRO NLP solvers (`PDIP_NLP` and `SQP_NLP`) in The MathWorks Model Predictive Control Toolbox (TM)
- Solver timeout option for `PDIP_NLP`, `SQP_NLP` and `PDIP`
- New code option exportBFGS which enables export of BFGS diagonal on every stage

1.5.45 Improvements in FORCESPRO 4.0.0

- Server now returns `interface/definitions.py` file independent of whether the request was sent from the MATLAB or Python client
- Added support for symbolic step size in Python integrators
- Added connection tester for the FORCESPRO server
- Added new parameter type Adense to allow copy of dense A matrix to sparse internally. Should be used within Model Predictive Control Toolbox plugin only!
- New option `nlp.parametricBFGSinit` for initializing BFGS matrix as a run-time parameter

1.5.46 Bug fixes in FORCESPRO 4.0.0

- Fixed export of root relaxation solution in MINLP solver
- Fixed number of outputs in ADMM method
- Added fix for floattype `'int'` and `'short'`
- Fixed issue occurring in Python client when all initial or all final variables are fixed
- Fixed reading issue in csmatio library

1.5.47 New features in FORCESPRO 3.1.0

- High-level Python interface for NLP solvers

1.5.48 Improvements in FORCESPRO 3.1.0

- Vectorized outer product on one-stage dense QP problems in double precision on Intel platforms
- Refactoring of clients and server to enable standalone release
- Check for vectorization instructions in Python client, refactored C code in DLL
- Made variables in generated interface static
- Improved efficiency of CasADi file postprocessing in MATLAB client
- Export of dual variables in solver `PDIP_NLP`
- Fixed updateClient scripts to delete old data
- Made FORCES_NLP return dumped formulation even if an error occurs during execution
- Allow to specify directory when saving dumped problem formulation/instance

1.5.49 Bug fixes in FORCESPRO 3.1.0

- Fix in detection of selection matrix
- Fix in CasADi for linux systems
- Fixed bug with stacked parametric bounds
- Updated accessing of Stage properties to work with obfuscation
- fix issue with variable number of equality constraints in convex problems
- Fixed issue in CasADi code generation
- Fixed internal rounding heuristic in MINLP solver

1.5.50 Improvements in FORCESPRO 3.0.1

- New `nlp.stack_parambounds` for stacking parametric bounds over stages with solvers `PDIP_NLP` and `SQP_NLP`
- Support for MicroAutoBox III

1.5.51 Bug fixes in FORCESPRO 3.0.1

- Bug fix in fraction to boundary rule
- Bug fixes for specific compilation settings
- Fixed download of CasADi for macos
- Fixed bug in model files declarations in `casadi2forces` with solver `SQP_NLP`

1.5.52 New features in FORCESPRO 3.0.0

- Real-time sequential quadratic programming solver via code option `SQP_NLP`
- Support for MathWorks Symbolic Math Toolbox and CasADi 3.5.1 (with limitations)
- Code option `nlp.compact_code` for generating small-size code on long horizon problems
- Support for license files
- Option for dumping problem formulation and data for support

1.5.53 Improvements in FORCESPRO 3.0.0

- Revamped licensing system
- Removed object files from downloaded solver package

1.5.54 Bug fixes in FORCESPRO 3.0.0

- Fixed bug with number of stages and integer guess in MINLP solver

1.5.55 New features in FORCESPRO 2.0.0

- Introduced support for FORCESPRO QP solvers in the The MathWorks Model Predictive Control Toolbox (TM)
- Created new examples for the MPC Toolbox plugin

1.5.56 Improvements in FORCESPRO 2.0.0

- Made tolerances on equalities, inequalities, stationarity and complementarity run-time parameters in NLP solver
- Automatic disabling of vectorization when some matrix parameters are sparse

1.5.57 Bug fixes in FORCESPRO 2.0.0

- Fixed linking issue with avx on linux host
- Fixed mex interface to not copy empty parameters
- Fixed bug with MINLP solver exitflag on infeasible problems

1.5.58 New features in FORCESPRO 1.9.1

- Adapted FORCESPRO license check to portal database
- Adapted floating license database checks to portal database
- Made linear algebra vectorization stage dependent

1.5.59 Improvements in FORCESPRO 1.9.1

- Fixed numerical bug in NLP line-search

1.5.60 New features in FORCESPRO 1.9.0

- New code-generation options for AVX and NEON vectorization
- New code generation options and parameters to provide an integer guess to the MINLP solver
- New runtime parameter parallelStrategy for MINLP solver
- Created dedicated Floating License web Server

1.5.61 Improvements in FORCESPRO 1.9.0

- Changed floating license communication to http
- Enabled user-defined outputs in MINLP solver
- Added code option `c90` to add extra C definitions in CasADi model files
- Added openmp flag to nvidia webcompiler
- Added support for Python 3.6
- Updated usysid files in client

1.5.62 Bug fixes in FORCESPRO 1.9.0

- Fixed bug with constraints handling in code-generation
- Fixed memory bug in MINLP solver
- Fixed bug in parameters indexing in client. Parameters are now indexed with a fixed number of digits depending on the horizon length. 1 digit below 10, 2 digits between 10 and 100 excluded,...
- Fixed bug with stacked parameter `ineq.p.b`

1.5.63 New features in FORCESPRO 1.8.0

- Mixed-integer nonlinear solver with parallelizable search and other customization features
- Support for the Speedgoat platform
- Support for the Integrity ARM platform
- Support for Docker containers
- Updated `newParam` API to allow for parameters stacked over stages

1.5.64 Improvements in FORCESPRO 1.8.0

- Improved performance of `compactSparse` feature
- Added custom headers to specify platforms

1.5.65 Bug fixes in FORCESPRO 1.8.0

- Fixed numerical bug in v1.7.0

1.5.66 New features in FORCESPRO 1.7.0

- MISRA 2012 compliance, no mandatory or required violations in generated C code
- Added support for dSPACE MicroAutoBox II
- Added support for ARM Cortex A72 platforms
- Added support for MinGW as a mex compiler
- New code option `compactSparse` for smaller code and faster compilation of sparse problems
- Added `threadSafeStorage` option, enabling creation of thread-safe solvers (requires C11 compilers)

1.5.67 Improvements in FORCESPRO 1.7.0

- Improved codegen speed for sparse problems
- Improved web compilation to reduce http timeouts
- Secure client-server communication under custom embotech domain
- Improved portability of functions used

- Added display of license and solver expiration as well as generation id on header files
- Updated `FORCEScleanup` to include all solver related files
- Improved messages and warnings returned from FORCESPRO client
- Now passing iteration number to function evaluations
- Added new error code for invalid parameter initial values

1.5.68 Bug fixes in FORCESPRO 1.7.0

- Changed default server when default server file is missing
- Always check for default server files when choosing server to use
- Corrected the logic for updating the best solution found so far (NLP)
- Fixed sparse linear algebra routine names

1.6 Version history of manual

The version history of this document is presented in *Version history of FORCESPRO manual*.

Table 1.2: Version history of FORCESPRO manual

Version	Revision	Date	Reason for change
1	0	2017-04-10	Initial version
2	0	2018-09-27	Overhaul of outdated manual
2	1	2018-11-19	Add dSPACE code deployment
3	0	2019-02-20	Updated manual for v1.7.0
4	0	2019-06-04	Updated manual for v1.8.0
4	1	2019-08-29	Updated manual for v1.9.0
5	0	2019-10-10	Updated manual for v1.9.1
6	0	2019-12-09	Updated manual for v2.0.0
7	0	2020-04-07	Updated manual for v3.0.0
7	1	2020-05-26	Updated manual for v3.0.1
7	2	2020-07-13	Updated manual for v3.1.0
8	0	2020-09-21	Updated manual for v4.0.0
8	1	2020-10-30	Updated manual for v4.1.0
8	2	2020-12-07	Updated manual for v4.1.1
8	3	2021-02-09	Updated manual for v4.2.0
8	4	2021-03-18	Updated manual for v4.2.1
8	5	2021-05-11	Updated manual for v4.3.0
8	6	2021-05-31	Updated manual for v4.3.1
8	7	2021-06-15	Updated manual for v4.4.0
9	0	2021-09-08	Updated manual for v5.0.0
9	1	2021-10-20	Updated manual for v5.0.1
9	2	2021-12-08	Updated manual for v5.1.0
10	0	2022-07-08	Updated manual for v6.0.0
10	1	2022-12-13	Updated manual for v6.0.1
10	2	2023-04-04	Updated manual for v6.1.0
10	3	2023-06-14	Updated manual for v6.2.0
10	4	2023-09-18	Updated manual for v6.3.0
10	5	2025-07-15	Updated manual for v6.4.0

Chapter 2

License Variants

- *Variant Summary*
- *Variant S*
- *Variant M*
- *Variant L*

Each problem type requires a dedicated solver method in order to be solved quickly and efficiently. FORCESPRO is available in different variants in order to adapt to each user's needs. When receiving a FORCESPRO license on the portal(https://my.embotech.com) a user can select one of the available variants which is best suited for the problem to be solved. At any point, a user can decide to upgrade to a larger variant in order to include additional solver methods in their available tool set for FORCESPRO.

The available variants are (smaller variants are included in larger ones):

- **S** (*Variant S*)
- **M** (*Variant M*)
- **L** (*Variant L*)

2.1 Variant Summary

In the tables below you can find a summary of the components provided with each variant of FORCESPRO.

Table 2.1: Problem types supported for each variant

	S	M	L
Problem Type			
LP	✓	✓	✓
QP	✓	✓	✓
QCQP	✓	✓	✓
BI-QP		✓	✓
NLP (SQP)		✓	✓
NLP (IP)			✓
MINLP			✓

Table 2.2: Interfaces provided for each variant

	S	M	L
Interface			
MATLAB Low-Level	✓*	✓	✓
Python Low-Level	✓*	✓	✓
MATLAB Y2F	✓	✓	✓
MathWorks MPC Toolbox™ (Linear MPC)	✓	✓	✓
MATLAB High-Level		✓**	✓
Python High-Level		✓**	✓
MathWorks MPC Toolbox™ (Nonlinear MPC)		✓**	✓
* No Binary Constraints			
** Only with SQP method			

2.2 Variant S

This variant is used for generation of convex solvers. This variant should be used for solving:

- LP problems
- QP problems
- QCQP problems

This variant is delivered with the following interfaces:

- MATLAB Low-level Interface (*Low-level interface*)
- Python Low-level Interface (*Low-level interface*)
- MATLAB Y2F Interface (*Y2F Interface*)
- MathWorks Model Predictive Control Toolbox™ - Linear MPC (*MathWorks Linear MPC Plugin*)

2.3 Variant M

This variant further enables the generation of SQP solvers for NLPs and the solution of Binary-Integer QPs. This variant should be used for solving:

- Binary-Integer QP problems (*Binary constraints*)
- NLP Problems using SQP methods (*Sequential quadratic programming algorithm*)

This variant is delivered with the following interfaces:

- MATLAB High-level Interface (*High-level Interface*) with *codeoptions.solvemethod = 'SQP_NLP';*
- Python High-level Interface (*High-level Interface*) with *codeoptions.solvemethod = 'SQP_NLP'*
- MathWorks Model Predictive Control Toolbox™ - Nonlinear MPC (*MathWorks Nonlinear MPC Plugin*) with *options.SolverType = 'SQP';*

2.4 Variant L

This variant provides the full experience of FORCESPRO and enables all its features. This variant further enables the solution of:

- NLP problems with Interior-Point Methods and SQP
- MINLP problems (*Mixed-integer nonlinear solver*)

This variant is delivered with the following interfaces:

- MATLAB High-level Interface (*High-level Interface*) with full support
- Python High-level Interface (*High-level Interface*) with full support
- MathWorks Model Predictive Control Toolbox™ - Nonlinear MPC (*MathWorks Nonlinear MPC Plugin*) with full support

Chapter 3

Installation

3.1 Obtaining FORCESPRO

FORCESPRO is a client-server code generation system. The user describes the optimization problem using the client software, which communicates with the server for code generation (and compilation if applicable). The client software is the same for all users, independent of their license type.

In order to obtain FORCESPRO, follow the steps below:

1. Inquire a license from https://www.embotech.com/products/forcespro/licensing/ or by directly contacting licenses@embotech.com.
2. After receiving a license, if registered on the portal, the FORCESPRO client can be downloaded from the portal after assigning an Engineering Node. For more information see https://my.embotech.com/readme. Otherwise the FORCESPRO client will be sent to you via email.
3. Unzip the downloaded client into a convenient folder.

Note: The FORCESPRO client contains several inner ZIP-files for the Python client named *forcesproXY.zip*. These do not need to be extracted!

3.2 Installation of the MATLAB Client

- *System requirements*
- *Keeping FORCESPRO up to date*
- *Installing and running older versions of FORCESPRO*

Add the path of the downloaded folder `FORCESPRO` to the MATLAB path by using the command `addpath DIRNAME`, e.g. by typing:

```
addpath /home/user/FORCESPRO
```

on your MATLAB command prompt. Alternatively, you can add the path of the `FORCESPRO` folder via the graphical user interface of MATLAB as seen in Figure 3.1.

Having added the root folder of the FORCESPRO MATLAB client to the MATLAB path one configures the client to the specific MATLAB version by running

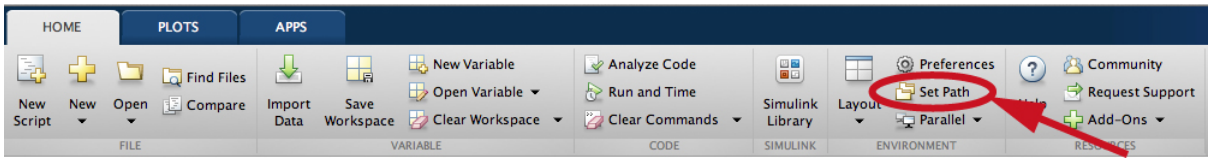


Figure 3.1: Adding the `FORCESPRO` folder to the MATLAB path.

```
FORCESconfigureClient;
```

in the MATLAB command window. After the FORCESPRO MATLAB client has been configured one can save the MATLAB path in order to always have access to FORCESPRO when initiating a new MATLAB session. Alternatively one perform the above 2 steps whenever initiating a new MATLAB session.

3.2.1 System requirements

FORCESPRO is supported on Windows, macOS and the different Linux distributions.

For the MATLAB and Simulink interfaces, 32 or 64 bit MATLAB 2012b (or higher) is required. Older versions might work but have not been tested. A MEX compatible C compiler is also required. A list of compilers that are supported by MATLAB can be found in https://www.mathworks.com/support/sysreq/previous_releases.html.

Run:

```
mex -setup
```

to configure your C compiler in MATLAB.

3.2.2 Keeping FORCESPRO up to date

FORCESPRO is actively developed and client modifications are frequent. Whenever your client version is not synchronized with the server version, you will receive a code generation error notifying you that your client is out of date.

To update your client simply type:

```
updateClient
```

on your MATLAB command prompt. `updateClient` without any arguments uses the default embotech server to grab the client, and updates all corresponding client files. The command:

```
updateClient(URL)
```

overrides the default server selection and uses the server located at URL instead.

Alternatively, the FORCESPRO client may also be updated through Python, see *Keeping FORCESPRO up to date*.

3.2.3 Installing and running older versions of FORCESPRO

Older versions of FORCESPRO can be installed and run by specifying the server URL as `https://forces-X-Y-Z.embotech.com`, replacing `X-Y-Z` with the version numbers of a selected FORCESPRO version `vX.Y.Z`. First, sync your client to the server by updating it with

```
updateClient('https://forces-X-Y-Z.embotech.com')
```

Then, in order to generate a solver, set the code generation server as

```
codeoptions.server = 'https://forces-X-Y-Z.embotech.com';
```

3.3 Installation of the Python Client

- *Quick Guide*
  - *Windows (PowerShell)*
  - *Linux Ubuntu*
  - *Mac*
- *Requirements*
  - *Python*
  - *Python Packages*
  - *Available Compiler*
- *Adding the FORCESPRO Python Client to your Python path*
  - *Option A: Setting the* PYTHONPATH *environment variable*
  - *Option B: Setting* sys.path *inside Python scripts*
- *Keeping FORCESPRO up to date*
- *Installing and running older versions of FORCESPRO*

FORCESPRO offers a Python interface that enables user to formulate a optimization problem, generating a solver for it through communication with the FORCESPRO server, and calling the generated solver directly from Python. It is contained within the FORCESPRO client package together with the MATLAB Client, which can be obtained with a valid license as described in *Obtaining FORCESPRO*.

3.3.1 Quick Guide

This section describes the most common commands needed to go from a blank system to generating and executing the first solver for different operating systems. Before doing so, you may want to double-check the section *Requirements* below, in particular with respect to supported versions of Python and external packages.

In the following, we assume you have obtained the FORCESPRO client as described in *Obtaining FORCESPRO*, and unzipped its files into the directory */path/to/forcespro* on Unix platforms or *C:\path\to\forcespro* on Windows. The following installation instructions slightly differ for the operating systems supported, so please refer to the appropriate section.

Windows (PowerShell)

```
C:\PythonXY\Scripts\pip.exe install -r C:\path\to\forcespro\requirements.txt
$env:PYTHONPATH="C:\path\to\forcespro"
```

(continues on next page)

(continued from previous page)

```
C:\PythonXY\python.exe C:\path\to\forcespro\examples\Python\HighLevelInterface\
↪RobotArmSqpRti\robot_sim.py
```

Linux Ubuntu

```
pip3 install -r /path/to/forcespro/requirements.txt
sudo apt-get install gcc libomp-dev
export PYTHONPATH="/path/to/forcespro":$PYTHONPATH
python3 /path/to/forcespro/examples/Python/HighLevelInterface/RobotArmSqpRti/robot_
↪sim.py
```

Mac

```
xcode-select --install
brew install python3 libomp
python3 -m pip install -r /path/to/forcespro/requirements.txt
export PYTHONPATH="/path/to/forcespro":$PYTHONPATH
python3 /path/to/forcespro/examples/Python/HighLevelInterface/RobotArmSqpRti/robot_
↪sim.py
```

This assumes you have the Homebrew package manager already installed. If not, run the following before any of the above instructions:

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/
↪install.sh)"
```

3.3.2 Requirements

The Python client has been tested with the following configurations:

Python

A Python installation is required. Note that only compiled Python bytecode for the versions listed below is currently shipped with the client:

- Python 3.6
- Python 3.7
- Python 3.8
- Python 3.9
- Python 3.10
- Python 3.11
- Python 3.12 (see note below)
- Python 3.13

Note: Due to a Python-internal issue *only observed in Python 3.12*, in rare situations the solver library may not be loaded correctly and an exception is thrown. Thus, use Python 3.12 client support with care or resort to any other supported Python version.

If you require a different version, please contact us at *forces@embotech.com*.

For purposes of readability, for Windows, we will assume you have installed the respective Python version into *C:\PythonXY* (where X is the major version number and Y the minor version number) throughout the rest of this documentation. On Linux and Mac, we assume you have Python 3 available in your *PATH* as *python3*.

Note: The FORCESPRO python client is not thread-safe. Generating solvers or creating solver objects concurrently with multiple threads can result in undefined behavior.

Python Packages

For any Python version, the following packages from the Python package index (PyPI) need to be installed in the *PYTHONPATH*:

- **numpy** (tested with version 1.26.1)
- **scipy** (tested with version 1.11.3)
- **requests** (tested with version 2.22; required for server connections)
- **casadi** (optional, only required for high-level interface; only versions 3.5.1, 3.5.5, 3.6.3 and 3.6.7 are supported)

All of these required packages can be conveniently installed through the command-line by running the following command from a terminal (Linux, Mac):

```
pip3 install -r /path/to/forcespro/requirements.txt
```

Or, on Windows:

```
C:\PythonXX\Scripts\pip.exe install -r C:\path\to\forcespro\requirements.txt
```

Additionally, the following optional packages from the Python package index (PyPI) may need to be installed in the *PYTHONPATH* for certain special use cases:

- **matplotlib** (tested with version 3.1; optional, only required for plotting in the example code)
- **msgpack** (tested with version 1.0.5; optional, only required for (de)serialization of C parameters)

Those optional packages (along with the required packages above) can be conveniently installed through the command-line by running the following command from a terminal (Linux, Mac):

```
pip3 install -r /path/to/forcespro/requirements-full.txt
```

Or, on Windows:

```
C:\PythonXX\Scripts\pip.exe install -r C:\path\to\forcespro\requirements-full.txt
```

Available Compiler

Nonlinear symbolic problem formulations are translated into C code by the FORCESPRO client. In order to generate solvers for these kinds of problems, a C compiler and linker must thus be present on the host machine. The following compilers have been tested and are supported by the FORCESPRO Python client:

- On Windows: Microsoft Visual Studio C Compiler 2019 and 2015 (Can be obtained by downloading the Microsoft Visual Studio Community IDE)
- On Linux: GNU Compiler Collection (GCC), tested with version 9.3.0
- On Mac: Apple clang version 11.0.3 (Can be obtained by installing the XCode command-line tools)

Additionally, on Linux, the following package must be installed if you wish to generate solvers making use of parallel execution (*options.parallel = True*) or mixed-integer nonlinear problem (MINLP) solvers:

```
sudo apt-get install libomp-dev
```

On Mac, for parallel solver generation and MINLP problems, the following package must be installed through Homebrew:

```
brew install libomp
```

3.3.3 Adding the FORCESPRO Python Client to your Python path

Once the FORCESPRO client has been downloaded and the requirements have been installed as outlined above, you will need to tell the Python interpreter where to look for the *forcespro* and *forcespro.nlp* packages which implement the FORCESPRO client interface in Python. Doing so will allow you to write *import forcespro* or *import forcespro.nlp* in your scripts to import the FORCESPRO functionality. To make the FORCESPRO client available this way, you have several options:

Option A: Setting the *PYTHONPATH* environment variable

Add the FORCESPRO client directory to your *PYTHONPATH* before calling any scripts that require FORCESPRO from the command line. In a Windows PowerShell this is done by:

```
$env:PYTHONPATH+=$(If (Test-Path env:PYTHONPATH) {";"} else {""})+"C:\path\to\
↪forcespro"
```

In Windows *cmd.exe*:

```
IF DEFINED PYTHONPATH (set PYTHONPATH=%PYTHONPATH%;C:\path\to\forcespro) ELSE (set␣
↪PYTHONPATH=C:\path\to\forcespro)
```

On Unix (Linux and Mac):

```
export PYTHONPATH="${PYTHONPATH:+${PYTHONPATH}:}/path/to/forcespro"
```

After doing so, you can call any script that requires FORCESPRO, and the script may include *import forcespro* or *import forcespro.nlp* statements without needing to know where your actual FORCESPRO client directory is.

Option B: Setting *sys.path* inside Python scripts

Add the FORCESPRO client directory to *sys.path* before importing:

```
import sys
sys.path.insert(0, '/path/to/forcespro')  # On Unix
sys.path.insert(0, 'C:\\path\\to\\forcespro')  # On Windows, note the doubly-escaped␣
↪backslashes
import forcespro
import forcespro.nlp
```

Note that this reduces the portability of any scripts using FORCESPRO, as it hard-codes the location of FORCESPRO inside the script.

3.3.4 Keeping FORCESPRO up to date

In order to obtain the latest version of the FORCESPRO client, a Python script for automatic upgrading is available.

In order to use it, navigate to the FORCESPRO client directory and execute the *updateClient.py* script in Python.

```
$ cd /path/to/forcespro
$ python updateClient.py
```

Alternatively, the FORCESPRO client can also be updated through MATLAB, see *Keeping FORCESPRO up to date*.

3.3.5 Installing and running older versions of FORCESPRO

Older versions of FORCESPRO can be installed and run by specifying the server URL as `https://forces-X-Y-Z.embotech.com`, replacing `X-Y-Z` with the version numbers of a selected FORCESPRO version `vX.Y.Z`. First, sync your client to the server by updating it with

```
python updateClient.py https://forces-X-Y-Z.embotech.com
```

Then, in order to generate a solver, set the code generation server as

```
codeoptions.server = 'https://forces-X-Y-Z.embotech.com'
```

Chapter 4

Backward Compatibility

- *Determining Client Version*
- *Changes from Version 6.4.0*
- *Changes from Version 6.3.0*
- *Changes from Version 6.2.0*
- *Changes from Version 6.1.0*
- *Changes from Version 6.0.0*
- *Changes from Version 5.0.1*
- *Changes from Version 5.0.0*
- *Changes from Version 4.3.0*
- *Changes from Version 4.2.0*
- *Changes from Version 4.1.0*

FORCESPRO uses continuous deployment as development policy, which may cause also minor new versions not to behave identical to previous ones. This chapter summarizes code-generation options that either recently changed default behavior or were introduced to allow restoring behavior of a previous FORCESPRO version.

4.1 Determining Client Version

For determining the current version of your client, you may invoke the following command:

Matlab

Python

```
VER = FORCESversion();
```

```
# not yet supported for Python client
```

The code-generation servers of FORCESPRO only remain available for a certain period of time (but for at least one year after release, see *Product Life Cycle*). The date when the codegen server of the current client version is planned to go offline can be retrieved as second output argument of the same command:

Matlab

Python

```
[VER, OFFLINEDATE] = FORCESversion();
```

```
# not yet supported for Python client
```

4.2 Changes from Version 6.4.0

From version `6.4.0`, support for server connections using `WSDL` has been dropped. To connect to the FORCESPRO server using `WSDL`, a downgrade to FORCESPRO `6.3.0` or earlier would be required.

From version `6.4.0`, the cross compilation options `win`, `mac` and `gnu` have been replaced by the `host_os` codeoption.

From version `6.4.0`, the prefix of internal FORCESPRO solvers generated from Y2F has been shortened from `internal_` to `i_`.

4.3 Changes from Version 6.3.0

From version `6.3.0`, the MEX interfaces `<solvername>_serialize_mex` and `<solvername>_deserialize_mex` for (de)serializing C parameters return `success=1` in case of success and `success=0` in case of failure (prior to that version it has been the other way around).

From version `6.3.0`, the MEX interface for evaluating the inequality constraints is actually called `<solvername>_inequalities` as documented in Section Section 9.6 (and not `<solvername>_ineq` as before).

4.4 Changes from Version 6.2.0

From version `6.2.0`, the BFGS initialization is a field of the model struct instead of the codeoptions. As such, they must be set via `model.bfgs_init` and not via `codeoptions.nlp.bfgs_init` anymore. Any existing formulations that still use `codeoptions.nlp.bfgs_init` would need to be adapted from `6.2.0` onwards. More information on BFGS initialization can be found in *Hessian approximation*.

From version `6.2.0`, the option `ADMMfactorize` is enabled (`= 1`) by default for the ADMM solvemethod. The prior default `ADMMfactorize = 0` has been deprecated and will be removed completely in a future release.

4.5 Changes from Version 6.1.0

From version `6.1.0`, the default MATLAB dump tool has been changed from legacy dumps to symbolic dumps. In order to continue using the legacy dump tool, you need to explicitly specify `ForcesDumpType.LegacyDumpGeneratedC` as input argument to the functions `ForcesDumpProblem, ForcesDumpFormulation, ForcesDumpAll`.

4.6 Changes from Version 6.0.0

From version `6.0.0`, the C interface has changed. The *<solvername>_solve* function takes as an additional argument the memory buffer containing all solver variables. In order to update to version `6.0.0`, you need to extend your C interface according to the following code snippet:

```
/* additional include required for memory buffer */
#include "<solvername>/include/<solvername>_memory.h"

/* opaque pointer to memory buffer */
<solvername>_mem * mem;
/* Get 0-th memory buffer */
mem = <solvername>_internal_mem(0);
/* call solve function */
exitflag = <solvername>_solve(&params, &output, &info, mem, NULL, extfunc_eval);
```

If you want full control over memory allocations, or if you require multiple memory buffers (e.g. for running a solver in parallel), you can find more instructions in sections *Main Targets* and *Memory allocations*.

Alternatively, you can resort to the legacy C interface equivalent to prior FORCESPRO versions by enabling the compatibility option

- `codeoptions.legacy_interface = 1`

From version `6.0.0`, the external function evaluations signature has changed. The new name of the function is *<solvername>_adtool2forces* (the function name is specific to the solver) and it returns an integer value. The signature is shown below:

```
int <solvername>_adtool2forces (
    double *x, /* primal vars */
    double *y, /* eq. constraint multipliers */
    double *l, /* ineq . constraint multipliers */
    void *p, /* runtime parameters (passed as void pointer) */
    double *f, /* objective function ( incremented in this function ) */
    double *nabla_f , /* gradient of objective function */
    double *c, /* dynamics */
    double *nabla_c , /* Jacobian of the dynamics ( column major ) */
    double *h, /* inequality constraints */
    double *nabla_h , /* Jacobian of inequality constraints ( column major ) */
    double *H, /* Hessian ( column major ) */
    int stage, /* stage number (0 indexed ) */
    int iteration, /* solver iteration count */
    int threadID /* thread id */
);
```

4.7 Changes from Version 5.0.1

From version `5.0.1`, license options for FORCESPRO code generation are set in `codeoptions.license`. Previous options:

- `codeoptions.useFloatingLicense`
- `codeoptions.license_file_name`

have now been replaced by:

- `codeoptions.license.use_floating_license`

- `codeoptions.license.static_license_file_name`

The old options are still available in `5.0.1` for backwards compatibility but the new options override them when set.

4.8 Changes from Version 5.0.0

From version `5.0.0`, FORCESPRO uses CasADi v3.5.5 as default AD tool for both MATLAB and Python client. The following code-generation option can be used for reverting to previous FORCESPRO behavior (since v4.3.0) using CasADi v3.5.1:

- `codeoptions.nlp.ad_tool = 'casadi-3.5.1'`

From version `5.0.0`, FORCESPRO uses an improved implementation of the linear solver as default whenever `nlp.linear_solver` is set to `'symm_indefinite'`. The following code-generation option can be used for reverting to previous FORCESPRO behavior:

- `codeoptions.nlp.linear_solver = 'symm_indefinite_legacy'`

From version `5.0.0`, FORCESPRO uses RestAPI for server communications. To revert to previous communication methods, the following options can be used (see *MATLAB network communications*/*Python network communications*):

Matlab

Python

```
codeoptions.server_connection = ForcesWeb.ServerConnections.WSDL
codeoptions.server_connection = ForcesWeb.ServerConnections.WSDL_legacy
```

```
codeoptions.server_connection = 'WSDL'
```

4.9 Changes from Version 4.3.0

From version `4.3.0`, FORCESPRO uses CasADi v3.5.1 as default AD tool for both MATLAB and Python client. The following code-generation option can be used for reverting to previous FORCESPRO behavior using CasADi v2.4.2:

- `codeoptions.nlp.ad_tool = 'casadi-2.4.2'`

4.10 Changes from Version 4.2.0

From version `4.2.0`, the following code-generation options can be used for reverting to previous FORCESPRO behaviors:

- `codeoptions.legacyNetworkConnections = 1`. From version `4.2.0`, a new communication method is used to connect to the codegen service for safety and stability reasons. Use this option to use the legacy method of communication.
- `codeoptions.platform = 'Speedgoat-Legacy-x86'`. From version `4.2.0`, use this option for Mobile Speedgoat platforms on earlier versions of MATLAB (earlier than R2018b).

  From version `4.2.0`, the option `codeoptions.platform = 'Speedgoat-x86'` supports MATLAB versions from R2018b till R2020a, while option `codeoptions.platform = 'Speedgoat-QNX'` supports MATLAB R2020b and later.

4.11 Changes from Version 4.1.0

From version `4.1.0`, the following code-generation options can be used for reverting to previous FORCESPRO behaviors:

- `codeoptions.separateCasadiFiles = 1`. From version `4.1.0`, the old `_model` files are all gathered in a single `_casadi` file. Use this option to enable the old behavior, i.e. split model files.
- `codeoptions.size_one_param_as_array = 1`. From version `4.1.0`, when using the `PDIP_NLP` method only, all parameters of size one are treated as scalars by default in order to be compatible with the MATLAB coder. This option enables users to revert to the previous behavior, i.e. scalar parameters as arrays of size one.

Chapter 5

Y2F Interface

- *Installing Y2F*
- *Generating a solver*
- *Calling the solver*
- *Solver info*
  - *Exitflags*
  - *Additional diagnostics*
- *Performance*
- *Examples*

YALMIP is a high-level modeling language for optimization in MATLAB. It is very convenient to use for modeling various optimization problems, including convex quadratic programs, for example. YALMIP allows you to write self-documenting code that reads very much like a mathematical description of the optimization model.

To combine the computational efficiency of FORCESPRO with the ease-of-use of YALMIP, we have created the interface Y2F. Y2F very efficiently detects the inherent structure in the optimization problem, and uses the FORCESPRO backend to generate efficient code for it. All you need to do is to replace YALMIP's `optimizer` function, which pre-builds the optimization problem such that subsequent evaluations become very inexpensive, by Y2F's `optimizerFORCES` function, which is fully API-compatible with `optimizer`.

This interface is provided with all variants of FORCESPRO, starting with *Variant S*.

You can read more about the concept of YALMIP's `optimizer` here.

Important: The Y2F interface supports convex decision making problems, with or without binary variables.

5.1 Installing Y2F

Y2F is included in the FORCESPRO client. If `optimizerFORCES` is not found on your MATLAB path, you need to add the `FORCESPRO/Y2F/Y2F` directory to it, e.g. by typing:

```
addpath /home/user/FORCESPRO/Y2F/Y2F
```

on your MATLAB command prompt.

Note: We recommend to always use the Y2F version shipped with the FORCESPRO client as Y2F versions are only compatible with certain FORCESPRO versions (see https://github.com/embotech/Y2F/#releases). For example, Y2F 0.2.0 requires FORCESPRO 6.2.0 or higher.

Of course, you also need a working installation of YALMIP, which you can download from https://yalmip.github.io/download/.

5.2 Generating a solver

A YALMIP model consists of a constraint object, which we name `const` and an objective function `obj`. You can create an `optimizer` object that has most of the work YALMIP needs to do before calling a solver (called canonicalization) already saved. The only parts missing are the parameters of the problem, which you can specify when calling `optimizer`:

```
P = optimizer(Con, Obj, Options, Parameters, WantedVariables); % YALMIP syntax
```

With Y2F, you can have the same syntax but creating a FORCESPRO solver:

```
P = optimizerFORCES(Con, Obj, Options, Parameters, WantedVariables, [ParameterNames],␣
↪[OutputNames]);
```

where Options is a FORCESPRO `codeoptions` struct (see the *Solver Options* section for more information). The two last arguments are optional cell arrays of strings specifying the names of the parameters and the wanted variables. These will be used for naming e.g. the in- and output ports of the generated Simulink block.

5.3 Calling the solver

There are several ways of calling the generated solver:

1. Using the `optimizerFORCES` object, which again is API compatible with YALMIP's `optimizer` object:

```
[wantedVariableValues, exitflag, info] = P{Parameters}; % YALMIP syntax
```

2. Using the generated MATLAB (MEX) interface (type `help solvername` at the MATLAB command prompt for more information):

```
problem.ParameterName1 = value1; problem.ParameterName2 = value2;
[output, exitflag, info] = solvername(problem);
wantedVariable = output.outputName1;
```

3. Via the generated Simulink block (see interfaces folder of the generated code).

5.4 Solver info

5.4.1 Exitflags

One should always check whether the solver has exited without an error before using the solution. Possible values of `exitflag` are presented in *Exitflag values*.

5.4.2 Additional diagnostics

The solver returns additional information to the optimizer in the `info` struct. Some of the fields are described in Table 5.1. Depending on the method used, there will also be other fields describing the quality of the returned result.

Table 5.1: Info values

Info	Description
`info.it`	Number of iterations. In branch-and-bound mode this is the number of convex problems solved in total.
`info.solvetime`	Total computation time in seconds.
`info.pobj`	Value of the objective function.
`info.it2opt`	(only branch-and-bound) Number of convex problems solved for finding the optimal solution. Note that often the optimal solution is found early in the search, but in order to certify (sub-)optimality, all branches have to be explored.

5.5 Performance

A performance measurement for the interface when compared to other solvers called via YALMIP and to the same problem formulated via the low-level interface of FORCESPRO (2 states, 1 input, box constraints, varying horizon) is presented in Figure 5.1. In this example, the code generated directly from YALMIP is about 10 times faster than other solvers, and only a factor 2 slower than the code generated with the low-level interface of FORCESPRO.

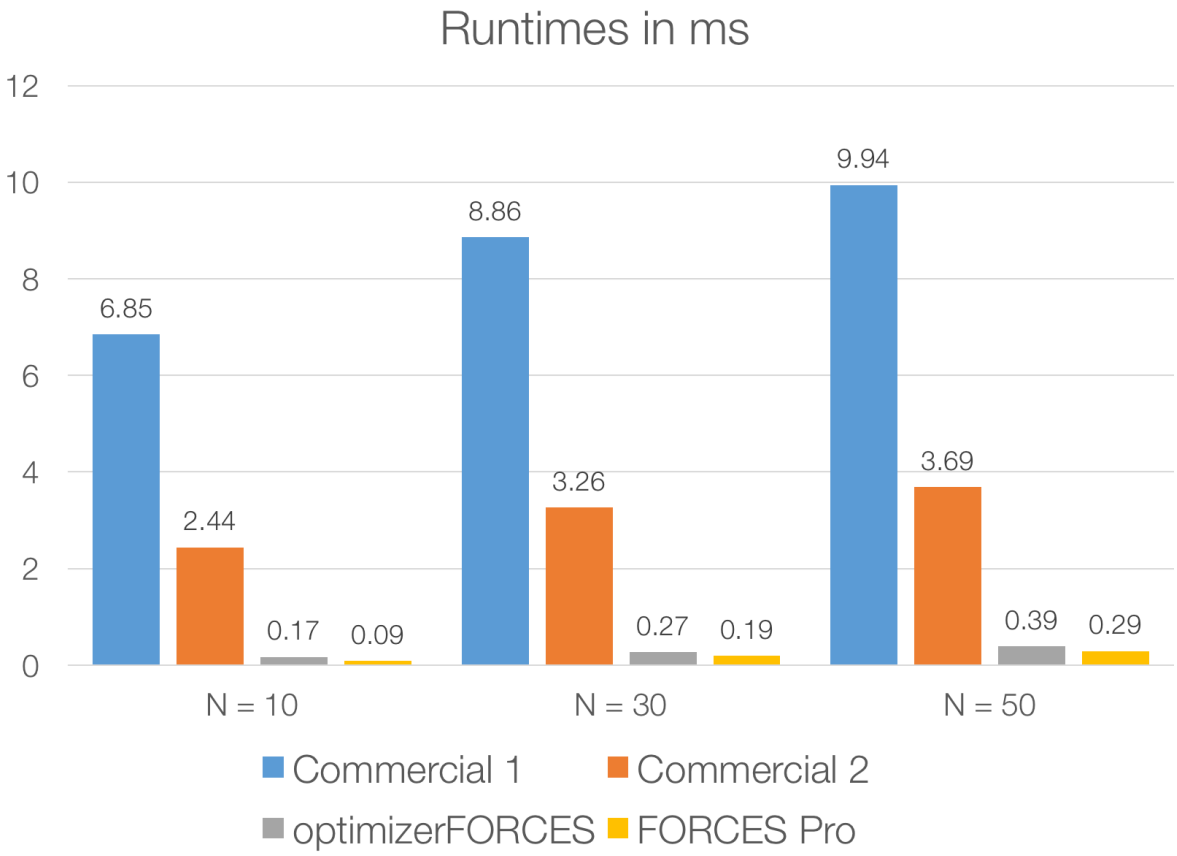


Figure 5.1: Performance comparison of the Y2F interface of FORCESPRO.

5.6 Examples

- *Y2F interface: Basic example*: Learn how to formulate problems in YALMIP easily, and then use the Y2F interface to generate code with FORCESPRO.
- *Y2F interface: Trajectory Optimization for Quadrotor Flight*: A more complex example optimizing the trajectory of a quadrotor within safe flight corridors.

Chapter 6

MathWorks Linear MPC Plugin

- *Different types of solvers*
- *Different algorithms*
- *Generating a QP solver from an MPC object*
- *Solving a QP from MPC online data*
- *Using the FORCESPRO MPC Simulink block*
- *Deploy to dSPACE MicroAutoBox II using the FORCESPRO MPC Simulink block*
- *Examples*
  - *Tracking control using a linearized model*
  - *Tracking control of a time-varying plant using LTI, adaptive and LTV MPC*

As a result of a long-term collaboration, MathWorks Inc. and embotech AG developed a MATLAB® plugin for FORCESPRO. Users are now able to use the FORCESPRO solver in MATLAB® and Simulink® from within the MATLAB® Model Predictive Control Toolbox. The plugin leverages the powerful design capabilities of the Model Predictive Control Toolbox™ and the computational performance of FORCESPRO. Starting from FORCESPRO 2.0, toolbox users can now easily define challenging control problems and solve long-horizon MPC problems more efficiently.

Model Predictive Control Toolbox™ provides functions, an app, and Simulink® blocks for designing and simulating model predictive controllers. The toolbox enables users to readily specify plant and disturbance models, horizons, constraints, and weights. User-friendly control design capabilities of Model Predictive Control Toolbox™, combined with the powerful numerical algorithms of FORCESPRO, enables code deployment of the FORCESPRO solver on real-time hardware from within MATLAB® and Simulink®, in addition to the QP solvers shipped by MathWorks. The new FORCESPRO interface comes with various features such as Simulink blocks that can generate code runnable on embedded targets such as dSPACE. The parameters of the MPC algorithm, such as plant and disturbance model, prediction horizon, constraints and move-blocking strategy can be specified directly. The toolbox enables users to run closed-loop simulations and evaluation of controller performance. User-friendly MPC design capabilities are combined with the powerful numerical algorithms of FORCESPRO. This combination of the Model Predictive Control Toolbox™ and FORCESPRO enables code deployment on real-time hardware. The generated code is highly optimized for fast computations and low memory footprint.

This interface is provided with all variants of FORCESPRO, starting with *Variant S*. It is compatible from MATLAB R2018b onwards.

The plugin mainly consists of the three following MATLAB commands which are described

in details in this chapter:

- `mpcToForces` for generating a FORCESPRO solver from an MPC object designed by the Model Predictive Control Toolbox
- `mpcmoveForces` for calling the generated solver on a specific linear time-invariant (LTI) MPC problem instance
- `mpcmoveAdaptiveForces` for calling the generated solver on a specific adaptive or time-varying MPC problem instance
- `mpcCustomSolver` for using the FORCESPRO dense QP solver as a custom solver

An auxiliary file is also exposed to the users for generating different solvers options, namely `mpcToForcesOptions`.

The following MPC features are supported:

- Continuous and discrete time plant models
- Move blocking
- Measured disturbances
- Unmeasured disturbances
- Disturbance and noise models
- Uniform or time-varying weights on outputs, manipulated variables, manipulated variables rates and a global slack variable
- Uniform or time-varying bounds on outputs, manipulated variables and manipulated variables rates
- Soft constraints
- Signal previewing on reference and measured disturbances
- Scale factors
- Nominal values
- Online updates of weights and constraints
- Built-in and custom state estimators
- Adaptive MPC and LTV MPC where plant model changes online (only available using sparse QP)

Currently, convex quadratic programs are supported by the MATLAB plugin. The current limitations of the plugin are the following:

- Mixed input-output constraints are not covered
- Off-diagonal terms on the hessian of the objective cannot be implemented
- Unconstrained problems are not supported
- No single-precision solvers, only double precision currently
- No suboptimal solutions

6.1 Different types of solvers

The plugin converts an MPC object (weights, bounds, horizons, prediction model) into a quadratic program (QP) formulated via the FORCESPRO API. One key design decision is to choose the decision variables in the quadratic program. There are two classic choices and they lead to two different formulations:

- **Dense QP**, where only the manipulated variables MV or ΔMV are decision variables. In this case, the hessian and linear constraints matrices are stored as dense matrices.
- **Sparse QP**, where MV, ΔMV, the outputs OV and the states X are decision variables. In this case, all matrices have a block sparse structure as in *Low-level interface*.

Typically, a dense QP has fewer optimization variables, zero equality constraints and many inequality constraints. Although the sparse QP is generally much larger than the dense QP its structure can be efficiently exploited to reduce the solve times. Besides, the dense formulation has an inherent flaw, which is that the condition number increases with the horizon length, especially when the plant states have large contributions to the plant inputs and outputs. Thus, the best solution is to allow users to switch to the sparse formulation, which prevents numerical blow-ups when the plant is unstable. Nevertheless, the dense formulation can be beneficial in terms of solve time when there is an important amount of move-blocking.

6.2 Different algorithms

In addition to the `general` QP solver, the FORCESPRO plugin for the MPC toolbox supports also supports a `fast` QP solver from FORCESPRO version 6.0.0. Both the `general` and the `fast` algorithms are supported for the two different solver types documented in *Different types of solvers*. The workflow for the `general` QP solver is exactly as described in the following sections. The difference in workflow for the `fast` QP solver is an additional "tuning" step which will be explained here. The core idea behind this tuning step is that it allows for a highly specialized/tailored QP solver which achieves optimal performance for a given MPC application. For this purpose FORCESPRO ships with a caching tool for collecting simulation data in order to tune the `fast` QP solver. The caching tool is a simple mechanism and interacting with it goes via three commands:

- `forcesMpcCache clear`: Deletes all stored caches completely.
- `forcesMpcCache on`: Turns on the cache by constructing a cache object. This command *must* be called before `mpcToForces` is called in order to properly cache simulation data. After the cache has been turned on and until it is turned off, every call to the generated solver via `mpcmoveForces` or `mpcmoveAdaptiveForces` will store a problem instance in the cache, so that it can later be used for tuning the `fast` QP solver.
- `forcesMpcCache off`: Stores the allocated cache in a folder named "FORCESPRO_CACHE" and deletes the cache object from the base workspace.
- `forcesMpcCache reset`: A single command for running `forcesMpcCache clear; forcesMpcCache on`.

When no cache has been stored (default) a `general` QP solver is generated when calling `mpcToForces`. If on the other hand a cache is available when calling `mpcToForces`, then a `fast` QP solver is generated. Hence, after constructing and specifying a MPC object `mpcobj`, the complete workflow for generating a `fast` QP solver is as follows:

1. Turn on the FORCESPRO cache by running `forcesMpcCache on` (or `forcesMpcCache reset`) in the MATLAB terminal.
2. Generate a `general` QP solver by calling `mpcToForces(mpcobj,options)` (see *Generating a QP solver from an MPC object*).
3. Run a simulation, using `mpcmoveForces` or `mpcmoveAdaptiveForces` respectively to compute optimal control moves. It is important that `mpcmoveForces` or `mpcmoveAdaptiveForces` is called as opposed to the generated MEX function, as that one will not be able to cache the problem instances which are needed for tuning the fast QP solver.
4. Turn off the cache by running `forcesMpcCache off` in the MATLAB terminal.

5. Generate a QP fast solver by calling `mpcToForces(mpcobj,options)` (see *Generating a QP solver from an MPC object*). Since the cache has been stored in step 4, this will trigger the automatic tuning procedure.

For examples displaying the workflow for the `fast` QP solver, see the "forcesmpc_motor" and "forcesmpc_cstr" examples shipped with the FORCESPRO client.

Important: If the MATLAB command `clear` is called in between the calls `forcesMpcCache on` and `forcesMpcCache off` the cache is deleted. Hence, this should be avoided.

6.3 Generating a QP solver from an MPC object

Given an MPC object created by the *mpc* command, users can generate a QP solver tailored to their specific problem via the following command:

```
% mpcobj is the output of mpc(...)
% options is the output of mpcToForcesOptions(...)

[coredata, statedata, onlinedata] = mpcToForces(mpcobj, options);
```

For the LTI MPC, two types of QP solvers can be generated via `mpcToForces`: a **sparse** solver that corresponds to a multistage formulation as in *Low-level interface* and a **dense** solver that corresponds to a one-stage QP with inequality constraints only. For the adaptive and time-varying MPC formulations only the **sparse** QP solver is supported.

The API of `mpcToForces` is described in more details in the tables below. The `mpcToForces` command expects an MPC object **mpcobj** and a structure **options** generated by `mpcToForcesOptions` as inputs.

- **mpcobj**: LTI/adaptive/LTV MPC controller designed by Model Predictive Control Toolbox
- **options**: Object that provides solver generation options.

Inside `mpcToForces`, the FORCESPRO server can generate two types of solvers:

- *customForcesSparseQP* when the option 'sparse' is set. An m-file named customForcesSparseQP.m with the corresponding mex interface as well as the solver libraries and header in the customForcesSparseQP folder. In this particular case (sparse), the name of the solver can be set by the user. The sparse formulation is available for LTI, adaptive and LTV MPC controllers.
- *customForcesDenseQP* when the option 'dense' is set. An m-file named customForcesDenseQP.m with the corresponding mex interface as well as the solver libraries and header in the customForcesDenseQP folder. In this particular case (dense), the solver name cannot be changed by the user. The dense formulation is available for the LTI MPC solver only.

The outputs of `mpcToForces` consist of three structures **coredata**, **statedata** and **onlinedata**.

- **coredata**: Stores constant data needed to construct quadratic program at run-time
- **statedata**: Stores prediction model states and last optimal MV.

  It contains 4 fields. The index k stands for the current simulation time.

  - When built-in state estimation is used:

    In this case, users should not manually change any field at run-time.

    * `Plant` is the estimated plant state $x_p[k|k-1]$
    * `Disturbance` is the estimated disturbance states $x_d[k|k-1]$

    * `Noise` is the estimated measurement noise states $x_n[k|k-1]$
    * `Covariance` is the state estimate error covariance matrix $P[k|k-1]$. In the LTI case, a static Kalman filter (KF) is used resulting in a constant covariance matrix $P[k|k-1] = P$. In the adaptive/LTV case, a linear time-varying Kalman filter (LTVKF) is utilized resulting in varying covariance matrices in general during a simulation.
    * `LastMove` is the optimal manipulated variables at the previous sample time
  - When custom state estimation is used:

    In this case, user should manually update Plant, Disturbance (if used), Noise (if used) fields at run-time but leave LastMove alone.

    * `Plant` is the estimated plant state $x_p[k|k]$
    * `Disturbance` is the estimated disturbance states $x_d[k|k]$
    * `Covariance` is an empty array
    * `Noise` is the estimated noise states $x_n[k|k]$
    * `LastMove` is the optimal manipulated variables at the previous solve

- **onlinedata**: Represents online signals

  It contains up to three fields:

  - `signals`, a structure containing following fields:
    * *ref*: (references of Output Variables). Default is a matrix of 1-by-ny values (up to p rows for previewing).
    * *mvTarget*: (references of Manipulated Variables). Default is a matrix of 1-by-nmv values (up to p rows for previewing).
    * *md*: (when Measured Disturbance is present). Default is a matrix of 1-by-nmd values (up to p+1 rows for previewing).
    * *ym*: (when using the built-in estimator). Default is a matrix of nmy-by-1 values.
    * *externalMV*: (when UseExternalMV is true in the options object). Default is a matrix of nmv-by-1 values.
  - `weights`, a structure containing the following fields:
    * *y*: (when UseOnlineWeightOV is enabled). Default is a matrix of 1-by-ny or p-by-ny values depending on if the weights changed over the prediction horizon during mpcobj generation (up to p rows for previewing).
    * *u*: (when UseOnlineWeightMV is enabled). Default is a matrix of 1-by-nmv or p-by-nmv values depending on if the weights change over the prediction horizon during mpcobj generation (up to p rows for previewing).
    * *du* :(when UseOnlineWeightMVRate is enabled). Default is a matrix of 1-by-nmv or p-by-nmv values depending on if the weights change over the prediction horizon during mpcobj generation (up to p rows for previewing).
    * *ecr*: (when UseOnlineWeightECR is enabled). Default is a matrix of 1-by-1 values.
  - `limits`, a structure containing the following fields:
    * *ymin*: (when UseOnlineConstraintOVMin is enabled). Default is a matrix of p-by-ny values (up to p rows for previewing).
    * *ymax*: (when UseOnlineConstraintOVMax is enabled). Default is a matrix of p-by-ny values (up to p rows for previewing).
    * *umin*: (when UseOnlineConstraintMVMin). Default is a matrix of p-by-nmv values (up to p rows for previewing).

    * *umax*: (when UseOnlineConstraintMVMax). Default is a matrix of p-by-nmv values (up to p rows for previewing).
    * *dumin*: (when UseOnlineConstraintMVRateMin). Default is a matrix of p-by-nmv values (up to p rows for previewing).
    * *dumax*: (when UseOnlineConstraintMVRateMax). Default is a matrix of p-by-nmv values (up to p rows for previewing).
  - `model` (when UseAdaptive or UseLTV), a structure containing following fields:
    * *A*: When UseAdaptive, *A* is a matrix of nxp-by-nxp values. When UseLTV, *A* is a matrix nxp-by-nxp-by-(p+1) values (up to p+1 for previewing).
    * *B*: When UseAdaptive, *B* is a matrix of nxp-by-nu values. When UseLTV, *B* is a matrix nxp-by-nu-by-(p+1) values (up to p+1 for previewing).
    * *C*: When UseAdaptive, *C* is a matrix of ny-by-nxp values. When UseLTV, *C* is a matrix ny-by-nxp-by-(p+1) values (up to p+1 for previewing).
    * *D*: When UseAdaptive, *D* is a matrix of ny-by-nu values. When UseLTV, *D* is a matrix ny-by-nu-by-(p+1) values (up to p+1 for previewing).
    * *U*: When UseAdaptive, *U* is a matrix of nu-by-1 values. When UseLTV, *U* is a matrix nu-by-1-by-(p+1) values (up to p+1 for previewing).
    * *Y*: When UseAdaptive, *Y* is a matrix of ny-by-1 values. When UseLTV, *Y* is a matrix ny-by-1-by-(p+1) values (up to p+1 for previewing).
    * *X*: When UseAdaptive, *X* is a matrix of nxp-by-1 values. When UseLTV, *X* is a matrix nxp-by-1-by-(p+1) values (up to p+1 for previewing).
    * *DX*: When UseAdaptive, *DX* is a matrix of nxp-by-1 values. When UseLTV, *DX* is a matrix nxp-by-1-by-(p+1) values (up to p+1 for previewing).

  Remark: When a field listed above has potentially multiple rows (e.g. up to p+1 rows), each row referring to a stage of a prediction horizon, the last provided row is repeated for the rest of the prediction horizon in case the number of rows is less than the number of stages. Moreover, if such a field is left empty, the default values from coredata will be utilized for the corresponding field instead.

In order to provide the code-generation options to `mpcToForces`, the user needs to run the command `mpcToForcesOptions` with one of the following two arguments as input:

- "dense" for generating the options of a one-stage dense QP solvers
- "sparse" for generating the options a multistage QP solver.

The structures provided by the `mpcToForcesOptions` command have the following MPC related fields in common between the "dense" and "sparse" case:

- *SkipSolverGeneration*. When set to *true*, only structures are returned. If set to *false*, a solver mex interface is generated and the structures are returned. Default value is *false*.
- *UseOnlineWeightOV*. When set to *true*, it allows Output Variables weights to vary at run time. Default is *false*.
- *UseOnlineWeightMV*. When set to *true*, it allows Manipulated Variables weights to vary at run time. Default is *false*.
- *UseOnlineWeightMVRate*. When set to *true*, it allows weights on the Manipulated Variables rates to vary at run time. Default is *false*.
- *UseOnlineWeightECR*. When set to *true*, it allows weights on the ECR to change at run time. Default is *false*.
- *UseOnlineConstraintOVMax*. When set to *true*, it allows updating the upper bounds on Output Variables at run time. Default is *false*.

- *UseOnlineConstraintOVMin*. When set to *true*, it allows updating the lower bounds on Output Variables at run time. Default is *false*.
- *UseOnlineConstraintMVMax*. When set to *true*, it allows updating the upper bounds on Manipulated Variables at run time. Default is *false*.
- *UseOnlineConstraintMVMin*. When set to *true*, it allows updating the lower bounds on Manipulated Variables at run time. Default is *false*.
- *UseExternalMV*. When set to *true*, the actual Manipulated Variable applied to the plant at time $k-1$ is provided as output. Default is *false*.
- *UseMVTarget*. When set to *true*, an MV reference signal is provided via the `onlinedata` structure. In this case, MV weights should be positive for proper tracking. When false, the MV reference is the nominal value by default and MV weights should be zero to avoid unexpected behavior. Default is *false*.

Both the "dense" and "sparse" options structures have the following solver related fields in common:

- *ForcesServer* is the FORCESPRO server url. Default is forces.embotech.com.
- *ForcesMaxIteration* is the maximum number of iterations in a FORCESPRO solver. Default value is 50.
- *ForcesPrintLevel* is the logging level of the FORCESPRO solver. If equal to 0, there is no output. If equal to 1, a summary line is printed after each solve. If equal to 2, a summary line is printed at every iteration. Default value is *0*.
- *ForcesInitMethod* is the initialization strategy used for the FORCESPRO interior point algorithm. If equal to 0, the solver is cold-started. If equal to 1, a centered start is computed. Default value is 1.
- *ForcesMu0* is the initial barrier parameter. It must be finite and positive. Its default value is equal to 10. A small value close to 0.1 generally leads to faster convergence but may be less reliable.
- *ForcesTolerance* is the tolerance on the infinity norm of the residuals of the inequality constraints. It must be positive and finite. Its default value is 10^{-6}.
- *ForcesTargetPlatform* for choosing a target platform to deploy the solver. Currently, dSPACE, Speedgoat, BeagleBone-Blue and AURIX are supported.

In the "sparse" solver case, there are the following additional fields:

- *UseAdaptive* When set to *true*, an adaptive MPC formulation will be created, where in each control horizon new plant matrices $(A, B, C, D)_k$ and/or a new nominals $(U, Y, X, DX)_k$ can be provided. Those matrices are kept constant over the prediction horizon.
- *UseLTV* When set to *true*, an time-varying (LTV) MPC formulation will be created, where in each control horizon a new set of plant matrices $(A, B, C, D)_{k:k+p-1}$ and/or a new set of nominals $(U, Y, X, DX)_{k:k+p-1}$ over the prediction horizon can be provided. If *UseAdaptive* is set to *true*, *UseLTV* will be ignored.
- *SolverName* for customizing the solver name.
- *UseOnlineConstraintMVRateMax* for setting MVRate upper bounds.
- *UseOnlineConstraintMVRateMin* for setting MVRate lower bounds.
- *UseOneSlackVariablePerStep* to enable one slack variable per prediction step.

6.4 Solving a QP from MPC online data

Once a QP solver has been generated it can be used to solve online MPC problems via the MATLAB command `mpcmoveForces` or `mpcmoveAdaptiveForces` as follows

```
% the coredata, statedata and onlinedata structures are outputs of␣
↪mpcToForces

[mv,statedata,info] = mpcmoveForces(coredata,statedata,onlinedata);
```

```
% the coredata, statedata and onlinedata structures are outputs of␣
↪mpcToForces
% in onlinedata, the additional model-struct is provided

[mv,statedata,info] = mpcmoveAdaptiveForces(coredata,statedata,onlinedata);
```

The outputs of the `mpcmoveForces` and `mpcmoveAdaptiveForces` commands are described below. In the list below n_m denotes the number of manipulated variables, n_x stands for the state dimension of the system implemented in the MPC object, p is the prediction horizon and k is the current solve time instant.

- **mv**: Optimal manipulated variables at current solve time instant. It is a vector of size n_m.
- **statedata**: The updated structure initialized by `mpcToForces`.
- **info**: Information structure about the FORCESPRO solve.

  It contains the following fields:

  - `Uopt` is a $p \times n_m$ matrix for the optimal manipulated variables over the prediction horizon k to $k+p-1$
  - `Yopt` is a $p \times n_y$ matrix for the optimal output variables over the prediction horizon $k+1$ to $k+p$
  - `Xopt` is a $p \times n_x$ matrix for the optimal state variables over the prediction horizon $k+1$ to $k+p$
  - `Slack` is a $p \times 1$ vector of slack variables
  - `Exitflag` is the FORCESPRO solve exit flag. If it is equal to 1, an optimal solution has been found. If it is equal to 0, the maximum number of solver iterations has been reached. A negative flag means that the solver failed to find a feasible solution.
  - `Iterations` is the number of solver iterations upon convergence or failure
  - `Cost` is the cost returned by the solver

6.5 Using the FORCESPRO MPC Simulink block

Both the FORCESPRO sparse and dense solvers can be used inside Simulink. The dense QP formulation is usable from the shipped Simulink MPC controller block directly. For this, the following steps are needed:

- Generate a custom dense FORCESPRO solver

```
options = mpcToForcesOptions('dense');
mpcToForces(mpcobj, options);
```

- Set the following settings in the MPC object

```
mpcobj.Optimizer.CustomSolver = true;
mpcobj.Optimizer.CustomSolverCodeGen = true;
```

The FORCESPRO sparse QP solver is also available via the Model Predictive Control Toolbox in Simulink. A dedicated block has been implemented for this purpose. All features of the MATLAB plugin are available through this Simulink block, namely measured disturbances, external manipulated variables, references for manipulated variables, custom state estimation as well as online weights and constraints. Configuring the block is done via the user interface shown in Figure 6.1 below. Currently only the sparse QP solver can be used via the Simulink API.

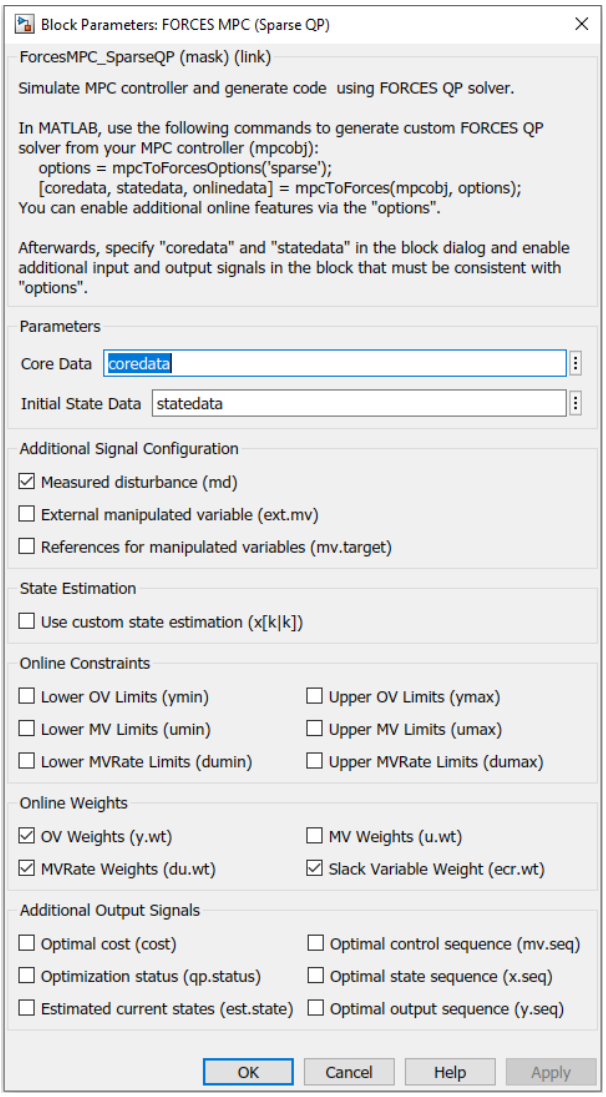


Figure 6.1: FORCESPRO LTI MPC block configuration window. FORCESPRO adaptive and LTV MPC block configuration windows have the same options available.

In order to run a simulation using the FORCESPRO Simulink block, a solver first needs to be generated via the following code for instance:

```
%% Generate FORCESPRO sparse QP solver
options = mpcToForcesOptions('sparse');

% If we want to use the adaptive MPC block, we need to set
% options.UseAdaptive = true;
```

(continues on next page)

(continued from previous page)

```
% If we want to use the time-varying MPC block, we need to set
% options.UseLTV = true;

% For this example we need to specify that online weights on the outputs,
% the input rates and the ECR slacks are used
options.UseOnlineWeightOV = true;
options.UseOnlineWeightMVRate = true;
options.UseOnlineWeightECR = true;
[coredata, statedata, onlinedata] = mpcToForces(mpcobj, options);
```

The structures *coredata* and *statedata* needed by the FORCESPRO solver are then provided to the Simulink block via the window shown in Figure 6.1.

- *coredata* is the variable name of the core data structure generated by `mpcToForces` in the base workspace.
- *initial state data* is the variable name of the state data structure generated by `mpcToForces` in the base workspace. The user is expected to populate this structure with initial states of the plant and disturbances.
- *md* checkbox should be selected if MD channels exist in the MPC object.
- *x[k|k]* checkbox needs to be selected for using a custom state estimator.
- Optional outputs provide more information. It is recommended to monitor the *qp.status* port to check whether the MPC block produces a feasible solution.

The integration of the FORCESPRO LTI MPC block in a Simulink model is shown in Figure 6.2 below. The FORCESPRO adaptive and LTV MPC block would work equivalently with the difference that both have an additional input port for the model-struct (see Figure 6.3).

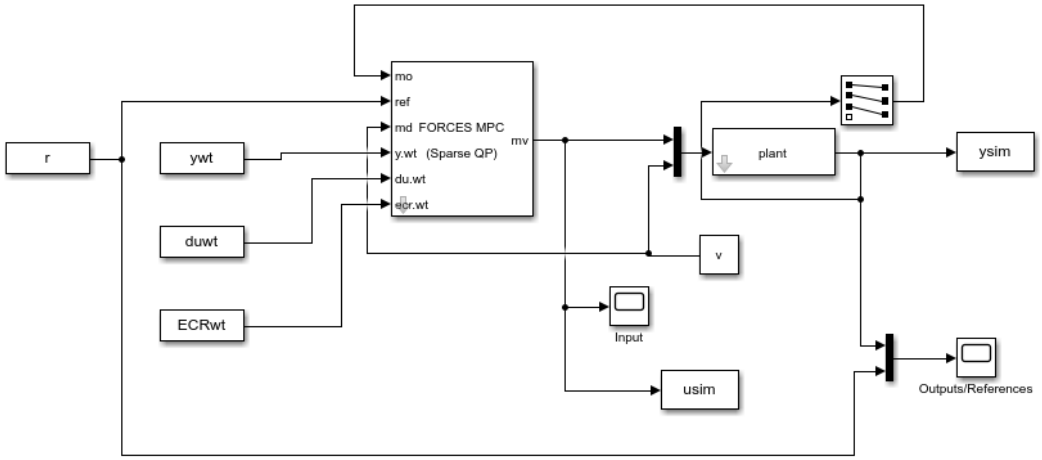


Figure 6.2: Simulink model illustrating the integration of the FORCESPRO LTI MPC block

The Simulink model can be run either by clicking on the **Run** button in Simulink or from MATLAB using the `sim` command.

```
% Start simulation.
mdl = 'forcesmpc_onlinetuning';
```

(continues on next page)

(continued from previous page)

```
open_system(mdl);                    % Open Simulink(R) Model
sim(mdl);                            % Start Simulation
```

Finally, the different FORCESPRO MPC blocks (LTI/adaptive/LTV) are available via the Library browser once the user has updated his client to the latest version of FORCESPRO, as shown in Figure 6.3 below.

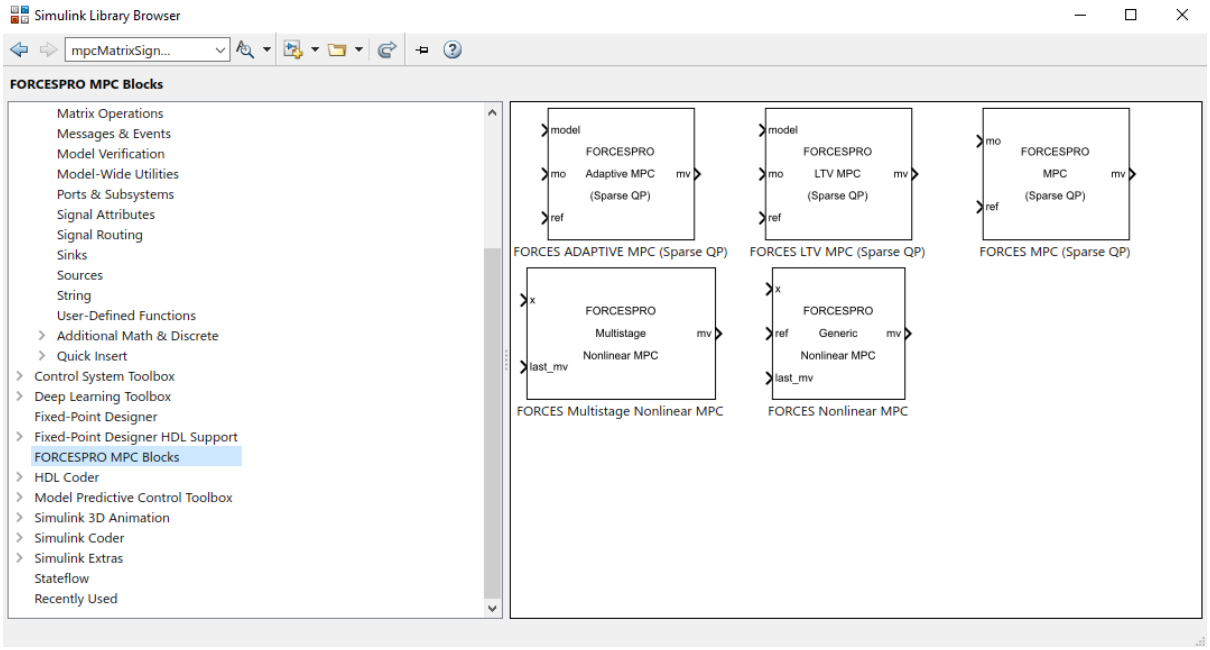


Figure 6.3: FORCESPRO MPC block in the library browser

6.6 Deploy to dSPACE MicroAutoBox II using the FORCESPRO MPC Simulink block

Note: FORCESPRO currently does not support the dSPACE MicroAutoBox II platform. This example has remained as a guideline for deployment to other target platforms.

The FORCESPRO sparse solvers can be used inside Simulink to deploy to dSPACE MicroAutoBox II. All features of the MATLAB plugin are available through this Simulink block, namely measured disturbances, external manipulated variables, references for manipulated variables, custom state estimation as well as online weights and constraints. Configuring the block is done via the user interface shown in Figure 6.4 below.

1) In order to run an MPC simulation in dSPACE using the FORCESPRO block, a solver first needs to be generated via the following code:

```
%% Generate FORCESPRO sparse QP solver
options = mpcToForcesOptions('sparse');
% For this example we need to specify that online weights on the outputs,
% the input rates and the ECR slacks are used
options.UseOnlineWeightOV = true;
options.UseOnlineWeightMVRate = true;
```

(continues on next page)

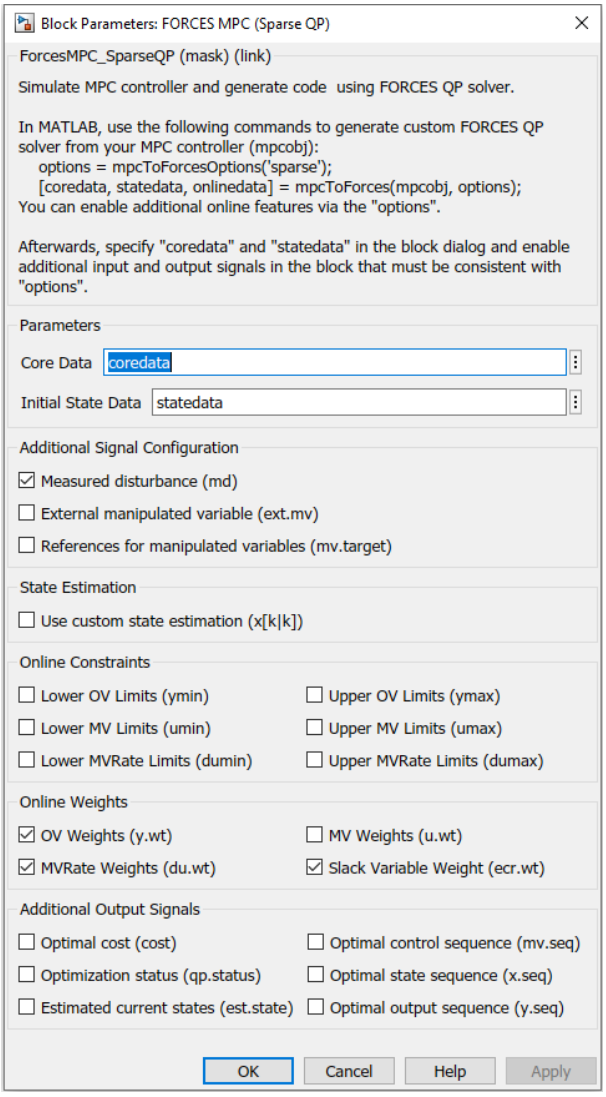


Figure 6.4: FORCESPRO MPC block configuration

(continued from previous page)

```
options.UseOnlineWeightECR = true;
options.ForcesTargetPlatform = 'dSPACE-MABII';

[coredata, statedata, onlinedata] = mpcToForces(mpcobj, options);
```

2) Note that the option *ForcesTargetPlatform* needs to be specified. The structures *coredata* and *statedata* needed by the FORCESPRO solver are then provided to the Simulink block via the window shown in Figure 6.4. The integration of the FORCESPRO MPC block in a Simulink model is shown in Figure 6.5 below.

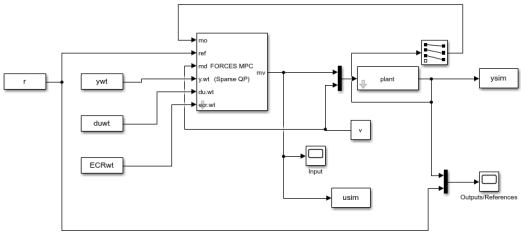


Figure 6.5: FORCESPRO MPC block integration in a Simulink model

3) When creating the Simulink Model, in the Configurations, in the “Code Generation” tab, set the options (see Figure 6.6 below):
   - System target file: **rti1401.tlc**
   - Language: C
   - Generate makefile: On
   - Template makefile: **rti1401.tmf**
   - Make command: **make_rti**
4) The Simulink model can be used for Code Generation from MATLAB in the usual way.

```
% Start Code Generation.
mdl = 'forcesmpc_onlinetuning_dSpace_MicroAutoBoxII';
open_system(mdl);                    % Open Simulink(R) Model
load_system(mdl);                    % Load Simulink(R) Model
rtwbuild(mdl);                       % Start Code Generation
```

5) After code generation the dspace compiler (Microtec PowerPC) generated files to use to run your model on the MicroAutoBox II (see Figure 6.7).
6) Open dSPACE Control Desk and select create new project (see Figure 6.8).
7) Name the project and the experiment (see Figure 6.9 and Figure 6.10).
8) Select the platform to which you will deploy the generated executable (see Figure 6.11).
9) Import the variable description file **forcesmpc_onlinetuning_dSpace_MicroAutoBoxII.sdf** in order to have access to the model variables and see the results of the execution (see Figure 6.12 and Figure 6.13).
10) Click Finish to create the project (see Figure 6.14).
11) On the project layout select the tab **Variables** and on the **forcesmpc_onlinetuning_dSpace_MicroAutoBoxII** category expand **Model Root** (see Figure 6.15).

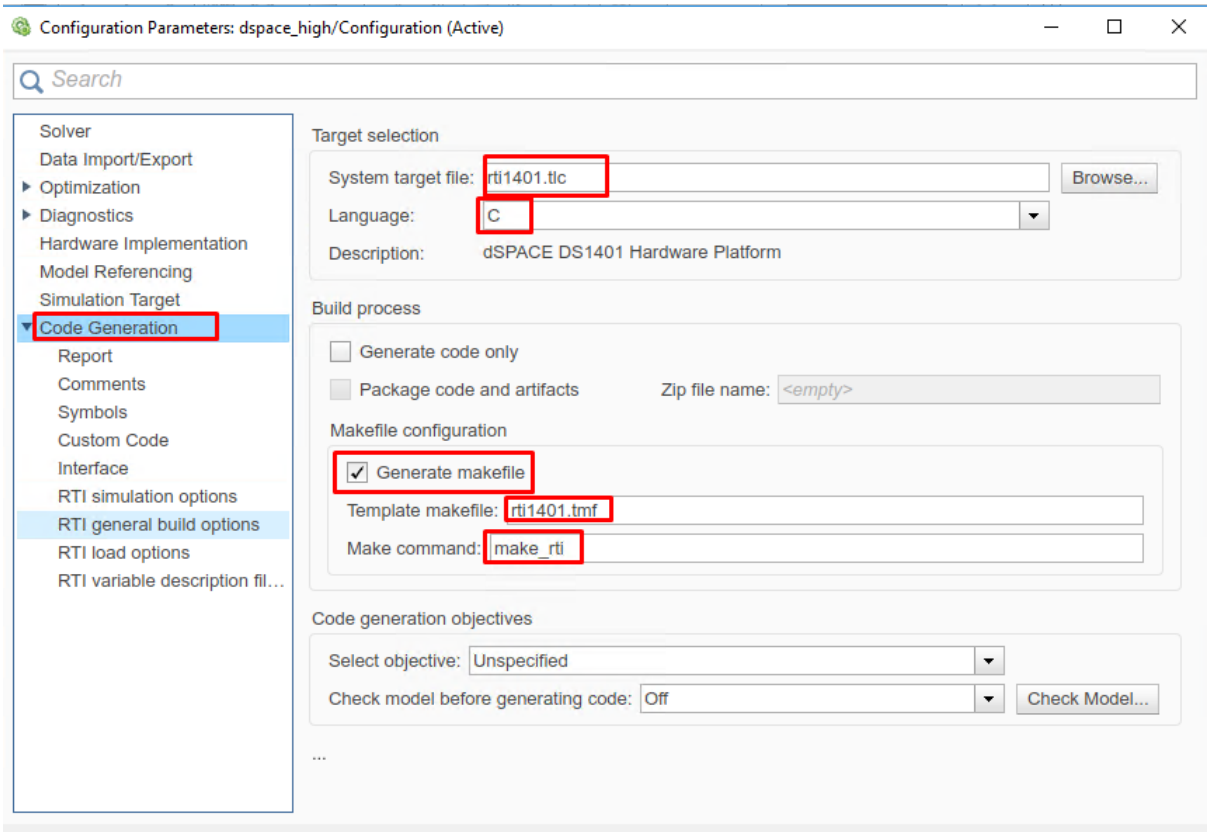


Figure 6.6: Configure Code Generation for dSPACE MicroAutoBox II

12) Select **FORCES MPC (Sparse QP)** and Drag & Drop all the output variables together to the Layout. In the opened menu select **Time Plotter** (see Figure 6.16).
13) Drag & Drop the output variables again and now choose **Display** (see Figure 6.17).
14) To see all the plots concurrently right-click on the left of the Y-axis and select **YAxes-view**> **Horizontal stacked** (see Figure 6.18).
15) Select the **Platforms/Devices** tab. Right-Click on your platform and select **Real-Time Application**> **Load**. Choose the executable file **forcesmpc_onlinetuning_dSpace_MicroAutoBoxII.ppc** (see Figure 6.19 and Figure 6.20).
16) Select **Go Online** and **Start Measuring** to see the results. (see Figure 6.21 and Figure 6.22).

6.7 Examples

The plugin comes with several examples to demonstrate its functionalities and flexibility.

You can find the MATLAB code of this example to try them out for yourself in the **examples/Matlab/mpc-toolbox-plugin/linearModels** folder that comes with your client.

The packaged examples are the following ones:

- **forcesmpc_cstr.m** is a linear time-invariant (LTI) MPC example with unmeasured outputs. It also shows how to use the MATLAB Coder for generating and running *mpcmoveForces* as a mex interface, which results in lower simulation times.
- **forcesmpc_targets.m** is an LTI MPC example with a reference on one manipulated variables

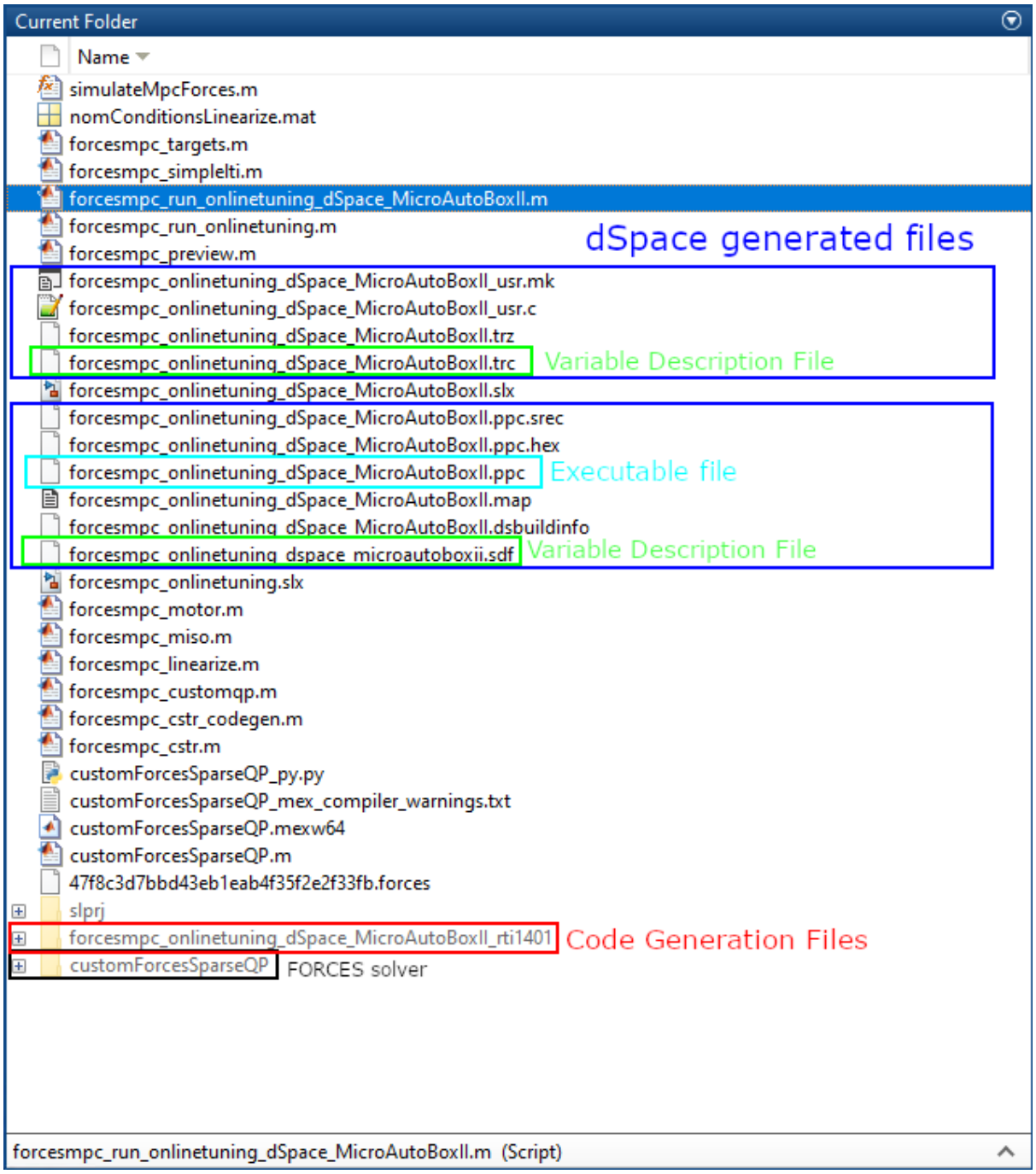


Figure 6.7: The generated files from the Simulink Code Generation

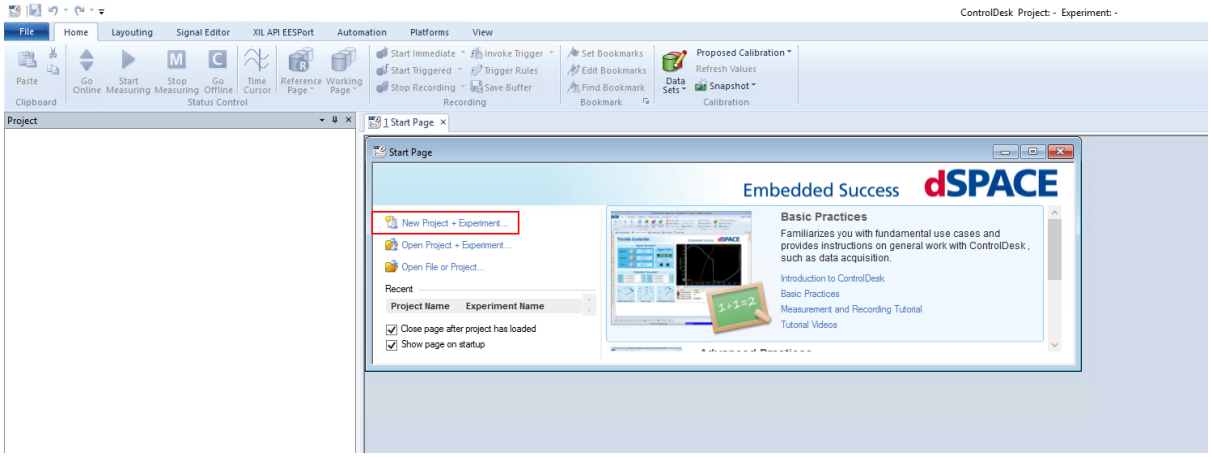


Figure 6.8: Start a new project

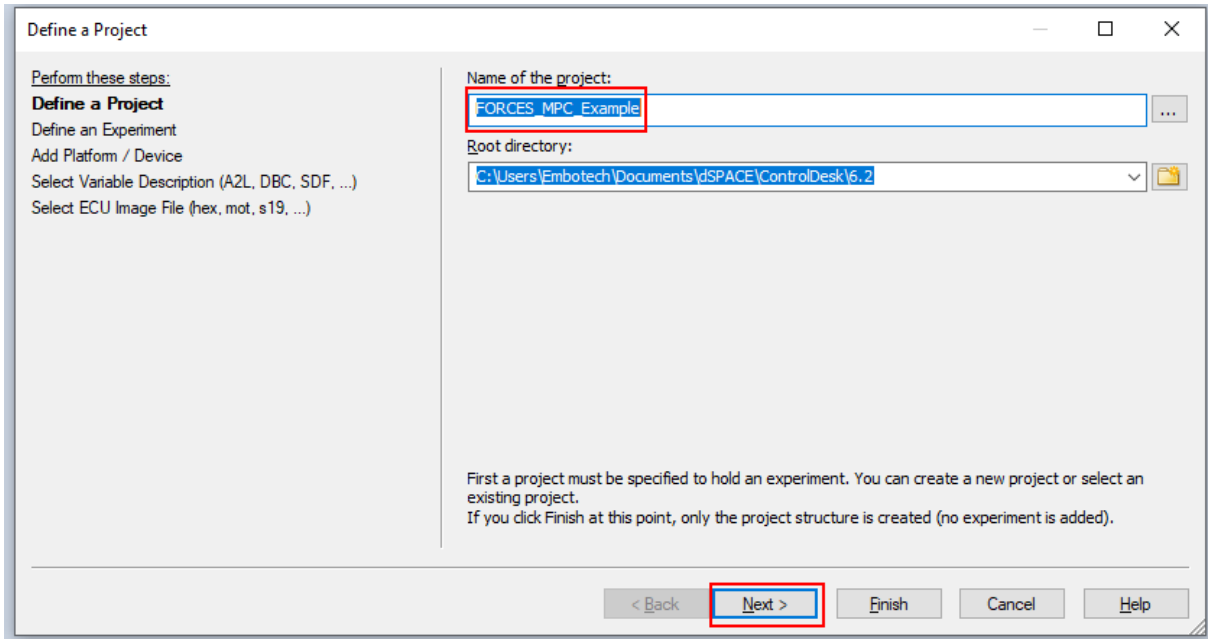


Figure 6.9: Name your project

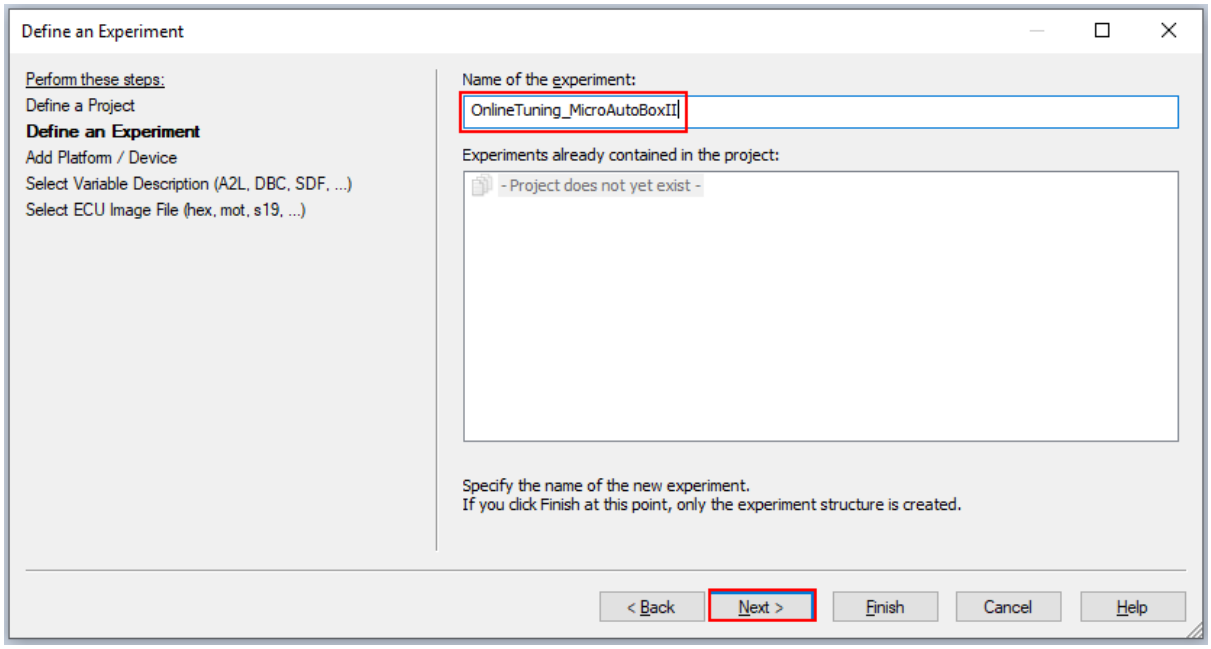


Figure 6.10: Name your experiment

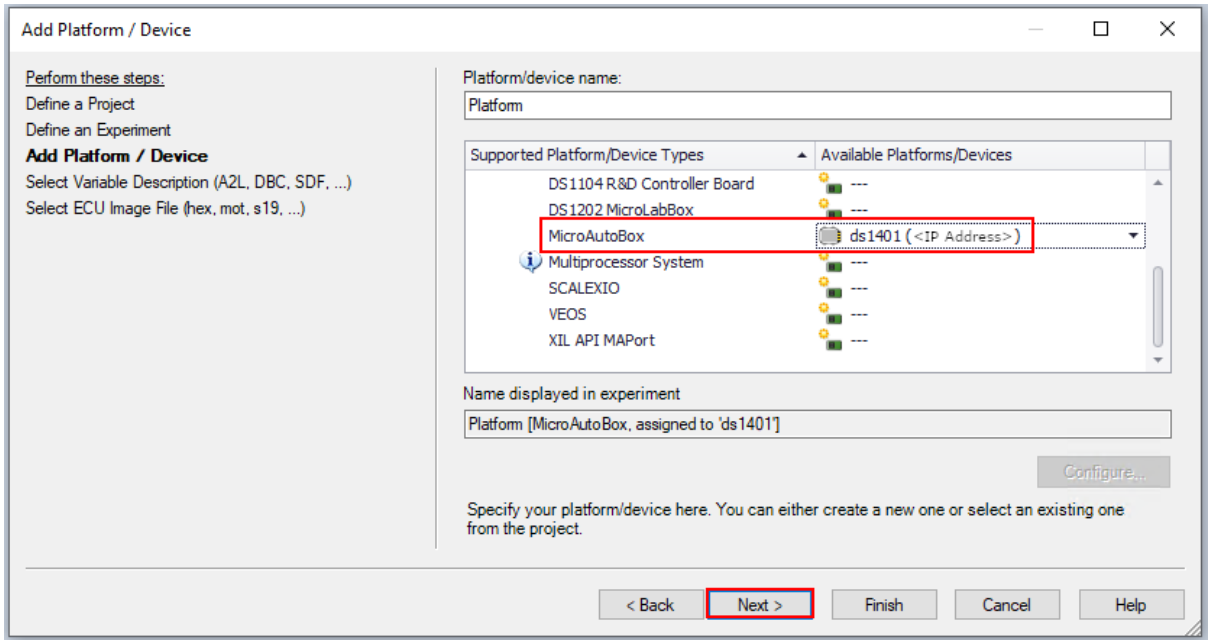


Figure 6.11: Select the MicroAutoBox platform

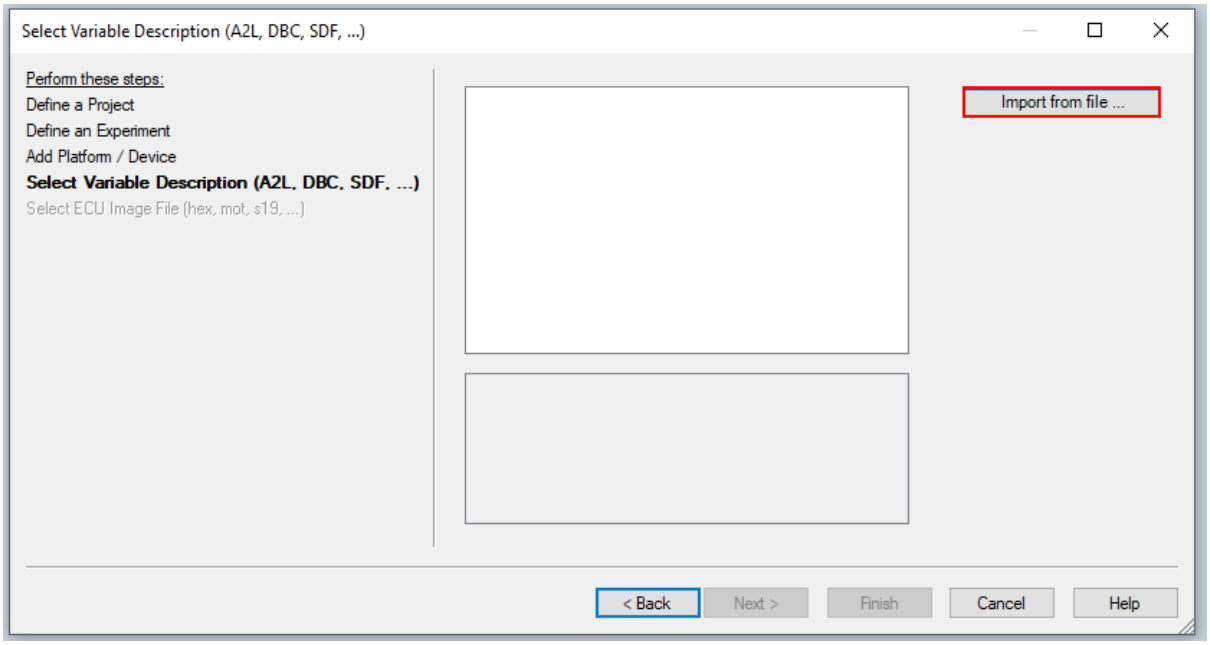


Figure 6.12: Import the variable description file

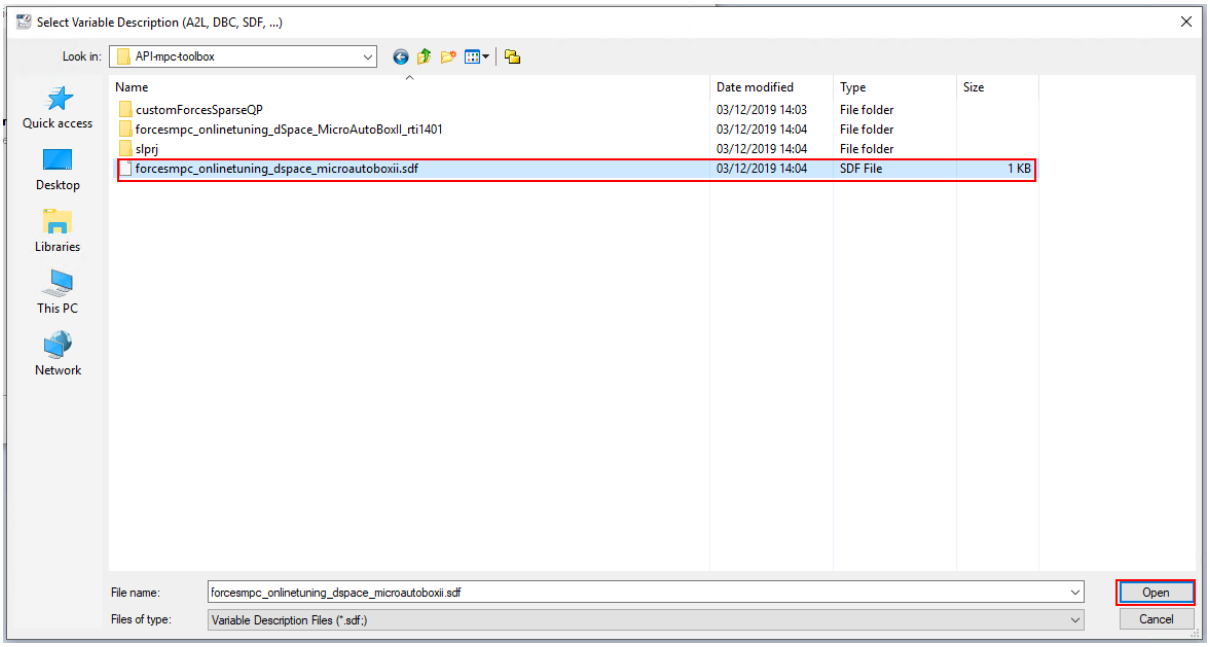


Figure 6.13: Select the sdf file with the variables description

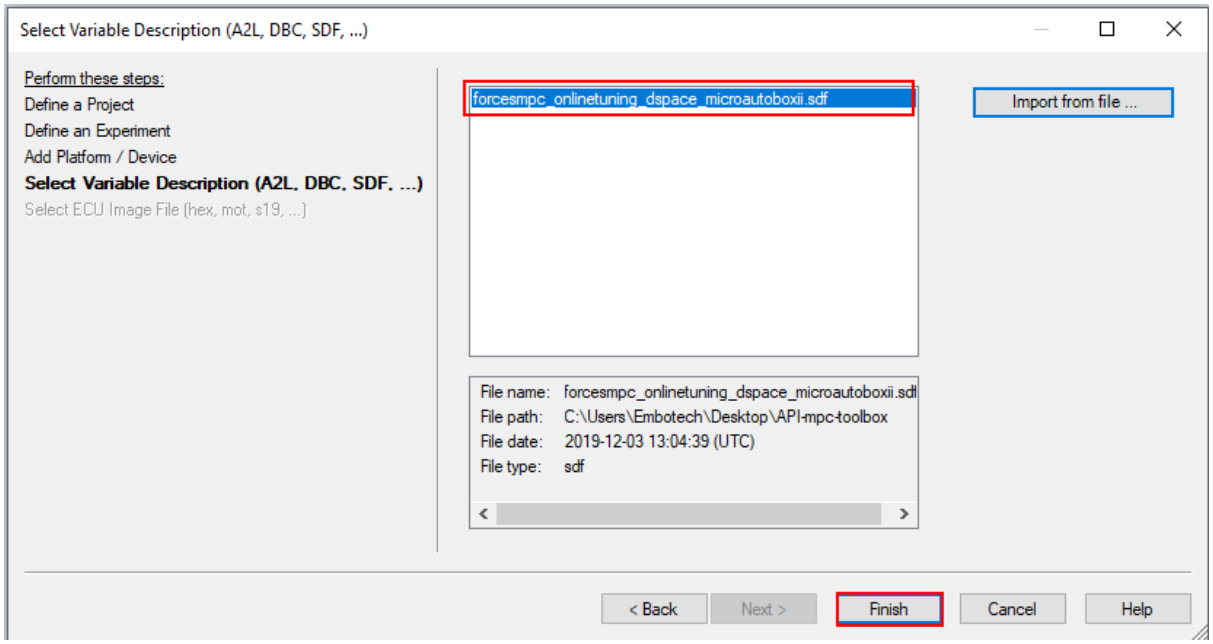


Figure 6.14: Click Finish to create the project

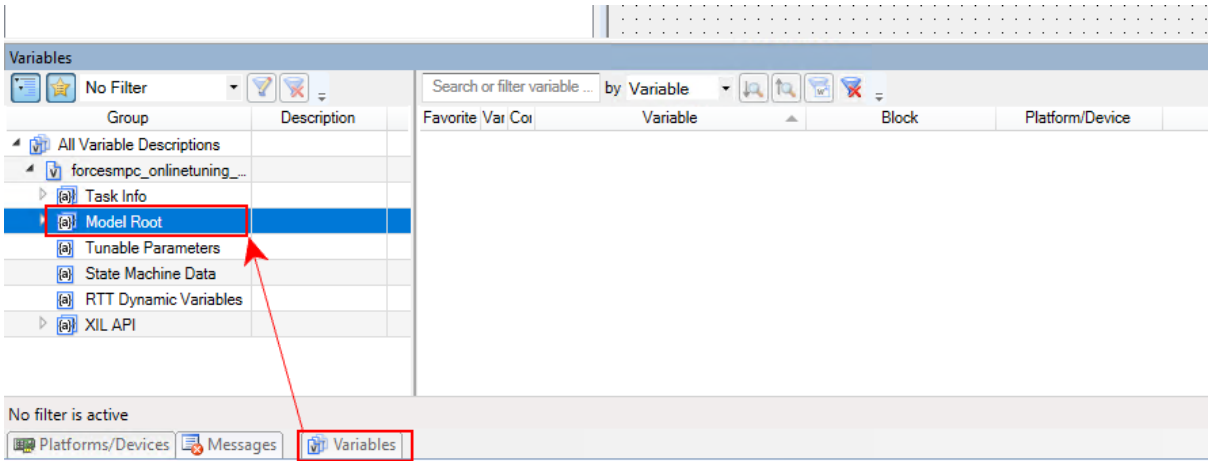


Figure 6.15: Find the model root in the variables tab

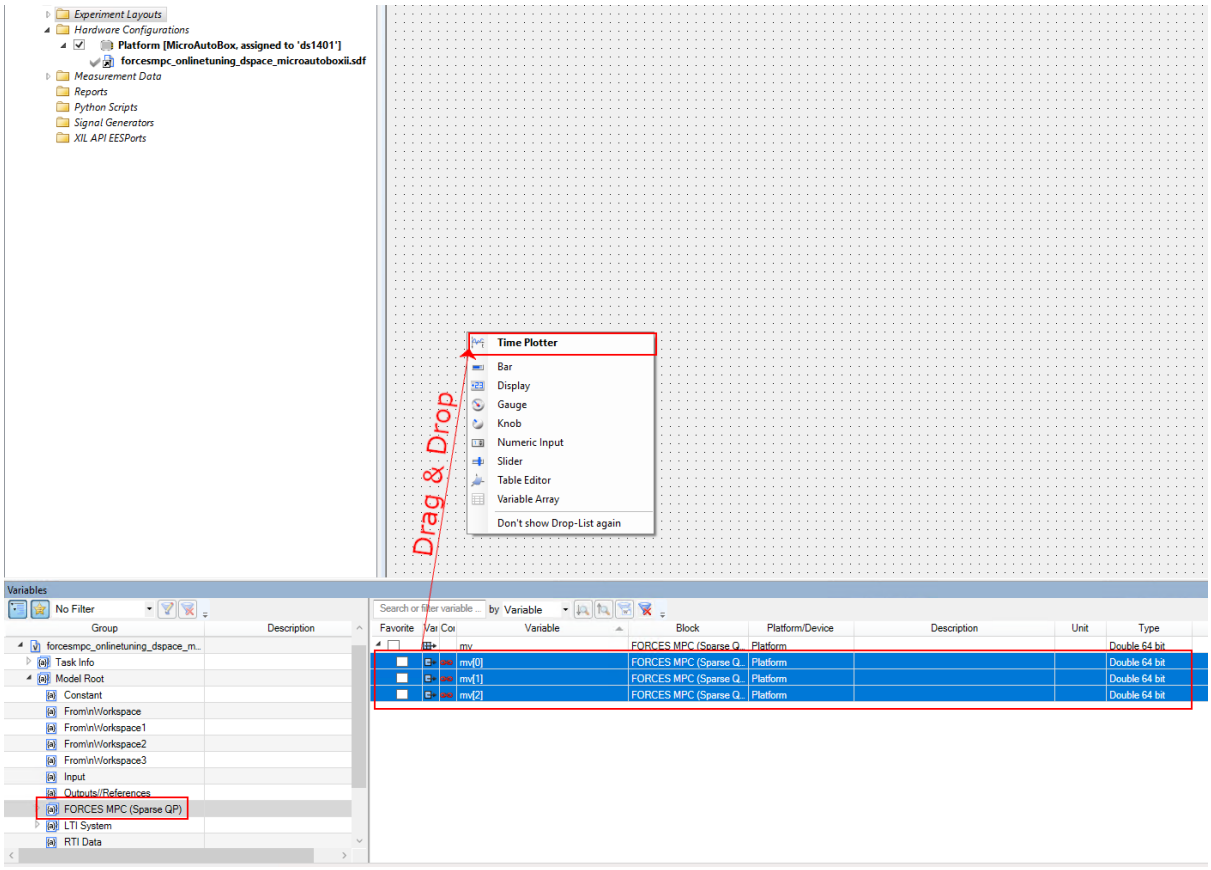


Figure 6.16: Add the variables as plots

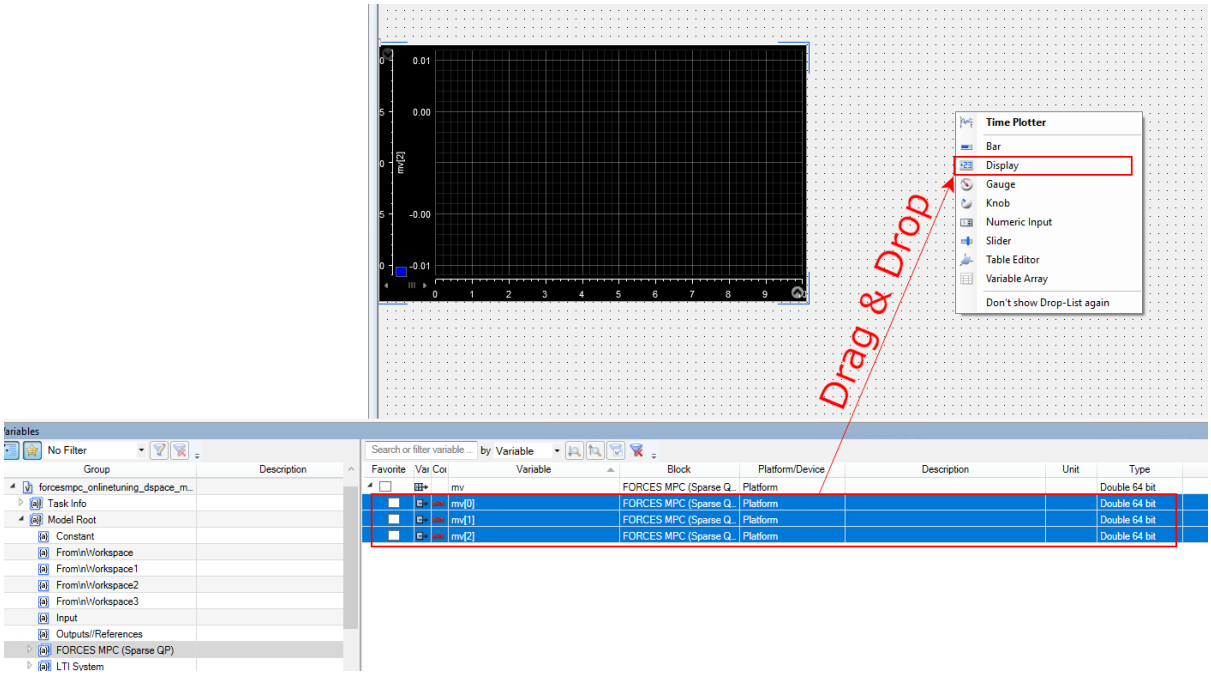


Figure 6.17: Add the variables as displays

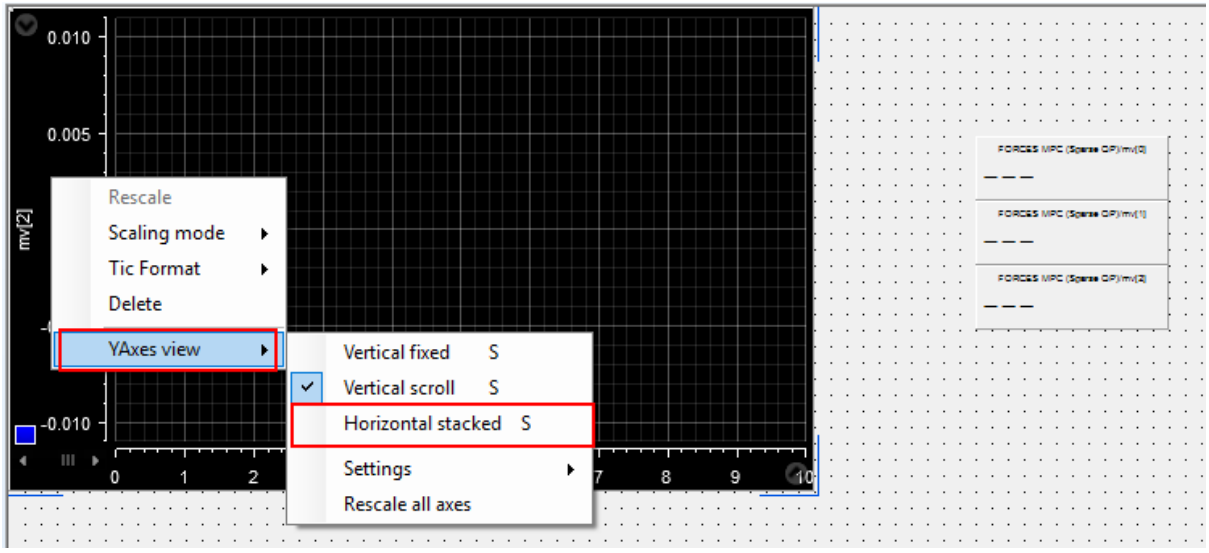


Figure 6.18: Select to show all the signals on the same plot with their own Y-axes

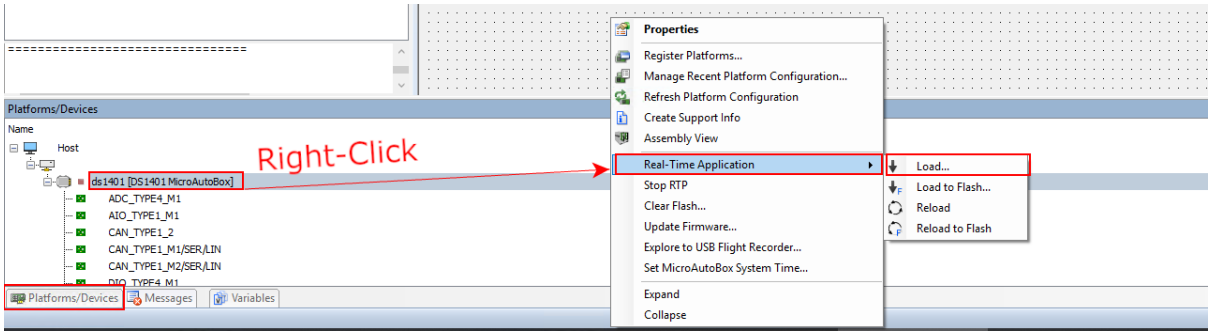


Figure 6.19: Load the application on the dSPACE MicroAutoBox II

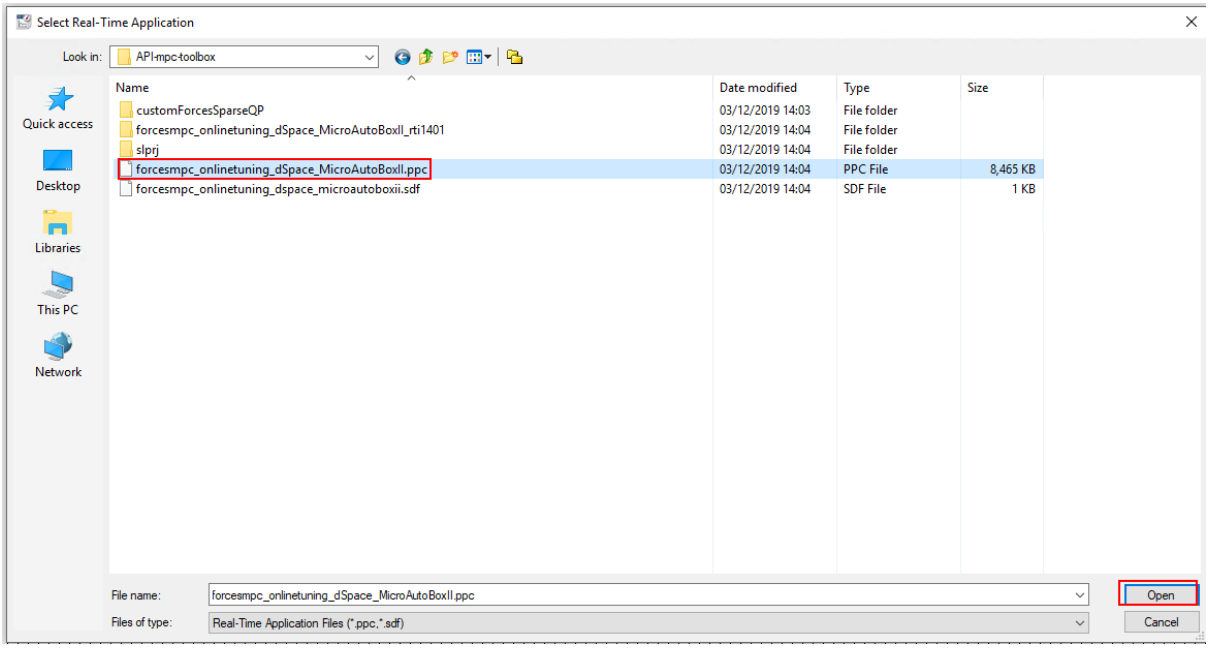


Figure 6.20: Select the executable to run the experiment

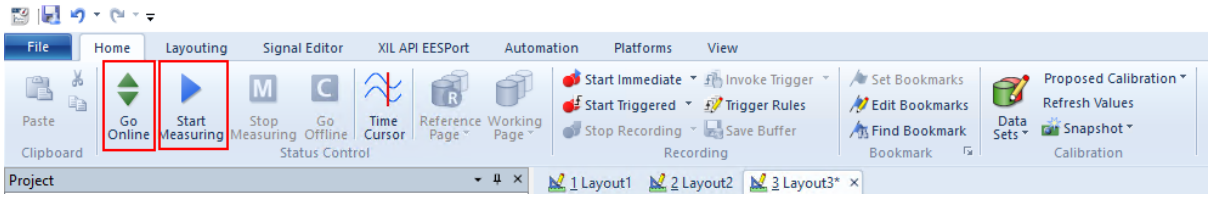


Figure 6.21: Buttons Go Online and Start Measuring to receive execution results

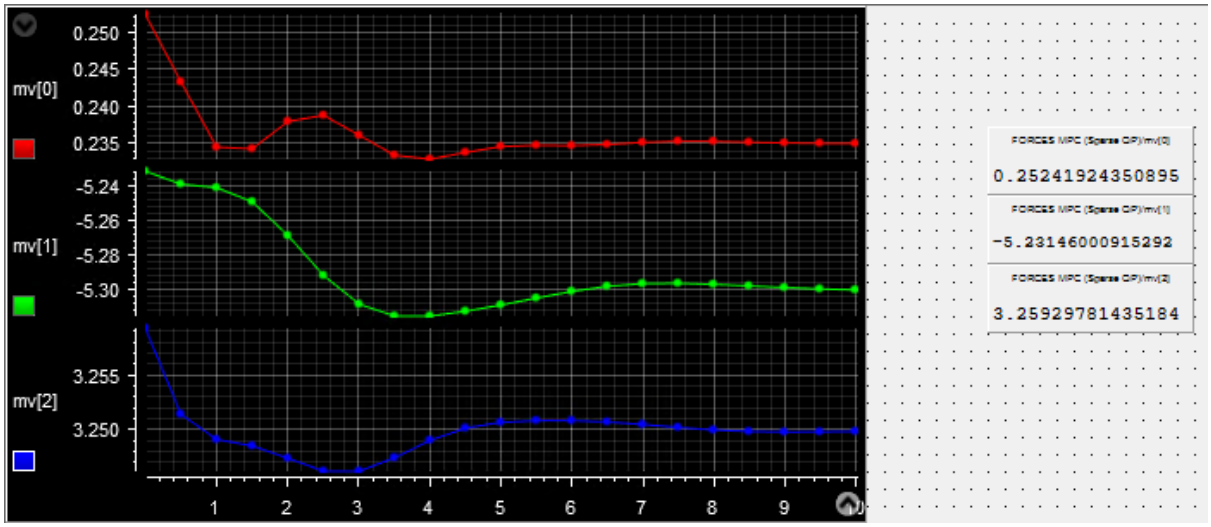


Figure 6.22: Plots and results from experiment on dSPACE MicroAutoBox II

- `forcesmpc_preview.m` is an LTI MPC example with previewing on the output reference and the measured disturbance
- `forcesmpc_motor.m` is an LTI MPC example with state and input constraints
- `forcesmpc_miso.m` is an LTI MPC example with one measured output, one manipulated variable, one measured disturbance, and one unmeasured disturbance
- `forcesmpc_simple_lti.m` demonstrates a simple LTI MPC designed
- `forcesmpc_linearize.m` is an example of linear MPC around an operating point of a non-linear system.
- `forcesmpc_customqp.m` shows how to use the FORCESPRO dense QP solver as a custom solver in an MPC object
- `forcesmpc_onlinetuning.zip` demonstrates how to run the MPC Simulink block.
- `forcesmpc_onlinetuning_dSpace_MicroAutoBoxII.zip` demonstrates how to generate code for dSPACE MicroAutoBox II using the MPC Simulink block.
- `forcesmpc_timevarying.zip` compares the performance of the LTI, adaptive and LTV MPC controllers, when having a time-varying plant. In particular, it demonstrates how to use the plugin for LTI, adaptive and LTV MPC in the MATLAB environment. In case the MATLAB Coder Toolbox is installed, it shows how to use automatically generated MEX functions for faster simulations in the MATLAB environment. In case the Simulink Toolbox is installed, it also showcases how to run a simulation using the corresponding Simulink blocks in the Simulink environment.

6.7.1 Tracking control using a linearized model

The `forcesmpc_linearize.m` example is described in more details below. First, the linearized model and the operating point are loaded from a MAT file.

```
%% Load plant model linearized at its nominal operating point (x0, u0, y0)
load('nomConditionsLinearize.mat');
```

An MPC controller object is then created with a prediction horizon of length $p = 20$, a control horizon $m = 3$ and a sampling period $T_s = 0.1$ seconds as explained here.

```
%% Design MPC Controller
% Create an MPC controller object with a specified sample time |Ts|,
% prediction horizon |p|, and control horizon |m|.
Ts = 0.1;
p = 20;
m = 3;
mpcobj = mpc(plant,Ts,p,m);
```

Nominal values need to be set in the MPC object.

```
% Set the nominal values in the controller.
mpcobj.Model.Nominal = struct('X',x0,'U',u0,'Y',y0);
```

Constraints are set on the manipulated variables and an output reference signal is provided.

```
% Set the manipulated variable constraint.
mpcobj.MV.Max = 0.2;

% Specify the reference value for the output signal.
r0 = 1.5*y0;
```

From the MPC object and a structure of options, a FORCESPRO solver can be generated.

```
% Create options structure for the FORCESPRO sparse QP solver
options = mpcToForcesOptions();
% Generates the FORCESPRO QP solver
[coredata, statedata, onlinedata] = mpcToForces(mpcobj, options);
```

Once a reference signal has been constructed, the simulation can be run using `mpcmoveForces`.

```
for t = 1:Tf
  % A measurement noise is simulated
  Y(:, t) = dPlant.C * (X(:, t) - x0) + dPlant.D * (U(:, t) - u0) + y0 + 0.
↪01 * randn;
  % Prepare inputs of mpcmoveForces
  onlinedata.signals.ref = r(t:min(t+mpcobj.PredictionHorizon-1,Tf),:);
  onlinedata.signals.ym = Y(:, t);
  % Call FORCESPRO solver
  [mv, statedata, info] = mpcmoveForces(coredata, statedata, onlinedata);
  if info.ExitFlag < 0
    warning('Internal problem in FORCESPRO solver');
  end
  U(:, t) = mv;
  X(:, t+1) = dPlant.A * (X(:, t) - x0) + dPlant.B * (U(:, t) - u0) + x0;
end
```

The resulting input and output signals are shown in Figure Figure 6.23 and Figure Figure 6.24 respectively.

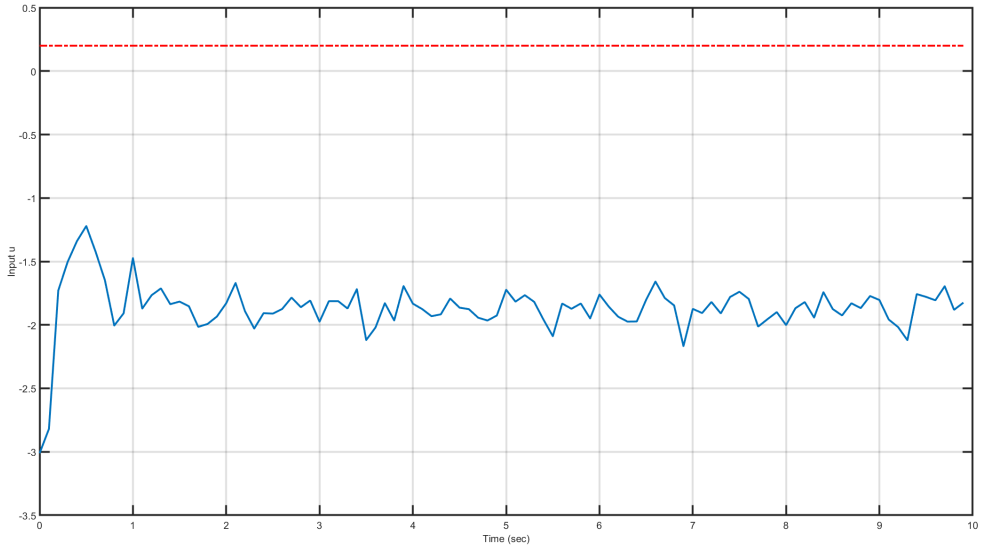


Figure 6.23: Manipulated variable computed by the FORCESPRO plugin.

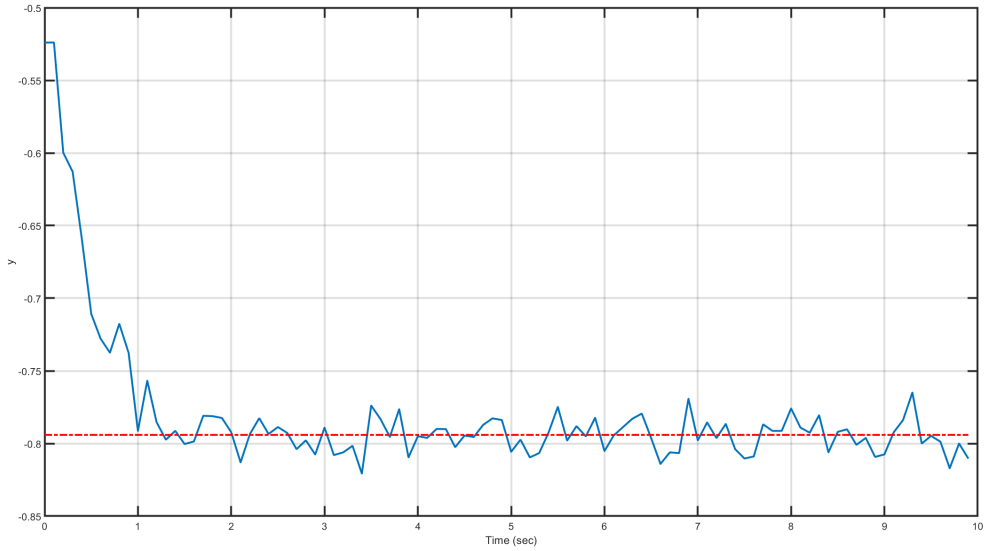


Figure 6.24: Output variable computed by the FORCESPRO plugin.

6.7.2 Tracking control of a time-varying plant using LTI, adaptive and LTV MPC

The **adaptiveMPCExample.zip** example is described in more details below. In particular, it showcases the difference in performance of the different linear MPC Plugin variants, namely LTI, adaptive and LTV, when controlling a time-varying plant.

This example is an adaptation of an example from MathWorks (see here).

The time-varying model is a single-input-single-output 3rd order time-varying linear system with poles, zeros and gain that vary periodically with time.

```
%% Initialization
% Set the simulation duration in seconds.
p = 3;                                  % prediction horizon
m = 3;                                  % control horizon
Ts = 0.1;
Tstop = 25;
Nsim = round(Tstop/Ts) + 1;

%% Time-Varying Plant
% $$G = \frac{{5s + 5 + 2\cos \left( {2.5t} \right)}}{{{s^3}
%   + 3{s^2} + 2s + 6 + \sin \left( {5t} \right)}}$$
%
% The plant poles move between being stable and unstable at run time,
% which leads to a challenging control problem.

% Generate an array of plant models at |t| = |0|, |Ts|, |2*Ts|, ...,
% |Tstop + p*Ts| seconds.
Models = tf;
ct = 1;
for t = 0:Ts:(Tstop + p*Ts)
    Models(:,:,ct) = tf([5 5+2*cos(2.5*t)],[1 3 2 6+sin(5*t)]);
    ct = ct + 1;
end

% Convert the models to state-space format and discretize them with a
% sample time of Ts second.
Models = ss(c2d(Models,Ts));
```

An MPC controller object is created with a prediction and control horizon of $p = m = 3$ and a sampling time $T_s = 0.1s$ as defined previously

```
%% MPC Controller Design
% The control objective is to track a step change in the reference signal.
% First, design an MPC controller for the average plant model. The
% controller sample time is Ts second.
sys = ss(c2d(tf([5 5],[1 3 2 6]),Ts));  % prediction model
mpcobj = mpc(sys,Ts,p,m);

% Set hard constraints on the manipulated variable and specify tuning
% weights.
mpcobj.MV = struct('Min',-2,'Max',2);
mpcobj.Weights = struct('MV',0,'MVRate',0.01,'Output',1);

% Save relevant dimensions
nx = length(mpcobj.Model.Nominal.X);
ny = length(mpcobj.Model.Nominal.Y);
```

(continues on next page)

(continued from previous page)

```
nu = length(mpcobj.Model.Nominal.U);

% Set the initial plant states to zero.
x0 = zeros(nx, 1);
```

With the MPC object and an additional options structure, the FORCESPRO solver for the LTI, adaptive and LTV case can be generated. In case of 'sparse' QP formulation, if the MathWorks Toolbox `MATLAB Coder` is installed, the `mpcToForces` will also generate a MEX version of `mpcmoveForces` and/or `mpcmoveAdaptiveForces` respectively. The generated MEX function will have the name `mpcmove_<options.SolverName>` and/or `mpcmoveAdaptive_<options.SolverName>` respectively, depending on the chosen MPC variant.

```
%% Solver Generation
% LTI
options = mpcToForcesOptions();
options.SolverName = 'LTICustomForcesSparseQP';
[coredataLTI, statedataLTI, onlinedataLTI] = mpcToForces(mpcobj, options);

% Adaptive
options = mpcToForcesOptions();
options.SolverName = 'AdaptiveCustomForcesSparseQP';
options.UseAdaptive = 1;
[coredataAdaptive, statedataAdaptive, onlinedataAdaptive] =␣
↪mpcToForces(mpcobj, options);

% LTV
options = mpcToForcesOptions();
options.SolverName = 'LTVCustomForcesSparseQP';
options.UseLTV = 1;
[coredataLTV, statedataLTV, onlinedataLTV] = mpcToForces(mpcobj, options);
```

We define the step reference changes to be tracked by the LTI, adaptive and LTV MPC controllers. This references array will be used for reference previewing in all three MPC controllers.

```
%% References
stepSwitch = round(Nsim/3);
references = ones(Nsim+p, 1);
references(stepSwitch:2*stepSwitch-1) = 0.6;
references(2*stepSwitch:Nsim+p) = 0.25;
```

For the LTI MPC we need to provide only the measured output and the reference preview over the prediction horizon. If we use the MEX function, we do not need to provide the constant MATLAB structure `coredataLTI`.

```
%% Simulation
coderIsInstalled = mpcchecktoolboxinstalled('matlabcoder');

% LTI
yyMPCFORCES = zeros(ny, Nsim);
uuMPCFORCES = zeros(nu, Nsim);
x = x0;
for ct = 1:Nsim
    % Get the real plant.
    real_plant = Models(:,:,ct);
    % Update and store the plant output.
    y = real_plant.C*x;
    yyMPCFORCES(ct) = y;
```

(continues on next page)

(continued from previous page)

```
    % Compute and store the MPC optimal move.
    onlinedataLTI.signals.ym = y;
    onlinedataLTI.signals.ref = references(ct:ct+p-1);
    if coderIsInstalled
        % MEX function name is mpcmove_<optionsLTI.SolverName>
        % It is not necessary to supply the constant coredataLTI (i.e.
        % it was already considered during code generation)
        [mv, statedataLTI, ~] = mpcmove_
↪LTICustomForcesSparseQP(statedataLTI, onlinedataLTI);
    else
        [mv, statedataLTI, ~] = mpcmoveForces(coredataLTI, statedataLTI,␣
↪onlinedataLTI);
    end
    uuMPCFORCES(ct) = mv;
    % Update the plant state.
    x = real_plant.A*x + real_plant.B*mv;
end
```

For the adaptive MPC we can provide the plant matrices and the nominals at the current timestep in addition to the measured output and the reference preview over the prediction horizon. If we use the MEX function, we do not need to provide the constant MATLAB structure `coredataAdaptive`.

```
% Adaptive
yyAMPCFORCES = zeros(ny, Nsim);
uuAMPCFORCES = zeros(nu, Nsim);
x = x0;
nominal = mpcobj.Model.Nominal; % Nominal conditions are constant over the␣
↪prediction horizon.
for ct = 1:Nsim
    % Get the real plant.
    real_plant = Models(:,:,ct);
    % Update and store the plant output.
    y = real_plant.C*x;
    yyAMPCFORCES(ct) = y;
    % Compute and store the MPC optimal move.
    onlinedataAdaptive.signals.ym = y;
    onlinedataAdaptive.signals.ref = references(ct:ct+p-1);

    onlinedataAdaptive.model.X = nominal.X;
    onlinedataAdaptive.model.U = nominal.U;
    onlinedataAdaptive.model.Y = nominal.Y;
    onlinedataAdaptive.model.DX = nominal.DX;

    onlinedataAdaptive.model.A = real_plant.A;
    onlinedataAdaptive.model.B = real_plant.B;
    onlinedataAdaptive.model.C = real_plant.C;
    onlinedataAdaptive.model.D = real_plant.D;

     if coderIsInstalled
        % MEX function name is mpcmoveAdaptive_<optionsAdaptive.SolverName>
        % It is not necessary to supply the constant coredataAdaptive
        % (i.e. it was already considered during code generation)
        [mv, statedataAdaptive, ~] = mpcmoveAdaptive_
↪AdaptiveCustomForcesSparseQP(statedataAdaptive, onlinedataAdaptive);
    else
```

(continues on next page)

(continued from previous page)

```
        [mv, statedataAdaptive, ~] = mpcmoveAdaptiveForces(coredataAdaptive,
↪ statedataAdaptive, onlinedataAdaptive);
    end
    uuAMPCFORCES(ct) = mv;
    % Update the plant state.
    x = real_plant.A*x + real_plant.B*mv;
end
```

For the LTV MPC we can provide the plant matrices and the nominals over the prediction horizon in addition to the measured output and the reference preview over the prediction horizon. If we use the MEX function, we do not need to provide the constant MATLAB structure `coredataLTV`.

```
% LTV
yyLTVMPCFORCES = zeros(ny, Nsim);
uuLTVMPCFORCES = zeros(nu, Nsim);
x = x0;
Nominals = repmat(mpcobj.Model.Nominal,p+1,1); % Nominal conditions are␣
↪constant over the prediction horizon.
for ct = 1:Nsim
    % Get the real plant.
    real_plant = Models(:,:,ct);
    % Update and store the plant output.
    y = real_plant.C*x;
    yyLTVMPCFORCES(ct) = y;
    % Compute and store the MPC optimal move.
    onlinedataLTV.signals.ym = y;
    onlinedataLTV.signals.ref = references(ct:ct+p-1);

    for k=1:p+1
        onlinedataLTV.model.X(:, 1, k) = Nominals(k).X;
        onlinedataLTV.model.DX(:, 1, k) = Nominals(k).DX;
        onlinedataLTV.model.U(:, 1, k) = Nominals(k).U;
        onlinedataLTV.model.Y(:, 1, k) = Nominals(k).Y;

        model = Models(:, :, ct+k-1);
        onlinedataLTV.model.A(:, :, k) = model.A;
        onlinedataLTV.model.B(:, :, k) = model.B;
        onlinedataLTV.model.C(:, :, k) = model.C;
        onlinedataLTV.model.D(:, :, k) = model.D;
    end

    if coderIsInstalled
        % MEX function name is mpcmoveAdaptive_<optionsLTV.SolverName>
        % It is not necessary to supply the constant coredataLTV (i.e.
        % it was already considered during code generation)
        [mv, statedataLTV, ~] = mpcmoveAdaptive_
↪LTVCustomForcesSparseQP(statedataLTV, onlinedataLTV);
    else
        [mv, statedataLTV, ~] = mpcmoveAdaptiveForces(coredataLTV,␣
↪statedataLTV, onlinedataLTV);
    end
    uuLTVMPCFORCES(ct) = mv;
    % Update the plant state.
    x = real_plant.A*x + real_plant.B*mv;
end
```

For plotting, we use the following code snippet

```
%% Plot
figure(1);
fontSize = 15;
t = 0:Ts:Tstop;

blue = [0, 0.4470, 0.7410];
orange = [0.8500, 0.3250, 0.0980];
yellow = [0.9290, 0.6940, 0.1250];

hold on;
plot(t, yyMPCFORCES, 'Color', blue, 'DisplayName', 'LTI');
plot(t, yyAMPCFORCES, 'Color', orange, 'DisplayName', 'Adaptive');
plot(t, yyLTVMPCFORCES, 'Color', yellow, 'DisplayName', 'LTV');
plot(t, references(1:Nsim), 'k--', 'DisplayName', 'Reference');
hold off;
title('FORCESPRO: Closed-Loop Output', 'FontSize', fontSize);
xlabel('t [s]', 'FontSize', fontSize);
ylabel('y', 'FontSize', fontSize);
xlim([t(1), t(end)]);
grid on;
legend('Location','NorthEast');
```

The resulting control performance of all three MPC variants are shown in Figure Figure 6.25. We can see that the LTI has big oscillatory deviations from the references. Using the adaptive MPC, which changes the model in each timestep while keeping it constant over the prediction horizon, improves performance resulting in smaller deviations from the references. By using the LTV MPC, the user can provide specific models for each of the p+1 stages of the prediction horizon in each timestep, so that the tracking performance becomes almost flawless with minimal deviations only.

In the provided example scripts, we additionally show how to run the closed-loop simulation for LTI, adaptive and LTV MPC controllers in the Simulink environment. The corresponding Simulink model looks like Figure Figure 6.26 and the corresponding results (see Figure Figure 6.27) match with the results from the MATLAB environment (see Figure Figure 6.25).

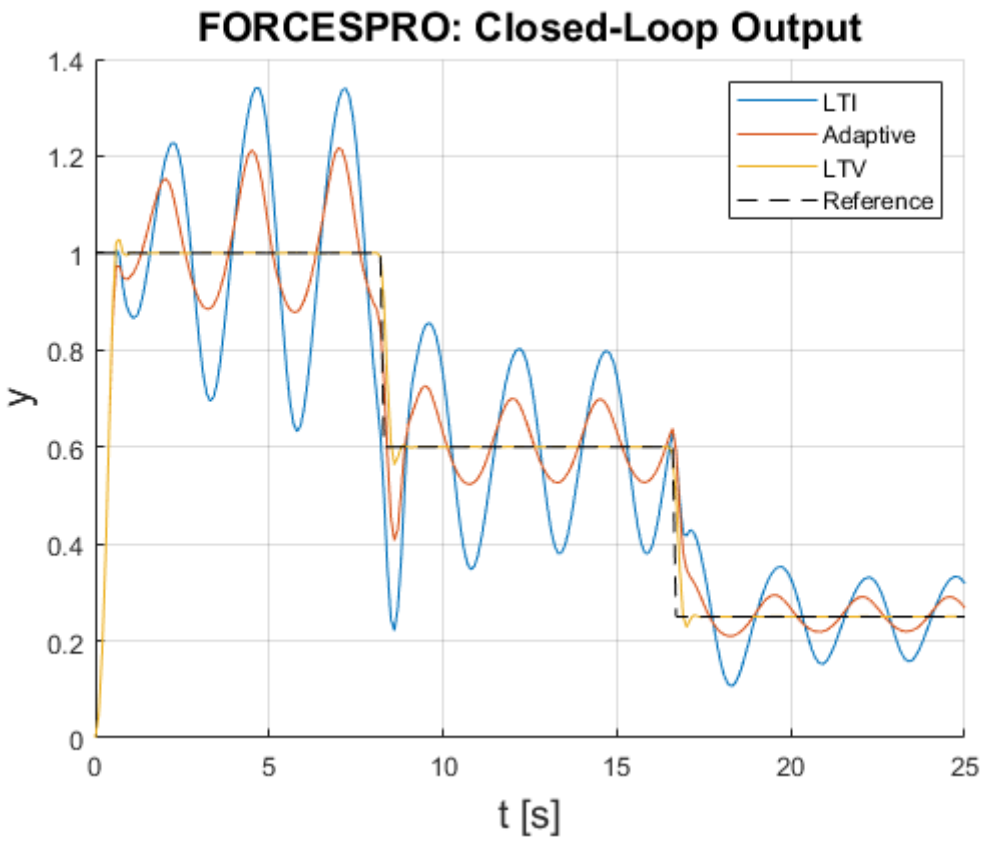


Figure 6.25: Output variable tracking performance for LTI, adaptive and LTV MPC variants.

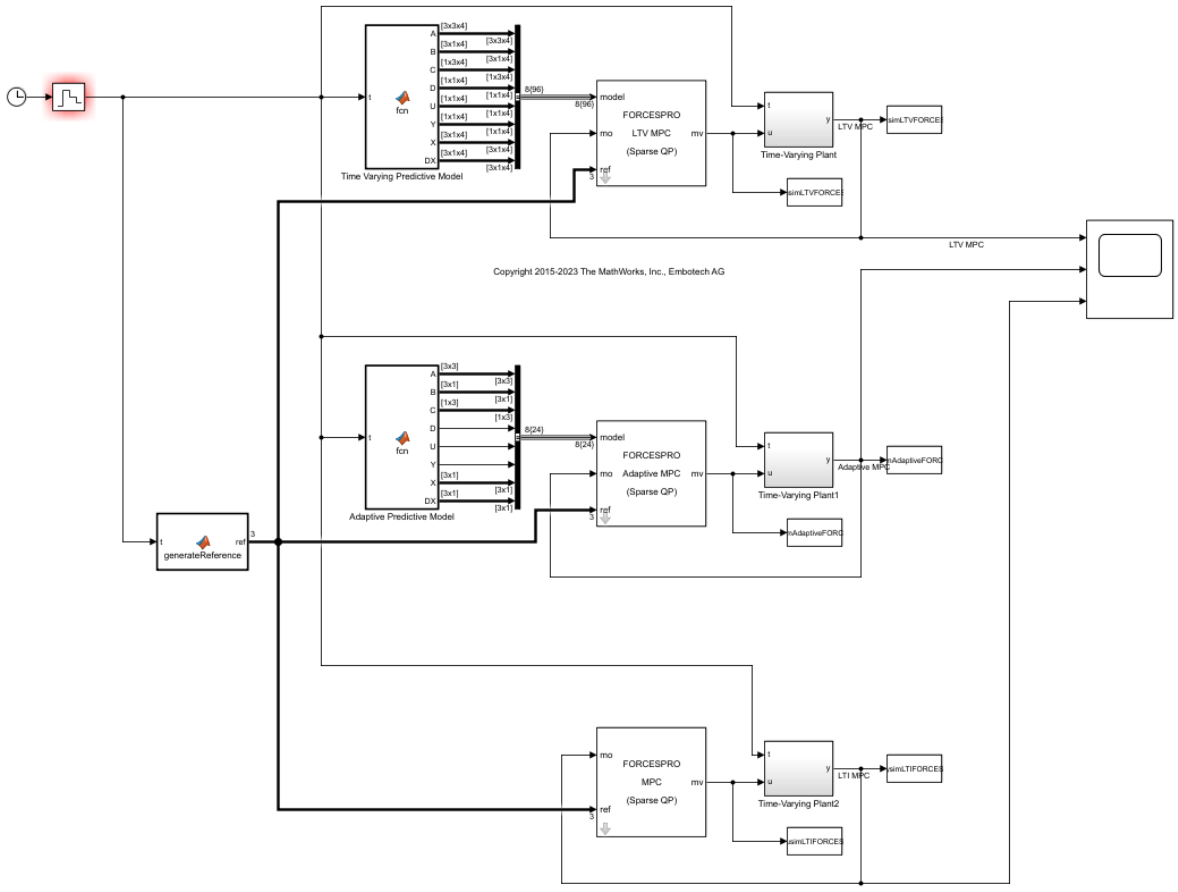


Figure 6.26: Simulink model of closed-loop simulation for a time-varying plant using LTI, adaptive and LTV MPC controllers

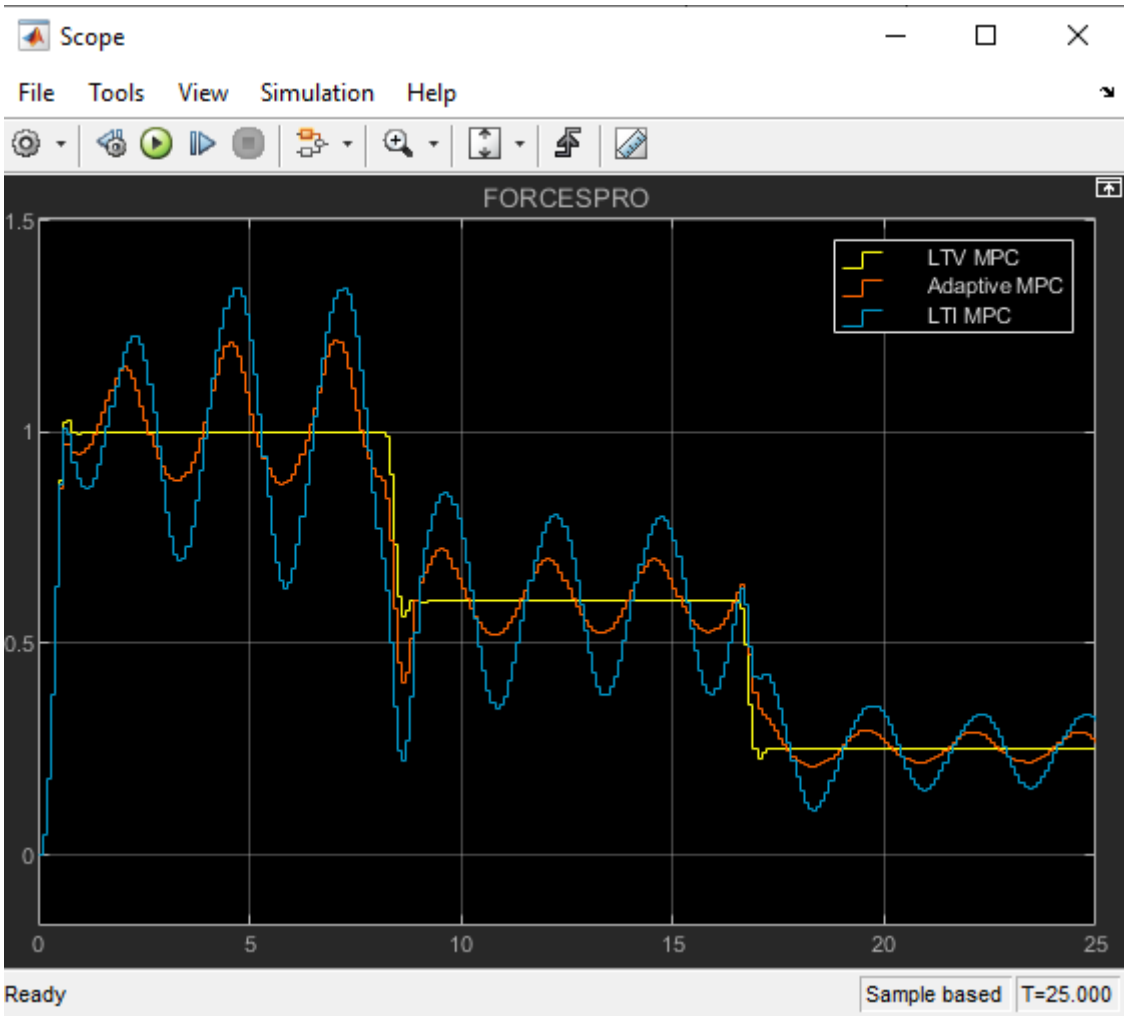


Figure 6.27: Scope plot of the Simulink closed-loop simulation result of Figure Figure 6.26

Chapter 7

MathWorks Nonlinear MPC Plugin

- *Introduction*
- *The SQP Fast algorithm for nlmpc*
- *Defining a nonlinear model*
- *Generating an NLP solver*
  - *Using an "nlmpc" object*
  - *Using an "nlmpcMultistage" object*
- *Simulation in MATLAB and Simulink*
- *Code generation in MATLAB and Simulink*
- *Examples*
  - *Controlling a CSTR reactor*
  - *Lane following example*
  - *Rocket landing example*

7.1 Introduction

As a result of a long-term collaboration, MathWorks Inc. and embotech AG have extended the Model Predictive Control Toolbox™ with a plugin for the FORCESPRO nonlinear solvers. Users are now able to use the FORCESPRO nonlinear interior-point (IP) and sequential quadratic programming (SQP) solvers in MATLAB® and Simulink® from within the MATLAB® Model Predictive Control Toolbox within the nonlinear MPC API. This plugin leverages the powerful design capabilities of the Model Predictive Control Toolbox™ and the computational performance of FORCESPRO. FORCESPRO extends the Model Predictive Control Toolbox with code-generated IP and SQP solvers that are not based on finite-difference derivatives computation, resulting in faster convergence. Thanks to FORCESPRO, the nonlinear API now comes with two classes of nonlinear solvers compatible with code generation that can be deployed to various real-time targets.

Generating a FORCESPRO solver through the Model Predictive Control Toolbox plugin is done by first generating either a *nlmpcMultistage* object (see here) or a *nlmpc* object (see here). The *nlmpcMultistage* formulation of a nonlinear MPC problem offers maximum flexibility and customizability while also ensures optimal performance. Meanwhile, the *nlmpc* formulation of a nonlinear MPC problem is very easy and requires a minimal amount of coding to get started. The different aspects of generating these different objects will be covered

in details below.

Depending on the object chosen, the FORCESPRO nonlinear MPC plugin consists of two API methods:

- `nlmpc`
  - `nlmpcToForces` generates a FORCESPRO nonlinear solver from a nonlinear MPC (`nlmpc`) object designed by the Model Predictive Control Toolbox
  - `nlmpcmoveForces` calls the generated solver to calculate optimal control actions
- `nlmpcMultistage`
  - `nlmpcMultistageToForces` generates a FORCESPRO nonlinear solver from a nonlinear MPC multistage (`nlmpcMultistage`) object designed by the Model Predictive Control Toolbox
  - `nlmpcmoveForcesMultistage` calls the generated solver to calculate optimal control actions

The nonlinear plugin also comes with Simulink® libraries that enable users to run the FORCESPRO solvers from within their Simulink® models. The generation of FORCESPRO nonlinear solvers from *nlmpc* objects is supported from MATLAB R2020a while generation from *nlmpcMultistage* objects is supported from MATLAB R2021a.

This interface is provided with *Variant L* and partially with *Variant M* of FORCESPRO.

Important: Note that when generating a FORCESPRO solver using a `nlmpcMultistage` object there is the following caveat concerning declaration of model functions: For any stage cost, equality and inequality constraint function, if it is defined differently than any other stage, it must be specified in a separate MATLAB function. In other words, do not define different cost and constraint terms in a single function using a switch yard based on stage number. Instead, use different functions, one for each unique definition.

For example, do *not* use a single cost function and assign it to every stage like showed in the following code-snippet:

```
% THIS WILL NOT WORK CORRECTLY!
function cost = LaneFollowingCostFcn(stage,x,u,dmv,para)
Wx = [0; 0; 0.05; 0; 3; 0];
Wdmv = [0.1; 0.2];
ref = [0; 0; para(2); 0; 0; 0];
p = para(1)
if stage==1
cost = (Wdmv.*dmv)'*(Wdmv.*dmv);
elseif stage==(p+1)
  cost = (Wx.*(x-ref))'*(Wx.*(x-ref));
else
  cost = (Wx.*(x-ref))'*(Wx.*(x-ref)) + (Wdmv.*dmv)'*(Wdmv.*dmv);
end
```

Instead, split it into three functions and assign them to 1, 2 to p and $p+1$ respectively.

7.2 The SQP Fast algorithm for nlmpc

From FORCESPRO version 6.0.0, in addition to the previously supported interior point and general SQP algorithms, a new *SQP Fast* algorithm was added. The workflow for the SQP Fast algorithm is slightly different from the workflow required to generate solvers using the other algorithms. The main difference is that generating a SQP Fast solver required a tuning step.

The core idea behind this is that it allows for a highly specialized/tailored SQP algorithm which achieves optimal performance for a given MPC application. For this purpose FORCESPRO ships a caching tool for collecting simulation data in order to tune the SQP Fast solver. The generation of a SQP Fast solver is controlled via the caching tool. The caching tool is a simple mechanism and interacting with it goes via three commands:

- `forcesMpcCache clear`: Deletes all stored caches completely.
- `forcesMpcCache on`: Turns on the cache by constructing a cache object. This command *must* be called before `nlmpcToForces` is called in order to properly cache simulation data. From when the cache is turned on until it is turned off, every call to the generated solver via `nlmpcmoveForces` will store a problem instance in the cache, so it can later be used for tuning the SQP Fast solver.
- `forcesMpcCache off`: Stores the allocated cache in a folder named "FORCESPRO_CACHE" and deletes the cache object from the base workspace.
- `forcesMpcCache reset`: Equivalent to `forcesMpcCache clear; forcesMpcCache on`.

When no cache has been stored (default) a SQP General solver is generated when calling `nlmpcToForces`. If on the other hand a cache is available when calling `nlmpcToForces`, then a SQP Fast solver is generated. Hence, after constructing and specifying an NLMPC object `nlmpcobj`, the complete workflow for generating a SQP Fast solver is as follows:

1. Turn on the FORCESPRO cache by running `forcesMpcCache on` (or `forcesMpcCache reset`) in the MATLAB terminal.
2. Generate a SQP General solver by calling `nlmpcToForces(nlmpcobj,options)` with `options.SolverType = 'SQP'` (see *Using an "nlmpc" object*).
3. Run a simulation, using `nlmpcmoveForces` to compute optimal control moves. It is important that `nlmpcmoveForces` is used as opposed to the generated MEX file, which is not able to cache the problem data for tuning the fast SQP solver.
4. Turn off the cache by running `forcesMpcCache off` in the MATLAB terminal.
5. Generate a QP fast solver by calling `nlmpcToForces(nlmpcobj,options)` (see *Using an "nlmpc" object*). Since the cache has been stored in step 4, a SQP Fast solver will be generated and the automatic tuning procedure will be triggered.

See an example `here` where the full workflow is described in full detail.

Important: If the MATLAB command `clear` is called in between the calls `forcesMpcCache on` and `forcesMpcCache off` the cache is deleted. Hence, this should be avoided.

Important: The SQP Fast solver is currently only supported when generating a solver using a `nlmpc` object (see *Using an "nlmpc" object*).

7.3 Defining a nonlinear model

In order to call the FORCESPRO code generation, both a `nlmpc` object as well as a `nlmpcMultistage` object need to be built from a *Model*. The process is essentially the same as the one described here. However one should note that the FORCESPRO code generation ignores the jacobian functions that may be provided in *Jacobian.StateFcn* and *Jacobian.OutputFcn*, since these will be automatically generated by the automatic differentiation tool CasADi. Moreover, the following requirements on the fields *Model.StateFcn* and *Model.OutputFcn* need to be fulfilled for the plugin to work seamlessly:

- they must be the name of a function file, not an anonymous functions

- they must be compatible with MATLAB code generation
- they must follow CasADi coding conventions. Most importantly, the state derivative *dxdt* has to be built explicitly, as shown below.

```
dxdt = [expression; expression; ...]
```

As a word of caution, the following code snippet will result in an undesired behavior from CasADI.

```
dxdt = x;            % Do not write this, CasADI takes it as reference !
dxdt(1,1) = a1*x(1) + a2*x(2) + b1*u(2);
dxdt(2,1) = a3*x(1) + a4*x(2) + b2*u(2);
dxdt(3,1) = x(2)*x(1) + x(4);
dxdt(4,1) = (1/tau)*(-x(4) + u(1));
dxdt(5,1) = x(1) + x(3)*x(6);
dxdt(6,1) = x(2) - 0*x(3);
```

FORCESPRO calls the model functions from its own objects, which follow an assignment by reference convention, hence the assignment *dxdt = x* is made by reference. This implies that updating *dxdt* also changes *x*, which builds the wrong symbolic dynamics.

For `nlmpc` objects, if the model contains a parameter, it must be a single vector parameter. In other words, users need to set *nlobj.Model.NumberOfParameters = 1* and at run-time write *onlinedata.Parameter = value* where value is a column vector.

7.4 Generating an NLP solver

7.4.1 Using an "nlmpc" object

When generating a FORCESPRO solver using an `nlmpc` object, the main difference compared to the existing nonlinear MPC from The MathWorks based on the `fmincon` solver from the Optimization Toolbox is a code generation step that takes the nonlinear MPC object as argument. This is needed in order to build a mex interface for a FORCESPRO nonlinear solver that is customized to the model provided by the user.

Given an NLMPC object created by the *nlmpc* command, users can generate an IP or SQP nonlinear solver tailored to their specific problem via the following command:

```
% nlobj is the output of nlmpc(...)
% options is the output of nlmpcToForcesOptions(...)

[coredata, onlinedata] = nlmpcToForces(nlobj, options);
```

Two types of nonlinear solvers can be generated via `nlmpcToForces`: a nonlinear interior-point solver and a sequential quadratic programming solver whose features are covered in details in *Sequential quadratic programming algorithm*.

The *nlmpcToForces* API is described in more details in the tables below. The `nlmpcToForces` command expects an NLMPC object **nlobj** and a structure **options** as arguments. It also has a few limitations as it currently does not support custom cost and constraints. Instead one should in this case use an `nlmpcMultistage` object to represent custom cost and constraints. It also requires double precision.

Table 7.1: *nlmpcToForces* arguments

Input	Description
nlobj	NMPC object constructed by Model Predictive Control Toolbox (see here)
options	Object that provides solver generation options.

The outputs of `nlmpcToForces` consist of two structures **coredata**, a structure containing the constant NLMPC information used by *nlmpcmoveForces* and **onlinedata**, a structure that allows you to specify online signals such as *x*, *lastMV*, *ref*, *MVTarget*, *md* as well as weights or bounds used by *nlmpcmoveForces*.

In order to provide the solver options to `nlmpcToForces`, the user needs to run the command `nlmpcToForcesOptions`. The options structure contains the following fields:

- *SolverName*. This is the solver name used by MEX and C files. Its default value is `myForcesNLPSolver`.
- *SolverType*. This option specifies which FORCESPRO nonlinear programming solver to use. Its default value is `InteriorPoint`. To use the FORCESPRO SQP algorithm set the value to `SQP`.
- *SkipSolverGeneration*. This option indicates whether `nlmpcToForces` should generate the custom NLP solver. When true, `nlmpcToForces` will return structures without regenerating the MEX and C files. Its default value is *false*.
- *Server*. This option specifies the FORCESPRO server address for remote solver generation. Its default value is https://forces.embotech.com.
- *PrintLevel*. This option specifies the amount of information displayed in the solver log.
- *ForcesTargetPlatform* for choosing a target platform to deploy the solver. Currently, dSPACE, Speedgoat, BeagleBone-Blue and AURIX are supported.
  - 0: no output will be written
  - 1: summary line of each solve
  - 2: summary line of each iteration

  Its default value is 0.
- *IntegrationMethod*. This option specifies the choice of integration scheme. I.e. the way in which the continuous dynamics are discretized. This field is only available from MATLAB R2021a onwards. The different integration schemes are
  - `"IRK2"`: Implicit Runge Kutta method of order 2
  - `"RK4"`: Explicit Runge Kutta method of order 4

  Its default value is `"IRK2"`.
- *IntegrationNodes*. This option specifies the the number of intermediate points between t and $t + Ts$ during numerical integration of a continuous time model. Use larger values when the plant is stiff at the price of computational efficiency. Its default value is 1. The approach used here is referred to as *direct multiple shooting*.
- *x0*. This option is used to create initial guess of optimal state trajectory at cold start. It must be a column vector of nx-by-1. The typical value should be the initial state of the prediction model. If it is left empty, zeros will be used for cold start. Its default value is [].
- *mv0*. This option is used to create initial guess of optimal manipulated variable trajectory at cold start. It must be a column vector of nmv-by-1. The typical value should be the last known control action. If it is left empty, zeros will be used for cold start.
- *Parameter*. This option should be specified if the prediction model has a parameter. It must be a column-vector and it can be updated at run-time.
- *UseMVTarget*. This option enables/disables MV reference signal. When equal to *true*, the MV reference signal is provided via the `onlinedata` structure. When equal to *false*, the MV reference is 0 by default. In this case, MV weights should be zero to avoid unexpected behavior. Default value is false.
- *UseOnlineWeightOV*. This option enables/disables online OV weight change. When equal to *true*, OV weight needs to be provided via `onlinedata` structure. Its default value is *false*.

- *UseOnlineWeightMV*. This option enables/disables online MV weight change. When equal to *true*, MV weight needs to be provided via `onlinedata` structure. Its default value is *false*.
- *UseOnlineWeightMVRate*. This option enables/disables online MVRate weight change. When equal to *true*, MVRate weight needs to be provided via `onlinedata` structure. Its default value is *false*.
- *UseOnlineWeightECR*. This field enables/disables online ECR weight change. When equal to *true*, ECR weight needs to be provided via `onlinedata` structure. Its default value is *false*.
- *UseOnlineConstraintStateMax*. This option enables/disables online state upper bound change. When equal to *true*, state upper bound needs to be provided via `onlinedata` structure. Its default value is *false*.
- *UseOnlineConstraintStateMin*. This field enables/disables online state lower bound change. When equal to *true*, state lower bound needs to be provided via `onlinedata` structure. Its default value is *false*.
- *UseOnlineConstraintOVMax*. This field enables/disables online OV upper bound change. When equal to *true*, OV upper bound needs to be provided via the `onlinedata` structure. Its default value is *false*.
- *UseOnlineConstraintOVMin*. This option enables/disables online OV lower bound change. When equal to *true*, OV lower bound needs to be provided via the `onlinedata` structure. Its default value is *false*.
- *UseOnlineConstraintMVMax*. This field enables/disables online MV upper bound change. When equal to *true*, MV upper bound needs to be provided via the `onlinedata` structure. Its default value is *false*.
- *UseOnlineConstraintMVMin*. This field enables/disables online MV lower bound change. When equal to *true*, MV lower bound needs to be provided via the `onlinedata` structure. Its default value is *false*.
- *UseOnlineConstraintMVRateMax*. This option enables/disables online MVRate upper bound change. When equal to *true*, MVRate upper bound needs to be provided via the `onlinedata` structure. Its default value is *false*.
- *UseOnlineConstraintMVRateMin*. This option enables/disables online MVRate lower bound change. When equal to *true*, MVRate lower bound needs to be provided via the `onlinedata` structure. Its default value is *false*.

The following set of options are specific to the nonlinear interior point solver:

- *IP_MaxIteration*. This field specifies the maximum number of iterations in the interior point solver. When the maximum number of iterations is reached (i.e. ExitFlag is 0), the NLP solver aborts calculations and the result should be discarded. Default value is 200.
- *IP_Mu0*. This field specifies initial barrier parameter. It must be positive and its default value is 0.1.
- *IP_BarrierStrategy*. This option specifies the strategy used to update the barrier parameter at every iteration of the nonlinear interior point solver. It needs to be either *monotone* or *loqo*. *loqo* often leads to faster convergence, while *monotone* may help convergence for difficult problems. Default value is *loqo*.
- *IP_LinearSolver*. This option sets the linear solver. It must be either *normal_eqs*, *symm_indefinite*, *symm_indefinite_fast* or *symm_indefinite_legacy*. With *normal_eqs*, the KKT system is solved in normal equations form. With *symm_indefinite*, the KKT system is solved with an improved variant of `'symm_indefinite_legacy'` introduced in FORCESPRO version 5.0.0. With *symm_indefinite_legacy*, the KKT system is solved using block-indefinite factorizations. With *symm_indefinite_fast*, the KKT system is solved in symmetric indefinite form, using regularization and positive definite Cholesky factorizations only. Default value is *normal_eqs*.

- *IP_EqualityTolerance*. This option specifies the tolerance on the nonlinear equality constraints used by the nonlinear interior point solver. It must be positive. Default value is 10^{-6}.
- *IP_InequalityTolerance*. This field specifies the tolerance on the nonlinear inequality constraints used by the interior-point solver. It needs to be positive and its default value is 10^{-6}.
- *IP_StationarityTolerance*. This option specifies the tolerance on the stationarity measure used in the nonlinear interior point solver. It needs to be positive and its default value is 10^{-5}.

The following set of options are specific to the sequential quadratic programming solver:

- *SQP_MaxIteration*. This field specifies the maximum number of iterations used by the inner QP solver. Its default value is 50.
- *SQP_MaxQPS*. This enables the SQP solver to solve a fixed amount of quadratic approximations at every call to the solver. In general, the more quadratic approximations are solved, the more accurate control performance is achieved. The tradeoff is that the solvetime also increases. The default value is 1.
- *SQP_RegHessian*. This field stands for the level of regularization of the hessian approximation. Increasing this parameter may help if the SQP solver returns exitflag -8 on your problem. The default value is $5 \cdot 10^{-9}$.
- *SQP_EqualityTolerance*. This option specifies the tolerance on the nonlinear equality constraints. It must be positive and its default value is 10^{-6}.
- *SQP_InequalityTolerance*. This option specifies the tolerance on the linear inequality constraints. It must be positive and its default value is 10^{-6}.
- *SQP_StationarityTolerance*. This field specifies the tolerance on stationarity. It must be positive and its default value is 10^{-5}.

7.4.2 Using an "nlmpcMultistage" object

When generating a FORCESPRO solver using an `nlmpcMultistage` object, the main difference compared to the existing nonlinear MPC from The MathWorks based on the `fmincon` solver from the Optimization Toolbox is a code generation step that takes the nonlinear MPC object as argument. This is needed in order to build a mex interface for a FORCESPRO nonlinear solver that is customized to the model provided by the user.

Given an `nlmpcMultistage` object, users can generate an IP nonlinear solver tailored to their specific problem via the following command:

```
% nlmul is the output of nlmpcMultistage(...)
% options is the output of nlmpcMultistageToForcesOptions(...)

[coredata, onlinedata] = nlmpcMultistageToForces(nlmul, options);
```

The `nlmpcMultistageToForces` API allows the user to customize the generated solver to a much higher extend than that of the `nlmpc` object. In particular it supports a different cost function associated with each stage, with the restriction that each cost function can only depend on the optimization variables of a single stage.

The `nlmpcMultistageToForces` API is described in more details in the tables below. The `nlmpcMultistageToForces` command expects an `nlmpcMultistage` object **nlmul** and a structure **options** as arguments.

Table 7.2: *nlmpcMultistageToForces* arguments

Input	Description
nlmul	`nlmpcMultistage` object constructed by Model Predictive Control Toolbox (see here)
options	Object that provides solver generation options.

The outputs of `nlmpcMultistageToForces` consist of two structures **coredata**, a structure containing the constant NLMPCMultistage information used by *nlmpcmoveForcesMultistage* and **onlinedata**, a structure that allows you to specify online signals such as *x*, *lastMV*, *ref*, *MVTarget*, *md* as well as weights or bounds used by *nlmpcmoveForces*.

In order to provide the solver options to `nlmpcmoveForcesMultistage`, the user needs to run the command `nlmpcMultistageToForcesOptions`. The options structure contains the following fields:

- *SolverName*. This is the solver name used by MEX and C files. Its default value is `myForcesNLPSolver`.
- *SolverType*. This option specifies which FORCESPRO nonlinear programming solver to use. Currently the only options is `InteriorPoint`.
- *SkipSolverGeneration*. This option indicates whether `nlmpcToForces` should generate the custom NLP solver. When true, `nlmpcMultistageToForces` will return structures without regenerating the MEX and C files. Its default value is *false*.
- *Server*. This option specifies the FORCESPRO server address for remote solver generation. Its default value is https://forces.embotech.com.
- *PrintLevel*. This option specifies the amount of information displayed in the solver log.
- *ForcesTargetPlatform* for choosing a target platform to deploy the solver. Currently, dSPACE, Speedgoat, BeagleBone-Blue and AURIX are supported.
  - 0: no output will be written
  - 1: summary line of each solve
  - 2: summary line of each iteration

  Its default value is 0.
- *IntegrationMethod*. This option specifies the choice of integration scheme. I.e. the way in which the continuous dynamics are discretized. This field is only available from MATLAB R2021a onwards. The different integration schemes are
  - `"IRK2"`: Implicit Runge Kutta method of order 2
  - `"RK4"`: Explicit Runge Kutta method of order 4
- *IntegrationNodes*. This option specifies the the number of intermediate points between t and $t + Ts$ during numerical integration of a continuous time model. Use larger values when the plant is stiff at the price of computational efficiency. Its default value is 1. The approach used here is referred to as *direct multiple shooting*.
- *x0*. This option is used to create initial guess of optimal state trajectory at cold start. It must be a column vector of nx-by-1. The typical value should be the initial state of the prediction model. If it is left empty, zeros will be used for cold start. Its default value is [].
- *mv0*. This option is used to create initial guess of optimal manipulated variable trajectory at cold start. It must be a column vector of nmv-by-1. The typical value should be the last known control action. If it is left empty, zeros will be used for cold start.
- *NumInequalityConstraints*. Must be a $(p + 1)$-by-1 vector where each entry specifies the number of inequality constraints generated by the `IneqConFcn` at that stage. Leave it `[]` if no `IneqConFcn` id defined in the `nlmpcMultistage` object.

- *NumEqualityConstraints*. Must be a p-by-1 vector where each entry specifies the number of equality constraints generated by the `EqConFcn` at that stage. Leave it `[]` if no `EqConFcn` is defined in the `nlmpcMultistage` object.
- *UseOnlineConstraintStateMax*. This option enables/disables online state upper bound change. When equal to *true*, state upper bound needs to be provided via `onlinedata` structure. Its default value is *false*.
- *UseOnlineConstraintStateMin*. This field enables/disables online state lower bound change. When equal to *true*, state lower bound needs to be provided via `onlinedata` structure. Its default value is *false*.
- *UseOnlineConstraintMVMax*. This field enables/disables online MV upper bound change. When equal to *true*, MV upper bound needs to be provided via the `onlinedata` structure. Its default value is *false*.
- *UseOnlineConstraintMVMin*. This field enables/disables online MV lower bound change. When equal to *true*, MV lower bound needs to be provided via the `onlinedata` structure. Its default value is *false*.
- *UseOnlineConstraintMVRateMax*. This option enables/disables online MVRate upper bound change. When equal to *true*, MVRate upper bound needs to be provided via the `onlinedata` structure. Its default value is *false*.
- *UseOnlineConstraintMVRateMin*. This option enables/disables online MVRate lower bound change. When equal to *true*, MVRate lower bound needs to be provided via the `onlinedata` structure. Its default value is *false*.
- *UseOnlineTerminalState*. This option enables/disables online terminal state condition. When equal to *true*, terminal state values need to be provided via the `onlinedata` structure. Its default value is *false*.

The following set of options are specific to the nonlinear interior point solver:

- *IP_MaxIteration*. This field specifies the maximum number of iterations in the interior point solver. When the maximum number of iterations is reached (i.e. ExitFlag is 0), the NLP solver aborts calculations and the result should be discarded. Default value is 200.
- *IP_Mu0*. This field specifies initial barrier parameter. It must be positive and its default value is 0.1.
- *IP_BarrierStrategy*. This option specifies the strategy used to update the barrier parameter at every iteration of the nonlinear interior point solver. It needs to be either *monotone* or *loqo*. *logo* often leads to faster convergence, while *monotone* may help convergence for difficult problems. Default value is *loqo*.
- *IP_LinearSolver*. This option sets the linear solver. It must be either *normal_eqs*, *symm_indefinite*, *symm_indefinite_fast* or *symm_indefinite_legacy*. With *normal_eqs*, the KKT system is solved in normal equations form. With *symm_indefinite*, the KKT system is solved with an improved variant of `'symm_indefinite_legacy'` introduced in FORCESPRO version 5.0.0. With *symm_indefinite_legacy*, the KKT system is solved using block-indefinite factorizations. With *symm_indefinite_fast*, the KKT system is solved in symmetric indefinite form, using regularization and positive definite Cholesky factorizations only. Default value is *normal_eqs*.
- *IP_EqualityTolerance*. This option specifies the tolerance on the nonlinear equality constraints used by the nonlinear interior point solver. It must be positive. Default value is 10^{-6}.
- *IP_InequalityTolerance*. This field specifies the tolerance on the nonlinear inequality constraints used by the interior-point solver. It needs to be positive and its default value is 10^{-6}.
- *IP_StationarityTolerance*. This option specifies the tolerance on the stationarity measure used in the nonlinear interior point solver. It needs to be positive and its default value is 10^{-5}.

7.5 Simulation in MATLAB and Simulink

Once a FORCESPRO nonlinear solver has been generated by calling either `nlmpcToForces` or `nlmpcMultistageToForces`, optimal control moves can be calculated in MATLAB by using either `nlmpcmoveForces` or `nlmpcmoveForcesMultistage` depending on the case. This API method expects a **coredata** structure as returned by `nlmpcToForces` or `nlmpcMultistageToForces` as well as the other inputs described in Table below.

Table 7.3: *nlmpcmoveForces* and *nlmpcMultistageToForces* arguments

Input	Description
coredata	A structure containing the constant controller settings. It is generated by the `nlmpcToForces` method and used as a constant
x	A n_x-by-1 column vector, representing the current prediction model states
lastMV	A nmv-by-1 column vector, representing the control action applied to the plant at the previous control interval
onlinedata	A structure containing run time signals

The outputs of `nlmpcmoveForces` and `nlmpcMultistageToForces` are described in the table below.

Table 7.4: *nlmpcmoveForces* and *nlmpcMultistageToForces* outputs

Output	Description
mv	Optimal control moves computed by a FORCESPRO solver
onlinedata	A structure prepared for the next control, containing e.g. the initial guess.
info	A structure containing extra information about the solver run

7.6 Code generation in MATLAB and Simulink

The *nlmpcmoveForces* and *nlmpcmoveForcesMultistage* commands can be turned into a MEX interface named *nlmpcmove_<solvername>* by means of the *SkipSolverGeneration*. If the option is set to *true*, then no MEX interface is built by the MATLAB Coder. If it is set to *false*, then the *nlmpcmove* MEX interface is generated and compiled, which requires the MATLAB Coder.

7.7 Examples

Here we present the following examples to illustrate the workflow of the FORCESPRO plugin for the MPC Toolbox:

- Example *Controlling a CSTR reactor* illustrates how to generate a FORCESPRO solver from an NLMPC object directly in MATLAB®.
- Example *Lane following example* illustrates how to generate a FORCESPRO solver from an NLMPC object and run it based on the nlmpc Simulink block.
- Example *Rocket landing example* illustrates how to generate a FORCESPRO solver from an NLMPCMultistage object.

7.7.1 Controlling a CSTR reactor

In this example we create a nonlinear MPC controller for a CSTR reactor using the MathWorks Nonlinear MPC Plugin. The objective is to control the concentration CA of reagent A.

You can find the code of this example to try it out for yourself in the `examples/Matlab/mpc-toolbox-plugin/nonlinearModels/nlmpc_cstr` folder that comes with your FORCESPRO client.

Click here for a detailed description of the model. The state of our plant will be denoted by x, while our control input will be denoted by u.

x_1 : Reactor temperature (K)

x_2 : Concentration of A in reactor tank $\left(\frac{kgmol}{m^3}\right)$

u_1 : Jacket coolant temperature (K)

u_2 : Concentration of A in inlet feed stream $\left(\frac{kgmol}{m^3}\right)$

u_3 : Inlet feed stream temperature (K)

The system dynamics are given by the following first order differential equation

$$\dot{x_1} = (u_3 - x_1) + 0.3 \cdot (u_1 - x_1) + 11.92 \cdot 27944640 \cdot \exp(\tfrac{-5894.14}{x_1}) \cdot u_2$$
$$\dot{x_2} = (u_2 - x_2) - 27944640 \cdot \exp(\tfrac{-5894.14}{x_1}) \cdot u_2$$

For the purpose of this demonstration the MATLAB function describing the state dynamics will be denoted by *exocstrStateFcnCT*. Our output y is simply given by the concentration of A:

$$y = x_2$$

Creating an NLMPC object

The MATLAB function implementing this output will be denoted by *exocstrOutputFcn*. With the implemented *exocstrStateFcnCT* and *exocstrOutputFcn* MATLAB functions at hand we can go ahead create our NLMPC object. The following code-snippet constructs the NLMPC object and specifies our model.

```
nx = 2;
ny = 1;
nu = 3;
nlobj = nlmpc(nx,ny,'MV',1,'MD',[2 3]);
Ts = 0.5;
nlobj.Ts = Ts;
nlobj.PredictionHorizon = 6;
nlobj.ControlHorizon = [2 2 2];
nlobj.MV.RateMin = -5;
nlobj.MV.RateMax = 5;
nlobj.Model.StateFcn = 'exocstrStateFcnCT';
nlobj.Model.OutputFcn = 'exocstrOutputFcn';
```

Specifying solver options

The following specifies the code options specific to FORCESPRO's MathWorks Nonlinear MPC Plugin:

```
options = nlmpcToForcesOptions();
options.SolverName = 'CstrSolver';
options.SolverType = 'SQP';
options.IntegrationNodes = 5;
options.SQP_MaxQPS = 5;
options.SQP_MaxIteration = 500;
options.x0 = [311.2639; 8.5698];
options.mv0 = 298.15;
```

Generating the NLP solver

Once we have our NLMPC object and our options we can generate an NLP solver through the *nlmpcToForces* function:

```
[coredata, onlinedata] = nlmpcToForces(nlobj,options);
```

Calling the solver

This will generate our NLP solver named *CstrSolver*. We can call this solver in two different ways:

- Through the generic *nlmpcmoveForces* function which comes with the FORCESPRO MathWorks Nonlinear MPC Plugin
- Or through the generated MEX function *nlmpcmove_CstrSolver* (the name of the MEX is always "nlmpc_<solvername>"). In general it is advantageous from a performance perspective to use the MEX over the generic *nlmpcmoveForces* function.

Calling the NLP solver through the generic *nlmpcmoveToForces* can be done as in the following code-snippet:

```
onlinedata.md = [10 298.15];
[mv, onlinedata, info] = nlmpcmoveForces(coredata,x,mv,onlinedata);
```

And the MEX can be called as follows:

```
[mv, onlinedata, info] = nlmpcmove_CstrSolver(x,mv,onlinedata);
```

Results

The NLP solver generated through the above code-snippets were applied in a simulation for 200 seconds. As can be seen in the plots Figure 7.1, Figure 7.2 and Figure 7.3 the generated solver succeeds in controlling the CSTR reactor with a very fast solvetime while the output stays close to the reference.

7.7.2 Lane following example

In this example, the use of the nlmpc plugin in Simulink is described. The example consists in making a vehicle follow a central line while keeping a user-specified velocity.

You can find the code of this example to try it out for yourself in the `examples/Matlab/mpc-toolbox-plugin/nonlinearModels/lane_following` folder that comes with your FORCESPRO client.

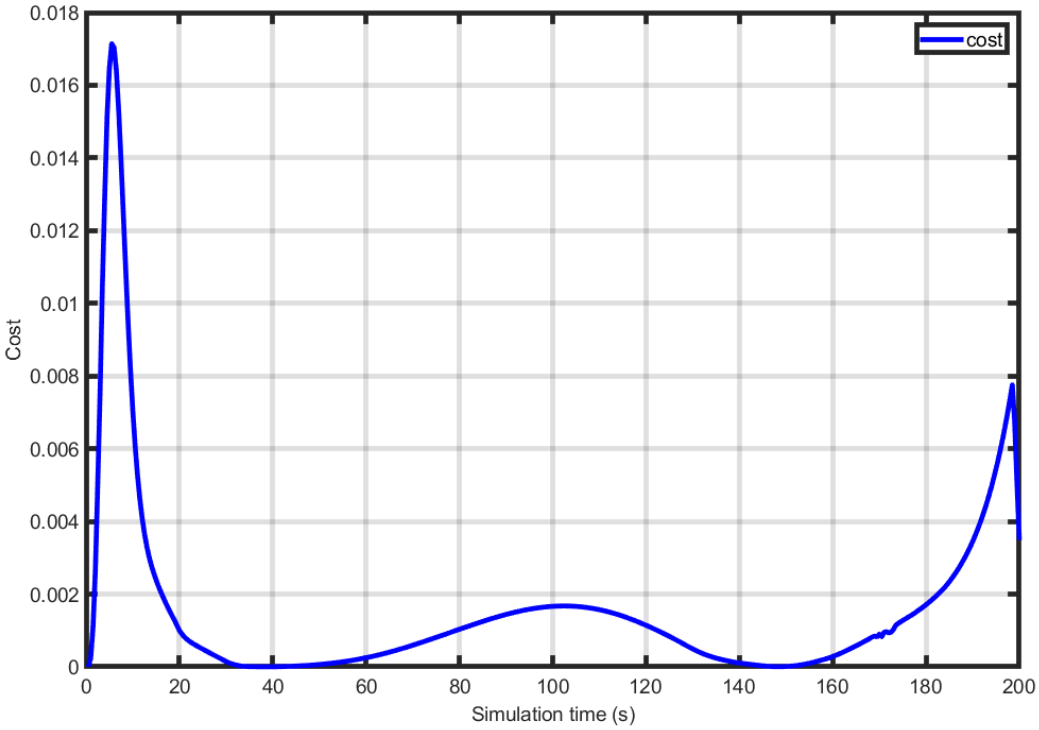


Figure 7.1: Cost as a function of time.

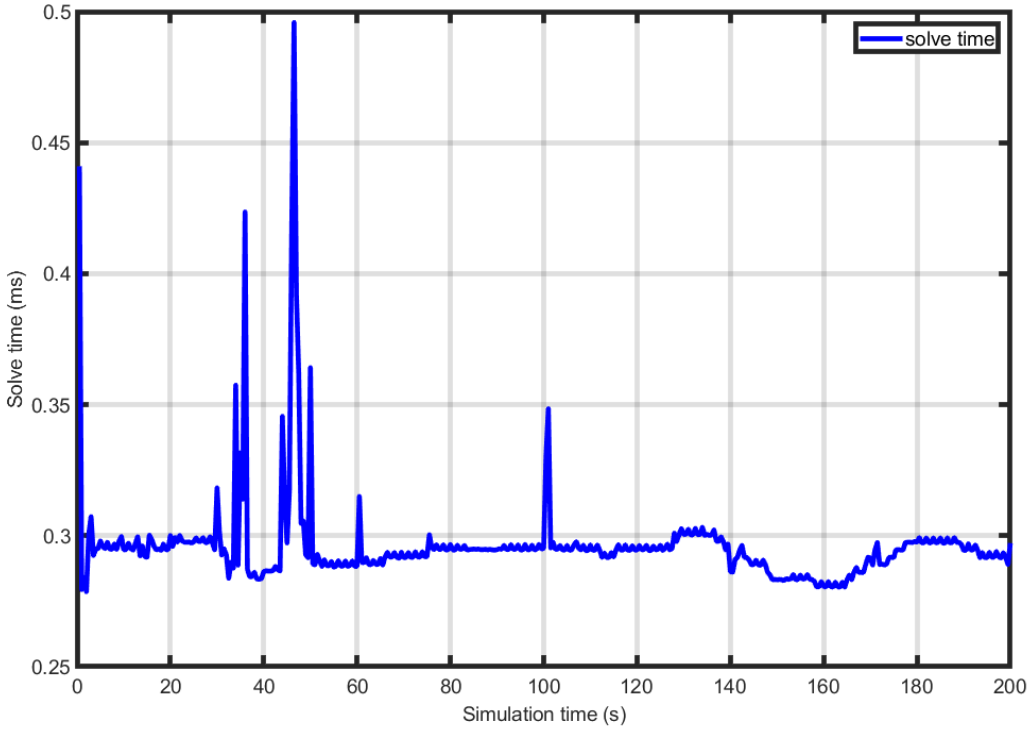


Figure 7.2: Solve time as a function of simulation time.

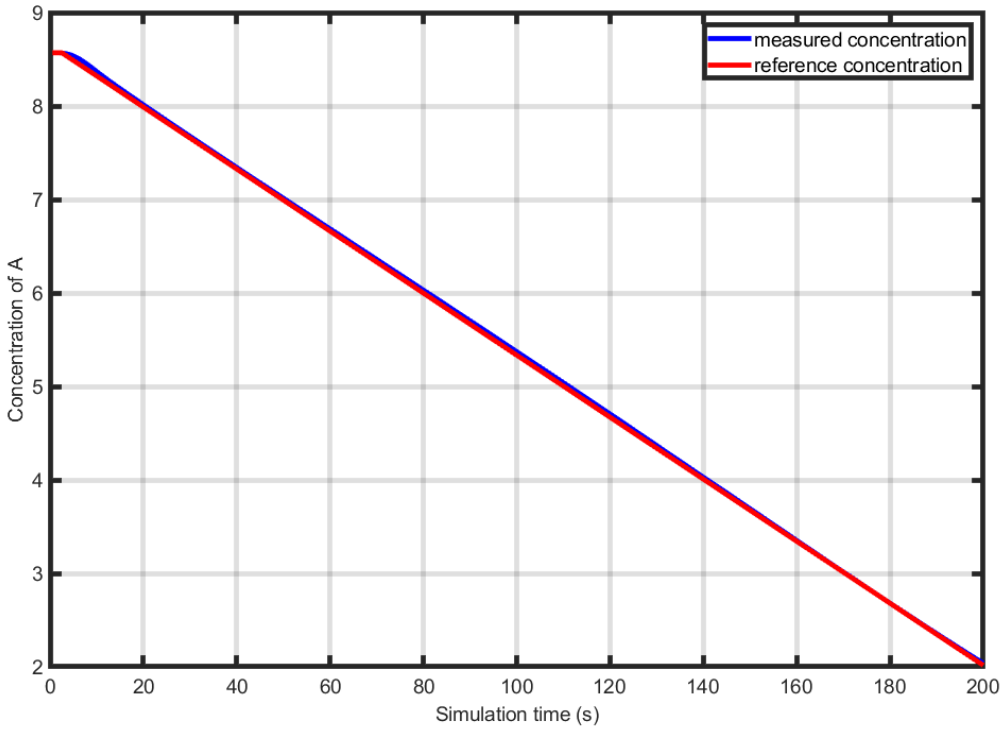


Figure 7.3: Concentration of A as a function of simulation time.

Create an NLMPC object

An nlmpc object with measured and unmeasured disturbance is first created.

```
nlobj = nlmpc(7,3,'MV',[1 2],'MD',3,'UD',4);
```

The NMPC controller sample time, prediction horizon and control horizon are then specified.

```
nlobj.Ts = Ts;
nlobj.PredictionHorizon = 10;
nlobj.ControlHorizon = 2;
```

The dynamics are provided as a function name.

```
nlobj.Model.StateFcn = 'LaneFollowingStateFcn';
```

The output variables returned by *LaneFollowingOutputFcn* are the longitudinal velocity, the lateral deviation and the sum of the yaw angle and yaw angle output disturbance

```
nlobj.Model.OutputFcn = 'LaneFollowingOutputFcn';
```

Bound constraints are set on the manipulated (input) variables.

```
nlobj.MV(1).Min = -3;      % Maximum acceleration 3 m/s^2
nlobj.MV(1).Max = 3;       % Minimum acceleration -3 m/s^2
nlobj.MV(2).Min = -1.13;   % Minimum steering angle -65
nlobj.MV(2).Max = 1.13;    % Maximum steering angle 65
```

Scaling factors are incorporated on output and manipulated variables to optimize solver performance.

```
nlobj.OV(1).ScaleFactor = 15;   % Typical value of longitudinal velocity
nlobj.OV(2).ScaleFactor = 0.5;  % Range for lateral deviation
nlobj.OV(3).ScaleFactor = 0.5;  % Range for relative yaw angle
nlobj.MV(1).ScaleFactor = 6;    % Range of steering angle
nlobj.MV(2).ScaleFactor = 2.26; % Range of acceleration
nlobj.MD(1).ScaleFactor = 0.2;  % Range of Curvature
```

Weights on outputs and the rates of manipulated variables are set in the NLMPC object objective function.

```
nlobj.Weights.OutputVariables = [1 1 0];

%%
% Penalize acceleration change more for smooth driving experience.
nlobj.Weights.ManipulatedVariablesRate = [0.3 0.1];
```

A nonlinear interior-point FORCESPRO solver is generated from a customizable options structure.

```
options = nlmpcToForcesOptions();
% Set solver name
options.SolverName = 'LaneFollowSolver';
% Choose solver type 'InteriorPoint' or 'SQP'
options.SolverType = 'InteriorPoint';
% x0 and u0 are used to create a primal initial guess
options.x0 = x0;
options.mv0 = u0;
tm = tic;
[coredata, onlinedata] = nlmpcToForces(nlobj,options);
tBuild = toc(tm);
```

The FORCESPRO NLMPC Simulink block can then be used seamlessly. It is available in the Simulink Library Browser in the *Model Predictive Control Toolbox* section, as shown in Figure Figure 7.4.

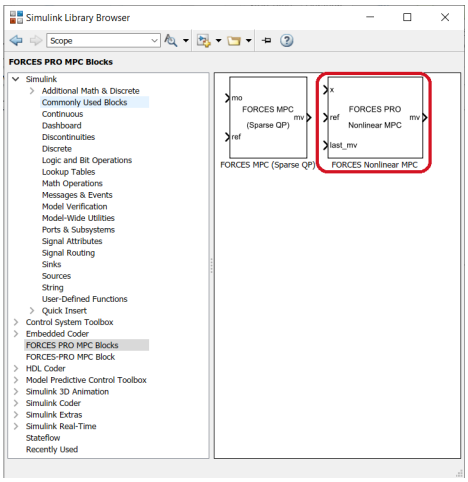


Figure 7.4: FORCESPRO NMPC block.

In order to run the nonlinear interior-point solver, the *coredata* structure returned by *nlmpcToForces* must be provided in the block mask, as shown in Figure Figure 7.5.

The Simulink model can finally be run using the *sim* command.

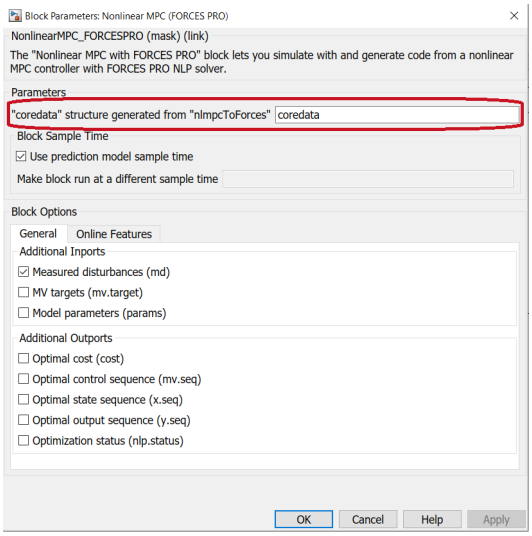


Figure 7.5: FORCESPRO NMPC block mask.

```
sim('LaneFollowingNMPC')
```

Results are shown in Figures Figure 7.6 and Figure 7.7.

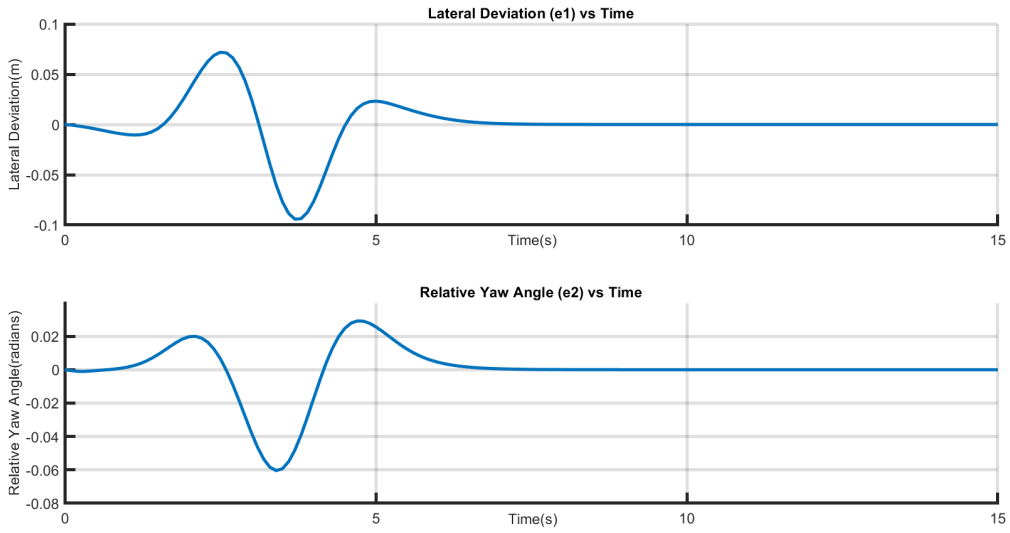


Figure 7.6: Vehicle lateral deviation.

Simulink Coder (R) enables users to generate an executable from the FORCESPRO NLMPC block, so that it can be deployed for real-time applications.

Deploying the Lane Following Model on Speedgoat

The lane following model in Figure Figure 7.8 can be easily deployed on Speedgoat platforms by means of the code below.

```
% Choose Speedgoat x86 platform to run FORCESPRO solver
options.ForcesTargetPlatform = 'Speedgoat-x86';
```

(continues on next page)

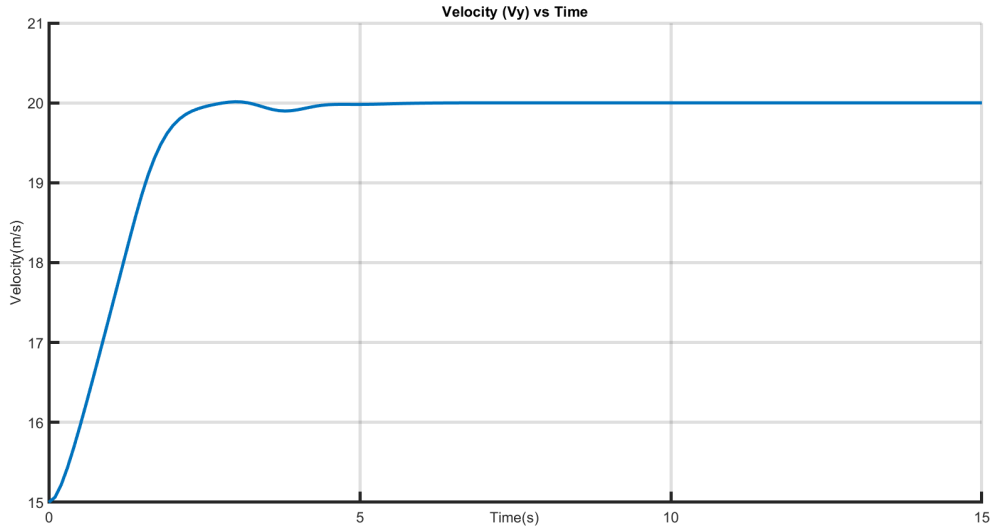


Figure 7.7: Vehicle velocity.

(continued from previous page)

```
% x0 and u0 are used to create a primal initial guess
options.x0 = x0;
options.mv0 = u0;
% Generate FORCESPRO solver
tm = tic;
[coredata, onlinedata] = nlmpcToForces(nlobj,options);
tBuild = toc(tm);

%%
% Start code generation for Speedgoat x86
mdl = 'LaneFollowingNMPC_Speedgoat_x86';
open_system(mdl);                                % Open Simulink(R) Model
load_system(mdl);                                % Load Simulink(R) Model
rtwbuild(mdl);                                   % Start Code Generation

% Deploy application from the start
tg = slrt;
if(~strcmpi(tg.Application, 'loader'))
    tg.unload();
end
tg.load(mdl);

% Execute application
tg.start();
while(strcmpi(tg.Status, 'running'))
    pause(Ts);
end
scope1 = tg.getscope(1);
scope2 = tg.getscope(2);
scope3 = tg.getscope(3);
```

All the files necessary to run this example can be downloaded **here**.

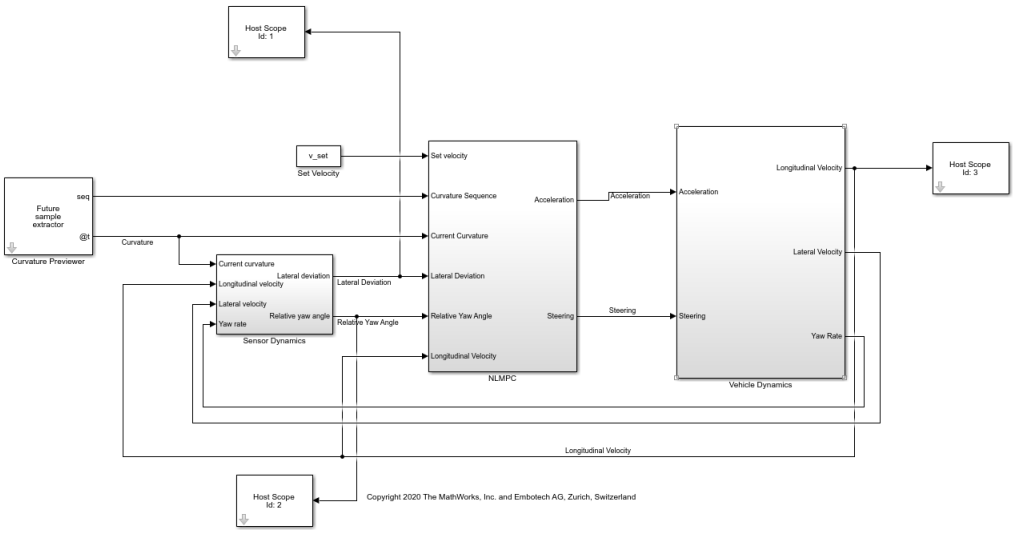


Figure 7.8: Simulink Real-Time Lane Following model for Speedgoat deployment.

7.7.3 Rocket landing example

In this example we consider the motion planning problem of landing a rocket safely. The FORCESPRO solver is generated using a NLMPCMultistage object. We will cover the details of the model below. For further details, see here.

You can find the code of this example to try it out for yourself in the `examples/Matlab/mpc-toolbox-plugin/nonlinearModels/rocket_planner` folder that comes with your FORCESPRO client.

The dynamical model

The model we consider is a first-principles nonlinear dynamical model. The state x of our system is 6-dimensional while the control u is 2-dimensional. The interpretation of the different states/control inputs is given as follows:

$$
\begin{aligned}
u_1 &: \text{ Left thrust} \quad (N) \\
u_2 &: \text{ Right thrust} \quad (N) \\
x_1 &: \text{ Horizontal position of the center of gravity} \quad (m) \\
x_2 &: \text{ Vertical position of the center of gravity} \quad (m) \\
x_3 &: \text{ Tilt with respect to the center of gravity} \quad (r) \\
x_4 &= \frac{dx_1}{dt} \quad (\frac{m}{s}) \\
x_5 &= \frac{dx_2}{dt} \quad (\frac{m}{s}) \\
x_6 &= \text{ Angular velocity} \quad (\frac{r}{s})
\end{aligned}
$$

The differential equation governing the dynamics is given by

$$\begin{aligned}
\dot{x}_1 &= x_4 \\
\dot{x}_2 &= x_5 \\
\dot{x}_3 &= x_6 \\
\dot{x}_4 &= \frac{-\sin(x_3)(u_1+u_2)}{m} \\
\dot{x}_5 &= \frac{\cos(x_3)(u_2-u_1)}{m} - g \\
\dot{x}_6 &= \frac{2L_2(u_2-u_1)}{mL_1^2},
\end{aligned}$$

where we use of the following constants:

Name	Value	Description
L_1	$10m$	Center of gravity to top/bottom end
L_2	$5m$	Center of gravity to left/right end
m	$1kg$	Mass of rocket
g	$9.806\frac{m}{s^2}$	Gravitational constant

Constructing a NLMPCMultistage object

The first step to generate a FORCESPRO solver is to construct a nlmpcMultistage object and set the constraints on our manipulated variables (*MV*) and states (*State*).

```
% Construct nlmpcMultistage object and set dynamics
Ts = 0.2;
pPlanner = 50;
planner = nlmpcMultistage(pPlanner,6,2);
planner.Ts = Ts;

% Limit thrusts between 0 and 8 Newton
planner.MV(1).Min = 0;
planner.MV(1).Max = 8;
planner.MV(2).Min = 0;
planner.MV(2).Max = 8;

% Specify lower bound on y-axis to avoid crashing
planner.States(2).Min = 10;
```

Then we specify the state transition function along with a cost function for every stage. Note that these functions are specified via the of the function.

```
planner.Model.StateFcn = 'RocketStateFcn';
for ct=1:pPlanner
    planner.Stages(ct).CostFcn = 'RocketPlannerCostFcn';
end
```

Specifying solver options and generating a solver

Once we have defined our nlmpcMultistage object *planner* we need to specify information about the solver we would like to generate. This can be done through the options generated by *nlmpcMultistageToForcesOptions*

```
%% Generate FORCES NLP Solver
options = nlmpcMultistageToForcesOptions;
options.Server = 'https://forces.embotech.com/';
options.x0 = x0;
options.mv0 = u0;
options.UseOnlineConstraintMVMin = true;
options.UseOnlineConstraintMVMax = true;
options.UseOnlineConstraintStateMin = true;
```

With both our *options* and *nlmpcMultistage* object at hand we can go ahead and generate the FORCESPRO solver:

```
[coredata, onlinedata, model] = nlmpcMultistageToForces(planner, options);
```

Results

In plot Figure 7.9 the optimal trajectory for landing the rocket is displayed. As can be observed in the generated animation which appears when running the code (see Figure 7.10), the FORCESPRO solver manages to control the rocket and land it safely.

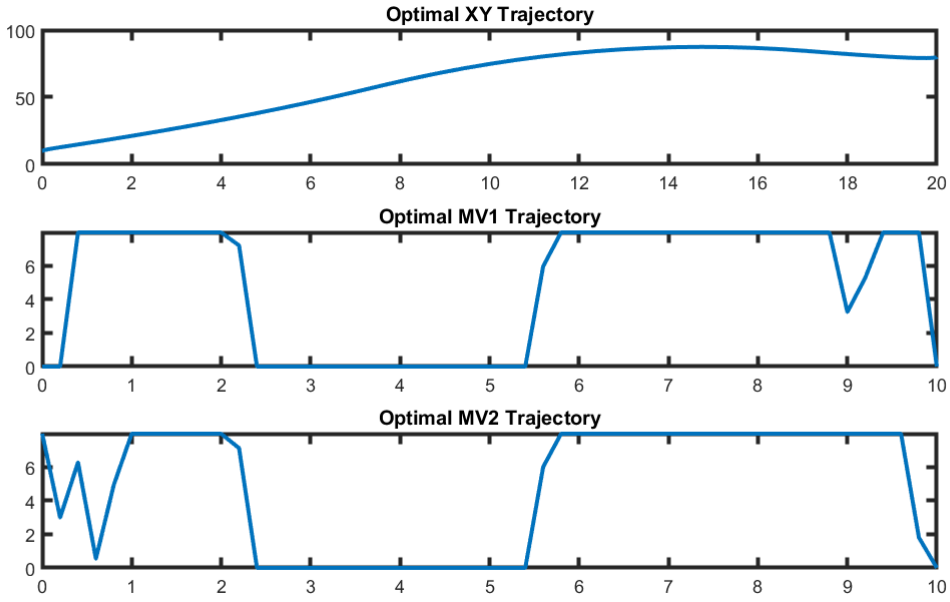


Figure 7.9: Optimal rocket trajectory.

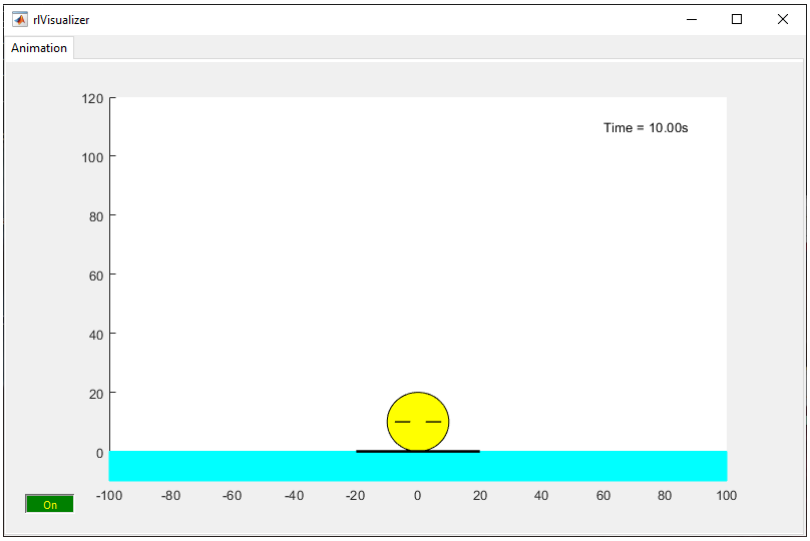


Figure 7.10: Rocket lander animation generated when running the rocket lander example.

Chapter 8

Low-level interface

- *Supported problem class*
- *Multistage struct*
- *Dimensions*
- *Cost function*
- *Equality constraints*
- *Lower and upper bounds*
- *Polytopic constraints*
- *Quadratic constraints*
  - *Example*
- *Binary constraints*
- *Declaring parameters*
- *Declaring Solver Outputs*
  - *Example*
- *Generating the solver*
- *Calling the generated low-level solver*
- *Debugging a formulation*
- *The QP_FAST algorithm*
- *Condensing (automatic state elimination)*

FORCESPRO supports designing solvers and controllers via MATLAB and Python scripts. When using the MATLAB client, a Simulink block is always created such that you can plug your advanced formulation directly into your simulation models, or download it to a real-time target platform.

The low-level interface gives advanced optimization users the full flexibility when designing custom optimization solvers and MPC controllers based on non-standard formulations.

This interface is provided with all variants of FORCESPRO, starting with *Variant S*.

8.1 Supported problem class

The FORCESPRO low-level interface supports the class of **convex multistage quadratically constrained programs (QCQPs)** of the form

$$
\begin{aligned}
\text{minimize} \quad & \sum_{i=1}^{N} \frac{1}{2} z_i^\top H_i z_i + f_i^\top z_i && \text{(separable objective)} \\
\text{subject to} \quad & D_1 z_1 = c_1 && \text{(initial equality)} \\
& C_{i-1} z_{i-1} + D_i z_i = c_i, \;\; i = 2, \ldots, N && \text{(inter-stage equality)} \\
& \underline{z}_i \leq z_i && \text{(lower bound)} \\
& z_i \leq \bar{z}_i && \text{(upper bound)} \\
& A_i z_i \leq b_i && \text{(polytopic inequalities)} \\
& z_i^\top Q_{i,k} z_i + L_{i,k}^\top z_i \leq r_{i,k} && \text{(quadratic inequalities)}
\end{aligned}
$$

for $i = 1, ..., N$ and $k = 1, ..., M$. To obtain a solver for this optimization program using the FORCESPRO client, you need to define all data in the problem, that is the matrices H_i, A_i, $Q_{i,k}$, D_i, C_i and the vectors $\underline{z}_i < \bar{z}_i, b_i, L_{i,k}, r_{i,k}, c_i$, in a MATLAB struct or Python dictionary, along with the corresponding dimensions. The following steps will take you through this process. Importantly, the matrices H_i and $Q_{i,k}$ should all be positive definite.

Note: FORCESPRO supports all problem data to be parametric, i.e. to be unknown at code generation time. Read Section 12 to learn how to use parameters correctly.

In the following, we describe how to model a problem of the above form with FORCESPRO. First make sure that the FORCESPRO client is on the MATLAB/Python path. See Section 3 for more details on how to set up the MATLAB client and Section 3.3.

After the *PYTHONPATH* has been appropriately set up to include your FORCESPRO client directory (see Section 3.3.3), Python users have to import the FORCESPRO module and their user ID.

```
import forcespro
import get_userid
```

8.2 Multistage struct

First, an empty struct/class has to be initialized, which contains all fields needed and initializes matrices and vectors to empty matrices. The command

Matlab

Python

```
stages = MultistageProblem(N);
```

```
stages = forcespro.MultistageProblem(N) # 0-indexed
```

creates such an empty structure/class of length N. Once this structure/class has been created, the corresponding matrices, vectors and dimensions can be set for each element of stages.

8.3 Dimensions

In order to define the dimensions of the stage variables z_i, the number of lower and upper bounds, the number of polytopic inequality constraints and the number of quadratic constraints use the following fields:

Matlab

Python

```
stages(i).dims.n = ...; % length of stage variable zi
stages(i).dims.r = ...; % number of equality constraints
stages(i).dims.l = ...; % number of lower bounds
stages(i).dims.u = ...; % number of upper bounds
stages(i).dims.p = ...; % number of polytopic constraints
stages(i).dims.q = ...; % number of quadratic constraints
```

```
# 0-indexed
stages.dims[i]['n'] = ... # length of stage variable zi
stages.dims[i]['r'] = ... # number of equality constraints
stages.dims[i]['l'] = ... # number of lower bounds
stages.dims[i]['u'] = ... # number of upper bounds
stages.dims[i]['p'] = ... # number of polytopic constraints
stages.dims[i]['q'] = ... # number of quadratic constraints
```

8.4 Cost function

The cost function is, for each stage, defined by the matrix H_i and the vector f_i. These can be set by

Matlab

Python

```
stages(i).cost.H = ...; % Hessian
stages(i).cost.f = ...; % linear term
```

```
# 0-indexed
stages.cost[i]['H'] = ... # Hessian
stages.cost[i]['f'] = ... # linear term
```

Note: whenever one of these terms is zero, you have to set them to zero (otherwise the default of an empty matrix is assumed, which is different from a zero matrix).

`H_i` should be square of size `dims[i]['n']` / `stages(i).dims.n` and `f_i` should also be of that length. It does not matter whether you use a column or row vector for `f`.

8.5 Equality constraints

The equality constraints for each stage, which are given by the matrices C_{i-1}, D_i and the vector c_i, have to be provided in the following form:

The matrices C_{i-1}, D_i should have dimension `dims[i]['r']` by `dims[i]['n']` and c_i should be of length `dim[i]['r']`.

Note the index shift in C_{i-1}. In particular you should take care, that the vertical dimension of C_{i-1} matches `dim[i]['r']` and not `i-1`.

Matlab

Python

```
stages(i).eq.C = ...;
stages(i).eq.c = ...;
stages(i).eq.D = ...;
```

```
# 0-indexed
stages.eq[i]['C'] = ...
stages.eq[i]['c'] = ...
stages.eq[i]['D'] = ...
```

In many parts of the literature a different notation is given for inter-stage equality, which places the next index on the right hand side of the equation like so:

$$A_i x_i + B_i u_i = x_{i+1}, \quad i = 1, \ldots, N-1 \quad \text{(inter-stage equality)}$$

with states x_i and control inputs u_i.

The correct way to translate this to FORCESPRO is as follows:

$$\left(B_{i-1} \mid A_{i-1}\right) z_{i-1} + \left(0 \mid -\mathbf{Id}\right) z_i = 0, \quad i = 2, \ldots, N$$

where $z_i = (u_i, x_i)$. Note that the number of columns of $-\mathbf{Id}$ must match the size of x_i.

There is the common case of a fixed initial value $x_1 = x_{init}$. This can be implemented by adding an equality constraint for $i = 1$:

$$\left(0 \mid -\mathbf{Id}\right) z_1 = -x_{init}$$

Alternatively, one stage can be eliminated by defining the state space as $z_i = (u_i, x_{i+1})$ for $i = 1, \ldots N-1$ (thus removing x_1 and u_N). For that choice, the inter-stage equalities are given by:

$$\begin{aligned} &\left(B_1 \mid -\mathbf{Id}\right) z_1 = -A_1 x_{init} \\ &\left(0 \mid A_i\right) z_{i-1} + \left(B_i \mid -\mathbf{Id}\right) z_i = 0 \quad i = 2, \ldots, N-1 \end{aligned}$$

8.6 Lower and upper bounds

Lower and upper bounds have to be set in sparse format, i.e. an index vector *lbIdx/ubIdx* that defines the elements of the stage variable z_i has to be provided, along with the corresponding upper/lower bound *lb/ub*:

These index vectors will be the same in both MATLAB and Python, which means that the Python indices need to be adjusted to match MATLAB's 1-indexed style.

Matlab

Python

```
stages(i).ineq.b.lbidx = ...; % index vector for lower bounds
stages(i).ineq.b.lb = ...;    % lower bounds
stages(i).ineq.b.ubidx = ...; % index vector for upper bounds
stages(i).ineq.b.ub = ...;    % upper bounds
```

```
stages.ineq[ i ]['b']['lbidx'] = ... # index vector for lower bounds, 1-indexed
stages.ineq[ i ]['b']['lb'] = ...    # lower bounds
stages.ineq[ i ]['b']['ubidx'] = ... # index vector for upper bounds, 1-indexed
stages.ineq[ i ]['b']['ub'] = ...    # upper bounds
```

Both `lb` and `lbIdx` must have length `stages(i).dims.l` / `stages.dims[ i ]['l']`, and both `ub` and `ubIdx` must have length `stages(i).dims.u` / `stages.dims[ i ]['u']`.

8.7 Polytopic constraints

In order to define the inequality $A_i z_i \leq b_i$, use

Matlab

Python

```
stages(i).ineq.p.A = ...; % Jacobian of linear inequality
stages(i).ineq.p.b = ...; % RHS of linear inequality
```

```
# 0-indexed
stages.ineq[i]['p']['A'] = ... # Jacobian of linear inequality
stages.ineq[i]['p']['b'] = ... # RHS of linear inequality
```

The matrix `A` must have `stages(i).dims.p` / `stages.dims[ i ]['p']` rows and `stages(i).dims.n` / `stages.dims[ i ]['n']` columns. The vector `b` must have `stages(i).dims.p` / `stages.dims[ i ]['p']` rows.

8.8 Quadratic constraints

Similar to lower and upper bounds, quadratic constraints are given in sparse form by means of an index vector, which determines on which variables the corresponding quadratic constraint acts.

Matlab

Python

```
stages(i).ineq.q.idx = { idx1, idx2, ...}; % index vectors
stages(i).ineq.q.Q = { Q1, Q2, ...};       % Hessians
stages(i).ineq.q.l = { L1, L2, ...};       % linear terms
stages(i).ineq.q.r = [ r1; r2; ... ];      % RHSs
```

```
stages.ineq[i]['q']['idx'] = ... # index vectors, 1-indexed
stages.ineq[i]['q']['Q'] = ...   # Hessians
stages.ineq[i]['q']['l'] = ...   # linear terms
stages.ineq[i]['q']['r'] = ...   # RHSs
```

If the index vector `idx1` has length m_1, then the matrix `Q` must be square and of size $m_1 \times m_1$, the column vector `l1` must be of size m_1 and `r_1` is a scalar. Of course these dimension rules apply to all further quadratic constraints that might be present. Note that `L_1`, `L_2` etc. are column vectors in MATLAB!

Since multiple quadratic constraints can be present per stage, in MATLAB we make use of the cell notation for the Hessian, linear terms, and index vectors. In Python we make use of Python object arrays for the Hessians, linear terms, and index vectors.

8.8.1 Example

To express the two quadratic constraints

$$z_{3,3}^2 + 2z_{3,5}^2 - z_{3,5} \leq 3$$
$$5z_{3,1}^2 \leq 1$$

on the third stage variable, use the code

Matlab

Python

```
stages(3).ineq.q.idx = { [3 5], [1] } % index vectors
stages(3).ineq.q.Q = { [1 0; 0 2], [5] }; % Hessians
stages(3).ineq.q.l = { [0; -1], [0] }; % linear terms
stages(3).ineq.q.r = [ 3; 1 ]; % RHSs
```

```
stages.ineq[3-1]['q']['idx'] = np.zeros((2,), dtype=object) # index vectors
stages.ineq[3-1]['q']['idx'][0] = np.array([3,5])
stages.ineq[3-1]['q']['idx'][1] = np.array([1])
stages.ineq[3-1]['q']['Q'] = np.zeros((2,), dtype=object) # Hessians
stages.ineq[3-1]['q']['Q'][0] = np.array([1.0 0],[0 2.0])
stages.ineq[3-1]['q']['Q'][1] = np.array([5])
stages.ineq[3-1]['q']['l'] = np.zeros((2,), dtype=object) # linear terms
stages.ineq[3-1]['q']['l'][0] = np.array([0], [-1])
stages.ineq[3-1]['q']['l'][1] = np.array([0])
stages.ineq[3-1]['q']['r'] = np.array([3],[1]) # RHSs
```

8.9 Binary constraints

To declare binary variables, you can use the **bidx** field of the **stages** struct or object. For example, the following code declares variables 3 and 7 of stage 1 to be binary:

Matlab

Python

```
stages(1).bidx = [3 7]
```

```
stages.bidx[0] = np.array([3, 7])
```

That's it! You can now generate a solver that will take into account the binary constraints on these variables. If binary variables are declared, FORCESPRO will add a branch-and-bound procedure to the standard convex solver it generates.

8.10 Declaring parameters

FORCESPRO is a parametric solver. As such, it is necessary to specify which parts of the model are to be parametric.

For a detailed introduction to setting parameters, see the section Section 12.

A common choice of parameter is an x_{init} value. To set this, use the following:

Matlab

Python

```
parameter = newParam('xinit', 1, 'eq.c')
```

```
stages.newParam("xinit", [1], 'eq.c') # 1-indexed
```

8.11 Declaring Solver Outputs

FORCESPRO gives you full control over the part of the solution that should be outputted by the solver. It is also possible to obtain the Lagrange multipliers of certain constraints. To define a standard output as a slice of the primal solution vector, use the function

Matlab

Python

```
output = newOutput(name, maps2stage, idxWithinStage)
```

```
stages.newOutput(name, maps2stage, idxWithinStage)
```

where `name` is the name you give to the output (you will need this to read it after calling the solver). The index vector (or integer) `maps2stage` defines to which stage this output maps to. The last argument, `idxWithinStage` allows the user to select which indices from the stage vector should be outputted by the solver. To retrieve the full primal output for all stages and all variables in a single vector, provide only the first argument (`name`).

To define an output as a slice of certain Lagrange multipliers, use the function

Matlab

Python

```
output = newOutput(name, maps2stage, idxWithinStage, maps2const)
```

```
stages.newOutput(name, maps2stage, idxWithinStage, maps2const)
```

where the remaining argument `maps2const` specifies the constraint associated with the Lagrange multipliers being requested.

Table 8.1: Possible string values for argument `maps2const`

maps2const	Constraint
`r`	Equalities
`u`	Upper bounds
`l`	Lower bounds
`p`	Polytopic bounds

8.11.1 Example

- To define an output to be the first two elements of the primal solution vector, use the following command:

  Matlab

  Python

  ```
  output1 = newOutput('u0', 1, 1:2)
  ```

```
stages.newOutput('u0', 1, range(1,3))
```

- To define an output to be the first and third indices of the Lagrange multipliers for the equality constraints of the second stage, use the following command:

  Matlab

  Python

```
output2 = newOutput('dual_eq0', 2, [1 3], 'r')
```

```
stages.newOutput('dual_eq0', 2, [1,3], 'r')
```

- To define an output to be the full primal solution vector, use the following command:

  Matlab

  Python

```
output2 = newOutput('z')
```

```
stages.newOutput('z')
```

  which, for `N` stages and `n` variables per stage, is equivalent to

  Matlab

  Python

```
output2 = newOutput('z', 1:N, 1:n)
```

```
stages.newOutput('z', range(1,N+1), range(1,n+1))
```

8.12 Generating the solver

After the optimization problem has been formulated into a structure *stages*, an optimized solver can be generated. To do so, the solver requires a name and a number of solver options, as described in Section 18.

Matlab

Python

```
codeoptions = getOptions('solver name');
generateCode(stages, params, codeoptions, outputs);
```

```
options = forcespro.CodeOptions('solver_name')
stages.codeoptions = options
stages.generateCode(get_userid.userid)
```

8.13 Calling the generated low-level solver

After solver generation has completed, the solver itself (as a compiled library) as well as several interfacing files will become available in your working directory. These files are named according to what you named your solver; in the following we assume “SOLVER_NAME”. Calling the solver from MATLAB or Python is then as simple as:

Matlab

Python

```
problem = {}  % a struct of solver parameters
SOLVER_NAME(problem)
```

```
import SOLVER_NAME_py  # notice the _py suffix
problem = {}  # a dictionary of solver parameters
SOLVER_NAME_py.SOLVER_NAME_solve(problem)
```

Note: Don't give your solver the same name as the script you are calling it from. Doing so will overwrite your calling script with the solver interface. For example, in a script named *test_problem.m*, choose a name such as *test_solver* instead of *test_problem*.

Note: The high-level Python interface provides more convenient access to solvers generated using the high-level interface. This method of calling a solver is only available for solvers generated through the low-level interface, and high-level solvers can only be called from Python through the means described in the high-level interface documentation.

8.14 Debugging a formulation

For debugging solvers returning bad exit flags (such as *-7* or *-10*, see *Exitflags*), it is often helpful to convert a FORCESPRO multistage formulation into a standard (QC)QP. The function *stages2qcqp* is provided for that purpose. To learn how to use this function, type

Matlab

Python

```
help stages2qcqp;
```

```
from forcespro.debug import stages2qcqp
help(stages2qcqp)
```

With the MATLAB client, common formulation errors can be detected automatically by making use of the utility *FORCESdiagnostics* (type *help FORCESdiagnostics*).

8.15 The QP_FAST algorithm

From FORCESPRO version 6.0.0 a new algorithm was introduced which allows one to generate extremely fast solvers which are especially well-suited for low-level hardware. Generating such a solver is done by specifying the model through the `stages` object as explained above. The main difference in the solver generation procedure is a tuning step which is explained in details in the following section.

Important: Currently the QP_FAST algorithm is only supported through the MATLAB client of FORCESPRO.

One of the novel features of the QP_FAST algorithm is that it can be tailored to a specific application by "tuning" it. FORCESPRO supports an automated tuning tool (see *Autotuner*)

for choosing the optimal tuning for a given application. A key step in tuning a fast QP solver is collecting data/problems on which it can be tuned. The way to do this is to first generate a general QP (FORCESPRO) solver which does not require tuning. This is done as follows:

Matlab

Python

```
codeoptions.solvemethod = 'PDIP';
generateCode(stages, params, codeoptions, outputs);
```

```
# not supported
```

Now one or more simulations can be performed and all the problems (i.e. the inputs to the solver at run-time) which are solved should be collected and stored in a cell array (here called `problems`). We refer to `problems` as the *tuning data*. Once tuning data has been collected, a fast QP solver can be generated and tuned as follows:

Matlab

Python

```
tuningoptions = ForcesAutotuneOptions(problems);
ForcesGenerateQpFastSolver(stages,params,codeoptions,tuningoptions,outputs);
```

```
# not supported
```

The autotuning procedure allows for quite a bit of customization which can be specified through the `ForcesAutotuneOptions` object (see *Autotuner Options*).

You can find an example called "FORCESPRO_ActiveSuspensionControlQpFast.m" in the `examples` folder of your FORCESPRO client, which shows the full workflow for the fast QP solver.

When generating a QP_FAST solver, additional codeoptions are available as described in Section *QP_FAST method*.

8.16 Condensing (automatic state elimination)

The term *condensing* refers to the elimination of all stage variables that correspond to dynamic states. This often yields a performance benefit for formulations with a small number of inputs (as compared to the total number of stage variables), and with a short control horizon. The FORCESPRO implementation of condensing internally maps the sparse multi-stage formulation to a dense single-stage formulation. This mapping is performed automatically inside FORCESPRO (offline at code generation and online at run time) so that the user can readily use condensing without needing to adapt his sparse formulation. This feature can be enabled by setting

Matlab

Python

```
codeoptions.condense = 1;
```

```
codeoptions.condense = 1
```

This works as long as a few requirements about the user formulation are satisfied, namely that there is a fixed initial state $x_i = x_{init}$ and that the equality constraints can be expressed

in terms of states x_i and control inputs u_i as

$$A_i x_i + B_i u_i = x_{i+1}, \quad i = 1, \ldots, N-1$$

The formulation needs to be set up as the first variant outlined in Section 8.5:

- the stage variables are ordered as

$$z_i = (u_i, x_i), \quad i = 1, \ldots, N$$

- The equality constraints matrices C_i and D_i must be of the form

$$\begin{aligned} C_i &= \left(B_i \mid A_i\right), & i &= 1, \ldots, N-1 \\ D_i &= \alpha_i \left(0 \mid -\mathbf{Id}\right), & i &= 1, \ldots, N \end{aligned}$$

where α_i are arbitrary scalars.

You can find an example for a condensing-based solver in `Matlab\LowLevelInterface\BasicExample\BasicExampleCondensing.m` (Matlab version) or `Python\LowLevelInterface\simplempc_forcespro_condensing.py` (Python version) in the `examples` folder of your FORCESPRO client. This is the same system as *Basic MPC Example* but adapted to satisfy the above requirements for condensing.

Note: Please note that if you use `codeoptions.init = 2` together with condensing (see Table 18.10), the run-time parameter `z_init_00` for the initial guess refers to the condensed formulation, i.e. it corresponds to all controls stacked together into a single vector.

Chapter 9

High-level Interface

- *Supported problems*
  - *Canonical problem for discrete-time dynamics*
  - *Continuous-time dynamics*
  - *Other variants*
  - *Function evaluations*
- *Expressing the optimization problem in code*
  - *Model Initialization*
  - *Dimensions*
  - *Objective*
  - *Equalities*
  - *Initial and final conditions*
  - *Inequalities*
  - *Variations*
  - *Single precision callbacks*
- *Generating a solver*
  - *Declaring outputs*
- *Calling the solver*
  - *Initial guess*
  - *Initial and final conditions*
  - *Real-time parameters*
  - *Tolerances as real-time parameters*
  - *Exitflags and quality of the result*
- *External function evaluations in C*
  - *Interface*
  - *Supplying function evaluation information*
  - *Rules for function evaluation code*
  - *Matrix format*

  - *Multiple source files*
  - *Stage-dependent functions*
  - *External function return values*
- *Calling the nonlinear functions from MATLAB or Python*
  - *Calling the nonlinear functions from MATLAB*
  - *Calling the nonlinear functions from Python*
- *Mixed-integer nonlinear solver*
  - *Writing a mixed-integer model*
  - *Mixed-integer solver customization via user callbacks*
  - *Providing a guess for the incumbent*
- *Sequential quadratic programming algorithm*
  - *How to generate a SQP solver*
  - *Different SQP variants*
  - *Tuning the SQP Fast solver*
  - *The hessian approximation and line search settings*
  - *Controlling the initial guess at run-time*
  - *Additional code options specific to the SQP-RTI solver*
- *Differences between the MATLAB and the Python client*
- *Examples*

The FORCESPRO high-level interface gives optimization users a familiar easy-to-use way to define an optimization problem. The interface also gives the advanced user full flexibility when importing external C-coded functions to evaluate the quantities involved in the optimization problem.

This interface is provided with *Variant L* and partially with *Variant M* of FORCESPRO.

Important: Starting with FORCESPRO `1.8.0`, the solver generated from the high-level interface supports nonlinear and convex decision making problems with integer variables.

Note: The high-level Python interface expects 0-based indices in the model formulation, such as for the indices in *lbidx*, *ubidx*, *hlidx*, *huidx*, *xinitidx* and *xfinalidx*, as is usual in Python programs. Note that this is contrary to the low-level interface, which requires 1-based indices for these fields.

9.1 Supported problems

9.1.1 Canonical problem for discrete-time dynamics

The FORCESPRO NLP solver solves (potentially) non-convex, finite-time nonlinear optimal control problems with horizon length N of the form:

$$
\begin{aligned}
\text{minimize} \quad & \sum_{k=1}^{N} f_k(z_k, p_k) && \text{(separable objective)} \\
\text{subject to} \quad & z_1(\mathcal{I}) = z_{\text{init}} && \text{(initial equality)} \\
& E_k z_{k+1} = c_k(z_k, p_k) && \text{(inter-stage equality)} \\
& \underline{h}_k \leq h_k(z_k, p_k) \leq \bar{h}_k && \text{(nonlinear constraints)} \\
& z_N(\mathcal{N}) = z_{\text{final}} && \text{(final equality)} \\
& \underline{z}_k \leq z_k \leq \bar{z}_k && \text{(lower-upper bounds)} \\
& F_k z_k \in [\underline{z}_k, \bar{z}_k] \cap \mathbb{Z} && \text{(integer variables)}
\end{aligned}
$$

for $k = 1, \ldots, N$, where $z_k \in \mathbb{R}^{n_k}$ are the optimization variables, for example a collection of inputs, states or outputs in an MPC problem; $p_k \in \mathbb{R}^{l_k}$ are real-time data, which are not necessarily present in all problems; the functions $f_k : \mathbb{R}^{n_k} \times \mathbb{R}^{l_k} \to \mathbb{R}$ are stage cost functions; the functions $c_k : \mathbb{R}^{n_k} \times \mathbb{R}^{l_k} \to \mathbb{R}^{w_k}$ represents (potentially nonlinear) equality constraints, such as a state transition function; the matrices E_k are used to couple variables from the $(k+1)$-th stage to those of stage k through the function c_k; and the functions $h_k : \mathbb{R}^{n_k} \times \mathbb{R}^{l_k} \to \mathbb{R}^{m_k}$ are used to express potentially nonlinear, non-convex inequality constraints. The index sets $\mathcal{I}$ and $\mathcal{N}$ are used to determine which variables are fixed to initial values z_{init} and final values z_{final}, respectively; initial and final values can also be changed in real-time. Finally, lower and upper bounds $\underline{z}_k < \bar{z}_k$ on z_k can be specified at every stage k; the matrix F_k is a selection matrix that may select certain coordinates in vector z_k to be integer.

All real-time data is coloured in red. Additionally, when integer variables are modelled, they need to be declared as real-time parameters (see section *Mixed-integer nonlinear solver*).

To obtain a solver for this optimization problem using the FORCESPRO client, you need to define all functions involved (f_k, c_k, h_k) along with the corresponding dimensions.

9.1.2 Continuous-time dynamics

Instead of having discrete-time dynamics as can be seen in Section 9.1.1, we also support using continuous-time dynamics of the form:

$$\dot{x} = f(x, u, p)$$

and then discretizing this equation by one of the standard integration methods. See Section 9.2.4 for more details.

9.1.3 Other variants

Not all elements in Section 9.1.1 have to be necessarily present. Possible variants include problems:

- where all functions are fixed at code generation time and do not need extra real-time data p;
- with no lower (upper) bounds for variable $z_{k,i}$, then $\underline{z}_i \equiv -\infty$ ($\bar{z}_i \equiv +\infty$);
- without nonlinear inequalities h;
- with $N = 1$ (single stage problem), then the inter-stage equality can be omitted;
- with final equality, then the final stage cost function f_N can be omitted;

- that optimize over the initial value z_{init} and do not include the initial equality;
- that optimize over the final value z_{final} and do not include the final equality;
- mixed-integer nonlinear programs, where some variables are declared as integers (see section *Mixed-integer nonlinear solver* for more information about the MINLP solver).

9.1.4 Function evaluations

The FORCESPRO NLP solver requires external functions to evaluate:

- the cost function terms $f_k(z_k)$ and their gradients $\nabla f_k(z_k)$,
- the dynamics $c_k(z_k)$ and their Jacobians $\nabla c_k(z_k)$, and
- the inequality constraints $h_k(z_k)$ and their Jacobians $\nabla h_k(z_k)$.

The FORCESPRO code generator supports the following ways of supplying these functions:

1. Automatic C-code generation of these functions from MATLAB using a supported automatic differentiation (AD) tool, such as CasADi. This happens automatically in the background, as long as the AD tool is found on your system. By doing so, the user does not need to adhere to any tool-specific syntax but can use standard MATLAB commands to define the necessary functions instead (which are then automatically converted to match the specifics of the chosen AD tool). This is the recommended way of getting started with FORCESPRO NLP. See Section 9.2 to learn how to use this approach.

2. C-functions (source files). These can be hand-coded, or generated by any automatic differentiation tool. See Section 9.5 for details on how to provide own function evaluations and derivatives to FORCESPRO.

9.2 Expressing the optimization problem in code

When solving nonlinear programs of the type in Section 9.1.1, FORCESPRO requires the functions f, c, h and their derivatives (Jacobians) to be evaluated in each iteration. There are two ways for accomplishing this: either implement all function evaluations in C by some other method (by hand or by another automatic differentiation tool), or use an AD tool integrated with FORCESPRO, such as the open-source package CasADi (see *Automatic differentiation tool* for a list of all supported tools). This is the easiest option to quickly get started with solving NLPs, and it generates efficient code. However, if you prefer other AD tools, see Section 9.5 to learn how to provide your own derivatives to FORCESPRO NLP solvers. This section will describe the CasADi-based approach in detail, using either the MATLAB or the Python client of FORCESPRO. Please note that even though both the MATLAB and the Python client are intended to behave largely identical, there are some differences between the two clients. For details, refer to *Differences between the MATLAB and the Python client*.

9.2.1 Model Initialization

Model Initialization in MATLAB

In the MATLAB high-level interface, the formulation of the optimization problem is given through a simple structure array. In the following, we will describe the problem in such an array named *model*. It is advisable to zero-initialize this variable at the beginning of your script such that no values set in previous iterations of your script interfere with this run:

```
model = {}
```

Model Initialization in Python

In the high-level Python interface, optimization problems are described through objects of different types, depending on the problem. The following classes are available:

- **SymbolicModel** - Allows you to describe your optimization problem using regular Python functions. These functions will be evaluated symbolically by CasADi to create optimized C code. Note that this model is meant to be used for nonlinear models. If you wish to express a convex model symbolically, consider using the *ConvexSymbolicModel* or forcing generation of a nonconvex solver by setting the option *forcenonconvex* to *True*.
- **ExternalFunctionModel** - Enables more flexibility in describing nonlinear problems by allowing any external function to be used as objective function and constraints. This requires C code or already compiled code (object files or shared libraries) from any language. The approach using external function evaluations for your objective function and constraints is described in *External function evaluations in C*, including the required call signature of the callback function.
- **ConvexSymbolicModel** - FORCESPRO can generate different solvers for convex problems.

Whichever model you choose, it can be initialized with no arguments, or with a single argument denoting the number of stages N in the problem:

```
import forcespro.nlp
model = forcespro.nlp.SymbolicModel(50)
```

Note that most symbolic problem descriptions will also require the Numpy and CasADi packages, so it is a good idea to import them at the beginning:

```
import numpy as np
import casadi
```

9.2.2 Dimensions

In order to define the dimensions of the stage variables z_i, the number of equality and inequality constraints and the number of real-time parameters use the following fields (properties) in the client:

Matlab

Python

```
model.N = 50;   % length of multistage problem
model.nvar = 6; % number of stage variables
model.neq = 4;  % number of equality constraints
model.nh = 2;   % number of nonlinear inequality constraints
model.npar = 0; % number of runtime parameters
```

```
model.N = 50    # not required if already specified in initializer
model.nvar = 6  # number of stage variables
model.neq = 4   # number of equality constraints
model.nh = 2    # number of nonlinear inequality constraints
model.npar = 0  # number of runtime parameters
```

If the dimensions vary for different stages use arrays of length N. See Section 9.2.7 for an example.

9.2.3 Objective

The high-level interface allows you to define the objective function using a handle to a MATLAB or Python function that evaluates the objective. This function is called with the variables of one stage as its first argument, i.e. a vector of *model.nvar* entries. FORCESPRO will process the given function symbolically and generate the necessary C code to be included in the solver.

Matlab

Python

```
model.objective = @eval_obj; % handle to objective function
```

```
model.objective = eval_obj  # eval_obj is a Python function
```

For instance, the function could be:

Matlab

Python

```
function f = eval_obj ( z )
    F = z(1);
    s = z(2);
    y = z(4);
    f = -100*y + 0.1*F^2 + 0.01* s^2;
end
```

```
def eval_obj(z):
    F = z[0]
    s = z[1]
    y = z[3]
    return -100*y + 0.1*F**2 + 0.01*s**2
```

If the cost function varies for different stages use a cell array of function handles of length N in MATLAB, or a list of function handles in Python. See Section 9.2.7 for an example.

Note: Python and MATLAB use different indexing bases. The first element of any variable has index 1 in MATLAB, whereas it is accessed at offset 0 in Python!

The objective evaluation function can optionally accept an additional argument *p* which serves as a run-time parameter. In order to be able to change the terms in the cost function during runtime, one can define the objective function as:

Matlab

Python

```
function f = eval_obj ( z, p )
    F = z(1);
    s = z(2);
    y = z(4);
    f = -100*y + p(1)*F^2 + p(2)* s^2;
end
```

```
def eval_obj(z, p):
    F = z[0]
    s = z[1]
```

(continues on next page)

(continued from previous page)

```
    y = z[3]
    return -100*y + p[0]*F**2 + p[1]*s**2
```

The length of this additional parameter vector in each stage is given by *model.npar*.

9.2.4 Equalities

Discrete-time

For discrete-time dynamics, one can define a handle to a function evaluating c as shown below. The selection matrix E that determines which variables are affected by the inter-stage equality must also be filled. For performance reasons, it is recommended to order variables such that the selection matrix has the following structure:

Matlab

Python

```
model.eq = @eval_dynamics;      % handle to inter-stage function
model.E = [zeros(4,2), eye(4)]; % selection matrix
```

```
model.eq = eval_dynamics  # handle to inter-stage function
model.E = np.concatenate([np.zeros((4, 2)), np.eye(4)], axis=1)  # selection matrix
```

If the equality constraint function varies for different stages use a cell array (or list in Python) of function handles of length $N-1$, and similarly for E_k. See Section 9.2.7 for an example.

Continuous-time

For continuous-time dynamics, FORCESPRO requires you to describe the dynamics of the system in the following form:

$$\dot{x} = f(x, u, p)$$

where x are the states of the system, u are the inputs and p a vector of parameters, e.g. the mass or inertia. The selection matrix E determines which components of the stage variable z_i are to be considered state x or input u in this representation.

For example, let's assume that the system to be controlled has the dynamics:

$$\dot{x} = p_1 x_1 x_2 + p_2 u$$

In order to discretize the system for use with FORCESPRO we have to:

1. Implement the continuous-time dynamics as a function:

Matlab

Python

```
function xdot = continuous_dynamics(x, u, p)
      xdot = p(1)*x(1)*x(2) + p(2)*u;
end
```

```
def continuous_dynamics(x, u, p):
    return p[0]*x[0]*x[1] + p[1]*u[0]
```

Note that in general the parameter vector p can be omitted if there are no parameters. You can also implement short functions as anonymous function handles:

Matlab

Python

```
continuous_dynamics_anonymous = @(x,u,p) p(1)*x(1)*x(2) + p(2)*u;
```

```
continuous_dynamics_anonymous = lambda x, u, p: p[0]*x[0]*x[1] + p[1]*u[0]
```

2. Tell FORCESPRO that you are using continuous-time dynamics by setting the `continuous_dynamics` field of the `model` to a function handle to one of the functions above:

Matlab

Python

```
model.continuous_dynamics = @continuous_dynamics;
```

```
model.continuous_dynamics = continuous_dynamics
```

or, if you are using anonymous functions:

Matlab

Python

```
model.continuous_dynamics = @continuous_dynamics_anonymous;
```

```
model.continuous_dynamics = continuous_dynamics_anonymous
```

3. Use the selection matrix E to link the stage variables z_i with the states x and inputs u of the continuous dynamics function:

Matlab

Python

```
model.E = [zeros(2, 1), eye(2)]
```

```
model.E = np.concatenate([np.zeros((2, 1)), np.eye(2)], axis=1)
```

Components of z_i are considered as state variables x according to the order prescribed by the selection matrix. If an entire k-th column of the selection matrix is zero, the k-th component of z_i is not governed by a dynamic equation and thus considered as input u.

4. Choose one of the integrator functions from the *Integrators* section (the default is ERK4):

Matlab

Python

```
codeoptions.nlp.integrator.type = 'ERK2';
codeoptions.nlp.integrator.Ts = 0.1;
codeoptions.nlp.integrator.nodes = 5;
```

```
codeoptions.nlp.integrator.type = 'ERK2'
codeoptions.nlp.integrator.Ts = 0.1
codeoptions.nlp.integrator.nodes = 5
```

where the integrator type is set using the type field of the options struct `codeoptions.nlp.integrator`. The field `Ts` determines the absolute time between two integration intervals, while

nodes defines the number of intermediate integration nodes within that integration interval. In the example above, we use 5 steps to integrate for 0.1 seconds, i.e. each integration step covers an interval of 0.02 seconds.

9.2.5 Initial and final conditions

The indices affected by the initial and final conditions can be set as follows:

Matlab

Python

```
model.xinitidx = 3:6;   % indices affected by initial condition
model.xfinalidx = 5:6;  % indices affected by final condition
```

```
model.xinitidx = range(2, 6)   # indices affected by the initial condition
model.xfinalidx = range(4, 6)  # indices affected by the final condition
```

Note: Python and MATLAB use different indexing bases. The first variable in a stage has index 1 in MATLAB, whereas it is accessed at offset 0 in Python! Further note that Python's *range* does not include the upper limit, thus:

```
list(range(2, 6)) == [2, 3, 4, 5]  # does not include upper limit
```

9.2.6 Inequalities

A function evaluating nonlinear inequalities can be provided in a similar way, for example:

Matlab

Python

```
function h = eval_const(z)
  x = z(3);
  y = z(4);
  h = [x^2 + y^2;
       (x+2)^2 + (y-2.5)^2 ];
end
```

```
def eval_const(z):
    x = z[2]
    y = z[3]
    return np.array([x**2 + y**2;
                     (x+2)**2 + (y-2.5)**2])
```

Note: For Python installations with Numpy version 1.20 onwards it is advised to use CasADi arrays and CasADi functions (where available) for the implementation of the functions assigned to: **model.objective**, **model.eq**, **model.ineq**, **model.continuous_dynamics** for the problem formulation in order to ensure maximum compatibility between CasADi and the FORCESPRO Python client.

The simple bounds and the nonlinear inequality bounds can have `+inf` and `-inf` elements, but must be the same length as the field **nvar** and **nh**, respectively:

Matlab

Python

```
model.ineq = @eval_const;              % handle to nonlinear constraints
model.hu = [9, +inf];                  % upper bound for nonlinear constraints
model.hl = [1, 0.95^2];                % lower bound for nonlinear constraints
model.ub = [+5, +1, 0, 3, 2, +pi];     % simple upper bounds
model.lb = [-5, -1, -3, -inf,  0, 0]; % simple lower bounds
```

```
model.ineq = eval_const                          # handle to nonlinear constraints
model.hu = [9, +float('inf')]                    # upper bound for nonlinear constraints
model.hl = [1, 0.95**2]                          # lower bound for nonlinear constraints
model.ub = [+5, +1, 0, 3, 2, +np.pi]             # simple upper bounds
model.lb = [-5, -1, -3, -float('inf'), 0, 0]  # simple lower bounds
```

Note: While the FORCESPRO Python client does not require you to use numpy arrays, we encourage their use for vector- and matrix-valued properties of the model, as it simplifies calculations for the user. Therefore, any of the above properties can also be set to Numpy arrays instead of lists. If lists are given, these are converted to Numpy arrays internally.

If the constraints vary for different stages, use cell arrays of length N for any of the quantities defined above. See *Varying dimensions, parameters, constraints, or functions* section for an example.

Bounds **model.lb**, **model.ub**, **model.hl** and **model.hu** can be made parametric by leaving said fields empty and using the **model.lbidx**, **model.ubidx**, **model.hlidx** and **model.huidx** fields respectively to indicate on which variables/inequalities lower and upper bounds are present. The numerical values will then be expected at runtime. The runtime parameters will be created stage-wise for the above bounds and will have the names **lb<n>**, **ub<n>**, **hl<n>**, **hu<n>** where **<n>** will be the 1-based stage number they belong it (padded with enough *0* based on the largest stage) For example, to set parametric lower bounds on states 1 and 2, and parametric upper bounds on states 2 and 3, use:

Matlab

Python

```
% Lower bounds are parametric (indices not mentioned here are -inf)
model.lbidx = [1 2]';

% Upper bounds are parametric (indices not mentioned here are +inf)
model.ubidx = [2 3]';

% lb and ub have to be empty when using parametric bounds
model.lb = [];
model.ub = [];
```

```
# Lower bounds are parametric (indices not mentioned here are -inf)
model.lbidx = [0, 1]

# Upper bounds are parametric (indices not mentioned here are +inf)
model.ubidx = [1, 2]

# There is no need to specify model.lb or model.ub to empty lists if
# model.lbidx or model.ubidx are set, and any non-empty value is disallowed.
```

and then specify the exact values at runtime through the fields **lb01**–**lbN** and **ub01**–**ubN**:

Matlab

Python

```
% Specify lower bounds
problem.lb01 = [0 0]';
problem.lb02 = [0 0]';
% ...

% Specify upper bounds
problem.ub01 = [3 2]';
problem.ub02 = [3 2]';
% ...
```

```
# Specify lower bounds
problem["lb01"] = [0, 0]
problem["lb02"] = [0, 0]

# Specify upper bounds
problem["ub01"] = [3, 2]
problem["ub02"] = [3, 2]
```

Tip: One could use `problem.(sprintf('lb%02u',i))` in an `i`-indexed loop to set the parametric bounds more easily in the MATLAB client. Similarly, the parametric bounds for the stages can be set using `problem["lb{:02d}".format(i+1)]` in Python. Alternatively, consider using the option *stack_parambounds*, described below.

If the `model.lbidx` and `model.ubidx` fields vary for different stages use cell arrays of length N. From Release `3.0.1`, the parametric bounds can be stacked on one same array covering all stages. To enable this behavior, users need to set the following code-generation option:

Matlab

Python

```
codeoptions.nlp.stack_parambounds = 1;
```

```
codeoptions.nlp.stack_parambounds = True
```

This option is effective for both the `PDIP_NLP` and `SQP_NLP` solve methods and works with bounds on variables and inequalities. At run-time, users can then write

Matlab

Python

```
% Lower and upper bounds stacked over all stages
problem.lb = [0 0 0 0 ...];
problem.ub = [3 2 3 2 ...];
```

```
# Lower and upper bounds stacked over all stages
problem["lb"] = [0, 0, 0, 0, ...]
problem["ub"] = [3, 2, 3, 2, ...]
```

Alternatively, if you want to use the same bounds across all stages:

Matlab

Python

```
problem.lb = repmat([0, 0], 1, model.N);
problem.ub = repmat([3, 2], 1, model.N);
```

```
problem["lb"] = np.tile([0, 0], (model.N,))
problem["ub"] = np.tile([3, 2], (model.N,))
```

9.2.7 Variations

Varying dimensions, parameters, constraints, or functions

The example described above has the same dimensions, bounds and functions for the whole horizon. One can define varying dimensions using arrays and varying bounds and functions using MATLAB cell arrays or Python lists. For instance, to remove the first and second variables from the last stage one could write the following:

Matlab

Python

```
for i = 1:model.N-1
  model.nvar(i) = 6;
  model.objective{i} = @(z) -100*z(4) + 0.1*z(1)^2 + 0.01*z(2)^2;
  model.lb{i} = [-5, -1, -3, 0, 0, 0];
  model.ub{i} = [+5 , +1, 0, 3, 2, +pi];
  if i < model.N-1
    model.E{i} = [zeros(4, 2), eye(4)];
  else
    model.E{i} = eye(4);
  end
end

model.nvar(model.N) = 4;
model.objective{model.N} = @(z) -100*z(2);
model.lb{model.N} = [-3, 0, 0, 0];
model.ub{model.N} = [ 0, 3, 2, +pi];
```

```
model = forcespro.nlp.SymbolicModel(50)  # to set values stage-wise, the model must␣
↪be initialized this way

for i in range(0,model.N-1):
  model.nvar[i] = 6
  model.objective[i] = lambda z: -100*z[3] + 0.1*z[0]**2 + 0.01*z[1]**2
  model.lb[i] = [-5, -1, -3, 0, 0, 0]
  model.ub[i] = [+5 , +1, 0, 3, 2, +np.pi]
  if i < model.N-2:
    model.E[i] = np.concatenate([np.zeros(4, 2), np.eye(4)], axis=1)
  else:
    model.E[i] = np.eye(4)

model.nvar[-1] = 4
model.objective[-1] = lambda z: -100*z[1]
model.lb[-1] = [-3, 0, 0, 0]
model.ub[-1] = [ 0, 3, 2, +np.pi]
```

It is also typical for model predictive control problems (MPC) that only the last stage differs from the others (excluding the initial condition, which is handled separately). Instead of defin-

ing cell arrays as above for all stages, FORCESPRO offers the following shorthand notations that alter the last stage:

- `nvarN`: number of variables in last stage
- `nparN`: number of parameters in last stage
- `objectiveN`: objective function for last stage
- `EN`: selection matrix E for last stage update
- `nhN`: number of inequalities in last stage
- `ineqN`: inequalities for last stage

Add any of these fields to the `model` struct/object to override the default values, which is to make everything the same along the horizon. For example, to add a terminal cost that is a factor 10 higher than the stage cost:

Matlab

Python

```
model.objectiveN = @(z) 10*model.objective(z);
```

```
model.objectiveN = lambda z: 10*model.objective(z)
```

Providing analytic derivatives

The algorithms inside FORCESPRO need the derivatives of the functions describing the objective, equality and inequality constraints. The code generation engine uses algorithmic differentiation (AD) to compute these quantities. Instead, when analytic derivatives are available, the user can provide them using the fields `model.dobjective`, `model.deq`, and `model.dineq`.

Note that the user must be particularly careful to make sure that the provided functions and derivatives are consistent, for example:

Matlab

Python

```
model.objective = @(z) z(3)^2;
model.dobjective = @(z) 2*z(3);
```

```
model.objective = lambda z: z[2]**2
model.dobjective = lambda z: 2*z[2]
```

The code generation system will not check the correctness of the provided derivatives.

9.2.8 Single precision callbacks

Evaluating objective function, dynamics and constraints as well as their respective derivatives may take a significant part of the overall solution time (both total and per iteration). In such situations solution time and memory consumption may be sped up by evaluating those functions in single, rather than double precision arithmetic. This can be done by specifying

Matlab

Python

```
codeoptions.callback_floattype = 'float';
```

```
# not yet supported
```

Note that this will allow to run the NLP solver in mixed-precision arithmetic, where the callbacks are evaluated in single precision, but the overall algorithm in double precision. In order for this to work well, all callbacks functions need to be numerically well-posed and overall accuracy requirements of the solution must not be too high. In particular, when using that feature, you may need to reduce some of the accuracy settings (such as `codeoptions.nlp.TolStat`) by one or two orders of magnitude, see *Accuracy requirements*.

Note: Single precision callbacks are currently supported for legacy and chainrule integrators, but not yet for VDE integrators. Also, this features is currently only available via the MATLAB client.

9.3 Generating a solver

In addition to the definition of the NLP, solver generation requires an (optional) set of options for customization (see the *Solver Options* section for more information). Using the default solver options we generate a solver using:

Matlab

Python

```
% Get the default solver options
codeoptions = getOptions('FORCESNLPsolver');

% Generate solver
FORCES_NLP(model, codeoptions);
```

```
# Get the default solver options
options = forcespro.CodeOptions('FORCESNLPsolver')

# Generate solver for previously initialized model
solver = model.generate_solver(options)
```

As the solver is generated, several files are downloaded into the current working directory of the calling script, including the compiled solver itself and MATLAB/Python interfaces for calling it.

Note: In the Python client, *generate_solver()* returns a **solver object**. This object can be used to call the solver. To get a solver object for a previously generated solver in some directory */path/to/solver*, use:

```
import forcespro.nlp
solver = forcespro.Solver.from_directory('/path/to/solver')
```

9.3.1 Declaring outputs

By default, the solver returns the solution vector for all stages as multiple outputs. Alternatively, the user can pass a third argument to the function `FORCES_NLP` with an array that specifies what the solver should output. For instance, to define an output, named `u0`, to be the first two elements of the solution vector at stage 1, use the following commands:

Matlab

Python

```
output1 = newOutput('u0', 1, 1:2);
FORCES_NLP(model, codeoptions, output1);
```

```
output_1 = ("u0", 0, [0, 1], "")
model.generate_solver(options, [output_1])
```

Additionally, you can request that the solver returns the solution vectors for all different stages "stacked" together into a single vector, say called `sol`, by using the following commands:

Matlab

Python

```
output1 = newOutput('sol');
FORCES_NLP(model, codeoptions, output1);
```

```
output1 = ("sol", [], [])
model.generate_solver(options, [output1])
```

Important: When using the MINLP solver and defining outputs, all integer variables need to be specified as custom outputs.

The dual variables at the solution returned by FORCESPRO provide useful information on the problem sensitivity. They can be exported from the nonlinear solver as well by giving the `maps2const` field one of the following values:

- *'nl_eq_dual'* for the dual variables associated to equality constraints
- *'nl_lb_var_dual'* for the dual variables associated to lower bounds on variables
- *'nl_ub_var_dual'* for the dual variables associated to upper bounds on variables
- *'nl_ip_ineq_dual'* for the dual variables associated to nonlinear inequalities
- *'nl_ineq_slack'* for the dual variables associated to slacks on nonlinear inequalities.

An example of exporting the marginals associated to nonlinear equalities is shown in the code snippet below.

```
outputs(4) = newOutput('dual_eq0', 1:model.N, 1:2, 'nl_eq_dual');
```

9.4 Calling the solver

After code generation has been successful, one can obtain information about the real-time data needed to call the generated solver by typing:

Matlab

Python

```
help FORCESNLPsolver
```

```
# Assuming `solver` is the return value of a `model.generate_solver()` call
solver.help()
```

In Python, a previously generated solver can be loaded as follows:

```
import forcespro.nlp
solver = forcespro.Solver.from_directory("/path/to/generated/solver/")
solver.help()
```

9.4.1 Initial guess

The FORCESPRO NLP solver solves NLPs to local optimality, hence the resulting optimal solution depends on the initialization of the solver. One can also choose another initialization point when a better guess is available. The following code sets the initial point to be in the middle of all bounds:

Matlab

Python

```
x0i = model.lb +(model.ub - model.lb)/2;
x0 = repmat(x0i', model.N, 1);
problem.x0 = x0;
```

```
xi = (model.lb + model.ub) / 2  # assuming lb and ub are numpy arrays
x0 = np.tile(xi, (model.N,))
problem = {"x0": x0}
```

9.4.2 Initial and final conditions

If there are initial and/or final conditions on the optimization variables, the solver will expect the corresponding runtime fields:

Matlab

Python

```
problem.xinit = model.xinit;
problem.xfinal = model.xfinal;
```

```
problem = {"xinit": np.array([1, 2, 3]),
           "xfinal": np.array([4, 5, 6])}
```

Note that the Python client does not allow setting *model.xinit* or *model.xfinal* properties, as those are run-time parameters not needed at solver generation time.

9.4.3 Real-time parameters

Whenever there are any runtime parameters defined in the problem, i.e. the field **npar** is not zero, the solver will expect the following field containing the parameters for all the N stages stacked in a single vector:

Matlab

Python

```
problem.all_parameters = repmat(1.0, model.N, 1);
```

```
problem["all_parameters"] = np.tile(1.0, (model.N,))
```

9.4.4 Tolerances as real-time parameters

From FORCESPRO *2.0* onwards, the NLP solver tolerances can be made real-time parameters, meaning that they do not need to be set when generating the solver but can be changed at run-time when calling the generated solver. The code-snippet below shows how to make the tolerances on the gradient of the Lagrangian, the equalities, the inequalities and the complementarity condition parametric. Essentially, when the tolerances are declared nonpositive at code-generation, the corresponding run-time parameter is created in the solver.

Matlab

Python

```
codeoptions.nlp.TolStat = -1; % Tolerance on gradient of Lagrangian
codeoptions.nlp.TolEq = -1;   % Tolerance on equality constraints
codeoptions.nlp.TolIneq = -1; % Tolerance on inequality constraints
codeoptions.nlp.TolComp = -1; % Tolerance on complementarity
```

```
codeoptions.nlp.TolStat = -1  # Tolerance on gradient of Lagrangian
codeoptions.nlp.TolEq = -1    # Tolerance on equality constraints
codeoptions.nlp.TolIneq = -1  # Tolerance on inequality constraints
codeoptions.nlp.TolComp = -1  # Tolerance on complementarity
```

Once the tolerance has been declared nonpositive and the solver has been generated, the corresponding parameter can be set at run-time as follows:

Matlab

Python

```
problem.ToleranceStationarity = 1e-1;
problem.ToleranceEqualities = 1e-1;
problem.ToleranceInequalities = 1e-1;
problem.ToleranceComplementarity = 1e-1;
```

```
problem["ToleranceStationarity"] = 1e-1
problem["ToleranceEqualities"] = 1e-1
problem["ToleranceInequalities"] = 1e-1
problem["ToleranceComplementarity"] = 1e-1
```

Tip: We do not recommend changing the tolerance on the complementarity condition since it is used internally to update the barrier parameter. Hence loosening it may hamper the solver convergence.

9.4.5 Exitflags and quality of the result

Once all parameters have been populated, the MEX interface of the solver can be used to invoke it:

Matlab

Python

```
[output, exitflag, info] = FORCESNLPsolver(problem);
```

```
output, exitflag, info = solver.solve(problem)
```

The possible exitflags are documented in *Exitflag values*. The exitflag should always be checked before continuing with program execution to avoid using spurious solutions later in the code. Check whether the solver has exited without an error before using the solution. For example, in MATLAB, we suggest to use an assert statement:

Matlab

Python

```
assert(exitflag == 1, 'Some issue with FORCESPRO solver');
```

```
assert exitflag == 1, "Some issue with FORCESPRO solver"
```

Besides the exitflag, the solver also returns an information structure containing detailed KPIs on the solver performance. All its entries are listed and explained in Table 9.1. Those values may also be useful in case the solver exitflag has been `0`, which means the solver did not fail but reached the maximum number of iterations. In that case, one should at least check whether the "solution" returned is sufficiently feasible. This can be done by examining `res_eq` and `res_ineq`, respectively, to check whether equality and inequality constraints are satisfied up to a sufficiently small tolerance. The exact tolerances to check may be strongly application dependent.

Note: Applying a premature "solution" returned along with an exitflag of `0` to control your system may have undesired effects to the behavior of that system. Only do so if you fully understand what you are doing.

Table 9.1: Info struct entries

Fieldname	Description
`it`	Number of solver iterations that led to this result
`res_eq`	Maximum norm of equality constraint residual
`res_ineq`	Maximum norm of inequality constraint residual
`rsnorm`	Maximum norm of stationarity condition
`rcompnorm`	Maximum norm of violations of the complementarity conditions
`pobj`	Primal objective value (as provided by the user)
`mu`	Duality measure
`solvetime`	Time needed to run the solver (wall clock time)
`fevalstime`	Time needed just for function evaluations of all user callbacks inside the solver (wall clock time)
`solver_id`	Solver ID of generated solver

9.5 External function evaluations in C

This approach allows the user to integrate existing efficient C implementations to evaluate the required functions and their derivatives with respect to the stage variable. This gives the user full flexibility in defining the optimization problem. In this case, the functions do not necessarily have to be differentiable, although the convergence of the algorithm is not guaranteed if they are not. When following this route the user does not have to provide MATLAB code to evaluate the objective or constraint functions. However, the user is responsible for making sure that the provided derivatives and function evaluations are coherent. The FORCESPRO NLP code generator will not check this.

9.5.1 Interface

Expected function signature

There are two ways in which a user can supply custom functions written in C:

- Either she can supply all functions (`model.objective`, `model.eq`, `model.ineq` etc.)
- Or she can supply one or a few of these together with its differential/Jacobian.

Depending on the case, the user will have to supply different information when generating a FORCESPRO solver.

When supplying all functions, the user will have supply a single C function having the following signature:

```
int myfunctions (
    double *x, /* primal vars */
    double *y, /* eq. constraint multipliers */
    double *l, /* ineq . constraint multipliers */
    double *p, /* runtime parameters */
    double *f, /* objective function ( incremented in this function ) */
    double *nabla_f , /* gradient of objective function */
    double *c, /* dynamics */
    double *nabla_c , /* Jacobian of the dynamics ( column major ) */
    double *h, /* inequality constraints */
    double *nabla_h , /* Jacobian of inequality constraints ( column major ) */
    double *H, /* Hessian ( column major ) */
    int stage, /* stage number (0 indexed ) */
    int iteration, /* solver iteration count */
    int threadID /* thread id */
)
```

If instead the user wants to supply just a single function she will have to supply a single C function having the following signature:

```
void function (
    const double * const x, /* primal vars */
    const double * const p, /* runtime parameters */
    double * const zeroOrderFcn, /* Zero order function */
    double * const firstOrderFcn , /* first order Fcn */
    const int stage, /* stage number (0 indexed ) */
    const int threadID /* thread number */
)
```

where `zeroOrderFcn` and `firstOrderFcn` denote the function which the user wants to supply, together with its differential/Jacobian respectively. E.g. if the user were to add the objective function and its differential externally, the function might look as follows:

```
void objective (
    const double * const x, /* primal vars */
    const double * const p, /* runtime parameters */
    double * const obj, /* objective function */
    double * const nabla_obj, /* jacobian of objective fcn */
    const int stage, /* stage number (0 indexed ) */
    const int threadID /* thread number */
)
```

Note: External C-functions should have the same name as the file it is contained in, minus the file extension. E.g. in the above example the source file containing the definition of the function `objective` would have to have the name `objective.c`. If all functions are provided as

external C functions through the FORCESPRO Python client, then one can provide a different name for the function and the file.

Custom data structures as parameters

If you have an advanced data structure that holds the user-defined run-time parameters, and you do not want to serialize it into an array of doubles to use the interface above, you can invoke the option:

```
codeoptions.customParams = 1;
```

```
options.customParams = 1
```

When doing this, it is important to note that run-time parameters can only be passed to externally provided functions. In particular, if some but not all function evaluations are provided externally, one will have to set `model.npar = 0`. When using custom parameters, if all functions are provided externally, the expected function signature is:

```
int myfunctions (
    double *x, /* primal vars */
    double *y, /* eq. constraint multipliers */
    double *l, /* ineq . constraint multipliers */
    void *p, /* runtime parameters (passed as void pointer) */
    double *f, /* objective function ( incremented in this function ) */
    double *nabla_f , /* gradient of objective function */
    double *c, /* dynamics */
    double *nabla_c , /* Jacobian of the dynamics ( column major ) */
    double *h, /* inequality constraints */
    double *nabla_h , /* Jacobian of inequality constraints ( column major ) */
    double *H, /* Hessian ( column major ) */
    int stage, /* stage number (0 indexed ) */
    int iteration, /* Solver iteration count */
    int threadID /* Thread id */
)
```

If instead only some of the functions are provided, the expected function signature of these is

```
void function (
    const double * const x, /* primal vars */
    void * const p, /* runtime parameters passed as void pointer */
    double * const zeroOrderFcn, /* Zero order function */
    double * const firstOrderFcn , /* first order Fcn */
    const int stage, /* stage number (0 indexed ) */
    const int threadID /* thread number */
)
```

At run time you can then pass arbitrary data structures to your own function by setting the pointer in the params struct:

```
myData p;   /* define your own parameter structure */
/* ... */   /* fill it with data */

/* Set parameter pointer to your data structure */
mySolver_params params; /* Define solver parameters */
params.customParams = &p;
```

(continues on next page)

(continued from previous page)

```
/* Call solver (assuming everything else is defined) */
mySolver_solve(&params, &output, &info, stdout, &external_func);
```

Note: Setting `customParams` to 1 will disable building high-level interfaces. Only C header- and source files will be generated. As a consequence, if CasADi is used to generate some of the function evaluations, the generated source code will have to be compiled by the user.

Note: When using custom parameters, generating callback code automatically is only supported for CasADi 3.5.x only.

9.5.2 Supplying function evaluation information

In MATLAB, if the user wants to supply all functions externally she can let the code generator know about the path to the C files implementing the necessary function as follows:

```
model.extfuncs = 'C/myfunctions.c';
```

Alternatively she could supply either one of the functions `model.objective`, `model.eq` or `model.ineq` together with its differential by specifying the path to the corresponding C source file in the corresponding field of `model.extfuncs` as follows:

```
model.extfuncs.objective = 'C/myobjective.c'; % adding model.objective externally %
model.extfuncs.dynamics = 'C/mydynamics.c'; % adding model.eq externally %
model.extfuncs.inequalities = 'C/myinequalities.c'; % adding model.ineq externally %
```

As noted above, FORCESPRO derives the function name used for the callback from the file name; the function must therefore have the same name as the file in which it is contained.

In Python, if the user wishes to add ALL functions externally she would use a *ExternalFunctionModel* as follows:

```
model = forcespro.nlp.ExternalFunctionModel(50)
model.add_auxiliary(["helper_functions.c", "compiled_helper_functions.obj"])
model.set_main_callback("myfunctions.c", function="myfunctions")
```

Herein, the *add_auxiliary()* method is used to add any helper C source files or object files that should be compiled and linked against, and the *set_main_callback()* function is used to define the path to a C source file or compiled object file, as well as the name of an exported function that conforms to the call signature given above. This function will be used to evaluate any nonlinear constraints and the objective function.

Alternatively, in order to add a single function externally the user would use the *SymbolicModel* and add the C source files containing code for the external functions through *model.extfuncs* as follows:

```
model = forcespro.nlp.SymbolicModel(50)
model.extfuncs.objective = "myobjective.c"
model.extfuncs.dynamics = "mydynamics.c"
model.extfuncs.inequalities = "myinequalities.c"
model.add_c_source("extra_source.c")
```

where the *extra_source.c* are additional C source files needed for evaluating some or all of the external functions.

9.5.3 Rules for function evaluation code

The contents of the function have to follow certain rules. We will use the following example to illustrate them:

```
/* cost */
if (f)
{ /* notice the increment of f */
    (*f) += -100*x[3] + 0.1* x[0]*x[0] + 0.01*x [1]*x [1];
}
/* gradient - only nonzero elements have to be filled in */
if ( nabla_f )
{
    nabla_f [0] = 0.2*x[0];
    nabla_f [1] = 0.02*x[1];
    nabla_f [3] = -100;
}

/* eq constr */
if (c)
{
    vehicle_dynamics (x, c);
}
/* jacobian equalities ( column major ) */
if ( nabla_c )
{
    vehicle_dynamics_jacobian (x, nabla_c );
}

/* ineq constr */
if (h)
{
    h[0] = x [2]*x[2] + x[3]*x [3];
    h[1] = (x[2]+2)*(x[2]+2) + (x[3] -2.5)*(x[3] -2.5);
}
/* jacobian inequalities ( column major )
- only non - zero elements to be filled in */
if ( nabla_h )
{
    /* column 3 */
    nabla_h [4] = 2*x [2];
    nabla_h [5] = 2*x[2] + 4;
    /* column 4 */
    nabla_h [6] = 2*x [3];
    nabla_h [7] = 2*x[3] - 5;
}
```

Notice that every function evaluation is only carried out if the corresponding pointer is not null. This is used by the FORCESPRO NLP solver to call the same interface with different pointers depending on the functions that it requires.

9.5.4 Matrix format

Matrices are assumed to be stored in dense column major format. However, only the non-zero components need to be populated, as FORCESPRO NLP makes sure that the arrays are initialized to zero before calling this interface.

9.5.5 Multiple source files

The use of multiple C files is also supported. In the example above, the functions `dynamics` and `dynamics_jacobian` are defined in another file and included as external functions using:

```
extern void dynamics ( double *x, double *c);
extern void dynamics_jacobian ( double *x, double *J);
```

In MATLAB, to let the code generator know about the location of these other files use a string with spaces separating the different files. In Python, use the *add_auxiliary()* method:

Matlab

Python

```
codeoptions.nlp.other_srcs = 'C/dynamics.c';
```

```
model.add_auxiliary('C/dynamics.c')
```

9.5.6 Stage-dependent functions

Whenever the cost function in one of the stages is different from the standard cost function f, one can make use of the argument stage to evaluate different functions depending on the stage number. The same applies to all other quantities.

9.5.7 External function return values

By default, FORCESPRO does not check the return value of the main callback function. If you write your external function so that it returns:

- a non-negative value in case of success,
- a negative value between `-99` and `-1` in case of failure,

you can set the FORCESPRO code option `codeoptions.callback_check_status = 1` to make the solver check these return values and stop should a value be negative. In the latter case, the solver returns `ret - 200`, given `ret` as the external function return value. Refer to section *Exitflag values* for a full description of the relevant exitflags. If the callbacks are evaluated concurrently (which is the case if `codeoptions.parallel > 0`), the solver exitflag refers to the most negative return value from external function evaluations.

Note: You need to verify yourself the correctness of the external function signature (as documented *here*) as this can't be verified automatically.

9.6 Calling the nonlinear functions from MATLAB or Python

From FORCESPRO version 5.0.1 onwards it is possible to evaluate the nonlinear functions and their derivatives directly in MATLAB or Python. This can be very useful for debugging a FORCESPRO solver. This in particular can be useful when calling the generated solver returns an exitflag -6, -8 or -10. For help on how to do this, see *Real-time SQP Solver: Robotic Arm Manipulator (MATLAB & Python)* and *Issues when running a generated solver*.

9.6.1 Calling the nonlinear functions from MATLAB

When generating a FORCESPRO solver one can request the FORCESPRO client to also generate a MEX entry point to any of the nonlinear functions (equality constraints, inequality constraints and objective function) so that one can evaluate these along with their derivatives directly in MATLAB. This is done via the following three options:

- `dynamics`: In order to generate a MEX interface for the equality constraints, set `codeoptions.MEXinterface.dynamics = 1`.
- `inequalities`: In order to generate a MEX interface for the inequality constraints, set `codeoptions.MEXinterface.inequalities = 1`.
- `objective`: In order to generate a MEX interface for the objective function, set `codeoptions.MEXinterface.objective = 1`.

The generated MEX entry point carries the name `<solvername>_dynamics`, `<solvername>_inequalities` or `<solvername>_objective` depending on the case.

The first argument to each of these generated MEX entry points is the primal variable z_k and the second argument is the real-time parameter p_k (see *Canonical problem for discrete-time dynamics*), both passed as column vectors. Additionally a third argument can be passed, which indicates the stage (1-based) at which the function should be evaluated. Each of the generated MEX entry points output two variables: The first is the constraint/objective evaluated at the supplied point and the second is its jacobian with respect to z_k. E.g. in order to generate a solver named `FORCESsolver` and generate a MEX entry point for the inequality constraints, one would proceed as follows:

```
model = getModel();
codeoptions = getOptions('FORCESsolver');
codeoptions.MEXinterface.inequalities = 1;
FORCES_NLP(model, codeoptions);
```

Now one can evaluate the inequality constraints on stage 8 at the point $z_8 = (4.0, 3.0, 2.4)^T$ with real-time parameters given by $p_8 = (3.4, 2.1)^T$ by doing

```
z = [4.0; 3.0; 2.4];
p = [3.4; 2.1];
ineq, jacineq = FORCESsolver_inequalities(z,p,8);
```

which stores the values of the inequality constraints in `ineq` and their jacobian in `jacineq`.

Note: If there are no real-time parameters one simply passes an empty array: `ineq, jacineq = FORCESsolver_inequalities(z,[],8)`

9.6.2 Calling the nonlinear functions from Python

When generating a solver one obtains a solver object

```
solver = model.generate_solver(options)
```

or

```
import forcespro.nlp
solver = forcespro.Solver.from_directory('/path/to/solver')
```

and one can directly evaluate any of the nonlinear functions (equality constraints, inequalities or objective function) via the solver object's methods. The methods for doing so are `dynamics`, `ineq` and `objective`. Each of these functions take as a first input the primal variable z_k and the

real-time parameter p_k as a second input, if the problem formulation has real-time parameters (see *Canonical problem for discrete-time dynamics*). All of these inputs are expected to be passed as numpy arrays. Recall that the way in which states and inputs are "organized" into z_k is determined by `model.E`. Additionally a `stage` input can be given in order to compute the dynamics at a given stage (it is not necessary to supply this argument if the same dynamics are used throughout the entire horizon). This `stage` input should give the 0-based number of the corresponding stage. The first output of each of these function is the evaluated constraint/objective and the second output is the derivative with respect to z_k. E.g. in order to evaluate the equality constraints at stage number 7 on the primal variable $z_8 = (4.0, 3.0, 2.4)^T$ in a formulation where there are no real-time parameters, one would do as follows:

```
import numpy as np
import forcespro.nlp

solver = forcespro.Solver.from_directory('/path/to/solver')
z = np.array([4.0, 3.0, 2.4])
c, jacc = solver.dynamics(z, stage=7)
```

This stores the value of the equality constraints in the numpy array `c` and its jacobian in the numpy array `jacc`.

9.7 Mixed-integer nonlinear solver

From FORCESPRO `1.8.0`, mixed-integer nonlinear programs (MINLPs) are supported. This broad class of problems encompasses all nonlinear programs with some integer decision variables.

This interface is provided with *Variant L* of FORCESPRO.

9.7.1 Writing a mixed-integer model

In order to use this feature, the user has to declare lower and upper bounds on **all variables** as parametric, as shown in the code below.

Matlab

Python

```
model.lb = [];
model.ub = [];
```

```
model.lbidx = range(0, model.nvar)
model.ubidx = range(0, model.nvar)
```

The user is then expected to provide lower and upper bounds as run-time parameters. FORCESPRO switches to the MINLP solver as soon as some variables are declared as integers in any stage. This information can be provided to FORCESPRO via the `intidx` array at every stage. An example is shown below.

Matlab

Python

```
%% Add integer variables to existing nonlinear model
for s = 1:5
    model.intidx{s} = [4, 5, 6];
end
```

```
# Add integer variables to existing nonlinear model
for s in range(0, 5):
    model.intidx[s] = [3, 4, 5]
```

In the above code snippet, the user declares variables `4`, `5` and `6` (`3`, `4` and `5` in Python's zero-based indexing) as integers from stage `1` to `5` (stages `0` to `4` in Python's zero-based indexing). The values that can be taken by an integer variable are derived from its lower and upper bounds. For instance, if the variable lies between `-1` and `1`, then it can take integer values `-1`, `0` or `1`. If a variable has been declared as integer and does not have lower or upper bounds, FORCESPRO raises an exception during code generation. Stating that a variable has lower and upper bounds should be done via the arrays `lbidx` and `ubidx`. For instance, in the code below, variables `1` to `6` (`0` to `5` in Python) in stage `1` (`0`) have lower and upper bounds, which are expected to be provided at run-time.

Matlab

Python

```
model.lbidx{1} = 1 : 6;
model.ubidx{1} = 1 : 6;
```

```
model.lbidx[0] = range(0, 6)
model.ubidx[0] = range(0, 6)
```

The FORCESPRO MINLP algorithm is based on the well-known branch-and-bound algorithm but comes with several customization features which generally help for improving performance on some models by enabling the user to provide application specific knowledge into the search process. At every node of the search tree, the FORCESPRO nonlinear solver is called in order to compute a solution of a relaxed problem. The generated MINLP solver code can be customized via the options described in Table 9.2, which can be changed before running the code generation.

One of the salient features of the MINLP solver is that the branch-and-bound search can be run in parallel on several threads. Therefore the search is split in two phases. It starts with a sequential branch-and-bound and switches to a parallelizable process when the number of nodes in the queue is sufficiently high. The node selection strategy can be customized in both phases, as described in Table 9.2.

Table 9.2: FORCESPRO MINLP solver options

Code generation setting	Values	Default
`minlp.int_gap_tol`	Any value ≥ 0	`0.001`
`minlp.max_num_nodes`	Any value ≥ 0	`10000`
`minlp.seq_search_strat`	`'BEST_FIRST'`, `'BREADTH_FIRST 'DEPTH_FIRST'`	`'BEST_FIRST'`
`minlp.par_search_strat`	`'BEST_FIRST'`, `'BREADTH_FIRST'`, `'DEPTH_FIRST'`	`'BEST_FIRST'`
`minlp.max_num_threads`	Any nonnegative value preferably smaller than 8	`4`
`minlp.output_relaxation`	0 or 1	0

- The `minlp.int_gap_tol` setting corresponds to the final optimality tolerance below which the solver is claimed to have converged. It is the difference between the objective incumbent, which is the best integer feasible solution found so far and the lowest lower bound. As the node problems are generally not convex, it can be expected to become negative. FORCESPRO claims convergence to a local minimum only when the integrality gap is nonnegative and below the tolerance `minlp.int_gap_tol`.
- The `minlp.max_num_nodes` setting is the maximum number of nodes which can be explored during the search.
- The `minlp.seq_search_strat` setting is the search strategy which is used to select candidate nodes during the sequential search phase.

- The `minlp.par_search_strat` setting is the search strategy which is used to select candidate nodes during the parallelizable search phase.
- The `minlp.max_num_threads` setting is the maximum number of threads allowed for a parallel search. The actual number of threads on which the branch-and-bound algorithm can be run can be set as a run-time parameter, as described below.
- The `minlp.output_relaxation` setting enables users to export the primal outputs of the root relaxation. With this option set to `1`, the server automatically generates one additional output for every defined output. The name of the root relaxation output is the name of the output followed by `_relax`.

Note: MINLP formulations are currently not supported on MacOS. Moreover, they cannot be used in combination with an SQP algorithm (see *Sequential quadratic programming algorithm*) nor with CasADi MX expressions (see *Automatic differentiation expression class*).

The FORCESPRO MINLP solver also features settings which can be set at run-time. These are the following:

- `minlp.numThreadsBnB`, the number of threads used to parallelize the search. Its default value is *1*, if not provided by the user.
- `minlp.solver_timeout`, the maximum amount of time allowed for completing the search. Its default value is *10.0* seconds, if not set by the user.
- `minlp.parallelStrategy`, the method used for parallelizing the mixed-integer search (from FORCESPRO 1.9.0). Value *0* (default) corresponds to a single priority queue shared between threads. Value *1* corresponds to having each thread managing its own priority queue.

9.7.2 Mixed-integer solver customization via user callbacks

For advanced users, the mixed-integer branch-and-bound search can be customized after the rounding and the branching phases. In the rounding phase, an integer feasible solution is computed after each relaxed problem solve. The user is allowed to modify the rounded solution according to some modelling requirements and constraints. This can be accomplished via the `postRoundCallback_template.c` file provided in the FORCESPRO client. This callback is applied at every stage in a loop and updates the relaxed solution stage-wise. It needs to be provided before code generation, as shown in the following code snippet.

Matlab

Python

```
%% Add post-rounding callback to existing model
postRndCall = fileread('postRoundCallback_template.c'); % The file name can be␣
↪changed by the user
model.minlpPostRounding = postRndCall;
```

```
with open('postroundCallback_template.c') as f:
  model.minlpPostRounding = f.read()
```

The branching process can be customized in order to discard some nodes during the search. To do so, the user is expected to overwrite the file `postBranchCallback_template.c` and pass it to FORCESPRO before generating the MINLP solver code.

Matlab

Python

```
%% Add as post-branching callbacks as you want
postBranchCall_1 = fileread('postBranchCallback_template_1.c');
postBranchCall_2 = fileread('postBranchCallback_template_2.c');
postBranchCall_3 = fileread('postBranchCallback_template_3.c');
model.minlpPostBranching{1} = postBranchCall_1;
model.minlpPostBranching{2} = postBranchCall_2;
model.minlpPostBranching{3} = postBranchCall_3;
```

```
# Add as post-branching callbacks as you want
with open('postBranchCallback_template_1.c') as f:
    model.minlpPostBranching[0] = f.read()
with open('postBranchCallback_template_2.c') as f:
    model.minlpPostBranching[1] = f.read()
with open('postBranchCallback_template_3.c') as f:
    model.minlpPostBranching[2] = f.read()
```

In each of those callbacks, the user is expected to update the lower and upper bounds of the sons computed during branching given the index of the stage in which the branched variables lies, the index of this variable inside the stage and the relaxed solution at the parent node.

9.7.3 Providing a guess for the incumbent

Internally, the mixed-integer branch-and-bound computes an integer feasible solution by rounding. Moreover, since version `1.9.0`, users are allowed to provide an initial guess for the incumbent. At code-generation, the following options need to be set:

- `minlp.int_guess`, which tells whether an integer feasible guess is provided by the user (value *1*). Its default value is *0*.
- `minlp.int_guess_stage_vars`, which specifies the indices of the integer variables that are user-initialized within one stage (MATLAB based indexing). If `minlp.int_guess = 1`, a parameter `int_guess` needs to be set at every stage. An example can be found there *Mixed-integer nonlinear solver: F8 Crusader aircraft*.

Another important related option is `minlp.round_root`. If set to `1`, the solution of the root relaxation is rounded and set as incumbent if feasible. Its default value is `1`. The mixed-integer solver behavior differs depending on the combinations of options. The different behaviors are listed below.

- If `minlp.int_guess = 0` and `minlp.round_root = 1`, then the solution of the root relaxation is taken as incumbent (if feasible). This is the default behavior.
- If `minlp.int_guess = 1` and `minlp.round_root = 0`, then the incumbent guess provided by the user is tested after the root solve. If feasible, it is taken as incumbent. Note that the user is allowed to provide guesses for a few integers per stage only. In this case, the other integer variables are rounded to the closest integer.
- If `minlp.int_guess = 1` and `minlp.round_root = 1`, then the rounded solution of the root relaxation and the user guess are compared. The best integer feasible solution in terms of primal objective is then taken as incumbent.

This feature is illustrated in Example *Mixed-integer nonlinear solver: F8 Crusader aircraft*. The ability of providing an integer guess for the incumbent is a key feature to run the mixed-integer solver in a receding horizon setting.

9.8 Sequential quadratic programming algorithm

The FORCESPRO real-time sequential quadratic programming (SQP) algorithm allows one to solve problems of the type specified in the section *High-level Interface*. The algorithm iteratively solves a convex quadratic approximations of the (generally non-convex) problem. Moreover, the solution is stored internally in the solver and used as an initial guess for the next time the solver is called. This and other features enables the solver to have fast solvetimes (compared to the interior point method), particularly suitable for MPC applications where the sampling time or the computational power of the hardware is small.

Important: The SQP algorithm currently only supports affine inequalities. This means that all the inequality functions $h_k, k = 1, ..., N$ from (9.1.1) must be *affine* functions of the variable z_k (not necessarily of p_k).

Moreover, the SQP algorithm currently does not support problems comprising final equality constraints (specified via `model.xfinalidx`).

9.8.1 How to generate a SQP solver

To generate a FORCESPRO sequential quadratic programming real-time iteration solver one sets

Matlab

Python

```
codeoptions.solvemethod = 'SQP_NLP';
```

```
codeoptions.solvemethod = "SQP_NLP"
```

(see *Generating a solver*). In addition to the general code options specified in the previous section here are some of the important code options one can use to customize the generated SQP solver.

By default the FORCESPRO SQP solver solves a single convex quadratic approximation. This accomplishes a fast solvetime compared to a "full" sequential quadratic programming solver (which solves quadratic approximations to the nonlinear program until a KKT point is reached). The user might prefer to manually allow the SQP solver to solve multiple quadratic approximations: By setting

Matlab

Python

```
codeoptions.sqp_nlp.maxqps = k;
```

```
codeoptions.sqp_nlp.maxqps = k
```

for a positive integer `k` one allows the solver to solve `k` quadratic approximations at every call to the solver. In general, the more quadratic approximations which are solved, the higher the control performance. The tradeoff is that the solvetime also increases.

9.8.2 Different SQP variants

In addition to the default FORCESPRO SQP solver (also referred to as *SQP General* here), FORCESPRO supports a new *SQP Fast* solver from version 6.0.0. An advantage of the fast

SQP solver is that it is typically faster and can be made to run on more low-level hardware. An advantage of the general (default) SQP solver is that it is very robust. See *Tuning the SQP Fast solver* for a detailed discussion on how to generate a SQP Fast solver and see *High-level interface: Path tracking example (MATLAB)* for an example of how to use the SQP Fast solver.

Note: The SQP Fast solver is only supported when using a Gauss-Newton hessian approximation and disabling the line search (see *The hessian approximation and line search settings*). The SQP Fast solver is currently only supported via the MATLAB client of FORCESPRO.

9.8.3 Tuning the SQP Fast solver

One of the novel features of the fast SQP solver is that it can be tuned/tailored to achieve optimal performance on a specific application. FORCESPRO provides a tool for "autotuning" the solver (see *Autotuner*). The main step in using this tool is to collect training problems which can be used to tune the solver. The standard way to do this is to first generate an SQP General (`codeoptions.sqp_nlp.qp_method = "general"`, which is also the default value) solver, perform a closed-loop simulation with this solver while caching/saving all problem instances (i.e. the inputs to the solver at run-time). In a second step a SQP Fast solver is generated and automatically tuned by calling `ForcesGenerateSqpFastSolver`. The complete workflow is as follows:

Matlab

```
tuningoptions = ForcesAutotuneOptions(problems)
ForcesGenerateSqpFastSolver(model, codeoptions, tuningoptions);
```

where `problems` is a cell array of problems collected from the simulation with the SQP General solver. The autotuning procedure allows for quite a bit of customization which can be specified through the `ForcesAutotuneOptions` object (see *Autotuner Options*), and one can also pass multiple problems for the tuning process which should result in better tuned parameters. See *High-level interface: Path tracking example (MATLAB)* for an example of how to tune the SQP Fast solver.

9.8.4 The hessian approximation and line search settings

The SQP code generation currently supports two different types of hessian approximations. A good choice of hessian approximation can often improve the number of iterations required by the solver and thereby its solvetime. The default option for a SQP solver is the BFGS hessian approximation. When the objective function of the optimization problem is a least squares cost it is often beneficial to use the Gauss-Newton hessian approximation instead. To enable this option one proceeds as specified in the sections *Hessian approximation* and *Gauss-Newton options*. When the Gauss-Newton hessian approximation is chosen one can also disable the the internal linesearch by setting

Matlab

Python

```
codeoptions.sqp_nlp.use_line_search = 0;
```

```
options.sqp_nlp.use_line_search = False
```

A linesearch is required to ensure global convergence of an SQP method, but is not needed in a real-time context when a Gauss-Newton hessian approximation is used.

Note: One cannot disable the line search when using the BFGS hessian approximation.

9.8.5 Controlling the initial guess at run-time

Upon the first call to the generated FORCESPRO SQP solver one needs to specify a primal initial guess (`problem.x0`, see also *Initial guess*). The default behavior of the FORCESPRO SQP solver is to use the solution from the previous call as initial guess in every subsequent call to it. However, one can also manually set an initial guess in subsequent calls to the solver. Wether a manual initial guess (provided through `problem.x0`) will be used or the internally stored solution from the previous call will be used can be controlled by the field `problem.reinitialize` of the `problem` struct which is passed as an argument to the solver when it is called.

The `reinitialize` field can take two values: 0 or 1. For the default usage of the solver

Matlab

Python

```
problem.reinitialize = 0;
```

```
problem["reinitialize"] = False
```

should be used. This choice results in the solver using the solution from the previous call as initial guess. This feature is useful when running the real-time iteration scheme because it ensures that the initial guess is close to the optimal solution. If you want to specify an initial guess at run-time, you will need to set

Matlab

Python

```
problem.reinitialize = 1;
```

```
problem["reinitialize"] = True
```

So in summary: The first time the solver is called the initial guess the solver will use has to be provided by `problem.x0`. In all subsequent calls the solver will only make use of `problem.x0` as its initial guess if `problem.reinitialize = 1`.

9.8.6 Additional code options specific to the SQP-RTI solver

In addition to the above codeoptions, the following options are specific to the SQP algorithm. Each of these options can be supplied when generating a solver as a field of `codeoptions.sqp_nlp` (e.g. `codeoptions.sqp_nlp.TolStat`).

Table 9.3: SQP specific codeoptions

option	Possible values	Default value	Description
TolStat	positive	10^{-6}	Set the stationarity tolerance required for terminating the algorithm (the tolerance required to claim convergence to a KKT point).
TolEq	positive	10^{-6}	Set the feasibility tolerance required for terminating the algorithm (the tolerance required to claim convergence to a feasible point).
reg_hessian	positive	$5 \cdot 10^{-9}$	Set the level of regularization of the hessian approximation (often increasing this parameter can help if the SQP solver returns exitflag -8 for your problem)
qpinit	0 or 1	0	Set the initialization strategy for the internal QP solver. 0 = cold start and 1 = centered start. See also *Solver Initialization* (note however, that for the SQP solver `qpinit=2` is not possible).

In addition to these options one can also specify the maximum number of iterations the internal QP solver is allowed to run in order to solve the quadratic approximation. If one wishes the QP solver use no more than `k` iterations to solve a problem one sets

```
codeoptions.maxit = k;
```

Note: The SQP fast solver currently does not support parallel execution, i.e. setting `codeoptions.parallel` will have no effect.

9.9 Differences between the MATLAB and the Python client

The Python NLP interface is largely similar to the MATLAB interface, but does come with some language- and implementation-specific differences.

- All indices in the problem formulation are expected to be 0-based in Python, as is usual in this language. This does not include the indices of the generated solver, however, where outputs are named *x01, x02, ...* as in MATLAB. Thus, the problem formulation *before generation* requires 0-based indices, whereas the returned solver from the server uses 1-based indices. This also does not apply to the low-level Python interface, where indices are 1-based even in the model formulation.
- In the Python client, different model objects must be used when using external functions or symbolic expressions, namely *nlp.ExternalFunctionModel()* and *nlp.SymbolicModel()*. Furthermore, if the high-level interface is to be used for convex problems, this is only possible using the *nlp.ConvexSymbolicModel()*. This is different from the MATLAB client, where the *FORCES_NLP* function accepts problems of any kind and switches to the appropriate solver automatically.
- When using the Python client with a *nlp.SymbolicModel()*, the C code generated for symbolic expressions is currently not entirely identical to the code generated by MATLAB. While the actual expression evaluation code generated by CasADi is the same, the structure of the files varies. Specifically, the MATLAB client creates individual C files for each problem stage with distinct symbolic expressions (leading to varying file names when changing the problem horizon) whereas all functions are gathered in one

file in the Python client. Yet, the Python client does add one additional file for the FORCESPRO-CasADi glue code, which is not present when using the MATLAB client. Lastly, function names of the evaluation functions differ.

If you want to get the same code for MATLAB and Python, you must generate the CasADi C code from one of both clients and then supply this code as an external function in the other client.

9.10 Examples

- *High-level interface: Basic example*: In this example, you learn the basics in how to use FORCESPRO to create an MPC regulation controllers.
- *High-level interface: Obstacle avoidance (MATLAB & Python)*: This example uses a simple nonlinear vehicle model to illustrate the use of FORCESPRO for real-time trajectory planning around non-convex obstacles.
- *High-level interface: Indoor localization (MATLAB & Python)*: This examples describes a nonlinear optimization approach for the indoor localization problem.
- *Mixed-integer nonlinear solver: F8 Crusader aircraft*: In this example, you learn the basics in how to use FORCESPRO MINLP solver to solve a mixed-integer optimal control problem.
- *Real-time SQP Solver: Robotic Arm Manipulator (MATLAB & Python)*: This example describes how to apply the FORCESPRO SQP solver to control a robotic arm.
- *Controlling a DC motor using a FORCESPRO SQP solver*: This example describes how to apply the FORCESPRO SQP solver to control a DC motor.
- *Controlling a crane using a FORCESPRO NLP solver*: This example describes how to apply the FORCESPRO interior point NLP solver to control a crane.
- *High-level interface: Optimal EV charging and speed profile example (MATLAB & PYTHON)*: This example describes how to apply the FORCESPRO interior point NLP solver to control the speed and charging profile of an electric vehicle.

Chapter 10

Simulating your custom controller in Simulink®

FORCESPRO provides Simulink® interfaces for easy simulation of your custom controllers within existing Simulink® diagrams. Once the solver has been generated and compiled, the appropriate Simulink® block and the necessary code for the Simulink® block definition are available along with your FORCESPRO solver inside the `interface` folder. Depending on your controller configuration you will have different input and output ports on your block. The port labels are self-explanatory. Just wire the ports of the FORCESPRO block to other blocks in your Simulink® diagram and run the simulation. Note that several instances of the FORCESPRO block can exist in the same Simulink® diagram.

The FORCESPRO solvers support two Simulink® interfaces:

- The *S-Function* interface
- The *Coder* interface

10.1 The S-Function interface

The S-function interface consists of:

- a Simulink® block definition of a level 2 S-Function (in *interface* folder);
- the C implementation of the S-Function (in *interface* folder) which defines how the FORCESPRO solver is used in the context of the Simulink® block.

The Simulink® block definition is used for the inclusion of the FORCESPRO Simulink® block in a Simulink® model. The C implementation is either built into a MEX interface to be used in the simulation of a Simulink® model or used in the build process of the code generation of a Simulink® model using the *Simulink® Coder*.

10.2 The Coder interface

The Coder interface consists of:

- a MATLAB Simulink® block (built-in in Simulink) along with the MATLAB code (in *interface* folder);
- a *Buildable* (*coder.ExternalDependency*) class (in *interface* folder);
- a FORCESPRO Simulink® block creation script (in *interface* folder).

The MATLAB Simulink® block is used for the inclusion of the FORCESPRO Simulink® block in a Simulink® model. The MATLAB code contains the code that the MATLAB Simulink® block will run in order to use the FORCESPRO solver. The Buildable class is used as the intermediate interface between the MATLAB Simulink® block (with the defined MATLAB code) and the FORCESPRO solver. Additionally, it is used to configure the build process of the Simulink® model. The FORCESPRO Simulink® block creation script handles the generation of the ready-to-use FORCESPRO Simulink® block from the rest components of the interface as well as defining the necessary properties for the block.

10.3 S-Function vs Coder interface

The two interfaces share a lot of similarities as their common purpose is to be used to perform a Simulink® model simulation/execution and as such any of the two can be used. However, there are also certain differences that distinguish them and make each of them either more or less suitable for specific use cases.

In more detail:

- The S-Function interface does not support in-lining of the generated code while the Coder interface does. As a result, the Coder interface can generate faster code compared to the S-Function interface.
- The Coder interface can only support a Fixed-Step solver for the Simulink® model, while the S-Function interface can support both a Fixed-Step solver and a Variable-Step solver.
- The S-Function interface requires custom configuration for it to be integrated into the build process of the Simulink® model code generation, while the Coder interface, other than the inclusion of the Simulink® block in the Simulink® model, handles all dependencies automatically.
- The Coder interface requires the MATLAB Coder Toolbox for a Simulink® model simulation while the S-Function interface can work without other dependencies. For code generation of a Simulink® block, the Simulink® Coder Toolbox is already assumed to be available, therefore both interfaces' requirements are fulfilled.

10.4 FORCESPRO Simulink® blocks

10.4.1 Getting Started - Basic MPC Regulation State Feedback Example

- *Generating the FORCESPRO solver*
- *Workflow for S-Function interface*
- *Workflow for Coder interface*
- *Simulation and checking of results*

This example will show how to get started with the Simulink® interface(s) of FORCESPRO by designing an MPC regulator for the system below.

You can find the MATLAB code of this example to try it out for yourself in the `examples` folder that comes with your client. This example can be used to show the workflow both for the S-Function interface and the Coder interface.

Generating the FORCESPRO solver

The state-space model of the system has been obtained by discretizing the continuous model with a sampling time of 0.1 seconds, and can be described by

$$x_{k+1} = \begin{pmatrix} 0.7115 & -0.4345 \\ 0.4345 & 0.8853 \end{pmatrix} x_k + \begin{pmatrix} 0.2173 \\ 0.0573 \end{pmatrix} u_k$$
$$y_k = \begin{pmatrix} 0 & 1 \end{pmatrix} x_k$$

In addition to the task of steering the two states to zero, there are constraints on the single actuator u and on the second state x_2. We require that the actuator u does not exceed $[-5, 5]$ and the state $x_2 \geq 0$ for all time steps.

After downloading the files we can start with the design of the controller. First open and run the script **examples/Matlab/SimulinkInterface/generateMyFirstController.m**. This script will define the underlying MPC problem and generate the FORCESPRO solver.

First we define the MPC problem parameters, i.e. the state transition matrix A and the input matrix B. The output matrix C and the feed-forward matrix D are not needed for solver generation, but are used in the Simulink® workspace to define the state-space parameters of the dynamical system block. Notice that we use the identity output matrix, since we want to regulate both states and not just the output of the system.

Matlab

```
% MPC parameter setup

% system
A = [0.7115 -0.4345; 0.4345 0.8853];
B = [0.2173; 0.0573];
[nx,nu] = size(B);
C = eye(nx);
D = zeros(nx,1);

% horizon length
N = 10;

% weights in the objective
R = eye(nu);
Q = 10*eye(nx);

% bounds
umin = -5;      umax = 5;
xmin = [-inf, 0]; xmax = [+inf, +inf];
```

We choose a prediction horizon of 10 steps, i.e. the controller looks 1 second into the future. We define the relative weights in the objective function by prioritizing the importance of regulating the states 10 times higher than reducing the use of the actuator. You are encouraged to change these weights and observe the effect on the control behavior.

We also input the details of the constraints described above. The second state must remain positive, whereas the first state is left unconstrained. We also have a constraint on the actuator, with the lower bound -5 and the upper bound 5.

Next, we define the **model** *struct* describing the multistage MPC problem formulation. This includes the model dynamics (i.e. equality constraints), as well as the bounds on state and control variables in the form of inequality constraints.

Matlab

```
% FORCESPRO multistage form
% assume variable ordering zi = [ui; xi] for i=1...N

% dimensions
model.N     = N;        % horizon length
model.nvar  = nx+nu;    % number of variables
model.neq   = nx;       % number of equality constraints

% objective
model.objective = @(z) z(1)*R*z(1) + [z(2);z(3)]'*Q*[z(2);z(3)];

% equalities
model.eq = @(z) [ A(1,:)*[z(2);z(3)] + B(1)*z(1);
                        A(2,:)*[z(2);z(3)] + B(2)*z(1)];

model.E = [zeros(2,1), eye(2)];

% inequalities
model.lb = [umin, xmin];
model.ub = [umax, xmax];

% initial state
model.xinitidx = 2:3;
```

Finally, we generate the FORCESPRO solver called *myFirstController*. Since we will incorporate the solver in the Simulink® model, we confine the output to the first element of the solution vector at stage 1, corresponding to the actuator control action of interest.

Matlab

```
% Generate FORCESPRO solver

% get options
codeoptions = getOptions('myFirstController');
codeoptions.printlevel = 2;
codeoptions.BuildSimulinkBlock = 1;

% generate code
output1 = newOutput('u0', 1, 1);
FORCES_NLP(model, codeoptions, output1);
```

This will send a request to the server which will generate a custom controller for your problem. The code is downloaded to your machine and the appropriate Simulink® block and Simulink® S-function can be found inside the `interface` folder.

We also provide additional initialization parameters needed for the Simulink® model.

Matlab

```
% Model Initialization (needed for Simulink® only)

% initial conditions
x_init = [0; 6];

% initial guess
x0 = zeros(model.N*model.nvar,1);
```

We can now open the Simulink® model `myFirstController_sim.slx` (by copying the `myFirstController_sim_template.slx` file) and incorporate the generated solver into the Simulink® diagram.

Workflow for S-Function interface

In this example we will use the compact FORCESPRO Simulink® block `myFirstControllercompact_lib.mdl` (see *Real-time control with the Simulink® block* for more details). We open `myFirstControllercompact_lib.mdl` and copy the FORCESPRO Simulink® block to the `myFirstController_sim.slx` Simulink® model.



Figure 10.1: Preview of the generated FORCESPRO Simulink® block.

The port labels are self-explanatory. All we have to do is wire the ports of the FORCESPRO block to other blocks in your Simulink® diagram and run the simulation.

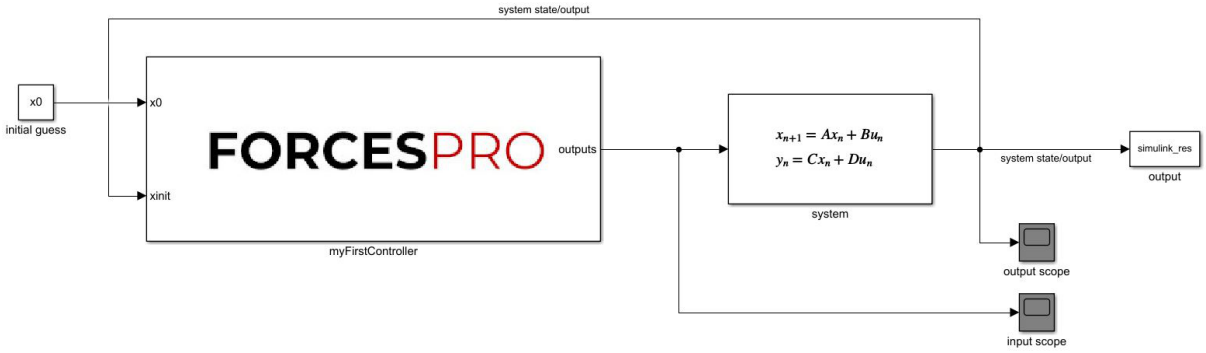


Figure 10.2: Simulink® diagram of the model with inclusion of the FORCESPRO Simulink® block.

Workflow for Coder interface

We change the Solver of the Simulink® model to `Fixed-Step` as the Coder interface does not support `Variable-Step`.

Matlab

```
% the coder interface only supports Fixed-Step Simulations
set_param('myFirstController_sim', 'Solver', 'FixedStep');
```

In this example we will use the compact FORCESPRO Simulink® block. To add the Simulink® block to the `myFirstController_sim.slx` Simulink® model we run the `myFirstController_createCoderMatlabFunction.m` from the `interface` folder.

Matlab

```
% add the interface folder to the path
addpath(fullfile('myFirstController', 'interface'));

% for the coder case run myFirstController_createCoderMatlabFunction in
% myFirstController/interface to create a FORCESPRO Simulink® block to
% the Simulink® model:
%     * The first parameter is the target Simulink® model
%     * The second parameter is the name of the created Simulink® model
```

(continues on next page)

(continued from previous page)

```
%     * The third parameter selects whether to use the standard or the
%       compact version
%     * The fourth parameter must be set to true if the Simulink® model
%       already exists (by default the script always tries to create a
%       new Simulink® model)
myFirstController_createCoderMatlabFunction('myFirstController_sim',
↪'myFirstController', true, true);

% (optionally) remove the interface folder from the path again
rmpath(fullfile('myFirstController', 'interface'));
```

Simulation and checking of results

From the left plot we can see that the actuator remains in the allowed range. The right plot shows how the second state x_2 is always non-negative (purple graph in the right plot) and both states are regulated to zero.

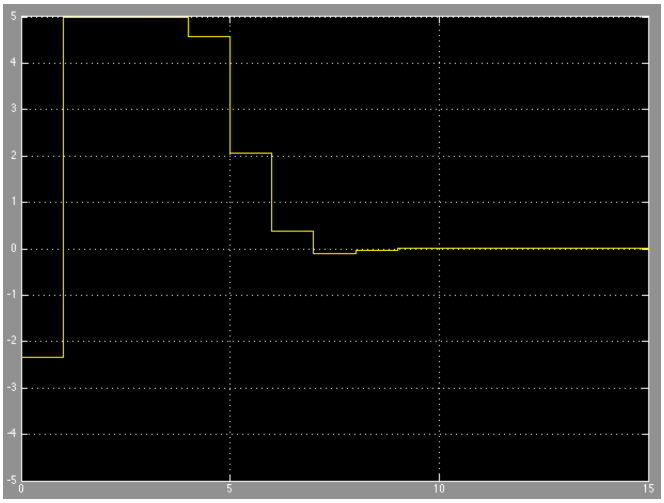


Figure 10.3: Actuator control signal.

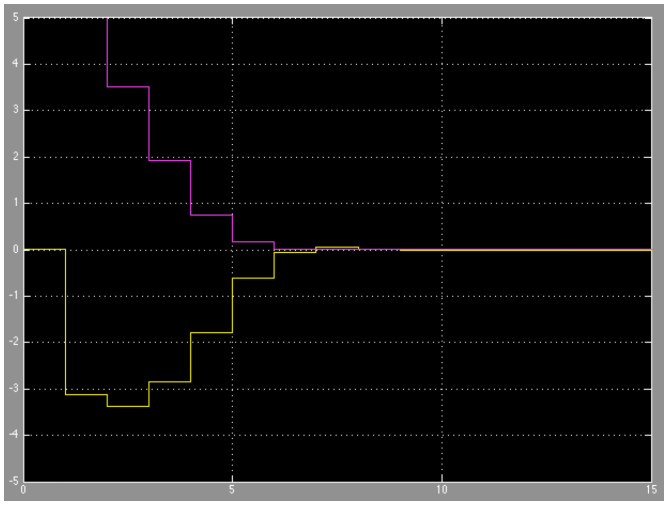


Figure 10.4: State evolution of the system.

10.4.2 Real-time control with the Simulink® block

- *Input and Output Ports in the Compact Interface*

When a user generates a new solver, a Simulink® block becomes available in the **interfaces** folder. Such block is useful to interface the solver with other Simulink® models for simulation, or for deployment to embedded prototyping hardware using tools such as dSPACE MicroAutobox or Simulink® Coder.

Input and Output Ports in the Compact Interface

For every solver, there are two Simulink® variants generated: a standard variant; and a compact variant, which groups parameters and outputs. For problems with many parameters and outputs, the compact variant is more suitable because it reduces the number of ports and connections that need to be wired up to the rest of the Simulink® model.

The criteria for grouping parameters is the following: parameters of the same type that have the same number of rows are grouped together into a single stacked parameter. These parameters are stacked horizontally, e.g. if there are two parameters mapping to **eq.c**, both of size 3x1, they will be grouped into a new parameter of size 3x2. The new parameter will get the name **c**.

To illustrate the conversion, consider a problem with the following parameters and with the corresponding standard (non-compact) Simulink® block:

Name	maps2data	Dimensions
Amat1	eq.D	2x4
Amat2	eq.D	3x4
Amat3	eq.D	3x4
Amat4	eq.D	3x4
linterm1	cost.f	4x1
linterm2	cost.f	4x1
linterm3	cost.f	4x1
linterm4	cost.f	4x1



For the compact Simulink® block, parameters **linterm1**, **linterm2**, **linterm3** and **linterm4** are stacked together into a new parameter **f** (because the problem data they map to is **cost.f**). For the parameters mapping to **eq.D**, **Amat2**, **Amat3** and **Amat4** can be stacked into the new parameter **D**. **Amat1** is not included into the new parameter because it has two rows and the concatenation is not possible with the other parameters, which all have three rows. Parameters are always stacked horizontally according to the stage number they map to.

Name	maps2data	Dimensions
`D`	`eq.D`	3x12
`f`	`cost.f`	4x4
`Amat1`	`eq.D`	2x4



The port dimensions of any FORCESPRO Simulink® block can be checked by double-clicking the block and clicking the 'Help' button.

Chapter 11

Examples

11.1 How to

11.1.1 Basic Example

Consider the following linear MPC problem with lower and upper bounds on state and inputs, and a terminal cost term:

$$\begin{aligned} \text{minimize} \quad & x_N^\top P x_N + \sum_{i=0}^{N-1} x_i^\top Q x_i + u_i^\top R u_i \\ \text{subject to} \quad & x_0 = x \\ & x_{i+1} = A x_i + B u_i \\ & \underline{x} \leq x_i \leq \bar{x} \\ & \underline{u} \leq u_i \leq \bar{u} \end{aligned}$$

This problem is parametric in the initial state x and the first input u_0 is typically applied to the system after a solution has been obtained. The following code generates a function that takes $-Ax$ as a calling argument and returns u_0, which can then be applied to the system.

Here is the MATLAB code:

```
%% FORCES multistage form
% assume variable ordering zi = [u{i-1}, x{i}] for i=1...N

stages = MultistageProblem(N); % get stages struct of length N

for i = 1:N

        % dimension
        stages(i).dims.n = nx+nu; % number of stage variables
        stages(i).dims.r = nx;    % number of equality constraints
        stages(i).dims.l = nx+nu; % number of lower bounds
        stages(i).dims.u = nx+nu; % number of upper bounds

        % cost
        if( i == N )
                stages(i).cost.H = blkdiag(R,P); % terminal cost (Hessian)
        else
                stages(i).cost.H = blkdiag(R,Q);
        end
        stages(i).cost.f = zeros(nx+nu,1); % linear cost terms
```

(continues on next page)

(continued from previous page)

```
        % lower bounds
        stages(i).ineq.b.lbidx = 1:(nu+nx); % lower bound acts on these indices
        stages(i).ineq.b.lb = [umin; xmin]; % lower bound for this stage variable

        % upper bounds
        stages(i).ineq.b.ubidx = 1:(nu+nx); % upper bound acts on these indices
        stages(i).ineq.b.ub = [umax; xmax]; % upper bound for this stage variable

        % equality constraints
        if( i < N )
                stages(i).eq.C = [zeros(nx,nu), A];
        end
        if( i>1 )
                stages(i).eq.c = zeros(nx,1);
        end
        stages(i).eq.D = [B, -eye(nx)];
end

% RHS of first eq. constr. is a parameter: stages(1).eq.c = -A*x0
params(1) = newParam('minusA_times_x0',1,'eq.c');
```

You can find the MATLAB code of this example to try it out for yourself in the **examples** folder that comes with your client.

And here's the Python code:

```
# FORCESPRO multistage form
# assume variable ordering zi = [u{i-1}, x{i}] for i=1...N

stages = forcespro.MultistageProblem(N) # get stages struct of length N

for i in range(N):

        # dimensions
        stages.dims[ i ]['n'] = nx+nu  # number of stage variables
        stages.dims[ i ]['r'] = nx     # number of equality constraints
        stages.dims[ i ]['l'] = nx+nu  # number of lower bounds
        stages.dims[ i ]['u'] = nx+nu  # number of upper bounds

        # cost
        if ( i == N-1 ):
                stages.cost[ i ]['H'] = np.vstack((np.hstack((R,np.zeros((nu,nx)))),
↪np.hstack((np.zeros((nx,nu)),P))))
        else:
                stages.cost[ i ]['H'] = np.vstack((np.hstack((R,np.zeros((nu,nx)))),
↪np.hstack((np.zeros((nx,nu)),Q))))
        stages.cost[ i ]['f'] = np.zeros((nx+nu,1)) # linear cost terms

        # lower bounds
        stages.ineq[ i ]['b']['lbidx'] = range(1,nu+nx+1) # lower bound acts on these␣
↪indices
        stages.ineq[ i ]['b']['lb'] = np.concatenate((umin,xmin),0) # lower bound for␣
↪this stage variable

        # upper bounds
        stages.ineq[ i ]['b']['ubidx'] = range(1,nu+nx+1)  # upper bound acts on␣
```

(continues on next page)

(continued from previous page)

```
↪these indices
        stages.ineq[ i ]['b']['ub'] = np.concatenate((umax,xmax),0) # upper bound for␣
↪this stage variable

        # equality constraints
        if ( i < N-1 ):
                stages.eq[i]['C'] = np.hstack((np.zeros((nx,nu)),A))
        if ( i>0 ):
                stages.eq[i]['c'] = np.zeros((nx,1))
        stages.eq[i]['D'] = np.hstack((B,-np.eye(nx)))


# RHS of first eq. constr. is a parameter: stages(1).eq.c = -A*x0
stages.newParam('minusA_times_x0', [1], 'eq.c')
# define output of the solver
stages.newOutput('u0', 1, range(1,nu+1))
```

11.1.2 How to Incorporate Preview Information in the MPC Problem

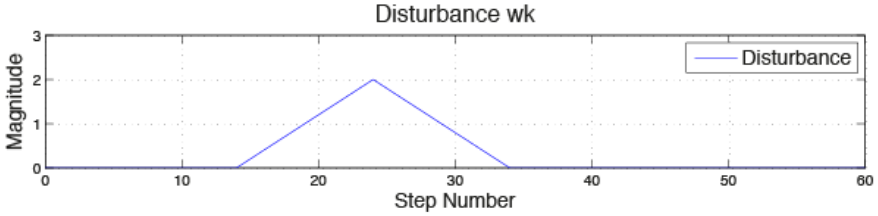
- *Introduction*
- *Use preview information in the MATLAB® interface*
- *Comparison of MPC with Preview and Standard MPC*

Introduction

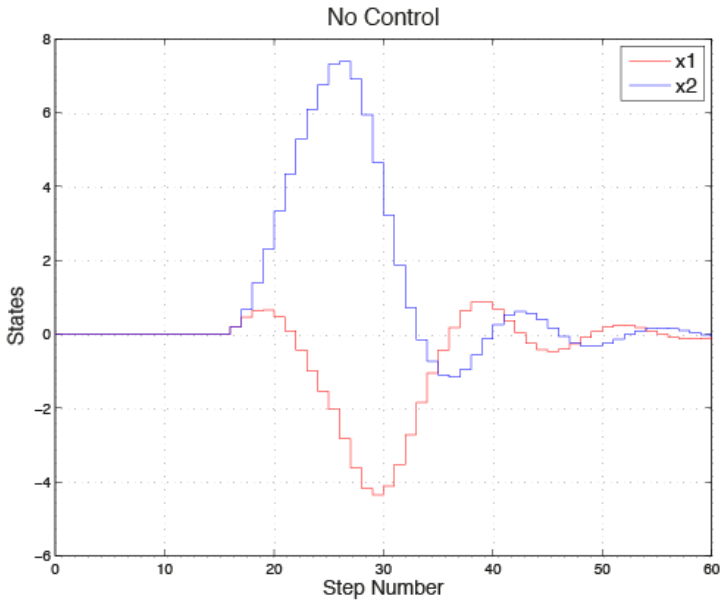
In this example the following discrete-time system is considered:

$$x_{k+1} = \begin{pmatrix} 0.7115 & -0.4345 \\ 0.4345 & 0.8853 \end{pmatrix} x_k + \begin{pmatrix} 1 \\ 1 \end{pmatrix} u_k + \begin{pmatrix} 1 \\ 1 \end{pmatrix} w_k$$

The control objective is to regulate the two states to zero using the input u_k, while a disturbance w_k is acting on the system. The disturbance w_k gets predicted for a horizon of length $N = 10$, which is equal to the control horizon of the model predictive control problem solved at each time step by the FORCESPRO controller. At each time step k, a predicted disturbance for the next N steps is considered by the FORCESPRO controller. For the cost function of the MPC problem, it is assumed that the relative importance of regulating the two states to zero is ten times as high as the penalty on applying an input. Further it is demanded, that the input magnitude of the input signal u lies in the range $[-1.8, 1.8]$. The initial state of the system is set to zero, i. e. $x_0 = [0; 0]$.



One can see that the disturbance drives the states far away from the desired value. In this example it is shown how FORCESPRO can significantly improve the dynamical behavior by using the concept of 'preview' when such future information is available.



Use preview information in the MATLAB® interface

The multistage problem is constructed as shown in the simple example *here* and is then extended as shown below.

The parametric additive terms **g** have to be defined. At each stage of the multistage problem, the equality constraint change, therefore we have to define a parameter for each stage. In the definition of the parameters, **distx** represents the name of the predicted disturbance at stage x of the multistage problem.

During runtime, the preview information is mapped to these parameters.

```
% RHS of first eq. constr. is a parameter: z1=-A*x0 -Bw*Road
parameter(1) = newParam('minusA_times_x0_BwDist',1,'eq.c');
% Parameter of Preview
parameter(2) = newParam('dist1',2,'eq.c');
parameter(3) = newParam('dist2',3,'eq.c');
parameter(4) = newParam('dist3',4,'eq.c');
parameter(5) = newParam('dist4',5,'eq.c');
parameter(6) = newParam('dist5',6,'eq.c');
parameter(7) = newParam('dist6',7,'eq.c');
parameter(8) = newParam('dist7',8,'eq.c');
parameter(9) = newParam('dist8',9,'eq.c');
parameter(10) = newParam('dist9',10,'eq.c');
```

After setting up the multistage problem with the parametric equality constraints, configure the solver settings (i. e. define solver output and solver options), the solver can be generated by using the command **generateCode(...)**. With the function provided by FORCESPRO, the system is now ready for simulation.

Comparison of MPC with Preview and Standard MPC

Figure 11.2 shows the dynamics of the system using a non-preview controller and a preview controller designed using FORCES Pro. One can see that the maximum deviation of the two states from their desired value is reduced by a factor 18, and 11, respectively. Compared to the open loop case, the magnitude of the deviation is reduced by a factor of 47, and 34, respectively.

Figure 11.1 shows the control action of both controllers. As expected, the input signal remains in the allowed range. One can see how the preview controller makes use of future information to provide a more aggressive control action that results in improved system performance.

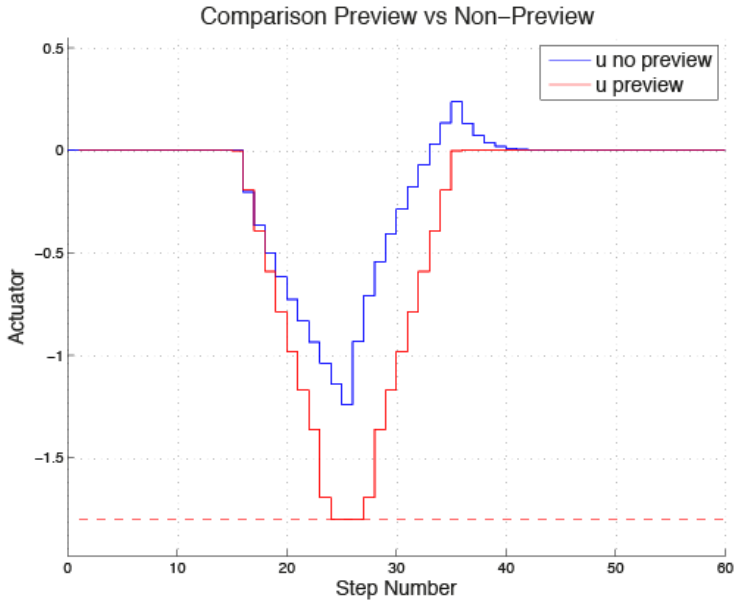


Figure 11.1: Comparison preview vs. non-preview

11.1.3 HOW TO: Implement an MPC Controller with a Time-Varying Dynamics

- *Introduction*
- *Implementation*
- *Comparison of the two approaches*

Introduction

This 'HOW TO' explains how FORCESPRO can be used to handle time-varying models to achieve better control performance than a standard MPC controller. For this example it is assumed that the time-varying model consists of four different systems. This could be four models derived from a nonlinear system at four operating points or from a periodic system. The systems are listed below. The first system is a damped harmonic oscillator, while the second system has eigenvalues on the right plane and is therefore unstable. System three is also a damped oscillator, but differs from system one. System four is an undamped harmonic

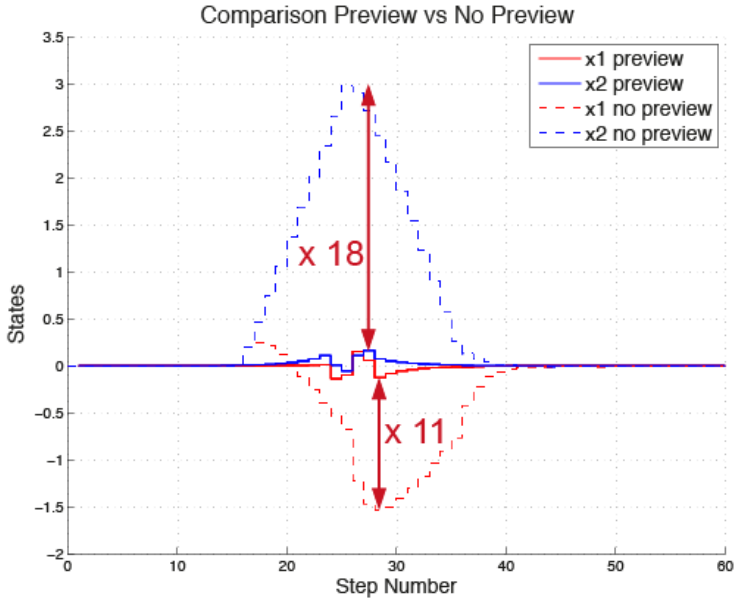


Figure 11.2: Comparison preview vs. no preview

oscillator.

$$\text{System 1:}\quad x_{k+1} = \begin{pmatrix} 0.7115 & -0.6 \\ 0.6 & 0.8853 \end{pmatrix} x_k + \begin{pmatrix} 0.2173 \\ 0.0573 \end{pmatrix} u_k$$

$$\text{System 2:}\quad x_{k+1} = \begin{pmatrix} 0.9 & 0.5 \\ 0.5 & 1 \end{pmatrix} x_k + \begin{pmatrix} 0 \\ 0.0666 \end{pmatrix} u_k$$

$$\text{System 3:}\quad x_{k+1} = \begin{pmatrix} 0.7115 & -0.5 \\ 0.5 & 1 \end{pmatrix} x_k + \begin{pmatrix} 0.5 \\ 0.01 \end{pmatrix} u_k$$

$$\text{System 4:}\quad x_{k+1} = \begin{pmatrix} 0 & 0.9 \\ -1 & 0 \end{pmatrix} x_k + \begin{pmatrix} 0 \\ 0.2 \end{pmatrix} u_k$$

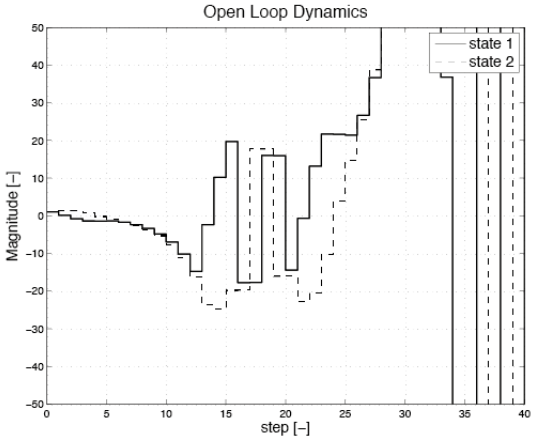
In this example we assume that system 1 is active for the first 4 steps. Then at step 5 the model changes to system 2, which stays active for 8 steps. Then we switch to system 3 for the following 3 steps and finally system 4 is active for the next 5 steps. This pattern is periodic, i. e. every 20 steps the cycle starts again. Also we have an initial condition of $x_0 = [1; 1]$, a prediction horizon $N = 15$ and the simulation runs for 40 steps.

The open loop dynamics of this time-varying model are shown on the right. One can see that the system becomes unstable. The goal is to regulate both states to zero while satisfying the different input constraints on each system. The constraints on the model are $u \in [-3, 5]$, $u \in [-5.5, 5.5]$, $u \in [-3, 5]$ and $u \in [-0.45, 4.5]$ for systems 1, 2, 3 and 4, respectively.

At each step k FORCESPRO takes the changing state space matrices and the corresponding input constraints into account, in order to regulate both states to zero as fast as possible. The following section shows how a controller for this problem can be implemented using the FORCESPRO MATLAB® Interface.

Implementation

The FORCESPRO MATLAB® Interface is used to pose a multistage problem problem as described *here*. When taking the changing dynamics over the prediction horizon into account, the matrices C_{i-1} and D_i of the inter-stage equality have to be defined as parameters for each prediction step i. Additionally the lower bounds $\underline{z}_i$ and the upper bounds $\overline{z}_i$ on the optimization variable have to be defined as parameters as they also change over the prediction



horizon. Also, the initial condition has to be set as a parameter. The code below shows the multistage problem and the commands to design the controller using FORCESPRO.

```
%% Multistage Problem: Varying Model in Prediction Horizon
stages = MultistageProblem(N); % get stages struct of length N

% Initial Equality
% c_1 = -A*x0
parameter(1) = newParam('minusA_times_x0',1,'eq.c');

for i = 1:N
        % dimension
        stages(i).dims.n = nx+nu; % number of stage variables
        stages(i).dims.r = nx; % number of equality constraints
        stages(i).dims.l = nu; % number of lower bounds
        stages(i).dims.u = nu; % number of upper bounds

        % lower bounds
        stages(i).ineq.b.lbidx = 1; % lower bound acts on these indices
        parameter(1+i) = newParam(['u',num2str(i),'min'],i,'ineq.b.lb');

        % upper bounds
        stages(i).ineq.b.ubidx = 1; % upper bound acts on these indices
        parameter(1+N+i) = newParam(['u',num2str(i),'max'],i,'ineq.b.ub');

        % cost
        stages(i).cost.H = blkdiag(R,Q);
        stages(i).cost.f = zeros(nx+nu,1);

        % Equality constraints
        if( i>1 )
                stages(i).eq.c = zeros(nx,1);
        end
        % Inter-Stage Equality
        % D_i*z_i = [B_i -I]*z_i
        parameter(1+2*N+i) = newParam(['D_',num2str(i)],i,'eq.D');
        if( i < n)
                % C_{i-1}*z_{i-1} = [0 A_i]*z_{i-1}
                parameter(1+3*N+i) = newParam(['C_',num2str(i)],i,'eq.C');
        end
end
```

(continues on next page)

(continued from previous page)

```
% define outputs of the solver
outputs(1) = newOutput('u0',1,1);
% solver settings
codeoptions = getOptions('Time_Varying_Model_wP');
% generate code
generateCode(stages,parameter,codeoptions,outputs);
```

You can find the MATLAB code of this example to try it out for yourself in the `examples` folder that comes with your client.

Comparison of the two approaches

The two plots in Figure 11.3 and Figure 11.4 respectively, show the difference between the response of a controller that assumes constant matrices A and B over the whole prediction horizon, and a controller that considers the changing dynamics, e. g. at time step 0 the second controller knows that system 1 will only be active for the first 4 steps. The left plot shows the system response and the right plot shows the actuator signals and the varying system constraints.

Both controllers can satisfy the constraints. To quantify the improvement in control performance, the cost function $\sum_{k=1}^{N} x_k^T Q x_k + u_k^T R u_k$ can be evaluated for the whole simulation length of $n = 40$. For the controller that uses a fixed model for the prediction horizon, the closed loop cost for regulating the states to zero is 2163.2. With the FORCESPRO time-varying controller the costs is reduced to 457.5. This is a cost reduction of almost 80%.

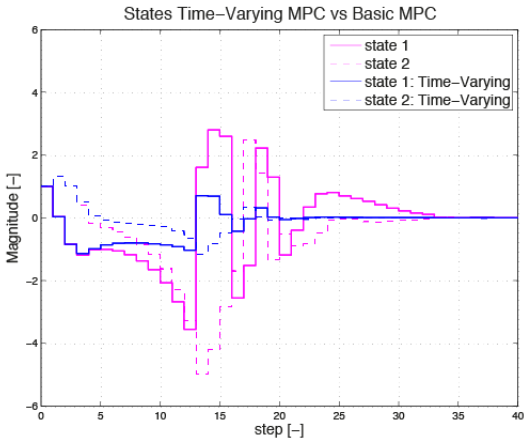


Figure 11.3: States Time-varying MPC vs. basic MPC

11.1.4 How to Implement 1-Norm and Infinity-Norm Cost Functions

- *Introduction*
- *1-norm reformulation*
- *∞-norm formulation*

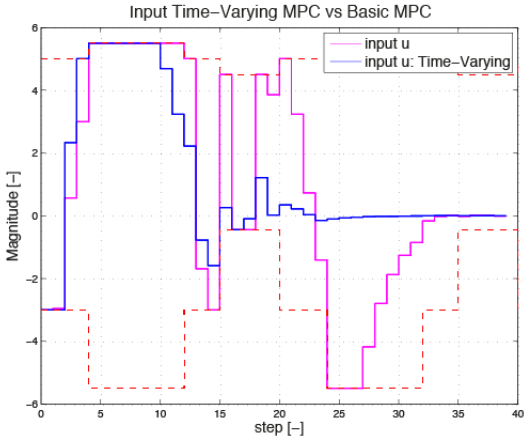


Figure 11.4: Input Time-varying MPC vs. basic MPC

Introduction

In this example we use the system described in the *Basic MPC Example*, but we will implement non-quadratic costs of the type

$$\|Ru_i\|_1$$

or

$$\|Qx_i\|_\infty$$

which are sometimes more meaningful for certain applications.

In both cases we will have to introduce slack variables and additional constraints, hence the optimization problem will become more challenging to solve, even if the cost function becomes linear instead of quadratic.

1-norm reformulation

The 1-norm is the absolute sum of a vector, hence a 1-norm penalty on the actuators can be a more meaningful objective when, for instance, the fuel consumption is directly proportional to actuation. The 1-norm also induces sparsity in the solution vector, i.e. a 1-norm cost leads to solutions where actuators are not used at all if possible, which can more accurately represent the objective of minimizing wear in certain applications.

To formulate a 1-norm cost as an optimization problem we introduce one slack variable ϵ_j per vector element of Ru_i (i.e. such that the vector ϵ has the same length as the vector Ru_i) and add it to the polytopic constraints. As a result, the problem

$$\begin{aligned} \text{minimize} \quad & \|Ru_i\|_1 \\ \text{subject to} \quad & \text{constraints} \end{aligned}$$

is transformed into the problem

$$\begin{aligned} \text{minimize} \quad & \sum_j \epsilon_j \\ \text{subject to} \quad & \pm Ru_i \leq \epsilon \\ & \text{constraints} \end{aligned}$$

The following MATLAB code shows how to model a problem with 1-norm penalties on the actuators and quadratic penalties on the states with FORCESPRO. In particular, note the changes to the cost function and the introduction of polytopic constraints.

```
%% FORCES multistage form
% assume variable ordering zi = [u{i-1}, x{i}, e{i-1}] for i=1...N

stages = MultistageProblem(N); % get stages struct of length N

for i = 1:N

        % dimension
        stages(i).dims.n = nx+2*nu; % number of stage variables
        stages(i).dims.r = nx;          % number of equality constraints
        stages(i).dims.l = nx+nu;       % number of lower bounds
        stages(i).dims.u = nx+nu;       % number of upper bounds
        stages(i).dims.p = 2*nu;        % number of polytopic constraints

        % cost
        if( i == N )
                stages(i).cost.H = blkdiag(zeros(nu),P,zeros(nu)); % terminal cost␣
↪(Hessian)
        else
                stages(i).cost.H = blkdiag(zeros(nu),Q,zeros(nu));
        end
        stages(i).cost.f = [zeros(nx+nu,1); ones(nu,1)]; % linear cost terms

        % lower bounds
        stages(i).ineq.b.lbidx = 1:(nu+nx); % lower bound acts on these indices
        stages(i).ineq.b.lb = [umin; xmin]; % lower bound for this stage variable

        % upper bounds
        stages(i).ineq.b.ubidx = 1:(nu+nx); % upper bound acts on these indices
        stages(i).ineq.b.ub = [umax; xmax]; % upper bound for this stage variable

        % polytopic bounds
   stages(i).ineq.p.A = [ R, zeros(nu,nx), -eye(nu); ...
                                                -R, zeros(nu,nx), -eye(nu)];
        stages(i).ineq.p.b = zeros(2*nu,1);

        % equality constraints
        if( i < N )
                 stages(i).eq.C = [zeros(nx,nu), A, zeros(nx,nu) ];
        end
        if( i>1 )
                 stages(i).eq.c = zeros(nx,1);
        end
        stages(i).eq.D = [B, -eye(nx), zeros(nx,nu)];
end

% RHS of first eq. constr. is a parameter: stages(1).eq.c = -A*x0
params(1) = newParam('minusA_times_x0',1,'eq.c');
```

You can download the MATLAB code of this example using this link.

∞-norm formulation

The ∞-norm is the maximum absolute value in a vector, hence an ∞-norm penalty on the states tries to minimize the maximum deviation of any state from the setpoint rather than the combined deviation of all the states in the system.

To formulate an ∞-norm cost as an optimization problem we need to introduce a single slack variable $epsilon$ and add polytopic constraints. As a result, the problem

$$\begin{aligned} \text{minimize} \quad & \|Qx_i\|_\infty \\ \text{subject to} \quad & \text{constraints} \end{aligned}$$

is transformed into the problem

$$\begin{aligned} \text{minimize} \quad & \epsilon \\ \text{subject to} \quad & \pm Qx_i \leq \mathbf{1}^T \epsilon \\ & \text{constraints} \end{aligned}$$

where the vector $\mathbf{1} = [1 \ \dots \ 1]$ has the same length as the vector Qx_i.

The following MATLAB code shows how to model a problem with ∞-norm penalties on the states and quadratic penalties on the inputs with FORCESPRO. In particular, note the changes to the cost function and the introduction of polytopic constraints. Also note that we only need to add one more variable per stage.

```
%% FORCES multistage form
% assume variable ordering zi = [u{i-1}, x{i}, e{i-1}] for i=1...N

stages = MultistageProblem(N); % get stages struct of length N

for i = 1:N

        % dimension
        stages(i).dims.n = nx+nu+1; % number of stage variables
        stages(i).dims.r = nx;          % number of equality constraints
        stages(i).dims.l = nx+nu;       % number of lower bounds
        stages(i).dims.u = nx+nu;       % number of upper bounds
        stages(i).dims.p = 2*nx;        % number of polytopic constraints

        % cost
        if( i == N )
                stages(i).cost.H = blkdiag(R,zeros(nx),0); % terminal cost (Hessian)
        else
                stages(i).cost.H = blkdiag(Q,zeros(nx),0);
        end
        stages(i).cost.f = [zeros(nx+nu,1); 1]; % linear cost terms

        % lower bounds
        stages(i).ineq.b.lbidx = 1:(nu+nx); % lower bound acts on these indices
        stages(i).ineq.b.lb = [umin; xmin]; % lower bound for this stage variable

        % upper bounds
        stages(i).ineq.b.ubidx = 1:(nu+nx); % upper bound acts on these indices
        stages(i).ineq.b.ub = [umax; xmax]; % upper bound for this stage variable

        % polytopic bounds
        if( i == N )
           stages(i).ineq.p.A = [ zeros(nx,nu), P, -ones(nx,1); ...
                                                        zeros(nx,nu), -P, -ones(nx,
↪1)];
        else
           stages(i).ineq.p.A = [ zeros(nx,nu), Q, -ones(nx,1); ...
                                                        zeros(nx,nu), -Q, -ones(nx,
↪1)];
```

(continues on next page)

(continued from previous page)

```
        end
        stages(i).ineq.p.b = zeros(2*nx,1);

        % equality constraints
        if( i < N )
                stages(i).eq.C = [zeros(nx,nu), A, zeros(nx,1)];
        end
        if( i>1 )
                stages(i).eq.c = zeros(nx,1);
        end
        stages(i).eq.D = [B, -eye(nx), zeros(nx,1)];
end

% RHS of first eq. constr. is a parameter: stages(1).eq.c = -A*x0
params(1) = newParam('minusA_times_x0',1,'eq.c');
```

Here you can download the MATLAB code of this example.

11.1.5 HOW TO: Implement Rate Constraints

- *Problem formulation*
- *Implementation*
- *Simulation Results*

Problem formulation

In this example it is illustrated how slew rate constraints on a system's actuators can be incorporated in the controller design. As a real world example one could think of an airplane, where the elevator cannot be switched instantaneously from one position to another, i. e. has a limited slew rate. Here the concept of constraints on the slew rate is shown on the following system:

$$x_{k+1} = \begin{pmatrix} 0.7115 & -0.4345 \\ 0.4345 & 0.8853 \end{pmatrix} x_k + \begin{pmatrix} 0.2173 \\ 0.0573 \end{pmatrix} u_k \Leftrightarrow x_{k+1} = Ax_k + Bu_k$$

To have a bound on the slew rate, $u_k - u_{k-1}$ has to lie in some range, i. e.

$$\Delta u_{min} \leq u_k - u_{k-1} \leq \Delta u_{max}.$$

One option to set the constraints on the slew rate is to augment the state as follows:

$$\hat{x}_k = \begin{pmatrix} x_k \\ u_{k-1} \end{pmatrix} \quad \Leftrightarrow \quad \hat{x}_{k+1} = \begin{pmatrix} A & B \\ 0 & I \end{pmatrix} \hat{x}_k + \begin{pmatrix} B \\ I \end{pmatrix} \hat{u}_k \quad \Leftrightarrow \quad \hat{x}_{k+1} = \hat{A}\hat{x}_k + \hat{B}\hat{u}_k$$

where $\hat{u}$ is defined as $u_k - u_{k-1}$. To implement the problem using FORCESPRO, the multistage problem has to be defined as stated *here*. The optimization variable is $z_i = [\hat{u}_i \quad \hat{x}_{i+1}]^T$.

$$\begin{aligned}
\hat{x}_{k+1} =& \hat{A}\hat{x}_k + \hat{B}\hat{u}_k \\
\Delta u_{min} \leq& \hat{u} \leq \Delta u_{max} \\
u_{min} \leq& u \leq \Delta u_{max} \\
& \Downarrow \\
\text{minimize} \quad & \frac{1}{2}\sum_{i=1}^{N} z_i^T H_i z_i \\
\text{subject to} \quad & D_1 z_1 = c_1 \\
& C_{i-1} z_{i-1} + D_i z_i = c_i \\
& z_{min} \leq z_i \leq z_{max}
\end{aligned}$$

The details on how the first equality and the interstage equality look like and how the constraints are implemented can be seen in the MATLAB® code below.

Implementation

```
%% FORCES multistage form
% assume variable ordering zi = [uhat_{i-1}, xhat_{i}] for i=1...N

stages = MultistageProblem(N); % get stages struct of length N

for i = 1:N

        % dimension
        stages(i).dims.n = 4; % number of stage variables
        stages(i).dims.r = 3; % number of equality constraints
        stages(i).dims.l = 2; % number of lower bounds: minimal slew rate and minimal␣
↪input
        stages(i).dims.u = 2; % number of upper bounds: maximal slew rate and maximal␣
↪input

        % cost
        if( i == N )
                stages(i).cost.H = blkdiag(R_sr, [P, zeros(2,1); zeros(1,2), 0]); %␣
↪terminal cost (Hessian)
        else
                stages(i).cost.H = blkdiag(R_sr, [Q, zeros(2,1); zeros(1,2), R]);
        end
        stages(i).cost.f = zeros(3,1); % linear cost terms

        % lower bounds
        stages(i).ineq.b.lbidx = [1,4]; % indices of lower bounds
        stages(i).ineq.b.lb = [dumin; umin]; % lower bounds

        % upper bounds
        stages(i).ineq.b.ubidx = [1,4]; % indices of upper bounds
        stages(i).ineq.b.ub = [dumax; umax]; % upper bounds

        % equality constraints
        if( i < N )
                 stages(i).eq.C = [zeros(3,1), [ A, B; zeros(1, 2), 1]];
        end
```

(continues on next page)

(continued from previous page)

```
        if( i>1 )
                stages(i).eq.c = zeros(3,1);
        end
        stages(i).eq.D = [[B; 1], -eye(3)];

end

% RHS of initial equality constraint is a parameter
parameter(1) = newParam('minusAhat_times_xhat0',1,'eq.c');

% Define outputs of the solver
output(1) = newOutput('uhat',1,1);

% Solver settings
codeoptions = getOptions('RateConstraints_Controller');

% Generate code
generateCode(stages,parameter,codeoptions,output);
```

You can download the MATLAB code of this example to try it out for yourself here

Simulation Results

For simulation the following specifications are assumed: the initial condition $x_0 \in [-2; 6]$, the input signal u is in the range $[-0.5, 2]$ and the constraints on the slew rate is $\hat{u} \in [-1, 0.5]$. Figure 11.5, Figure 11.6 and Figure 11.7 show how the controller regulates both states to zero while $\hat{u}$ and u remain in the required range.

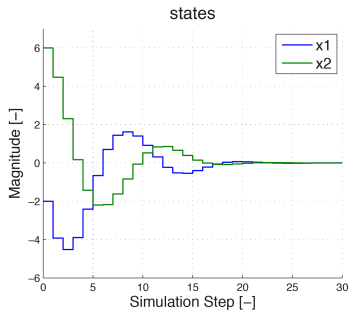


Figure 11.5: The states are both regulated to zero. No constraints are imposed on the states.

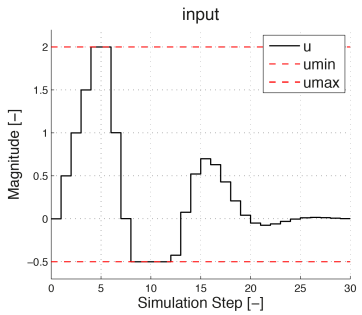


Figure 11.6: Plot of u

In Figure 11.6 and Figure 11.7 one sees how the input signal is maximally increased in the beginning with a slew rate of 0.5, until it reaches its upper bound of 2. In the figure on the right

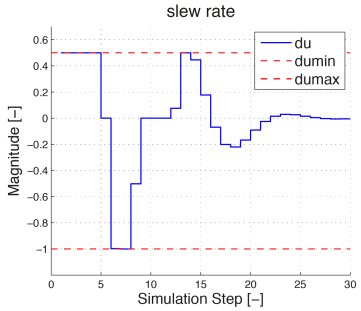


Figure 11.7: Plot of du

the slew rate is depicted. One can see that in the beginning, the slew rate stays at its upper bound 0.5. At simulation step 6 the input signal is maximally reduced. Again this is visible from the slew rate being at its lower bound -1.

11.1.6 Binary MPC Example

- *Simulation result*
- *Details on problem reformulation*

Let us consider a simple MPC example where the system has inputs that can take only two values, u_{min} or u_{max}. The original problem (shown on the left) can be reformulated into the problem on the right, which corresponds to a standard form for which FORCESPRO can generate a solver. The details of the reformulation are given at the end of this example.

Simple MPC problem with discrete inputs:

$$\begin{aligned} \text{minimize} \quad & x_N^T P x_N + \sum_{i=0}^{N-1} x_i^T Q x_i + u_i^T R u_i \\ \text{subject to} \quad & x_0 = x \\ & x_{i+1} = A x_i + B u_i \\ & x_{min} \leq x_i \leq x_{max} \\ & u_i \in \{u_{min}, u_{max}\} \end{aligned}$$

Equivalent problem with binary inputs

$$\begin{aligned} \text{minimize} \quad & x_N^T P x_N + \sum_{i=0}^{N-1} x_i^T Q x_i + \delta_i^T \tilde{R} \delta_i + \tilde{f}^T \delta_i \\ \text{subject to} \quad & x_0 = x \\ & x_{i+1} = A x_i + \tilde{B} \delta_i + b \\ & x_{min} \leq x_i \leq x_{max} \\ & \delta_i \in \{0,1\}^{n_u} \end{aligned}$$

The problem on the right can now be easily formulated in FORCESPRO. Note that the problem description is very similar to that of the simple MPC example, with the only modification that certain variables are marked to be binary. Download and run a complete simulation script to see the output.

```
nx = 2; nu = 2;
```

(continues on next page)

(continued from previous page)

```
% assume variable ordering zi = [delta_{i-1}; x{i}] for i=1...N
stages = MultistageProblem(N);
for i = 1:N
  % dimension
  stages(i).dims.n = nx+nu; % number of stage variables
  stages(i).dims.r = nx; % number of equality constraints
  stages(i).dims.l = nx; % number of lower bounds
  stages(i).dims.u = nx; % number of upper bounds
  stages(i).bidx = 1:nu; % index of binary variables

  % cost
  if( i == N )
        stages(i).cost.H = blkdiag(Rtilde,P);
  else
        stages(i).cost.H = blkdiag(Rtilde,Q);
  end
  stages(i).cost.f = [ftilde; zeros(nx,1)];

  % lower bounds
  stages(i).ineq.b.lbidx = (nu+1):(nu+nx); % lower bound on states
  stages(i).ineq.b.lb = xmin; % upper bound values

  % upper bounds
  stages(i).ineq.b.ubidx = (nu+1):(nu+nx); % upper bound for this stage variable
  stages(i).ineq.b.ub = umax; % upper bound for this stage variable

  % equality constraints
  if( i < N )
        stages(i).eq.C = [zeros(nx,nu), A];
  end
  if( i>1 )
        stages(i).eq.c = -Bconst;
  end
  stages(i).eq.D = [Btilde, -eye(nx)];
end

% RHS of first eq. constr. is a parameter: z1=-A*x0
params(1) = newParam('minusA_times_x0',1,'eq.c');
```

You can find the MATLAB code of this example to try it out for yourself in the `examples` folder that comes with your client.

You can download the Python code of this example here.

Simulation result

When running the example, you should see the following closed-loop behavior:

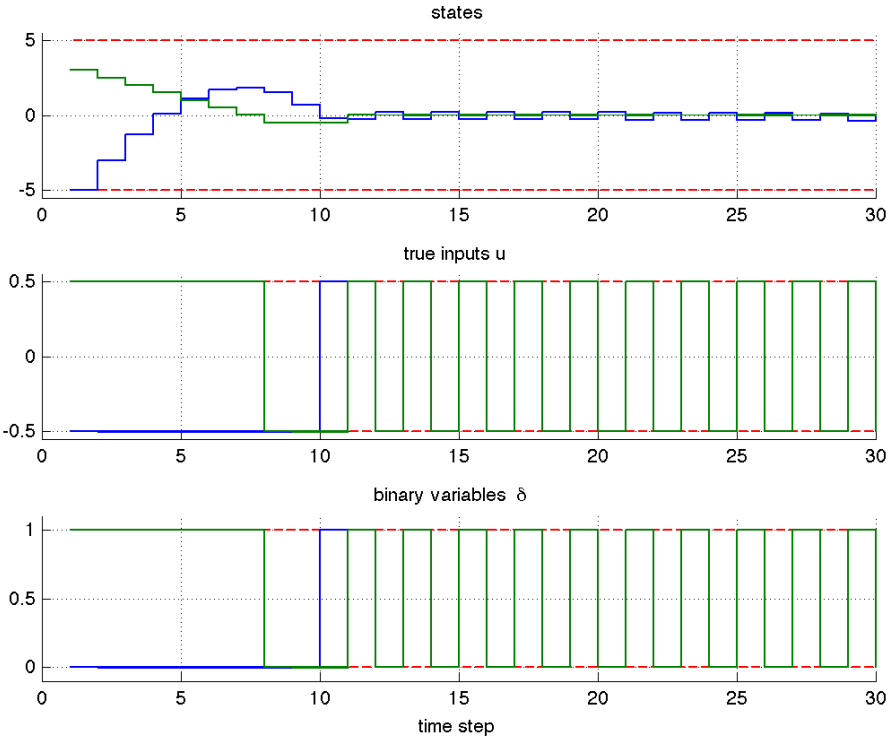
Details on problem reformulation

The reformulation is done as follows: we introduce a variable $delta$ such that

$$\delta = 0 \Leftrightarrow u = u_{min} \quad \text{and} \quad \delta = 0 \Leftrightarrow u = u_{max}$$

This can be formulated by the equality constraint

$$u = u_{min} + \text{diag}(u_{max} - u_{min})\delta$$



where diag denotes a diagonal matrix. To keep the number of variables at a minimum, we will directly insert this equation into the dynamics:

$$\begin{aligned} x^+ &= Ax + Bu \\ &= Ax + Bu_{min} + B\text{diag}(u_{max} - u_{min})\delta \\ &= Ax + \tilde{B}\delta + b \end{aligned}$$

where $\tilde{B} := B\text{diag}(u_{max} - u_{min})$ and $b := Bu_{min}$.

Similarly for the cost function,

$$\begin{aligned} u^T R u &= (u_{min} + \text{diag}(u_{max} - u_{min})\delta)^T R(u_{min} + \text{diag}(u_{max} - u_{min})\delta) \\ &= \delta^T \text{diag}(u_{max} - u_{min})R\text{diag}(u_{max} - u_{min})\delta + 2u_{min}\text{diag}(u_{max} - u_{min})R\delta + \text{const} \\ &= \delta^T \tilde{R}\delta + \tilde{f}^T\delta + \text{const} \end{aligned}$$

where

$$\begin{aligned} \tilde{R} &= \text{diag}(u_{max} - u_{min})R\text{diag}(u_{max} - u_{min}) \\ \tilde{f} &= 2R\text{diag}(u_{max} - u_{min})u_{min} \end{aligned}$$

11.2 Y2F interface: Basic example

- *Defining the problem data*
- *Defining the MPC problem*
- *Generating a solver*
- *Calling the generated solver*

- *Simulation*
- *Results*
- *Variation 1: Parametric cost*
- *Variation 2: Time-varying dynamics*
- *Variation 3: Time-varying constraints*

Consider the following linear MPC problem with lower and upper bounds on state and inputs, and a terminal cost term:

$$\begin{aligned} \text{minimize} \quad & x_N^\top P x_N + \sum_{i=0}^{N-1} x_i^\top Q x_i + u_i^\top R u_i \\ \text{subject to} \quad & x_0 = x \\ & x_{i+1} = A x_i + B u_i \\ & \underline{x} \leq x_i \leq \bar{x} \\ & \underline{u} \leq u_i \leq \bar{u} \end{aligned}$$

This problem is parametric in the initial state x and the first input u_0 is typically applied to the system after a solution has been obtained. Here we present the problem formulation with YALMIP, how you can use Y2F to easily generate a solver with FORCESPRO, and how you can use the resulting controller for simulation.

You can download the MATLAB code of this example to try it out for yourself from https://raw.githubusercontent.com/embotech/Y2F/master/examples/mpc_basic_example.m.

Important: Make sure to have YALMIP installed correctly (run `yalmiptest` to verify this).

11.2.1 Defining the problem data

Let's define the known data of the MPC problem, i.e. the system matrices A and B, the prediction horizon N, the stage cost matrices Q and R, the terminal cost matrix P, and the state and input bounds:

```
%% MPC problem data

% system matrices
A = [1.1 1; 0 1];
B = [1; 0.5];
[nx,nu] = size(B);

% horizon
N = 10;

% cost matrices
Q = eye(2);
R = eye(1);
if exist('dlqr', 'file')
    [~,P] = dlqr(A,B,Q,R);
else
    fprintf('Did not find dlqr (part of the Control Systems Toolbox). Will use 10*Q␣
↪for the terminal cost matrix.\n');
    P = 10*Q;
end
```

(continues on next page)

(continued from previous page)

```
% constraints
umin = -0.5;      umax = 0.5;
xmin = [-5; -5]; xmax = [5; 5];
```

11.2.2 Defining the MPC problem

Let's now dive in right into the problem formulation:

```
%% Build MPC problem in Yalmip

% Define variables
X = sdpvar(nx,N+1,'full'); % state trajectory: x0,x1,...,xN (columns of X)
U = sdpvar(nu,N,'full'); % input trajectory: u0,...,u_{N-1} (columns of U)

% Initialize objective and constraints of the problem
cost = 0; const = [];

% Assemble MPC formulation
for i = 1:N
    % cost
    if( i < N )
        cost = cost + 0.5*X(:,i+1)'*Q*X(:,i+1) + 0.5*U(:,i)'*R*U(:,i);
    else
        cost = cost + 0.5*X(:,N+1)'*P*X(:,N+1) + 0.5*U(:,N)'*R*U(:,N);
    end

    % model
    const = [const, X(:,i+1) == A*X(:,i) + B*U(:,i)];

    % bounds
    const = [const, umin <= U(:,i) <= umax];
    const = [const, xmin <= X(:,i+1) <= xmax];
end
```

Thanks to YALMIP, defining the mathematical problem is very much like writing down the mathematical equations in code.

11.2.3 Generating a solver

We have now incrementally built up the `cost` and `const` objects, which are both YALMIP objects. Now comes the magic: use the function `optimizerFORCES` to generate a solver for the problem defined by `const` and `cost` with the initial state as a parameter, and the first input move u_0 as an output:

```
%% Create controller object (generates code)
% for a complete list of codeoptions, see
% https://www.embotech.com/FORCES-Pro/User-Manual/Low-level-Interface/Solver-Options
codeoptions = getOptions('simpleMPC_solver'); % give solver a name
controller = optimizerFORCES(const, cost, codeoptions, X(:,1), U(:,1), {'xinit'}, {'u0
↪'});
```

That's it! Y2F automatically figures out the structure of the problem and generates a solver.

11.2.4 Calling the generated solver

We can now use the `controller` object to call the solver:

```
% Evaluate controller function for parameters
[output,exitflag,info] = controller{ xinit };
```

or call the generated MEX code directly:

```
% This is an equivalent call, if the controller object is deleted from the workspace
[output,exitflag,info] = simpleMPC_solver({ xinit });
```

Tip: Type `help solvername` to get more information about how to call the solver.

11.2.5 Simulation

Let's now simulate the closed loop over the prediction horizon N:

```
%% Simulate
x1 = [-4; 2];
kmax = 30;
X = zeros(nx,kmax+1); X(:,1) = x1;
U = zeros(nu,kmax);
problem.z1 = zeros(2*nx,1);
for k = 1:kmax

  % Evaluate controller function for parameters
  [U(:,k),exitflag,info] = controller{ X(:,k) };

  % Always check the exitflag in case something went wrong in the solver
  if( exitflag == 1 )
      fprintf('Time step %2d: FORCES took %2d iterations and %5.3f ', k,  info.it,␣
↪info.solvetime*1000);
      fprintf('milliseconds to solve the problem.\n');
  else
      info
      error('Some problem in solver');
  end

  % State update
  X(:,k+1) = A*X(:,k) + B*U(:,k);
end
```

11.2.6 Results

The results of the simulation are presented in Figure 11.8. The plot on the top shows the system's states over time, while the plot on the bottom shows the input commands. We can see that all constraints are respected.

11.2.7 Variation 1: Parametric cost

One possible variation is if we consider the weighting matrices Q, R and P as parameters, so that we can tune them after the code generation. The following problem is solved at each

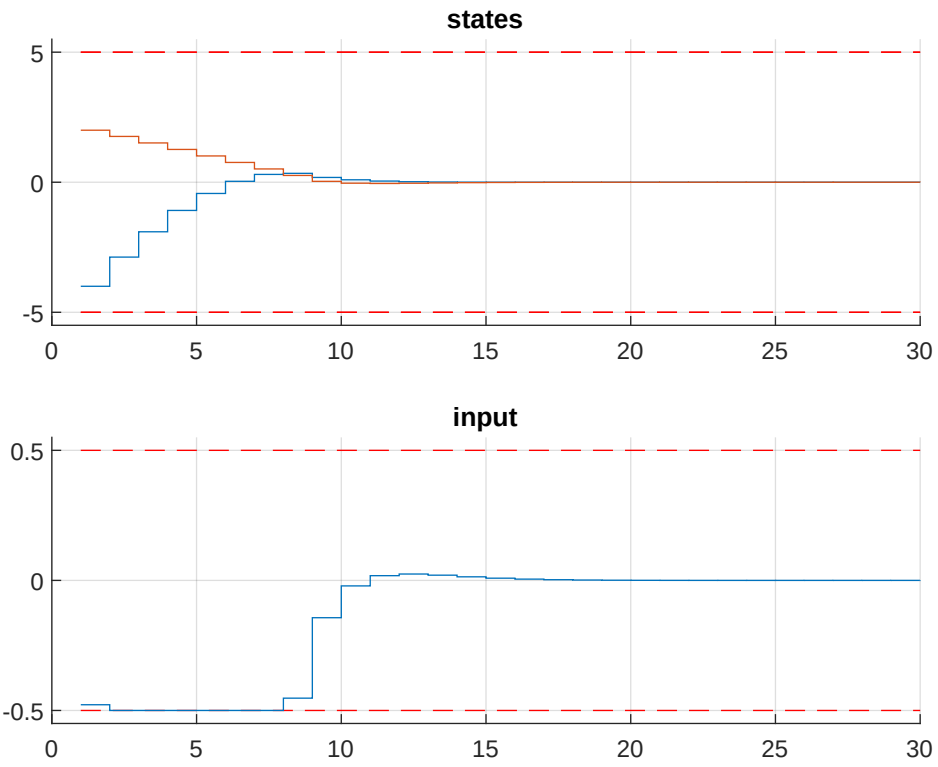


Figure 11.8: Simulation results of the states (top, in blue and red) and input (bottom, in blue) over time. The state and input constraints are plotted in red dashed lines.

time step:

$$\begin{aligned} \text{minimize} \quad & x_N^\top P x_N + \sum_{i=0}^{N-1} x_i^\top Q x_i + u_i^\top R u_i \\ \text{subject to} \quad & x_0 = x \\ & x_{i+1} = A x_i + B u_i \\ & \underline{x} \leq x_i \leq \bar{x} \\ & \underline{u} \leq u_i \leq \bar{u} \end{aligned}$$

As usual, this problem is also parametric in the initial state x and the first input u_0 is applied to the system after a solution has been obtained. To be able to define the weighting matrices Q, R and P as parameters, first we define them as `sdpvars` and then tell `optimizerFORCES` that they are parameters:

```
% Cost matrices - these will be parameters later
Q = sdpvar(nx);
R = sdpvar(nu);
P = sdpvar(nx);

% [... formulate MPC problem in YALMIP ...]

% Define parameters and outputs
codeoptions = getOptions('parametricCost_solver'); % give solver a name
parameters     = { X(:,1),   Q,   R,   P  };
parameterNames = { 'xinit', 'Q', 'R', 'P' };
outputs     =      U(:,1)   ;
outputNames = {'controlInput'};
controller = optimizerFORCES(const, cost, codeoptions, parameters, outputs,␣
↪parameterNames, outputNames);
```

You can download the MATLAB code of this variation to try it out for yourself from https://raw.githubusercontent.com/embotech/Y2F/master/examples/mpc_parametric_cost.m.

11.2.8 Variation 2: Time-varying dynamics

Another possible variation is if we consider the state-space dynamics matrices A and B as parameters, so that we can change them after the code generation. The following problem is solved at each time step:

$$\begin{aligned} \text{minimize} \quad & x_N^\top P x_N + \sum_{i=0}^{N-1} x_i^\top Q x_i + u_i^\top R u_i \\ \text{subject to} \quad & x_0 = x \\ & x_{i+1} = A x_i + B u_i \\ & \underline{x} \leq x_i \leq \bar{x} \\ & \underline{u} \leq u_i \leq \bar{u} \end{aligned}$$

As usual, this problem is also parametric in the initial state x and the first input u_0 is applied to the system after a solution has been obtained. To be able to define the state-space dynamics matrices A and B as parameters, first we define them as `sdpvars` and then tell `optimizerFORCES` that they are parameters:

```
A = sdpvar(nx,nx,'full'); % system matrix - parameter
B = sdpvar(nx,nu,'full'); % input matrix - parameter
```

(continues on next page)

(continued from previous page)

```
% [... formulate MPC problem in YALMIP ...]

% Define parameters and outputs
codeoptions = getOptions('parametricDynamics_solver'); % give solver a name
parameters = { x0, A, B };
parameterNames = { 'xinit', 'Amatrix', 'Bmatrix' };
controller = optimizerFORCES(const, cost, codeoptions, parameters, U(:,1),␣
↪parameterNames, {'u0'} );
```

You can download the MATLAB code of this variation to try it out for yourself from https://raw.githubusercontent.com/embotech/Y2F/master/examples/mpc_parametric_dynamics.m.

11.2.9 Variation 3: Time-varying constraints

One final variation is if we consider the constraint inequalities as parameters, so that we can change them after the code generation. The inequalities are defined by a time-varying 2×2 matrix that is defined by 2 parameters:

$$R_k x \leq R_k \bar{x}$$

where k is the simulation step and the rotation matrix is defined by:

$$R_k = \begin{bmatrix} \cos(kw) & -\sin(kw) \\ \sin(kw) & \cos(kw) \end{bmatrix} = \begin{bmatrix} r_1 & -r_2 \\ r_2 & r_1 \end{bmatrix}$$

where k is the simulation step and w a fixed number. Overall, the following problem is solved at each time step:

$$\begin{aligned} \text{minimize} \quad & x_N^\top P x_N + \sum_{i=0}^{N-1} x_i^\top Q x_i + u_i^\top R u_i \\ \text{subject to} \quad & x_0 = x \\ & x_{i+1} = A x_i + B u_i \\ & \underline{x} \leq x_i \leq \bar{x} \\ & \underline{u} \leq u_i \leq \bar{u} \\ & R_k x_i \leq R_k \bar{x} \end{aligned}$$

As usual, this problem is also parametric in the initial state x and the first input u_0 is applied to the system after a solution has been obtained. To be able to define the rotation matrix R_k as a parameter, first we define r_1 and r_2 as `sdpvars` and then tell `optimizerFORCES` that they are parameters:

```
sdpvar r1 r2 % parameters for rotation matrix
R = [r1, -r2; r2, r1];

% [... formulate MPC problem in YALMIP ...]

% Define parameters and outputs
parameters = { X(:,1), r1, r2 };
parameterNames = { 'xinit', sprintf('cos(k*%4.2f)',w), sprintf('sin(k*%4.2f)',w) };
outputs = U(:,1);
outputNames = {'u0'};
controller = optimizerFORCES(const, cost, codeoptions, parameters, outputs,␣
↪parameterNames, outputNames);
```

You can download the MATLAB code of this variation to try it out for yourself from https://raw.githubusercontent.com/embotech/Y2F/master/examples/mpc_parametric_inequalities.m.

11.3 Y2F interface: Trajectory Optimization for Quadrotor Flight

- *Defining the problem parameters*
- *Defining the MPC problem*
- *Generating a solver*
- *Calling the generated solver*
- *Results*

This is a more complex example optimizing the trajectory of a quadrotor within safe flight corridors. It follows the formulation give in *S. Liu et al., "Planning Dynamically Feasible Trajectories for Quadrotors Using Safe Flight Corridors in 3-D Complex Environments," IEEE Robotics and Automation Letters, vol. 2, no. 3, pp. 1688-1695, July 2017* and makes the following assumptions:

- The system is differentially flat, with flat outputs $[x, y, z, \psi]^T$
- Piece-wise trajectory constrained by polytopes for each piece
- Trajectory segment parametrized as n-th order polynomial in time, separable in states

Based on those assumptions, the following convex QP problem needs to be solved in real-time:

$$
\begin{aligned}
\underset{\Phi}{\text{argmin}} \quad & J = \sum_{i=0}^{N-1} \int_0^{\Delta t_i} \left\| \frac{\mathrm{d}^4}{\mathrm{d}t^4} \Phi_i(t) \right\|^2 \mathrm{d}t \\
\text{subject to} \quad & \frac{\mathrm{d}^k}{\mathrm{d}t^k} \Phi_i(\Delta t_i) = \frac{\mathrm{d}^k}{\mathrm{d}t^k} \Phi_{i+1}(0) \quad k = 0, \ldots, 4 \\
& A_i^T \Phi_i(t_s) < b_i \qquad\qquad t_s = 0, \Delta t_s, 2\Delta t_s, \ldots t_{i+1} - t_i \\
& \Phi(t) = \begin{cases} \Phi_0(t - t_0) & t_0 \le t < t_1 \\ \Phi_1(t - t_1) & t_1 \le t < t_2 \\ \vdots & \\ \Phi_{N-1}(t - t_{N-1}) & t_{N-1} \le t < t_N \end{cases}
\end{aligned}
$$

with

$$
\Phi_i(t) = \begin{bmatrix} {}_x\Phi_i(t) \\ {}_y\Phi_i(t) \\ {}_z\Phi_i(t) \\ {}_\psi\Phi_i(t) \end{bmatrix} = \begin{bmatrix} {}_xc_i^0 & {}_xc_i^1 & {}_xc_i^2 & \cdots & {}_xc_i^n \\ {}_yc_i^0 & {}_yc_i^1 & {}_yc_i^2 & \cdots & {}_yc_i^n \\ {}_zc_i^0 & {}_zc_i^1 & {}_zc_i^2 & \cdots & {}_zc_i^n \\ {}_\psi c_i^0 & {}_\psi c_i^1 & {}_\psi c_i^2 & \cdots & {}_\psi c_i^n \end{bmatrix} \begin{bmatrix} 1 \\ t \\ t^2 \\ \vdots \\ t^n \end{bmatrix}
$$

This problem has $4 * (n + 1)$ optimization variables. Here we present a problem formulation with FORCESPRO's Y2F interface for YALMIP and also show how you can use the resulting controller for simulation.

You can download the code of this example to try it out for yourself (in MATLAB) by clicking **here**.

Important: Make sure to have YALMIP installed correctly (run `yalmiptest` to verify this). Visualizations of this example additionally require the MPT Toolbox and MATLAB interface for the CDD solver to be installed.

11.3.1 Defining the problem parameters

At the top of the example file, basic parameters are defined such number of states, the order of the piece-wise polynomial basis functions and number of samples to check the constraints:

```
%% Parameters
nStates = 4;  % [-]   Number of states
              %       Flat outputs [x position; y position; z position; yaw angle]
n = 8;        % [-]   Order of piece-wise polynomial used as basis function
nSample = 5;  % [-]   Number of intermediate samples (where constraints are checked)

withVisualization = true;  % [-]   Bool if MPT Toolbox for visualization is installed
bbConstr = false;          % [-]   true:   bounding-box constraints (separable in␣
↪coordinates) (n=7,8,9)
                           %       false:  Polyhedron along path (non-separable␣
↪polytopic constraints) (n=8)
```

The quadrotor is supposed to fly along piece-wise segments in 3D space that are defined by a list of way points:

```
%% WayPoints and time needed for segment
% Simple case with 3 segments
p0 = [0; 0; 0; 0];
p1 = [1; 1; 1; 0];
p2 = [3; 1; 1; pi];
p3 = [4; 2; 2; pi];
```

These waypoints are then used to construct artificial polyhedrons around each path segment.

11.3.2 Defining the MPC problem

Afterwards, YALMIP variables `Z` and `T` are defined, gathering the trajectory parameters and the trajectory positions, respectively.

```
%% YALMIP Variables
Z = sdpvar((n+1)*nStates,N,'full');   % Trajectory parameters: z0,z1,...,z{N-1}␣
↪(columns of Z for N stages/segm.)
                                      % z_i = [c_0^x,c_1^x,...,c_n^x, ...       (n-th␣
↪order polynomials -> n+1 coeff.
                                      %         c_0^y,c_1^y,...,c_n^y, ...
                                      %         c_0^z,c_1^z,...,c_n^z, ...
                                      %         c_0^phi,c_1^phi,...,c_n^phi]
                                      % where [x,y,z,phi] are the flat outputs (# of␣
↪flat outputs == nStates)
T = sdpvar(nStates,N+1,'full');       % Trajectory positions used as parameters
```

Afterwards, the QP formulation is setup in YALMIP syntax, including the quadratic cost function as well as various constraints.

11.3.3 Generating a solver

We have now incrementally built up the **`cost`** and **`constr`** objects, which are both YALMIP objects. Using the function **`optimizerFORCES`** to generate a solver named **`TrajOptQuadrotor`** that will return the optimized coefficients z_opt as an output:

```
%% Generate Solver
codeoptions = getOptions('TrajOptQuadrotor'); % solverName
codeoptions.optlevel = 3;
codeoptions.timing = 1;
codeoptions.BuildSimulinkBlock = 0;

controller  = optimizerFORCES(constr, cost, codeoptions, T, Z, {'wayPoints'}, {'z_opt
↪'});
```

That's it! Y2F automatically figures out the structure of the problem and generates a solver.

11.3.4 Calling the generated solver

We can now use the `TrajOptQuadrotor` object to call the solver:

```
%% Solve
[out_opt, exitflags, info] = TrajOptQuadrotor({pathSegments});
```

Tip: Type `help TrajOptQuadrotor` to get more information about how to call the solver.

11.3.5 Results

The example also includes additional lines of code to illustrate the results.

Figure 11.9 illustrates the quadrotor flight in 3D, while Figure 11.10 shows the individual trajectories in time.

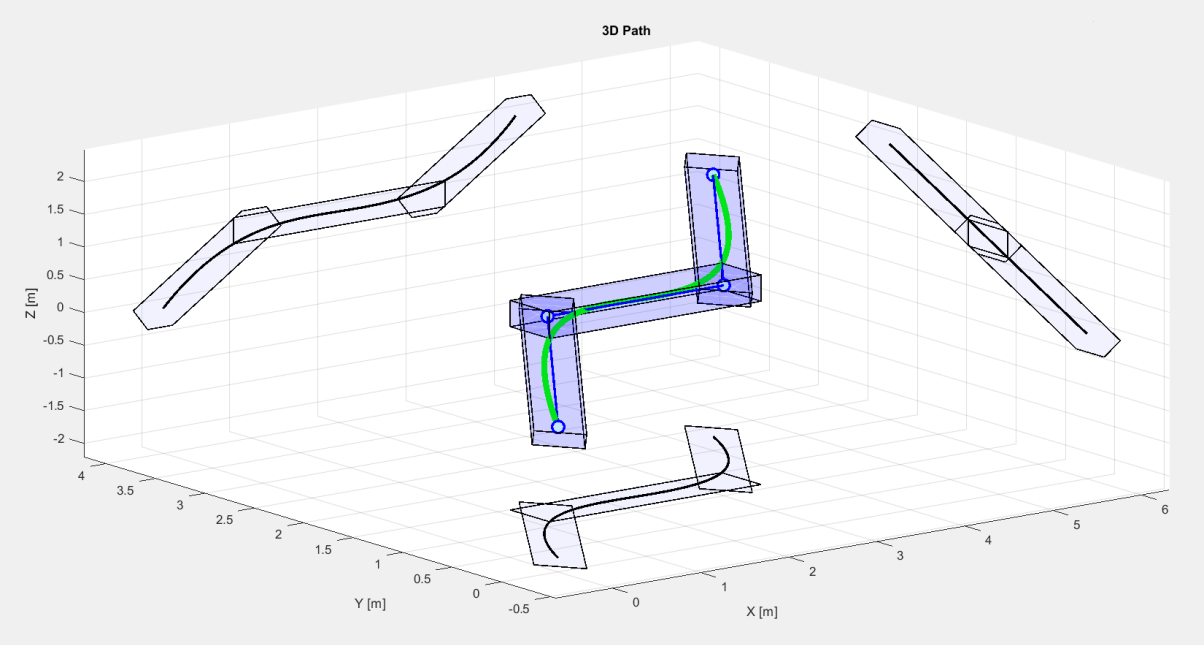


Figure 11.9: Quadrotor flight in 3D (green line) including waypoints/segments (dark blue) and bounding boxes (light blue); also projected onto each dimension.

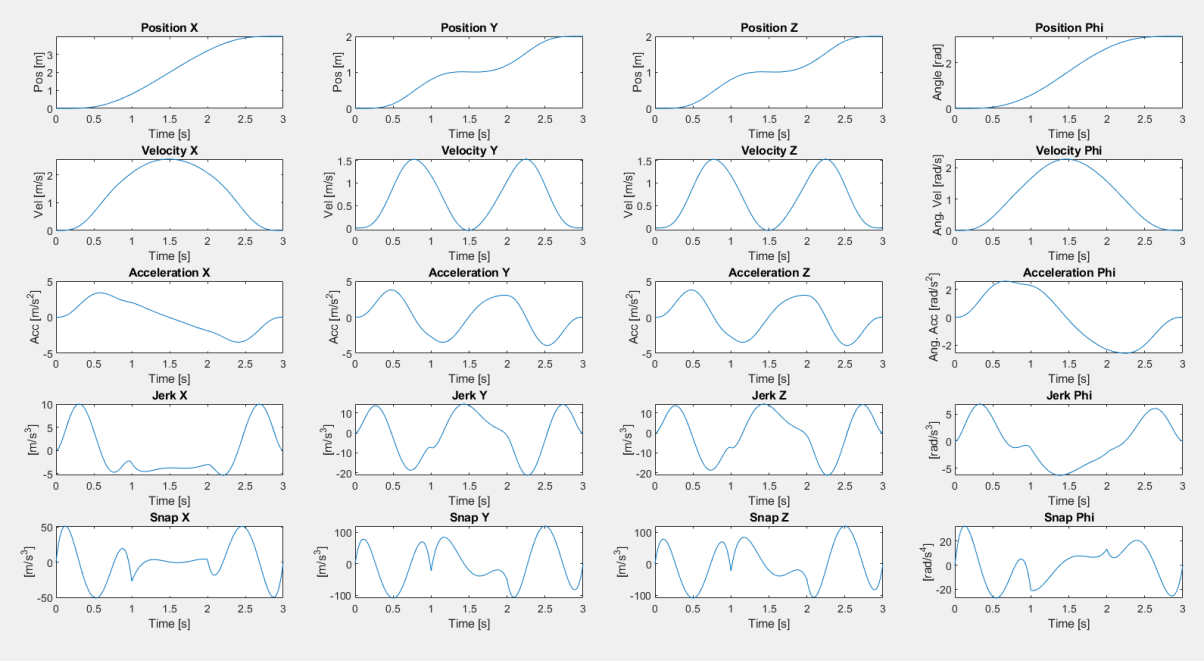


Figure 11.10: Individual trajectories of the quadrotor flight in time for all three dimensions and the angular orientation.

11.4 Low-level interface: Active Suspension Control

- *Introduction*
- *Disturbance Model: Speed Bump*
- *Implementation of Preview Information*
- *Comparison of Passive Vehicle and Active Suspension Control Using Preview Information*

11.4.1 Introduction

The concept of using future information, as described in the section *How to Incorporate Preview Information in the MPC Problem* can be applied to more advanced systems. In this example a driving vehicle is considered, equipped with sensors that measure the unevenness of the road ahead as shown in the picture below.

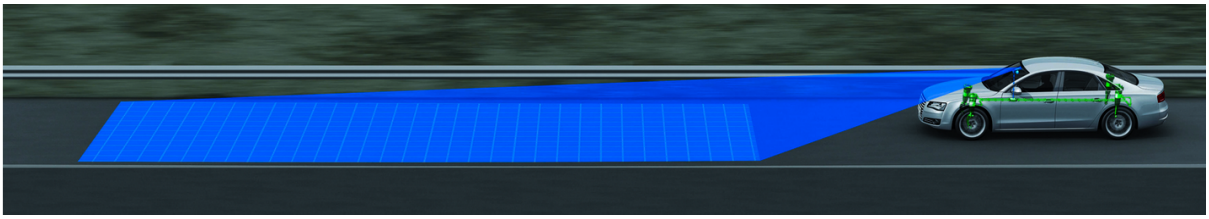


Figure 11.11: Figure borrowed from [GörSch]

The preview information can be used to improve the riding comfort, i. e. minimize the heave, pitch and roll accelerations, by actively controlling the suspension of the vehicle. This example is based on the reduced car model described in [GörSch]

The states x of the system are 'heave displacement' z_b [m], 'pitch angle' φ [rads], 'roll angle' θ [rads], 'heave velocity' $\dot{z}_b$ [m/s], 'pitch rate' $\dot{\varphi}$ [rads/s] and 'roll rate' $\dot{\varphi}$ [rads/s]. The input u [m] to the system are the 'active spring displacements'. The output y is given by the 'heave acceleration' $\ddot{z}_b$ [m/s^2], the 'pitch acceleration' $\ddot{\varphi}$ [m/s^2] and the 'roll acceleration' $\ddot{\theta}$ [m/s^2]. In the reduced model, the input contains not only the active spring displacements but also the measurements of the height profile of the upcoming road w and its first derivative $\dot{w}$.

$$
\begin{aligned}
x &:= \begin{pmatrix} \text{heave displacement [m]} \\ \text{pitch angle [rads]} \\ \text{roll angle [rads]} \\ \text{heave velocity [m/s]} \\ \text{pitch rate [rads/s]} \\ \text{roll rate [rads/s]} \end{pmatrix} \\
u &:= \begin{pmatrix} \text{active spring displacements [m]} \end{pmatrix} \\
y &:= \begin{pmatrix} \text{heave acceleration [m/s}^2\text{]} \\ \text{pitch acceleration [rads/s}^2\text{]} \\ \text{roll acceleration [rads/s}^2\text{]} \end{pmatrix}
\end{aligned}
$$

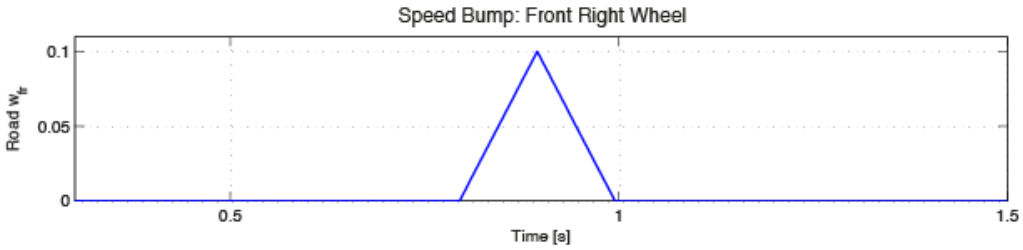
There are constraints on the actuators, i. e. minimal and maximal adjustment track, $\underline{u} = -0.04[m]$ and $\overline{u} = 0.04[m]$. This results in the following state space system:

$$
\begin{aligned}
\dot{x}(t) &= Ax(t) + B_u u(t) + B_w \begin{pmatrix} w(t) \\ \dot{w}(t) \end{pmatrix} \\
y(t) &= Cx(t) + Du(t)
\end{aligned}
$$

In the following it is shown how the FORCESPRO MATLAB Interface can be used to design a controller using preview information, substantially increasing the riding comfort compared to a vehicle with a passive suspension. The discrete vehicle model is sampled at 0.025 [s] and it is assumed that road preview information for 0.5 [s] (20 steps) is available to the controller.

11.4.2 Disturbance Model: Speed Bump

The vehicle is assumed to be driving at a constant speed of 5 [m/s] over a speed bump of length 1 [m] with a height of 0.1 [m]. The disturbance in time domain is depicted on the right side. The road bump only hits the front right wheel, while the front left wheel is not affected. The same bump will hit the rear right wheel 1.12 [s] after it hits the front wheel.



11.4.3 Implementation of Preview Information

This is a linear MPC problem with lower and upper bounds on inputs and a terminal cost term:

$$\begin{aligned} \text{minimize} \quad & x_N^T P x_N + \sum_{i=0}^{N-1} x_i^T Q x_i + u_i^T R u_i \\ \text{subject to} \quad & x_0 = x \\ & x_{i+1} = A x_i + B u_i + B_w w_i + B_w \dot{w}_i \\ & \underline{u} \leq u_i \leq \overline{u} \end{aligned}$$

At each sampling instant the initial state x and the preview information w_i and $\dot{w}_i$ change, and the first input u_0 is typically applied to the system after an optimal solution has been obtained.

```
% Parameters: First Equation RHS
parameter(1) = newParam('minusA_times_x0_minusBw_times_w_pre',1,'eq.c');
% Parameters: Preview Information
parameter(2) = newParam('pre2_w',2,'eq.c');
...
parameter(n) = newParam('pren_w',n,'eq.c');
...
parameter(N) = newParam('preN_w',N,'eq.c');
```

As described in the section *How to Incorporate Preview Information in the MPC Problem*, the parametric additive terms `g`, which corresponds to the term $B_w wi + B_w \dot{w}_i$, has to be defined. At each stage of the multistage problem, the 'g' term (containing the preview information) in the equality constraint is different, therefore we have to define a parameter for each stage. In the definition of the parameters, 'pren_w' represents the name of the term $B_w w_n + B_w \dot{w}_n$ at stage n of the multistage problem. During runtime, the preview information is mapped to these parameters.

N is the length of the prediction horizon which is set to be equal to the preview horizon. The MATLAB code below, generates the function `VEHICLE_MPC_withPreview` that takes -Ax and the additive term `g` as a calling argument and returns u_0, which can then be applied to the system:

```
%% MPC with Preview

% FORCESPRO multistage form
% assume variable ordering z(i) = [ui; x_i+1] for i=1...N-1

% Parameters: First Eq. RHS
parameter(1) = newParam('minusA_times_x0_minusBw_times_w_pre',1,'eq.c');

stages = MultistageProblem(N);
for i = 1:N

        % dimension
        stages(i).dims.n = nx+nu; % number of stage variables
        stages(i).dims.r = nx;    % number of equality constraints
        stages(i).dims.l = nu;    % number of lower bounds
        stages(i).dims.u = nu;    % number of upper bounds

        % cost
        if( i == N )
                stages(i).cost.H = blkdiag(R,P);
        else
                stages(i).cost.H = blkdiag(R,Q);
        end
        stages(i).cost.f = zeros(nx+nu,1);

        % lower bounds
        stages(i).ineq.b.lbidx = 1:nu;        % lower bound acts on these indices
        stages(i).ineq.b.lb = umin*ones(4,1); % lower bound for the input signal

        % upper bounds
        stages(i).ineq.b.ubidx = 1:nu;        % upper bound acts on these indices
        stages(i).ineq.b.ub = umax*ones(4,1); % upper bound for the input signal

        % equality constraints
        if( i < N )
                stages(i).eq.C =  [zeros(nx,nu), Ad];
        end
        stages(i).eq.D = [Bdu, -eye(nx)];

        % Parameters for Preview
        if (i > 1)
                parameter(i)  = newParam(['pre',num2str(i),'_w'],i,'eq.c');
        end

end

% define outputs of the solver
outputs(1) = newOutput('u0',1,1:nu);

% solver settings
codeoptions = getOptions('VEHICLE_MPC_withPreview');

% generate code
generateCode(stages,parameter,codeoptions,outputs);
```

You can find the MATLAB code of this example to try it out for yourself in the `examples` folder that comes with your client.

11.4.4 Comparison of Passive Vehicle and Active Suspension Control Using Preview Information

In Figure 11.12, Figure 11.13 and Figure 11.14, the accelerations in the direction heave, pitch and roll respectively are depicted. The blue graphs represent the the outputs when using MPC with preview information, while the red ones show the results of the simulation without using the preview information. One can see that taking the preview information into account leads to decreased maximal accelerations.

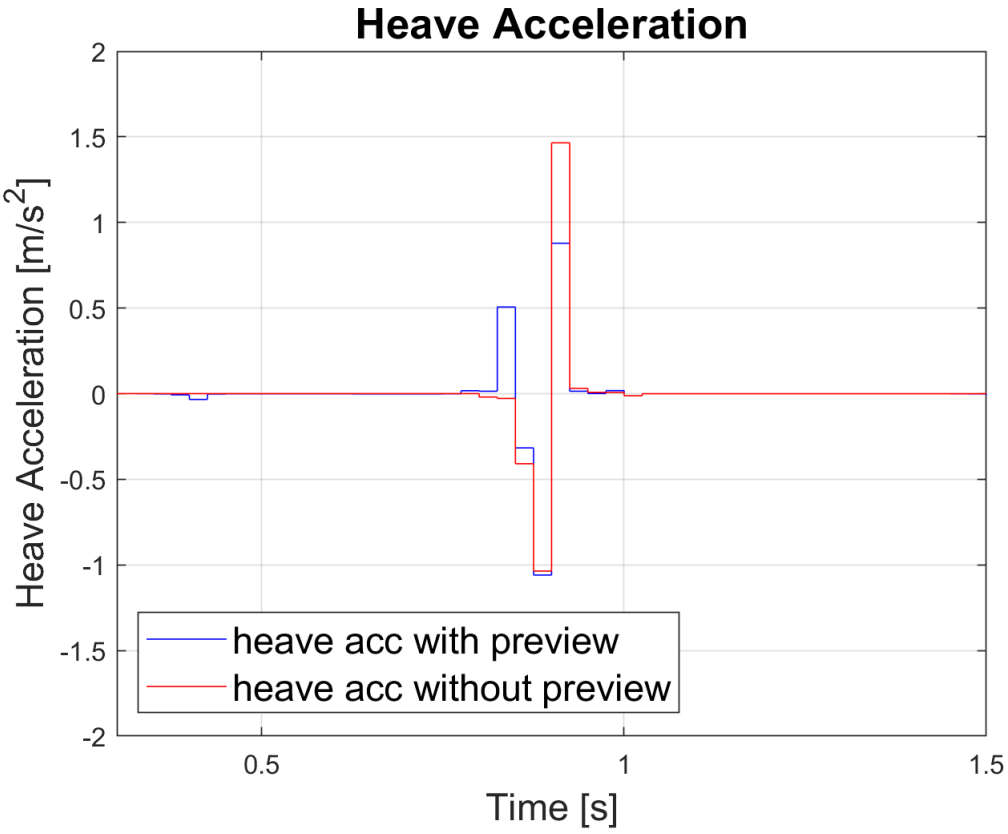


Figure 11.12: Acceleration in heave direction

Applying Model Predictive Control with Preview using FORCESPRO the riding comfort is improved significantly with minimum effort for designing the controller and generating code which can be deployed on any embedded automotive control unit.

The four graphs in Figure 11.15, Figure 11.16, Figure 11.17 and Figure 11.18 below show the input signal on each of the four actuators. One can see that model predictive controller starts lifting the front right part of the vehicle body as soon as the bump is in sight of the preview sensor, i. e. at time $t = 0.3$ [s]. This is 0.5 seconds, the length of the preview horizon, before the front right wheel hits the bump at time $t = 0.8$ [s]. This causes a better absorption of the shock and therefore reduced accelerations. The input constraints are also satisfied and u never exceeds the admitted range.

11.5 Low-level interface: Robust estimation (Kalman filter)

- *System Description*
- *Robust Kalman filter*

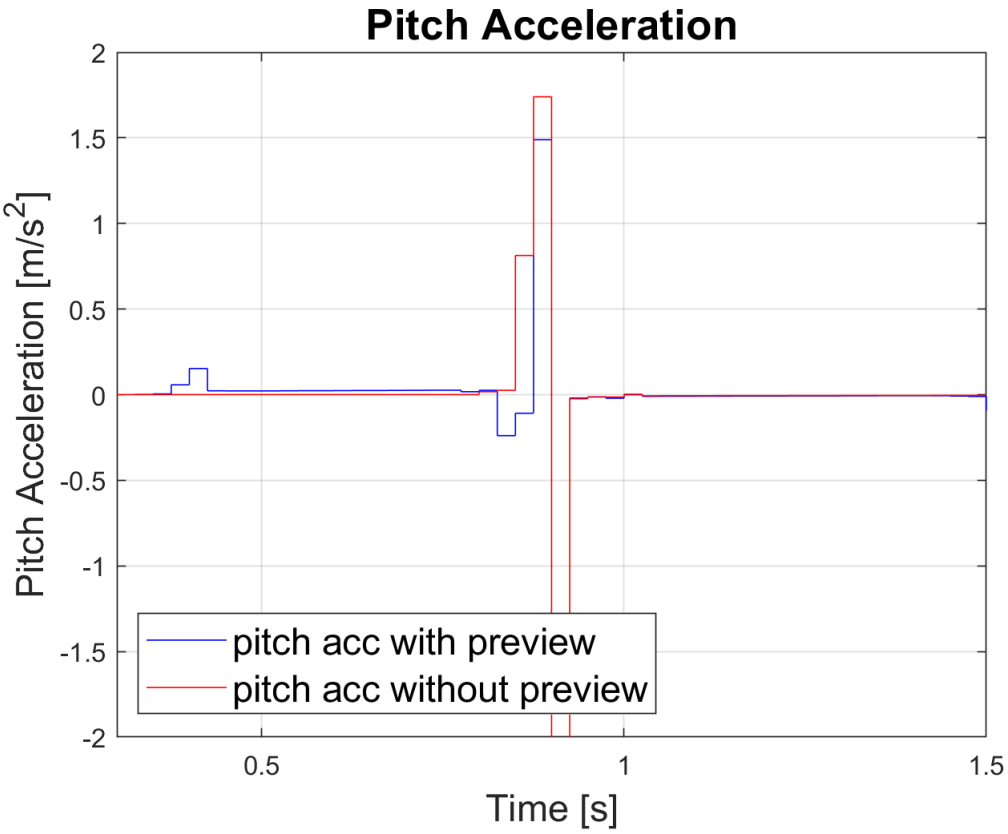


Figure 11.13: Acceleration in pitch direction

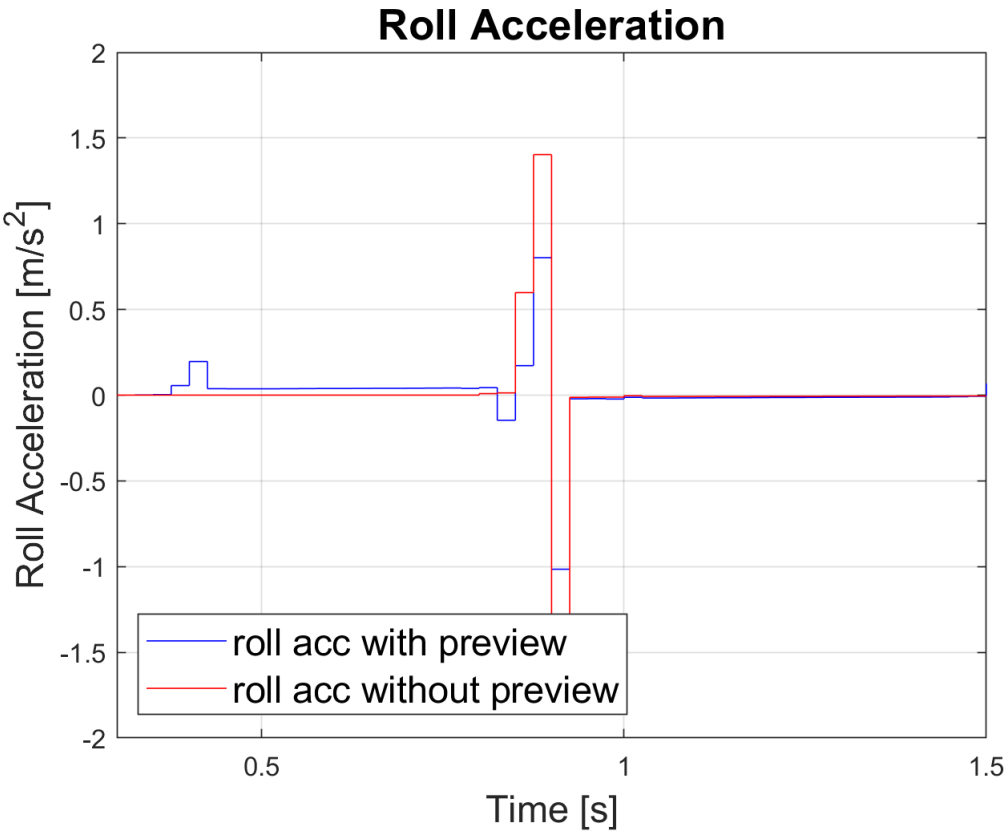


Figure 11.14: Acceleration in roll direction

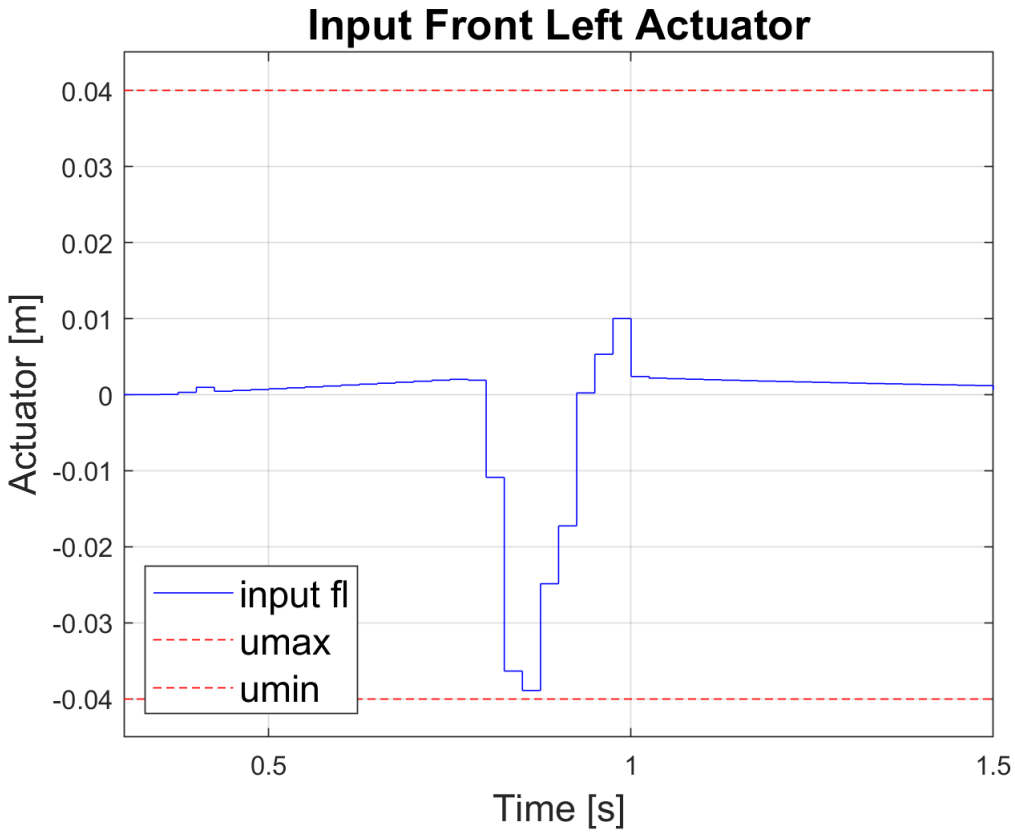


Figure 11.15: Input front left actuator

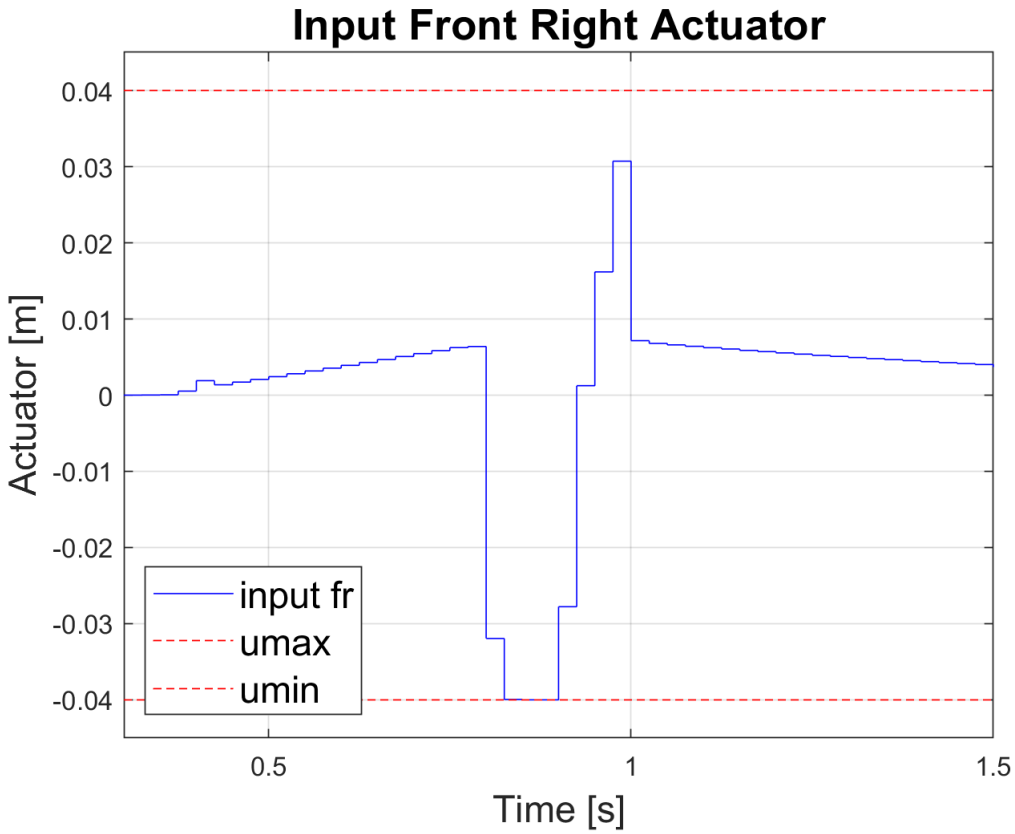


Figure 11.16: Input front right actuator

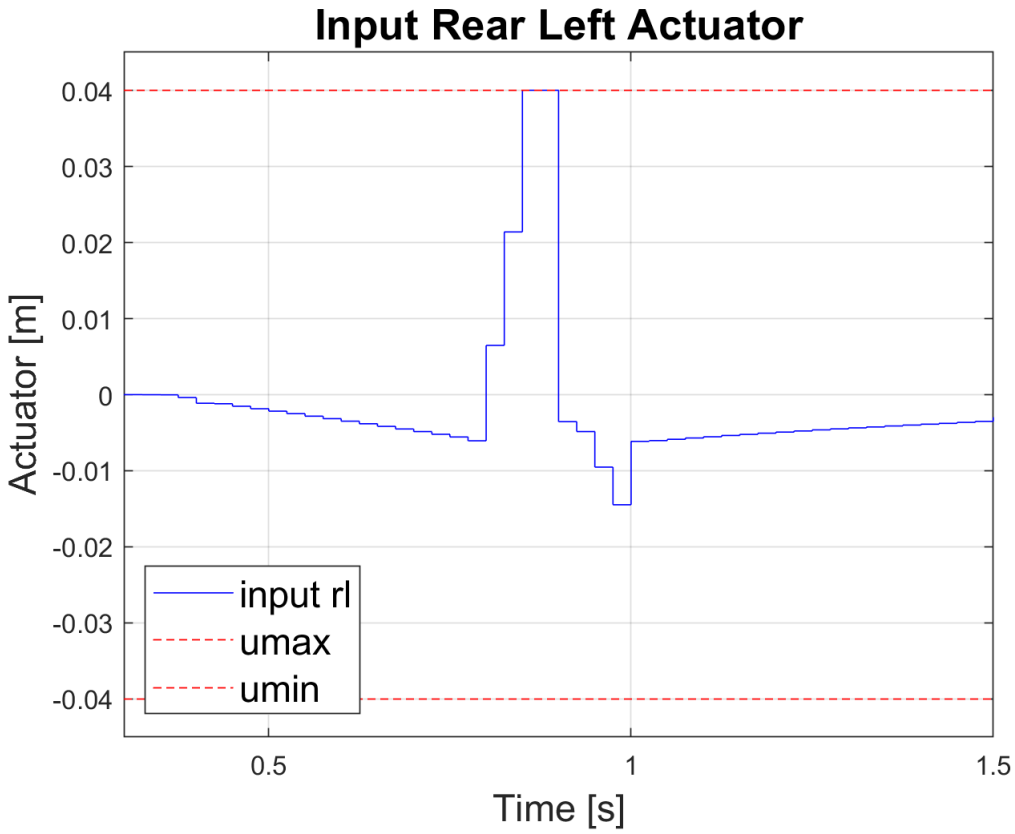


Figure 11.17: Input rear left actuator

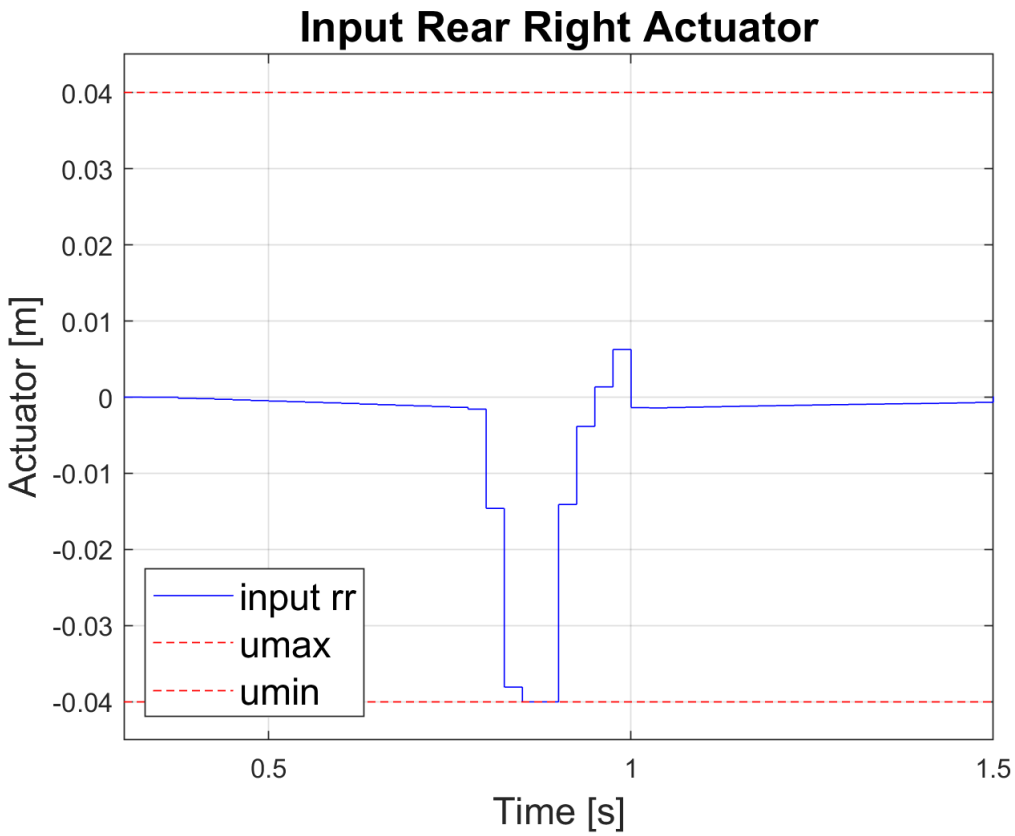


Figure 11.18: Input rear right actuator

· *Simulation and Comparison*

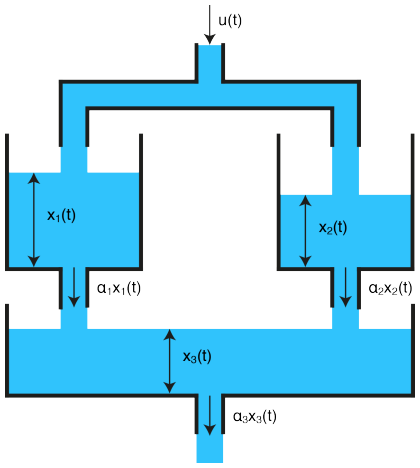
11.5.1 System Description

In this example we consider the water tank system depicted on the right. Tank 1 has one input flow and one output flow. Also tank 2 has one input flow and one output flow. Tank 3 has two input flows and one output flow. The system dynamics are represented via the first equation below. As an output z we have a measurement of the level of tank 1 and of the level of tank 3.

$$x^+ = Ax + Bu + v = \begin{pmatrix} 1-\alpha_1 & 0 & 0 \\ 0 & 1-\alpha_2 & 0 \\ \alpha_+ & \alpha_2 & 1-\alpha_3 \end{pmatrix} x + \begin{pmatrix} 0.5 \\ 0.5 \\ 0 \end{pmatrix} u + v$$
$$z = Hx + w + y = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} x + w + y$$

The states of the system are $x = \begin{pmatrix} x_1 & x_2 & x_3 \end{pmatrix}^T$ ‘$and as an input the flow : math : ‘u$ is given. There is a process noise v and a measurement noise w, both are Gaussian Random Variables with mean 0 and variance Q and R, i. e. $v \sim \mathcal{N}(0, Q)$ and $w \sim \mathcal{N}(0, R)$. The sparse signal y, which is used to model sensor failures, distorts the measurement signal additionally.

The goal of this example is to show, that the standard Kalman Filter is not working that good anymore if sensor failures are present. There does not exist an analytic solution to the problem if the disturbance y is present. Using the robust Kalman Filter, i. e. replacing the standard measurement update step with an extended optimization problem, which is solved by FORCESPRO, the filter is robust against y and the estimated states are much more accurate compared to the standard Kalman Filter. To process the measurement data online, the optimization problem has to be solved in a sufficiently short amount of time.



11.5.2 Robust Kalman filter

Recall that the standard Kalman Filter, which would be applied if disturbance signal y were not present, consists of two steps: First a prediction step is made, where a predicted stated $x^p(k)$ is calculated based on the estimated state $x^m(k-1)$. Additionally, the predicted variance $P_p(k)$ gets calculated in the prediction step. The measurement step returns the variance $P_m(k)$ and the state estimate $x^m(k)$. This state estimate $x^m(k)$ is basically the solution of the optimization problem

$$\begin{aligned} \text{minimize} \quad & w^T R^{-1} w + (x - \hat{x}_p)^T P^{-1} (x - \hat{x}_p) \\ \text{subject to} \quad & z = Hx + w \end{aligned}$$

In this example, we assume that out of 100 measurements the sensors of tank 1 gand tank 3 gives each 5 bogus signals. In order to make the state estimator robust against the sensor failures y, we solve the following convex optimization problem at every time instance

$$\begin{aligned} \text{minimize} \quad & w^T R^{-1} w + (x - \hat{x}_p)^T P^{-1} (x - \hat{x}_p) + \lambda \|y\|_1 \\ \text{subject to} \quad & z = Hx + w + y \end{aligned}$$

In the optimization problem w, x and y are optimization variables. The cost function of the optimization problem is extended with the l_1-penalty which is non-quadratic. The value $\lambda \geq 0$ is a tuning parameter. For λ large enough, the solution of the optimization problem has $y = 0$ and therefore the estimates of the robust Kalman Filter coincides with the standard Kalman Filter solution. This optimization problem can be transformed as described in here. We transform this problem to the form required by FORCESPRO, which reads as

$$\begin{aligned} \text{minimize} \quad & \frac{1}{2} \tilde{z}^T \tilde{H} \tilde{z} + f^T \tilde{z} \\ \text{subject to} \quad & D\tilde{z} = z \\ & A\tilde{z} \leq b \end{aligned}$$

where the optimization variable is given by $\tilde{z} = \begin{pmatrix} x^T & w^T & y^T & e^T \end{pmatrix}^T$. Please find below the MATLAB code to generate the solver for the optimization problem with FORCESPRO. The covariance matrix P^{-1} is updated at every time step and therefore the problem can't be solved explicitly. In this problem three parameters need to be defined, which are H, f - containing the predicted covariance and the predicated state - and c - contains the current measurement.

```
%% Create FORCESPRO Solver for Measurement Update Step
% assume variable ordering z = [x,w,y,e]

% Create multistage struct
stages = MultistageProblem(1);

% Dimension
[ny nx] = size(H);
nw = ny;
ne = ny;
stages(1).dims.n = nx+nw+ny+ne; % number of stage variables
stages(1).dims.r = ny;          % number of equality constraints
stages(1).dims.p = 2*ne;        % number of polytopic constraints

% Polytopic bounds
stages(1).ineq.p.A = [zeros(ny,nx), zeros(ny,nw), lambda*eye(ny), -eye(ne); ...
                      zeros(ny,nx), zeros(ny,nw), -lambda*eye(ny), -eye(ne)];
stages(1).ineq.p.b = zeros(2*ne,1);

% Equality constraints
stages(1).eq.D = [H, eye(nw), eye(ny), zeros(ne)];

% Parameters
params(1) = newParam('H_i',1,'cost.H');
params(2) = newParam('f_i',1,'cost.f');
params(3) = newParam('z_i',1,'eq.c');

% Output
outputs(1) = newOutput('x_hat_RKF',1,1:3);

% Code Generation
codeoptions = getOptions('Robust_KF');
generateCode(stages,params,codeoptions,outputs);
```

You can find the MATLAB code of this example to try it out for yourself in the `examples` folder that comes with your client.

11.5.3 Simulation and Comparison

In the simulation the optimization problem has to be solved at every time instance. In the prediction step the state x^p is calculated based on the estimation of the current state. Also the the variance is updated in every prediction step. In the measurement update step the estimated state x^m is calculated based on the predicted state, its predicted variance and the current measurement z by the function `Robust_KF()` generated by FORCESPRO.

```
for i = 2:(N+1)

    % Prediction Step
    x_p_RKF = Ak(:,:,i-1)*x_hat_RKF(:,i-1)+B*u(i-1);
    P_p_RKF(:,:,i) = Ak(:,:,i-1)*P_hat_RKF(:,:,i-1)*Ak(:,:,i-1)' + Q;

    % Measurement Update Step - Optimization Problem
    problem.H_i = [2*inv(P_p_RKF(:,:,i)),zeros(nx,nw+ny+ne); ...
                   zeros(ny,nx),2*R_inv,zeros(ny,ny+ne); ...
                   zeros(ny+ne,nx+nw+ny+ne)];
    problem.f_i = [-2*(inv(P_p_RKF(:,:,i))*x_p_RKF); ...
                   zeros(nw,1); ...
                   zeros(ny,1); ...
                   ones(ne,1)];
    problem.z_i = z(:,i);
    [solverout,~,info] = Robust_KF(problem);
    solve_time(i-1) = info.solvetime;
    P_hat_RKF(:,:,i) = P_p_RKF(:,:,i);
    x_hat_RKF(:,i) = solverout.x_hat_RKF;

end
```

In the plots in Figure 11.19, Figure 11.20 and Figure 11.21 respectively, the estimated states are depicted. The estimates calculated via the robust Kalman Filter, in blue, are much more accurate then the standard approach. The peaks in the red line indicate sensor failures against which the standard Kalman Filter is not robust.

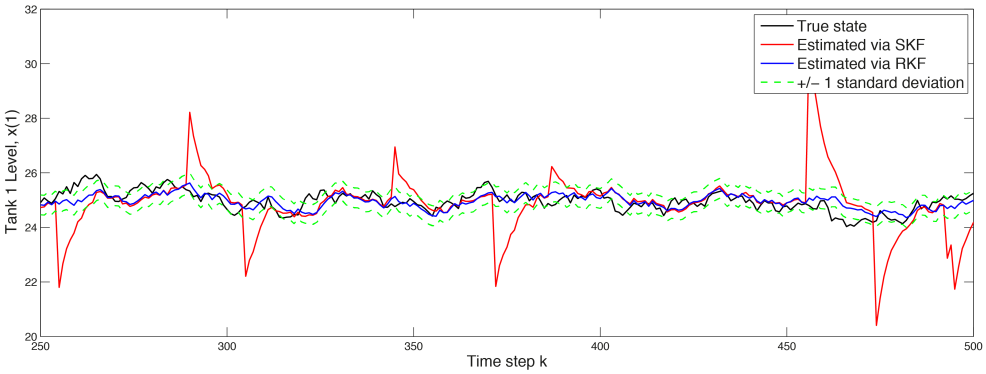


Figure 11.19: Estimated state $x(1)$

The impact on the RMS error magnitude of the robust Kalman Filter can be seen in the plots in Figure 11.22, Figure 11.23 and Figure 11.24. The magnitude of the robust Kalman Filter depicted in blue, is reduced by $\sim 65\%$ for state 1, $\sim 12\%$ for state 2, $\sim 61\%$ for state 3 (this values vary).

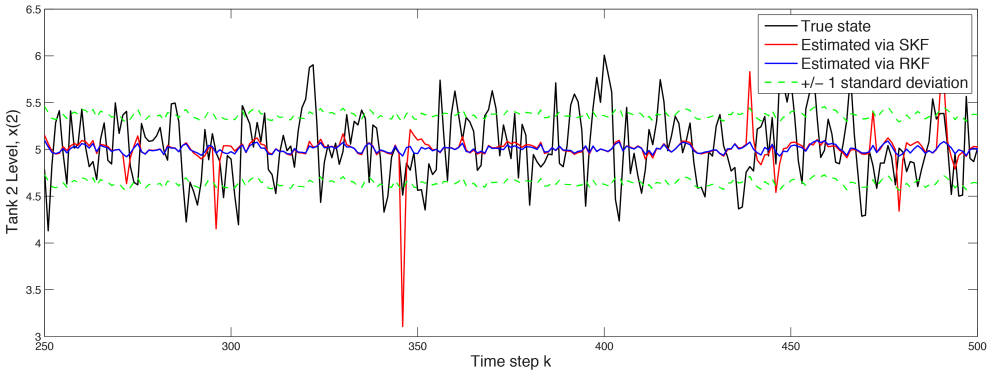


Figure 11.20: Estimated state $x(2)$

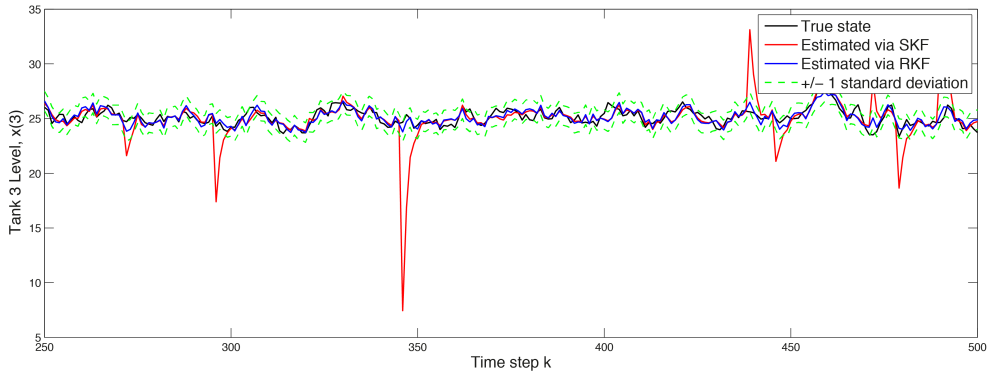


Figure 11.21: Estimated state $x(3)$

Applying online optimization with FORCESPRO improves the quality of the state estimations significantly.

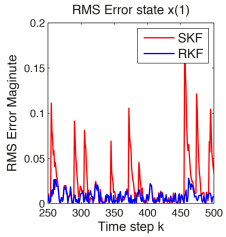


Figure 11.22: RMS error for $x(1)$

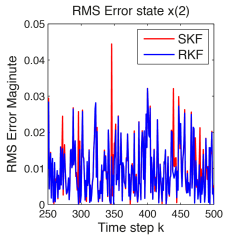


Figure 11.23: RMS error for $x(2)$

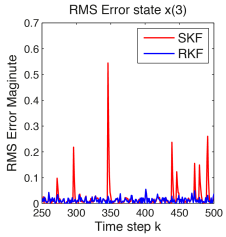


Figure 11.24: RMS error for $x(3)$

With FORCESPRO convex optimization can be embedded directly in signal processing algorithms that run online, with strict real-time deadlines, even at rates of tens of kilohertz. In this example the optimization problem is solved in $\sim 0.1ms$.

11.6 Low-level interface: Spacecraft Rendezvous (MATLAB & Python)

- *Introduction*
- *Model*
- *Constraints*
- *Objective*
- *Spacecraft Rendezvous Maneuvers with and without 1-Norm Cost*

11.6.1 Introduction

This example uses the concepts described in the subsections *HOW TO: Implement an MPC Controller with a Time-Varying Dynamics* and *How to Implement 1-Norm and Infinity-Norm Cost Functions*.

The goal is to design a controller to perform a spacecraft rendezvous operation, where a controlled chaser spacecraft is performing rendezvous with a passive target that is orbiting around Mars. Using a time-varying prediction model allows to perform spacecraft maneuvers in elliptical orbits and allows the controller to be updated when the are changes in the system parameters or control objectives. This example is based on the models described in [HarMac14] and the references therein.

11.6.2 Model

The Yamanaka-Ankersen (Y-A) equations are used to describe the dynamics, where the six states x of the system represent the relative position and velocity of the chaser with respect to the target in the three dimensions. These equations apply in elliptical orbits, but are time-varying in terms of the true anomaly, v, of the target, i.e. the model is given by

$$x_{k+1} = A(v)x_k + B(v)u_k$$

and the requirement is that the state at the end of the horizon is at the target. The plant input is modeled as an impulsive change in velocity, such that

$$B(v) = A(v)\begin{pmatrix} 0 \\ I_3 \end{pmatrix}$$

You can find the MATLAB code of this example to try it out for yourself in the `examples` folder that comes with your client.

11.6.3 Constraints

The three impulsive control inputs can give a maximum change in velocity of 5 meters per second along each axis. In addition, the chaser spacecraft is required to remain within a cone of vision of 20 degrees from the target and must not go behind the target to facilitate the docking maneuver.

11.6.4 Objective

The goal of the controller is to balance the following objectives:

- the chaser should be always as close as possible to the target,
- use as little fuel as possible to get there.

The second objective is more important, hence it is weighed higher. We consider two types of cost functions: one where all the terms are weighed using standard quadratic penalties; and one where the inputs are penalized using the 1-norm, which better reflects the propellant consumption being directly proportional to delivered thrust and also attempts to minimize the use of the actuators. In order to implement the 1-norm cost we need to add slack variables and additional constraints as described in *How to Implement 1-Norm and Infinity-Norm Cost Functions*.

The following code shows how to generate an MPC controller for the spacecraft rendezvous problem with a time-varying model and a 1-norm penalty on the actuators.

Matlab

Python

```
%% system
nx = 6;         % velocities and acceleration in 3 axes
nu = 3;         % thrusters on 3 axis

%% MPC setup
N = 31;
Q = diag([0.001 0.001 0.001 0 0 0]);
R = diag([15000 15000 15000]);
umin = -5;
umax = 5;
cone_angle = 20*pi/180;
Hx = [1 0 0 0 0 0; ...
      cone_angle 0 1 0 0 0; ...       % Visibility cone
      cone_angle 0 -1 0 0 0];         % Visibility cone
hx = [0; 0; 0];

xs = [-1000; 0; 0; 0; 0; 0];    % target state (1km away)
us = zeros(3,1);

%% FORCESPRO multistage form - 2-norm
% assume variable ordering z(i) = [ui, x_i+1, eu_i] for i=1...N-1

stages = MultistageProblem(N-1);

params(1) = newParam('minusA_times_x0',1,'eq.c'); % RHS of first eq. constr. is a␣
↪parameter: z1=-A*x0

for i = 1:N-1

        % dimension
        stages(i).dims.n = nx+2*nu; % number of stage variables
        stages(i).dims.r = nx;      % number of equality constraints
        stages(i).dims.l = nu;      % number of lower bounds
        stages(i).dims.u = nu;      % number of upper bounds
        stages(i).dims.p = 3+2*nu;  % number of polytopic constraints

        % cost
        stages(i).cost.H = blkdiag(zeros(nu),Q,zeros(nu));
        stages(i).cost.f = [zeros(nu,1); -Q*xs; ones(nu,1)];

        % lower bounds
        stages(i).ineq.b.lbidx = 1:nu;      % lower bound acts on these indices
        stages(i).ineq.b.lb = umin*ones(nu,1); % lower bound for this stage variable

        % upper bounds
        stages(i).ineq.b.ubidx = 1:nu;      % upper bound acts on these indices
        stages(i).ineq.b.ub = umax*ones(nu,1); % upper bound for this stage variable

        % polytopic bounds
        stages(i).ineq.p.A  = [ zeros(3,nu), Hx, zeros(3,nu); ...
                                R, zeros(nu,nx), -eye(nu); ...
                                -R, zeros(nu,nx), -eye(nu)];
        stages(i).ineq.p.b  = [ hx; R*us; -R*us ];

        % equality constraints
        if( i > 1 )
            stages(i).eq.c = zeros(nx,1);
```

(continues on next page)

(continued from previous page)

```
        end
        params(2*i) = newParam(['D_',num2str(i)],i,'eq.D');
        if( i < N-1 )
            params(2*i+1) = newParam(['C_',num2str(i)],i,'eq.C');
        end

end
```

```
nx = 6
nu = 3

N = 31
Q = np.diag([0.001, 0.001, 0.001, 0, 0, 0])
R = np.diag([15000, 15000, 15000])

umin = -5
umax = 5
cone_angle = 20*np.pi/180
Hx = np.array([[1, 0, 0, 0, 0, 0], [cone_angle, 0, 1, 0, 0, 0], [cone_angle, 0, -1, 0,
↪ 0, 0]])
hx = np.zeros(3)

xs = np.array([-1000, 0, 0, 0, 0, 0])
us = np.zeros(3)

# assume variable ordering z[i-1] = [ui, x_i+1, eu_i] for i=1...N-1

stages = forcespro.MultistageProblem(N-1)
stages.newParam("minusA_times_xinit", [1], 'eq.c') # 1-indexed

for i in range(N-1):

    # dimensions
    stages.dims[i]['n'] = nx + 2*nu # number of stage variables
    stages.dims[i]['r'] = nx        # number of equality constraints
    stages.dims[i]['l'] = nu        # number of lower bounds
    stages.dims[i]['u'] = nu        # number of upper bounds
    stages.dims[i]['p'] = 3 + 2*nu  # number of polytopic constraints

    # cost
    stages.cost[i]['H'] = block_diag(np.zeros((nu, nu)), Q, np.zeros((nu, nu)))
    stages.cost[i]['f'] = np.concatenate([np.zeros(nu), -Q @ xs, np.ones(nu)])

    # lower bounds
    stages.ineq[i]['b']['lbidx'] = range(1, nu + 1)   # 1-indexed, lower bound acts␣
↪on these indices
    stages.ineq[i]['b']['lb'] = umin*np.ones(nu) # lower bound for this stage variable

    # upper bounds
    stages.ineq[i]['b']['ubidx'] = range(1, nu + 1)   # 1-indexed, upper bound acts␣
↪on these indices
    stages.ineq[i]['b']['ub'] = umax*np.ones(nu) # upper bound for this stage variable

    # polytopic bounds
    stages.ineq[i]['p']['A'] = np.block([[np.zeros((3, nu)), Hx, np.zeros((3, nu))],
                                         [R, np.zeros((nu, nx)), -np.eye(nu)],
```

(continues on next page)

(continued from previous page)

```
                                        [-R, np.zeros((nu, nx)), -np.eye(nu)]])
    stages.ineq[i]['p']['b'] = np.concatenate([hx, R @ us, -R @ us])

    # equality constraints
    if i > 0:
        stages.eq[i]['c'] = np.zeros(nx)

    stages.newParam(f"D_{i+1}", [i+1], 'eq.D') # 1-indexed

    if i < N-2:
        stages.newParam(f"C_{i+1}", [i+1], 'eq.C') # 1-indexed
```

11.6.5 Spacecraft Rendezvous Maneuvers with and without 1-Norm Cost

The simulation describes a rendezvous maneuver were the chaser starts 15km away from the target spacecraft and the goal is to approach the target to within 1000 meter distance, while respecting the constraints, to start the docking maneuver. The target is modeled as being in a Keplerian orbit around Mars with an orbital radius of 3,600,000 meters. The controller sampling time is 200s but the target and chaser dynamics are simulated in intervals of 1s for illustration purposes. The plots in Figure 11.25 illustrates the behavior of the controlled spacecraft with standard quadratic cost, while the plots in Figure 11.26 shows the behavior of the controller when the quadratic cost on the actuators is swapped with a 1-norm penalty. Notice the sparsity in the actuation commands - the thrusters are only engaged when necessary to keep the spacecraft within the cone of visibility of the target.

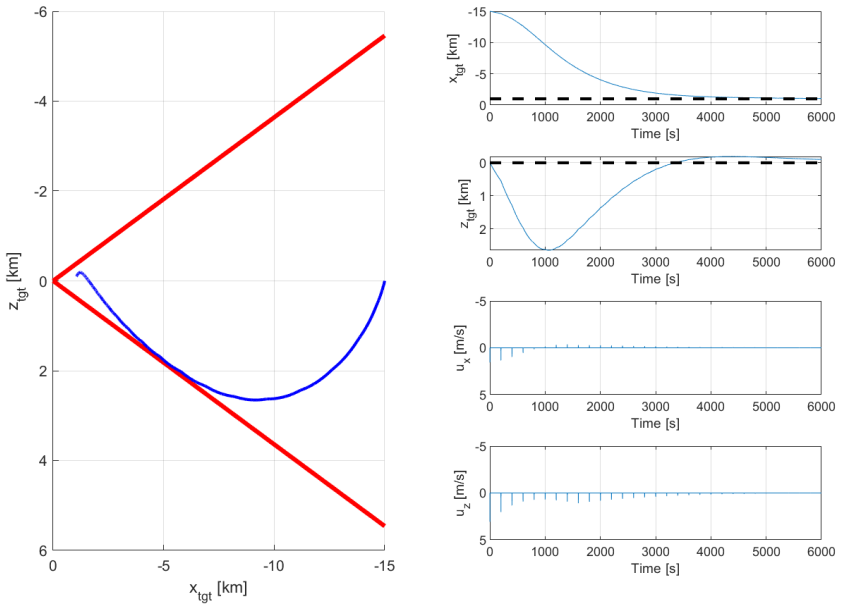


Figure 11.25: Behavior with quadratic cost.

11.7 Low-level interface: DC/DC converter

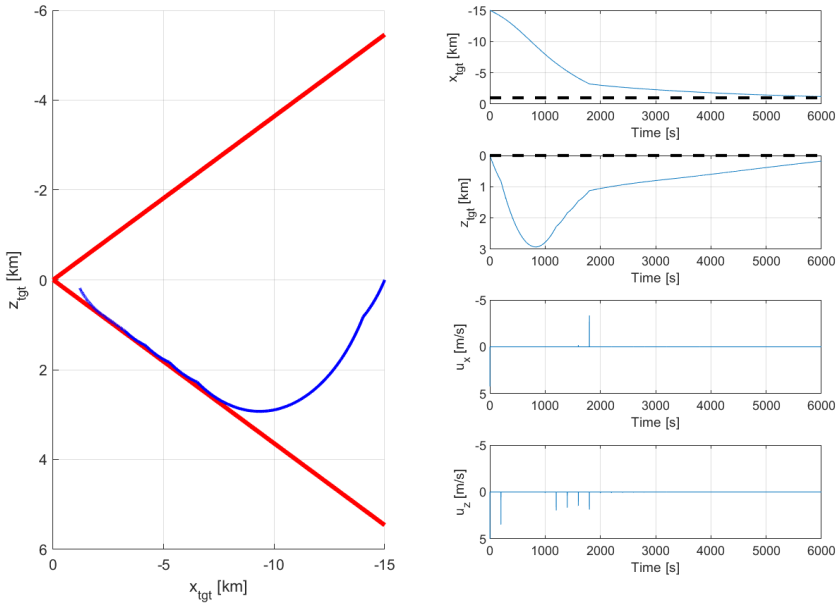


Figure 11.26: Behavior with cost given by 1-norm.

- *Example Overview*
- *Special Requirements*
- *Introduction - Control of a DC/DC Converter*
- *Control Objective by Using Model Predictive Control*
- *Model Predictive Control Design via FORCESPRO MATLAB® Interface*
- *Simulation of the PLECS® Model with Model Predictive Control*
- *Comparison of Model Predictive Control and PI Control*

11.7.1 Example Overview

The example starts by describing the power electronics of the DC/DC converter and how the control oriented model of the system is derived. Then the potential advantages of model predictive control over a conventional PI controller are discussed. Afterwards the design of the MPC controller using FORCESPRO is presented. Finally, the simulation setup is explained and the simulation results using PI and MPC are compared.

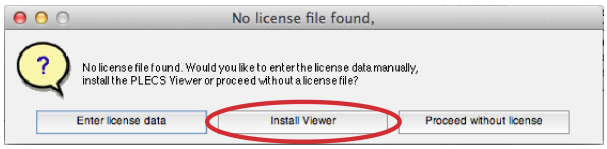
- *Introduction*: General introduction to the example.
- *Control Objective*: What can be gained by applying MPC with FORCESPRO.
- *MPC via FORCESPRO*: How to generate a solver with FORCESPRO for the power electronic converter.
- *Simulation*: Illustration on how to simulate the system with the generated controller.
- *Comparison*: Discussion of the results of the simulation.

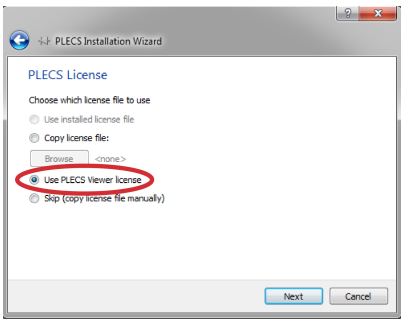
11.7.2 Special Requirements

For the simulation of the power electronic converter in this example PLEXIM provided their software PLECS®. PLECS® is the tool for high-speed simulations of power electronic systems. To simulate this example, PLECS Blockset with a viewer license is required. Please follow the instructions on how to install PLECS® below.

PLECS Blockset installation instructions:

- Download PLECS® Blockset installation script available from here.
- Download the required PLECS® Blockset package file here and save it in the same directory as the file installplecs.m.
- Run the file installplecs.m in MATLAB® from the command line.
- During the installation a dialog asks where to save 'PLECS'. Choose a location which is in the MATLAB® search path.
- During the installation a dialog asks for a license. Install the 'viewer license' as shown in the figures below.





Once the installation is completed you are ready to simulate the files provided with this example.

11.7.3 Introduction - Control of a DC/DC Converter

An important field of application for model predictive control are power electronic systems. In this example a typical DC/DC converter which supplies an isolated DC voltage to a telecom system is considered. Assume that the input voltage of the two-transistor forward converter, depicted below on the left, is a constant voltage U_{IN} delivered by a previous PFC rectifier stage. The load attached to the converter has an ohmic-capacitive characteristic.

This two-transistor forward converter can be modelled as a buck converter from which it is more convenient to derive a control oriented model. The buck converter has only one switch and the input voltage U_{in} is the actual input voltage scaled by the transformer turn ratio. The equivalent circuit is depicted on the right in the figure below.

The states of the control oriented model, which is used as a model for the predictive controller, are the inductor current i_L and the capacitor voltage u_C. Further there are the input signal d and the disturbances in the input voltage and the load current $w = \begin{pmatrix} u_{in} & i_{Load} \end{pmatrix}^T$. As an output signal the states i_L and u_C as well as the output voltage uout are considered. The small signal

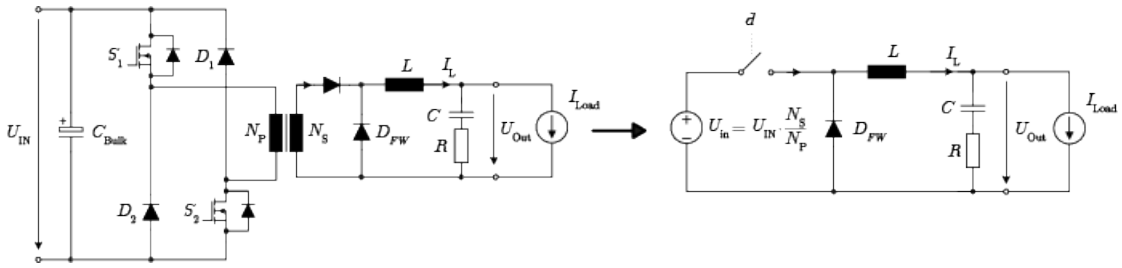


Figure 11.27: Based on the lecture material Power Electronic Systems II, Institute for Power Electronic Systems, ETH Zürich

model (small signals are marked with a hat) in state-space form reads as:

$$\frac{d}{dt}\hat{x} = \begin{pmatrix} -\frac{R}{L} & -\frac{1}{L} \\ \frac{1}{C} & 0 \end{pmatrix} \hat{x} + \begin{pmatrix} \frac{U_{in}}{L} \\ 0 \end{pmatrix} \hat{d} + \begin{pmatrix} \frac{D}{L} & -\frac{R}{L} \\ 0 & -\frac{1}{C} \end{pmatrix} \hat{w}$$

$$\hat{y} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ R & 1 \end{pmatrix} \hat{x} + \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 0 & -R \end{pmatrix} \hat{w}$$

$$\Updownarrow$$

$$\frac{d}{dt}\hat{x} = A \cdot \hat{x} + B1 \cdot u + B2 \cdot \hat{w}$$

$$\hat{y} = C \cdot \hat{x} + \begin{pmatrix} D2 \\ D4 \end{pmatrix} \cdot \hat{w}$$

11.7.4 Control Objective by Using Model Predictive Control

The converter should provide a constant output voltage U_{Out} of 60 V while delivering the power required by the load. The nominal load current I_{Load} is 22 A. The input voltage U_{in} is constant at level 144 V, while the load resistance varies in the range $[1.5, 5]\Omega$.

Conventionally the output voltage of the Buck Converter was controlled by a PI controller. In the first plot below, the current i_L in the inductor is shown, when the resistance in the load is reduced from 5Ω to 1.5Ω, i. e. from upper bound to the lower bound of the possibly required load resistance. The red curve represents the current in the inductor. Also the change in the output voltage is depicted when changing the load resistance.

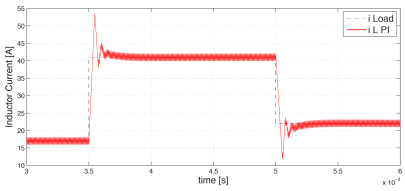


Figure 11.28: Inductor current vs. time

From Figure 11.28 and Figure 11.29 one can see that the current in the inductor has a high overshoot and the output voltage has a relatively long settling time when a change in the load resistance occurs.

Important: Some of the potential benefits of model predictive control are the following

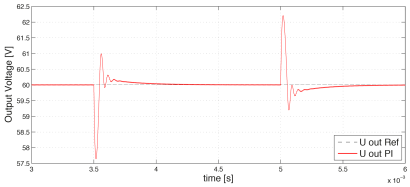


Figure 11.29: Output voltage vs. time

- Below it is shown that the size of the converter can be reduced by using a MPC controller designed with FORCESPRO. With the MPC controller it will be possible to limit the current in the inductor. With the warranty that the current does not exceed a certain upper bound, a smaller inductor can be built in and the costs are reduced.
- Also the controller designed with FORCESPRO will calculate the optimal input at every time step. The performance of the system is increased, i. e. less overshoot and faster settling time.

11.7.5 Model Predictive Control Design via FORCESPRO MATLAB® Interface

To design the FORCESPRO controller, the MPC setup has to be defined first. Below the requirements are shown. A prediction horizon of 25 is chosen. In the cost function R penalizes the deviation of the input signal from its reference value. The matrix Q penalizes the deviation of the states from its reference values. Notice that Q is defined such that a deviation of the inductor current to its reference value is less penalized than a deviation of the output voltage to its reference value. The input signal d to the PWM is limited to [0, 1], while the inductor current should not exceed a current of 42 A. This overshoot limitation concerns the average inductor current. Below one can see, that this limit is exceeded by half of the currents peak-to-peak value. The constraints are consistently defined with the model, i. e. a current reduction by -20 A and a current enhancement by 20 A is allowed at most. This is equivalent to a current in the inductor in the range of [2, 42] A.

```
% MPC Setup
N = 25;
Q = [.01, 0; 0, 10];
R = 1;
nx = 2;
nu = 1;

% Constraints
umin = 0;
umax = 1;
xmin = -20;
xmax = 20;
```

Next, the multistage problem is formulated. In this example, there exists a linear term f in the cost function due to the variable load, i. e. the steady-state inductor current changes. The cost function therefore reads as

$$(x^+ - x_{ref})^T Q (x^+ - x_{ref}) + (u - u_{ref})^T R (u - u_{ref})$$

To solve the optimization problem, the reference values need to be re-calculated at every time step. Below the parameters of the problem are marked red. The optimization variable of the

multistage problem is $z_i = \begin{pmatrix} u_i & x_{i+1} \end{pmatrix}^T$, where u is the input signal given to the system.

$$\begin{aligned} \text{minimize} \quad & \sum_{i=1}^{N} \frac{1}{2} z_i^T H_i z_i + f_i^T z_i && \text{(separable objective)} \\ \text{subject to} \quad & D_1 z_1 = c_1 && \text{(initial equality)} \\ & C_{i-1} z_{i-1} + D_i z_i = c_i && \text{(inter-stage equality)} \\ & \underline{z}_i \leq z_i \leq \overline{z}_i && \text{(bounds)} \end{aligned}$$

In this example three parameters have to be given to the solver.

- `parameter(1)`: Represents the right hand side of the initial equality of the problem in standard form above.
- `parameter(2)`: The linear term f of the cost function. This term contains the reference values of the states which are calculated based on the resistance of the load.
- `parameter(3)`: Represents the right hand side of the inter-stage equality constraint for the stages $i = 2 : N$ of the problem.

Next to the parameters, the dimensions of the variables, the equality constraints and the bounds have to be defined. The values defined in the MPC setup are added to the multistage problem in the section 'cost'. The terms in the equality constraints which are constant over all stages are defined in the section 'equality constraints'. After defining the output of the solver and the solver settings, the code for the controller can be generated.

```
% Multistage Problem
stages = MultistageProblem(N); % get stages struct of length N

% RHS of first eq. constr. is a parameter: stages(1).eq.c = -A*x0
params(1) = newParam('minusAx0_minusB2w',1,'eq.c');

% Linear Term depends on x_ref and u_ref
params(2) = newParam('Linear_Term',1:N,'cost.f');

% Read in a measurement of the disturbance
params(3) = newParam('minusB2w',2:N,'eq.c');

for i = 1:N

        % dimension
        stages(i).dims.n = nx+nu; % number of stage variables
        stages(i).dims.r = nx;    % number of equality constraints
        stages(i).dims.l = 2;     % number of lower bounds
        stages(i).dims.u = 2;     % number of upper bounds

        % cost
        stages(i).cost.H = blkdiag(R,Q);

        % lower bounds
        stages(i).ineq.b.lbidx = 1:2; % lower bound acts on these indices
        stages(i).ineq.b.lb = [umin; xmin]; % lower bound on input d and state iL

        % upper bounds
        stages(i).ineq.b.ubidx = 1:2; % upper bound acts on these indices
        stages(i).ineq.b.ub = [umax; xmax]; % upper bound on input d and state iL

        % equality constraints
        if( i < N )
                stages(i).eq.C = [zeros(nx,nu), Ad];
```

(continues on next page)

(continued from previous page)

```
        end
        stages(i).eq.D = [Bd1, -eye(nx)];
end

% define outputs of the solver
outputs(1) = newOutput('u0',1,1);

% solver settings
codeoptions = getOptions('DCDC_FORCES_Pro_Controller');

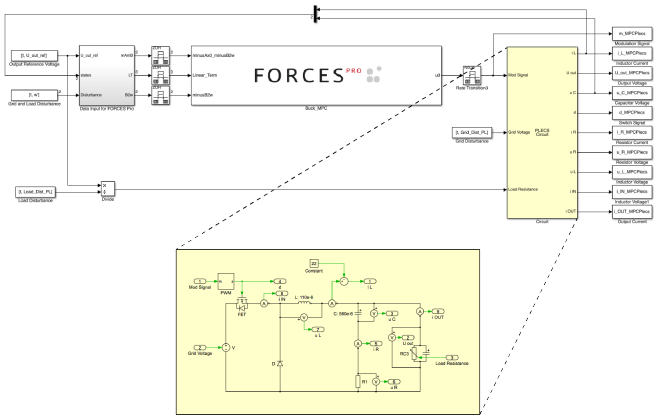
% generate code
generateCode(stages,params,codeoptions,outputs);
```

You can find the MATLAB code of this example to try it out for yourself in the `examples` folder that comes with your client.

11.7.6 Simulation of the PLECS® Model with Model Predictive Control

After the code is generated, the FORCESPRO Simulink® block can be added to the model `DCDC_FORCES_Pro_viewer.slx` as shown in the figure below (copy/paste it from the file `DCDC_FORCES_Pro_Controllercompact_lib.mdl` in the folder `DCDC_FORCES_Pro_Controller/Interface` generated by FORCESPRO).

The controller has a frequency of 100 kHz. To simulate the system with a time step of $1e-7s$, rate transition blocks are used. Below the Simulink® model `DC_DC_FORCES_Pro.slx` with the PLECS® circuit and the FORCESPRO controller is depicted.



In the grey box in the model depicted above, the three parameters which are the input to the FORCESPRO controller, are calculated.

- `parameter(1)`: The right hand side of the initial equality constraint is $-Ad \cdot x - Bd2 \cdot w$.
- `parameter(2)`: For the linear term of the cost function the reference values for the states and the input signal need to be calculated.

The reference values are calculated by solving the linear system

$$\begin{pmatrix} Ad - I & Bd1 \\ Cd2 & Dd3 \end{pmatrix} \cdot \begin{pmatrix} x_{ref} \\ u_{ref} \end{pmatrix} = \begin{pmatrix} -Bd2 \cdot w \\ U_{out,ref} - Dd4 \cdot w \end{pmatrix}$$

which follows from the system equations in steady-state. To calculate the linear term f the reference values are plugged into the linear term $f = \begin{pmatrix} -u_{ref} \cdot R & -x_{ref}^T \cdot Q \end{pmatrix}^T$, which is equal to

$$f = \begin{pmatrix} Ad - I & Bd1 \\ Cd2 & Dd3 \end{pmatrix}^{-1} \cdot \begin{pmatrix} 0 & -Bd2 \\ 1 & -Dd4 \end{pmatrix} \cdot \begin{pmatrix} U_{out,ref} \\ w \end{pmatrix} \cdot \begin{pmatrix} 0 & -R \\ -Q & 0 \end{pmatrix}$$

The matrices in the derivation above are explained in more detail in the system presented in the code available for this example.

- parameter(3) is equal to $-Bd2 \cdot w$.

11.7.7 Comparison of Model Predictive Control and PI Control

In the Figure 11.30 and Figure 11.31 below the evolution of the inductor current and the output voltage are compared when controlling the system with PI and with the MPC controller designed using FORCESPRO. It can be seen that the MPC controller is able to keep the inductor current within the limits defined above. However, this limits the tracking speed of the output voltage in the corresponding time interval. Overall, the tracking performance of the output voltage is increased compared to the baseline PI controller.

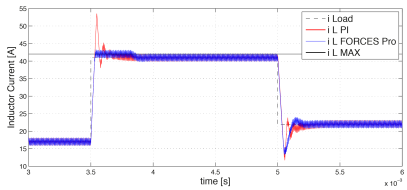


Figure 11.30: Inductor current vs. time

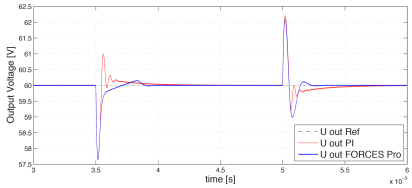


Figure 11.31: Output voltage vs. time

11.8 High-level interface: Basic example

- *Defining the problem data*
- *Defining the MPC problem*
- *Generating a solver*
- *Calling the generated solver*
- *Simulation*
- *Results*

Consider the following linear MPC problem with lower and upper bounds on state and inputs, and a terminal cost term:

$$\begin{aligned} \text{minimize} \quad & x_N^\top P x_N + \sum_{i=0}^{N-1} x_i^\top Q x_i + u_i^\top R u_i \\ \text{subject to} \quad & x_0 = x \\ & x_{i+1} = A x_i + B u_i \\ & \underline{x} \leq x_i \leq \bar{x} \\ & \underline{u} \leq u_i \leq \bar{u} \end{aligned}$$

This problem is parametric in the initial state x and the first input u_0 is typically applied to the system after a solution has been obtained.

You can find the MATLAB code of this example to try it out for yourself in the `examples` folder that comes with your client.

11.8.1 Defining the problem data

Let's define the known data of the MPC problem, i.e. the system matrices A and B, the prediction horizon N, the stage cost matrices Q and R, the terminal cost matrix P, and the state and input bounds:

```
%% system
A = [1.1 1; 0 1];
B = [1; 0.5];
[nx,nu] = size(B);

%% MPC setup
Q = eye(nx);
R = eye(nu);
if( exist('dlqr','file') )
    [~,P] = dlqr(A,B,Q,R);
else
    P = 10*Q;
end
umin = -0.5;     umax = 0.5;
xmin = [-5, -5]; xmax = [5, 5];
```

11.8.2 Defining the MPC problem

Let's now dive in right into the problem formulation:

```
%% FORCESPRO multistage form
% assume variable ordering z(i) = [ui; xi] for i=1...N

% dimensions
model.N     = 11;   % horizon length
model.nvar  = 3;    % number of variables
model.neq   = 2;    % number of equality constraints

% objective
model.objective = @(z) z(1)*R*z(1) + [z(2); z(3)]'*Q*[z(2); z(3)];
model.objectiveN = @(z) [z(2); z(3)]'*P*[z(2); z(3)];

% equalities
model.eq = @(z) [ A(1,:)*[z(2); z(3)] + B(1)*z(1);
                  A(2,:)*[z(2); z(3)] + B(2)*z(1)];

model.E = [zeros(2,1), eye(2)];

% initial state
model.xinitidx = 2:3;

% inequalities
model.lb = [ umin,    xmin  ];
model.ub = [ umax,    xmax  ];
```

11.8.3 Generating a solver

We have now populated `model` with the necessary fields to generate a solver for our problem. Now we use the function `FORCES_NLP` to generate a solver for the problem defined by `model` with the first state as a parameter:

```
%% Generate FORCESPRO solver

% get options
codeoptions = getOptions('FORCESNLPsolver');
codeoptions.printlevel = 2;
codeoptions.nlp.ad_tool = 'casadi';
%codeoptions.nlp.ad_tool = 'symbolic-math-tbx';

% generate code
FORCES_NLP(model, codeoptions);
```

11.8.4 Calling the generated solver

Once all parameters have been populated, the MEX interface of the solver can be used to invoke it:

```
problem.x0 = zeros(model.N*model.nvar,1);
problem.xinit = xinit;
[solverout,exitflag,info] = FORCESNLPsolver(problem);
```

Tip: Type `help solvername` to get more information about how to call the solver.

11.8.5 Simulation

Let's now simulate the closed loop over the prediction horizon N:

```
%% simulate
x1 = [-4; 2];
kmax = 30;
X = zeros(2,kmax+1); X(:,1) = x1;
U = zeros(1,kmax);
problem.x0 = zeros(model.N*model.nvar,1);
for k = 1:kmax

    problem.xinit = X(:,k);

    [solverout,exitflag,info] = FORCESNLPsolver(problem);

    if( exitflag == 1 )
        U(:,k) = solverout.x01(1);
    else
        error('Some problem in solver');
    end

    %X(:,k+1) = A*X(:,k) + B*U(:,k);
    X(:,k+1) = model.eq( [U(:,k); X(:,k)] )';
end
```

11.8.6 Results

The results of the simulation are presented in Figure 11.8. The plot on the top shows the system's states over time, while the plot on the bottom shows the input commands. We can see that all constraints are respected.

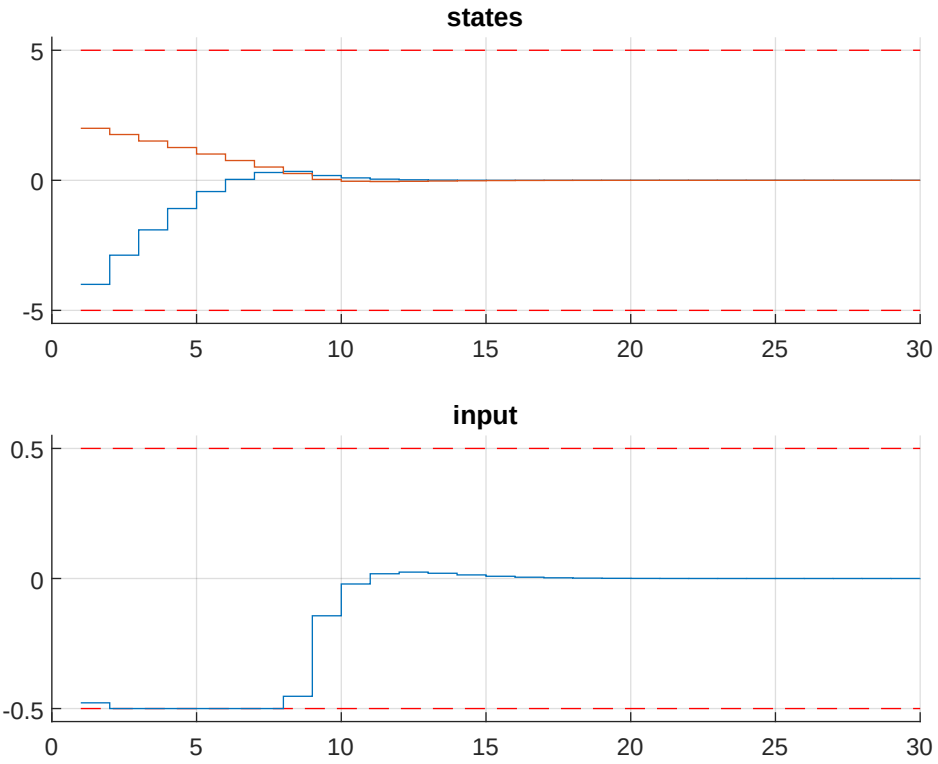


Figure 11.32: Simulation results of the states (top, in blue and red) and input (bottom, in blue) over time. The state and input constraints are plotted in red dashed lines.

11.9 High-level interface: Obstacle avoidance (MATLAB & Python)

- *Defining the MPC Problem*
  - *Objective*
  - *Matrix equality constraints*
  - *Runtime Parameters*
  - *Inequality constraints*
  - *Dimensions*
  - *Initial conditions*
- *Generating a solver*
- *Calling the generated solver*
- *Results*
- *Variation: External functions*

In this example we illustrate the simplicity of the high-level user interface on a vehicle optimal trajectory generation problem. The user can place an obstacle in front of the vehicle using an interactive window and the car trajectory is automatically adjusted.

In particular, we use a kinematic bicycle model described by a set of ordinary differential equations (ODEs):

$$\begin{aligned}\dot{x} &= v\cos(\theta+\beta)\\ \dot{y} &= v\sin(\theta+\beta)\\ \dot{v} &= \frac{F}{m}\\ \dot{\theta} &= \frac{v}{l_r}\sin(\beta)\\ \dot{\delta} &= \phi\end{aligned}$$

with:

$$\beta = \arctan\left(\frac{l_r}{l_r+l_f}\tan(\delta)\right)$$

The model consists of five differential states: x and y are the Cartesian coordinates of the car, and v is the linear velocity. The angles θ and δ denote the heading angle of the car and its steering angle. Next, there are two control inputs to the model: the acceleration force F and the steering rate ϕ. The angle β describes the direction of movement of the car's center of gravity relative to the heading angle θ. The remaining three constant parameters of the system are the car mass $m = 1\,\mathrm{kg}$, and the lengths $l_r = 0.5\,\mathrm{m}$ and $l_f = 0.5\,\mathrm{m}$ specifying the distance from the car's center of gravity to the rear wheels and the front wheels, respectively.

The trajectory of the vehicle will be defined as an NLP. First, we define stage variable z by stacking the input and differential state variables:

$$z = [F, \phi, x, y, v, \theta, \delta]^\top$$

You can find the code of this example to try it out for yourself in the `examples` folder that comes with your client.

11.9.1 Defining the MPC Problem

Objective

In this example the cost function provided by `model.objective` is the same for all stages. We have a target position $(3, 0)$ and we want to minimize the distance of the car to that point. Therefore, the distance is penalized with linear costs. Plus, some small quadratic costs are added to the inputs F and s, i.e.:

$$f(z) = 100|z_3 - 0| + 100|z_4 - 3| + 0.1z_1^2 + 0.01z_2^2$$

The stage cost function is coded in MATLAB and Python as the following function:

Matlab

Python

```
model.objective = @objective

function f = objective(z)
    F = z(1);
    phi = z(2);
    x = z(3)
    y = z(4);
    f = 100*abs(x-0) + 100*abs(y-3) + 0.1*F^2 + 0.01*phi^2;
end
```

```
model = forcespro.nlp.SymbolicModel()  # create empty model
model.objective = lambda z: 100 * casadi.fabs(z[2] - 0.) \
                          + 100 * casadi.fabs(z[3] - 3.) \
                          + 0.1 * z[0]**2 + 0.01 * z[1]**2
```

Matrix equality constraints

The matrix equality constraints `model.eq` in this example result from the vehicle's dynamics given above. First, the continuous dynamic equations are implemented as follows:

Matlab

Python

```
function [ xDot ] = continuousDynamics(x,u)
% state x = [xPos,yPos,v,theta,delta], input u = [F, phi]

    % set physical constants
    l_r = 0.5; % distance rear wheels to center of gravity of the car
    l_f = 0.5; % distance front wheels to center of gravity of the car
    m = 1.0;   % mass of the car

    % set parameters
    beta = atan(l_r/(l_f + l_r) * tan(x(5)));

    % calculate dx/dt
    xDot = [ x(3) * cos(x(4) + beta); % dxPos/dt = v*cos(theta+beta)
             x(3) * sin(x(4) + beta); % dyPos/dt = v*cos(theta+beta)
             u(1)/m;                  % dv/dt = F/m
             x(3)/l_r * sin(beta);    % dtheta/dt = v/l_r*sin(beta)
             u(2)];                   % ddelta/dt = phi
end
```

```python
def continuous_dynamics(x, u):
  """ state x = [xPos,yPos,v,theta,delta], input u = [F,phi]"""

  # set physical constants
  l_r = 0.5 # distance rear wheels to center of gravity of the car
  l_f = 0.5 # distance front wheels to center of gravity of the car
  m = 1.0   # mass of the car

  # set parameters
  beta = casadi.arctan(l_r/(l_f + l_r) * casadi.tan(x[4]))

  # calculate dx/dt
  return np.array([x[2] * casadi.cos(x[3] + beta), # dxPos/dt = v*cos(theta+beta)
                   x[2] * casadi.sin(x[3] + beta),  # dyPos/dt = v*sin(theta+beta)
                   u[0] / m,                        # dv/dt = F/m
                   x[2]/l_r * casadi.sin(beta),     # dtheta/dt = v/l_r*sin(beta)
                   u[1]])                           # ddelta/dt = phi
```

Now, these continuous dynamics are discretized using an explicit Runge-Kutta integrator of order 4 as shown below. Note that the function `RK4` is included in the FORCESPRO client software.

Matlab

Python

```
integrator_stepsize = 0.1;
model.eq = @(z) RK4( z(3:7), z(1:2), @continuousDynamics,...
    integrator_stepsize);
```

```python
integrator_stepsize = 0.1
# z[2:7] = states x, z[0:2] = inputs u
model.eq = lambda z: forcespro.nlp.integrate(continuous_dynamics, z[2:7], z[0:2],
                                             integrator=forcespro.nlp.integrators.RK4,
                                             stepsize=integrator_stepsize)
```

As a last step, the indices of the left hand side of the dynamical constraint are defined. For efficiency reasons, make sure the matrix has structure [0 I].

Matlab

Python

```
model.E = [zeros(5,2), eye(5)];
```

```python
model.E = np.concatenate([np.zeros((5,2)), np.eye(5)], axis=1)
```

Runtime Parameters

The user can place an obstacle to be avoided in front of the car. The x- and y-coordinates of the position p of this obstacle are considered as runtime parameters of the system.

$$p = [p_x, p_y]^\top$$

The runtime parameters are the same for all stages. Their values will be set later on at runtime.

Inequality constraints

The maneuver is subjected to a set of constraints, involving both the simple bounds:

$$\begin{aligned}
-5\,\mathrm{N} \leq & F \leq 5\,\mathrm{N} \\
-40\,\mathrm{deg/s} \leq & \phi \leq 40\,\mathrm{deg/s} \\
-3\,\mathrm{m} \leq & x \leq 0\,\mathrm{m} \\
0\,\mathrm{m} \leq & y \leq 3\,\mathrm{m} \\
0\,\mathrm{m/s} \leq & v \leq 2\,\mathrm{m/s} \\
-\,\mathrm{inf} \leq & \theta \leq \mathrm{inf} \\
-0.48\pi\,\mathrm{rad} \leq & \delta \leq 0.48\pi\,\mathrm{rad}
\end{aligned}$$

as well the nonlinear nonconvex constraints in dependence of the runtime parameters p

$$\begin{aligned}
1\,\mathrm{m}^2 \leq & x^2 + y^2 \leq 9\,\mathrm{m}^2 \\
0.7^2\,\mathrm{m}^2 \leq & (x - p_x)^2 + (y - p_y)^2
\end{aligned}$$

The implementation of the simple bounds is given here:

Matlab

Python

```
% upper/lower variable bounds lb <= z <= ub
%            inputs             |             states
%             F      phi            x     y     v    theta    delta
model.lb = [ -5.0,  deg2rad(-40),  -3.,   0.,   0.,  -inf,   -0.48*pi];
model.ub = [ +5.0,  deg2rad(+40),   0.,   3.,   2.,  +inf,    0.48*pi];
```

```
#  upper/lower variable bounds lb <= z <= ub
#                     inputs                 |  states
#                     F          phi           x    y     v    theta          delta
model.lb = np.array([-5.,  np.deg2rad(-40.),  -3.,   0.,   0,   -np.inf, -0.48*np.pi])
model.ub = np.array([+5.,  np.deg2rad(+40.),   0.,   3.,   2.,   np.inf,  0.48*np.pi])
```

The nonlinear constraint function h with its bounds can be coded in MATLAB/Python as follows:

Matlab

Python

```
% General (differentiable) nonlinear inequalities hl <= h(x,p) <= hu
model.ineq = @(z,p)  [   z(3)^2 + z(4)^2;                  % x^2 + y^2
                       (z(3)-p(1))^2 + (z(4)-p(2))^2 ];  % (x-p_x)^2 + (y-p_y)^2

 % Upper/lower bounds for inequalities
model.hu = [9, +inf]';
model.hl = [1, 0.7^2]';
```

```
# General (differentiable) nonlinear inequalities hl <= h(x,p) <= hu
model.ineq = lambda z,p:  np.array([z[2]**2 + z[3]**2,                 # x^2 + y^2
                                    (z[2] - p[0])**2 + (z[3] - p[1])**2]) # (x-p_x)^2 +␣
↪(y-p_y)^2

# Upper/lower bounds for inequalities
```

(continues on next page)

(continued from previous page)

```
model.hu = np.array([9, +np.inf])
model.hl = np.array([1, 0.7**2])
```

Dimensions

Furthermore, the number of variables, constraints and real-time parameters explained above needs to be provided as well as the length of the multistage problem. For this example, we chose to use $N = 50$ stages in the NLP:

Matlab

Python

```
model.N = 50;     % horizon length
model.nvar = 7;   % number of variables
model.neq  = 5;   % number of equality constraints
model.nh = 2;     % number of inequality constraint functions
model.npar = 2;   % number of runtime parameters
model.bfgs_init = 3.0*eye(7);  % set initialization of the hessian approximation
```

```
model.N = 50     # horizon length
model.nvar = 7   # number of variables
model.neq = 5    # number of equality constraints
model.nh = 2     # number of inequality constraint functions
model.npar = 2   # number of runtime parameters
model.bfgs_init = 3.0*np.identity(7) # initialization of the hessian approximation
```

Initial conditions

The goal of the maneuver is to steer the vehicle from a set of initial conditions:

$$x_{\text{init}} = -2\,\text{m}, \quad y_{\text{init}} = 0\,\text{m}, \quad v_{\text{init}} = 0\,\text{m/s}, \quad \theta_{\text{init}} = 0.5\pi\,\text{rad}, , \quad \delta_{\text{init}} = 0\,\text{rad}$$

For the code generation, only the indices of the variables to which initial values will be applied are required. This is coded as follows:

Matlab

Python

```
model.xinitidx = 3:7;
```

```
model.xinitidx = range(2,7)
```

11.9.2 Generating a solver

We have now populated `model` with the necessary fields to generate a solver for our problem. Now we set some options for our solver and then use the function `FORCES_NLP` to generate a solver for the problem defined by `model`:

Matlab

Python

```
%% Define solver options
codeoptions = getOptions('FORCESNLPsolver');
codeoptions.maxit = 400;    % Maximum number of iterations
codeoptions.printlevel = 0; % Use printlevel = 2 to print progress (but not for␣
↪timings)
codeoptions.optlevel = 3;   % 0: no optimization, 1: optimize for size, 2: optimize␣
↪for speed, 3: optimize for size & speed
codeoptions.cleanup = false;
codeoptions.timing = 1;
codeoptions.nlp.ad_tool = 'casadi';

%% Generate FORCESPRO solver
FORCES_NLP(model, codeoptions);
```

```
# Set solver options
codeoptions = forcespro.CodeOptions('FORCESNLPsolver')
codeoptions.maxit = 400                         # Maximum number of iterations
codeoptions.printlevel = 0
codeoptions.optlevel = 0                        # 0 no optimization, 1 optimize for␣
↪size, 2 optimize for speed, 3 optimize for size & speed
codeoptions.noVariableElimination = 1.

# Creates code for symbolic model formulation given above, then contacts server to␣
↪generate new solver
solver = model.generate_solver(codeoptions)
```

11.9.3 Calling the generated solver

Once all parameters of the problem instance to be solved have been populated, the MEX interface of the solver can be used to invoke it.

Matlab

Python

```
% Set runtime parameters
params = [-1.5; 1.0];
problem.all_parameters = repmat(params,model.N,1);

% Set initial guess to start solver from (here, middle of upper and lower
% bound)
x0i=[0.0,0.0,-1.5,1.5,1.,pi/4.,0.];
x0=repmat(x0i',model.N,1);
problem.x0=x0;

% Set initial conditions. This is usually changing from problem instance
% to problem instance:
model.xinit = [-2., 0., 0., deg2rad(90), 0.]';
problem.xinit = model.xinit; % xPos=-2, yPos=0, v=0 (standstill),
% heading angle=90deg, steering angle=0deg

% Time to solve the NLP!
[output,exitflag,info] = FORCESNLPsolver(problem);

% Make sure the solver has exited properly.
if (exitflag < 0)
```

(continues on next page)

(continued from previous page)

```
    fprintf([newline 'Some error in FORCESPRO solver. exitflag=%d'...
        newline],exitflag)
elseif (exitflag == 0)
    fprintf([newline 'Error in FORCESPRO solver: Maximum number'...
        ' of iterations was reached. ' newline]);
else
    fprintf([newline 'FORCESPRO took %d iterations and %f seconds '...
        'to solve the problem.' newline], info.it, info.solvetime);
end
```

```
# Set initial guess to start solver from (here, middle of upper and lower bound)
x0i = np.array([0.,0.,-1.5,1.5,1.,np.pi/4.,0.])
x0 = np.transpose(np.tile(x0i, (1, model.N)))

# set initial condition
xinit = np.transpose(np.array([-2.,0.,0.,np.deg2rad(90),0.]))

problem = {"x0": x0,
           "xinit": xinit,
           "xfinal": xfinal}

 # Set runtime parameters
params = np.array([-1.5,1.]) # In this example, the user can change these parameters␣
↪by clicking into an interactive window
problem["all_parameters"] = np.transpose(np.tile(params,(1,model.N)))

# Time to solve the NLP!
output, exitflag, info = solver.solve(problem)

# Make sure the solver has exited properly.
assert exitflag == 1, "bad exitflag"
print("FORCES took {} iterations and {} seconds to solve the problem.".format(info.it,
↪ info.solvetime))
```

11.9.4 Results

The goal is to find a trajectory that steers the vehicle from point A as close as possible to point B while avoiding obstacles. The trajectory should also be feasible with respect to the vehicle dynamics and its safety and physical limitations. The calculated vehicle's trajectory in 2D space is presented in Figure 11.33. The progress of the other states and the inputs over time is shown in Figure 11.34. One can see that all constraints are respected. To try out other obstacle positions you can run the example file on your own machine and click into the interactive window.

You can find the code of this example in the `examples` folder that comes with your client.

11.9.5 Variation: External functions

In this variation, we want to supply the required functions through external functions in C. To do so we have to provide the directory that contains said source files in the MATLAB code:

```
%% Define source file containing function evaluation code
model.extfuncs = 'C/myfevals.c';
```

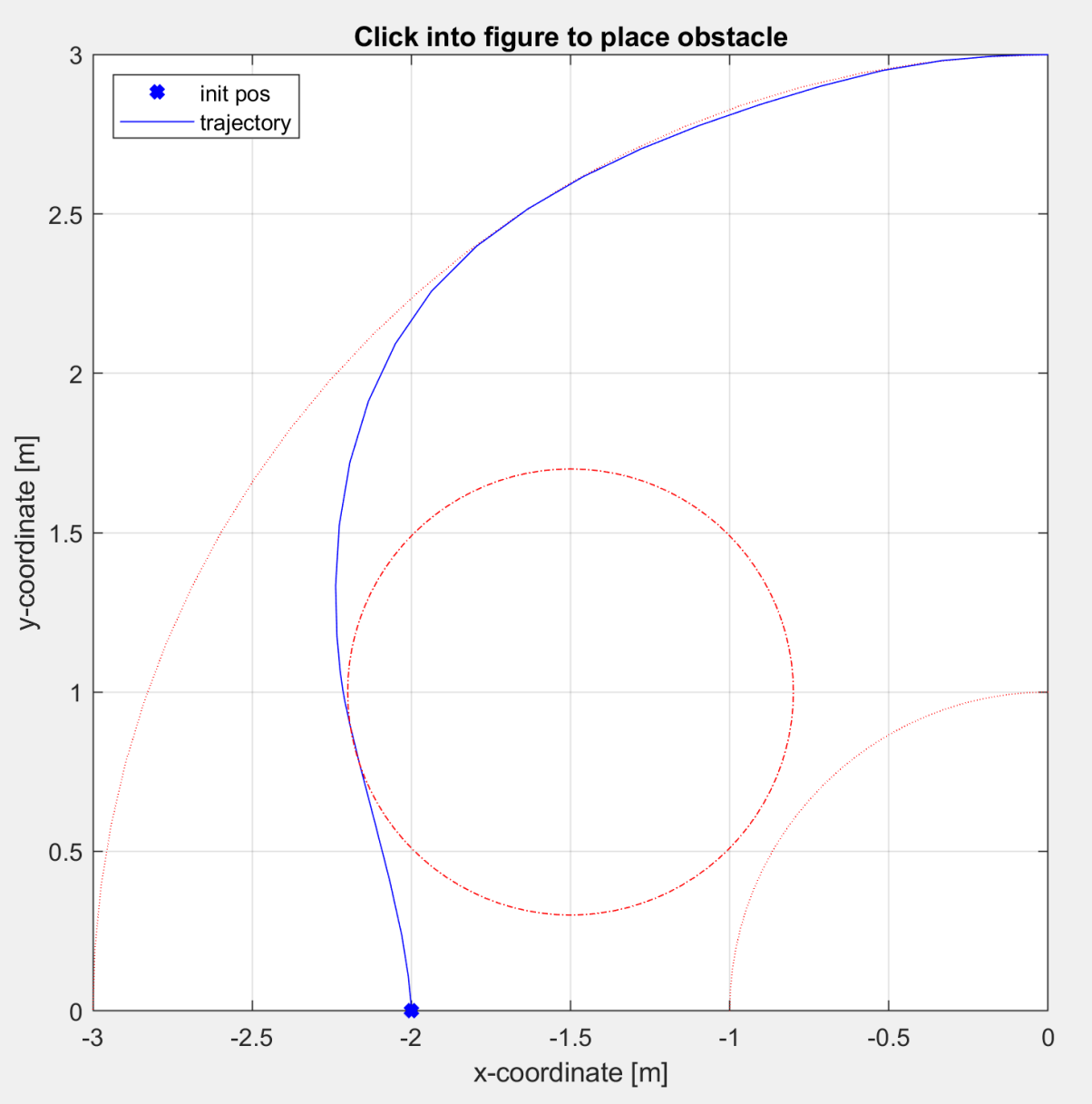


Figure 11.33: The calculated trajectory of the car

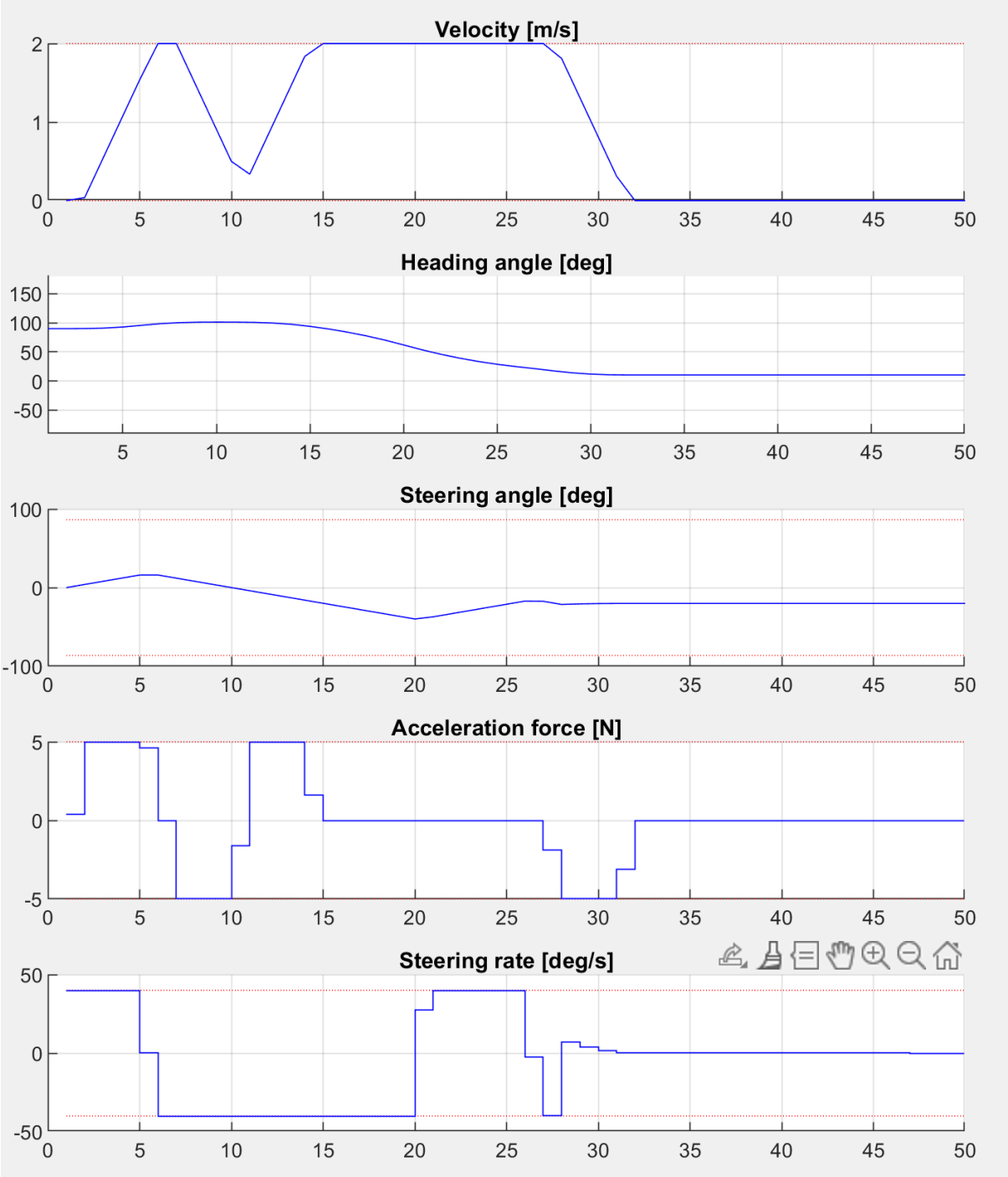


Figure 11.34: Development of the vehicle's states and the system's inputs over time

We also need to include the two **extern** functions **car_dynamics** and **car_dynamics_jacobian**, both contained in the **car_dynamics.c** file, through the **other_srcs** options field:

```
% add additional source files required - separate by spaces if more than 1
codeoptions.nlp.other_srcs = 'C/car_dynamics.c';
```

In Python, we need to switch to an **ExternalFunctionModel** if we intend to use external callbacks. We give the main callback evaluating the objective function, equality constraints and inequality constraints, using the **set_main_function()** , and supply any additional files required by this callback using **add_auxiliary()**.

```
model = forcespro.nlp.ExternalFunctionModel()

# Define source file containing function evaluation code
model.set_main_callback("C/myfevals.c", function="myfevals")
model.add_auxiliary("C/car_dynamics.c")
# One can also add a 'relative_to' argument specifying the paths to be understood
# relative to this file's location. if not supplied, paths are relative to the
# current working directory in which this script is executed:
# model.set_main_callback('c/myfevals.c', function="myfevals" relative_to=os.path.
↪dirname(__file__))
# model.add_auxiliary('c/car_dynamics.c', relative_to=os.path.dirname(__file__))
```

You can find the code of this example to try it out for yourself in the **examples** folder that comes with your client.

11.10 High-level interface: Indoor localization (MATLAB & Python)

- *Time of flight measurements*
- *Estimation error*
- *Minimize the error*
- *Implementation*

The indoor localization problem is to estimate the position of a target by measurements from various anchors with known location. Outdoors, this well known as GPS, while indoors other frequency bands (and less accurate clocks) are usually used. In this example, we show how to generate code for a position estimator that relies on time-of-flight (TOF) measurements (GPS uses time-difference-of-arrival, TDOA). The latter can be easily implemented with FORCESPRO as well with only minor changes to the code below.

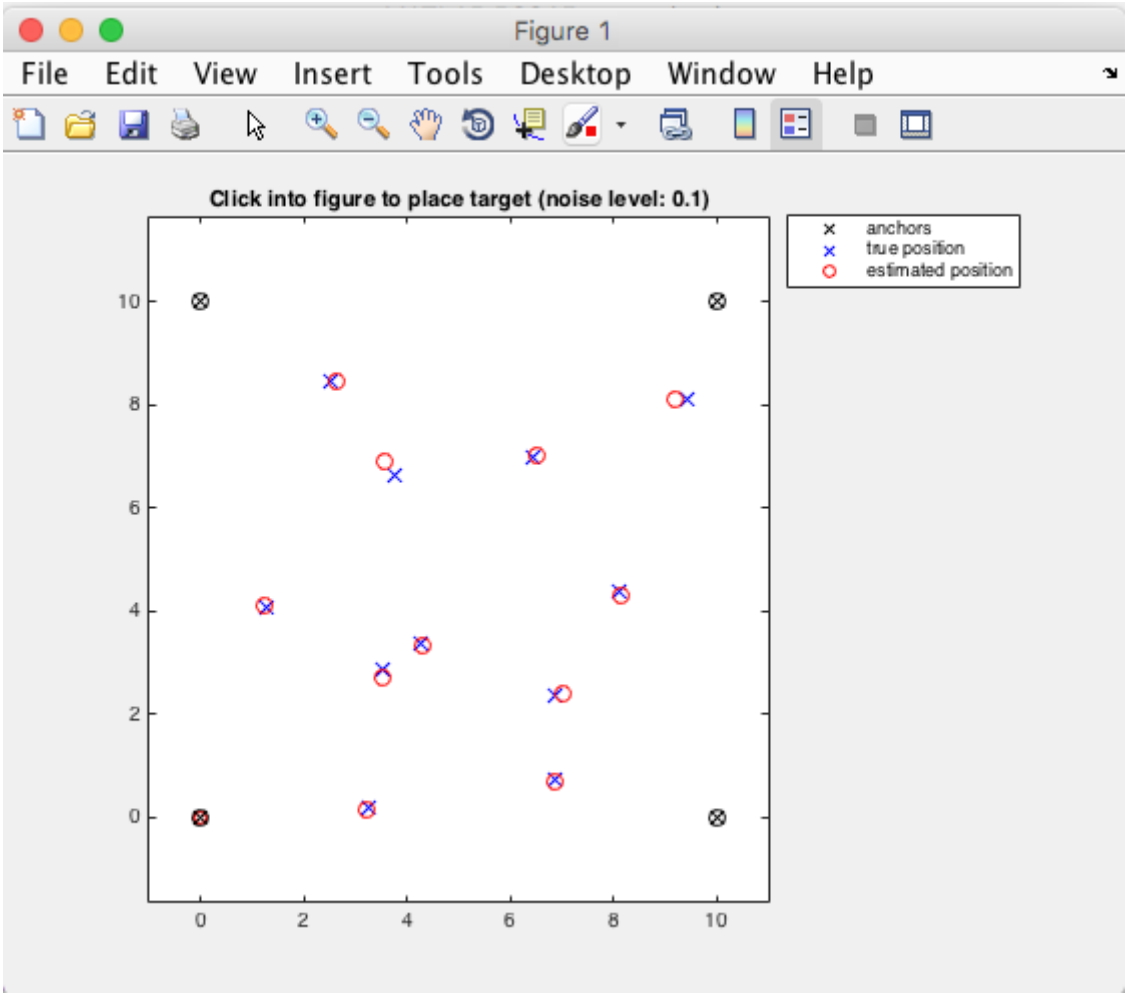


Figure 11.35: Indoor localization example GUI.

You can find the code of this example to try it out for yourself in the `examples` folder that comes with your client.

Running the code will produce an interactive window like in Figure 11.35.

11.10.1 Time of flight measurements

Given N anchors with known positions (x_i^a, y_i^a), $i = 1, \ldots, N$, the distance to the target with unknown position (x, y) is given by:

$$d_i = ct_i = \sqrt{(x - x_i^a)^2 + (y - y_i^a)^2}$$

where t_i is the time the signal from anchor i travels at the speed $c = 299\,792\,458\,\mathrm{m/s}$

11.10.2 Estimation error

Instead of the real distance, we work with squared distances to define the estimation error:

$$e_i = (x - x_i^a)^2 + (y - y_i^a)^2 - d_i^2$$

11.10.3 Minimize the error

The objective is a least-squares error function:

$$\min_{x,y} \sum_{i=1}^{N} e_i^2$$

11.10.4 Implementation

The following MATLAB/Python code generates C-code for implementing an optimizer for minimizing the least-squares error function from above. It takes the anchor positions and the distance measurements, and returns the estimated position of the target.

Matlab

Python

```
%% This function generates the estimator
function [] = generateEstimator(numberOfAnchors,xlimits,ylimits)
% Generates 2D decoding code for localization using FORCESPRO NLP

    na = numberOfAnchors;

    %% NLP problem definition
    % no need to change anything below
    model.N = 1;      % number of distance measurements
    model.nvar = 2;   % number of variables (use 3 if 3D)
    model.npar = numberOfAnchors*3; % number of parameters: coordinates of anchors in␣
↪2D, plus measurements
    model.objective = @(z,p) objective(z,p,na);
    model.lb = [xlimits(1) ylimits(1)]; % lower bounds on (x,y)
    model.ub = [xlimits(2) ylimits(2)]; % upper bounds on (x,y)

    %% codesettings
    codesettings = getOptions('localizationDecoder');
    codesettings.printlevel = 0; % set to 2 to see some prints
    % codesettings.server = 'http://winner10:2470';
    codesettings.maxit = 50; % maximum number of iterations
    codesettings.nlp.ad_tool = 'casadi';
```

(continues on next page)

(continued from previous page)

```
    %codesettings.nlp.ad_tool = 'symbolic-math-tbx';

    %% generate code
    FORCES_NLP(model, codesettings);

end

%% This function implements the objective
% We assume that the parameter vector p is ordered as follows:
% p(1:na)        - x-coordinates of the anchors
% p(na+(1:na))   - y-coordinates of the anchors
% p(2*na+(1:na)) - distance measurements of the anchors
function [ obj ] = objective(z, p, na)

    obj=0;
    for i = 1:na
        obj = obj + ( (p(i)-z(1))^2 + (p(i+na)-z(2))^2 - p(i+2*na)^2 )^2;
    end
end
```

```
def generate_estimator(number_of_anchors, xlimits, ylimits):
    """
    Generates and returns a FORCESPRO solver that estimates a position based on
    noisy measurement inputs.
    """

    # NLP problem definition
    # ----------------------

    model = forcespro.nlp.SymbolicModel(1)  # number of distance measurements
    model.nvar = 2  # number of variables (use 3 if 3D)
    model.npar = number_of_anchors * 3  # number of parameters: coordinates of␣
↪anchors in 2D, plus measurements
    model.objective = objective  # objective is defined as it's own function below
    model.lb = np.array([xlimits[0], ylimits[0]])  # lower bounds on (x,y)
    model.ub = np.array([xlimits[1], ylimits[1]])  # upper bounds on (x,y)

    # FORCESPRO solver settings
    # -------------------------

    codesettings = forcespro.CodeOptions()
    codesettings.printlevel = 0  # set to 2 to see some prints
    codesettings.maxit = 50  # maximum number of iterations

    # Generate a solver
    # -----------------
    solver = model.generate_solver(codesettings)

    return solver


def objective(z, p):
    """
    This function implements the objective to be minimized.

    We assume that the parameter vector p is ordered as follows:
```

(continues on next page)

(continued from previous page)

```
     - p[0:(na-1)]        - x-coordinates of the anchors
     - p[na:(2*na-1)]     - y-coordinates of the anchors
     - p[(2*na):(3*na-1)] - distance measurements of the anchors
    """
    obj = 0
    for i in range(n):
        obj += ((p[i] - z[0])**2 + (p[i + n] - z[1])**2 - p[i + 2*n]**2)**2
    return obj


def distance(xa, xtrue, ya, ytrue):
    return np.sqrt((xa - xtrue)**2 + (ya - ytrue)**2)
```

11.11 High-level interface: Path tracking example (MATLAB)

- *Defining the MPC Problem*
  - *Objective*
  - *Equality constraints*
  - *Bounds*
  - *Dimensions*
- *Generating a solver*
- *Calling the generated solver*
  - *Choosing the Path*
- *Results*
- *Using the SQP Fast solver*

In this example we illustrate the simplicity of the SQP_NLP API on a path-tracking problem. We also illustrate the full workflow for the SQP Fast solver which must be tuned to the specific application. See *Using the SQP Fast solver* for specifics on the workflow for the SQP Fast solver. In every simulation step, the predicted trajectory of the car is optimized to follow a set of path points, either an ellipse or an artificial racetrack. The example visualizes how the predicted trajectory changes while the car moves along the path.

We use a kinematic bicycle model described by a set of ordinary differential equations (ODEs):

$$\begin{aligned}\dot{x} &= v\cos(\theta+\beta)\\ \dot{y} &= v\sin(\theta+\beta)\\ \dot{v} &= \frac{F}{m}\\ \dot{\theta} &= \frac{v}{l_r}\sin(\beta)\\ \dot{\delta} &= \phi\end{aligned}$$

with:

$$\beta = \arctan\left(\frac{l_r}{l_r+l_f}\tan(\delta)\right)$$

The model consists of five differential states: x and y are the Cartesian coordinates of the car, and v is the linear velocity. The angles θ and δ denote the heading angle of the car and its steering angle, respectively. Next, there are two control inputs to the model: the longitudinal acceleration force F and the steering rate ϕ. The angle β describes the direction of movement of the car's center of gravity relative to the heading angle θ. The remaining three constant parameters of the system are the car mass $m = 1\,\mathrm{kg}$, and the lengths $l_r = 0.5\,\mathrm{m}$ and $l_f = 0.5\,\mathrm{m}$ specifying the distance from the car's center of gravity to the rear wheels and the front wheels, respectively.

The trajectory of the vehicle will be defined as an NLP. First, we define stage variable z by stacking the input and differential state variables:

$$z = [F, \phi, x, y, v, \theta, \delta]^\top$$

You can find the code of this example (both for the SQP solver using the general and fast QP solver) in the `examples` folder that comes with your client.

11.11.1 Defining the MPC Problem

Objective

In this example the cost function is the same for all stages except for the last stage N. The objective of this example is to follow a set of path points. At runtime, a target position p_i for each stage i is provided. Each point consists of a x- and a y-coordinate:

$$p_i = [p_{i,x}, p_{i,y}]^\top$$

The goal is to minimize the distance of the car to these target points. The distance is penalized with quadratic costs. Plus, some small quadratic costs are added to the inputs F and s, i.e.:

$$f(z, p_i) = 100(z_3 - p_{i,x})^2 + 100(z_4 - p_{i,y})^2 + 0.1z_1^2 + 5z_2^2$$

Since all cost terms are quadratic and summed up, we can formulate the objective as a least squares problem:

$$f(z, p_i) = \frac{1}{2}||r(z, p_i)||_2^2$$
$$r(z, p_i) = [\sqrt{200}(z_3 - p_{i,x}), \sqrt{200}(z_4 - p_{i,y}), \sqrt{0.2}z_1, \sqrt{10}z_2]^\top$$

The stage cost function is coded in MATLAB as the following function:

Matlab

```
%% Indices - struct I keeps track of variable sorting in z
% Inputs
I.FLon = 1; I.steeringRate = 2;
I.inputs = [I.FLon, I.steeringRate];
% States
I.xPos = 3; I.yPos = 4; I.velocity = 5; I.heading = 6; I.steeringAngle = 7;
I.states = [I.xPos, I.yPos, I.velocity, I.heading, I.steeringAngle];

model.LSobjective = @(z,p)LSobj(z,p,I);

function [r] = LSobj(z,currentTarget, I)
% Least square costs on deviating from the path and on the inputs
% currentTarget = point on path that is tracked in this stage

    r = [sqrt(200.0)*(z(I.xPos)-currentTarget(1));  % costs for deviating from the␣
↪path in x-direction
         sqrt(200.0)*(z(I.yPos)-currentTarget(2));   % costs for deviating from the␣
↪path in y-direction
         sqrt(0.2)*z(I.FLon);                        % penalty on input FLon
         sqrt(10.0)*z(I.steeringRate)];              % penalty on input steeringRate
end
```

Note that using the `model.LSobjective` option instead of `model.objective` allows you to try out the `gauss-newton` method for the hessian approximation.

For the last stage, the terminal costs are slightly increased by adapting the weighting factors:

$$f(z, p_i) = 200(z_3 - p_{i,x})^2 + 200(z_4 - p_{i,y)}^2 + 0.1z_1^2 + 5z_2^2$$

The code looks a follows:

Matlab

```
model.LSobjectiveN = @(z,p)LSobjN(z,p,I);

function [r] = LSobjN(z,currentTarget, I)
% Increased least square costs for last stage on deviating from the path and on the␣
↪inputs
% currentTarget = point on path that is tracked in this stage

    r = [sqrt(400.0)*(z(I.xPos)-currentTarget(1));  % costs for deviating from the␣
↪path in x-direction
         sqrt(400.0)*(z(I.yPos)-currentTarget(2));   % costs for deviating from the␣
↪path in y-direction
         sqrt(0.2)*z(I.FLon);                        % penalty on input FLon
         sqrt(10.0)*z(I.steeringRate)];              % penalty on input steeringRate
end
```

Equality constraints

The equality constraints `model.eq` in this example result from the vehicle's dynamics given above. First, the continuous dynamic equations are implemented as follows:

Matlab

```
function [xDot] = continuousDynamics(x,u,I)

    % Number of inputs is needed for indexing
    nu = numel(I.inputs);

    % Set physical constants
    l_r = 0.5; % Distance rear wheels to center of gravity of the car
    l_f = 0.5; % Distance front wheels to center of gravity of the car
    m = 1.0;   % Mass of the car

    % Set parameters
    beta = atan(l_r/(l_r + l_f) * tan(x(I.steeringAngle-nu)));

    % Calculate dx/dt
    xDot = [x(I.velocity-nu) * cos(x(I.heading-nu) + beta);    % dxPos/dt =␣
↪v*cos(theta+beta)
            x(I.velocity-nu) * sin(x(I.heading-nu) + beta);    % dyPos/dt =␣
↪v*cos(theta+beta)
            u(I.FLon)/m;                                       % dv/dt = F/m
            x(I.velocity-nu)/l_r * sin(beta);                  % dtheta/dt = v/l_
↪r*sin(beta)
            u(I.steeringRate)];                                % ddelta/dt = phi

end
```

Now, these continuous dynamics are discretized using an explicit Runge-Kutta integrator of order 4 as shown below. Note that the function `RK4` is included in the FORCESPRO client software.

Matlab

```
timeStep = 0.1;
model.eq = @(z) RK4( z(I.states), z(I.inputs), @(x,u)continuousDynamics(x,u,I),...
    timeStep);
```

As a last step, the indices of the left hand side of the dynamical constraint are defined. For efficiency reasons, make sure the matrix has structure [0 I].

Matlab

```
model.E = [zeros(nStates,nInputs), eye(nStates)];
```

Bounds

All variables except the heading angle θ are bounded:

$$\begin{aligned} -5\,\mathrm{N} \leq & F \leq 5\,\mathrm{N} \\ -90\,\mathrm{deg/s} \leq & \phi \leq 90\,\mathrm{deg/s} \\ -100\,\mathrm{m} \leq & x \leq 100\,\mathrm{m} \\ -100\,\mathrm{m} \leq & y \leq 100\,\mathrm{m} \\ 0\,\mathrm{m/s} \leq & v \leq 5\,\mathrm{m/s} \\ -\inf \leq & \theta \leq \inf \\ -50\,\mathrm{deg} \leq & \delta \leq 50\,\mathrm{deg} \end{aligned}$$

The implementation of the simple bounds is given here:

Matlab

```
% Upper/lower variable bounds lb <= z <= ub
%           inputs                |  states
%           FLon   steeringRate     xPos    yPos  velocity  heading   steeringAngle
model.lb = [ -5.,  deg2rad(-90),   -100.,  -100.,   0.,     -inf,     deg2rad(-50)];
model.ub = [ +5.,  deg2rad(90),     100.,   100.,   5.,     +inf,     deg2rad(50)];
```

Dimensions

Furthermore, the number of variables, constraints and real-time parameters explained above needs to be provided as well as the length of the multistage problem. For this example, we chose to use $N = 10$ stages in the NLP:

Matlab

```
horizonLength = 10;
nInputs = numel(I.inputs);
nStates = numel(I.states);

model.N = horizonLength;            % Horizon length
model.nvar = nInputs+nStates;       % Number of variables
model.neq  = nStates;               % Number of equality constraints
model.npar = 2;                     % Number of runtime parameters (waypoint␣
↪coordinates)
```

11.11.2 Generating a solver

The actual solver generation happens inside the function `generatePathTrackingSolver`:

Matlab

```
%% Define solver options
codeoptions = getOptions('PathTrackingSolver');
codeoptions.maxit = 200;        % Maximum number of iterations
codeoptions.optlevel = 3;       % 0: No optimization, good for prototyping
codeoptions.timing = 1;
codeoptions.printlevel = 0;
codeoptions.nohash = 1;         % Enforce solver regeneration
codeoptions.overwrite = 1;      % Overwrite existing solver
codeoptions.BuildSimulinkBlock = 0;

% SQP options
codeoptions.solvemethod = 'SQP_NLP';
codeoptions.nlp.hessian_approximation = 'gauss-newton';
codeoptions.sqp_nlp.maxqps = 1;     % Maximum number of quadratic problems to be␣
↪solved in one solver call
codeoptions.sqp_nlp.use_line_search = 0;

%% Generate FORCESPRO solver
cd(solverDir);
FORCES_NLP(model, codeoptions);
```

First, appropriate codeoptions are set. In particular, we chose to use an SQP algorithm with Gauss-Newton approximation for the Hessian matrix and perform only a single QP solve per solver call. Finally, we use the function `FORCES_NLP` to generate a solver called `PathTrackingSolver`.

11.11.3 Calling the generated solver

The goal of this example is to optimize the predicted car trajectory for the next *N* time steps and then apply the calculated input for the current time step. The procedure is repeated for the entire simulation period.

Choosing the Path

The path to follow can either be an ellipse or a racetrack profile.

Matlab

```
mapName = 'racetrack';      % Current options: 'ellipse' and 'racetrack'
```

The initial conditions for the dynamic states are chose as the first waypoint of the path and the real-time parameters are initialized with the waypoint coordinates:

Matlab

```
% Set initial condition
problem.xinit = simulatedZ(I.states,k);

% Set runtime parameters (here, the next N points on the map)
nextPathPoints = resamplePathForTracker(simulatedZ(:,k), I, map, timeStep,␣
↪horizonLength);
problem.all_parameters = reshape(nextPathPoints,2*model.N,1);
```

Note: The racetrack map used for this example originates from the repository `https://github.com/alexliniger/MPCC`. See `PathTracking.m` and `PathTrackingSqpFast.m` inside the ex-

ample folder/archive for more details.

The simulation itself is performed using the following loop, which makes use of several auxiliary functions:

Matlab

```
tuningProblems = cell(simLength,1); % only used in PathTrackingSqpFast.m

% Initialize plot handles
handles = [];

for k = 1:simLength

    % Set initial condition
    problem.xinit = simulatedZ(I.states,k);

    % Set runtime parameters (here, the next N points on the map)
    nextPathPoints = resamplePathForTracker(simulatedZ(:,k), I, map, timeStep,␣
↪horizonLength);
    problem.all_parameters = reshape(nextPathPoints,2*model.N,1);

    % Solve optimization problem
    [output,exitflag,info] = PathTrackingSolver(problem);

    % Make sure the solver has exited properly
    if (exitflag == 1)
        fprintf('FORCESPRO took %d iterations and ',info.it);
        fprintf('%.3f milliseconds to solve the problem.\n\n',info.solvetime*1000);
    elseif (exitflag == -8)
        warning('FORCESPRO finished with exitflag=-8. The underlying QP might be␣
↪infeasible.');
        fprintf('FORCESPRO took %d iterations and ',info.it);
        fprintf('%.3f milliseconds to solve the problem.\n\n',info.solvetime*1000);
    else
        error('Some problem in solver!');
    end

    % Apply optimized input u to system and save simulation data
    simulatedZ(I.inputs,k) = output.x01(I.inputs);
    simulatedZ(I.states,k+1) = model.eq( simulatedZ(:,k) );

    % Extract output for prediction plots
    cwidth = floor(log10(model.N))+1;
    for i = 1:model.N
        predictedZ(:,i) = output.(sprintf(sprintf('x%%0%iu',cwidth), i));
    end
    % Plot
    handles = plotPathTrackerData(k,simulatedZ,predictedZ,model,I,simLength,map,
↪nextPathPoints,handles);

    if k == 1
        % From now on, the solver should be initialized with the solution of its last␣
↪call
        problem.reinitialize = 0;
    end
```

(continues on next page)

(continued from previous page)

```
    % Pause to slow down plotting
    pause(0.02);

end
```

11.11.4 Results

The goal is to find a trajectory that steers the vehicle as close to the provided racetrack waypoints as possible. The trajectory should also be feasible with respect to the vehicle dynamics and its safety and physical limitations. The 2D calculated vehicle's trajectory at timestep $k = 100$ is presented in blue in Figure 11.36. Here, you can see the current predictions for the trajectory marked green. The progress of the other states and inputs over time as well as their predictions is shown in Figure 11.37.

The trajectory and the progress of the system variables over the entire simulation period of 360 steps are presented in Figure 11.38 and Figure 11.39. One can see that all constraints are respected.

To see how the predictions of the system variables develop over all timesteps you can run the example file on your own machine.

11.11.5 Using the SQP Fast solver

The above sections illustrate how to specify a dynamical model and generate a FORCESPRO solver to use in a reference-tracking MPC algorithm. Since FORCESPRO version 6.0.0, FORCESPRO supports a `SQP Fast` solver (see *Tuning the SQP Fast solver*). This algorithmic choice changes the workflow a little as it requires the user to tune the solver in order to obtain a solver which is tailored to the specific application. This tuning procedure can be done using the FORCESPRO "autotune" tool (see *Autotuner*).

A variant of this example, showing the complete workflow for the fast QP solver, is available under the name "PathTrackingSqpFast" in the `examples` folder in your FORCESPRO client.

The key step in tuning a fast SQP solver is to collect problem data on which certain algorithmic parameters are tuned. For this, one generates a solver (with `codeoptions.sqp_nlp.qp_method = "general"`) exactly as above and then performs the simulation in order to save the simulation data (the `tuningProblems` in the code-snippet in *Calling the generated solver*). Once this data has been stored a fast SQP solver can be generated and tuned as follows:

Matlab

```
% Generate fast SQP solver based on a fast QP solver
tuningoptions = ForcesAutotuneOptions(tuningProblems);
ForcesGenerateSqpFastSolver(model, codeoptions, tuningoptions);
```

Note that the FORCESPRO autotuning tool allows for a great deal of customization. Such customization is specified via the `ForcesAutotuneOptions` object (see *Autotuner Options*).

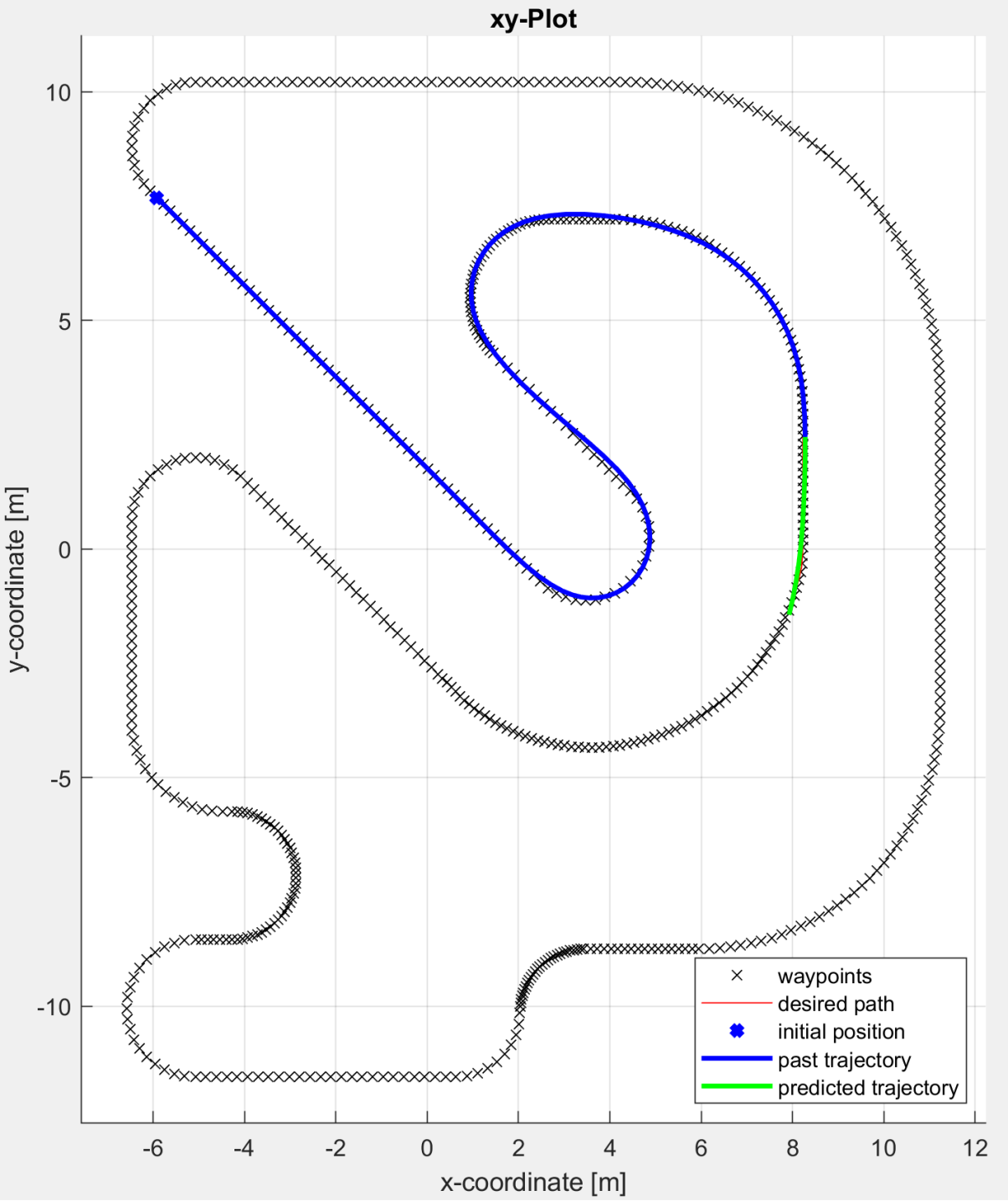


Figure 11.36: The calculated trajectory of the car (blue) and its predictions (green) at timestep $k = 100$ (racetrack map from repository https://github.com/alexliniger/MPCC)

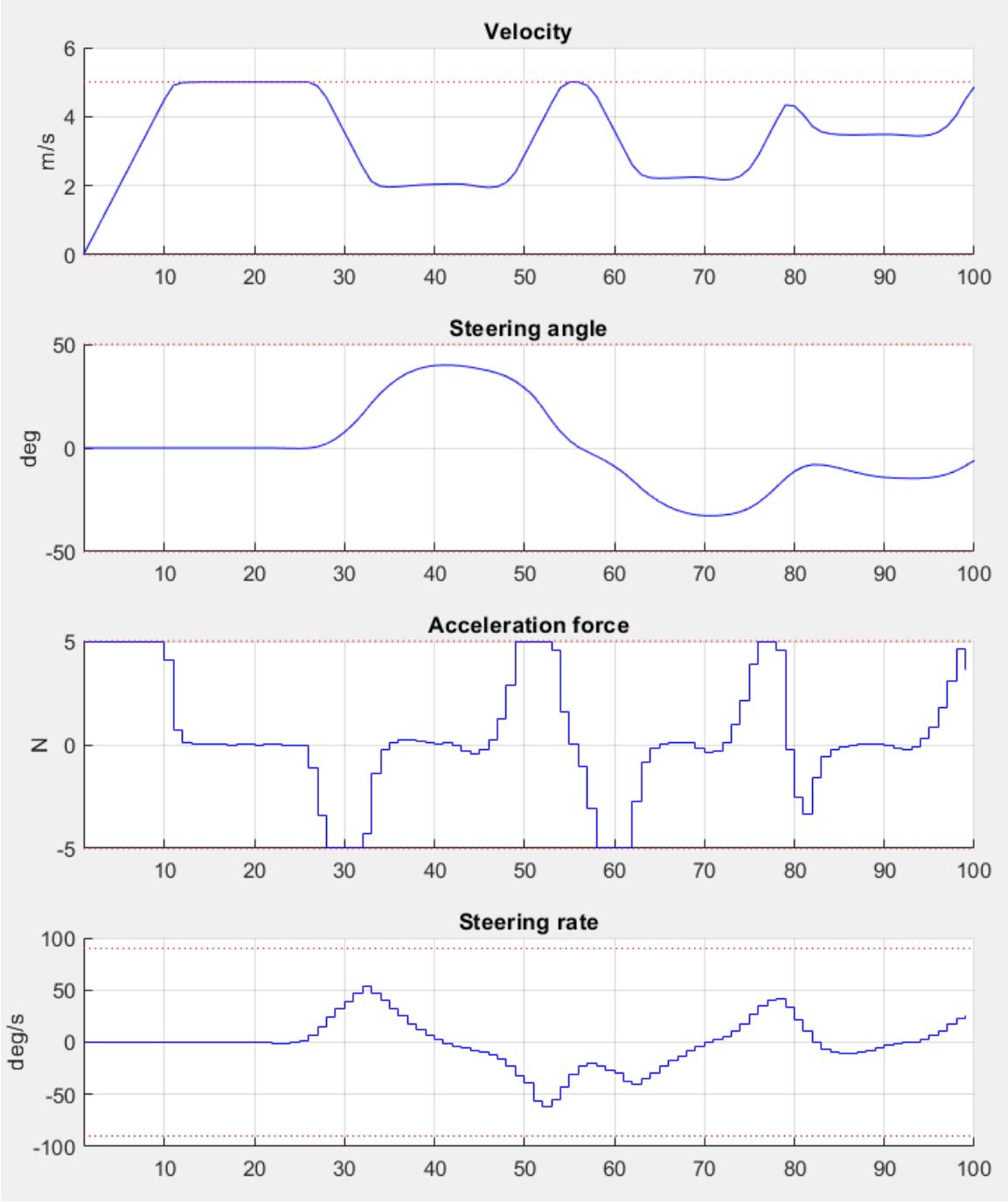


Figure 11.37: Development of the vehicle's states and the system's inputs over time at timestep $k = 100$

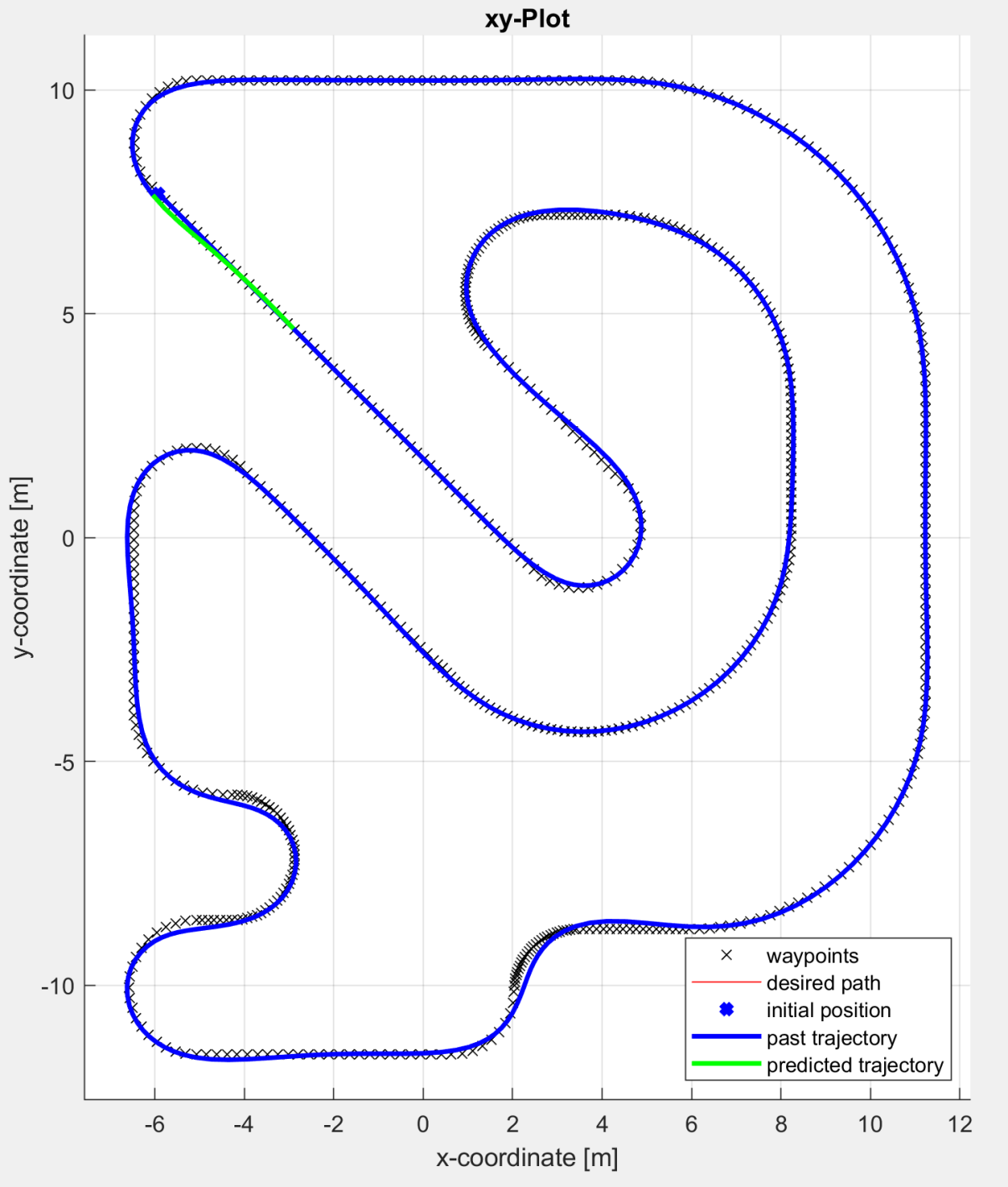


Figure 11.38: The calculated trajectory of the car (racetrack map from repository https://github.com/alexliniger/MPCC)

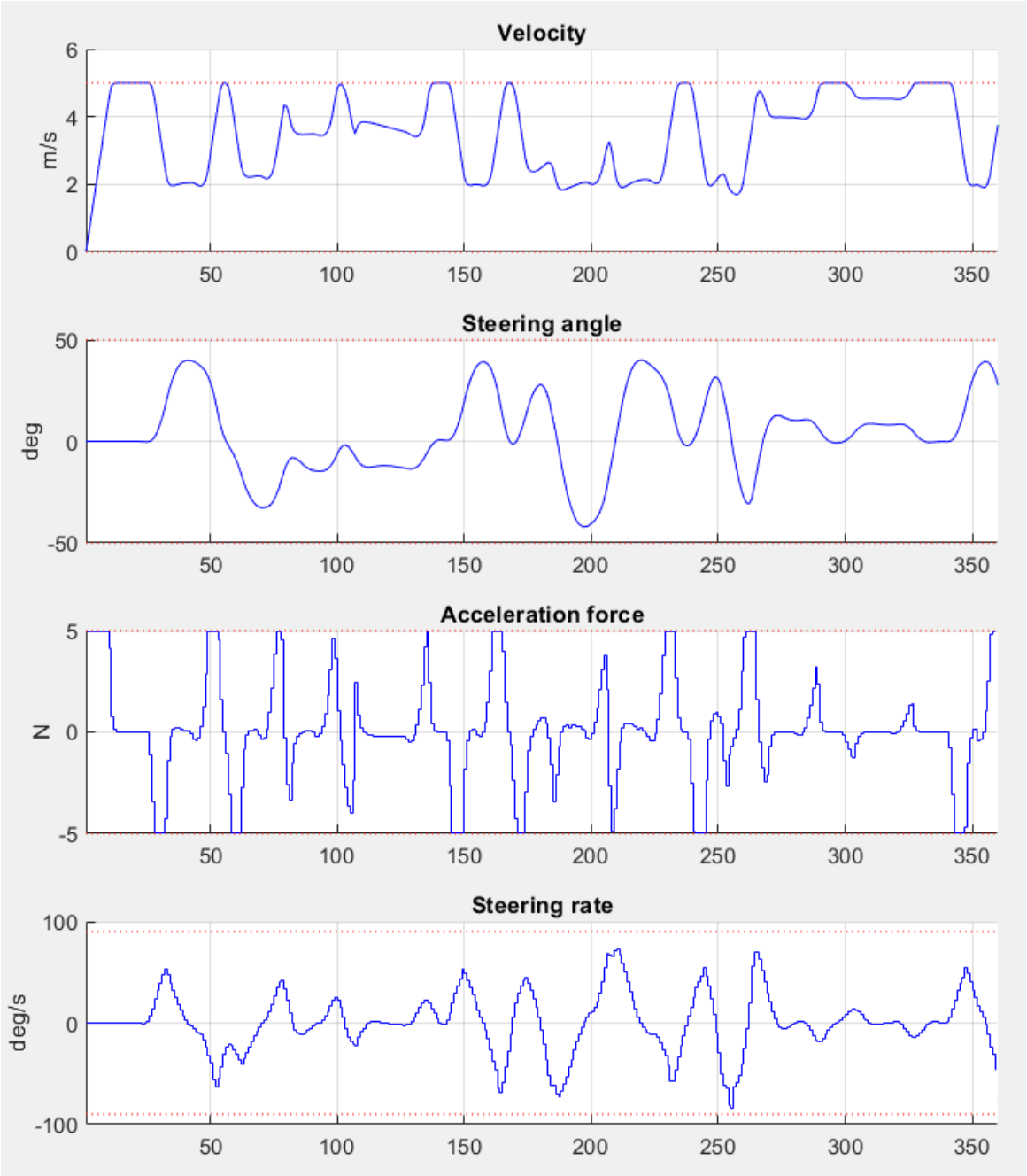


Figure 11.39: Development of the vehicle's states and the system's inputs over time

11.12 High-level interface: Legacy path tracking example (MATLAB & Python)

- *Defining the MPC Problem*
  - *Objective*
  - *Matrix equality constraints*
  - *Bounds*
  - *Dimensions*
  - *Initial conditions*
- *Generating a solver*
- *Calling the generated solver*
- *Results*

In this example we illustrate the simplicity of the SQP_NLP API on a path-tracking problem. In every simulation step, the predicted trajectory of the car is optimized to follow a set of path points. The example also visualizes how the predicted trajectory changes while the car moves forward.

Note that an improved version of this example is available for MATLAB, see *High-level interface: Path tracking example (MATLAB)*.

We use a kinematic bicycle model described by a set of ordinary differential equations (ODEs):

$$\begin{aligned}\dot{x} &= v\cos(\theta+\beta)\\ \dot{y} &= v\sin(\theta+\beta)\\ \dot{v} &= \frac{F}{m}\\ \dot{\theta} &= \frac{v}{l_r}\sin(\beta)\\ \dot{\delta} &= \phi\end{aligned}$$

with:

$$\beta = \arctan\left(\frac{l_r}{l_r+l_f}\tan(\delta)\right)$$

The model consists of five differential states: x and y are the Cartesian coordinates of the car, and v is the linear velocity. The angles θ and δ denote the heading angle of the car and its steering angle. Next, there are two control inputs to the model: the acceleration force F and the steering rate ϕ. The angle β describes the direction of movement of the car's center of gravity relative to the heading angle θ. The remaining three constant parameters of the system are the car mass $m = 1\,\mathrm{kg}$, and the lengths $l_r = 0.5\,\mathrm{m}$ and $l_f = 0.5\,\mathrm{m}$ specifying the distance from the car's center of gravity to the rear wheels and the front wheels, respectively.

The trajectory of the vehicle will be defined as an NLP. First, we define stage variable z by stacking the input and differential state variables:

$$z = [F, \phi, x, y, v, \theta, \delta]^\top$$

You can find the code of this example to try it out for yourself in the `examples` folder that comes with your client.

11.12.1 Defining the MPC Problem

Objective

In this example the cost function is the same for all stages except for the last stage N. The objective of this example is to follow a set of path points. At runtime, a target position p_i for each stage i is provided. Each point consists of a x- and a y-coordinate:

$$p_i = [p_{i,x}, p_{i,y}]^\top$$

The goal is to minimize the distance of the car to these target points. The distance is penalized with quadratic costs. Plus, some small quadratic costs are added to the inputs F and s, i.e.:

$$f(z, p_i) = 100(z_3 - p_{i,x})^2 + 100(z_4 - p_{i,y})^2 + 0.1z_1^2 + 0.1z_2^2$$

Since all cost terms are quadratic and summed up, we can formulate the objective as a least squares problem:

$$f(z, p_i) = \frac{1}{2}||r(z, p_i)||_2^2$$
$$r(z, p_i) = [\sqrt{200}(z_3 - p_{i,x}), \sqrt{200}(z_4 - p_{i,y}), \sqrt{0.2}z_1, \sqrt{0.2}z_2]^\top$$

The stage cost function is coded in MATLAB and Python as the following function:

Matlab

Python

```
model.LSobjective = @LSobj;

function [ r ] = LSobj(z,currentTarget)
% least square costs on deviating from the path and on the inputs F and phi
% z = [F,phi,xPos,yPos,v,theta,delta]
% currentTarget = point on path that is to be headed for

    r = [sqrt(200.0)*(z(3)-currentTarget(1)); % costs for deviating from
        % the path in x-direction
        sqrt(200.0)*(z(4)-currentTarget(2));  % costs for deviating from
        % the path in y-direction
        sqrt(0.2)*z(1);                        % penalty on input F
        sqrt(0.2)*z(2)];                       % penalty on input phi
end
```

```
model = forcespro.nlp.SymbolicModel()  # create empty model
model.objective = obj

def obj(z,current_target):
    """Least square costs on deviating from the path and on the inputs F and phi
    z = [F,phi,xPos,yPos,v,theta,delta]
    current_target = point on path that is to be headed for
    """
    return (100.0*(z[2]-current_target[0])**2   # costs on deviating on the path in x-
↪direction
            + 100.0*(z[3]-current_target[1])**2 # costs on deviating on the path in y-
↪direction
            + 0.1*z[0]**2                       # penalty on input F
            + 0.1*z[1]**2)                      # penalty on input phi
```

Note that using the `model.LSobjective` option instead of `model.objective` allows you to try out the `gauss-newton` method for the hessian approximation.

For the last stage, the terminal costs are slightly increased by adapting the weighting factors:

$$f(z, p_i) = 200(z_3 - p_{i,x})^2 + 200(z_4 - p_{i,y}^2) + 0.2z_1^2 + 0.2z_2^2$$

The code looks a follows:

Matlab

Python

```
model.LSobjectiveN = @LSobjN;

function [ r ] = LSobjN(z,currentTarget)
% Increased least square costs for last stage on deviating from the path
% and on the inputs F and phi
% z = [F,phi,xPos,yPos,v,theta,delta]
% currentTarget = point on path that is to be headed for

    r = [sqrt(400.0)*(z(3)-currentTarget(1)); % costs for deviating from
        % the path in x-direction
        sqrt(400.0)*(z(4)-currentTarget(2));  % costs for deviating from
        % the path in y-direction
        sqrt(0.4)*z(1);                        % penalty on input F
        sqrt(0.4)*z(2)];                       % penalty on input phi
end
```

```
model.objectiveN = objN

def objN(z,current_target):
    """Increased least square costs for last stage on deviating from the path and
    on the inputs F and phi
    z = [F,phi,xPos,yPos,v,theta,delta]
    current_target = point on path that is to be headed for
    """
    return (200.0*(z[2]-current_target[0])**2   # costs on deviating on the path in x-
↪direction
        + 200.0*(z[3]-current_target[1])**2     # costs on deviating on the path in y-
↪direction
        + 0.2*z[0]**2                           # penalty on input F
        + 0.2*z[1]**2)                          # penalty on input phi
```

Matrix equality constraints

The matrix equality constraints `model.eq` in this example result from the vehicle's dynamics given above. First, the continuous dynamic equations are implemented as follows:

Matlab

Python

```
function [ xDot ] = continuousDynamics(x,u)
% state x = [xPos,yPos,v,theta,delta], input u = [F, phi]

% set physical constants
    l_r = 0.5; % distance rear wheels to center of gravity of the car
    l_f = 0.5; % distance front wheels to center of gravity of the car
    m = 1.0;   % mass of the car

    % set parameters
    beta = atan(l_r/(l_f + l_r) * tan(x(5)));

    % calculate dx/dt
```

(continues on next page)

(continued from previous page)

```
    xDot = [x(3) * cos(x(4) + beta); % dxPos/dt = v*cos(theta+beta)
            x(3) * sin(x(4) + beta); % dyPos/dt = v*cos(theta+beta)
            u(1)/m;                  % dv/dt = F/m
            x(3)/l_r * sin(beta);    % dtheta/dt = v/l_r*sin(beta)
            u(2)];                   % ddelta/dt = phi

end
```

```
def continuous_dynamics(x, u):
    """Defines dynamics of the car, i.e. equality constraints.
    parameters:
    state x = [xPos,yPos,v,theta,delta]
    input u = [F,phi]
    """
    # set physical constants
    l_r = 0.5 # distance rear wheels to center of gravity of the car
    l_f = 0.5 # distance front wheels to center of gravity of the car
    m = 1.0   # mass of the car

    # set parameters
    beta = casadi.arctan(l_r/(l_f + l_r) * casadi.tan(x[4]))

    # calculate dx/dt
    return casadi.vertcat(  x[2] * casadi.cos(x[3] + beta),  # dxPos/dt =␣
↪v*cos(theta+beta)
                            x[2] * casadi.sin(x[3] + beta),  # dyPos/dt =␣
↪v*sin(theta+beta)
                            u[0] / m,                        # dv/dt = F/m
                            x[2]/l_r * casadi.sin(beta),     # dtheta/dt = v/l_
↪r*sin(beta)
                            u[1])                            # ddelta/dt = phi
```

Now, these continuous dynamics are discretized using an explicit Runge-Kutta integrator of order 4 as shown below. Note that the function `RK4` is included in the FORCESPRO client software.

Matlab

Python

```
integrator_stepsize = 0.1;
% z(3:7) = states x, z(1:2) = inputs u
model.eq = @(z) RK4(z(3:7), z(1:2), @continuousDynamics, integrator_stepsize);
```

```
integrator_stepsize = 0.1
# z[2:7] = states x, z[0:2] = inputs u
model.eq = lambda z: forcespro.nlp.integrate(continuous_dynamics, z[2:7], z[0:2],
                                             integrator=forcespro.nlp.integrators.RK4,
                                             stepsize=integrator_stepsize)
```

As a last step, the indices of the left hand side of the dynamical constraint are defined. For efficiency reasons, make sure the matrix has structure [0 I].

Matlab

Python

```
model.E = [zeros(5,2), eye(5)];
```

```
model.E = np.concatenate([np.zeros((5,2)), np.eye(5)], axis=1)
```

Bounds

All variables except the heading angle θ are bounded:

$$
\begin{aligned}
-5\,\mathrm{N} \leq & F \leq 5\,\mathrm{N} \\
-90\,\mathrm{deg/s} \leq & \phi \leq 90\,\mathrm{deg/s} \\
-2\,\mathrm{m} \leq & x \leq 2\,\mathrm{m} \\
-2\,\mathrm{m} \leq & y \leq 2\,\mathrm{m} \\
0\,\mathrm{m/s} \leq & v \leq 4\,\mathrm{m/s} \\
-\inf \leq & \theta \leq \inf \\
-0.48\pi\,\mathrm{rad} \leq & \delta \leq 0.48\pi\,\mathrm{rad}
\end{aligned}
$$

The implementation of the simple bounds is given here:

Matlab

Python

```
% upper/lower variable bounds lb <= z <= ub
%            inputs             |             states
%             F      phi            x     y     v    theta    delta
model.lb = [ -5.,  deg2rad(-90),   -2.,  -2.,   0.,  -inf,   -0.48*pi];
model.ub = [ +5.,  deg2rad(90),     2.,   2.,   4.,  +inf,    0.48*pi];
```

```
#  upper/lower variable bounds lb <= z <= ub
#                     inputs                 |  states
#                     F          phi           x    y     v    theta       delta
model.lb = np.array([-5.,  np.deg2rad(-90.),  -2.,   -2.,  0., -np.inf, -0.48*np.pi])
model.ub = np.array([+5.,  np.deg2rad(+90.),   2.,   2.,   4.,  np.inf,  0.48*np.pi])
```

Dimensions

Furthermore, the number of variables, constraints and real-time parameters explained above needs to be provided as well as the length of the multistage problem. For this example, we chose to use $N = 10$ stages in the NLP:

Matlab

Python

```
model.N = 10;       % horizon length
model.nvar = 7;     % number of variables
model.neq  = 5;     % number of equality constraints
model.npar = 2;     % number of runtime parameters
```

```
model.N = 10    # horizon length
model.nvar = 7  # number of variables
model.neq = 5   # number of equality constraints
model.npar = 2  # number of runtime parameters
model.bfgs_init = 2.5*np.identity(7)  # set initialization of the hessian␣
↪approximation
```

Initial conditions

The goal of the maneuver is to steer the vehicle from a set of initial conditions:

$$x_{\text{init}} = 0.8\,\text{m}, \quad y_{\text{init}} = 0\,\text{m}, \quad v_{\text{init}} = 0\,\text{m/s}, \quad \theta_{\text{init}} = 0.5\pi\,\text{rad},, \quad \delta_{\text{init}} = 0\,\text{rad}$$

For the code generation, only the indices of the variables to which initial values will be applied are required. This is coded as follows:

Matlab

Python

```
% Initial condition on vehicle states
model.xinit = [0.8, 0., 0., deg2rad(90), 0.]';  % xPos=0.8, yPos=0, v=0 (standstill),␣
↪heading angle=90, steering angle=0
model.xinitidx = 3:7; % use this to specify on which variables initial conditions are␣
↪imposed
```

```
model.xinitidx = range(2,7)
```

11.12.2 Generating a solver

We have now populated `model` with the necessary fields to generate a solver for our problem. We choose the SQP solve method and set some further options for our solver. Then, we use the function `FORCES_NLP` to generate a solver for the problem defined by `model`:

Matlab

Python

```
%% Set solver options
codeoptions = getOptions('FORCESNLPsolver');
codeoptions.maxit = 200;                        % Maximum number of iterations
codeoptions.printlevel = 2;                     % Use printlevel = 2 to print␣
↪progress (but(not for timings)
codeoptions.optlevel = 0;                       % 0: no optimization, 1: optimize for␣
↪size, 2: optimize for speed, 3: optimize for size & speed
codeoptions.cleanup = false;
codeoptions.timing = 1;
codeoptions.printlevel = 0;
codeoptions.nlp.hessian_approximation = 'bfgs'; % set initialization of the hessian␣
↪approximation
codeoptions.solvemethod = 'SQP_NLP';            % choose the solver method Sequential␣
↪Quadratic Programming
codeoptions.sqp_nlp.maxqps = 1;                 % maximum number of quadratic␣
↪problems to be solved during one solver call
codeoptions.sqp_nlp.reg_hessian = 5e-9;         % increase this parameter if␣
↪exitflag=-8
```

```
# Set solver options
codeoptions = forcespro.CodeOptions('FORCESNLPsolver')
codeoptions.maxit = 200                         # Maximum number of iterations
codeoptions.printlevel = 0                      # Use printlevel = 2 to print␣
↪progress (but not for timings)
codeoptions.optlevel = 0                        # 0 no optimization, 1 optimize for␣
↪size, 2 optimize for speed, 3 optimize for size & speed
```

(continues on next page)

(continued from previous page)

```
codeoptions.cleanup = False
codeoptions.timing = 1
codeoptions.nlp.hessian_approximation = 'bfgs'
codeoptions.solvemethod = 'SQP_NLP'             # choose the solver method Sequential␣
↪Quadratic Programming
codeoptions.sqp_nlp.maxqps = 1                  # maximum number of quadratic␣
↪problems to be solved
codeoptions.sqp_nlp.reg_hessian = 5e-9          # increase this if exitflag=-8
# change this to your server or leave uncommented for using the
# standard embotech server at https://forces.embotech.com
# codeoptions.server = 'https://forces.embotech.com'

# Creates code for symbolic model formulation given above, then contacts
# server to generate new solver
solver = model.generate_solver(options=codeoptions)
```

11.12.3 Calling the generated solver

The goal of this example is to optimize the predicted car trajectory for the next N time steps and then apply the calculated input for the current time step. The procedure is repeated for the entire simulation period.

This means, after setting up the initial problem instance, the solver is called in a loop for every simulation time step. The MEX interface of the solver is used to invoke it.

Matlab

Python

```
%% Simulate and call solver
simLength = 80; % simulate 8sec

% Variables for storing simulation data
x = zeros(5,simLength+1); % states
x(:,1) = model.xinit;
u = zeros(2,simLength);   % inputs

% Set initial guess to start solver from
x0i = zeros(model.nvar,1);
problem.x0 = repmat(x0i,model.N,1);
problem.reinitialize = 1; % initialize first simulation step of solver
% always with problem.x0

% Create 2D points on ellipse which the car is supposed to follow
numPoints = 80;
pathPoints = calcPointsOnEllipse(numPoints);

% Create empty plot
startPred = reshape(problem.x0,7,model.N); % first prediction corresponds
% to initial guess

% Create plot with initial values
createPlot(x,u,startPred,simLength,model,pathPoints);

for k = 1:simLength
```

(continues on next page)

(continued from previous page)

```
    % Set initial condition
    problem.xinit = x(:,k);

    % Set runtime parameters (here, the next N points on the path)
    % Definition of extractNextPathPoints further below in this file
    nextPathPoints = extractNextPathPoints(pathPoints, x(1:2,k), model.N);
    problem.all_parameters = reshape(nextPathPoints,2*model.N,1);

    % Solve optimization problem
    [output,exitflag,info] = FORCESNLPsolver(problem);

    % Make sure the solver has exited properly
    if( exitflag == 1 )
        fprintf('\nFORCES took %d iterations and ',info.it);
        fprintf('%f seconds to solve the problem.\n',info.solvetime);
    else
        error('Some problem in solver');
    end

    % Apply optimized input u to system and save simulation data
    u(:,k) = output.x01(1:2);
    x(:,k+1) = model.eq( [u(:,k); x(:,k)] )';

    % Extract output for prediction plots
    predictedZ = zeros(7,model.N);
    for i = 1:model.N
        if i < 10 && model.N >= 10
            predictedZ(:,i) = output.(['x0',int2str(i)]);
        else
            predictedZ(:,i) = output.(['x',int2str(i)]);
        end
    end
    predictedU = predictedZ(1:2,:);
    predictedX = predictedZ(3:7,:);

    if k == 1
        % from now on, the solver should be initialized with the solution
        % of its last call
        problem.reinitialize = 0;
    end

    % plot current progress
    updatePlot(x,u,predictedX,predictedU,model,k)

end
```

```
# Simulation
# ----------
sim_length = 80 # simulate 8sec

# Variables for storing simulation data
x = np.zeros((5,sim_length+1)) # states
u = np.zeros((2,sim_length)) # inputs

# Set initial guess to start solver from
x0i = np.zeros((model.nvar,1))
```

(continues on next page)

(continued from previous page)

```
x0 = np.transpose(np.tile(x0i, (1, model.N)))

# Set initial condition
xinit = np.transpose(np.array([0.8, 0., 0., np.deg2rad(90), 0.]))
x[:,0] = xinit

problem = {"x0": x0,
        "xinit": xinit}

for k in range(sim_length):

    # Set initial condition
    problem["xinit"] = x[:,k]

    # Set runtime parameters (here, the next N points on the path)
    next_path_points = extract_next_path_points(path_points, x[0:2,k], model.N)
    problem["all_parameters"] = np.reshape(np.transpose(next_path_points), \
        (2*model.N,1))

    # Time to solve the NLP!
    output, exitflag, info = solver.solve(problem)

    # Make sure the solver has exited properly.
    assert exitflag == 1, "bad exitflag"
    sys.stderr.write("FORCES took {} iterations and {} seconds to solve the problem.\n
↪"\
        .format(info.it, info.solvetime))

    # Extract output
    temp = np.zeros((np.max(model.nvar), model.N))
    for i in range(0, model.N):
        temp[:, i] = output['x{0:02d}'.format(i+1)]
    pred_u = temp[0:2, :] # predicted inputs
    pred_x = temp[2:7, :] # predicted states

    # Apply optimized input u of first stage to system and save simulation data
    u[:,k] = pred_u[:,0]
    x[:,k+1] = np.transpose(model.eq(np.concatenate((u[:,k],x[:,k]))))
```

11.12.4 Results

The goal is to find a trajectory that steers the vehicle as close to the provided path points as possible. The trajectory should also be feasible with respect to the vehicle dynamics and its safety and physical limitations. The 2D calculated vehicle's trajectory at timestep $k = 40$ is presented in blue in Figure 11.40. Here, you can see the current predictions for the trajectory marked green. The progress of the other states and inputs over time as well as their predictions is shown in Figure 11.41.

The trajectory and the progress of the system variables over the entire simulation period are presented in Figure 11.42 and Figure 11.43. One can see that all constraints are respected.

To see how the predictions of the system variables develop over all timesteps you can run the example file on your own machine.

You can find the code of this example in the **examples** folder that comes with your client.

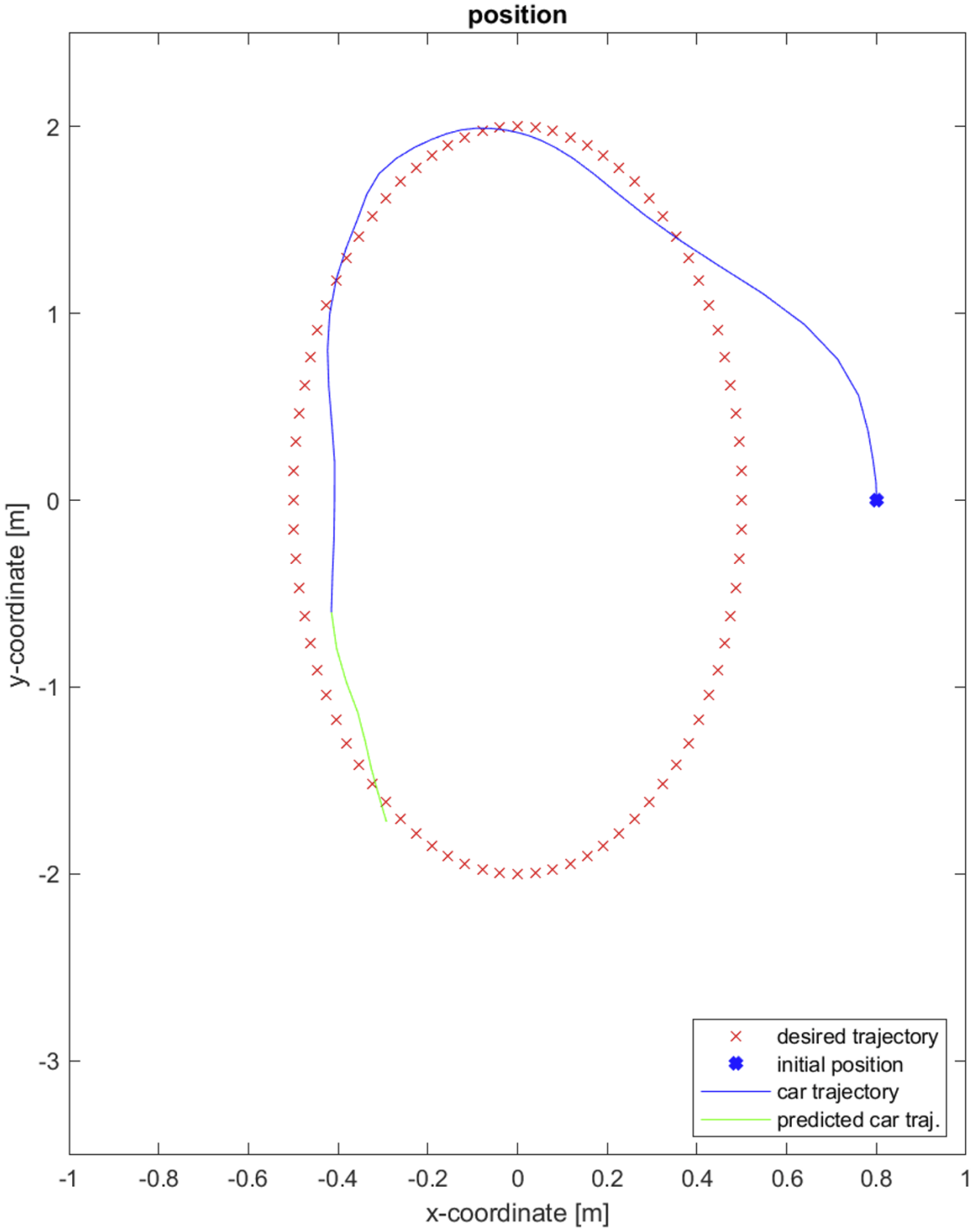


Figure 11.40: The calculated trajectory of the car (blue) and its predictions (green) at timestep $k = 40$

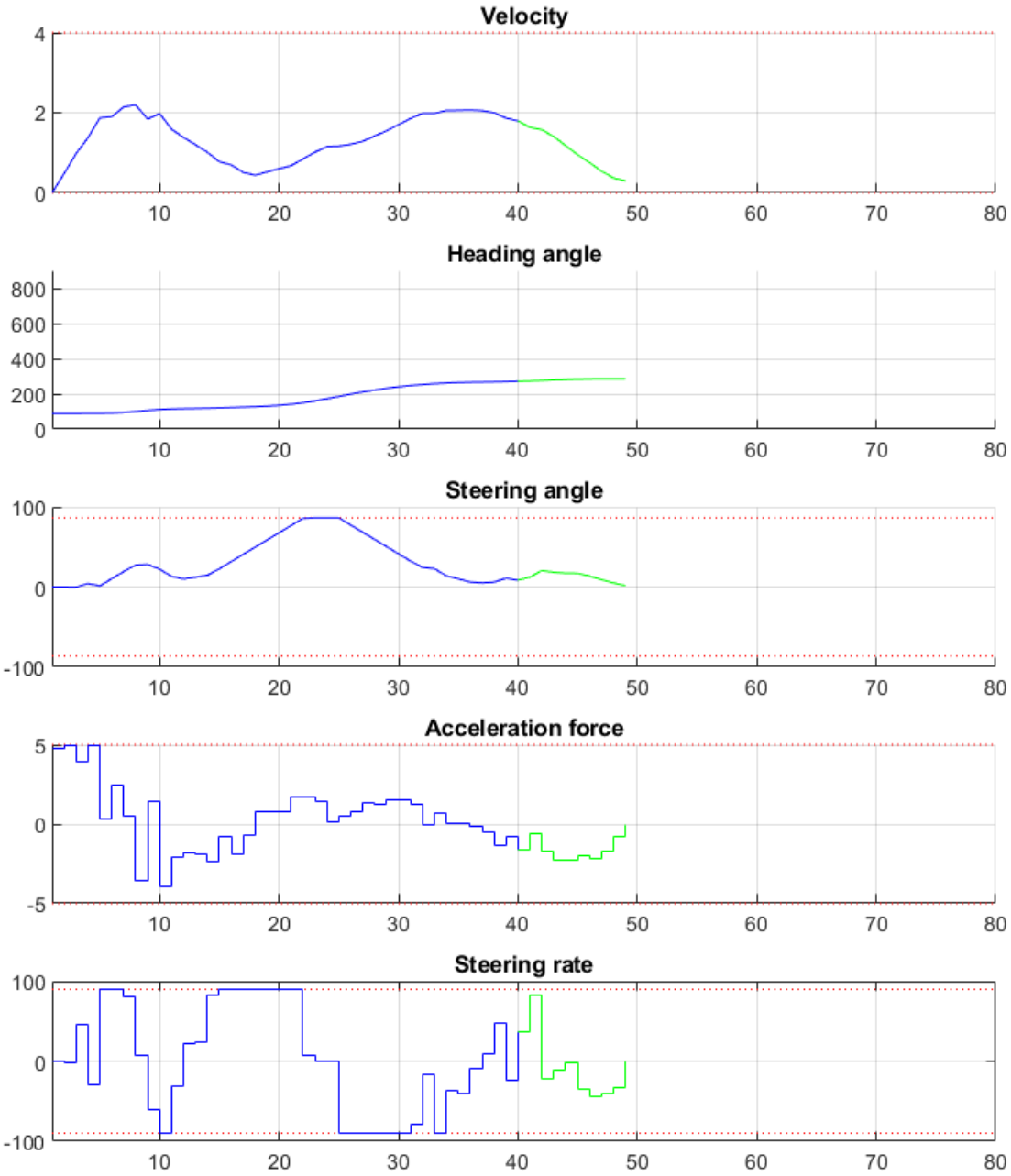


Figure 11.41: Development of the vehicle's states and the system's inputs over time (timestep $k = 40$)

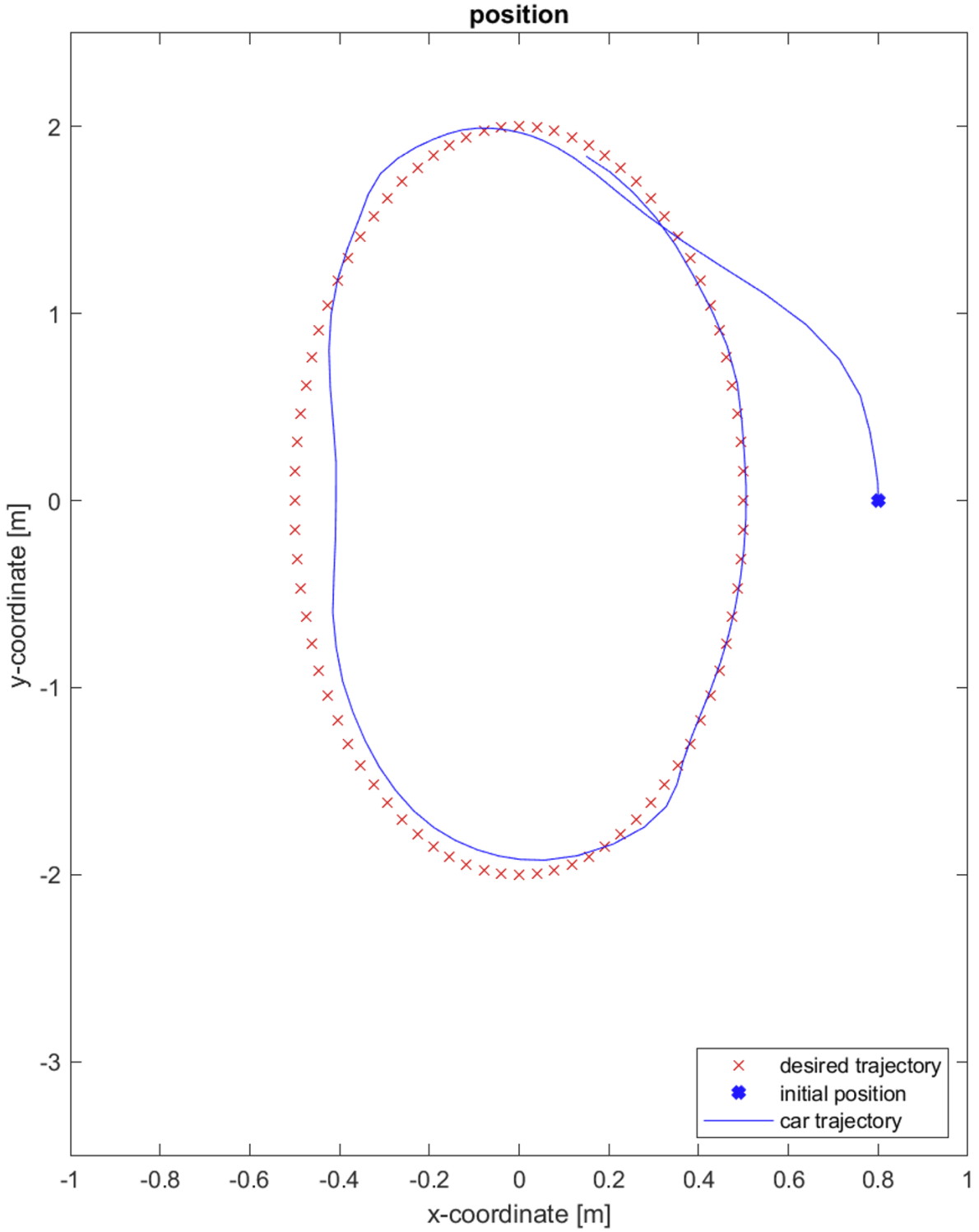


Figure 11.42: The calculated trajectory of the car

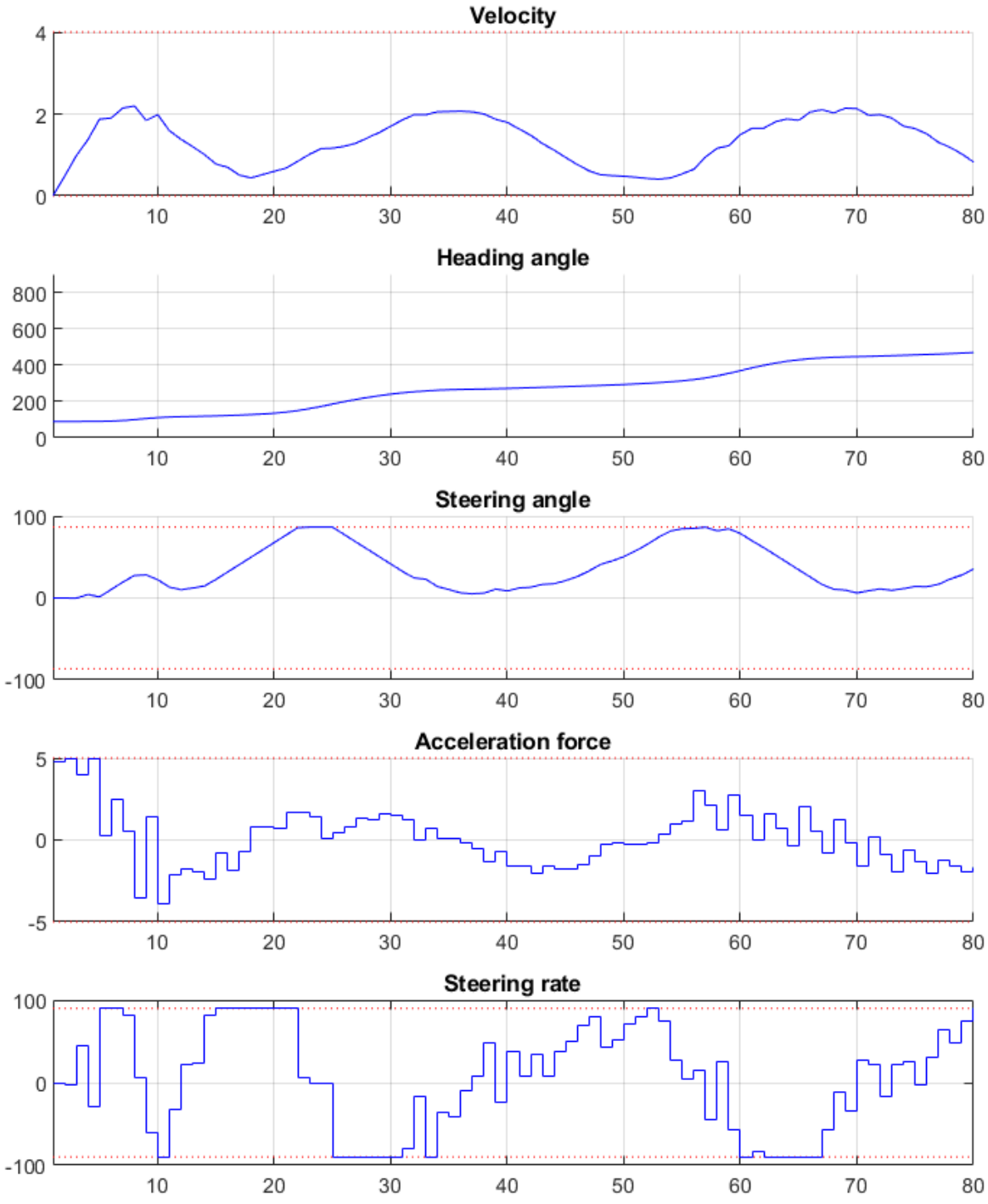


Figure 11.43: Development of the vehicle's states and the system's inputs over time

11.13 High-level interface: Rate Constraints

- *Implementation in MATLAB*
- *Results*

As in section *High-level interface: Basic example* we consider the following linear MPC problem with lower and upper bounds on state and inputs, and a terminal cost term:

$$\begin{aligned} \text{minimize} \quad & x_N^\top P x_N + \sum_{i=0}^{N-1} \left(x_i^\top Q x_i + u_i^\top R u_i \right) \\ \text{subject to} \quad & x_0 = x \\ & x_{i+1} = A x_i + B u_i \\ & \underline{x} \leq x_i \leq \bar{x} \\ & \underline{u} \leq u_i \leq \bar{u} \end{aligned}$$

This problem is parametric in the initial state x and the first input u_0 is typically applied to the system after a solution has been obtained. For the sake of this example, we assume $u_i \in \mathbb{R}$ and $x_i \in \mathbb{R}^2$ and write

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_1 \\ B_2 \end{pmatrix}.$$

In addition, we impose constraints on the input rate change $\Delta u_i = u_{i+1} - u_i$:

$$\underline{\Delta u} \leq \Delta u_i \leq \overline{\Delta u}.$$

The constraints can be included by defining states

$$z_i = \begin{pmatrix} \Delta u_i \\ u_i \\ x_i \end{pmatrix} \in \mathbb{R}^4.$$

The MPC problem now reads

$$\begin{aligned} \text{minimize} \quad & x_N^\top P x_N + \sum_{i=0}^{N-1} \left(x_i^\top Q x_i + u_i^\top R u_i \right), \\ \text{subject to} \quad & x_0 = x, \\ & E z_{i+1} = \hat{A} z_i, \\ & \underline{z} \leq z_i \leq \bar{z}. \end{aligned}$$

with

$$E = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \quad \hat{A} = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & B_1 & A_{11} & A_{12} \\ 0 & B_2 & A_{21} & A_{22} \end{pmatrix},$$
$$\underline{z} = (\underline{\Delta u}, \underline{u}, \underline{x})^\top, \quad \overline{z} = (\overline{\Delta u}, \overline{u}, \overline{x})^\top.$$

11.13.1 Implementation in MATLAB

The following code is based on the MATLAB code in *High-level interface: Basic example*. We define a variable **absrate** which limits the absolute value of Δu.

Matlab

Python

```
%% system
A = [1.1 1; 0 1];
B = [1; 0.5];
[nx,nu] = size(B);

%% MPC setup
Q = eye(nx);
R = eye(nu);
if( exist('dlqr','file') )
    [~,P] = dlqr(A,B,Q,R);
else
    P = 10*Q;
end

absrate = 0.5;
umin = -0.5;     umax = 0.5;
dumin = -absrate; dumax = absrate;
xmin = [-5, -5]; xmax = [5, 5];

%% FORCESPRO multistage form
% assume variable ordering z(i) = [u_i+1 - ui; ui; xi] for i=1...N

% dimensions
model.N     = 11;           % horizon length
model.nvar  = nu+nu+nx;     % number of variables
model.neq   = nu+nx;        % number of equality constraints

% objective
model.objective = @(z) z(2)*R*z(2) + [z(3); z(4)]'*Q*[z(3); z(4)];
model.objectiveN = @(z) [z(3); z(4)]'*P*[z(3); z(4)];

% equalities
model.eq = @(z) [ z(1) + z(2);
                  A(1,:)*[z(3); z(4)] + B(1)*z(2);
                  A(2,:)*[z(3); z(4)] + B(2)*z(2)];

model.E = [zeros(3,1), eye(3)];

% initial state
model.xinitidx = 3:4;

% inequalities
model.lb = [ dumin, umin,    xmin  ];
model.ub = [ dumax, umax,    xmax  ];
```

```
# system
A = np.array([[1.1, 1], [0, 1]])
B = np.array([[1], [0.5]])
nx, nu = np.shape(B)

# MPC setup
N = 10
Q = np.eye(nx)
R = np.eye(nu)
P = 10*Q
umin = -0.5
```

(continues on next page)

(continued from previous page)

```
umax = 0.5
absrate = 0.05
dumin = -absrate
dumax = absrate
xmin = np.array([-5, -5])
xmax = np.array([5, 5])

# FORCESPRO multistage form
# assume variable ordering zi = [u{i+1}-ui; ui; xi] for i=0...N

# dimensions
model = forcespro.nlp.ConvexSymbolicModel(11)   # horizon length N+1
model.nvar = 4   # number of variables
model.neq = 3   # number of equality constraints

# objective
model.objective = (lambda z: z[1]*R*z[1] +
                   casadi.horzcat(z[2], z[3]) @ Q @ casadi.vertcat(z[2], z[3]))
model.objectiveN = (lambda z:
                    casadi.horzcat(z[2], z[3]) @ P @ casadi.vertcat(z[2], z[3]))

# equalities
model.eq = lambda z: casadi.vertcat( z[0] + z[1],
                                casadi.dot(A[0, :], casadi.vertcat(z[2], z[3])) + B[0,␣
↪:]*z[1],
                                casadi.dot(A[1, :], casadi.vertcat(z[2], z[3])) + B[1,␣
↪:]*z[1])

model.E = np.concatenate([np.zeros((3, 1)), np.eye(3)], axis=1)

# initial state
model.xinitidx = [2, 3]

# inequalities
model.lb = np.concatenate([[dumin, umin], xmin])
model.ub = np.concatenate([[dumax, umax], xmax])
```

You can find the MATLAB code of this example to try it out for yourself in the **examples** folder that comes with your client.

11.13.2 Results

We run the simulation for different values of **absrate**. The results of the simulation are presented below. The plot on the top shows the system's states over time, the plot in the middle shows the input commands, the plot on the bottom shows the input rate change. We can see that all constraints are respected. We observe that compared to *High-level interface: Basic example* the behavior does not change for **absrate >= 0.1** (see Figure 11.44). If **absrate = 0.05**, it takes more time to steer the state to its setpoint (see Figure 11.45).

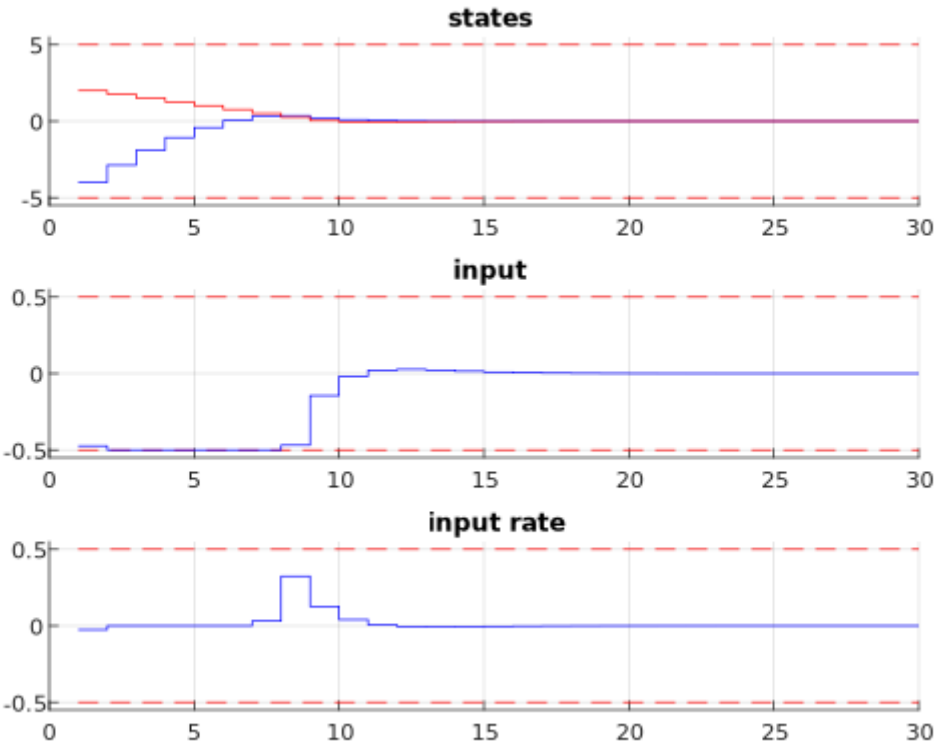


Figure 11.44: Simulation results of the states (top, in blue and red), input (middle, in blue), and input rate change (bottom, in blue) over time. The constraints are plotted in red dashed lines. The rate constraint is set to `0.5` and is not active at any moment.

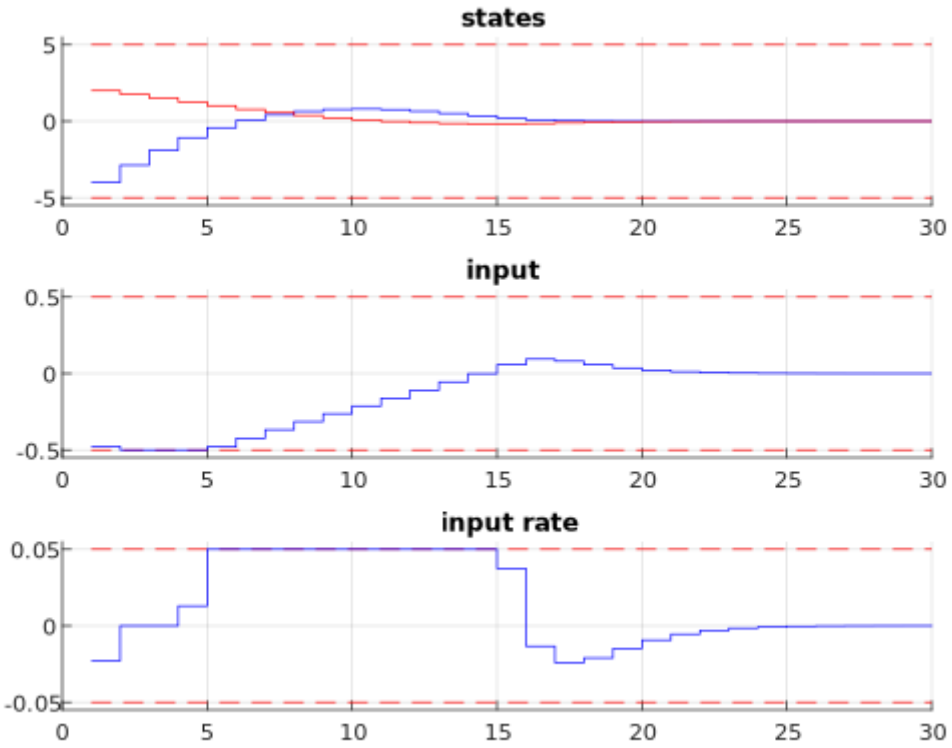


Figure 11.45: Simulation results of the states (top, in blue and red), input (middle, in blue), and input rate change (bottom, in blue) over time. The constraints are plotted in red dashed lines. The rate constraint is set to `0.05` and is active at some points.

11.14 High-level interface: Soft Constraints

- *Implementation in MATLAB*
- *Results*

As in section *High-level interface: Basic example* we consider the following linear MPC problem with lower and upper bounds on state and inputs, and a terminal cost term:

$$\begin{aligned} \text{minimize} \quad & x_N^\top P x_N + \sum_{i=0}^{N-1} \left(x_i^\top Q x_i + u_i^\top R u_i \right) \\ \text{subject to} \quad & x_0 = x \\ & x_{i+1} = A x_i + B u_i \\ & \underline{x} \leq x_i \leq \bar{x} \\ & \underline{u} \leq u_i \leq \bar{u} \end{aligned}$$

This problem is parametric in the initial state x and the first input u_0 is typically applied to the system after a solution has been obtained. For the sake of this example, we assume $u_i \in \mathbb{R}$ and $x_i \in \mathbb{R}^2$ and write

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_1 \\ B_2 \end{pmatrix}.$$

Suppose we want to allow the inequality constraints for u_i to be slightly violated. In this case, we introduce a slack variable $s_i \geq 0$ and write

$$\underline{u} - s_i \leq u_i \leq \overline{u} + s_i.$$

We want to punish positive values of s_i by adding a penalty term to the objective function. We use a hyperparameter $\lambda \geq 0$ and write

$$x_N^\top P x_N + \sum_{i=0}^{N-1} \left(x_i^\top Q x_i + u_i^\top R u_i + \lambda s_i \right).$$

Here, we use a penalty term linear in s_i. For λ large enough, the slack variables s_i are only chosen positive if the problem is infeasible with a hard constraint. For small λ, it may be optimal to slightly violate the constraints even though the original problem is feasible. It is also common to choose a penalty term quadratic in s_i. In order to use the FORCESPRO framework, we need to recast the new inequality constraints:

$$\begin{aligned} \text{minimize} \quad & x_N^\top P x_N + \sum_{i=0}^{N-1} \left(x_i^\top Q x_i + u_i^\top R u_i + \lambda s_i \right), \\ \text{subject to} \quad & x_0 = x, \\ & x_{i+1} = A x_i + B u_i, \\ & \underline{x} \leq x_i \leq \bar{x}, \\ & h_1(u_i, s_i) \leq \bar{u}, \\ & \underline{u} \leq h_2(u_i, s_i), \end{aligned}$$

with $h_1(u_i, s_i) = u_i - s_i$ and $h_2(u_i, s_i) = u_i + s_i$.

11.14.1 Implementation in MATLAB

The following code is based on the MATLAB code in *High-level interface: Basic example*. The modified inequality constraints can be implemented as follows:

Matlab

Python

```
%% relaxed inequality constraints
% assume variable ordering zi = [si, ui, xi]

model.nh    = 2;                    % number of inequality constraints
model.ineq  = @(z) [ z(2) - z(1);   % h_1
                     z(2) + z(1)];  % h_2
model.hu    = [umax, +inf];         % upper bound on inequality constraints
model.hl    = [-inf, umin];         % lower bound on inequality constraints
```

```
# relaxed inequalities constraints
# assume variable ordering zi = [si, ui, xi]

model.ineq = lambda z: casadi.vertcat( z[1] - z[0],
                                       z[1] + z[0])
model.hu = np.array([umax, +float('inf')])
model.hl = np.array([-float('inf'), umin])
```

The resulting code is depicted below.

Matlab

Python

```
%% system
A = [1.1 1; 0 1];
B = [1; 0.5];
[nx,nu] = size(B);
lambda = 8; % measure for penalty term

%% MPC setup
Q = eye(nx);
R = eye(nu);
if( exist('dlqr','file') )
    [~,P] = dlqr(A,B,Q,R);
else
    P = 10*Q;
end
umin = -0.5;     umax = 0.5;
xmin = [-5, -5]; xmax = [5, 5];

%% FORCESPRO multistage form
% assume variable ordering z(i) = [si; ui; xi] for i=1...N

% dimensions
model.N     = 11;   % horizon length
model.nvar  = 4;    % number of variables
model.neq   = 2;    % number of equality constraints
model.nh    = 2;    % number of inequality constraints

% objective with penalty term
model.objective = @(z)  z(2)*R*z(2) + [z(3); z(4)]'*Q*[z(3); z(4)] + lambda*z(1);
model.objectiveN = @(z) [z(3); z(4)]'*P*[z(3); z(4)];

% equalities
model.eq = @(z) [ A(1,:)*[z(3); z(4)] + B(1)*z(2);
                  A(2,:)*[z(3); z(4)] + B(2)*z(2)];

model.E = [zeros(2,2), eye(2)];
```

(continues on next page)

(continued from previous page)

```
% initial state
model.xinitidx = 3:4;

% relaxed inequalities
model.ineq = @(z) [ z(2) - z(1);
                    z(2) + z(1)];
model.hu = [umax, +inf];
model.hl = [-inf, umin];


model.lb = [0,      -inf, xmin  ];
model.ub = [+inf,   +inf, xmax  ];
```

```
# system
A = np.array([[1.1, 1], [0, 1]])
B = np.array([[1], [0.5]])
nx, nu = np.shape(B)
lam = 8 # measure for penalty term

# MPC setup
N = 10
Q = np.eye(nx)
R = np.eye(nu)
P = 10*Q
umin = -0.5
umax = 0.5
xmin = np.array([-5, -5])
xmax = np.array([5, 5])

# FORCESPRO multistage form
# assume variable ordering zi = [si; ui; xi] for i=0...N

# dimensions
model = forcespro.nlp.ConvexSymbolicModel(11)   # horizon length N+1
model.nvar = 4    # number of variables
model.neq = 2   # number of equality constraints
model.nh = 2

# objective with penalty term
model.objective = (lambda z: z[1]*R*z[1] + lam*z[0] +
                  casadi.horzcat(z[2], z[3]) @ Q @ casadi.vertcat(z[2], z[3]))
model.objectiveN = (lambda z:
                    casadi.horzcat(z[2], z[3]) @ P @ casadi.vertcat(z[2], z[3]))

# equalities
model.eq = lambda z: casadi.vertcat(casadi.dot(A[0, :], casadi.vertcat(z[2], z[3])) +␣
↪B[0, :]*z[1],
                                    casadi.dot(A[1, :], casadi.vertcat(z[2], z[3])) + B[1,␣
↪:]*z[1])

model.E = np.concatenate([np.zeros((2, 2)), np.eye(2)], axis=1)

# initial state
model.xinitidx = [2, 3]
```

(continues on next page)

(continued from previous page)

```
# relaxed inequalities
model.ineq = lambda z: casadi.vertcat( z[1] - z[0],
                                       z[1] + z[0])
model.hu = np.array([umax, +float('inf')])
model.hl = np.array([-float('inf'), umin])

# inequalities
model.lb = np.concatenate([[0, -float('inf')], xmin])
model.ub = np.concatenate([[float('inf'), float('inf')], xmax])
```

You can find the MATLAB code of this example to try it out for yourself in the `examples` folder that comes with your client.

11.14.2 Results

We run the simulation for $\lambda = 2, 8$. The results of the simulation are presented in Figure 11.46 below. The plot on the top shows the system's states over time, the plot on the bottom shows the input commands. For $\lambda = 2$, the constraints on u are clearly violated, for $\lambda = 8$, these constraints are only slightly violated.

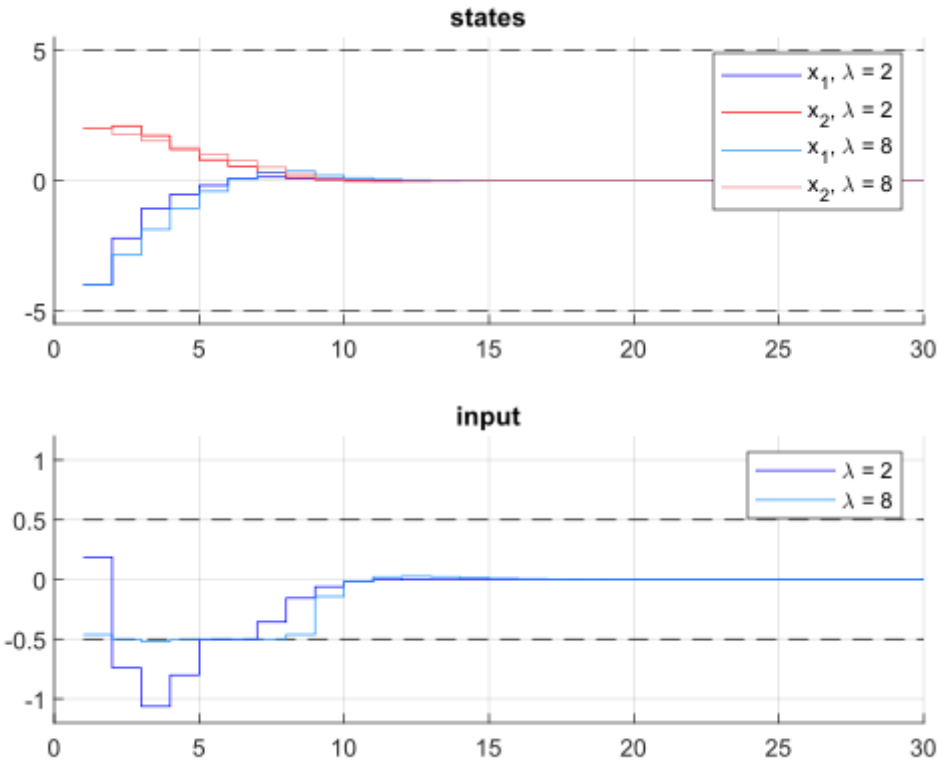


Figure 11.46: Simulation results of the states (top), input (middle) over time. The constraints are plotted in black dashed lines.

11.15 Controlling a crane using a FORCESPRO NLP solver

- *Defining the MPC problem*
  - *Model dimensions and dynamics*
  - *System constraints*
  - *Objective function*
- *Generating a FORCESPRO interior point NLP solver*
- *Calling the crane solver*
- *Results*

In this example we will see how to control a crane using the FORCESPRO interior point NLP solver. One interesting feature of this system is that is has a rather large linear subsystem which FORCESPRO can exploit for performance (see *Linear subsystem exploitation*). The crane is described by the following states:

x_C : cart position
v_C : cart velocity
x_L : cable length
v_L : rate of change of cable length
θ : angle of pendulum
ω : rate of change of angle
uC : voltage for horizontal actuator
uL : voltage for rotating actuator

and the control inputs are given by the voltage rate for the horizontal actuator u_{CR} and the voltage rate of the rotating actuator u_{LR}. The system dynamics are described by the following ODE:

$$\begin{aligned}
\dot{x}_C &= v_C \\
\dot{v}_C &= a_C \\
\dot{x}_L &= v_L \\
\dot{v}_L &= a_L \\
\dot{\theta} &= \omega \\
\dot{\omega} &= -\frac{a_C \cos(\theta) + g \sin(\theta) + 2 v_L \omega}{x_L} \\
\dot{u}_C &= u_{CR} \\
\dot{u}_L &= u_{LR}
\end{aligned}$$

where $a_C = \frac{-v_C}{\tau} + \frac{A_C u_C}{\tau}$ and $a_L = \frac{-v_L}{\tau} + \frac{A_L u_L}{\tau}$ and the constants are given by

$g = 9.81$ (gravitational constant)
$A_C = 0.0474$ (gain of GC(s) in m/s/V)
$A_L = 0.0341$ (gain of GL(s) m/s/V)
$\tau = 0.0247$ (time constant of winch dynamics in seconds)

For further details on these models we refer to [VukLoock] and [QuirDiehl].

You can find the MATLAB code below for this example to try it out for yourself in the `examples` folder that comes with your client.

11.15.1 Defining the MPC problem

Model dimensions and dynamics

The following code-snippet shows how to define the MPC problem associated with controlling the crane in FORCESPRO. The primal variable of our optimization problem is

$$z = \begin{pmatrix} u \\ x \end{pmatrix}$$

```
%% Define crane model
% Dimensions
model.N     = 20;       % horizon length
model.nvar  = 10;       % number of variables
model.neq   = 8;        % number of equality constraints
model.nh    = 0;        % number of inequality constraint functions
model.npar  = 2;        % number of parameters (these will be the reference values to␣
↪track)
nx = 8;
nu = 2;

% Dynamics
model.E = [zeros(nx,nu), eye(nx)];
model.continuous_dynamics = @(x,u,p) ode(x,u,p);
```

Here the right-hand-side of the differential equation **ode** is defined by the following MATLAB function

```
function [ dx ] = ode(x,u,~)
    g = 9.81; % gravitational constant
    AC = 0.0474; % gain of GC(s) in m/s/V
    AL = 0.0341; % gain of GL(s) m/s/V
    tau = 0.0247; % time constant of winch dynamics in seconds


    uCR = u(1); % voltage rate for horizontal actuator
    uLR = u(2); % voltage rate for rotating actuator

    xC = x(1); % cart position
    vC = x(2); % cart velocity
    xL = x(3); % cable length
    vL = x(4); % rate of change of cable length
    theta = x(5); % angle of pendulum
    omega = x(6); % rate of change of angle
    uC = x(7); % voltage for horizontal actuator
    uL = x(8); % voltage for rotating actuator

    aT = -(1/tau)*vC + (AC/tau)*uC;
    aL = -(1/tau)*vL + (AL/tau)*uL;

    dx = [  vC; ...
            aT; ...
            vL; ...
            aL; ...
            omega; ...
            -(1/xL)*(aT*cos(theta) + g*sin(theta) + 2*vL*omega); ...
            uCR; ...
```

(continues on next page)

(continued from previous page)

```
            uLR   ];
end
```

System constraints

We put simple constraints on both of the control inputs as well as the voltage for horizontal actuator and the voltage for rotating actuator. We also specify that we have an initial condition for all the states.

```
% Bounds
model.lb = [ -100, -100, -inf, -inf, -inf, -inf, -inf, -inf, -10, -10 ];
model.ub = [ +100, +100, +inf, +inf, +inf, +inf, +inf, +inf, +10, +10 ];

% Initial state
xinitidx = 3:10;
model.xinitidx = xinitidx;
```

Objective function

The goal of the control will be to track reference values for the cart position and cable length of the crane. Hence, it makes sense to use a Gauss-Newton hessian approximation in our optimization problem. Hence, in FORCESPRO we specify a least squares objective function

```
% Least squares objective function
model.LSobjective = @(z, p) LScost(z, p);
```

where the `LScost` function is defined as follows

```
function [ r ] = LScost(z,p)
    ep = 1e-5;
    cst = 50;
    sep = sqrt(ep);
    scst = sqrt(cst);
    r = [ sep*z(1); sep*z(2); scst*(z(3)-p(1)); sep*z(4); scst*(z(5)-p(2)); sep*z(6);␣
↪sep*z(7); sep*z(8); sep*z(9); sep*z(10)];
end
```

11.15.2 Generating a FORCESPRO interior point NLP solver

In order to generate a solver we first need to choose options to specify the algorithmic specifications (see *Solver Options*) we want implemented in our solver. The two most important options to mention here is that we specify to use a Gauss-Newton hessian approximation and we want to allow FORCESPRO to exploit linear subsystems of our dynamics.

```
%% Set codeoptions to specify solver settings
codeoptions = getOptions('CraneSolver');
Ts = 1/100; % sampling time
codeoptions.nlp.integrator.Ts = Ts;
nodes = 4;
codeoptions.nlp.integrator.nodes = nodes;
codeoptions.nlp.integrator.type = 'ERK4';
codeoptions.nlp.integrator.attempt_subsystem_exploitation = 1; % Enable subsystem␣
```

(continues on next page)

(continued from previous page)

```
↪exploitation for performance
codeoptions.printlevel = 0;
codeoptions.nlp.hessian_approximation = 'gauss-newton';
codeoptions.nlp.ad_tool = 'casadi';

% Generate solver
FORCES_NLP(model, codeoptions);
```

The last command will generate a FORCESPRO solver which can now be called from MATLAB via the name `CraneSolver`.

11.15.3 Calling the crane solver

With our FORCESPRO controller at hand we can easily simulate our system in MATLAB as the following code-snippet shows.

```
%% Simulation
totalTime = 100; % number of seconds
nSamples = totalTime / Ts;

t = 0;
all_refs = zeros(2, nSamples);
cart_data = zeros(2, nSamples);
time = zeros(1, nSamples);
x = [ 0.15; 0; 0.7; 0; 0; 0; 0; 0];
for ii = 1:nSamples

    % get current reference
    ref = getRef(t, totalTime);

    % record data for plotting
    all_refs(:,ii) = ref;
    cart_data(:,ii) = x([1, 3]);
    time(ii) = t;

    % set up problem data
    problem.xinit = x;
    problem.x0 = repmat([0; 0; x],model.N,1);
    problem.all_parameters = repmat(ref,model.N,1);

    % call FORCESPRO solver and check exit status
    [solution, exitflag, info] = CraneSolver(problem);
    if exitflag ~= 1
        error('Encountered solver failure.');
    end

    % extract control and update state
    u = solution.x01(1:2);
    x = RK4( x, u, @(x,u,p) ode(x,u,p), Ts, ref, nodes);
    t = t + Ts;
end
```

11.15.4 Results

As can be seen from figures Figure 11.47 and Figure 11.48 below the FORCESPRO controller achieves tracking the reference values almost perfectly. A benchmark running the `CraneSolver` with and without linear subsystem exploitation on a Raspberry Pi 3 showed an overall reduction of computation time by 22% when exploiting linear subsystems.

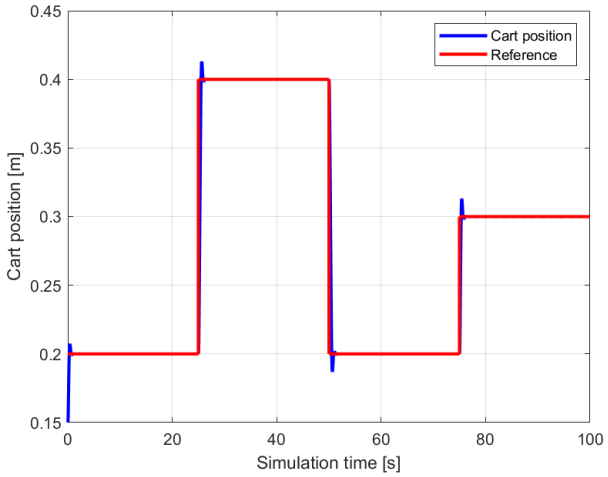


Figure 11.47: The reference values for the cart position (x_C) seen in red and the simulated cart position seen in blue.

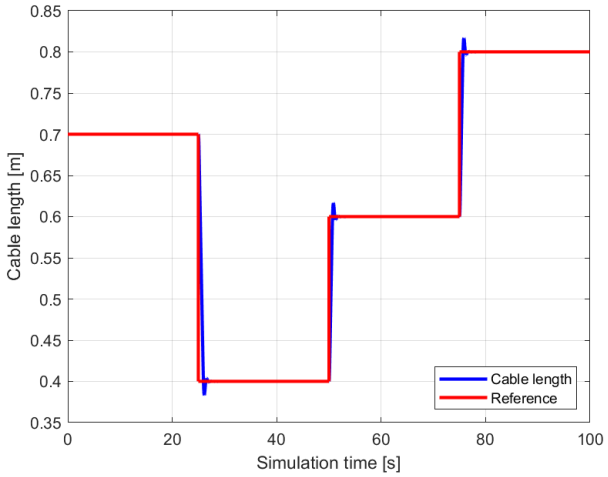


Figure 11.48: The reference values for the cable length (x_L) seen in red and the simulated cable length seen in blue.

11.16 Real-time SQP Solver: Robotic Arm Manipulator (MATLAB & Python)

- *Defining the MPC problem*
  - *Tracking objective*

  - *State and input constraints*
  - *Initial condition and horizon length*
- *Generating a real-time SQP solver*
- *Calling the generated SQP solver*
- *Calling the dynamics used in the model directly in MATLAB/Python*
- *Results*

In this example we illustrate the use of the real-time Sequential Quadratic Programming (SQP) solver and how to generate and call the dynamics used in the formulation of the optimization problem directly in MATLAB/Python. In particular, we use a robotic arm manipulator described by a set of ordinary differential equations (ODEs):

$$\begin{aligned}\ddot{\theta}_1 &= \gamma \\ \ddot{\theta}_2 &= \frac{1}{\beta_2}(\tau_2 - \beta_1\gamma - \beta_3\dot{\theta}_1^2 - \beta_4) \\ \dot{\tau}_1 &= u_1 \\ \dot{\tau}_2 &= u_2\end{aligned}$$

where θ_1, θ_2 are joint angles modelling the manipulator configuration, u_1, u_2 are the rates (inputs) of the torques τ_1, τ_2 applied to the joints and

$$\gamma \hat{=} \frac{1}{\alpha_1 - \alpha_2\frac{\beta_1}{\beta_2}}(\frac{\alpha_2}{\beta_2}(\beta_4 + \beta_3\dot{\theta}_1^2 - \tau_2) - \alpha_3\dot{\theta}_1\dot{\theta}_2 - \alpha_4\dot{\theta}_2 - \alpha_5 + \tau_1).$$

The coefficients $\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5$ and $\beta_1, \beta_2, \beta_3, \beta_4$ depend on the inertia and mass of the robot arm components. Their expressions can be found in [SicSci09]. The optimal control problem is formalized from the state x defined by

$$x \hat{=} (\theta_1, \dot{\theta}_1, \theta_2, \dot{\theta}_2, \tau_1, \tau_2)^\top$$

and the input u defined as

$$u \hat{=} (\dot{\tau}_1, \dot{\tau}_2)^\top.$$

The control objective is to make the first joint angle θ_1 follow a reference of $1.2\,\mathrm{rad}$ from 0 to $10\,\mathrm{s}$ and $-1.2\,\mathrm{rad}$ from 10 to $20\,\mathrm{s}$. Similarly, the second joint angle θ_2 should follow a reference of $-1.2\,\mathrm{rad}$ from 0 to $10\,\mathrm{s}$ and $1.2\,\mathrm{rad}$ from 10 to $20\,\mathrm{s}$. The stage variable z is defined by stacking the input and differential state variables:

$$z = (\dot{\tau}_1, \dot{\tau}_2, \theta_1, \dot{\theta}_1, \theta_2, \dot{\theta}_2, \tau_1, \tau_2)^\top$$

You can find the code of this example to try it out for yourself in the `examples` folder that comes with your client.

11.16.1 Defining the MPC problem

Tracking objective

Our goal is to minimize the distance of the joint angles to the reference, which can be translated in the following stage cost function:

$$f(z,p) = 1000(z_3 - 1.2p)^2 + 0.1z_4^2 + 1000(z_5 + 1.2p)^2 + 0.1z_6^2 + 0.01z_7^2 + 0.01z_8^2 + 0.01z_1^2 + 0.01z_2^2,$$

where p is a run-time parameter taking value 1 from 0 to $10\,\mathrm{s}$ and -1 from 10 to $20\,\mathrm{s}$.

The stage cost function is coded in MATLAB as the least-squares vector:

Matlab

Python

```
model.LSobjective = @(z,p)[sqrt(1000) * (z(3)-p(1)*1.2); ...
                           sqrt(0.1) * z(4); ...
                           sqrt(1000) * (z(5)+p(1)*1.2); ...
                           sqrt(0.1) * z(6); ...
                           sqrt(0.01) * z(7); ...
                           sqrt(0.01) * z(8); ...
                           sqrt(0.01) * z(1); ...
                           sqrt(0.01) * z(2)];
```

```
model.LSobjective = lambda z, p: casadi.vertcat(np.sqrt(1000) * (z[2] - p[0]*1.2),
                                                np.sqrt(0.1) * (z[3]),
                                                np.sqrt(1000) * (z[4] + p[0]*1.2),
                                                np.sqrt(0.1) * z[5],
                                                np.sqrt(0.01) * z[6],
                                                np.sqrt(0.01) * z[7],
                                                np.sqrt(0.01) * z[0],
                                                np.sqrt(0.01) * z[1])
```

In the MATLAB example, this is needed to compute a Gauss-Newton approximation from the Jacobian of the least-squares vector. In the Python example, where Gauss-Newton approximations are not yet available, we use the *objective* field to supply the target function.

State and input constraints

The following constraints are imposed on the torques and torque rates:

$$\begin{aligned} -100\,\mathrm{Nm} \leq &\tau_1 \leq 70\,\mathrm{Nm} \\ -100\,\mathrm{Nm} \leq &\tau_2 \leq 70\,\mathrm{Nm} \\ -200\,\mathrm{Nm/s} \leq &\dot{\tau}_1 \leq 200\,\mathrm{Nm/s} \\ -200\,\mathrm{Nm/s} \leq &\dot{\tau}_2 \leq 200\,\mathrm{Nm/s} \end{aligned}$$

This corresponds to the code below.

Matlab

Python

```
% upper/lower variable bounds lb <= x <= ub
model.lb = [ -200,  -200,   -pi,  -100,   -pi,  -100,  -100,  -100 ];
model.ub = [  200,   200,    pi,   100,    pi,   100,    70,    70 ];
```

```
# Upper/lower variable bounds lb <= x <= ub
#                                Inputs  |                States
#                    dtau1  dtau2  theta1  dtheta1  theta2  dtheta2  tau1  tau2
model.lb = np.array([ -200,  -200, -np.pi,    -100, -np.pi,    -100, -100, -100])
model.ub = np.array([  200,   200,  np.pi,     100,  np.pi,     100,   70,   70])
```

Initial condition and horizon length

The prediction horizon is set to 21 and the following initial condition is set

Matlab

Python

```
model.xinit = [-0.4 0 0.4 0 0 0 ]';
model.xinitidx = 3:8;
```

```
xinit = np.array([-0.4, 0, 0.4, 0, 0, 0])
model.xinitidx = range(2, 8)
```

11.16.2 Generating a real-time SQP solver

We have now populated `model` with the necessary fields to generate an SQP solver, which requires settings a few options, as follows:

Matlab

Python

```
%% Define solver options
codeoptions = getOptions('RobotArmSolver');
codeoptions.maxit = 200;                                    % Maximum number of␣
↪iterations of inner QP solver
codeoptions.printlevel = 0;                                 % Use printlevel = 2 to␣
↪print progress (but not for timings)
codeoptions.optlevel = 3;
codeoptions.nlp.integrator.Ts = integrator_stepsize;
codeoptions.nlp.integrator.nodes = 5;
codeoptions.nlp.integrator.type = 'ERK4';
codeoptions.nlp.ad_tool = 'casadi';

% Options for SQP solver
codeoptions.solvemethod = 'SQP_NLP';
codeoptions.nlp.hessian_approximation = 'gauss-newton';     % Gauss-Newton hessian␣
↪approximation of nonlinear least-squares objective
codeoptions.MEXinterface.dynamics = 1; % Generate a MEX entrypoint for the nonlinear␣
↪dynamics of the robot
```

```
# Define solver options
codeoptions = forcespro.CodeOptions()
codeoptions.maxit = 200  # Maximum number of iterations
codeoptions.printlevel = 0  # Use printlevel = 2 to print progress (but not for␣
↪timings)
codeoptions.optlevel = 3  # 0 no optimization, 1 optimize for size, 2 optimize for␣
↪speed, 3 optimize for size & speed
codeoptions.nlp.integrator.Ts = integrator_stepsize
codeoptions.nlp.integrator.nodes = 5
codeoptions.nlp.integrator.type = 'ERK4'
codeoptions.solvemethod = 'SQP_NLP'
codeoptions.sqp_nlp.rti = 1
codeoptions.sqp_nlp.maxSQPit = 1
codeoptions.nlp.hessian_approximation = 'gauss-newton'

# Generate FORCESPRO solver
solver = model.generate_solver(codeoptions)
```

The number of solved QPs in every iteration is set via `sqp_nlp.maxSQPit`. It is important to note that disabling the line search in the SQP algorithm does not guarantee global con-

vergence and hence may result in less robust performance, but produces much faster solve times. Turning off the line search via **sqp_nlp.use_line_search** is only allowed when the Gauss-Newton approximation is on.

11.16.3 Calling the generated SQP solver

Once all parameters have been populated, the MEX interface of the solver can be used to invoke it from MATLAB, or the Python *Solver* class can be used to use it from within Python:

Matlab

Python

```
% Set primal initial guess
x0i = model.lb+(model.ub-model.lb)/2;
x0 = repmat(x0i',model.N,1);
problem.x0 = x0;

% Set reference as run-time parameter
problem.all_parameters = ones(model.N,1);

% Set initial condition
problem.xinit = X(:,i);

% Call SQP solver
[output, exitflag, info] = RobotArmSolver(problem);
```

```
# Set solver parameters
x0i = (model.ub + model.lb) / 2
x0 = np.tile(x0i, (1, model.N))
problem = {"x0": x0,  # Primal initial guess to start solver from
           "xinit": xinit,  # Initial condition
           "all_parameters": np.ones((model.N, 1))}  # Reference as a real-time␣
↪parameter

# Call SQP solver
output, exitflag, info = solver.solve(problem)
```

The *RobotArmSolver* is expected to return an *exitflag* equal to 1, which corresponds to a successful solver. However, note that the QP could become infeasible in some cases. In this case, one should expect an exitflag of −8.

11.16.4 Calling the dynamics used in the model directly in MATLAB/Python

The MEX interface for the dynamics used in the formulation of the optimization problem can also be called directly in MATLAB and in Python, the *Solver* class has a method which can compute the dynamics along with its derivative (see section *Calling the nonlinear functions from MATLAB or Python*). This is convenient for debugging a solver formulation or simulating the system dynamics. This can be done as follows:

Matlab

Python

```
% Set data at first stage
z = problem.x0(1:model.nvar);
p = problem.all_parameters(1:model.npar);

[c, jacc] = RobotArmSolver_dynamics(z, p);
```

```
# Set data at first stage
z = problem['x0'][0:model.nvar]
p = problem['all_parameters'][0:model.npar]
c, jacc = solver.dynamics(z,p)
```

This stores the integrated stage in `c` and its jacobian with respect to `z` in `jacc`.

11.16.5 Results

The control objective is to track the joint references of $-1.2\,\mathrm{rad}$ and $1.2\,\mathrm{rad}$ respectively, while keeping the input torque rates below $200\,\mathrm{Nm/s}$ in magnitude and the torque states between $-100\,\mathrm{N}$ and $70\,\mathrm{Nm}$.

The joint angle and torques trajectories are shown in Figure Figure 11.49 and Figure Figure 11.50 respectively, while the input torque rates are plotted in Figure Figure 11.51. The closed-loop objective, which is a measure of the control performance is shown in Figure Figure 11.52.

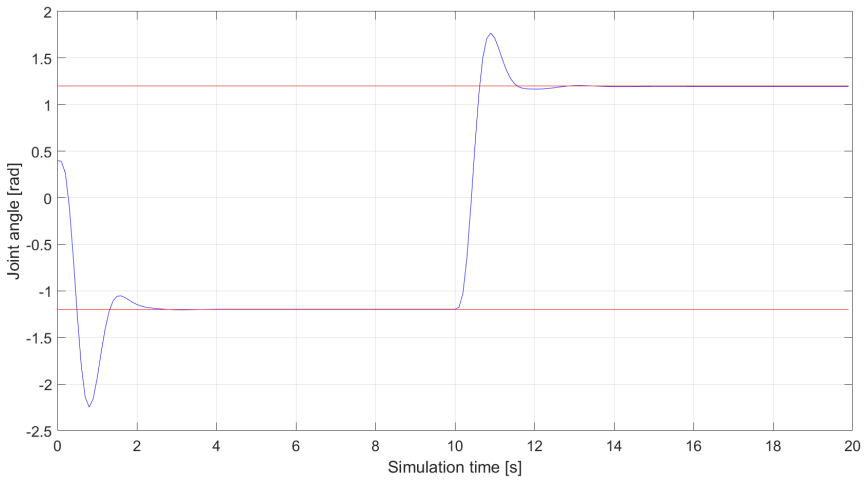


Figure 11.49: Manipulator's joint angle.

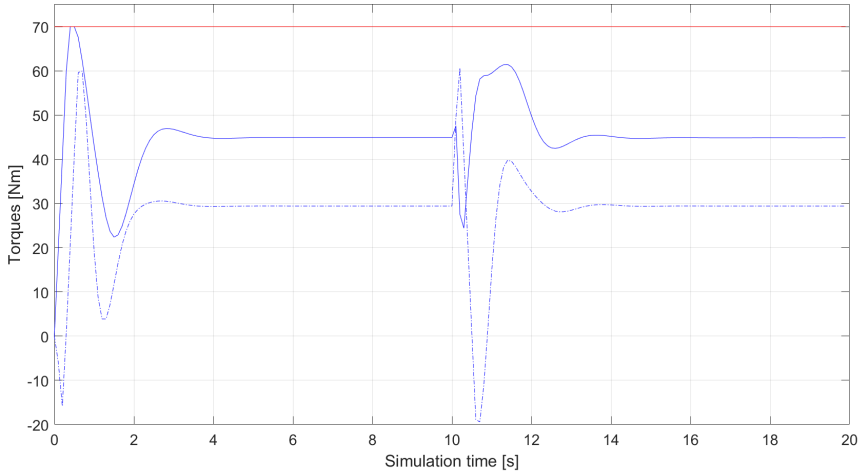


Figure 11.50: Manipulator's torques at joints.

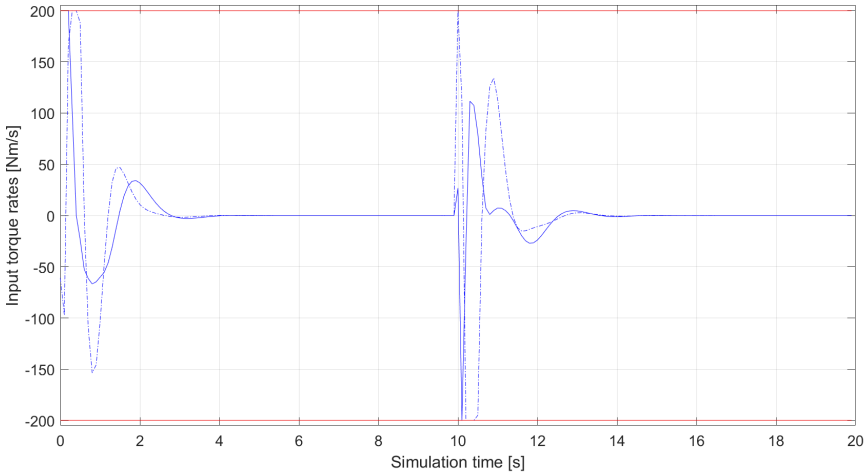


Figure 11.51: Manipulator's torque rates.

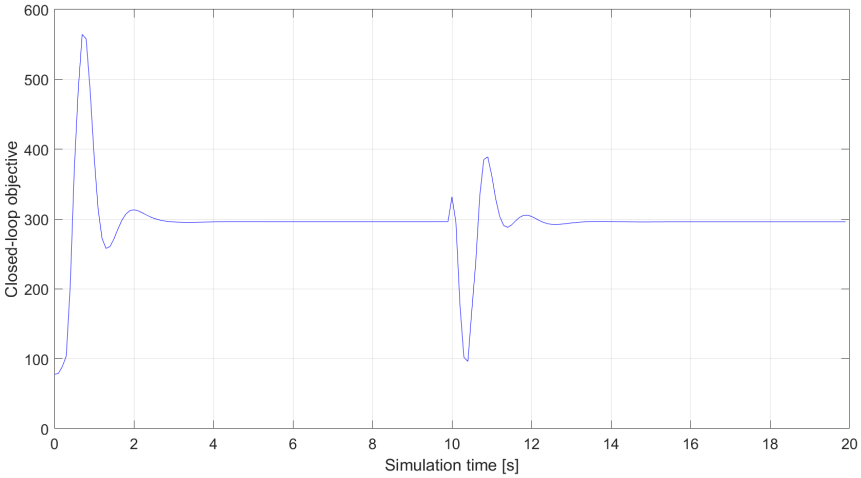


Figure 11.52: Manipulator's closed loop objective.

11.17 Controlling a DC motor using a FORCESPRO SQP solver

- *Defining the MPC problem*
  - *The tracking objective function*
  - *The dynamics*
  - *Input and state constraints*
  - *Generating the FORCESPRO SQP solver*
  - *Calling the solver*
  - *Results*

In this example our aim is to control a DC-motor using a FORCESPRO SQP solver. The model for the DC motor which we consider is borrowed from [BerUnb], to which we refer for further details. The dynamics of our model is described by the following set of ordinary differential equations:

$$\dot{x}_1(t) = -\frac{R_a}{L_a}x_1(t) - \frac{k_m}{L_a}u(t)x_2(t) + \frac{u_a}{L_a}$$
$$\dot{x}_2(t) = -\frac{B}{J}x_2(t) + \frac{k_m}{J}u(t)x_1(t) - \frac{\tau_l}{J}.$$

The states x_1 and x_2 model the armature current and motor angular speed of out DC motor respectively and the control u models the input field current. The following values are chosen for our model constants

$R_a = 12.548\Omega$	(armature resistance)
$L_a = 0.307\text{H}$	(armature inductance)
$k_m = 0.23576\text{Nm/A}^2$	(motor constant)
$u_a = 60\text{V}$	(armature voltage)
$B = 0.00783\text{Nmsec}$	(total viscous damping)
$\tau_L = 1.47\text{Nmsec}$	(Load torque)
$J = 0.00385\text{Nmsec}^2$	(total moment of inertia)

The control objective is to track a piecewise constant angular speed. To test the robustness of out resulting controller we switch reference half way through our simulation. In the first half of the simulation we will track a constant angular speed $x_2^{ref1} = 2$ and then switch to tracking a constant angular speed $x_2^{ref2} = -2$. We collect the 2-dimensional state vector $x = (x_1, x_2)^T$ and the 1-dimensional control u in the vector

$$z = \begin{pmatrix} u \\ x_1 \\ x_2 \end{pmatrix}$$

You can find the MATLAB code below for this example to try it out for yourself in the **`examples`** folder that comes with your client.

11.17.1 Defining the MPC problem

The tracking objective function

Since we want to track a reference value it is natural to consider a least squared cost $f(z,p) = \frac{\|r(z,p)\|^2}{2}$ for

$$r(z,p) = z_3 - p$$

Recall that $z_3 = x_2$ models the motor angular speed which is made to track a piecewise constant reference. The parameter p will be equal to x_2^{ref1} during the first half of the simulation and equal to x_2^{ref2} during the second.

The following code snippet reads in the least squared objective

```
model.LSobjective = @(z,p) sqrt(100) * (z(3) - p);
model.LSobjectiveN = @(z,p) sqrt(100) * (z(3) - p);
```

The dynamics

The following code snippet reads in the dynamics (11.17) of our model:

```
%% model parameters
% Armature inductance (H)
La = 0.307;
% Armature resistance (Ohms)
Ra = 12.548;
% Motor constant (Nm/A^2)
km = 0.23576;
% Total moment of inertia (Nm.sec^2)
J = 0.00385;
% Total viscous damping (Nm.sec)
B = 0.00783;
% Load torque (Nm)
tauL = 1.47;
% Armature voltage (V)
ua = 60;

model.E = [zeros(2,1), eye(2)];
model.continuous_dynamics = @(x,u) [(-1/La)*(Ra*x(1) + x(2)*u(1) - ua); ...
                                    (-1/J)*(B*x(2) - km*x(1)*u(1) + tauL)];
```

Input and state constraints

The following constraints are to be met by out control and state vectors:

$$1\text{A} \leq u \leq 1.6\text{A}$$
$$-5\text{A} \leq x_1 \leq 5\text{A}$$
$$-10\frac{\text{rad}}{\text{sec}} \leq x_2 \leq 2.004\frac{\text{rad}}{\text{sec}}$$

This can be read into the FORCESPRO model as follows

```
model.lb = [1; -5; -10];
model.ub = [1.6; 5; 2.004];
```

Generating the FORCESPRO SQP solver

To generate a suitable SQP solver for out MPC problem one need a `model` struct as well as a `codeoptions` struct. Our model struct has been populated above and we now specify the codeoptions we want and generating the solver by calling `FORCES_NLP`. The following codesnippet shows how this can be done:

```
%% set codeoptions
codeoptions = getOptions('FORCESPROSolver');
codeoptions.solvemethod = 'SQP_NLP'; % generate SQP-RTI solver
codeoptions.nlp.ad_tool = 'casadi';
codeoptions.BuildSimulinkBlock = 0;
codeoptions.nlp.integrator.type = 'ERK4';
integration_step = 0.01;
codeoptions.nlp.integrator.Ts = integration_step;
codeoptions.nlp.integrator.nodes = 1;
codeoptions.nlp.hessian_approximation = 'gauss-newton';
codeoptions.timing = 1;

%% generate FORCESPRO solver
FORCES_NLP(model, codeoptions);
```

Calling the solver

Once the solver has been generated it needs a struct containing an initial guess, initial condition of the ODE, the run-time parameters and the **reinitialize** field as explained in *Sequential quadratic programming algorithm*. The following code-snippet shows how this can be done:

```
% populate run time parameters struct
params.all_parameters = repmat(2, model.N, 1);
params.xinit = zeros(model.neq, 1); % initial condition to ODE
params.x0 = repmat([1.2; zeros(2,1)], model.N, 1); % initial guess
params.reinitialize = 0;

% Solve optimization problem
[output, exitflag, info] = FORCESPROSolver(params);
```

The FORCESPROSolver is expected to return an **exitflag** equal to 1, which corresponds to a successful solve. However, note that the QP could become infeasible in some cases. In this case, one should expect an **exitflag** equal to -8.

Results

The control objective is to track an angular speed of 2 and -2 respectively. As can be seen in Figure 11.57 the controller completes this task. The control (u) is plotted in Figure 11.53, the first state (x_1) is plotted in Figure 11.54 and second state (x_2) in Figure 11.55. As a measure of control quality, the closed loop objective value is plotted in Figure 11.56.

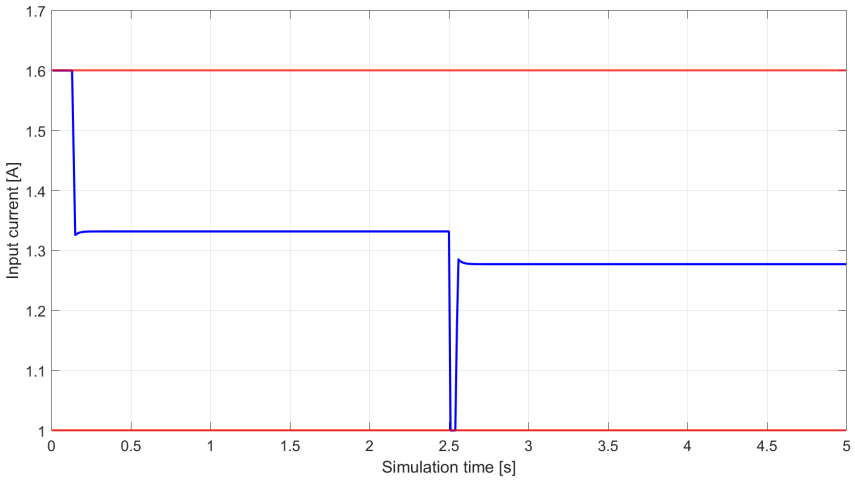


Figure 11.53: The control (u, in blue) as a function of simulation time (s). The control obeys its constraints (red) throughout the simulation.

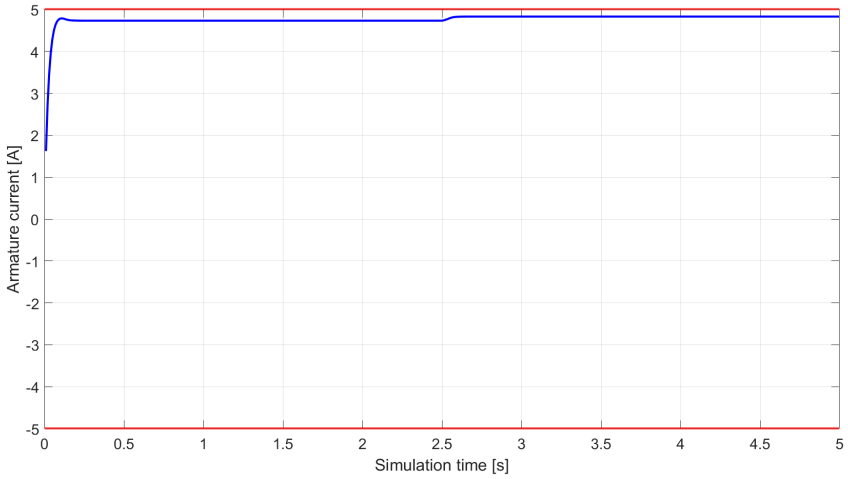


Figure 11.54: The first state (x_1, in blue) as a function of simulation time. It obeys its constraints (red) throughout the simulation.

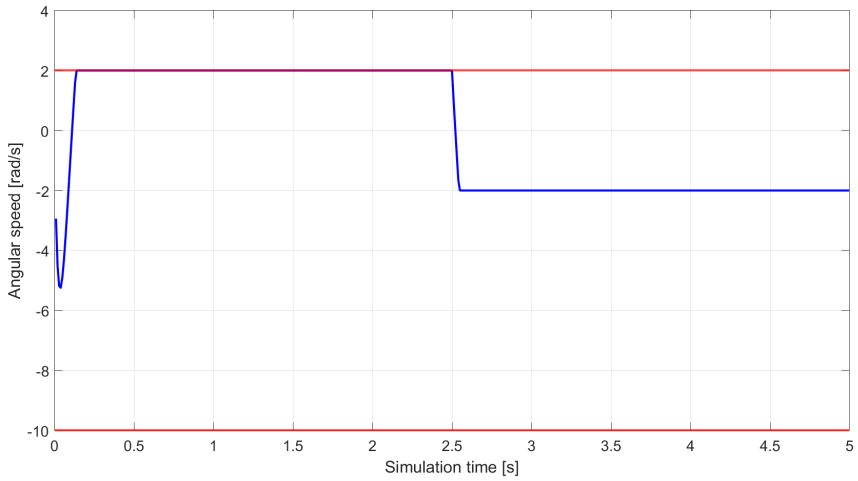


Figure 11.55: The second state (x_2, in blue) as a function of simulation time. It obeys its constraints (red) throughout the simulation.

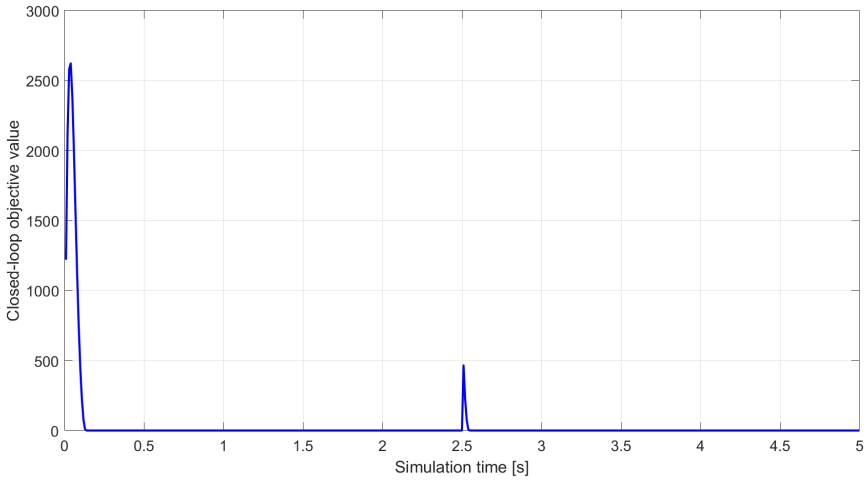


Figure 11.56: Closed-loop objective value as a function of time

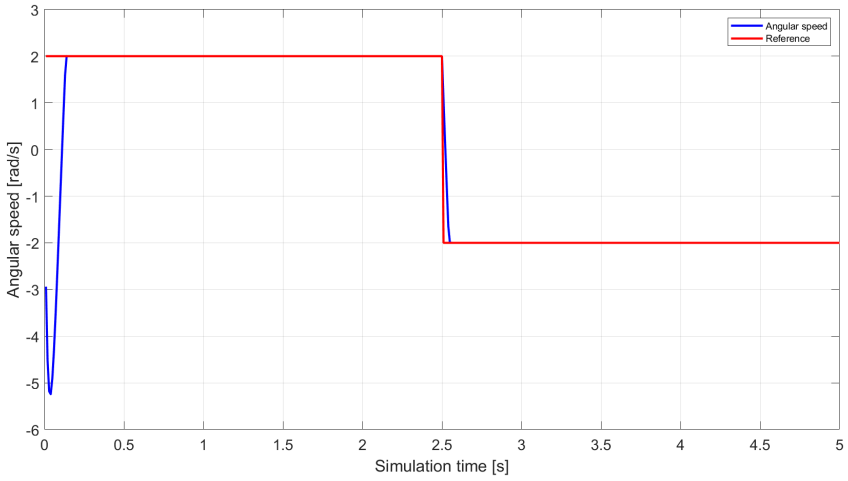


Figure 11.57: Angular speed (blue) and tracked reference (red) value as a function of time.

11.18 Mixed-integer nonlinear solver: F8 Crusader aircraft

- *Defining the problem data*
  - *Objective*
  - *Equality constraints*
  - *Inequality constraints*
  - *Initial and final conditions*
- *Defining the MPC problem*
- *Generating an MINLP solver*
- *Calling the generated MINLP solver*
- *Providing an initial guess at run-time*
- *Changing the parallelization strategy at run-time*
- *Results*

In this example we illustrate the simplicity of the high-level user interface on a mixed-integer nonlinear program. In particular, we use an F8 Crusader aircraft model described by a set of ordinary differential equations (ODEs):

$$\begin{aligned}\dot{x}_0 =& -0.877x_0 + x_2 - 0.088x_0x_2 + 0.47x_0^2 - 0.019x_1^2 - x_0^2x_2 + 3.846x_0^3 \\ & - 0.215w + 0.28x_0^2w + 0.47x_0w^2 + 0.63w^3 \\ \dot{x}_1 =& x_2 \\ \dot{x}_2 =& -0.4208x_0 - 0.396x_2 - 0.47x_0^2 - 3.564x_0^3 - 20.967w \\ & + 6.265x_0^2w + 46x_0w^2 + 61.4w^3\end{aligned}$$

The model is taken from [GarJor77] and consists of three differential states: x_0 the angle of attack in radians, x_1 the pitch angle in radians and x_2 the pitch angle rate in radians per second. There is one control input w, the tail deflection angle in radians. The input is the discrete component of the model, since it can take values within the discrete set $\{-0.05236, 0.05236\}$. This makes the solution process more complicated in comparison to a nonlinear program, as the different combinations of inputs have to be checked over the control horizon.

The trajectory of the aircraft is to be computed by solving a mixed-integer nonlinear program (MINLP). First, we define the stage variable z by stacking the input and differential state variables:

$$z = [w, x_0, x_1, x_2]^\top$$

You can find the MATLAB code of this example to try it out for yourself in the `examples` folder that comes with your client.

11.18.1 Defining the problem data

Objective

Our goal is to minimize the distance of the final state to the origin, which can be translated in the following cost function on the final stage variable:

$$f(z) = 150x_0^2 + 5x_1^2 + 5x_2^2$$

The terminal cost function is coded in MATLAB as the following function:

```
model.objectiveN = @(z) 150 * z(2)^2 + 5 * z(3)^2 + 5 * z(4)^2;
```

Moreover, control inputs are penalized at every stage via the following stage cost function:

```
model.objective =  @(z) 0.0001 * z(1)^2;
```

Equality constraints

In this example, the only equality constraints are related to the dynamics. They are provided to FORCESPRO in continuous form. The discretization is then computed internally by the FORCESPRO integrators.

In the code snippet below, it is important to notice that the control input w is replaced with u such that

$$w \;\hat{=}\; 0.05236 \cdot (2u - 1)$$

If w has values within $\{-0.05236, 0.05236\}$, then u lies within the binary set $\{0, 1\}$.

```
wa = 0.05236;
  wa2 = wa^2;
  wa3 = wa^3;
  continuous_dynamics = @(x, u) [ -0.877 * x(1) + x(3) - 0.088 * x(1) * x(3) + 0.47 *␣
↪x(1) * x(1) - 0.019 * x(2) * x(2) - x(1) * x(1) * x(3)...
                                  + 3.846 * x(1) * x(1) * x(1) - 0.215 * wa * (2 *␣
↪u(1) - 1) + 0.28 * x(1) * x(1) * wa * (2 * u(1) - 1) + 0.47 * x(1) * wa2 * (2 *␣
↪u(1) - 1) * (2 * u(1) - 1)...
                                  + 0.63 * wa3 * (2 * u(1) - 1) * (2 * u(1) - 1) * (␣
↪2 * u(1) - 1);
                                  x(3);
                                  -4.208 * x(1) - 0.396 * x(3) - 0.47 * x(1) * x(1) -␣
↪3.564 * x(1) * x(1) * x(1) - 20.967 * wa * (2 * u(1) - 1) + 6.265 * x(1) * x(1) *␣
↪wa * (2 * u(1) -1 )...
                                  + 46.0 * x(1) * wa2 * (2 * u(1) - 1) * (2 * u(1) -␣
↪1) + 61.4 * wa3 * (2 * u(1) - 1) * (2 * u(1) - 1) * (2 * u(1) - 1)];
  model.continuous_dynamics = continuous_dynamics;
  model.E = [zeros(params.dimX, params.dimU), eye(params.dimX)];
```

Inequality constraints

The maneuver is subjected to a set of constraints, involving only the simple bounds:

$$\begin{aligned} 0\,\text{rad} \leq & u \leq 1\,\text{rad} \\ -10\,\text{rad} \leq & x_0 \leq 10\,\text{rad} \\ -10\,\text{rad} \leq & x_1 \leq 10\,\text{rad} \\ -10\,\text{rad/sec} \leq & x_2 \leq 10\,\text{rad/sec} \end{aligned}$$

Initial and final conditions

The goal of the maneuver is to steer the aircraft from an initial condition with nose pointing upwards

$$(0.4655, 0, 0)^T$$

to the origin.

11.18.2 Defining the MPC problem

With the above defined MATLAB functions for objective and equality constraints, we can completely define the MINLP formulation in the next code snippet. For this example, the number of stages has been set to $N = 100$.

In the code snippet below, it is important to notice that the lower and upper bounds are declared as parametric before generating the solver. This needs to be done for generating mixed-integer NLP solvers. Lower and upper bounds are meant to be provided at run-time.

```
Nstages = 100;

params.dimU = 1;
params.dimX = 3;
params.dimZ = params.dimU + params.dimX;

%% Model with stage variable (u', x')'
model.N = Nstages;
model.nvar = params.dimZ;
model.neq = params.dimX;

%% Boundary conditions
model.xinitidx = params.dimU+1:model.nvar;

%% Bounds
model.lb = [];
model.ub = [];
model.lbidx{1} = 1 : params.dimU;
model.ubidx{1} = 1 : params.dimU;
for i = 2 : model.N
    model.lbidx{i} = 1 : model.nvar;
    model.ubidx{i} = 1 : model.nvar;
end

%% Dynamics
wa = 0.05236;
wa2 = wa^2;
wa3 = wa^3;
continuous_dynamics = @(x, u) [ -0.877 * x(1) + x(3) - 0.088 * x(1) * x(3) + 0.47 *␣
↪x(1) * x(1) - 0.019 * x(2) * x(2) - x(1) * x(1) * x(3)...
                                + 3.846 * x(1) * x(1) * x(1) - 0.215 * wa * (2 * u(1)␣
↪- 1) + 0.28 * x(1) * x(1) * wa * (2 * u(1) - 1) + 0.47 * x(1) * wa2 * (2 * u(1) -␣
↪1) * (2 * u(1) - 1)...
                                + 0.63 * wa3 * (2 * u(1) - 1) * (2 * u(1) - 1) * ( 2␣
↪* u(1) - 1);
                                x(3);
                                -4.208 * x(1) - 0.396 * x(3) - 0.47 * x(1) * x(1) - 3.
↪564 * x(1) * x(1) * x(1) - 20.967 * wa * (2 * u(1) - 1) + 6.265 * x(1) * x(1) * wa␣
↪* (2 * u(1) -1 )...
                                + 46.0 * x(1) * wa2 * (2 * u(1) - 1) * (2 * u(1) - 1)␣
↪+ 61.4 * wa3 * (2 * u(1) - 1) * (2 * u(1) - 1) * (2 * u(1) - 1)];
model.continuous_dynamics = continuous_dynamics;
model.E = [zeros(params.dimX, params.dimU), eye(params.dimX)];

%% Objective
model.objective =  @(z) 0.0001 * z(params.dimU)^2;
model.objectiveN = @(z) 150 * z(params.dimU+1)^2 + 5 * z(params.dimU+2)^2 + 5 *␣
↪z(params.dimU+3)^2;
```

(continues on next page)

(continued from previous page)

```
%% Integer indices
for s = 1:model.N
    model.intidx{s} = 1;
end

%% Define outputs
outputs(1) = newOutput('TailAngle', 1:model.N, 1);
outputs(2) = newOutput('AngleAttack', 1:model.N, 2);
outputs(3) = newOutput('PitchAngle', 1:model.N, 3);
outputs(4) = newOutput('PitchAngleRate', 1:model.N, 4);
```

11.18.3 Generating an MINLP solver

We have now populated `model` with the necessary fields to generate a mixed-integer solver for our problem. Now we set some options for our solver and then use the function `FORCES_NLP` to generate a solver for the problem defined by `model` with the initial state and the lower and upper bounds as a parameters:

```
%% Set code-generation options
codeoptions = getOptions('F8aircraft_solver');
codeoptions.printlevel = 0;
codeoptions.maxit = 2000;
codeoptions.timing = 0;
codeoptions.nlp.integrator.type = 'IRK2';
codeoptions.nlp.integrator.Ts = 0.05;
codeoptions.nlp.integrator.nodes = 20;
% Specify maximum number of threads to parallelize minlp search
codeoptions.minlp.max_num_threads = 8;

%% Generate MINLP solver
FORCES_NLP(model, codeoptions, outputs);
```

In the code snippet above, we have set some integrator options, since the continuous-time dynamics has been provided in the model. The branch-and-bound search can be run on several threads in parallel by setting the run-time parameter `numThreadsBnB` equal to the number of threads to be used. The default value is `1`. Moreover, the maximum number of threads for the branch-and-bound search can be set via the option `max_num_threads`. By default, `max_num_threads = 4`.

11.18.4 Calling the generated MINLP solver

Once all parameters have been populated, the MEX interface of the solver can be used to invoke it:

```
%% Set run-time parameters
model.nu = params.dimU;
model.nx = params.dimX;
problem.(sprintf('lb%03d', 1)) = 0;
problem.(sprintf('ub%03d', 1)) = 1;
for s = 2:Nstages-1
    problem.(sprintf('lb%03d', s)) = [0, -1e1 * ones(1, model.nx)]';
    problem.(sprintf('ub%03d', s)) = [1, 1e1 * ones(1, model.nx)]';
```

(continues on next page)

(continued from previous page)

```
end
problem.(sprintf('lb%03d', Nstages)) = [0, -1e1 * ones(1, model.nx)]';
problem.(sprintf('ub%03d', Nstages)) = [1, 1e1 * ones(1, model.nx)]';

problem.x0 = repmat([0; zeros(model.nx, 1)], Nstages, 1);
problem.xinit = zeros(model.nx, 1);
problem.xinit(1) = 0.4655;

%% Call MINLP solver
[sol, exitflag, info] = F8aircraft_solver(problem);
```

11.18.5 Providing an initial guess at run-time

In order to provide an guess for the incumbent, the following code-generation options need to be enabled:

```
codeoptions.minlp.int_guess = 1;
codeoptions.minlp.round_root = 0; % Default value is 1
codeoptions.minlp.int_guess_stage_vars = [1]; % An integer guess is provided for␣
↪variable 1 at every stage
```

Then the incumbent guess can be set at run-time via

```
for s = 1:Nstages
  problem.(sprintf('int_guess%03d', s)) = [0];
end
for s = 1:2
  problem.(sprintf('int_guess%03d', s)) = [1];
end
problem.(sprintf('int_guess%03d', 39)) = [1];
for s = 41:42
  problem.(sprintf('int_guess%03d', s)) = [1];
end
for s = 85:90
  problem.(sprintf('int_guess%03d', s)) = [1];
end
```

11.18.6 Changing the parallelization strategy at run-time

When running the MINLP solver on several threads with `numThreadsBnB >= 1`, the parallelization strategy can be changed via

```
problem.parallelStrategy = 0; % 0 (one shared priority queue, default), 1 (one␣
↪priority queue per thread)
```

11.18.7 Results

The control objective is to drive the angle of attack as close as possible to zero within a five seconds time frame. The control input is the tail deflection angle, which can take values with the set $\{-0.05236, 0.05236\}$ and the initial state is $(0.4655, 0, 0)^T$, where the first component is the angle of attack, the second component is the pitch angle and the third component is the pitch angle rate.

The angle of attack computed by FORCESPRO MINLP solver running on one thread is shown in Figure Figure 11.58 and the input sequence is in Figure Figure 11.59. One can notice the bang-bang behavior of the solution. When running on three threads the FORCESPRO MINLP solver provides a solution with lower final primal objective. Results are shown on Figures Figure 11.60 and Figure 11.61.

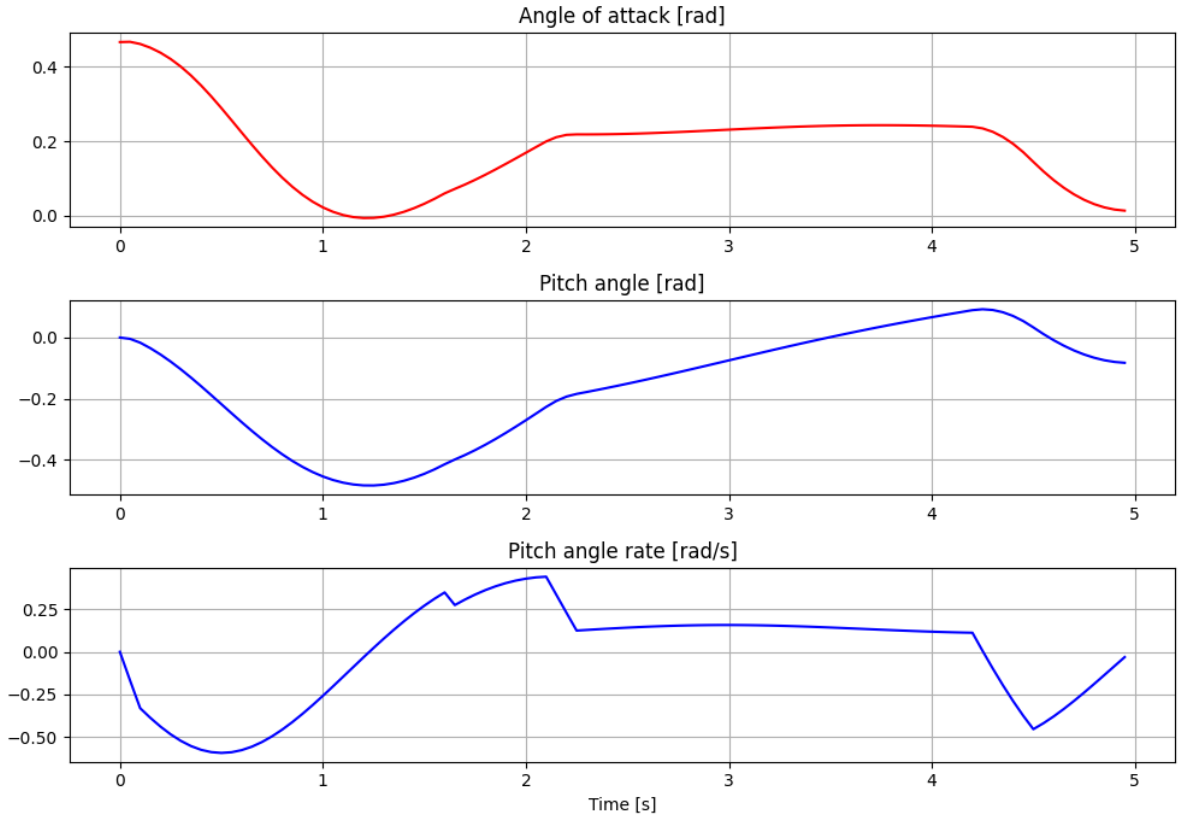


Figure 11.58: Aircraft's angle of attack over time computed with one thread.

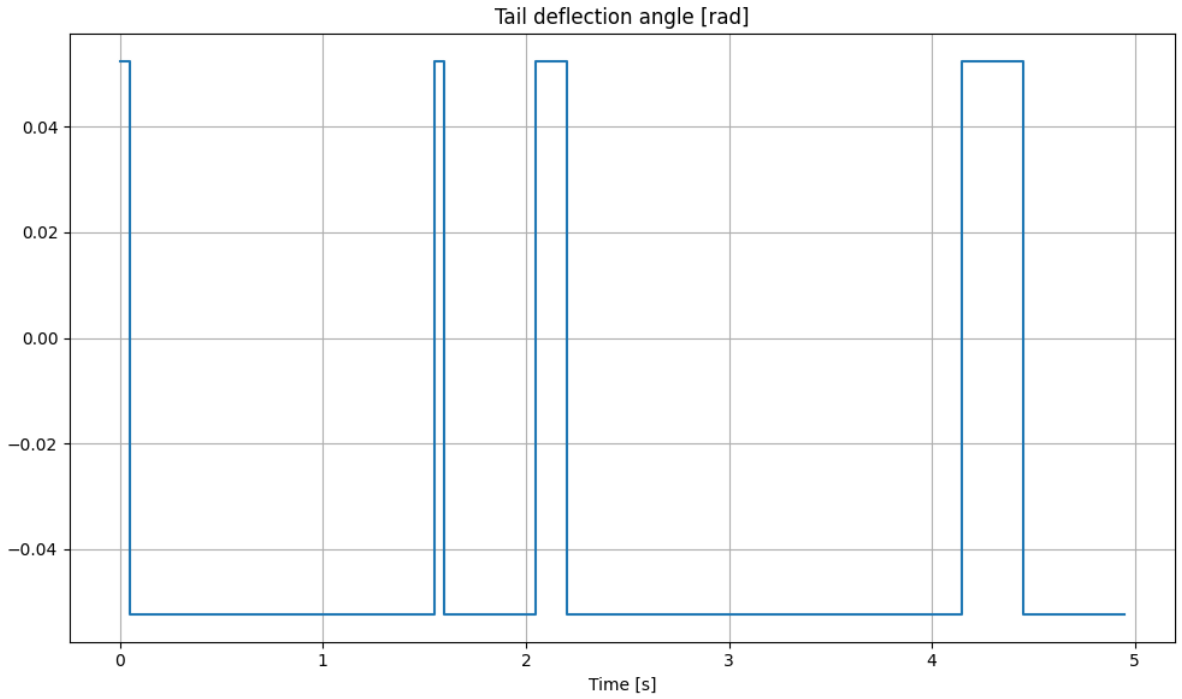


Figure 11.59: Aircraft's tail deflection angle over time with one thread.

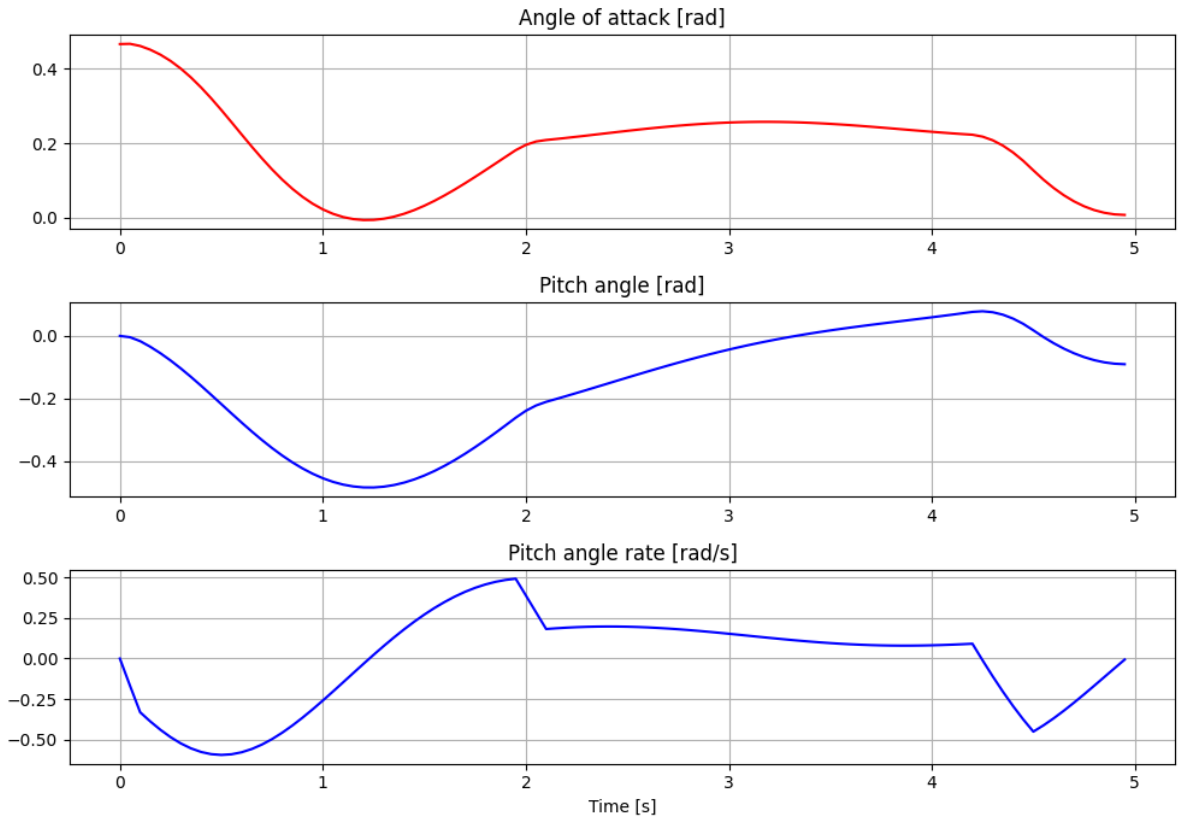


Figure 11.60: Aircraft's angle of attack over time computed with three threads.

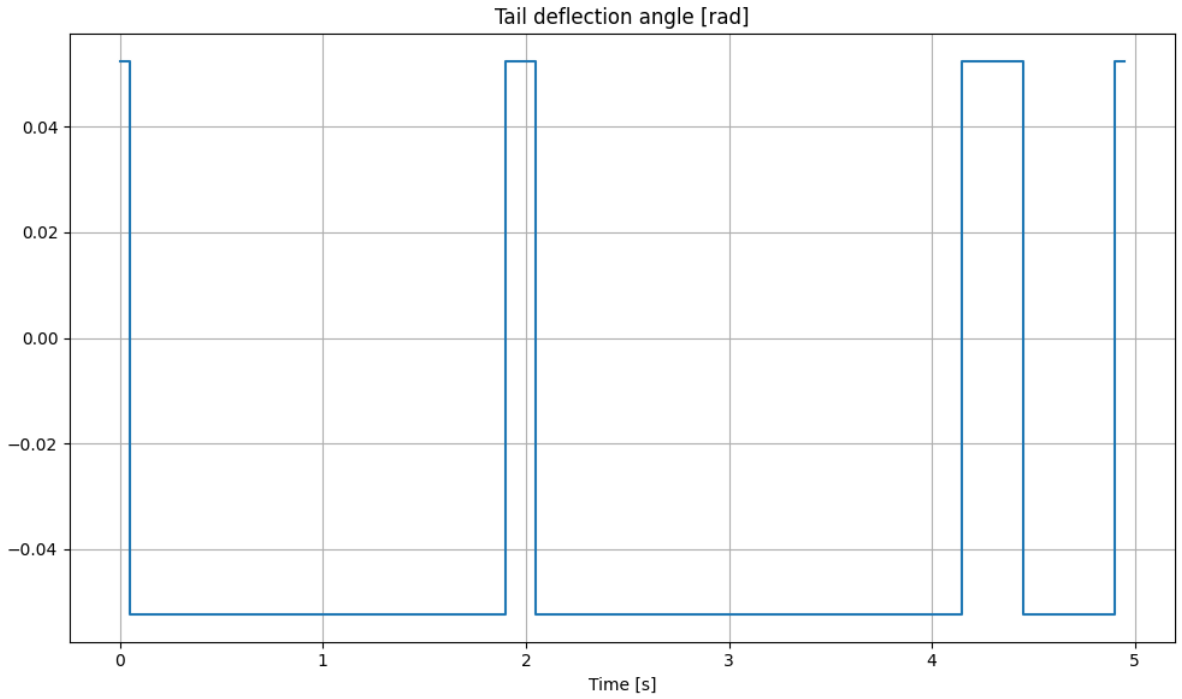


Figure 11.61: Aircraft's tail deflection angle over time with three threads.

11.19 High-level interface: Optimal EV charging and speed profile example (MATLAB & PYTHON)

- *Problem Overview*
  - *Vehicle Dynamics*
  - *Vehicle Energetics*
  - *Bounds*
  - *MPC Formulation*
  - *Model Parameters*
- *Defining the MPC Problem*
  - *Objective function*
  - *Equality constraints*
  - *Inequality constraints*
  - *Generating the FORCESPRO NLP solver*
  - *Calling the solver*
  - *Results*

Note: We can find an updated and more realistic version of this example in Section 11.20. This example is kept in the documentation for legacy purposes.

In this example we illustrate the simplicity of the high-level user interface on an optimal electric vehicle (EV) speed and charging management. The controller plans the car's speed and charging trajectory such that the given route is traversed in the shortest time possible, while simultaneously respecting speed limits as well as vehicle's and battery's technical requirements.

11.19.1 Problem Overview

We consider a single EV and a fixed route with given road slopes (i.e. angles) and speed limits. Optionally, some charging stations could be available at predefined locations along the road. Vehicle operation is defined by longitudinal vehicle dynamics, powertrain efficiency and battery capacity. Due to the spatial characteristic of the problem, the formulation is discretized in space domain. This problem is primarily based on the work in [JiaJibItoGor19] and [JiaJibGor20].

Vehicle Dynamics

Assuming a sample distance L_s, the longitudinal velocity of the vehicle in space domain at the discrete step $(i+1) \in \mathcal{I}$ can be determined using the velocity at the previous step (i) and the difference in kinetic energy between these two consecutive steps, as follows:

$$\frac{1}{2} m_\mathrm{eq} \left(v_{i+1}^2 - v_i^2 \right) = L_\mathrm{s} \left(F_{\mathrm{t},i} - F_{\mathrm{b},i} - F_{\mathrm{r},i} \right)$$

where the resistance force is comprised of the following three components:

$$F_{\mathrm{r},i} = \underbrace{c_{\mathrm{r}} m_{\mathrm{v}} g \cos(\alpha_{\mathrm{i}})}_{\text{rolling resistance}} + \underbrace{m_{\mathrm{v}} g \sin(\alpha_{\mathrm{i}})}_{\text{gravitational force}} + \underbrace{\frac{1}{2} c_{\mathrm{a}} A_{\mathrm{f}} \rho_{\mathrm{a}} v_i^2}_{\text{air drag force}}$$

The equivalent mass $m_{\mathrm{eq}} = m_{\mathrm{v}}(1 + e_{\mathrm{I}})$ includes vehicle mass m_{v} and the effect of all rotational masses captured by the mass factor e_{I}. The road slope is represented by the road angle α. Variables $F_{\mathrm{t}} \geq 0$ and $F_{\mathrm{b}} \geq 0$ represent the traction and braking forces on the wheels, which are generated by the powertrain and mechanical braking system separately (*note: regenerative braking capabilities could be included by allowing the traction power to be negative*). The total driving resistance F_{r} comprises rolling resistance force, gravitational force and air drag force (while neglecting the direct impact of wind speed), with c_{a} being the air drag coefficient, A_{f} the projected frontal area, ρ_{a} the air density, g the gravitational acceleration, and c_{r} the rolling resistance coefficient.

The vehicle's velocity dynamics are thus defined by

$$v_{i+1} = \sqrt{\frac{2\left(F_{\mathrm{t},i} - F_{\mathrm{b},i} - F_{\mathrm{r},i}\right) L_{\mathrm{s}}}{m_{\mathrm{eq}}} + v_i^2}$$

with the assumption that the speed is constant while traversing a single discrete road segment. Note that the expression below the square root must be strictly positive in order for the problem to be feasible, as will be discussed later on. In addition to velocity, the time is also a state variable of the model and can be computed as

$$t_{i+1} = t_i + \frac{L_{\mathrm{s}}}{v_i}$$

Vehicle Energetics

EV's powertrain efficiency $\eta(v, F_{\mathrm{t}}, \zeta)$ is a function of vehicle's speed, traction force and battery's state-of-charge (SoC), as shown by the efficiency maps depicted in Figure 11.62. Nevertheless, the impact of SoC on powertrain efficiency is negligible as SoC does not change dramatically within a short time period if the battery capacity is sufficiently large. Therefore, one could generate several 2D maps of the form $\eta(v, F_{\mathrm{t}})$, pre-calculated at different SoC levels such that the most accurate one can be selected according to the current SoC.

For the purpose of this example, we have have generated a single 2D efficiency map depicted in Figure 11.63 used across all SoC levels. However, for simplicity, the implementation in this example is confined to 1D efficiency map $\eta(F_{\mathrm{t}})$ extracted for $v = 15\,\mathrm{m/s}$.

The 1D efficiency map $\eta(F_{\mathrm{t}})$ can be computed in the form of a cubic spline function as follows:

Matlab

Python

```
function [ etaMotorSpline ] = setupMotorEfficiency(FtMax)
% Returns cubic spline representing motor efficiency as a function of
% traction force.
%
% The data is taken from "Jia, Y.; Jibrin, R.; Goerges, D.: Energy-Optimal
% Adaptive Cruise Control for Electric Vehicles Based on Linear and
% Nonlinear Model Predictive Control. In: IEEE Transactions on Vehicular
% Technology, vol. 69, no. 12, pp. 14173-14187, Dec. 2020."
    FtSample = 0:0.1*FtMax:FtMax;
    etaSample = [0.835, 0.865, 0.9, 0.85, 0.79, 0.75, 0.72, 0.72, 0.72, 0.72, 0.72];

    etaMotorSpline = ForcesInterpolationFit(FtSample,etaSample);
end
```

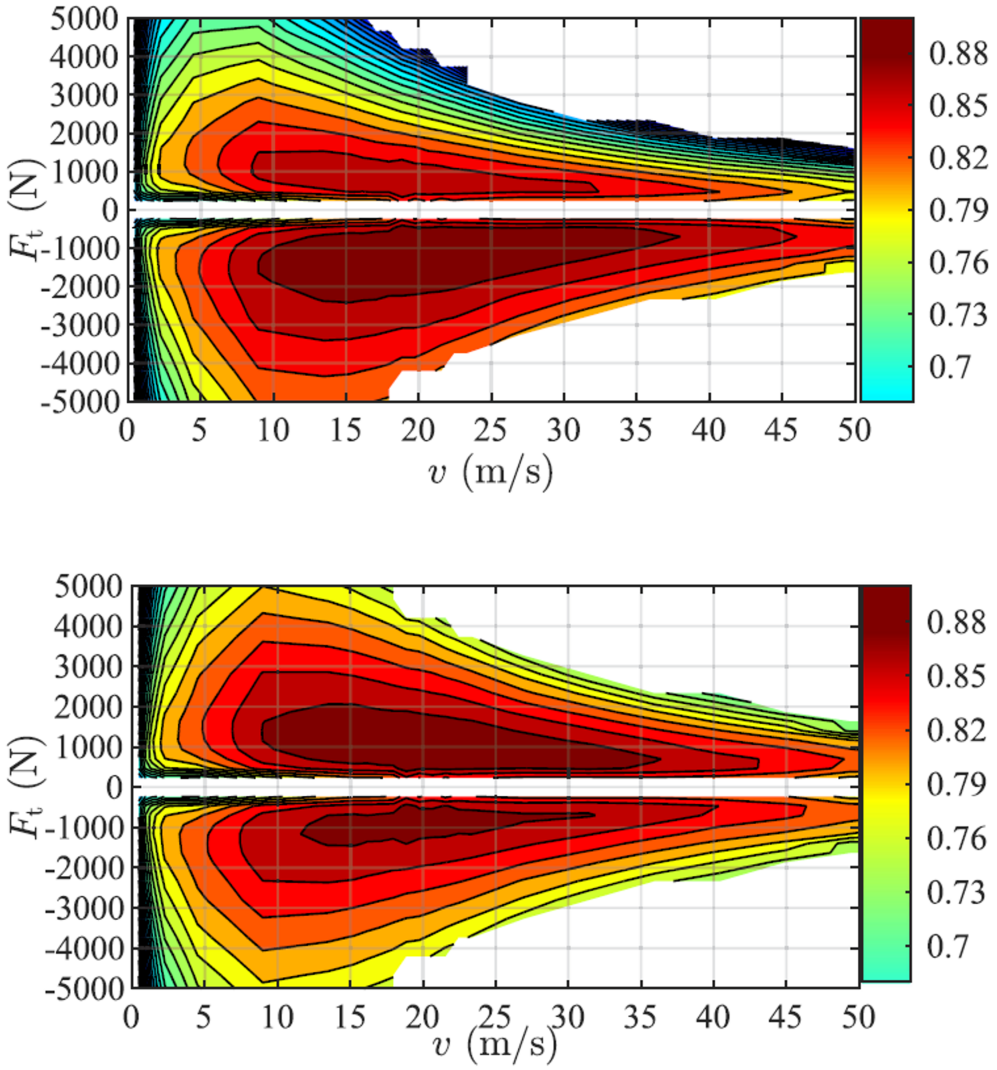


Figure 11.62: 2D efficiency maps of the EV's powertrain with two different SoC levels taken from [JiaJibGor20]: (top) η with SoC $\zeta = 10\,\%$; (bottom) η with SoC $\zeta = 90\,\%$.

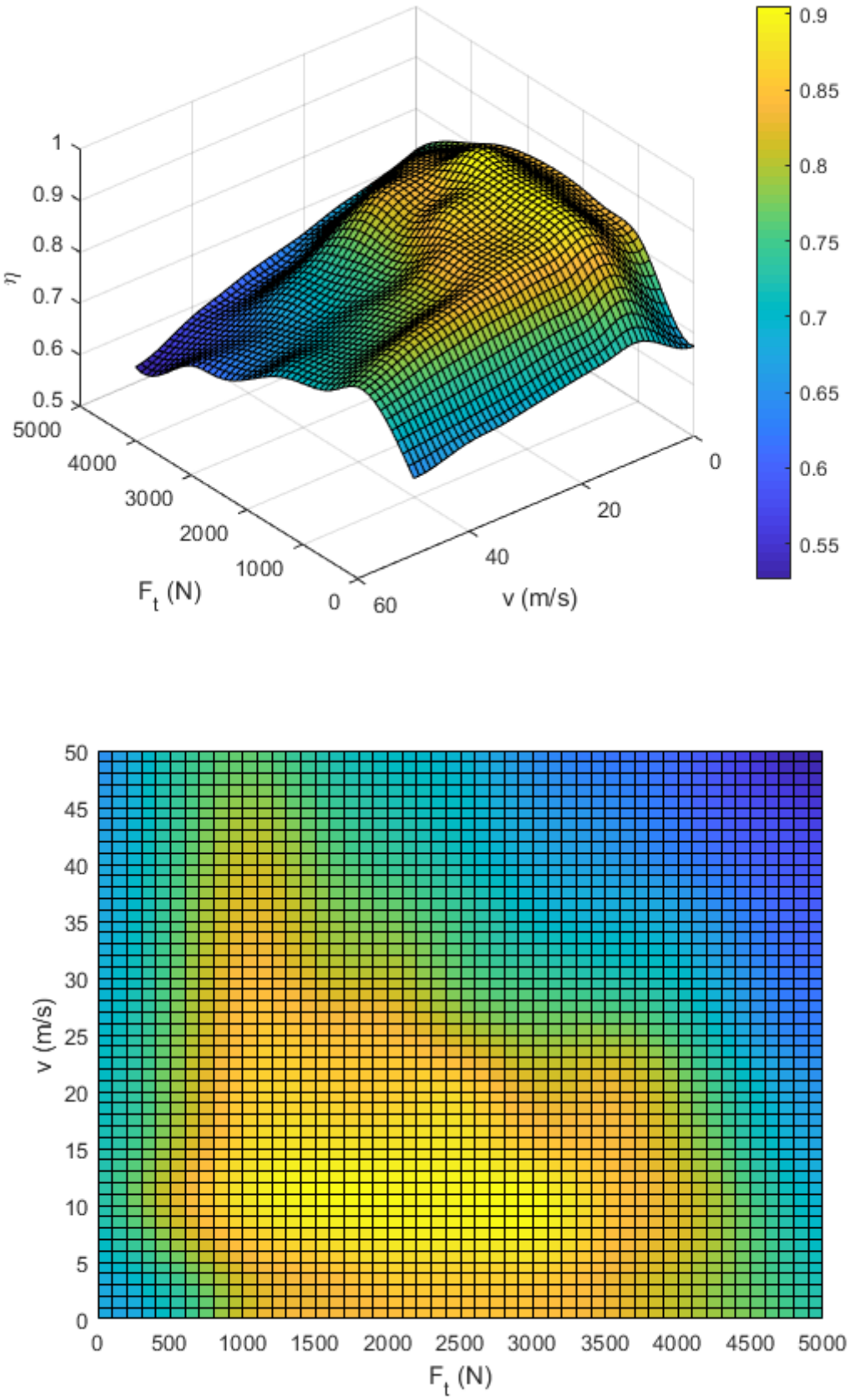


Figure 11.63: Generated 2D EV efficiency map. Note that we are using a simplified 1D efficiency map $\eta(F_\mathrm{t})$ taken for $v = 15\,\mathrm{m/s}$ in this example.

```
def setupMotorEfficiency(self):
    """
    Returns cubic spline representing motor efficiency as a function of traction␣
↪force.

    The data is taken from "Jia, Y.; Jibrin, R.; Goerges, D.: Energy-Optimal Adaptive␣
↪Cruise Control for Electric Vehicles Based on Linear and Nonlinear Model Predictive␣
↪Control. In: IEEE Transactions on Vehicular Technology, vol. 69, no. 12, pp. 14173-
↪14187, Dec. 2020."
    """
    FtSample = np.linspace(0, self.FtMax, 11)
    etaSample = np.array([0.835, 0.865, 0.9, 0.85, 0.79, 0.75, 0.72, 0.72, 0.72, 0.72,
↪ 0.72])
    etaMotorSpline = forcespro.modelling.InterpolationFit(FtSample, etaSample)

    return etaMotorSpline
```

Assuming a set of predefined charging stops at discrete spatial steps $k \in \mathcal{K} \subset \mathcal{I}$, we can include either **optional** or **mandatory** charging at each of those stops (to be defined by the user). Given the implemented powertrain efficiency $\eta_i(F_{\mathrm{t},i})$ and potential charging decisions, the driving energy depletion and charging instances are incorporated into the SoC dynamics at discrete step i as:

$$\zeta_{i+1} = \zeta_i - \frac{F_{\mathrm{t},i} L_{\mathrm{s}}}{\eta_i E_{\mathrm{cap}}} + \frac{P_{\mathrm{ch},i}}{E_{\mathrm{cap}}} \Delta t_i$$

where E_{cap} denotes the battery's energy capacity, $P_{\mathrm{ch},i}(\zeta_i)$ is a vehicle's charging power dependent on the SoC, and Δt_i is a total charging time spent at location $i \in \mathcal{K}$. Note that SoC ζ is normalized on an interval $[0, 1]$.

In this example we will particularly focus on the performance of a BMW i3. The charging profiles $P_{\mathrm{ch}}(\zeta)$ for three production models with different battery capacities are depicted in Figure 11.64 [Fastned], and can be approximated by a piecewise-affine function shown in Figure 11.65.

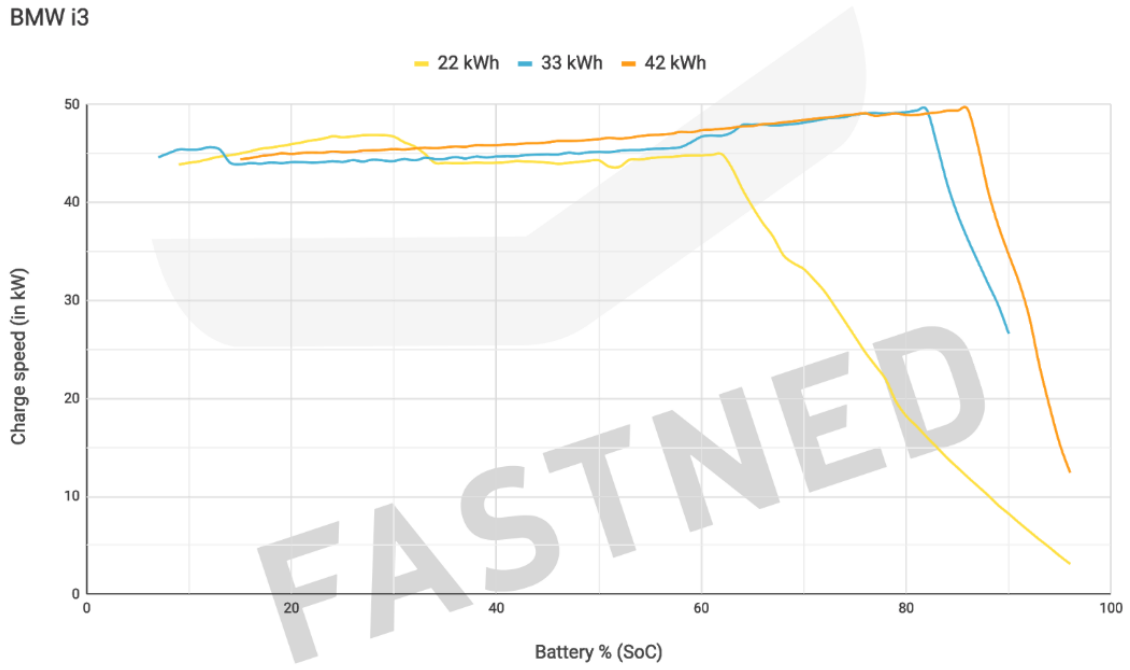


Figure 11.64: Actual charging profile of BMW i3 as a function of vehicle's SoC.

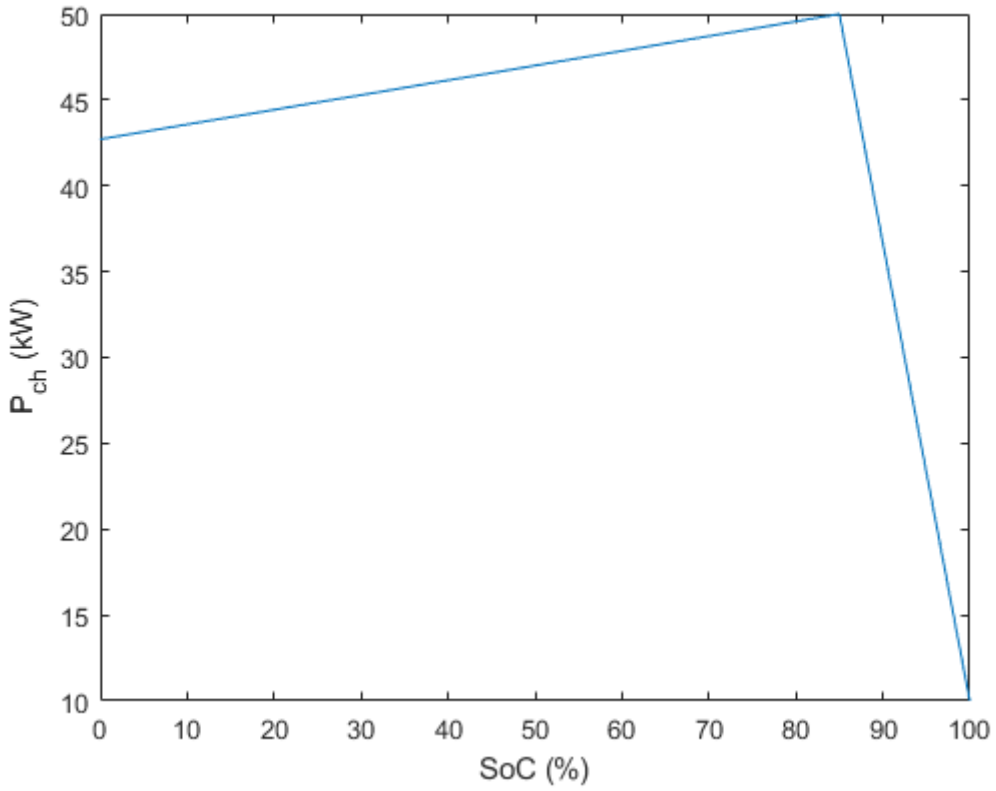


Figure 11.65: Implemented piecewise-affine charging profile for the $42\,\mathrm{kWh}$ vehicle as a function of vehicle's SoC.

The code for computing the $P_{\mathrm{ch}}(\zeta)$ spline function is given below. Note that the Python implementation uses a shape-preserving piecewise-cubic representation instead of piecewise-linear one.

Matlab

Python

```
function [ Pch ] = setupChargingPower(PchMax)
% Returns piecewise linear representation of charging power as a
% function of vehicle's SoC.
%
% The data is taken from Fastned charging chart available at
% https://support.fastned.nl/hc/en-gb/articles/204784718-Charging-with-a-BMW-i3
%
% NOTE: Python example uses shape-preserving piecewise cubic spline
% approximation ('pchip') which might lead to negligible result
% discrepancies between the two clients when directly compared.
    socSample = [0.15, 0.85, 1.0];
    PchSample = [0.88, 1, 0.2]*PchMax;
    Pch = ForcesInterpolationFit(socSample,PchSample,'linear');
end
```

```
def setupChargingPower(self):
    """
    Returns shape-preserving piecewise cubic spline representation of charging power␣
↪as a function of SoC.
```

(continues on next page)

(continued from previous page)

```
    The data is taken from Fastned charging chart available at https://support.
↪fastned.nl/hc/en-gb/articles/204784718-Charging-with-a-BMW-i3

    NOTE: MATLAB example uses piecewise linear approximation which might lead to␣
↪negligible result discrepancies between the two clients when directly compared.
    """
    socSample = np.array([0.15, 0.85, 1.0])
    PchSample = np.array([0.88, 1, 0.2]) * self.PchMax
    PchSpline = forcespro.modelling.InterpolationFit(socSample, PchSample, 'pchip')

    return PchSpline
```

Clearly, the time dynamics would have to be modified as well:

$$t_{i+1} = t_i + \frac{L_\mathrm{s}}{v_i} + \Delta t_i$$

with $\Delta t_i \approx 0$ for all non-charging spatial steps $i \notin \mathcal{K}$, and $v_i \approx \underline{v}$ for all charging spatial steps $i \in \mathcal{K}$, with $\underline{v} > 0$ being th lower speed bound. The latter is not set to zero for two reasons: 1) it would lead to infeasibility due to v_i being in the denominator; 2) it reflects the average speed across the distance L_s, which includes both the driving and the charging instances, and is therefore greater than zero. This implies that the final vehicle model comprises three states $x = [v, t, \zeta]^\top$ and three control inputs $u = [F_\mathrm{t}, F_\mathrm{b}, \Delta t]^\top$, with the model dynamics described by

$$\begin{aligned}
v_{i+1} &= \sqrt{\frac{2\left(F_{\mathrm{t},i} - F_{\mathrm{b},i} - F_{\mathrm{r},i}\right) L_\mathrm{s}}{m_\mathrm{eq}} + v_i^2} \\
t_{i+1} &= t_i + \frac{L_\mathrm{s}}{v_i} + \Delta t_i \\
\zeta_{i+1} &= \zeta_i - \frac{F_{\mathrm{t},i} L_\mathrm{s}}{\eta_i E_\mathrm{cap}} + \frac{P_{\mathrm{ch},i}}{E_\mathrm{cap}} \Delta t_i
\end{aligned}$$

Additionally, we will include a slack variable $s \geq 0$ into control vector u, as it is used to relax the upper and lower bounds on vehicle's SoC.

Bounds

In terms of bounds on state variables, we impose upper and lower limits on vehicle's velocity and SoC, dictated by the road speed limits and battery specifications:

$$\begin{aligned}
\underline{v} \leq v_i \leq \bar{v}_i \\
\underline{\zeta} - s_i \leq \zeta_i \leq \bar{\zeta} + s_i
\end{aligned}$$

The upper speed bound $\bar{v}_i$ is stage-dependent, dictated by the spatial discretization of the road speed limits, whereas the lower speed bound $\underline{v}$ is kept constant across all stages. For charging instances $i \in \mathcal{K}$, the upper speed bound can optionally be set to $\bar{v}_i \approx \underline{v}$, indicating that the vehicle has to reduce speed. The SoC limits $\underline{\zeta}$ and $\bar{\zeta}$ are also stage independent and implemented as soft bounds.

All control variables are nonnegative and upper bounded with appropriate physical and technical limits:

$$\begin{aligned}
0 \leq \Delta t_i \leq \overline{\Delta t}_i \\
0 \leq F_{\mathrm{t},i} \leq \overline{F}_{\mathrm{t},i} \\
0 \leq F_{\mathrm{b},i} \leq \overline{F}_\mathrm{b}
\end{aligned}$$

Here, $\overline{\Delta t}_i$ denotes maximum permissible charging time and is set to $\overline{\Delta t}_i \approx 0, \forall i \notin \mathcal{K}$, whereas $\overline{F}_{\mathrm{t},i}$ and $\overline{F}_\mathrm{b}$ are vehicle's physical upper bounds on maximum traction and breaking force,

respectively. Note that the traction force limit is comprised of the constant friction limit $\overline{F}_{\mathrm{t}}$ and the hyperbolic traction function of vehicle's velocity $\overline{F}_{\mathrm{t},i}^{\mathrm{hyp}}(v_i)$, i.e.

$$\overline{F}_{\mathrm{t},i} = \min\left(\overline{F}_{\mathrm{t}}, \overline{F}_{\mathrm{t},i}^{\mathrm{hyp}}(v_i)\right)$$

as depicted in Figure 11.66. The actual function implemented in this example is showcased in Figure 11.67.

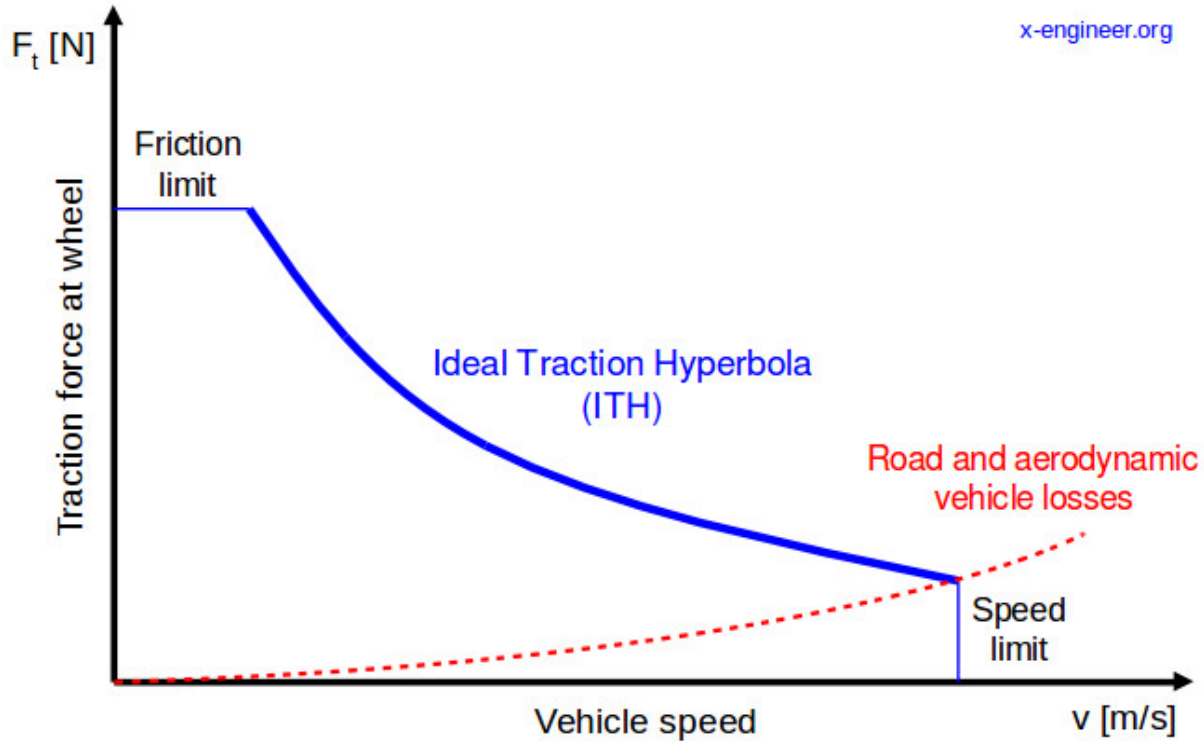


Figure 11.66: Maximum permissible traction force as a function of vehicle's velocity taken from [xEngineer].

The computation of ideal traction hyperbola $\overline{F}_{\mathrm{t,hyp}}(v)$ in the cubic spline form is provided below.

Matlab

Python

```
function [ FtMaxHypSpline ] = setupTractionForceHyperbola(vMax,FtMax)
% Returns cubic spline representing maximum traction force as a function of
% vehicle's speed
%
% The data is generated based on the technical article provided at
% https://x-engineer.org/need-gears/
    vSample = [0.25, 0.4, 0.6, 0.8, 1.0]*vMax;
    FtMaxHypSample = [1.0, 0.67, 0.43, 0.32, 0.28]*FtMax;
    FtMaxHypSpline = ForcesInterpolationFit(vSample,FtMaxHypSample);
end
```

```
def setupTractionForceHyperbola(self):
    """
    Returns cubic spline representing motor efficiency as a function of traction␣
↪force.

```

(continues on next page)

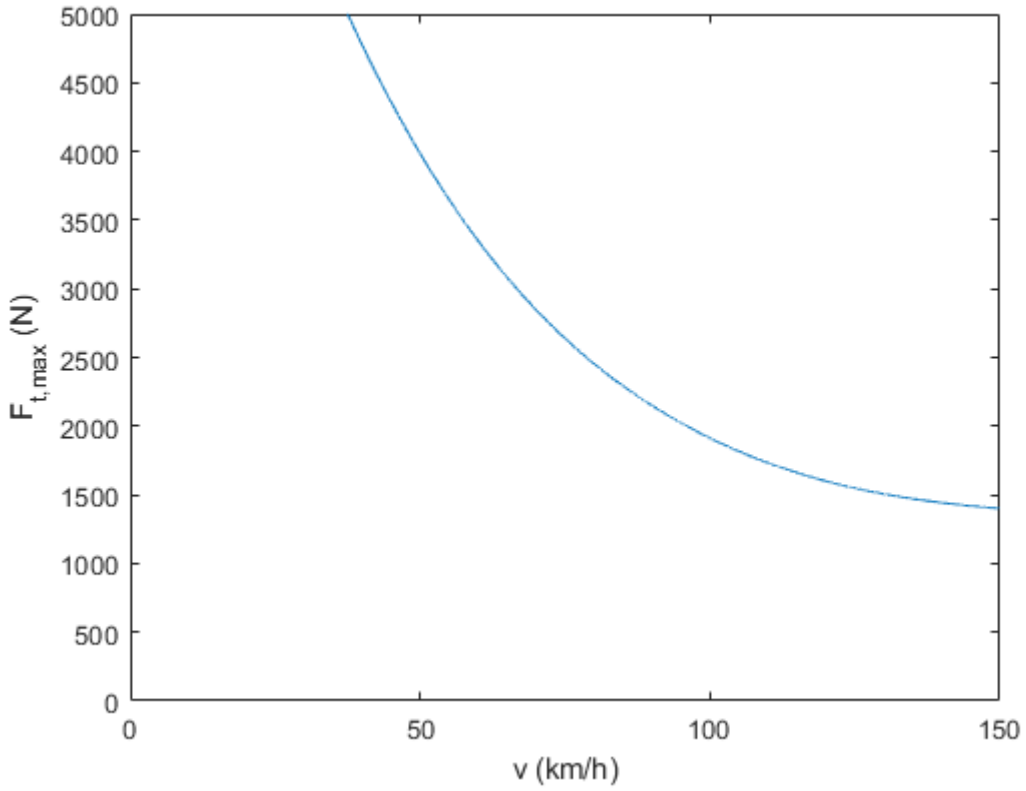


Figure 11.67: Implemented traction force hyperbola as a function of vehicle's velocity.

(continued from previous page)

```
    The data is generated based on the technical article provided at https://x-
↪engineer.org/need-gears/
    """
    vSample = np.array([0.25, 0.4, 0.6, 0.8, 1.0]) * self.kmh2ms(self.vMax)
    FtMaxHypSample = np.array([1.0, 0.67, 0.43, 0.32, 0.28]) * self.FtMax
    FtMaxHypSpline = forcespro.modelling.InterpolationFit(vSample, FtMaxHypSample)

    return FtMaxHypSpline
```

MPC Formulation

A complete MPC problem formulation can thus be described as follows:

$$
\begin{aligned}
\text{minimize} \quad & t_N + \sum_{i=0}^{N-1} \left(\omega_\mathrm{s} s_i + \omega_\mathrm{e} F_{\mathrm{t},i}^2 + \omega_\mathrm{b} F_{\mathrm{b},i}^2 \right) \\
\text{subject to} \quad & v_{i+1} = \sqrt{\frac{2L_\mathrm{s}}{m_\mathrm{eq}} \left(F_{\mathrm{t},i} - F_{\mathrm{b},i} - F_{\mathrm{r},i} \right) + v_i^2} \\
& t_{i+1} = t_i + L_\mathrm{s} v_i^{-1} + \Delta t_i \\
& \zeta_{i+1} = \zeta_i - \frac{L_\mathrm{s}}{\eta_i E_\mathrm{cap}} F_{\mathrm{t},i} + \frac{P_{\mathrm{ch},i}}{E_\mathrm{cap}} \Delta t_i \\
& \underline{v} \leq v_i \leq \bar{v}_i \\
& \underline{\zeta} - s_i \leq \zeta_i \leq \bar{\zeta} + s_i \\
& 0 \leq \Delta t_i \leq \overline{\Delta t}_i \\
& 0 \leq F_{\mathrm{t},i} \leq \overline{F}_{\mathrm{t},i} \\
& 0 \leq F_{\mathrm{b},i} \leq \overline{F}_\mathrm{b} \\
& 0 \leq s_i \\
& x_0 = x_\mathrm{init}
\end{aligned}
$$

with the objective function explained in detail in the subsequent sections.

Model Parameters

As stated previously, in this example we focus on a highest energy capacity model of BMW i3, defined by the following parameters obtained from [BMWi3]:

$$
\begin{aligned}
m_\mathrm{v} &= 1345\,\mathrm{kg} \\
e_\mathrm{I} &= 1.06 \\
A_\mathrm{f} &= 2.38\,\mathrm{m}^2 \\
\rho_\mathrm{a} &= 1.206\,\mathrm{kg/m}^3 \\
c_\mathrm{a} &= 0.29 \\
c_\mathrm{r} &= 0.01 \\
\overline{F}_\mathrm{t} &= 5\,\mathrm{kN} \\
\overline{F}_\mathrm{b} &= 10\,\mathrm{kN} \\
E_\mathrm{cap} &= 37.9\,\mathrm{kWh} \\
\bar{P}_\mathrm{ch} &= 50\,\mathrm{kW} \\
\underline{v} &= 30\,\mathrm{km/h} \\
\bar{v} &= 150\,\mathrm{km/h} \\
\underline{\zeta} &= 0.1 \\
\bar{\zeta} &= 0.9
\end{aligned}
$$

Matlab

Python

```
function [ params ] = defineVehicleParameters(trip)
% Returns vehicle parameters
    % Constant vehicle parameters (not to be changed)
```

(continues on next page)

(continued from previous page)

```
    % The data is obtained from "Technical specifications of the BMW i3
    % (120 Ah)" valid from 11/2018 and available at
    % https://www.press.bmwgroup.com/global/article/detail/T0285608EN/

    % Gravitational acceleration (m/s^2)
    params.g = 9.81;
    % Vehicle mass (kg)
    params.mv = 1345;
    % Mass factor (-)
    params.eI = 1.06;
    % Equivalent mass (kg)
    params.meq = (1+params.eI)*params.mv;
    % Projected frontal area (m^2)
    params.Af = 2.38;
    % Air density (kg/m^3)
    params.rhoa = 1.206;
    % Air drag coefficient (-)
    params.ca = 0.29;
    % Rolling resistance coefficient (-)
    params.cr = 0.01;
    % Maximum traction force (N)
    params.FtMax = 5e3;
    % Minimum traction force (N)
    params.FtMin = 0;
    % param.FtMin = -1.11e3;
    % Maximum breaking force (N)
    params.FbMax = 10e3;
    % Battery's energy capacity (Wh)
    % BMW i3 60Ah - 18.2kWh; BMW i3 94Ah - 27.2kWh; BMW i3 120Ah - 37.9kWh
    params.Ecap = 37.9e3;
    % Maximum permissible charging time (s)
    params.TchMax = 3600;
    % Charging power (W)
    % 7.4 kW on-board charger on IEC Combo AC, optional 50 kW Combo DC
    params.PchMax = 50e3;
    % Power dissipation factor (-) (adjusted for Munich - Cologne trip in
    % order to achieve realistic BMW i3 range of ~260km)
    if trip.type == 1
        params.pf = 1;
    else
        params.pf = 0.4;
    end
    % Minimum vehicle velocity (km/h)
    params.vMin = 30;
    % Maximum vehicle velocity (km/h)
    params.vMax = 150;
    % Minimum SoC (-)
    params.SoCmin = 0.1;
    % Maximum SoC (-)
    params.SoCmax = 0.9;

    % Variable vehicle parameters
    % Motor efficiency (-)
    params.eta = setupMotorEfficiency(params.FtMax);
    % Charging power (W)
    params.Pch = setupChargingPower(params.PchMax);
```

(continues on next page)

(continued from previous page)

```
    % Traction force hyperbola (upper limit) (N)
    params.FtMaxHyp = setupTractionForceHyperbola(kmh2ms(params.vMax), params.FtMax);
end
```

```
class VehicleParameters:
"""Constant and variable vehicle parameters (not to be changed)"""
def __init__(self, trip):
    # Constant vehicle parameters
    # The data is obtained from "Technical specifications of the BMW i3 (120 Ah)"␣
↪valid from 11/2018 and available at https://www.press.bmwgroup.com/global/article/
↪detail/T0285608EN/
    # ----------------------------

    # Gravitational acceleration (m/s^2)
    self.g = 9.81
    # Vehicle mass (kg)
    self.mv = 1345
    # Mass factor (-)
    self.eI = 1.06
    # Equivalent mass (kg)
    self.meq = (1 + self.eI) * self.mv
    # Projected frontal area (m^2)
    self.Af = 2.38
    # Air density (kg/m^3)
    self.rhoa = 1.206
    # Air drag coefficient (-)
    self.ca = 0.29
    # Rolling resistance coefficient (-)
    self.cr = 0.01
    # Maximum traction force (N)
    self.FtMax = 5e3
    # Minimum traction force (N)
    self.FtMin = 0
    # self.FtMin = -1.11e3
    # Maximum breaking force (N)
    self.FbMax = 10e3
    # Battery's energy capacity (Wh)
    # BMW i3 60Ah - 18.2kWh BMW i3 94Ah - 27.2kWh BMW i3 120Ah - 37.9kWh
    self.Ecap = 37.9e3
    # Maximum permissible charging time (s)
    self.TchMax = 3600
    # Charging power (W)
    # 7.4 kW on-board charger on IEC Combo AC, optional 50 kW Combo DC
    self.PchMax = 50e3
    # Power dissipation factor (-) (adjusted for Munich - Cologne trip in order to␣
↪achieve realistic BMW i3 range of ~260km)
    if trip.type == 1:
        self.pf = 1
    else:
        self.pf = 0.4
    # Minimum vehicle velocity (km/h)
    self.vMin = 30
    # Maximum vehicle velocity (km/h)
    self.vMax = 150
    # Minimum SoC (-)
    self.SoCmin = 0.1
```

(continues on next page)

(continued from previous page)

```
    # Maximum SoC (-)
    self.SoCmax = 0.9

    # Variable vehicle parameters
    # ---------------------------

    # Motor efficiency (-)
    self.eta = self.setupMotorEfficiency()
    # Charging power (W)
    self.Pch = self.setupChargingPower()
    # Traction force hyperbola (upper limit) (N)
    self.FtMaxHyp = self.setupTractionForceHyperbola()
```

Trip parameters are defined by the user. Here we focus on two benchmark studies: 1) a $50\,\mathrm{km}$ long trip with five EV charging stations and low initial SoC; and 2) an actual $573\,\mathrm{km}$ long Munich - Cologne trip with four EV charging stations and fully charged vehicle.

Matlab

Python

```
%% Trip parameters (provided by the user)
% Note: Accuracy and feasibility of the solution are not guaranteed when
% the `trip.reduceSpeed` flag is disabled due to inherent mixed-integer
% nature of the problem!

% Trip type (0 - long (Munich - Cologne); 1 - short)
trip.type = 1;

switch trip.type
    case 1
        % Spatial discretization step (km)
        trip.Ls = 1;
        % Trip distance (km)
        trip.dist = 50;
        % EV charging station locations (km)
        trip.kmPosCh = [8, 18, 30, 40, 45];
        % Force vehicle to reduce speed at charging locations (1 - yes; 0 - no)
        trip.reduceSpeed = 0;
        % Initial vehicle speed (km/h)
        trip.initSpeed = 30;
        % Initial SoC [0, 1]
        trip.initSoC = 0.25;
    case 0
        % Spatial discretization step (km)
        trip.Ls = 1;
        % Trip distance (km)
        trip.dist = 573;
        % EV charging station locations (km)
        trip.kmPosCh = [110, 150, 250, 375];
        % Force vehicle to reduce speed at charging locations (1 - yes; 0 - no)
        trip.reduceSpeed = 1;
        % Initial vehicle speed (km/h)
        trip.initSpeed = 30;
        % Initial SoC [0, 1]
        trip.initSoC = 0.75;
end
```

```
class TripParameters:
"""Trip parameters (to be defined by the user)"""
def __init__(self):
    # Note: Accuracy and feasibility of the solution are not guaranteed when the␣
↪`trip.reduceSpeed` flag is disabled due to inherent mixed-integer nature of the␣
↪problem!

    # Trip type (0 - long (Munich - Cologne) 1 - short)
    self.type = 1

    assert self.type in [0, 1], 'Incorrect input: trip type should be 0 or 1.'

    if self.type == 1:
        # Spatial discretization step (km)
        self.Ls = 1
        # Trip distance (km)
        self.dist = 50
        # EV charging station locations (km)
        self.kmPosCh = np.array([8, 18, 30, 40, 45])
        # Force vehicle to reduce speed at charging locations
        self.reduceSpeed = False
        # Initial vehicle speed (km/h)
        self.initSpeed = 30
        # Initial SoC [0, 1]
        self.initSoC = 0.35
    else:
        # Spatial discretization step (km)
        self.Ls = 1
        # Trip distance (km)
        self.dist = 573
        # EV charging station locations (km)
        self.kmPosCh = np.array([110, 150, 250, 375])
        # Force vehicle to reduce speed at charging locations
        self.reduceSpeed = True
        # Initial vehicle speed (km/h)
        self.initSpeed = 30
        # Initial SoC [0, 1]
        self.initSoC = 0.9
```

You can find the code of this example to try it out for yourself in the **examples** folder that comes with your client.

11.19.2 Defining the MPC Problem

Objective function

The primary goal is to minimize the total driving time, i.e. minimize terminal variable t_N at the last stage $i = N$:

$$f(z, p_N) = z(6)$$

In addition, we want to minimize the stage cost associated with slack variable $s, \forall i \in \mathcal{I}$:

$$f(z, p_i) = \omega_s z(1)$$

where ω_s is an appropriately high weight factor.

Optionally, one could opt for ensuring comfortable driving operation and pursuing energy economy improvements. In such case, additional terms could be included in the stage cost function, e.g.

$$f(z, p_i) = \omega_s z(1) + \omega_e z(2)^2 + \omega_b z(3)^2$$

The second term imposes a penalty on consumed energy during driving by reducing the applied traction force, whereas the third term minimizes the total braking force. The corresponding weight factors ω_e and ω_b provide a trade-off between different objectives. Furthermore, one could also penalize jerking, i.e. a change in the traction force between consecutive steps, using another slack variable.

The cost functions are coded in MATLAB and Python as follows:

Matlab

Python

```
% Assume variable ordering zi = [u{i}; x{i}] for i=1...N
% zi = [slack(i); Ft(i); Fb(i); deltaTch(i); v(i); t(i); SoC(i)]
% pi = [vMax(i); vMin(i); alpha(i); TchMax(i); Ls(k)] #k - iteration No.

% Objective function
R = [ 1e-7,  0;
      0,    1e-6];
slackCostFactor = 1e6;

model.objective = @(z) slackCostFactor*z(1) + [z(2); z(3)]'*R*[z(2); z(3)];
model.objectiveN = @(z) z(6);
```

```
"""
Assume variable ordering zi = [u{i}, x{i}] for i=1...N
zi = [slack{i}, Ft{i}, Fb{i}, deltaTch{i}, v{i}, t{i}, SoC{i}]
pi = [vMax{i}, vMin{i}, alpha{i}, TchMax{i}, Ls{k}] #k - iteration No.
"""

# Objective function
R = np.diag([1e-7, 1e-6])
slackCostFactor = 1e6

model.objectiveN = (lambda z: z[5])
model.objective = (lambda z: slackCostFactor * z[0])
# model.objective = (lambda z: slackCostFactor * z[0] + casadi.horzcat(z[1], z[2]) @␣
↪R @ casadi.vertcat(z[1], z[2]))
```

Equality constraints

The equality constraints *model.eq* in this example results from the vehicle's dynamics and energetics model given above, and is implemented as follows:

Matlab

Python

```
% Problem dimensions
nx = 3;
nu = 4;
np = 5;
nh = 6;
```

(continues on next page)

(continued from previous page)

```
model.N = round(trip.dist/trip.Ls); % horizon length
if mod(model.N,1)~=0
    error('Incorrect input: trip distance should be an exact multiple of the spatial␣
↪discretization step.');
end
model.nvar = nx + nu;                % number of variables
model.neq  = nx;                     % number of equality constraints
model.nh   = nh;                     % number of inequality constraints
model.npar = np;                     % number of runtime parameters

% Dynamics, i.e. equality constraints
model.eq = @(z,p) [sqrt(ForcesMax(2*p(5)/param.meq*(z(2) - z(3) - param.cr*param.
↪mv*param.g*cos(p(3)) - param.mv*param.g*sin(p(3)) - 0.5*param.ca*param.Af*param.
↪rhoa*z(5)^2) + z(5)^2, 0));
                   z(6) + p(5)/z(5) + z(4);
                   z(7) - param.pf*p(5)*z(2)/(3600*param.eta(z(2))*param.Ecap) +␣
↪param.Pch(z(7))*z(4)/(3600*param.Ecap)];

model.E = [zeros(nx,nu),eye(nx)];

% Initial and final conditions
model.xinitidx = nu+1:nu+nx;
```

```
# Problem dimensions
nx = 3
nu = 4
npar = 5
nh = 6

model = forcespro.nlp.SymbolicModel()
model.N = round(trip.dist / trip.Ls); # horizon length
assert model.N % 1 == 0, 'Incorrect input: trip distance should be an exact multiple␣
↪of the spatial discretization step'
model.nvar = nx + nu;                # number of variables
model.neq  = nx;                     # number of equality constraints
model.nh   = nh;                     # number of inequality constraints
model.npar = npar;                     # number of runtime parameters

# Objective function
R = np.diag([1e-7, 1e-6])
slackCostFactor = 1e6
# model.objective = (lambda z: casadi.horzcat(z[1], z[2]) @ R @ casadi.vertcat(z[1],␣
↪z[2]))
model.objective = (lambda z: slackCostFactor * z[0])
model.objectiveN = (lambda z: z[5])

# Dynamics, i.e. equality constraints
model.eq = lambda z, p: casadi.vertcat( np.sqrt(forcespro.modelling.smooth_max(2 *␣
↪p[4] /
                                        param.meq * (z[1] - z[2] - param.cr * param.
↪mv * param.g * np.cos(p[2]) - param.mv * param.g * np.sin(p[2]) - 0.5 * param.ca *␣
↪param.Af * param.rhoa * z[4]**2) + z[4]**2, 0)),
                                        z[5] + p[4] / z[4] + z[3],
                                        z[6] - param.pf * p[4] * z[1] / (3600 * param.
↪eta(z[1])* param.Ecap) + param.Pch(z[6]) * z[3] / (3600 * param.Ecap))
```

(continues on next page)

(continued from previous page)

```
model.E = np.concatenate([np.zeros((nx, nu)), np.eye(nx)], axis=1)

# Initial and final conditions
model.xinitidx = np.arange(nu, nu + nx);
```

The use of *smooth maximum* approximation (see section Section 20.3) ensures a nonnegative square root term and thus a feasible solution.

Inequality constraints

The inequality constraints *model.ineq* comprise all decision variable bounds described previously. For purposes of model feasibility, two additional trivial constraints pertaining to the vehicle velocity's square root term are included in the model, as shown in the code-snippets.

Matlab

Python

```
%% Inequality constraints
% Upper/lower variable bounds lb <= z <= ub
%                             inputs                          |                states
%          slack       Ft       Fb            Tch                       v           t␣
↪    SoC
model.lb = [0,     param.FtMin,  0.,           0.,            kmh2ms(param.vMin),  0.,
↪    0.];
model.ub = [0.1,   param.FtMax,  param.FbMax,  param.TchMax,  kmh2ms(param.vMax), ␣
↪+inf,  1.];

% Nonlinear inequalities hl <= h(z,p) <= hu
model.ineq = @(z,p) [z(2) - param.FtMaxHyp(z(5));
                     z(4) - p(4);
                     z(7) - param.SoCmax - z(1);
                     param.SoCmin - z(7) - z(1);
                     2*p(5)/param.meq*(z(2) - z(3) - param.cr*param.mv*param.
↪g*cos(p(3)) - param.mv*param.g*sin(p(3)) - 0.5*param.ca*param.Af*param.rhoa*z(5)^2)␣
↪+ z(5)^2 - kmh2ms(ForcesMax((1-z(4))*p(1), param.vMin+1))^2;
                     kmh2ms(p(2))^2 - (2*p(5)/param.meq*(z(2) - z(3) - param.cr*param.
↪mv*param.g*cos(p(3)) - param.mv*param.g*sin(p(3)) - 0.5*param.ca*param.Af*param.
↪rhoa*z(5)^2) + z(5)^2)];

% Upper/lower bounds for inequalities
model.hu = [0,    0,    0,    0,    0,    0];
model.hl = [-inf, -inf, -inf, -inf, -inf, -inf];
```

```
# Inequality constraints
# Upper/lower variable bounds lb <= z <= ub
#                             inputs                              |              ␣
↪states
#                  slack       Ft     Fb           Tch                     v         ␣
↪ t      SoC
model.lb = np.array([0.,  param.FtMin, 0.,          0.,           kmh2ms(param.vMin),␣
↪ 0.,      0.])
model.ub = np.array([0.1, param.FtMax, param.FbMax, param.TchMax, kmh2ms(param.vMax),␣
↪ np.inf,  1.])
```

(continues on next page)

(continued from previous page)

```
# Nonlinear inequalities hl <= h(z,p) <= hu
model.ineq = lambda z,p: casadi.vertcat(z[1] - param.FtMaxHyp(z[4]),
                                        z[3] - p[3],
                                        z[6] - param.SoCmax - z[0],
                                        param.SoCmin - z[6] - z[0],
                                        2 * p[4] / param.meq * (z[1] - z[2] - param.
↪cr * param.mv * param.g * np.cos(p[2]) - param.mv * param.g * np.sin(p[2]) - 0.5 *␣
↪param.ca * param.Af * param.rhoa * z[4]**2) + z[4]**2 - kmh2ms(forcespro.modelling.
↪smooth_max((1 - z[3]) * p[0], param.vMin + 1))**2,
                                        kmh2ms(p[1])**2 - (2 * p[4] / param.meq *␣
↪(z[1] - z[2] - param.cr * param.mv * param.g * np.cos(p[2]) - param.mv * param.g *␣
↪np.sin(p[2]) - 0.5 * param.ca * param.Af * param.rhoa * z[4]**2) + z[4]**2))

# Upper/lower bounds for inequalities
model.hu = np.array([0,       0,        0,       0,      0,       0])
model.hl = np.array([-np.inf, -np.inf, -np.inf, -np.inf, -np.inf, -np.inf])
```

Generating the FORCESPRO NLP solver

To generate a suitable NLP solver for our MPC problem one needs to provide the *model* and *codeoptions*. The *model* has been populated above and we now specify the desired *codeoptions* and generate the solver by calling *FORCES_NLP*. The following code-snippets show how this can be done:

Matlab

Python

```
%% Generate FORCESPRO solver
% Define solver options
codeoptions = getOptions('FORCESNLPsolver');
codeoptions.printlevel = 0;
codeoptions.nlp.compact_code = 1;

% Generate code
FORCES_NLP(model, codeoptions);
```

```
# Generate FORCESPRO solver
# -------------------------

# Define solver options
codeoptions = forcespro.CodeOptions("FORCESNLPsolver")
codeoptions.printlevel = 0
codeoptions.nlp.compact_code = 1

# Generate code
solver = model.generate_solver(codeoptions)
```

Calling the solver

Once the solver has been generated it needs to be provided with initial and runtime parameters. In this example we are running a single full-horizon snapshot instead of a traditional rolling-horizon MPC, as follows:

Matlab

Python

```
function [ sim ] = runSimulation(trip, param, model)
% Defines initial and stage-dependent runtime parameters and solves the
% problem
    %% Define spatially distributed (i.e. stage-dependent) parameters
    % Road speed limits and slope angles
    [vMaxRoad, ~, vMinRoad, roadSlope] = setupRoadParameters(param.vMin, trip);

    % Maximum permissible charging time
    deltaTchMax = setupChargingTime(param.TchMax, trip);

    %% Initialize the problem
    problem.x0 = zeros(model.N*model.nvar,1);
    sim.kMax = trip.dist/trip.Ls;
    np = model.npar;

    if trip.initSpeed < param.vMin || trip.initSpeed > param.vMax
        v1 = kmh2ms(param.vMin);
        warning('Initial vehicle speed is outside of the speed limits. It will be set␣
↪to the minimum speed limit at t=0.')
    else
        v1 = kmh2ms(trip.initSpeed);
    end

    if trip.initSoC < param.SoCmin || trip.initSoC > param.SoCmax
        SoC1 = 0.5;
        warning('Initial vehicle state-of-charge is outside of the limits. It will be␣
↪set to 50% at t=0.')
    else
        SoC1 = trip.initSoC;
    end

    % X(1) = [v(1); t(1); SoC(1)]
    problem.xinit = [v1; 0; SoC1];

    %% Solve the problem

    % Set runtime parameters
    problem.all_parameters = zeros(np*model.N,1);
    for i = 1:model.N
        problem.all_parameters((i-1)*np+1:i*np) = [vMaxRoad(i); vMinRoad(i);␣
↪roadSlope(i*trip.Ls*1e3); deltaTchMax(i); trip.Ls*1e3];
    end

    % Call solver
    [solverout,exitflag,info] = FORCESNLPsolver(problem);
    sim.exitflag = exitflag;

    % Extract state and control vector
    if exitflag == 1
        sim.Z = unpackStruct(solverout,model.nvar);
        sim.solvetime = info.solvetime;
        sim.iters = info.it;
    else
        error('Some problem in solver.');
    end
end
```

```
def runSimulation(trip, param, model, solver):
    """Defines initial and stage-dependent runtime parameters and solves the problem""
↪"
    x0 = np.zeros(model.N * model.nvar)
    problem = {"x0": x0}
    kMax = model.N
    npar = model.npar

    # Initialize system states
    # ------------------------

    assert trip.initSpeed >= param.vMin or trip.initSpeed <= param.vMax, 'Initial␣
↪vehicle speed is outside of the speed limits.'
    assert trip.initSoC >= param.SoCmin or trip.initSoC <= param.SoCmax, 'Initial␣
↪vehicle state-of-charge is outside of the limits.'

    # Initialize the problem
    # X(1) = [v(1); t(1); SoC(1)]
    problem["xinit"] = [kmh2ms(trip.initSpeed), 0, trip.initSoC]

    # Define spatially distributed (i.e. stage-dependent) parameters
    # --------------------------------------------------------------

    # Road speed limits and slope angles
    vMaxRoad, _, vMinRoad, roadSlope = setupRoadParameters(param.vMin, trip)

    # Maximum permissible charging time
    deltaTchMax = setupChargingTime(param.TchMax, trip);

    # Set runtime parameters
    problem["all_parameters"] = np.zeros(npar * model.N)
    for i in range(model.N):
        problem["all_parameters"][i * npar : (i + 1) * npar] = np.array([vMaxRoad[i],␣
↪vMinRoad[i], roadSlope((i + 1) * trip.Ls * 1e3).item(), deltaTchMax[i], trip.Ls *␣
↪1e3])

    # Solve the problem
    # -----------------
    solverout, exitflag, info = solver.solve(problem)

    assert exitflag == 1, 'Some problem in solver.'

    sim = {"exitflag": exitflag}
    sim["Z"] = unpackDict(solverout)
    sim["kMax"] = kMax
    sim["solvetime"] = info.solvetime
    sim["iters"] = info.it
    sim = displayResults(sim);

    return sim
```

Results

We consider two trips of different lengths previously described in section Section 11.19.1 with three benchmark studies conducted for the shorter trip and two studies for the longer trip:

1. a short trip of $50\,\mathrm{km}$ with arbitrarily generated road profile, 5 charging stations along the road, and vehicle's initial SoC of: a) $75\,\%$; b) $50\,\%$; and c) $25\,\%$.
2. a long trip of $573\,\mathrm{km}$ representing a Munich - Cologne trip with actual road profile, vehicle's initial SoC of $90\,\%$, and: a) 4 charging stations along the road; and b) 2 charging stations along the road.

Studies **1a - 1c** have the same allocation of charging stations, with chargers placed at $8\,\mathrm{km}$, $18\,\mathrm{km}$, $30\,\mathrm{km}$, $40\,\mathrm{km}$ and $45\,\mathrm{km}$ marks. Furthermore, we allow **optional** charging for all three studies. On the other hand, a Munich - Cologne trip **2a** considers four chargers placed at $110\,\mathrm{km}$, $150\,\mathrm{km}$, $250\,\mathrm{km}$ and $375\,\mathrm{km}$, while trip **2b** has two available chargers at $150\,\mathrm{km}$ and $375\,\mathrm{km}$. In contrast to the short trip, here we impose **mandatory** charging requirement. The most notable trip metrics for all five studies are showcased in Table 11.1.

Table 11.1: Summary of vehicle performance for different trip parameters and initial SoC levels.

Trip	Distance	SoC (t=0)	# stations	# stops	Total time	Charge time
1a	50 km	75 %	5	0	34.14 min	0 min
1b	50 km	50 %	5	0	44.56 min	0 min
1c	50 km	25 %	5	1	51.82 min	16.3 min
2a	573 km	90 %	4	4	345.03 min	66.52 min
2b	573 km	90 %	2	2	346.91 min	68.81 min

The simulation results for studies **1a** and **1b** are depicted in Figure 11.68 and Figure 11.69, respectively. Since the vehicle has sufficiently high SoC, the charging is not needed in both cases. However, in the second trip the vehicle is forced to reduce its velocity significantly below the speed limit in order to preserve energy and complete the trip with sufficient battery charge, which leads to a higher trip time.

A low initial SoC in study **1c** forces the vehicle to charge, as shown in Figure 11.70, which leads to an even higher trip time. The EV decides to stop at the first charging stop and charge just enough to complete the trip with the SoC at the lower bound.

The longer trip studies with **mandatory** charging presented in Figure 11.71 and Figure 11.72 again indicate that the EV will complete the trip at the minimum permissible SoC in order to minimize the total charging time. The results also suggest that the vehicle will intentionally reduce speed in order to preserve energy: the availability of only 2 chargers in study **2b** forces the vehicle to slightly reduce speed midway through the trip. The overall trip time increases by $0.55\,\%$, and the total charging time is $3.44\,\%$ bigger compared to study **2a**.

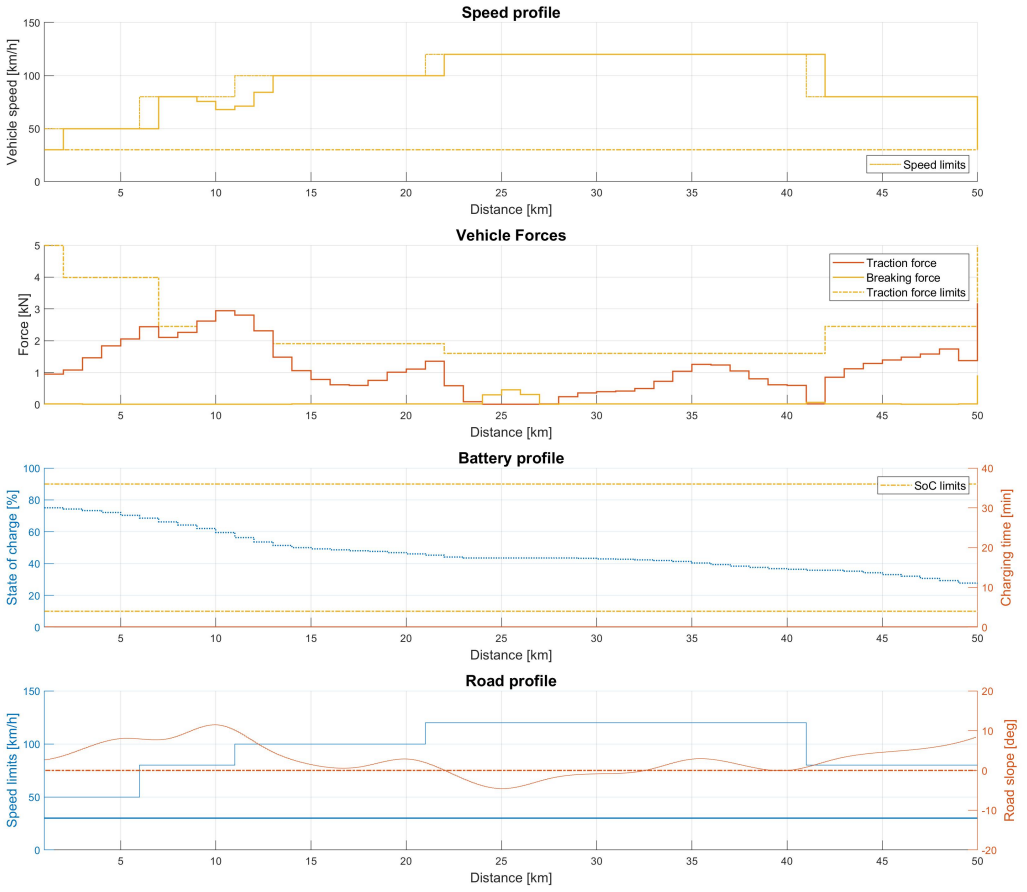


Figure 11.68: Vehicle performance on a short trip with 5 charging stations and 75 % initial SoC.

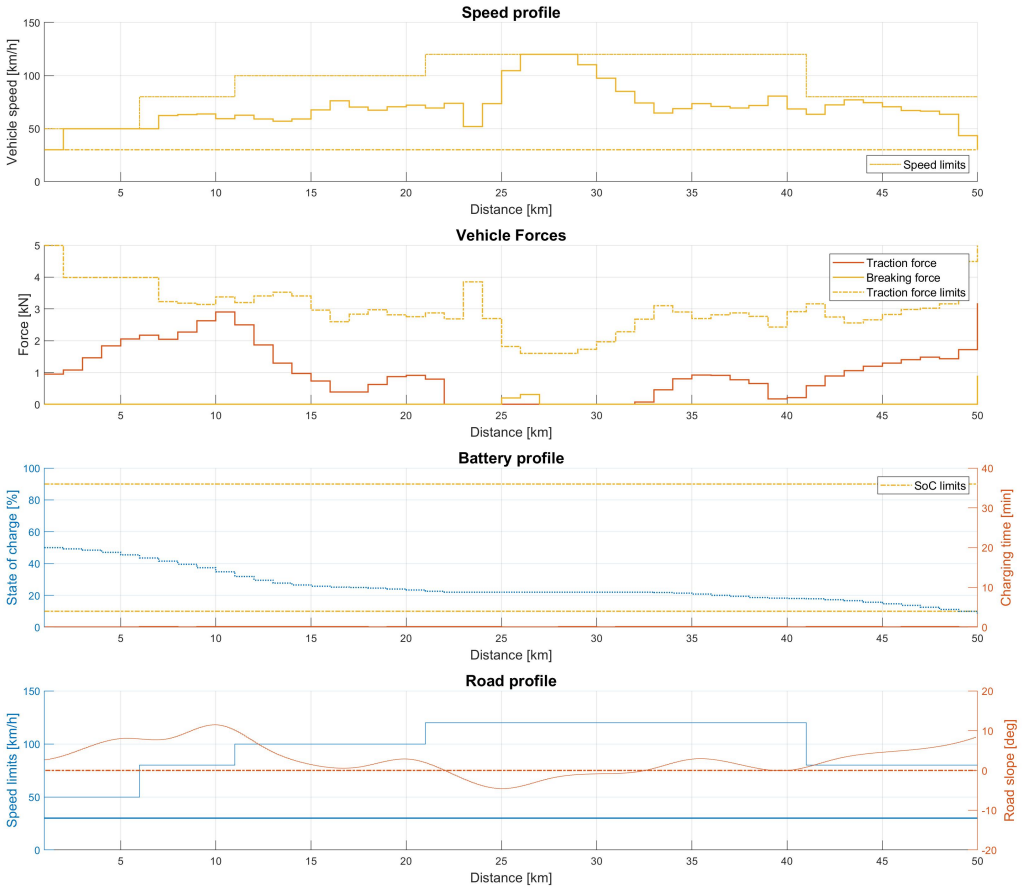


Figure 11.69: Vehicle performance on a short trip with 5 charging stations and 50 % initial SoC.

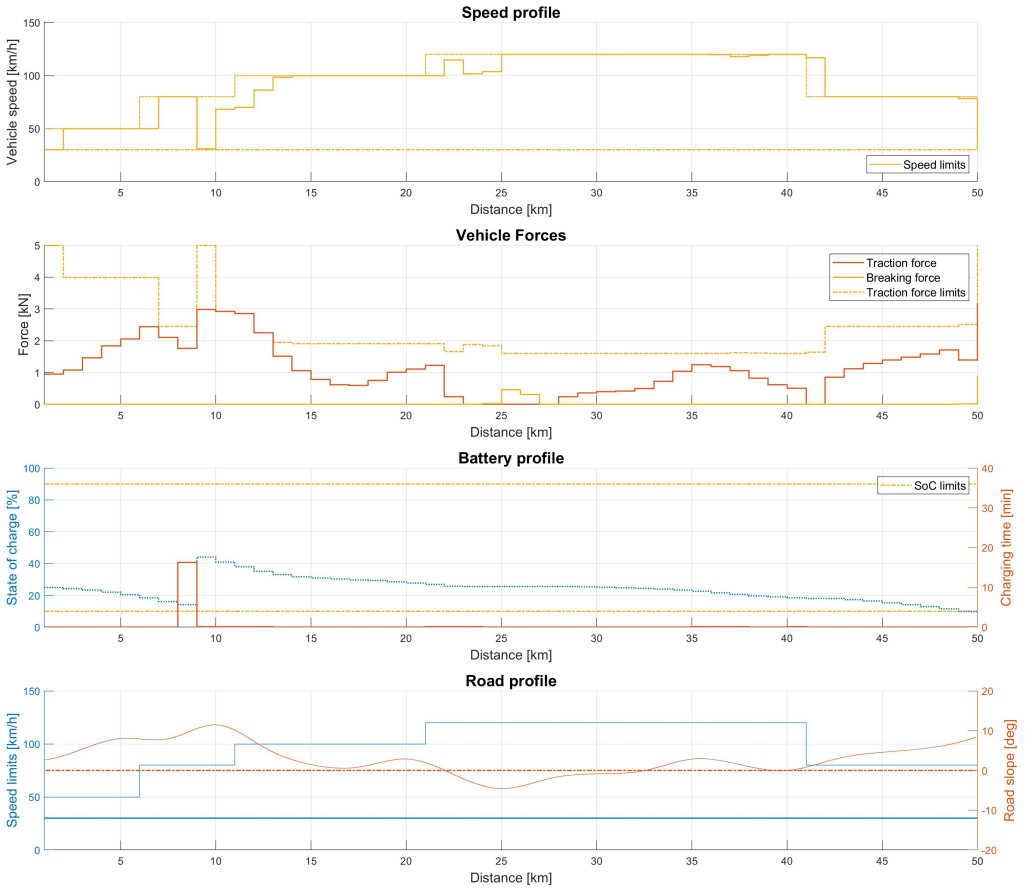


Figure 11.70: Vehicle performance on a short trip with 5 charging stations and 25 % initial SoC.

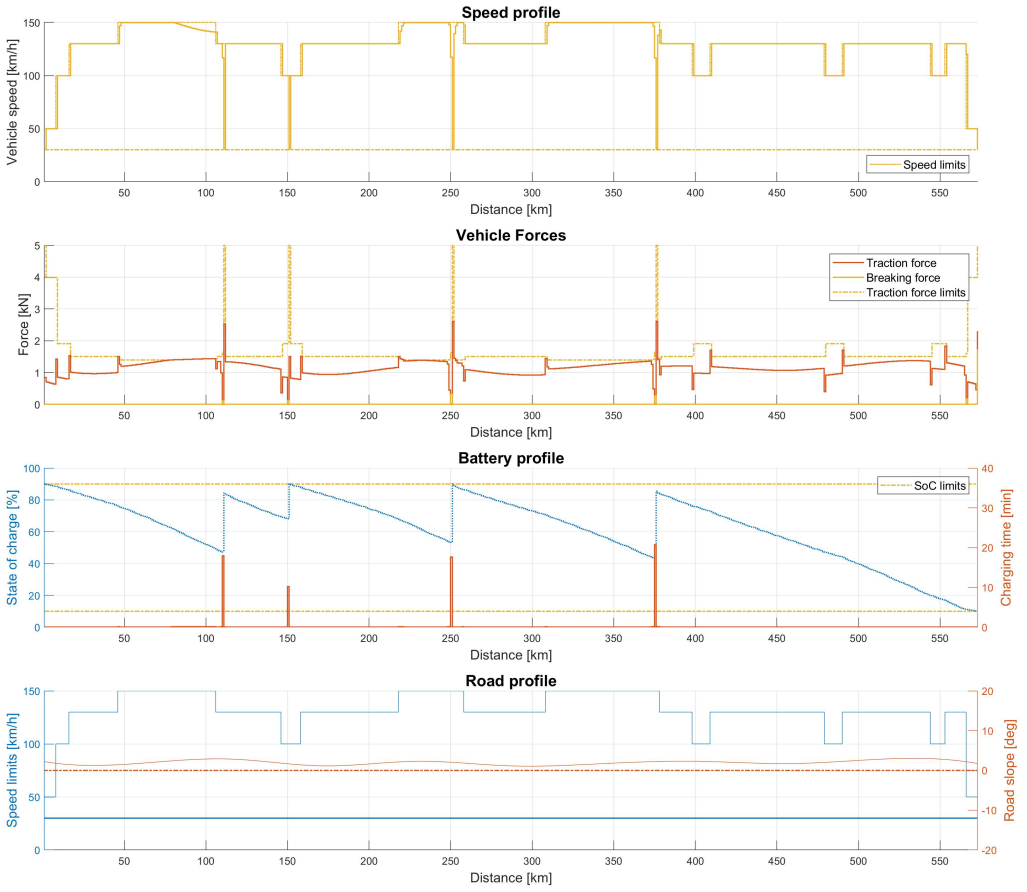


Figure 11.71: Vehicle performance on a long trip with 4 charging stations and 90 % initial SoC.

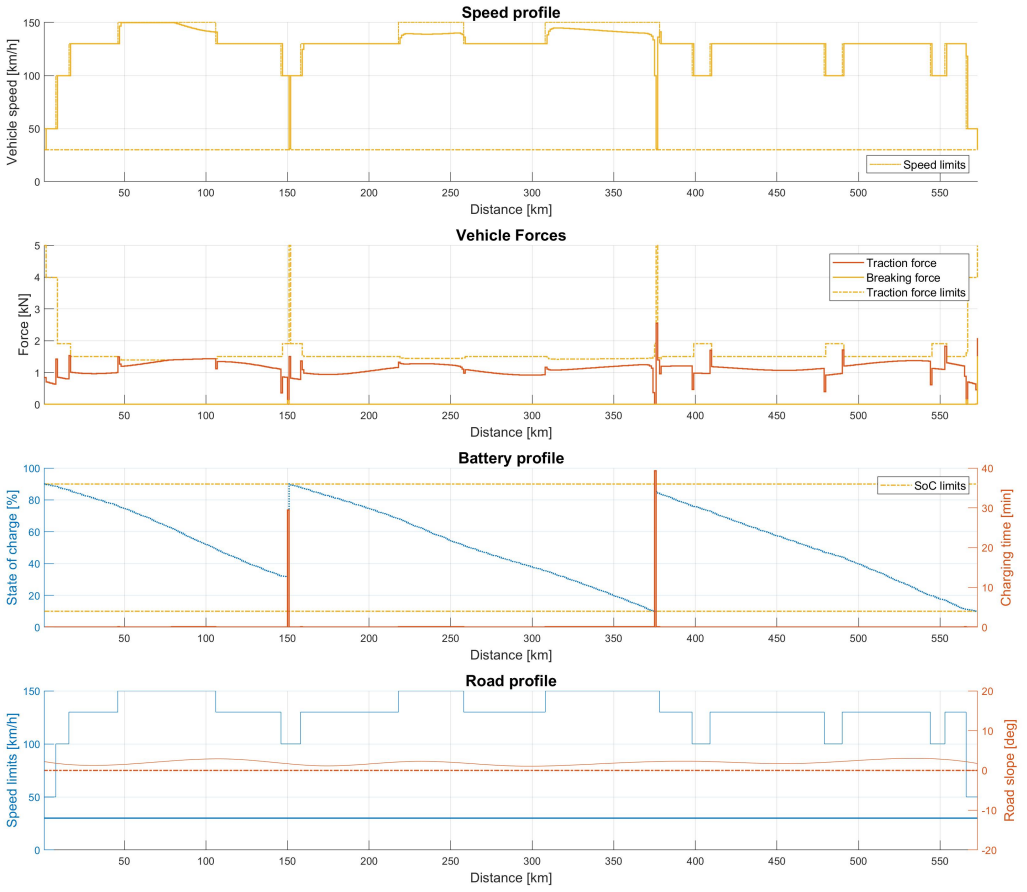


Figure 11.72: Vehicle performance on a long trip with 2 charging stations and 90 % initial SoC.

11.20 High-level interface: Extended optimal EV charging and speed profile example using a 2D motor efficiency map (MATLAB & PYTHON)

- *Problem Overview*
  - *Vehicle Dynamics*
  - *Vehicle Energetics*
  - *Bounds*
  - *MPC Formulation*
  - *Model Parameters*
- *Defining the MPC Problem*
  - *Scaling*
  - *Objective function*
  - *Equality constraints*
  - *Inequality constraints*
  - *Generating the FORCESPRO NLP solver*
  - *Calling the solver*
  - *Results*

This example is an updated and more realistic version of Section 11.19, since it uses 2D motor efficiency map data. In addition to showcasing 2D spline usage, this example also illustrates how to scale optimization variables, which can be used to increase numerical stability of optimization problems in general.

In this example we illustrate the simplicity of the high-level user interface on an optimal electric vehicle (EV) speed and charging management. The controller plans the car's speed and charging trajectory such that the given route is traversed in the shortest time possible, while simultaneously respecting speed limits as well as vehicle's and battery's technical requirements. Furthermore, this example illustrates how to use two-dimensional splines as well as scaling of an optimization problem to make it numerically better conditioned.

11.20.1 Problem Overview

We consider a single EV and a fixed route with given road slopes (i.e. angles) and speed limits. Optionally, some charging stations could be available at predefined locations along the road. Vehicle operation is defined by longitudinal vehicle dynamics, powertrain efficiency and battery capacity. Due to the spatial characteristic of the problem, the formulation is discretized in space domain. This problem is primarily based on the work in [JiaJibItoGor19_2D] and [JiaJibGor20_2D].

Vehicle Dynamics

Assuming a sample distance L_s, the longitudinal velocity of the vehicle in space domain at the discrete step $(i+1) \in \mathcal{I}$ can be determined using the velocity at the previous step (i) and the difference in kinetic energy between these two consecutive steps, as follows:

$$\frac{1}{2} m_\mathrm{eq} \left(v_{i+1}^2 - v_i^2 \right) = L_\mathrm{s} \left(F_{\mathrm{t},i} - F_{\mathrm{b},i} - F_{\mathrm{r},i} \right)$$

where the resistance force is comprised of the following three components:

$$F_{\mathrm{r},i} = \underbrace{c_\mathrm{r} m_\mathrm{v} g \cos(\alpha_\mathrm{i})}_{\text{rolling resistance}} + \underbrace{m_\mathrm{v} g \sin(\alpha_\mathrm{i})}_{\text{gravitational force}} + \underbrace{\frac{1}{2} c_\mathrm{a} A_\mathrm{f} \rho_\mathrm{a} v_i^2}_{\text{air drag force}}$$

The equivalent mass $m_\mathrm{eq} = m_\mathrm{v}(1 + e_\mathrm{I})$ includes vehicle mass m_v and the effect of all rotational masses captured by the mass factor e_I. The road slope is represented by the road angle α. Variables $F_\mathrm{t} \geq 0$ and $F_\mathrm{b} \geq 0$ represent the traction and braking forces on the wheels, which are generated by the powertrain and mechanical braking system separately (*note: regenerative braking capabilities could be included by allowing the traction power to be negative*). The total driving resistance F_r comprises rolling resistance force, gravitational force and air drag force (while neglecting the direct impact of wind speed), with c_a being the air drag coefficient, A_f the projected frontal area, ρ_a the air density, g the gravitational acceleration, and c_r the rolling resistance coefficient.

The vehicle's velocity dynamics are thus defined by

$$v_{i+1} = \sqrt{\frac{2\left(F_{\mathrm{t},i} - F_{\mathrm{b},i} - F_{\mathrm{r},i}\right) L_\mathrm{s}}{m_\mathrm{eq}} + v_i^2}$$

with the assumption that the speed is constant while traversing a single discrete road segment. Note that the expression below the square root must be strictly positive in order for the problem to be feasible, as will be discussed later on. In addition to velocity, the time is also a state variable of the model and can be computed as

$$t_{i+1} = t_i + \frac{L_\mathrm{s}}{v_i}$$

Vehicle Energetics

EV's powertrain efficiency $\eta(v, F_\mathrm{t}, \zeta)$ is a function of vehicle's speed, traction force and battery's state-of-charge (SoC), as shown by the efficiency maps depicted in Figure 11.73. Nevertheless, the impact of SoC on powertrain efficiency is negligible as SoC does not change dramatically within a short time period if the battery capacity is sufficiently large. Therefore, one could generate several 2D maps of the form $\eta(v, F_\mathrm{t})$, pre-calculated at different SoC levels such that the most accurate one can be selected according to the current SoC.

For the purpose of this example, we have have generated a single 2D efficiency map depicted in Figure 11.74 used across all SoC levels.

In Option 1, we fit the 2D efficiency map $\eta(v, F_\mathrm{t})$ data as a cubic bivariate spline using Scipy's `SmoothBivariateSpline` routine. In order to utilize Scipy's fitting routines, a valid Python installation needs to be present, even if it is called via MATLAB. Note that not all Python versions are compatible with all MATLAB versions. For a compatibility list see here (Compatibility).

In principle, we could have utilized any of the provided 2D-spline fitting methods. There is one that uses CasADi and does not require a valid Python installation. For an extensive list of the available 2D-spline related methods, see *Interpolations 2D (B-splines)*.

In Option 2, we assume that we have already fitted the 2D-splines (i.e. must be representable as a B-spline) and therefore, we can provide the coefficients matrix in matrix-indexed form (i.e. x corresponds to rows and y corresponds to columns) as well as the knots tx and ty directly.

In this example, we can toggle between the two options by setting `useScipy`. In MATLAB, the example additionally checks that Python as well as the Python package Scipy are installed using the function `validPythonPackages`. The Python interface of FORCESPRO already requires a valid Scipy installation, which is why the check is omitted there.

The two 2D-spline options are generated as follows:

Matlab

Python

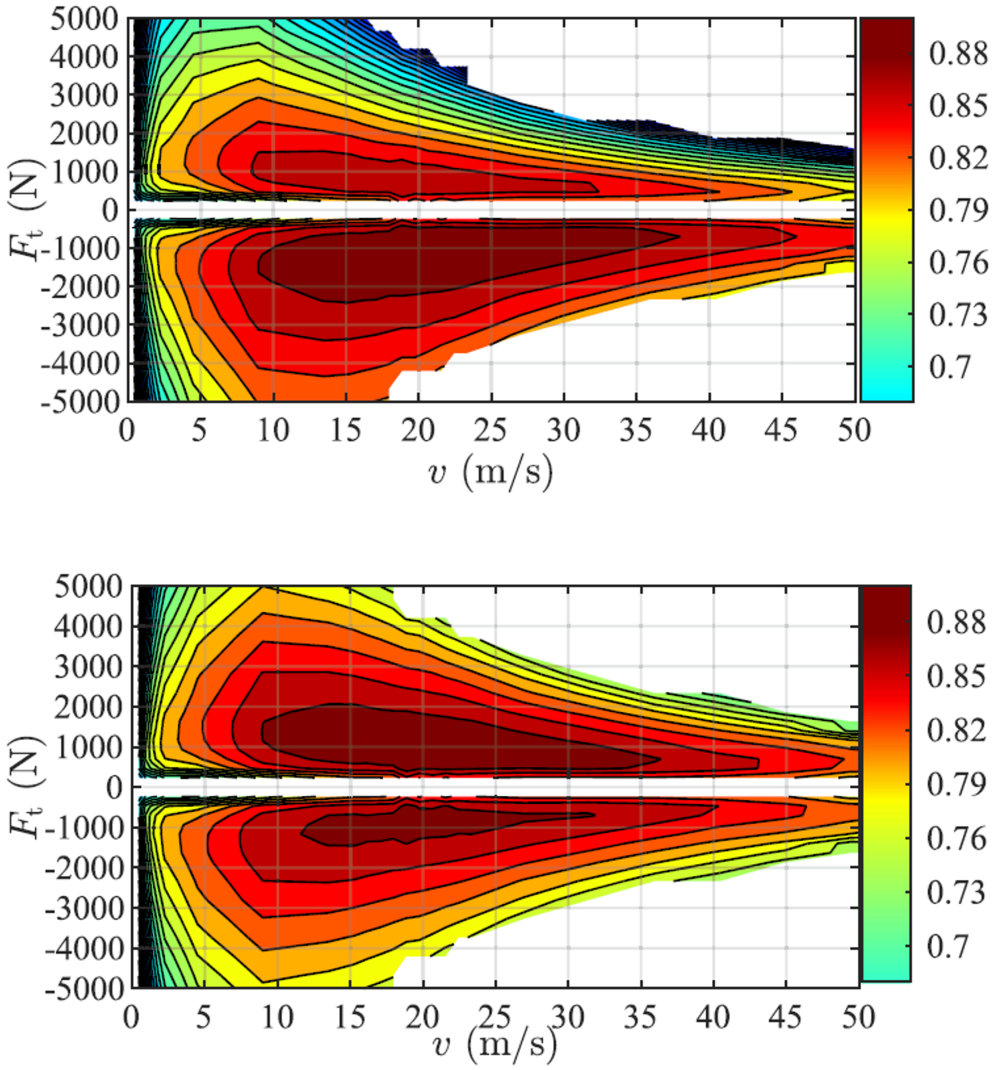


Figure 11.73: 2D efficiency maps of the EV's powertrain with two different SoC levels taken from [JiaJibGor20_2D]: (top) η with SoC $\zeta = 10\,\%$; (bottom) η with SoC $\zeta = 90\,\%$.

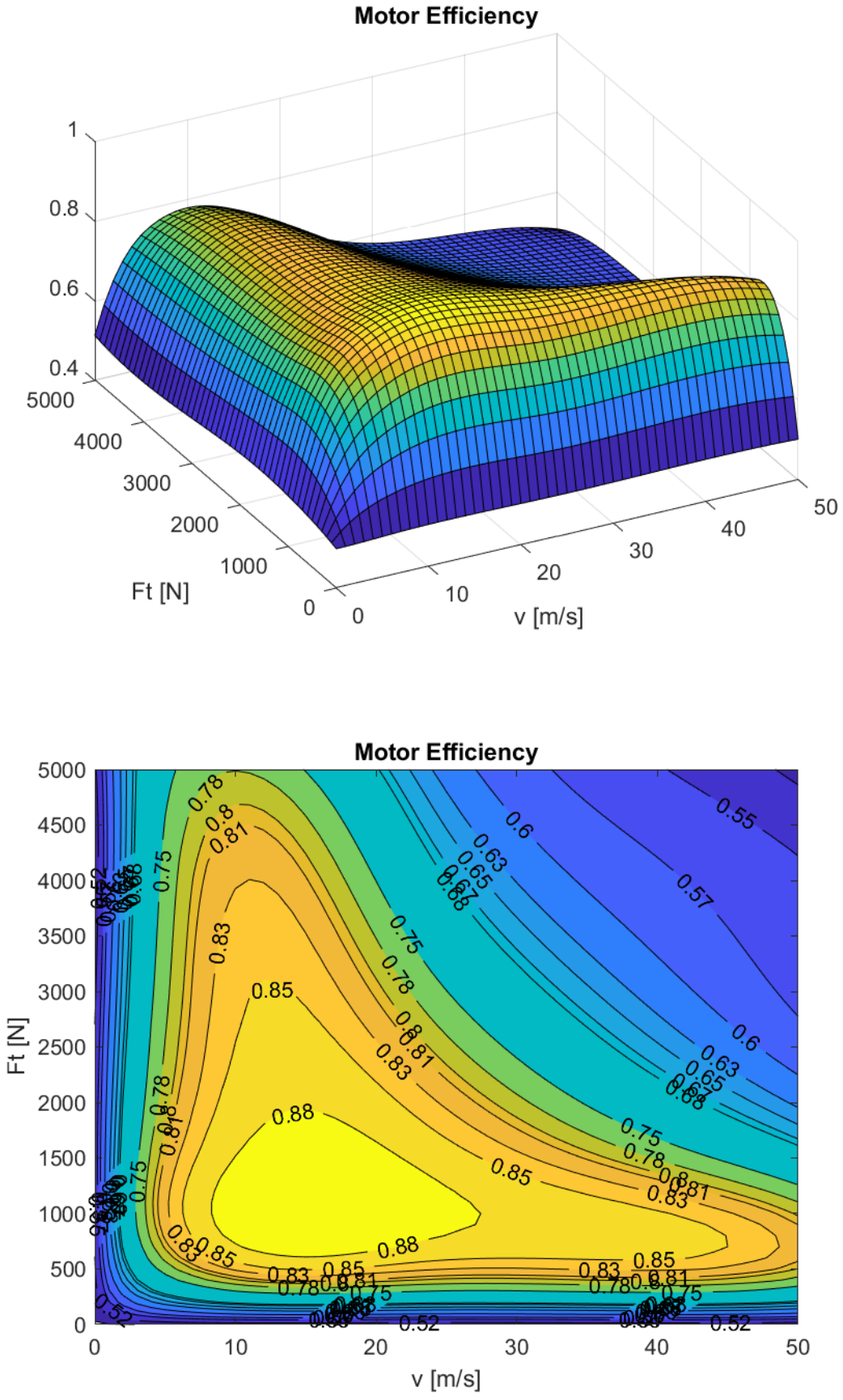


Figure 11.74: Generated 2D EV efficiency map.

```
function [ etaMotorSpline ] = setupMotorEfficiency()
% Returns a cubic bivariate spline representing motor efficiency as a
% function of velocity and traction force.
%
% The data is taken from "Jia, Y.; Jibrin, R.; Goerges, D.: Energy-Optimal
% Adaptive Cruise Control for Electric Vehicles Based on Linear and
% Nonlinear Model Predictive Control. In: IEEE Transactions on Vehicular
% Technology, vol. 69, no. 12, pp. 14173-14187, Dec. 2020."

    vMaxMEff = 180;     % maximum velocity in the lookup table [km/h]
    FtMaxMEff = 5e3;    % maximum traction force in the lookup table [N]
    vMaxMEff = kmh2ms(vMaxMEff);
    vSampleOrig = 0:vMaxMEff/10:vMaxMEff;
    FtSampleOrig = 0:FtMaxMEff/5:FtMaxMEff;

    % denser grid point selection near the edges with the sharp transitions
    vSampleOrig = [vSampleOrig(1), mean(vSampleOrig(1:2)), vSampleOrig(2:end)];
    FtSampleOrig = [FtSampleOrig(1), mean(FtSampleOrig(1:2)), FtSampleOrig(2:end)];

    %     v [m/s] =        0,  2.5,    5,   10,   15,   20,   25,   30,   35,   40,
↪45,   50
    etaSampleOrig = [
                        0.50, 0.50, 0.50, 0.50, 0.50, 0.50, 0.50, 0.50, 0.50, 0.50, 0.
↪50, 0.50; ... % F [N] = 0
                        0.50, 0.68, 0.80, 0.85, 0.85, 0.85, 0.85, 0.85, 0.85, 0.85, 0.
↪83, 0.81; ... % F [N] = 500
                        0.50, 0.78, 0.81, 0.88, 0.88, 0.88, 0.88, 0.88, 0.85, 0.83, 0.
↪81, 0.80; ... % F [N] = 1000
                        0.50, 0.75, 0.81, 0.85, 0.88, 0.85, 0.83, 0.81, 0.78, 0.68, 0.
↪65, 0.63; ... % F [N] = 2000
                        0.50, 0.68, 0.80, 0.83, 0.83, 0.81, 0.78, 0.68, 0.65, 0.63, 0.
↪60, 0.57; ... % F [N] = 3000
                        0.50, 0.67, 0.78, 0.81, 0.80, 0.78, 0.65, 0.63, 0.60, 0.57, 0.
↪55, 0.55; ... % F [N] = 4000
                        0.50, 0.67, 0.75, 0.78, 0.75, 0.65, 0.63, 0.60, 0.57, 0.55, 0.
↪52, 0.52; ... % F [N] = 5000
                    ];

    % spline
    useScipy = false;
    if useScipy && validPythonPackages()
        % Option 1: fitting with Scipy's SmoothBivariateSpline
        % convert to 1-D sequences of data points
        [vTemp, FtTemp] = meshgrid(vSampleOrig, FtSampleOrig);
        vSample = reshape(vTemp.', 1, []);
        FtSample = reshape(FtTemp.', 1, []);
        etaSample = reshape(etaSampleOrig', [1, numel(etaSampleOrig)]);

        s = 0.025;
        w = ones(1, length(vSample));
        bbox = [min(vSample), max(vSample), min(FtSample), max(FtSample)];
        kx = 3;
        ky = 3;
        rankTol = 1e-8;
        scaling = [1e1, 1e3, 1];
        etaMotorSpline = ForcesInterpolationFit_SmoothBivariateSpline(vSample,
↪FtSample, etaSample, w, bbox, kx, ky, s, rankTol, scaling);
```

(continues on next page)

(continued from previous page)

```
    else
        % Option 2: providing 2D-spline coefficients
        kx = 3;
        ky = 3;
        tx = [0, 0, 0, 0, 6.3993742001266, 24.1805094335686, 50, 50, 50, 50];
        ty = [0, 0, 0, 0, 760.320551795358, 1503.23831410745, 5000, 5000, 5000, 5000];
        coeffs = [0.498727471092637   0.511037494098402   0.524901875945660   0.
↪548511907792580   0.475607487652389   0.513135686437586
                  0.498465143841756   0.654046675851965   0.783006359203673   0.
↪713858506960411   0.676705972846789   0.681154627267599
                  0.510442836827170   0.854158083414314   1.007125597517271   0.
↪847628255554849   1.018658592375758   0.878826758082995
                  0.495093430686428   0.735511289747710   0.857756122056489   0.
↪863078750613390   0.548595365620131   0.497927393614425
                  0.510240152112442   0.835399001169469   0.952683895958243   0.
↪536982511482952   0.563982968586042   0.577416237725016
                  0.501474795696226   0.773879473939183   0.878143062979889   0.
↪444534437467682   0.615960539904494   0.508404545245928];
        etaMotorSpline = ForcesInterpolation2D(tx, ty, coeffs, kx, ky);
    end
end
```

```
def setupMotorEfficiency(self):
    """
    Returns a cubic bivariate spline representing motor efficiency as a function of␣
↪velocity and traction force.

    The data is taken from "Jia, Y.; Jibrin, R.; Goerges, D.: Energy-Optimal Adaptive␣
↪Cruise Control for Electric Vehicles Based on Linear and Nonlinear Model Predictive␣
↪Control. In: IEEE Transactions on Vehicular Technology, vol. 69, no. 12, pp. 14173-
↪14187, Dec. 2020."
    """
    vMaxMEff = 180  # maximum velocity in the lookup table [km/h]
    FtMaxMEff = 5e3 # maximum traction force in the lookup table [N]
    vMaxMEff = self.kmh2ms(vMaxMEff)

    vSampleOrig = np.linspace(0, vMaxMEff, 11)
    FtSampleOrig = np.linspace(0, FtMaxMEff, 6)

    # denser grid point selection near the edges with the sharp transitions
    vSampleOrig = np.insert(vSampleOrig, 1, np.mean(vSampleOrig[0:2]))
    FtSampleOrig = np.insert(FtSampleOrig, 1, np.mean(FtSampleOrig[0:2]))

    #     v [m/s] =         0,  2.5,    5,   10,   15,   20,   25,   30,   35,   40, ␣
↪ 45,   50
    etaSampleOrig = [
                        [0.50, 0.50, 0.50, 0.50, 0.50, 0.50, 0.50, 0.50, 0.50, 0.50,␣
↪0.50, 0.50], # F [N] = 0
                        [0.50, 0.68, 0.80, 0.85, 0.85, 0.85, 0.85, 0.85, 0.85, 0.85,␣
↪0.83, 0.81], # F [N] = 500
                        [0.50, 0.78, 0.81, 0.88, 0.88, 0.88, 0.88, 0.88, 0.85, 0.83,␣
↪0.81, 0.80], # F [N] = 1000
                        [0.50, 0.75, 0.81, 0.85, 0.88, 0.85, 0.83, 0.81, 0.78, 0.68,␣
↪0.65, 0.63], # F [N] = 2000
                        [0.50, 0.68, 0.80, 0.83, 0.83, 0.81, 0.78, 0.68, 0.65, 0.63,␣
↪0.60, 0.57], # F [N] = 3000
```

(continues on next page)

(continued from previous page)

```
                        [0.50, 0.67, 0.78, 0.81, 0.80, 0.78, 0.65, 0.63, 0.60, 0.57,␣
↪0.55, 0.55], # F [N] = 4000
                        [0.50, 0.67, 0.75, 0.78, 0.75, 0.65, 0.63, 0.60, 0.57, 0.55,␣
↪0.52, 0.52]  # F [N] = 5000
                      ]
    etaSampleOrig = np.array(etaSampleOrig)

    # spline
    useScipy = False
    if useScipy:
        # spline
        # Option 1: fitting with Scipy's SmoothBivariateSpline
        # convert to 1d sequences of data points
        [vTemp, FtTemp] = np.meshgrid(vSampleOrig, FtSampleOrig)
        vSample = vTemp.flatten()
        FtSample = FtTemp.flatten()
        etaSample = etaSampleOrig.flatten()

        s = 0.025
        w = np.ones((1, len(vSample)))
        bbox = [min(vSample), max(vSample), min(FtSample), max(FtSample)]
        kx = 3
        ky = 3
        eps = 1e-8
        scaling = [1e1, 1e3, 1]
        etaMotorSpline = InterpolationFit_SmoothBivariateSpline(x=vSample, y=FtSample,
↪ z=etaSample, w=w, bbox=bbox, kx=kx, ky=ky, s=s, eps=eps, scaling=scaling)
    else:
        # Option 2: providing 2D-spline coefficients
        kx = 3
        ky = 3
        tx = np.array([0, 0, 0, 0, 6.3993742001266, 24.1805094335686, 50, 50, 50, 50])
        ty = np.array([0, 0, 0, 0, 760.320551795358, 1503.23831410745, 5000, 5000,␣
↪5000, 5000])
        coeffs = np.array([
                           [0.498727471092637,   0.511037494098402,   0.
↪524901875945660,   0.548511907792580,   0.475607487652389,   0.513135686437586],
                           [0.498465143841756,   0.654046675851965,   0.
↪783006359203673,   0.713858506960411,   0.676705972846789,   0.681154627267599],
                           [0.510442836827170,   0.854158083414314,   1.
↪007125597517271,   0.847628255554849,   1.018658592375758,   0.878826758082995],
                           [0.495093430686428,   0.735511289747710,   0.
↪857756122056489,   0.863078750613390,   0.548595365620131,   0.497927393614425],
                           [0.510240152112442,   0.835399001169469,   0.
↪952683895958243,   0.536982511482952,   0.563982968586042,   0.577416237725016],
                           [0.501474795696226,   0.773879473939183,   0.
↪878143062979889,   0.444534437467682,   0.615960539904494,   0.508404545245928]
                          ])

        etaMotorSpline = Interpolation2D(tx, ty, coeffs, kx, ky)

    return etaMotorSpline
```

Note that for the fitting you can set a scaling vector for numerically robustifying Scipy's 2D spline fitting routine in case the input and output variables have vastly different magnitudes. The returned spline takes care of scaling such that the inputs to the spline are still the physical (unscaled) quantities. This scaling is solely utilized for the spline fitting and is different from

optimization variable scaling, which follows later in this example (see Section 11.20.2).

Assuming a set of predefined charging stops at discrete spatial steps $k \in \mathcal{K} \subset \mathcal{I}$, we can include either **optional** or **mandatory** charging at each of those stops (to be defined by the user). Given the implemented powertrain efficiency $\eta_i(F_{\mathrm{t},i})$ and potential charging decisions, the driving energy depletion and charging instances are incorporated into the SoC dynamics at discrete step i as:

$$\zeta_{i+1} = \zeta_i - \frac{F_{\mathrm{t},i} L_{\mathrm{s}}}{\eta_i E_{\mathrm{cap}}} + \frac{P_{\mathrm{ch},i}}{E_{\mathrm{cap}}} \Delta t_i$$

where E_{cap} denotes the battery's energy capacity, $P_{\mathrm{ch},i}(\zeta_i)$ is a vehicle's charging power dependent on the SoC, and Δt_i is the total charging time spent at location $i \in \mathcal{K}$. Note that SoC ζ is normalized on an interval $[0, 1]$.

In this example we will particularly focus on the performance of a BMW i3. The charging profiles $P_{\mathrm{ch}}(\zeta)$ for three production models with different battery capacities are depicted in Figure 11.75 [Fastned_2D], and can be approximated by a piecewise-affine function as shown in Figure 11.76.

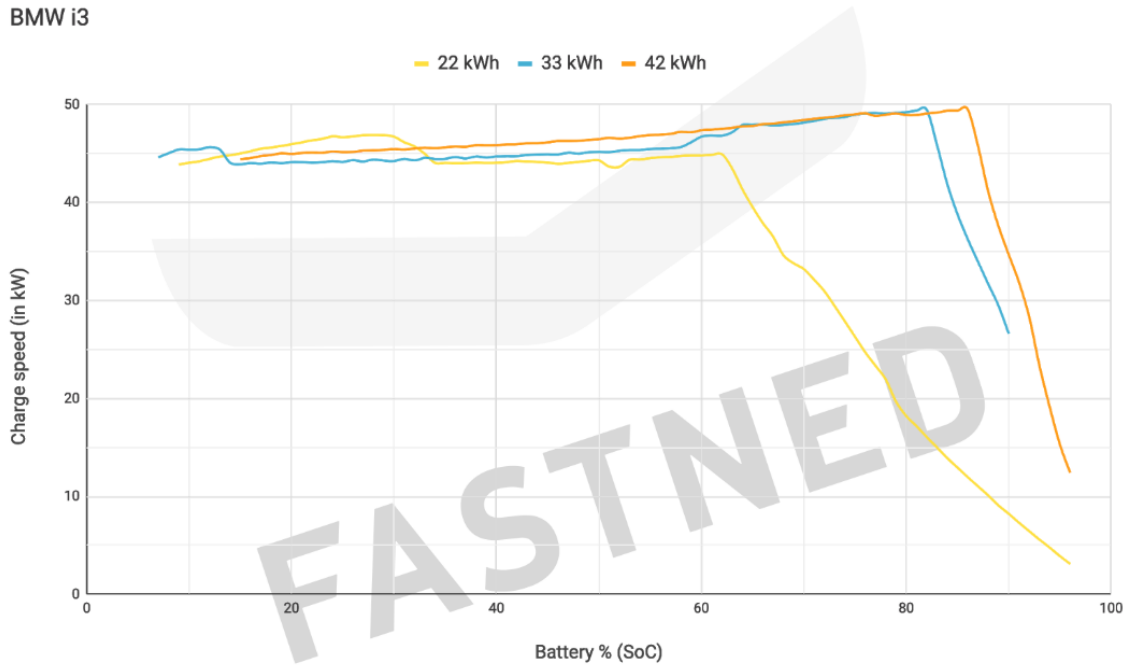


Figure 11.75: Actual charging profile of BMW i3 as a function of vehicle's SoC.

The code for computing the $P_{\mathrm{ch}}(\zeta)$ spline function is given below. Note that the Python implementation uses a shape-preserving piecewise-cubic representation instead of piecewise-linear one.

Matlab

Python

```
function [ Pch ] = setupChargingPower(PchMax)
% Returns piecewise linear representation of charging power as a
% function of vehicle's SoC.
%
% The data is taken from Fastned charging chart available at
% https://support.fastned.nl/hc/en-gb/articles/204784718-Charging-with-a-BMW-i3
%
```

(continues on next page)

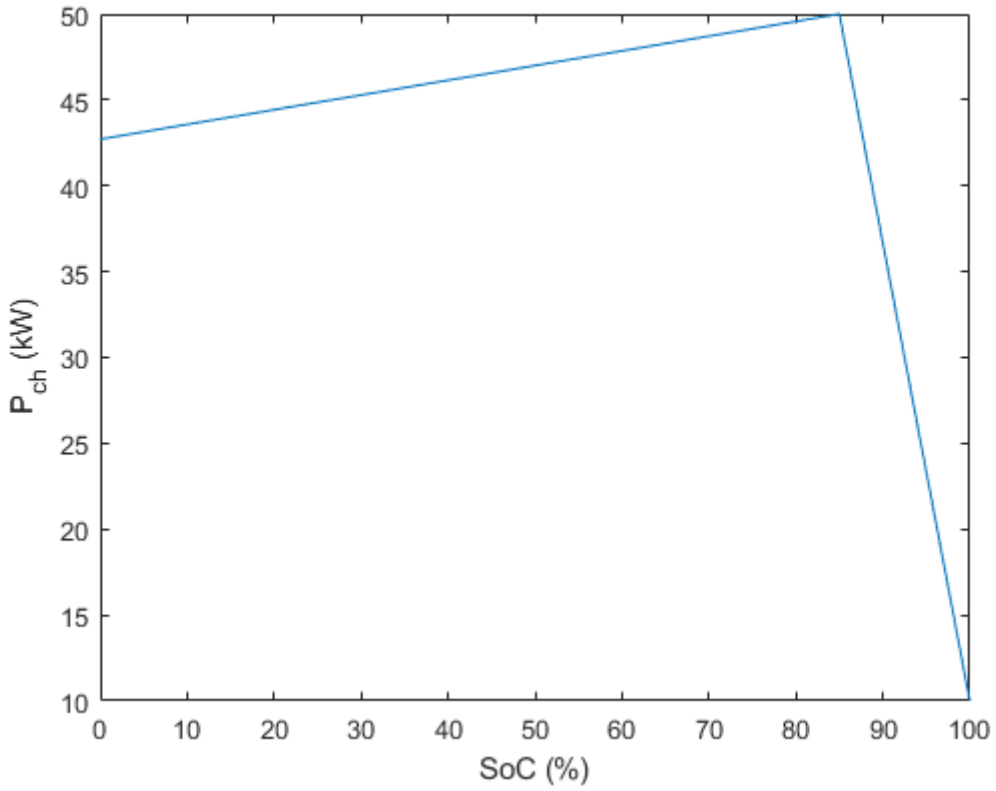


Figure 11.76: Implemented piecewise-affine charging profile for the $42\,\mathrm{kWh}$ vehicle as a function of vehicle's SoC.

(continued from previous page)

```
% NOTE: Python example uses shape-preserving piecewise cubic spline
% approximation ('pchip') which might lead to negligible result
% discrepancies between the two clients when directly compared.
    socSample = [0.15, 0.85, 1.0];
    PchSample = [0.88, 1, 0.2]*PchMax;
    Pch = ForcesInterpolationFit(socSample,PchSample,'linear');
end
```

```
def setupChargingPower(self):
    """
    Returns shape-preserving piecewise cubic spline representation of charging power␣
↪as a function of SoC.

    The data is taken from Fastned charging chart available at https://support.
↪fastned.nl/hc/en-gb/articles/204784718-Charging-with-a-BMW-i3

    NOTE: MATLAB example uses piecewise linear approximation which might lead to␣
↪negligible result discrepancies between the two clients when directly compared.
    """
    socSample = np.array([0.15, 0.85, 1.0])
    PchSample = np.array([0.88, 1, 0.2]) * self.PchMax
    PchSpline = forcespro.modelling.InterpolationFit(socSample, PchSample, 'pchip')

    return PchSpline
```

Clearly, the time dynamics would have to be modified as well:

$$t_{i+1} = t_i + \frac{L_\mathrm{s}}{v_i} + \Delta t_i$$

with $\Delta t_i \approx 0$ for all non-charging spatial steps $i \notin \mathcal{K}$, and $v_i \approx \underline{v}$ for all charging spatial steps $i \in \mathcal{K}$, with $\underline{v} > 0$ being th lower speed bound. The latter is not set to zero for two reasons: 1) it would lead to infeasibility due to v_i being in the denominator; 2) it reflects the average speed across the distance L_s, which includes both the driving and the charging instances, and is therefore greater than zero. This implies that the final vehicle model comprises three states $x = [v, t, \zeta]^\top$ and three control inputs $u = [F_\mathrm{t}, F_\mathrm{b}, \Delta t]^\top$, with the model dynamics described by

$$\begin{aligned} v_{i+1} &= \sqrt{\frac{2\left(F_{\mathrm{t},i} - F_{\mathrm{b},i} - F_{\mathrm{r},i}\right) L_\mathrm{s}}{m_\mathrm{eq}} + v_i^2} \\ t_{i+1} &= t_i + \frac{L_\mathrm{s}}{v_i} + \Delta t_i \\ \zeta_{i+1} &= \zeta_i - \frac{F_{\mathrm{t},i} L_\mathrm{s}}{\eta_i E_\mathrm{cap}} + \frac{P_{\mathrm{ch},i}}{E_\mathrm{cap}} \Delta t_i \end{aligned}$$

Additionally, we will include a slack variable $s \geq 0$ into control vector u, as it is used to relax the upper and lower bounds on vehicle's SoC.

Bounds

In terms of bounds on state variables, we impose upper and lower limits on vehicle's velocity and SoC, dictated by the road speed limits and battery specifications:

$$\begin{aligned} \underline{v} \leq v_i \leq \bar{v}_i \\ \underline{\zeta} - s_i \leq \zeta_i \leq \bar{\zeta} + s_i \end{aligned}$$

The upper speed bound $\bar{v}_i$ is stage-dependent, dictated by the spatial discretization of the road speed limits, whereas the lower speed bound $\underline{v}$ is kept constant across all stages. For charging instances $i \in \mathcal{K}$, the upper speed bound can optionally be set to $\bar{v}_i \approx \underline{v}$, indicating that the vehicle has to reduce speed. The SoC limits $\underline{\zeta}$ and $\bar{\zeta}$ are also stage independent and implemented as soft bounds.

All control variables are nonnegative and upper bounded with appropriate physical and technical limits:

$$\begin{aligned} 0 \leq \Delta t_i \leq \overline{\Delta t}_i \\ 0 \leq F_{\mathrm{t},i} \leq \overline{F}_{\mathrm{t},i} \\ 0 \leq F_{\mathrm{b},i} \leq \overline{F}_\mathrm{b} \end{aligned}$$

Here, $\overline{\Delta t}_i$ denotes maximum permissible charging time and is set to $\overline{\Delta t}_i \approx 0, \forall i \notin \mathcal{K}$, whereas $\overline{F}_{\mathrm{t},i}$ and $\overline{F}_\mathrm{b}$ are vehicle's physical upper bounds on maximum traction and breaking force, respectively. Note that the traction force limit is comprised of the constant friction limit $\overline{F}_\mathrm{t}$ and the hyperbolic traction function of vehicle's velocity $\overline{F}_{\mathrm{t},i}^{\mathrm{hyp}}(v_i)$, i.e.

$$\overline{F}_{\mathrm{t},i} = \min\left(\overline{F}_\mathrm{t}, \overline{F}_{\mathrm{t},i}^{\mathrm{hyp}}(v_i)\right)$$

as depicted in Figure 11.77. The actual function implemented in this example is showcased in Figure 11.78.

The computation of ideal traction hyperbola $\overline{F}_{\mathrm{t,hyp}}(v)$ in the cubic spline form is provided below.

Matlab

Python

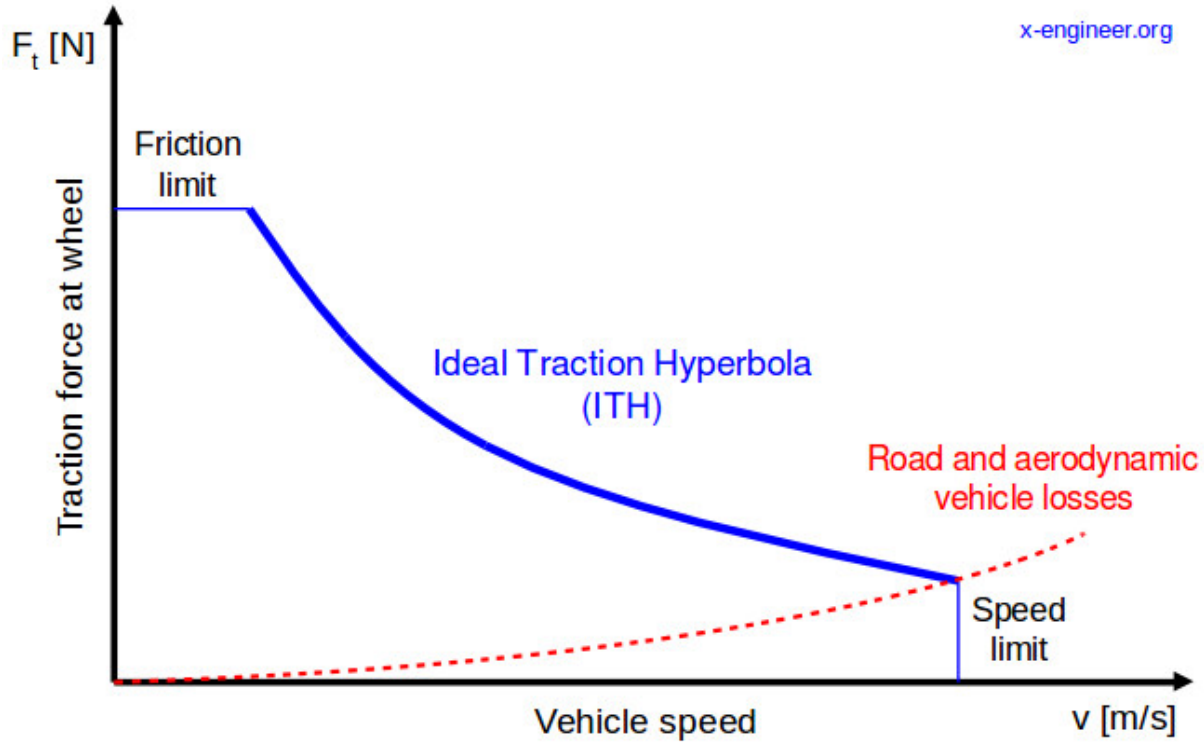


Figure 11.77: Maximum permissible traction force as a function of vehicle's velocity taken from [xEngineer_2D].

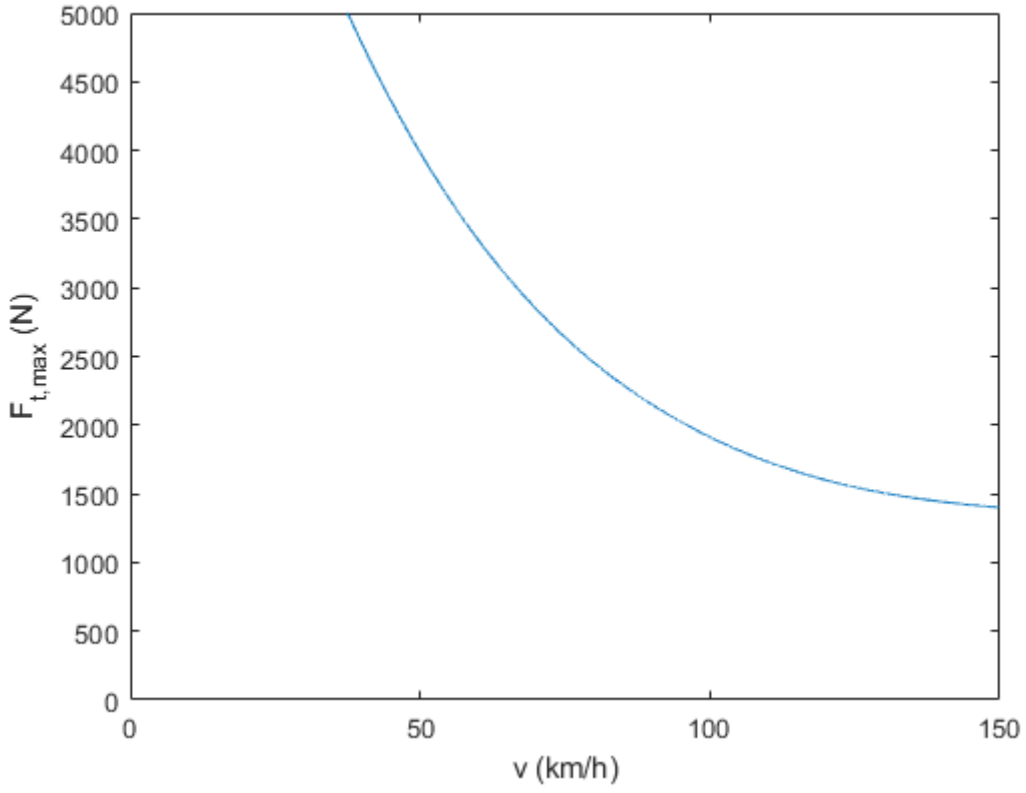


Figure 11.78: Implemented traction force hyperbola as a function of vehicle's velocity.

```
function [ FtMaxHypSpline ] = setupTractionForceHyperbola(vMax,FtMax)
% Returns cubic spline representing maximum traction force as a function of
% vehicle's speed
%
% The data is generated based on the technical article provided at
% https://x-engineer.org/need-gears/
    vSample = [0.25, 0.4, 0.6, 0.8, 1.0]*vMax;
    FtMaxHypSample = [1.0, 0.67, 0.43, 0.32, 0.28]*FtMax;
    FtMaxHypSpline = ForcesInterpolationFit(vSample,FtMaxHypSample);
end
```

```
def setupTractionForceHyperbola(self):
    """
    Returns cubic spline representing motor efficiency as a function of traction␣
↪force.

    The data is generated based on the technical article provided at https://x-
↪engineer.org/need-gears/
    """
    vSample = np.array([0.25, 0.4, 0.6, 0.8, 1.0]) * self.kmh2ms(self.vMax)
    FtMaxHypSample = np.array([1.0, 0.67, 0.43, 0.32, 0.28]) * self.FtMax
    FtMaxHypSpline = forcespro.modelling.InterpolationFit(vSample, FtMaxHypSample)

    return FtMaxHypSpline
```

MPC Formulation

A complete MPC problem formulation can thus be described as follows:

$$
\begin{aligned}
\text{minimize} \quad & t_N + \sum_{i=0}^{N-1} \left(\omega_\mathrm{s} s_i + \omega_\mathrm{e} F_{\mathrm{t},i}^2 + \omega_\mathrm{b} F_{\mathrm{b},i}^2 \right) \\
\text{subject to} \quad & v_{i+1} = \sqrt{\frac{2L_\mathrm{s}}{m_\mathrm{eq}} \left(F_{\mathrm{t},i} - F_{\mathrm{b},i} - F_{\mathrm{r},i} \right) + v_i^2} \\
& t_{i+1} = t_i + L_\mathrm{s} v_i^{-1} + \Delta t_i \\
& \zeta_{i+1} = \zeta_i - \frac{L_\mathrm{s}}{\eta_i E_\mathrm{cap}} F_{\mathrm{t},i} + \frac{P_{\mathrm{ch},i}}{E_\mathrm{cap}} \Delta t_i \\
& \underline{v} \leq v_i \leq \bar{v}_i \\
& \underline{\zeta} - s_i \leq \zeta_i \leq \bar{\zeta} + s_i \\
& 0 \leq \Delta t_i \leq \overline{\Delta t}_i \\
& 0 \leq F_{\mathrm{t},i} \leq \overline{F}_{\mathrm{t},i} \\
& 0 \leq F_{\mathrm{b},i} \leq \overline{F}_\mathrm{b} \\
& 0 \leq s_i \\
& x_0 = x_\mathrm{init}
\end{aligned}
$$

with the objective function explained in detail in the subsequent sections.

Model Parameters

As stated previously, in this example we focus on a highest energy capacity model of BMW i3, defined by the following parameters obtained from [BMWi3_2D]:

$$
\begin{aligned}
m_{\mathrm{v}} &= 1345\,\mathrm{kg} \\
e_{\mathrm{I}} &= 1.06 \\
A_{\mathrm{f}} &= 2.38\,\mathrm{m}^2 \\
\rho_{\mathrm{a}} &= 1.206\,\mathrm{kg/m}^3 \\
c_{\mathrm{a}} &= 0.29 \\
c_{\mathrm{r}} &= 0.01 \\
\overline{F}_{\mathrm{t}} &= 5\,\mathrm{kN} \\
\overline{F}_{\mathrm{b}} &= 10\,\mathrm{kN} \\
E_{\mathrm{cap}} &= 37.9\,\mathrm{kWh} \\
\bar{P}_{\mathrm{ch}} &= 50\,\mathrm{kW} \\
\underline{v} &= 30\,\mathrm{km/h} \\
\bar{v} &= 180\,\mathrm{km/h} \\
\underline{\zeta} &= 0.1 \\
\bar{\zeta} &= 0.9
\end{aligned}
$$

Matlab

Python

```
function [ params ] = defineVehicleParameters(trip)
% Returns vehicle parameters
    % Constant vehicle parameters (not to be changed)
    % The data is obtained from "Technical specifications of the BMW i3
    % (120 Ah)" valid from 11/2018 and available at
    % https://www.press.bmwgroup.com/global/article/detail/T0285608EN/

    % Gravitational acceleration (m/s^2)
    params.g = 9.81;
    % Vehicle mass (kg)
    params.mv = 1345;
    % Mass factor (-)
    params.eI = 1.06;
    % Equivalent mass (kg)
    params.meq = (1+params.eI)*params.mv;
    % Projected frontal area (m^2)
    params.Af = 2.38;
    % Air density (kg/m^3)
    params.rhoa = 1.206;
    % Air drag coefficient (-)
    params.ca = 0.29;
    % Rolling resistance coefficient (-)
    params.cr = 0.01;
    % Maximum traction force (N)
    params.FtMax = 5e3;
    % Minimum traction force (N)
    params.FtMin = 0;
    % param.FtMin = -1.11e3;
    % Maximum breaking force (N)
```

(continues on next page)

(continued from previous page)

```
    params.FbMax = 10e3;
    % Battery's energy capacity (Wh)
    % BMW i3 60Ah - 18.2kWh; BMW i3 94Ah - 27.2kWh; BMW i3 120Ah - 37.9kWh
    params.Ecap = 37.9e3;
    % Maximum permissible charging time (s)
    params.TchMax = 3600;
    % Charging power (W)
    % 7.4 kW on-board charger on IEC Combo AC, optional 50 kW Combo DC
    params.PchMax = 50e3;
    % Power dissipation factor (-) (adjusted for Munich - Cologne trip in
    % order to achieve realistic BMW i3 range of ~260km)
    if trip.type == 1
        params.pf = 1;
    else
        params.pf = 0.4;
    end
    % Minimum vehicle velocity (km/h)
    params.vMin = 30;
    % Maximum vehicle velocity (km/h)
    params.vMax = 150;
    % Minimum SoC (-)
    params.SoCmin = 0.1;
    % Maximum SoC (-)
    params.SoCmax = 0.9;

    % Variable vehicle parameters
    % Motor efficiency (-)
    params.eta = setupMotorEfficiency();
    % Charging power (W)
    params.Pch = setupChargingPower(params.PchMax);
    % Traction force hyperbola (upper limit) (N)
    params.FtMaxHyp = setupTractionForceHyperbola(kmh2ms(params.vMax), params.FtMax);
end
```

```
class VehicleParameters:
"""Constant and variable vehicle parameters (not to be changed)"""
def __init__(self, trip):
    # Constant vehicle parameters
    # The data is obtained from "Technical specifications of the BMW i3 (120 Ah)"␣
↪valid from 11/2018 and available at https://www.press.bmwgroup.com/global/article/
↪detail/T0285608EN/
    # ----------------------------

    # Gravitational acceleration (m/s^2)
    self.g = 9.81
    # Vehicle mass (kg)
    self.mv = 1345
    # Mass factor (-)
    self.eI = 1.06
    # Equivalent mass (kg)
    self.meq = (1 + self.eI) * self.mv
    # Projected frontal area (m^2)
    self.Af = 2.38
    # Air density (kg/m^3)
    self.rhoa = 1.206
    # Air drag coefficient (-)
```

(continues on next page)

(continued from previous page)

```
    self.ca = 0.29
    # Rolling resistance coefficient (-)
    self.cr = 0.01
    # Maximum traction force (N)
    self.FtMax = 5e3
    # Minimum traction force (N)
    self.FtMin = 0
    # self.FtMin = -1.11e3
    # Maximum breaking force (N)
    self.FbMax = 10e3
    # Battery's energy capacity (Wh)
    # BMW i3 60Ah - 18.2kWh BMW i3 94Ah - 27.2kWh BMW i3 120Ah - 37.9kWh
    self.Ecap = 37.9e3
    # Maximum permissible charging time (s)
    self.TchMax = 3600
    # Charging power (W)
    # 7.4 kW on-board charger on IEC Combo AC, optional 50 kW Combo DC
    self.PchMax = 50e3
    # Power dissipation factor (-) (adjusted for Munich - Cologne trip in order to␣
↪achieve realistic BMW i3 range of ~260km)
    if trip.type == 1:
        self.pf = 1
    else:
        self.pf = 0.4
    # Minimum vehicle velocity (km/h)
    self.vMin = 30
    # Maximum vehicle velocity (km/h)
    self.vMax = 150
    # Minimum SoC (-)
    self.SoCmin = 0.1
    # Maximum SoC (-)
    self.SoCmax = 0.9

    # Variable vehicle parameters
    # ---------------------------

    # Motor efficiency (-)
    self.eta = self.setupMotorEfficiency()
    # Charging power (W)
    self.Pch = self.setupChargingPower()
    # Traction force hyperbola (upper limit) (N)
    self.FtMaxHyp = self.setupTractionForceHyperbola()
```

Trip parameters are defined by the user. Here we focus on two benchmark studies: 1) a $50\,\mathrm{km}$ long trip with five EV charging stations and low initial SoC; and 2) an actual $573\,\mathrm{km}$ long Munich - Cologne trip with four EV charging stations and fully charged vehicle.

Matlab

Python

```
%% Trip parameters (provided by the user)
% Note: Accuracy and feasibility of the solution are not guaranteed when
% the `trip.reduceSpeed` flag is disabled due to inherent mixed-integer
% nature of the problem!

% Trip type (0 - long (Munich - Cologne); 1 - short)
```

(continues on next page)

(continued from previous page)

```
trip.type = 1;

switch trip.type
    case 1
        % Spatial discretization step (km)
        trip.Ls = 1;
        % Trip distance (km)
        trip.dist = 50;
        % EV charging station locations (km)
        trip.kmPosCh = [8, 18, 30, 40, 45];
        % Force vehicle to reduce speed at charging locations (1 - yes; 0 - no)
        trip.reduceSpeed = 0;
        % Initial vehicle speed (km/h)
        trip.initSpeed = 30;
        % Initial SoC [0, 1]
        trip.initSoC = 0.25;
    case 0
        % Spatial discretization step (km)
        trip.Ls = 1;
        % Trip distance (km)
        trip.dist = 573;
        % EV charging station locations (km)
        trip.kmPosCh = [110, 150, 250, 375];
        % Force vehicle to reduce speed at charging locations (1 - yes; 0 - no)
        trip.reduceSpeed = 1;
        % Initial vehicle speed (km/h)
        trip.initSpeed = 30;
        % Initial SoC [0, 1]
        trip.initSoC = 0.75;
end

%% Check trip setup
trip = checkTripSetup(trip);

function [ trip ] = checkTripSetup(trip)
% Checks validity of provided trip setup and modifies it if needed.
    % Trip type has to be 0 or 1
    if trip.type~=0 && trip.type~=1
        error('Incorrect input: trip type should be either 0 or 1.');
    end
    % Force vehicle to reduce speed at charging locations when running the
    % long trip scenario
    if trip.type==0
        trip.reduceSpeed = 1;
    end
end
```

```
class TripParameters:
    """Trip parameters (to be defined by the user)"""
    def __init__(self):
        # Note: Accuracy and feasibility of the solution are not guaranteed when the␣
↪`trip.reduceSpeed` flag is disabled due to inherent mixed-integer nature of the␣
↪problem!

        # Trip type (0 - long (Munich - Cologne); 1 - short)
        self.type = 1
```

(continues on next page)

(continued from previous page)

```
        if self.type == 1:
            # Spatial discretization step (km)
            self.Ls = 1
            # Trip distance (km)
            self.dist = 50
            # EV charging station locations (km)
            self.kmPosCh = np.array([8, 18, 30, 40, 45])
            # Force vehicle to reduce speed at charging locations
            self.reduceSpeed = False
            # Initial vehicle speed (km/h)
            self.initSpeed = 30
            # Initial SoC [0, 1]
            self.initSoC = 0.25
        else:
            # Spatial discretization step (km)
            self.Ls = 1
            # Trip distance (km)
            self.dist = 573
            # EV charging station locations (km)
            self.kmPosCh = np.array([110, 150, 250, 375])
            # Force vehicle to reduce speed at charging locations
            self.reduceSpeed = True
            # Initial vehicle speed (km/h)
            self.initSpeed = 30
            # Initial SoC [0, 1]
            self.initSoC = 0.75

        self.checkTripSetup()

    def checkTripSetup(self):
        """Checks validity of provided trip setup and modifies it if needed"""
        # Trip type has to be 0 or 1
        assert self.type in [0, 1], 'Incorrect input: trip type should be 0 or 1.'
        # Force vehicle to reduce speed at charging locations when running shrinking␣
↪horizon MPC or the long trip scenario
        if self.type==0:
            self.reduceSpeed = True
```

You can find the code of this example to try it out for yourself in the `examples` folder that comes with your client.

11.20.2 Defining the MPC Problem

Scaling

Since the physical quantities of this example (e.g. SoC or traction force) have vastly different magnitudes, it is a numerically sensitive optimization problem. Hence, we utilize scaling to keep the optimization variables (i.e. scaled down physical quantities) in a similar range numerically. It is enough to approximately set the scaling vector equal to the order of magnitudes of the physical states and inputs. For this purpose we define a scaling vector, which will be constant throughout the example:

Matlab

Python

```
% Assume variable ordering zi = [u{i}; x{i}] for i=1...N
% zi = [slack(i); Ft(i); Fb(i); deltaTch(i); v(i); t(i); SoC(i)]
% pi = [vMax(i); vMin(i); alpha(i); TchMax(i); Ls(k)] #k - iteration No.

%% Scaling Vector
% Scaling factors approximately set equal to order of magnitude of
% the corresponding physical states and inputs
scalingVec = [1, 1e4, 1e4, 1e4, 1e2, 1e4, 1].';
```

```
"""
Assume variable ordering zi = [u{i}, x{i}] for i=1...N
zi = [slack{i}, Ft{i}, Fb{i}, deltaTch{i}, v{i}, t{i}, SoC{i}]
pi = [vMax{i}, vMin{i}, alpha{i}, TchMax{i}, Ls{k}] #k - iteration No.
"""

# Scaling
# Scaling factors approximately set equal to order of magnitude of
# the corresponding physical states and inputs
scalingVec = np.array([1, 1e4, 1e4, 1e4, 1e2, 1e4, 1])
```

Objective function

The primary goal is to minimize the total driving time, i.e. minimize terminal variable t_N at the last stage $i = N$:

$$f(z, p_N) = z(6)$$

In addition, we want to minimize the stage cost associated with slack variable $s, \forall i \in \mathcal{I}$:

$$f(z, p_i) = \omega_\mathrm{s} z(1)$$

where ω_s is an appropriately high weight factor.

Optionally, one could opt for ensuring comfortable driving operation and pursuing energy economy improvements. In such case, additional terms could be included in the stage cost function, e.g.

$$f(z, p_i) = \omega_\mathrm{s} z(1) + \omega_\mathrm{e} z(2)^2 + \omega_\mathrm{b} z(3)^2$$

The second term imposes a penalty on consumed energy during driving by reducing the applied traction force, whereas the third term minimizes the total braking force. The corresponding weight factors ω_e and ω_b provide a trade-off between different objectives. Furthermore, one could also penalize jerking, i.e. a change in the traction force between consecutive steps, using another slack variable.

In the code, we scale the optimization variables back to their physical values before multiplying them with the cost weightings. At the end, we scale down the entire cost down again by a scalar **costScaling** factor in order to have limited cost magnitudes. We could have also omitted the scaling back up to the physical quantities in the cost function by choosing different cost weighting factors. The cost functions are coded in MATLAB and Python as follows:

Matlab

Python

```
% Objective function
costScaling = 1e3;
model.objective = @(z) objective(z, scalingVec, costScaling);
```

(continues on next page)

(continued from previous page)

```
model.objectiveN = @(z) objectiveN(z, scalingVec, costScaling);

function [ stageCost ] = objective(z, scalingVec, costScaling)
    z = z.*scalingVec;

    R = [ 1e-7,  0;
          0,    1e-6];
    slackCostFactor = 1e6;
    stageCost = slackCostFactor*z(1) + [z(2); z(3)]'*R*[z(2); z(3)];

    stageCost = stageCost/costScaling;
end

function [ terminalCost ] = objectiveN(z, scalingVec, costScaling)
    z = z.*scalingVec;

    terminalCost = z(6);

    terminalCost = terminalCost/costScaling;
end
```

```
# Objective function
costScaling = 1e3
model.objective = lambda z: objective(z, scalingVec, costScaling)
model.objectiveN = lambda z: objectiveN(z, scalingVec, costScaling)

def objective(z, scalingVec, costScaling):
    z *= scalingVec

    R = np.diag([1e-7, 1e-6])
    slackCostFactor = 1e6

    stageCost = slackCostFactor * z[0] + casadi.horzcat(z[1], z[2]) @ R @ casadi.
↪vertcat(z[1], z[2])

    stageCost /= costScaling

    return stageCost

def objectiveN(z, scalingVec, costScaling):
    z *= scalingVec

    terminalCost = z[5]

    terminalCost /= costScaling

    return terminalCost
```

Equality constraints

The equality constraints `model.eq` in this example results from the vehicle's dynamics and energetics model given above. Importantly, we have to scale back up the optimization variables to the physical quantities at the very beginning of the function. With the physical quantities we calculate the dynamics equation as usual and then we scale back down the next state (or state derivative in case of continuous dynamics). Note that we are using the fitted 2D-spline

in the dynamics equation for the SoC. The code is implemented as follows:

Matlab

Python

```
%% Problem dimensions
nx = 3;
nu = 4;
np = 5;
nh = 6;

model.N = round(trip.dist/trip.Ls); % horizon length
model.nvar = nx + nu;               % number of variables
model.neq  = nx;                    % number of equality constraints
model.nh   = nh;                    % number of inequality constraints
model.npar = np;                    % number of runtime parameters

% Dynamics, i.e. equality constraints
model.eq = @(z,p) dynamics(z, p, param, scalingVec, nu, nx);

model.E = [zeros(nx,nu),eye(nx)];

% Initial and final conditions
model.xinitidx = nu+1:nu+nx;

function [xNext] = dynamics(z, p, param, scalingVec, nu, nx)
    z = z.*scalingVec;

    xNext = [sqrt(ForcesMax(2*p(5)/param.meq*(z(2) - z(3) - param.cr*param.mv*param.
↪g*cos(p(3)) - param.mv*param.g*sin(p(3)) - 0.5*param.ca*param.Af*param.rhoa*z(5)^2)␣
↪+ z(5)^2, 1e-5));
            z(6) + p(5)/z(5) + z(4);
            z(7) - param.pf*p(5)*z(2)/(3600*param.eta(z(5), z(2))*param.Ecap) + param.
↪Pch(z(7))*z(4)/(3600*param.Ecap)];

    xNext = xNext./scalingVec(nu+1:nu+nx);
end
```

```
# Problem dimensions
nx = 3
nu = 4
npar = 5
nh = 7

model = forcespro.nlp.SymbolicModel()
model.N = round(trip.dist / trip.Ls); # horizon length
assert model.N % 1 == 0, 'Incorrect input: trip distance should be an exact multiple␣
↪of the spatial discretization step'
model.nvar = nx + nu;               # number of variables
model.neq  = nx;                    # number of equality constraints
model.nh   = nh;                    # number of inequality constraints
model.npar = npar;                    # number of runtime parameters

# Dynamics, i.e. equality constraints
model.eq = lambda z, p: dynamics(z, p, param, scalingVec, nx, nu)

model.E = np.concatenate([np.zeros((nx, nu)), np.eye(nx)], axis=1)
```

(continues on next page)

(continued from previous page)

```
# Initial and final conditions
model.xinitidx = np.arange(nu, nu + nx);

def dynamics(z, p, param, scalingVec, nx, nu):
    z *= scalingVec

    xNext = casadi.vertcat( np.sqrt(forcespro.modelling.smooth_max(2 * p[4] / param.
↪meq * (z[1] - z[2] - param.cr * param.mv * param.g * np.cos(p[2]) - param.mv *␣
↪param.g * np.sin(p[2]) - 0.5 * param.ca * param.Af * param.rhoa * z[4]**2) +␣
↪z[4]**2, 1e-5)),
                            z[5] + p[4] / z[4] + z[3],
                            z[6] - param.pf * p[4] * z[1] / (3600 * param.eta(z[4],␣
↪z[1])* param.Ecap) + param.Pch(z[6]) * z[3] / (3600 * param.Ecap))


    xNext /= scalingVec[nu:nu+nx]

    return xNext
```

The use of *smooth maximum* approximation (see section Section 20.3) ensures a positive square root term and thus a feasible solution.

Inequality constraints

Since our optimization variables are scaled, it is crucial to scale down the physical box-bound constraints.

The nonlinear inequality constraints `model.ineq` comprise all decision variable bounds described previously. For the nonlinear inequality constraints, we scale back up the optimization variables to physical quantities and define the physical constraints as usual. Optionally, we scale down again the entire nonlinear constraints by their approximate order of magnitudes for additional numerical stability. We keep in mind that if the bounds of the nonlinear inequality constraints `model.hl` and `model.hu` were not `-inf` or `0`, they would have been affected by the scaling as well. For purposes of model feasibility, two additional trivial constraints pertaining to the vehicle velocity's square root term are included in the model, as shown in the code-snippets.

Matlab

Python

```
% Inequality constraints
% Upper/lower variable bounds lb <= z <= ub
%                             inputs                          |               states
%           slack       Ft       Fb            Tch                     v           t ␣
↪    SoC
model.lb = [0,     param.FtMin,  0.,           0.,           kmh2ms(param.vMin),  0.,
↪    0.]./scalingVec.';
model.ub = [0.1,   param.FtMax,  param.FbMax,  param.TchMax,  kmh2ms(param.vMax), ␣
↪+inf,  1.]./scalingVec.';

% Nonlinear inequalities hl <= h(z,p) <= hu
model.ineq = @(z,p) ineq(z, p, param, scalingVec);

% Upper/lower bounds for inequalities
model.hu = [0,    0,    0,    0,    0,    0];
```

(continues on next page)

(continued from previous page)

```
model.hl = [-inf, -inf, -inf, -inf, -inf, -inf];

function [h] = ineq(z, p, param, scalingVec)
    z = z.*scalingVec;

    h = [z(2) - param.FtMaxHyp(z(5));
        z(4) - p(4);
        z(7) - param.SoCmax - z(1);
        param.SoCmin - z(7) - z(1);
        2*p(5)/param.meq*(z(2) - z(3) - param.cr*param.mv*param.g*cos(p(3)) - param.
↪mv*param.g*sin(p(3)) - 0.5*param.ca*param.Af*param.rhoa*z(5)^2) + z(5)^2 -␣
↪kmh2ms(ForcesMax((1-z(4))*p(1), param.vMin+1))^2;
        kmh2ms(p(2))^2 - (2*p(5)/param.meq*(z(2) - z(3) - param.cr*param.mv*param.
↪g*cos(p(3)) - param.mv*param.g*sin(p(3)) - 0.5*param.ca*param.Af*param.rhoa*z(5)^2)␣
↪+ z(5)^2)];

    h = h./[scalingVec(2); scalingVec(4); scalingVec(7); scalingVec(7); scalingVec(5);
↪ scalingVec(5)];
end
```

```
# Inequality constraints
# Upper/lower variable bounds lb <= z <= ub
#                            inputs                                |              ␣
↪states
#                  slack       Ft     Fb           Tch                    v          ␣
↪ t      SoC
model.lb = np.array([0.,  param.FtMin, 0.,          0.,           kmh2ms(param.vMin),␣
↪ 0.,      0.]) / scalingVec
model.ub = np.array([0.1, param.FtMax, param.FbMax, param.TchMax, kmh2ms(param.vMax),␣
↪ np.inf,  1.]) / scalingVec

# Nonlinear inequalities hl <= h(z,p) <= hu
model.ineq = lambda z,p: ineq(z, p, param, scalingVec)

# Upper/lower bounds for inequalities
model.hu = np.array([0,       0,        0,       0,      0,       0])
model.hl = np.array([-np.inf, -np.inf, -np.inf, -np.inf, -np.inf, -np.inf])

def ineq(z, p, param, scalingVec):
    z *= scalingVec

    h = casadi.vertcat(z[1] - param.FtMaxHyp(z[4]),
                       z[3] - p[3],
                       z[6] - param.SoCmax - z[0],
                       param.SoCmin - z[6] - z[0],
                       2 * p[4] / param.meq * (z[1] - z[2] - param.cr * param.mv *␣
↪param.g * np.cos(p[2]) - param.mv * param.g * np.sin(p[2]) - 0.5 * param.ca * param.
↪Af * param.rhoa * z[4]**2) + z[4]**2 - kmh2ms(forcespro.modelling.smooth_max((1 -␣
↪z[3]) * p[0], param.vMin + 1))**2,
                       kmh2ms(p[1])**2 - (2 * p[4] / param.meq * (z[1] - z[2] - param.
↪cr * param.mv * param.g * np.cos(p[2]) - param.mv * param.g * np.sin(p[2]) - 0.5 *␣
↪param.ca * param.Af * param.rhoa * z[4]**2) + z[4]**2))


    h /= np.array([scalingVec[1], scalingVec[3], scalingVec[6], scalingVec[6],␣
↪scalingVec[4], scalingVec[4]])
```

(continues on next page)

(continued from previous page)

```
    return h
```

Generating the FORCESPRO NLP solver

To generate a suitable NLP solver for our MPC problem one needs to provide the *model* and *codeoptions*. The *model* has been populated above and we now specify the desired *codeoptions* and generate the solver by calling *FORCES_NLP*. Importantly, in order to use the provided 2D-spline functionality, the automatic differentiation tool must be CasADi. Furthermore, the CasADi 2D-splines require the usage of MX CasADi-variables, which can be set in the codeoptions. The following code-snippets show how this can be done:

Matlab

Python

```
% Generate FORCESPRO solver

% Define solver options
codeoptions = getOptions('FORCESNLPsolver');
codeoptions.printlevel = 0;
codeoptions.nlp.compact_code = 1;
codeoptions.maxit = 300

% 2D-splines require the MX CasADi variables
codeoptions.nlp.ad_expression_class = 'MX'

% Generate code
FORCES_NLP(model, codeoptions);
```

```
# Generate FORCESPRO solver
# -------------------------

# Define solver options
codeoptions = forcespro.CodeOptions("FORCESNLPsolver")
codeoptions.printlevel = 0
codeoptions.nlp.compact_code = 1
codeoptions.maxit = 300

# 2D-splines require the MX CasADi variables
codeoptions.nlp.ad_expression_class = "MX"

# Generate code
solver = model.generate_solver(codeoptions)
```

Calling the solver

Once the solver has been generated it needs to be provided with initial and runtime parameters. Importantly, we need to scale down the physical initial state and scale back up the FORCESPRO solution to the physical quantities. If a physically valid trajectory would be utilized as an initial guess, we would need to scale the initial guess down as well. Here we use 0 as initial guess for simplicity's sake. In this example we are running a single full-horizon snapshot instead of a traditional rolling-horizon MPC, as follows:

Matlab

Python

```
function sim = runSimulation(trip, param, model, scalingVec)
% Defines initial and stage-dependent runtime parameters and solves the
% problem
    %% Define spatially distributed (i.e. stage-dependent) parameters
    % Road speed limits and slope angles
    [vMaxRoad, ~, vMinRoad, roadSlope] = setupRoadParameters(param.vMin, trip);

    % Maximum permissible charging time
    deltaTchMax = setupChargingTime(param.TchMax, trip);

    %% Initialize the problem
    problem.x0 = zeros(model.N*model.nvar,1);
    sim.kMax = trip.dist/trip.Ls;
    nx = model.neq;
    nu = model.nvar - model.neq;
    np = model.npar;

    if trip.initSpeed < param.vMin || trip.initSpeed > param.vMax
        v1 = kmh2ms(param.vMin);
        warning('Initial vehicle speed is outside of the speed limits. It will be set␣
↪to the minimum speed limit at t=0.')
    else
        v1 = kmh2ms(trip.initSpeed);
    end

    if trip.initSoC < param.SoCmin || trip.initSoC > param.SoCmax
        SoC1 = 0.5;
        warning('Initial vehicle state-of-charge is outside of the limits. It will be␣
↪set to 50% at t=0.')
    else
        SoC1 = trip.initSoC;
    end

    % X(1) = [v(1); t(1); SoC(1)]
    problem.xinit = [v1; 0; SoC1]./scalingVec(nu+1:nu+nx);

    %% Solve the problem

    % Set runtime parameters
    problem.all_parameters = zeros(np*model.N,1);
    for i = 1:model.N
        problem.all_parameters((i-1)*np+1:i*np) = [vMaxRoad(i); vMinRoad(i);␣
↪roadSlope(i*trip.Ls*1e3); deltaTchMax(i); trip.Ls*1e3];
    end

    % Call solver
    [solverout,exitflag,info] = FORCESNLPsolver(problem);
    sim.exitflag = exitflag;

    % Extract state and control vector
    if exitflag == 1
        sim.Z = unpackStruct(solverout,model.nvar).*scalingVec.';
        sim.solvetime = info.solvetime;
        sim.iters = info.it;
    else
```

(continues on next page)

(continued from previous page)

```
        error(['Solver terminated with exitflag ', num2str(exitflag)]);
    end

    sim = displayResults(sim);
end
```

```
def runSimulation(trip, param, model, scalingVec, solver):
    """Defines initial and stage-dependent runtime parameters and solves the problem""
↪"
    # Define spatially distributed (i.e. stage-dependent) parameters
    # ---------------------------------------------------------------

    # Road speed limits and slope angles
    vMaxRoad, vMaxRoadStrict, vMinRoad, roadSlope = setupRoadParameters(param.vMin,␣
↪trip)
    sim = {"vMaxRoad": vMaxRoad, "vMaxRoadStrict": vMaxRoadStrict, "vMinRoad":␣
↪vMinRoad, "roadSlope": roadSlope}

    # Maximum permissible charging time
    deltaTchMax = setupChargingTime(param.TchMax, trip);

    # Initialize the problem
    # ----------------------

    x0 = np.zeros(model.N * model.nvar)
    problem = {"x0": x0}
    kMax = int(trip.dist / trip.Ls)
    sim["kMax"] = kMax
    nx = model.neq
    nu = model.nvar - model.neq
    npar = model.npar

    assert trip.initSpeed >= param.vMin or trip.initSpeed <= param.vMax, 'Initial␣
↪vehicle speed is outside of the speed limits.'
    assert trip.initSoC >= param.SoCmin or trip.initSoC <= param.SoCmax, 'Initial␣
↪vehicle state-of-charge is outside of the limits.'

    # X(1) = [v(1); t(1); SoC(1)]
    problem["xinit"] = np.array([kmh2ms(trip.initSpeed), 0, trip.initSoC]) /␣
↪scalingVec[nu:nu+nx]

    # Solve the problem
    # -----------------

    # Set runtime parameters
    problem["all_parameters"] = np.zeros(npar * model.N)
    for i in range(model.N):
        problem["all_parameters"][i * npar : (i + 1) * npar] = np.array([vMaxRoad[i],␣
↪vMinRoad[i], roadSlope((i + 1) * trip.Ls * 1e3).item(), deltaTchMax[i], trip.Ls *␣
↪1e3])

    # Call solver
    solverout, exitflag, info = solver.solve(problem)
    assert exitflag == 1, f'Solver terminated with exitflag {exitflag}.'

    # Extract state and control vector
```

(continues on next page)

(continued from previous page)

```
    sim["exitflag"] = exitflag
    sim["Z"] = unpackDict(solverout) * scalingVec
    sim["solvetime"] = info.solvetime
    sim["iters"] = info.it

    sim = displayResults(sim)

    return sim
```

Results

We consider two trips of different lengths previously described in section Section 11.20.1 with three benchmark studies conducted for the shorter trip and two studies for the longer trip:

1. a short trip of $50\,\mathrm{km}$ with arbitrarily generated road profile, 5 charging stations along the road, and vehicle's initial SoC of: a) $75\,\%$; b) $50\,\%$; and c) $25\,\%$.
2. a long trip of $573\,\mathrm{km}$ representing a Munich - Cologne trip with actual road profile, vehicle's initial SoC of $90\,\%$, and: a) 4 charging stations along the road; and b) 2 charging stations along the road.

Studies **1a - 1c** have the same allocation of charging stations, with chargers placed at $8\,\mathrm{km}$, $18\,\mathrm{km}$, $30\,\mathrm{km}$, $40\,\mathrm{km}$ and $45\,\mathrm{km}$ marks. Furthermore, we allow **optional** charging for all three studies. On the other hand, a Munich - Cologne trip **2a** considers four chargers placed at $110\,\mathrm{km}$, $150\,\mathrm{km}$, $250\,\mathrm{km}$ and $375\,\mathrm{km}$, while trip **2b** has two available chargers at $150\,\mathrm{km}$ and $375\,\mathrm{km}$. In contrast to the short trip, here we impose **mandatory** charging requirement. The most notable trip metrics for all five studies are showcased in Table 11.2.

Table 11.2: Summary of vehicle performance for different trip parameters and initial SoC levels.

Trip	Distance	SoC (t=0)	# stations	# stops	Total time	Charge time
1a	50 km	75 %	5	0	34.14 min	0 min
1b	50 km	50 %	5	1	37.92 min	2.11 min
1c	50 km	25 %	5	2	52.09 min	15.12 min
2a	573 km	90 %	4	4	359.35 min	79.58 min
2b	573 km	90 %	2	2	362.97 min	80.41 min

The simulation results for studies **1a** and **1b** are depicted in Figure 11.79 and Figure 11.80, respectively. Since the vehicle has sufficiently high SoC, no charging is needed in first case and only a tiny amount of charging is required in the latter case. However, in the second trip the vehicle is forced to reduce its velocity below the speed limit in order to preserve energy and complete the trip with sufficient battery charge, which leads to a higher trip time.

A low initial SoC in study **1c** forces the vehicle to charge, as shown in Figure 11.81, which leads to drastically higher trip time. The EV decides to stop at first two charging stops and charge just enough to complete the trip with the SoC at the lower bound.

The longer trip studies with **mandatory** charging presented in Figure 11.82 and Figure 11.83 again indicate that the EV will complete the trip at the minimum permissible SoC in order to minimize the total charging time. The results also suggest that the vehicle will intentionally reduce speed in order to preserve energy, thus achieving a trade-off between driving and charging time. The availability of only 2 chargers in study **2b** forces the vehicle to drastically reduce speed midway through the trip.

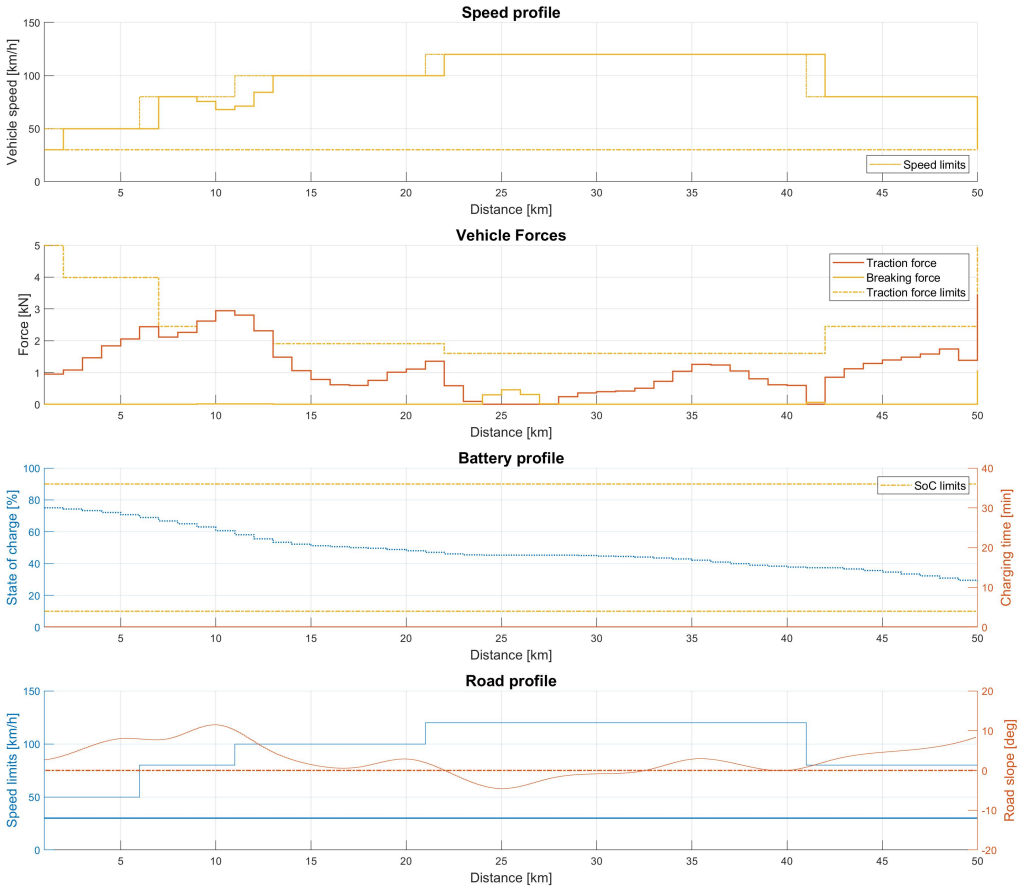


Figure 11.79: Vehicle performance on a short trip with 5 charging stations and 75 % initial SoC.

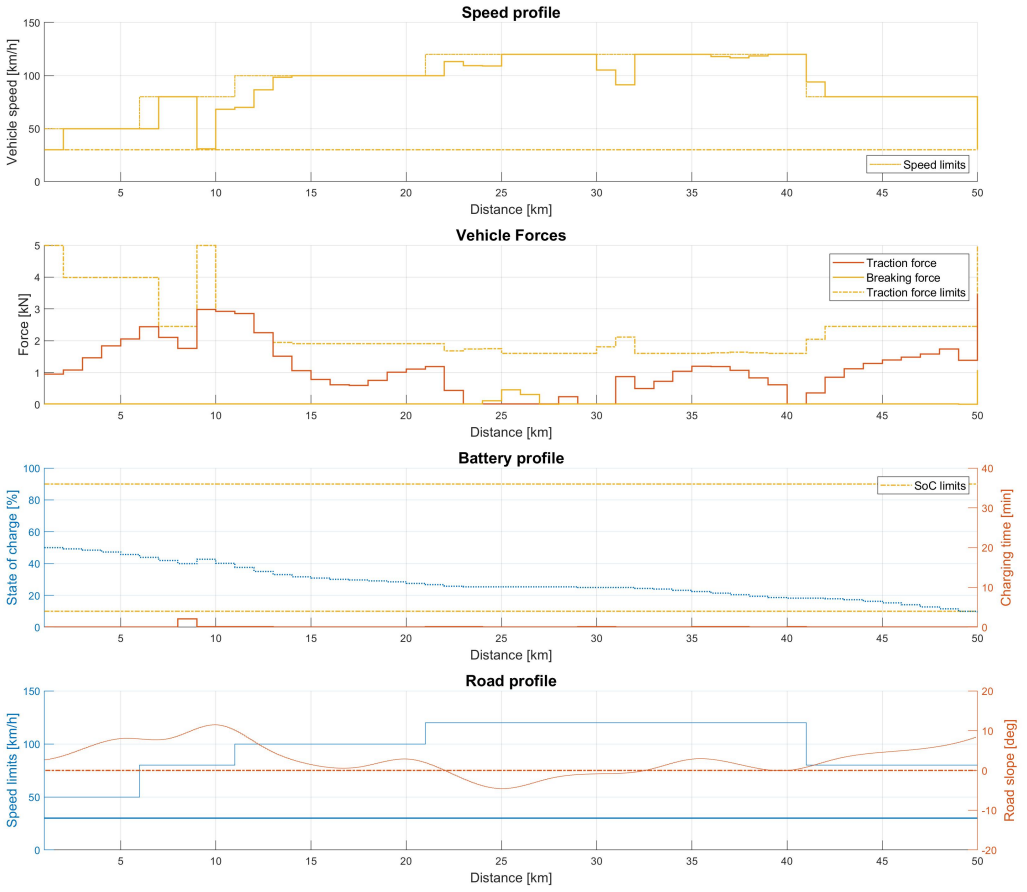


Figure 11.80: Vehicle performance on a short trip with 5 charging stations and 50 % initial SoC.

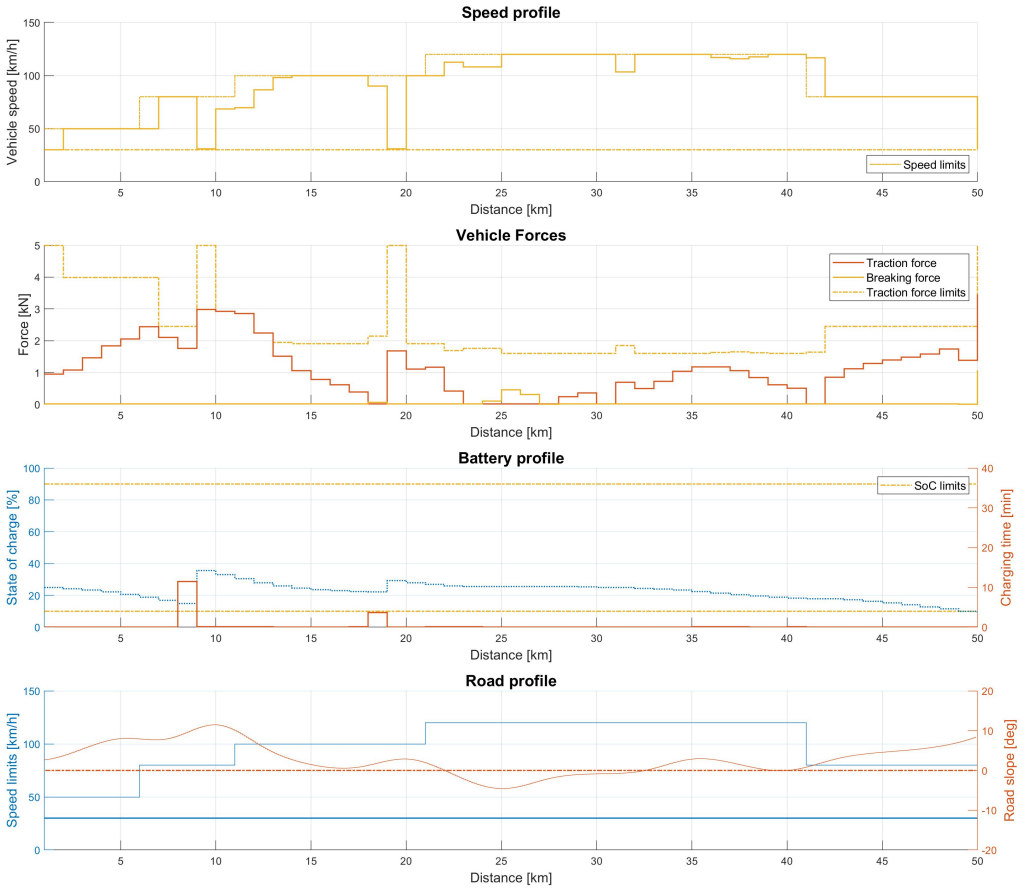


Figure 11.81: Vehicle performance on a short trip with 5 charging stations and 25 % initial SoC.

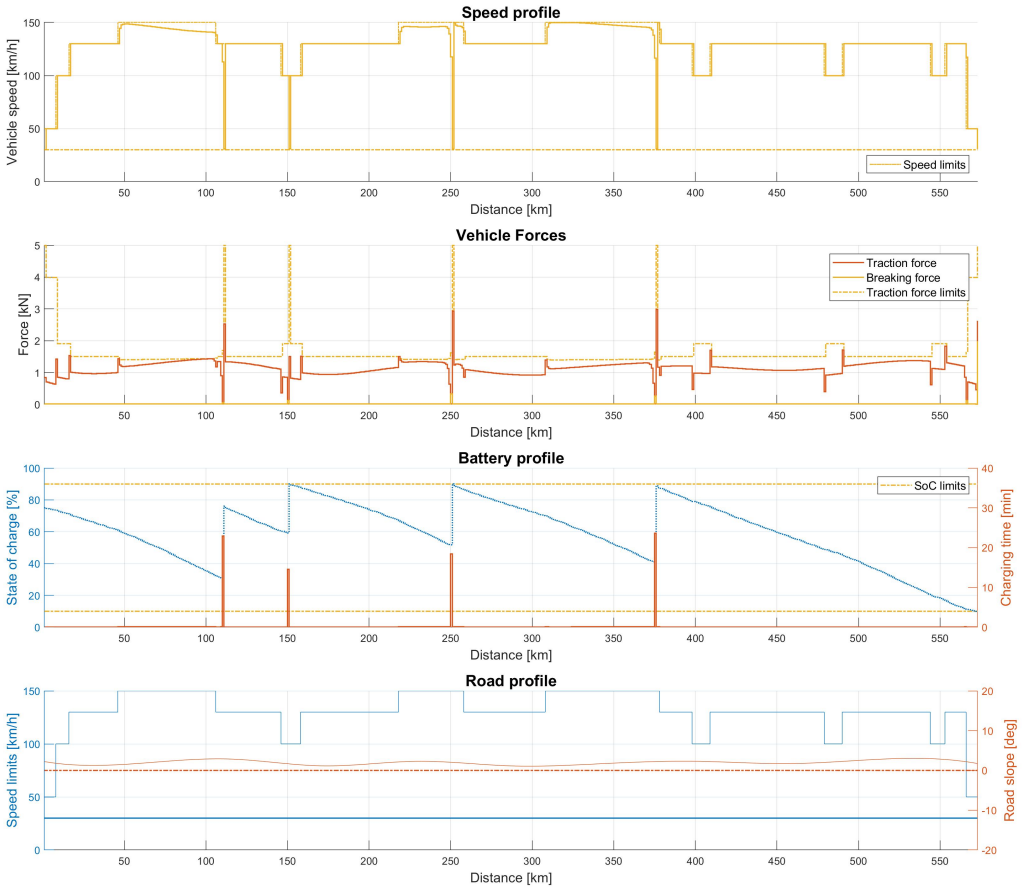


Figure 11.82: Vehicle performance on a long trip with 4 charging stations and 90 % initial SoC.

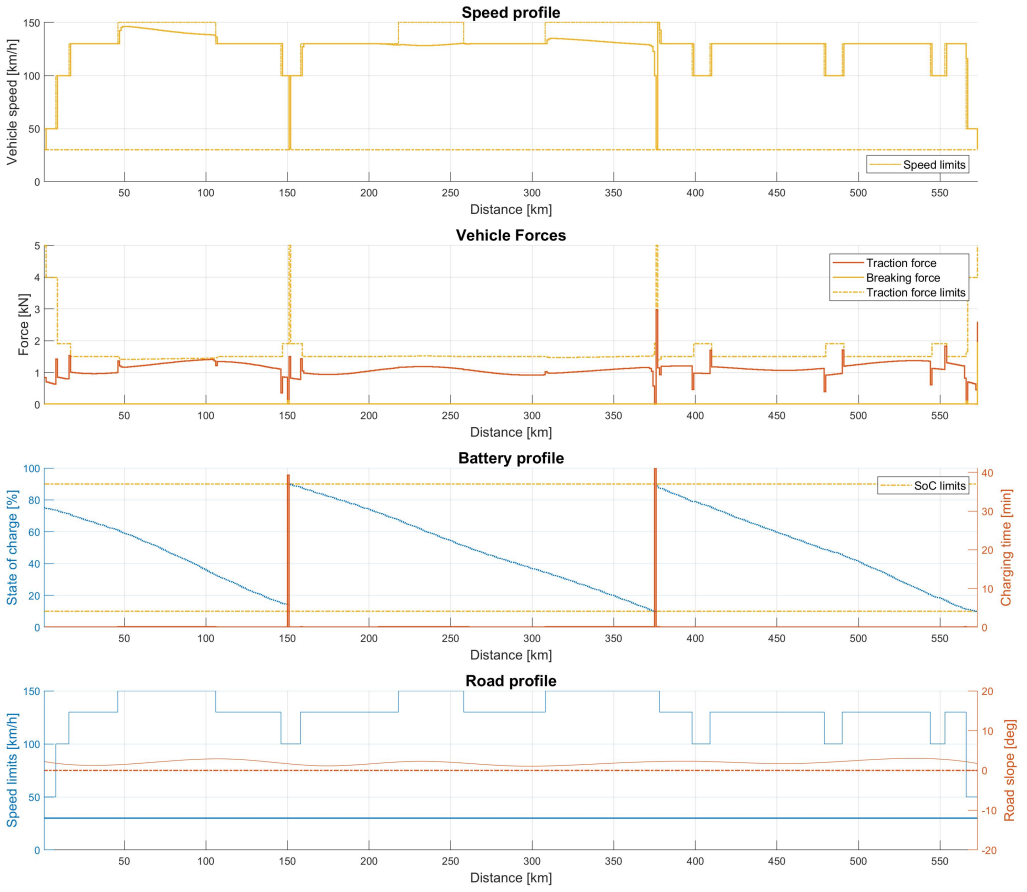


Figure 11.83: Vehicle performance on a long trip with 2 charging stations and 90 % initial SoC.

Chapter 12

Parametric problems

- *Defining parameters*
- *Example*
- *Parametric Quadratic Constraints*
- *Diagonal Hessians*
- *Sparse Parameters*
- *Special Parameters*
- *Python: Column vs Row Major Storage Format*

Parameters (or real-time data) are a key concept in FORCESPRO. Usually at least one vector in an embedded optimization problem will change between two calls to the solver. In MPC, the initial state changes usually between two sampling times. But other data can change too, for example because you are working with linearizations of nonlinear dynamics, or because the cost matrices of a quadratic objective function are tuned online. The following API is available when using the low-level interface only and cannot be used with the high-level interface.

12.1 Defining parameters

FORCESPRO gives you full control over the parametrization of the optimization problem: You can define all data matrices and vectors to be parametric. To define a parameter, use the function

Matlab

Python

```
parameter = newParam(name, maps2stage, maps2data);
```

```
stages.newParam(name, maps2stage, maps2data)
```

where `name` is the parameter name, which you need to be set before calling the solver. The vector of indices `maps2stage` defines to which stages the parameters maps. The last argument `maps2data` has to be one of the following strings

Table 12.1: Possible string values for argument `maps2data`

Cost function	Equality constraints	Inequality constraints
`'cost.H'`	`'eq.c'`	`'ineq.b.lb'`
`'cost.f'`	`'eq.C'`	`'ineq.b.ub'`
	`'eq.D'`	`'ineq.p.A'`
		`'ineq.p.b'`
		`'ineq.q.Q'`
		`'ineq.q.l'`
		`'ineq.q.r'`

From FORCESPRO `1.8.0`, the user is allowed to provide a parameter for all problem stages at once. All stage parameters are then stacked into one vector (each stage matrix converted into a column major column vector) before getting passed to the solver at runtime. FORCESPRO is notified about this by having

Matlab

Python

```
maps2stage = [];
```

```
maps2stage = []
```

For instance, in order to provide a parametric linear cost across all stages, one should use the following code at codegen.

Matlab

Python

```
parameter = newParam('linear_stage_cost', [], 'cost.f');
```

```
parameter = stages.newParam('linear_stage_cost', [], 'cost.f')
```

At runtime, the user is expected to provide the linear stage cost as follows.

Matlab

Python

```
problem.linear_stage_cost = repmat(rand(problem.nvar, 1), problem.horzLength, 1);
```

```
problem['linear_stage_cost'] = numpy.vstack((numpy.random.rand(problem.nvar, 1) for _␣
↪in range(problem.horzLength)))
```

where `problem.horzLength` is the horizon length and `problem.nvar` is the number of stage variables.

Note: Stacked parameters are not supported for quadratic constraints.

12.2 Example

To define the linear term of the cost of stages 1 to 5 as a parameter, use the following command

Matlab

Python

```
parameter1 = newParam('linear_cost', 1:5, 'cost.f');
```

```
stages.newParam('linear_cost', range(1, 6), 'cost.f')
```

Note that this will generate only one parameter and the same runtime data will be mapped to stages 1 to 5. If the runtime data should be different for each stage one would have to generate five different ones in this case.

We can also have a second parameter. For instance, the right-hand side of the first equality constraints, which is a very common case in MPC.

Matlab

Python

```
parameter2 = newParam('RHS_first_equality_constraint', 1, 'eq.c');
```

```
stages.newParam('RHS_first_equality_constraint', [1], 'eq.c')
```

12.3 Parametric Quadratic Constraints

As there may be multiple quadratic constraints for every stage, one needs to specify which ones are to be parametric. One can use a fourth argument in the `newParam` call, as shown below.

Matlab

Python

```
parameter = newParam(name, maps2stage, maps2data, idxWithinStage);
```

```
stages.newParam(name, maps2stage, maps2data, idxWithinStage)
```

where *idxWithinStage* denotes the index of the quadratic constraints to which this parameters applies.

12.4 Diagonal Hessians

In case your parametric Hessian is diagonal, you should use the fourth argument of `newParam` as shown below.

Matlab

Python

```
parameter1 = newParam('Hessians', 1:5, 'cost.H', 'diag');
```

```
stages.newParam('Hessians', range(1,6), 'cost.H', 'diag')
```

The FORCESPRO solver will then only expect a vector as a parameter. The `'diag'` keyword is currently only valid for hessian matrices related to the objective function.

12.5 Sparse Parameters

If your parameters are not diagonal but they have a sparse structure that can be exploited for performance, you can use the fourth and fifth arguments of *newParam* to let FORCESPRO know about the sparsity pattern.

Matlab

Python

```
parameter2 = newParam('Ai', 1:5, 'ineq.p.A', 'sparse', [zeros(5, 6) rand(5, 2)]);
```

```
stages.newParam('Ai',range(1,6),'ineq.p.A','sparse',numpy.hstack((numpy.zeros(5,6),
↪random.random((5,2)))))
```

The fifth argument is used to let FORCESPRO know about the location of the non-zero elements. When a solver is generated using sparse parameters it is the responsibility of the user to pass on parameters with the correct sparsity pattern to the solver. There will be no warnings thrown at runtime.

Sparse parameter values have to be passed as a column vector of nonzero elements, i.e. to assign the values of matrix B to sparse parameter Ci one should use the following:

Matlab

Python

```
problem.Ci = nonzeros(sparse(B));
```

```
problem.Ci = B[numpy.nonzeros(B)]
```

Note that parameters with a general sparsity structure defined by the fifth argument are currently only supported for polytopic constraints. For the equality constraint matrices, only the structure `[0 A]`, where `A` is assumed to be dense, is currently supported.

12.6 Special Parameters

To prevent having to transfer entire matrices for parameters with few changing elements at runtime, one can specify a sixth argument to let FORCESPRO know about the location of the elements that will be supplied at runtime.

Matlab

Python

```
parameter2 = newParam('Ci', 1:5,'eq.C','sparse',Cstruc,Cvar)
```

```
stages.newParam('Ci',range(1,6),'eq.C','sparse',Cstruc,Cvar)
```

Note that in this case the constant values will be taken from the data supplied in the field `Cstruc`. At runtime the user only has to supply a column vector including the time-varying elements marked in the field `Cvar`. The ordering should be column major.

12.7 Python: Column vs Row Major Storage Format

Unlike MATLAB, numpy stores arrays by default in row-major format internally. Since FORCESPRO expects the parameters in column major storage format, a conversion is necessary.

This conversion is automatically performed by the Python interface when the solver is called. To avoid the conversion every time the solver is called, you should use the following way of creating the arrays storing parameters:

```
a = array([1, 2, 3, 4, 5, 6])
b = a.reshape(2,3,order='F')
```

The above code reshapes the array into a (2,3) Matrix stored in column major (Fortran) format.

Chapter 13

Calling a solver from C or C++

- *Generating a solver library*
- *C interface*
- *Compiling solver executable*
  - *Compiling with C++*
- *Memory allocations*
  - *Internal memory*
  - *External memory*
  - *Code options related to solver memory*
  - *Obtaining memory size*

A generated FORCESPRO solver comes with a C interface and a C library, allowing the user to call the solver directly from C or C++. This enables direct and efficient integration of a FORCESPRO solver into existing software, and is the most portable way of deploying a solver to any supported hardware platform. This section describes how to call the solver from C, and how to compile your C program. The procedure to deploy to a target platform is described in section *Main Targets*.

13.1 Generating a solver library

For generating a solver to be used in C, the following options are relevant:

Matlab

Python

```
codeoptions.platform = '<platform_name>'; % to specify the target platform
codeoptions.printlevel = 0; % optional, on some platforms printing is not supported
codeoptions.cleanup = 0; % to keep auxiliary source files (not needed for low-level␣
↪interface)
```

```
codeoptions.platform = '<platform_name>' # to specify the target platform
codeoptions.printlevel = 0 # optional, on some platforms printing is not supported
codeoptions.cleanup = 0 # to keep auxiliary source files (not needed for low-level␣
↪interface)
```

After having generated a solver with name `codeoptions.name = 'myFORCESsolver'`, the following C libraries are made available for host and target systems:

- For the host system:

```
myFORCESsolver
├── include
├── lib
```

- For the target system:

```
myFORCESsolver
├── include
├── lib_target
```

For most platforms (see *Target platform*), `lib` and `lib_target` contain both static and shared/dynamic library files.

For the high-level interface you'll receive auxiliary sources after solver generation which contain the nonlinear functions of your formulation (if not provided externally). These are generated as follows:

- From MATLAB, the following auxiliary sources will be generated:

```
myFORCESsolver
├── myFORCESsolver_adtool2forces.c
├── myFORCESsolver_casadi.c
├── myFORCESsolver_casadi.h
```

- From Python, the following auxiliary sources will be generated:

```
myFORCESsolver
├── myFORCESsolver_interface.c
├── myFORCESsolver_model.c
├── myFORCESsolver_model.h
```

13.2 C interface

The following code snippets demonstrate how to use the C interface of any generated solver (for simplicity, we assume that the solver has been generated with `codeoptions.name = 'myFORCESsolver'`). The C variables `params` (solver input), `output` (solver output) and `info` (solver diagnostics) are C structs whose definitions depend on the problem formulation. Their exact definitions can be found in the solver header `myFORCESsolver.h`.

Low-level interface

```
#include "myFORCESsolver/include/myFORCESsolver.h"         /* solver symbols␣
↪definition        */
#include "myFORCESsolver/include/myFORCESsolver_memory.h"  /* handle to internal␣
↪solver memory */

int main()
{
    int exitflag;
    myFORCESsolver_params params; /* struct for run-time parameters, see␣
↪myFORCESsolver.h  */
    myFORCESsolver_output output; /* struct for solver output, see myFORCESsolver.h  ␣
↪     */
```

(continues on next page)

(continued from previous page)

```
    myFORCESsolver_info info;     /* struct for solver diagnostics, see␣
↪myFORCESsolver.h   */
    myFORCESsolver_mem * mem;     /* handle to solver memory, from myFORCESsolver_
↪memory.h */

    mem = myFORCESsolver_internal_mem(0);

    /* omitted: assign params field */

    /*  solver call:
     *  - in: params
     *  - out: exitflag, output, info
     *  - 5th argument: optional file pointer for printing, NULL means stdout
     */
    exitflag = myFORCESsolver_solve(&params, &output, &info, mem, NULL);
}
```

High-level interface

```
#include "myFORCESsolver/include/myFORCESsolver.h"        /* solver symbols␣
↪definition        */
#include "myFORCESsolver/include/myFORCESsolver_memory.h" /* handle to internal␣
↪solver memory */

/* AD tool - FORCESPRO interface
 * Note: if you use your own external C functions, replace 'myFORCESsolver_
↪adtool2forces'
 * with your main callback function name
 */
extern solver_int32_default myFORCESsolver_adtool2forces(
    myFORCESsolver_float *x,        /* primal vars                                ␣
↪      */
    myFORCESsolver_float *y,        /* eq. constraint multiplers                  ␣
↪      */
    myFORCESsolver_float *l,        /* ineq. constraint multipliers               ␣
↪      */
    myFORCESsolver_float *p,        /* parameters                                 ␣
↪      */
    myFORCESsolver_float *f,        /* objective function (scalar)                ␣
↪      */
    myFORCESsolver_float *nabla_f,  /* gradient of objective function             ␣
↪      */
    myFORCESsolver_float *c,        /* dynamics                                   ␣
↪      */
    myFORCESsolver_float *nabla_c,  /* Jacobian of the dynamics (column major)    ␣
↪      */
    myFORCESsolver_float *h,        /* inequality constraints                     ␣
↪      */
    myFORCESsolver_float *nabla_h,  /* Jacobian of inequality constraints (column␣
↪major)   */
    myFORCESsolver_float *hess,     /* Hessian (column major)                     ␣
↪      */
    solver_int32_default stage,     /* stage number (0 indexed)                   ␣
↪      */
    solver_int32_default iteration, /* iteration number of solver                 ␣
↪      */
```

(continues on next page)

(continued from previous page)

```
    solver_int32_default threadID   /* Id of caller thread                    ␣
↪     */);

int main()
{
    int exitflag;
    myFORCESsolver_params params;   /* struct for run-time parameters, see␣
↪myFORCESsolver.h  */
    myFORCESsolver_output output;   /* struct for solver output, see myFORCESsolver.h␣
↪       */
    myFORCESsolver_info info;       /* struct for solver diagnostics, see␣
↪myFORCESsolver.h   */
    myFORCESsolver_mem * mem;       /* handle to solver memory, from myFORCESsolver_
↪memory.h */

    /* handle to external function interface */
    myFORCESsolver_extfunc extfunc_eval = &myFORCESsolver_adtool2forces;

    /* get handle to solver memory, from myFORCESsolver_memory.h */
    mem = myFORCESsolver_internal_mem(0);

    /* omitted: assign params field */

    /*  solver call:
     *  - in: params
     *  - out: exitflag, output, info
     *  - 5th argument: optional file pointer for printing, NULL means stdout
     */
    exitflag = myFORCESsolver_solve(&params, &output, &info, mem, NULL, extfunc_eval);
}
```

Note: More advanced use cases of the C interface are treated in the sections *Memory allocations* (for controlling memory allocations) and *External parallelism* (for concurrency and thread-safety aspects).

13.3 Compiling solver executable

In order to create a standalone executable, all user-provided and generated auxiliary source files need to be compiled and linked against the solver library. The full command for compiling a solver looks as follows (assuming you are in the directory which contains the `myFORCESsolver` folder):

- For the low-level interface

```
<Compiler> <user sources> <solver library> <additional libraries>
```

- For the high-level interface

  Matlab

  Python

```
<Compiler> <user sources> myFORCESsolver_adtool2forces.c myFORCESsolver_casadi.c
↪<solver library> <additional libraries>
```

```
<Compiler> <user sources> myFORCESsolver_interface.c myFORCESsolver_model.c
↪<solver library> <additional libraries>
```

Where:

- `<Compiler>` would be the compiler executable
- `<user sources>` would be your sources calling the solver (see *C interface*)
- `<solver library>` would be one of the solver library files from `lib` or `lib_target` directory (see *Generating a solver library*)
- `<additional libraries>` would be possible libraries that need to be linked to resolve existing dependencies
  - For Linux/MacOS it's usually necessary to link the math library (`-lm`)
  - For Windows you usually need to link the `iphlpapi.lib` library (it's distributed with the Intel Compiler, MinGW as well as MATLAB) and unless you're using MinGW some additional intel libraries (those are included in the FORCESPRO client under the folder `libs_Intel` – if missing they are downloaded after code generation)

13.3.1 Compiling with C++

Compiling a FORCESPRO solver with a C++ compiler is straightforward: Simply use the C interface as described in section *C interface* along with the `extern "C"` syntax to ensure the C linkage conventions are used.

A complete example of compiling a nonlinear FORCESPRO solver with a C++ compiler using CMake can be found in the `examples\StandaloneExecution\Cpp` folder that comes with your client. In order to run this example, do the following:

1. Execute `OverheadCrane.m` in MATLAB to generate a FORCESPRO solver called `CraneSolver` (uncomment line 96 to generate a solver for Linux).
2. Run `cmake .`
3. Run `cmake --build .`
4. An executable `CraneSolver_standalone` should be available, feel free to run it.

13.4 Memory allocations

The C interface provides some flexibility in how the solver memory is allocated:

- internal memory: memory buffer is statically allocated inside the solver library
- external memory: the user is responsible for allocating the memory buffer

The internal memory option is easier to use and thus the default way of interfacing the solver in C. The external memory option is recommended for users who want full control over memory allocation (static or dynamic), or who require multiple memory buffers (e.g. for running a solver in parallel, see *External parallelism*).

Henceforth, we assume a generated solver named `FORCESNLPsolver` and demonstrate how to use external and internal memory buffers. We assume the reader is already acquainted with the C interface (see section *C interface*).

You can find the full code for working examples for the internal and external memory interfaces including instructions on how to run them in the `examples\StandaloneExecution\C` folder that comes with your client.

13.4.1 Internal memory

If you don't need control over the memory buffer, the internal memory C interface is the recommended way of calling a generated solver in C:

```
/* additional header for internal memory functionality */
#include "FORCESNLPsolver/include/FORCESNLPsolver_memory.h"

/* handle to the solver memory */
FORCESNLPsolver_mem * mem_handle;

/* Get i-th memory buffer */
int i = 0;
mem_handle = FORCESNLPsolver_internal_mem(i);
/* Note: number of available memory buffers is controlled by code option max_num_mem␣
↪*/

/* check that memory is in valid state: */
if (mem_handle == NULL)
{
    /* this happens if i >= max_num_mem */
    return 1;
}
exit_code = FORCESNLPsolver_solve(..., mem_handle, ...)
```

By default, one memory buffer is available (`max_num_mem = 1`). For further information on the code option `max_num_mem`, see Table 13.1.

13.4.2 External memory

The following code sample demonstrates the use of external memory with dynamic allocation:

```
/* memory buffer allocated by the user of type char (representing bytes) */
char * mem;

/* handle to the solver memory */
FORCESNLPsolver_mem * mem_handle;

/* required memory size in bytes */
size_t mem_size = FORCESNLPsolver_get_mem_size();

/* dynamically allocate memory buffer */
mem = malloc(mem_size);

/* cast memory buffer to solver memory
 * note: i can be set to 0 if no thread safety required */
int i = 0;
mem_handle = FORCESNLPsolver_external_mem(mem, i, mem_size);

/* check that memory is in valid state: */
if (mem_handle == NULL)
{
    return 1;
}
exit_code = FORCESNLPsolver_solve(..., mem_handle, ...)
```

(continues on next page)

(continued from previous page)

```
/* free user-allocated memory */
free(mem);
```

For static allocation, the memory size `MEM_SIZE` needs to be set at compile time:

```
/* memory size in bytes s.t. MEM_SIZE >= FORCESNLPsolver_get_mem_size(): */
#define MEM_SIZE 12345

/* statically allocated memory buffer of size MEM_SIZE bytes */
static char mem[MEM_SIZE];
/* cast buffer to solver memory */
mem_handle = FORCESNLPsolver_external_mem(mem, 0, MEM_SIZE);

/* check that memory is in valid state: */
if (mem_handle == NULL)
{
    /* this happens if MEM_SIZE < FORCESNLPsolver_get_mem_size() */
    return 1;
}
```

The minimum required `MEM_SIZE` is system and compiler dependent. It can be easily obtained by compiling `FORCESNLPsolver\interface\FORCESNLPsolver_get_mem_size.c` and by linking against the solver library on the target device. The output of the obtained executable is equal to the value returned by `FORCESNLPsolver_get_mem_size()`.

For memory-critical systems, we advise to disable the internal memory buffer by setting (see also Table 13.1)

Matlab

Python

```
codeoptions.max_num_mem = 0;
```

```
codeoptions.max_num_mem = 0
```

Otherwise, the solver library still contains one internal memory buffer required by the client to run the solver.

13.4.3 Code options related to solver memory

By default, one memory buffer is statically allocated within the solver library, to be used only with the internal memory interface. The number of internal memory buffers `n` can be controlled by setting the option

Matlab

Python

```
codeoptions.max_num_mem = n;
```

```
codeoptions.max_num_mem = n
```

The required number of buffers depends on the use case:

Table 13.1: Code option `max_num_mem`

`max_num_mem`	Use case	Side effects
`0`	external memory interface: no excess memory.	external memory int
`1` (default)	internal memory interface: default use.	external memory int
`n`	internal memory interface: thread safe for up to `n` concurrent solvers	external memory int

13.4.4 Obtaining memory size

The required memory size for a solver can be obtained by calling the two utility functions provided by the solver library:

```
size_t FORCESNLPsolver_get_mem_size( void );
size_t FORCESNLPsolver_get_const_size( void );
```

These functions return the memory size for all non-const / const variables, respectively. This information is also printed in the solver output if `printlevel = 2`. Note that the memory size depends on the system and compiler. The actual memory footprint might be larger as reported by these functions since they account only for the memory to store data (data or bss segment in the binary), and not for the full binary size.

The size returned by `FORCESNLPsolver_get_mem_size` refers to an internal or external memory buffer. The size returned by `FORCESNLPsolver_get_const_size` refers to additional constant memory that is not exposed to the user. To obtain the total memory size of a solver that is called in parallel, the size of the memory buffer must be multiplied with the number of threads: `total_size = NUM_THREADS * FORCESNLPsolver_get_mem_size() + FORCESNLPsolver_get_const_size()`.

Chapter 14

Code Deployment

14.1 Main Targets

- *Deployment from Low-level or High-level Interface*
- *Deployment from Y2F interface*

The most portable way of deploying a solver to any supported hardware platform is to write a C interface to the generated solver, and to compile it for the target platform. The C interface is described in section *Calling a solver from C or C++*, as well as the process to compile it and to link it against the solver library. This section describes aspects which are relevant for deployment.

The supported targets include:

- x86 platforms
- x86_64 platforms
- 32bit ARM-Cortex-A platforms
- 32bit ARM-Cortex-M platforms (no shared libraries)
- 64bit ARM-Cortex-A platforms (AARCH64 toolchain)
- 64bit ARM-Cortex-A platforms (Integrity toolchain)
- NVIDIA platforms with ARM-Cortex-A processors
- PowerPC platforms with GCC compiler
- National Instruments compactRIO platforms with NILRT GCC compiler (Linux RTOS)

You can check *here* to find the correct naming option for each platform.

14.1.1 Deployment from Low-level or High-level Interface

The below steps hold generally for solver deployment but are exemplified for the examples *Basic low-level example* and *Basic high-level example*.

1. Set the code generation options in the Matlab or Python solver formulation as:

Matlab

Python

```
codeoptions.platform = '<platform_name>'; % to specify the platform
codeoptions.printlevel = 0; % optional, on some platforms printing is not supported
codeoptions.cleanup = 0; % to keep auxiliary source files (not needed for low-level␣
↪interface)
```

```
codeoptions.platform = '<platform_name>' # to specify the platform
codeoptions.printlevel = 0 # optional, on some platforms printing is not supported
codeoptions.cleanup = 0 # to keep auxiliary source files (not needed for low-level␣
↪interface)
```

and then generate the code for your solver (henceforth referred to as `myFORCESsolver`, placed in the folder `BasicExample`).

2. For most target platforms you will receive the following compiled files with your generated solver:

- For MinGW/Linux/MacOS:
  - a static library file `libmyFORCESsolver.a` inside the folder `lib_target`
  - a shared library file `libmyFORCESsolver.so` inside the folder `lib_target`
- For Windows:
  - a static library file `myFORCESsolver_static.lib` inside the folder `lib_target`
  - a dynamic library file `myFORCESsolver.dll` with its definition file for compilation `myFORCESsolver.lib` inside the folder `lib_target`

You need only one of those to build the solver.

Important: The shared library and the dynamic library if used for building need to be present during runtime as well.

---

3. Create an interface to call the solver (as described in *C interface*) and perform a simulation/test.

You can find a standalone C code for this example provided with your client in the following directories:

- basic low-level example C interface: `examples/DeploymentSimulation/MainTargets/LowLevel_BasicExample.c`
- basic high-level example C interface: `examples/DeploymentSimulation/MainTargets/HighLevel_BasicExample.c`

4. Copy to the target platform:

- The `myFORCESsolver` folder
- The interface from step 3
- Any additional source files for external functions (does not apply to this example)

5. Compile the solver as described in *Compiling solver executable* .

Important: When deploying to a target hardware platform, the library included in the **lib_target** directory of the generated solver should be used instead of the library in the **lib** directory.

14.1.2 Deployment from Y2F interface

The steps to deploy and simulate a FORCESPRO controller on most targets are detailed below.

1. In the Y2F interface example mpc_basic_example.m set the code generation options:

```
codeoptions.platform = '<platform_name>'; % to specify the platform
codeoptions.printlevel = 0; % optional, on some platforms printing is not supported
```

and then generate the code for your solver (henceforth referred to as `simpleMPC_solver`, placed in the folder `Y2F`) using the Y2F interface.

2. The Y2F solver is composed of a main solver which calls multiple internal solvers. The file describing the main solver is:

- `simpleMPC_solver.c` inside the folder `interface`

3. The internal solvers are provided as compiled files. For most target platforms you will receive the following compiled files:

- For MinGW/Linux/MacOS:
  - a static library file `libi_simpleMPC_solver_1.a` inside the folder `lib_target`
  - a shared library file `libi_simpleMPC_solver_1.so` inside the folder `lib_target`
- For Windows:
  - a static library file `i_simpleMPC_solver_1_static.lib` inside the folder `lib_target`
  - a dynamic library file `i_simpleMPC_solver_1.dll` with its definition file for compilation `i_simpleMPC_solver_1.lib` inside the folder `lib_target`

You need only one of those to build the solver.

Important: The shared library and the dynamic library if used for building need to be present during runtime as well.

---

4. Create an interface to call the solver and perform a simulation/test.

You can find a C interface for this example to try it out for yourself in the `examples` folder that comes with your client.

5. Copy in the target platform:

- The `simpleMPC_solver` folder
- The interface from step 4

6. Compile the solver. The compilation command would be (supposing you are in the directory which contains the `simpleMPC_solver` folder):

```
<Compiler> Y2F_mpc_basic_example.c simpleMPC_solver/interface/simpleMPC_solver.c
↪<solver library> <additional libraries>
```

Where the placeholders <Compiler>, <solver library> and <additional libraries> are documented in *Compiling solver executable*.

14.2 dSPACE deployment through Simulink Coder

- *Platform Specific Configurations*
  - *Platform name codeoption*
  - *Simulink Model HW Target Configuration*
- *High-level interface*
- *Y2F interface*
  - *Instructions*

This process applies to the following dSPACE platforms

- **dSPACE MicroAutoBox II**
- **dSPACE AutoBox**
- **dSPACE MicroLabBox**

Note: FORCESPRO currently does not support the dSPACE MicroAutoBox II platform. This example has remained as a guideline for deployment to other target platforms.

Important: When deploying to a target hardware platform, the library included in the **lib_target** directory of the generated solver should be used instead of the library in the **lib** directory.

14.2.1 Platform Specific Configurations

Platform name codeoption

When generating code for HW target platforms, `codeoptions.platform` needs to be set.

- **dSPACE MicroAutoBox II**: `'dSPACE-MABII'`
- **dSPACE AutoBox**: `'dSPACE-AutoBox'`
- **dSPACE MicroLabBox**: `'dSPACE-MicroLabBox'`

Simulink Model HW Target Configuration

When creating a Simulink Model for HW target platforms, certain hardware options need to be set.

- `Simulink Model Template`:
  - **dSPACE MicroAutoBox II**: `RTI1401`
  - **dSPACE AutoBox**: `RTI1007`
  - **dSPACE MicroLabBox**: `RTI1202`
- `System target file`:
  - **dSPACE MicroAutoBox II**: `rti1401.tlc`
  - **dSPACE AutoBox**: `rti1007.tlc`

  - **dSPACE MicroLabBox**: `rti1202.tlc`
- `Template makefile`:
  - **dSPACE MicroAutoBox II**: `rti1401.tmf`
  - **dSPACE AutoBox**: `rti1007.tmf`
  - **dSPACE MicroLabBox**: `rti1202.tmf`

14.2.2 High-level interface

The steps to deploy and simulate a FORCESPRO controller on a dSPACE platform are detailed below.

1. (Figure 14.1) Set the code generation options (for **<platform_name>** see *Platform name codeoption*):

```
codeoptions.platform = '<platform_name>'; % to specify the platform
codeoptions.printlevel = 0; % on some platforms printing is not supported
codeoptions.cleanup = 0; % to keep necessary files for target compile
```

and then generate the code for your solver (henceforth referred to as "FORCESNLPsolver", placed in the folder "BasicExample") using the high-level interface.

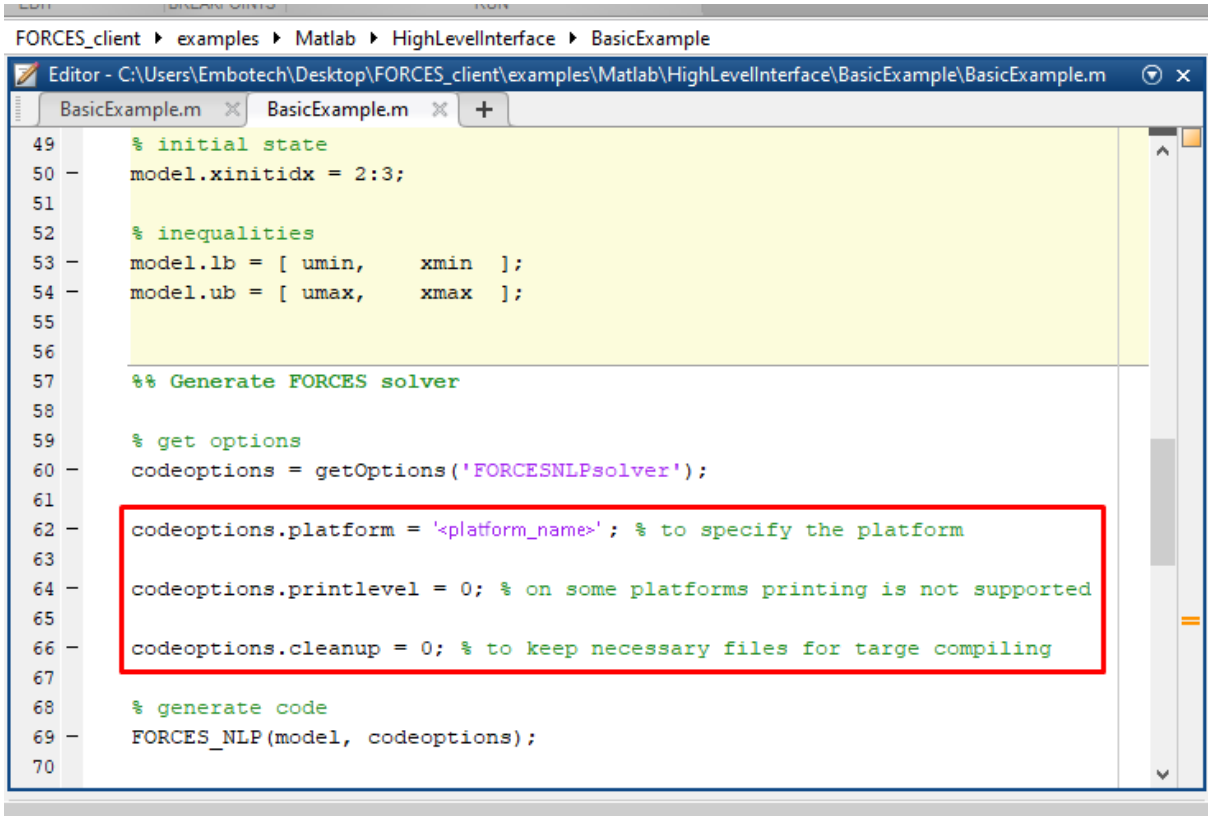


Figure 14.1: Set the appropriate code generation options.

2. (Figure 14.2) Create a new Simulink model using the Simulink model template provided by dSPACE (for **<simulink_model_template>** see *Simulink Model HW Target Configuration*).
3. (Figure 14.3) Populate the Simulink model with the system you want to control.



Figure 14.2: Create a Simulink model.

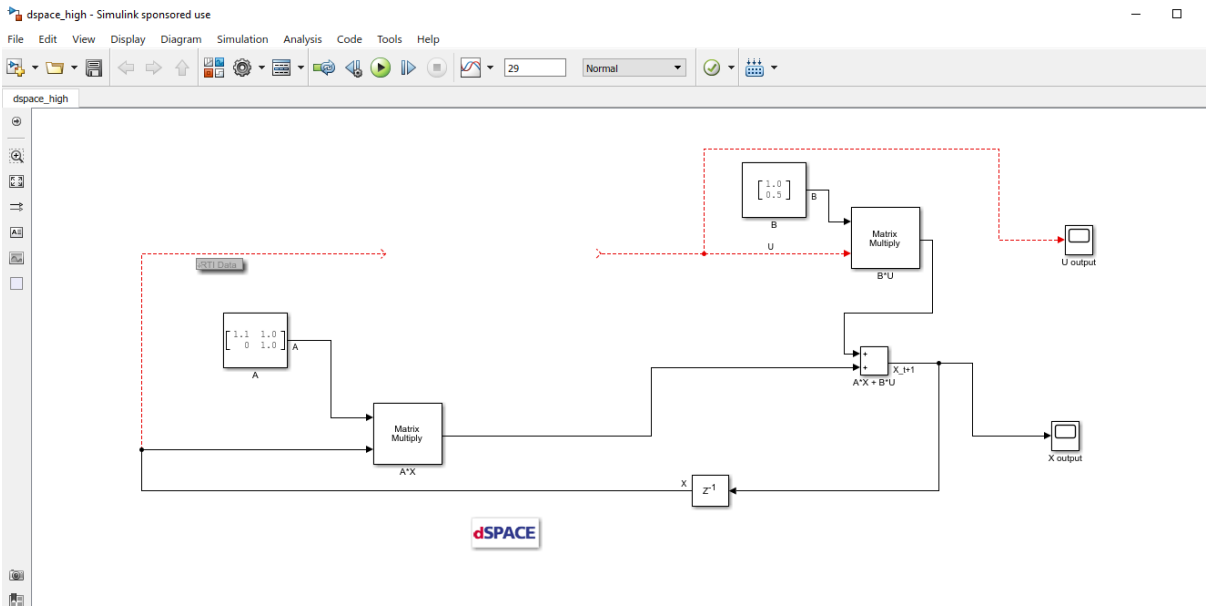


Figure 14.3: Populate the Simulink model.

4. (Figure 14.4) Make sure the `FORCESNLPsolver_simulinkBlock.mexw64` file (created during code generation) is on the MATLAB path.
5. (Figure 14.5) Open the `FORCESNLPsolver_lib.mdl` Simulink model file, contained in the `interface` folder of the `FORCESNLPsolver` folder created during code generation.
6. (Figure 14.6) Copy-paste the FORCESPRO Simulink block into your simulation model and connect its inputs and outputs appropriately.
7. (Figure 14.7) Access the Simulink model's options.
8. (Figure 14.8) In the "Solver" tab, set the options:
   - Simulation start/stop time: Depending on the simulation wanted.
   - Solver type: Discrete or fixed-step.
   - Fixed-step size: Needs to be higher than the execution time of the solver.
9. (Figure 14.9) In the "Code Generation" tab, set the options (for `<tlc_file>` and `<makefile_template>` see *Simulink Model HW Target Configuration*):
   - System target file: `<tlc_file>`
   - Language: C
   - Generate makefile: On
   - Template makefile: `<makefile_template>`
   - Make command: `make_rti`
10. (Figure 14.10) In the "Code Generation/Custom Code" tab, include the directories:
    - `BasicExample`
    - `BasicExample\FORCESNLPsolver\interface`
    - `BasicExample\FORCESNLPsolver\lib_target`
11. (Figure 14.11) In the "Code Generation/Custom Code" tab, add the source files:
    - `FORCESNLPsolver_simulinkBlock.c`
    - `FORCESNLPsolver_adtool2forces.c`
    - `FORCESNLPsolver_casadi.c`
12. (Figure 14.12) In the "Code Generation/Custom Code" tab, add the library file:
    - `FORCESNLPsolver.lib`
13. (Figure 14.13) Access the FORCESPRO block's parameters.
14. (Figure 14.14) Remove the "FORCESNLPsolver" prefix from the S-function module.
15. (Figure 14.15) Compile the code of the Simulink model. This will also automatically load the model to the connected dSPACE platform.
16. Deployment is complete and simulations can now be run on the dSPACE platform.

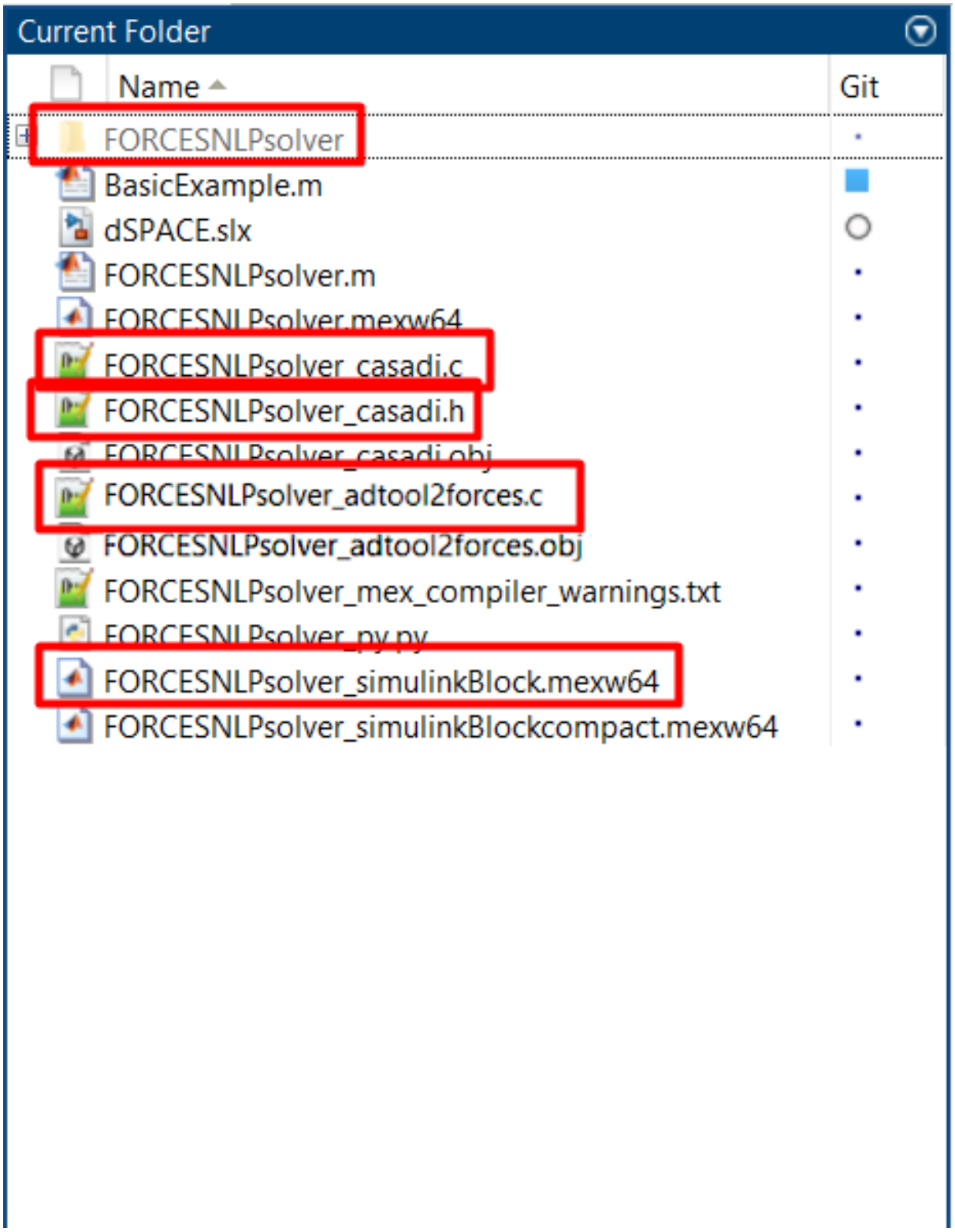


Figure 14.4: Add the folder containing the `.mexw64` solver file to the MATLAB path.

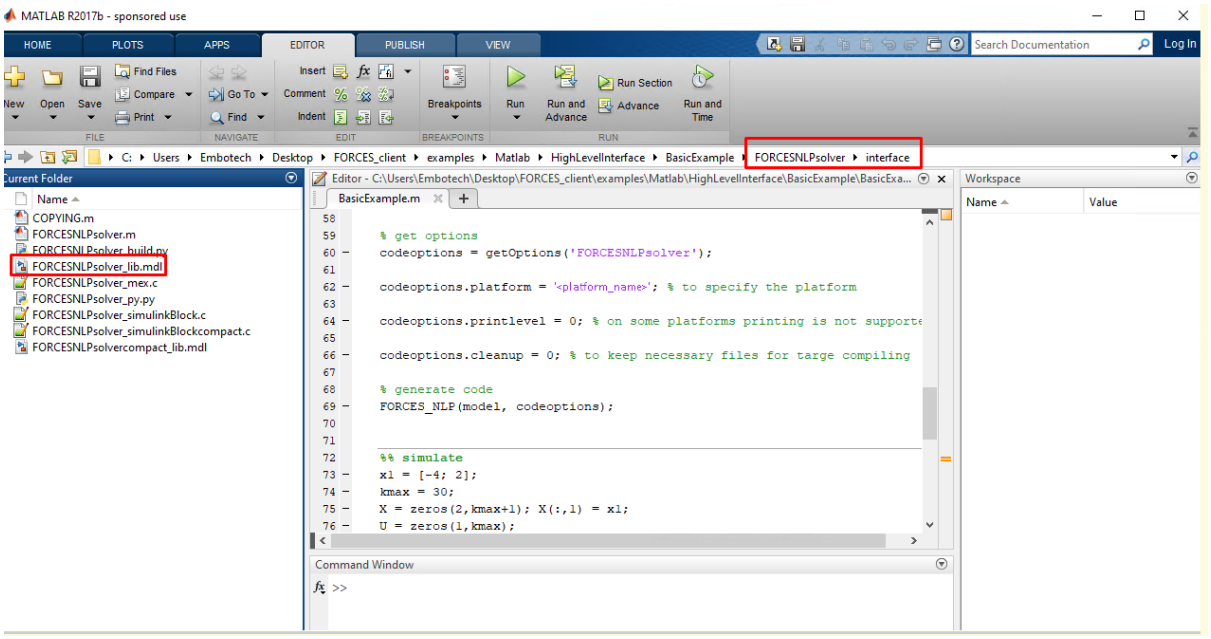


Figure 14.5: Open the generated Simulink solver model.

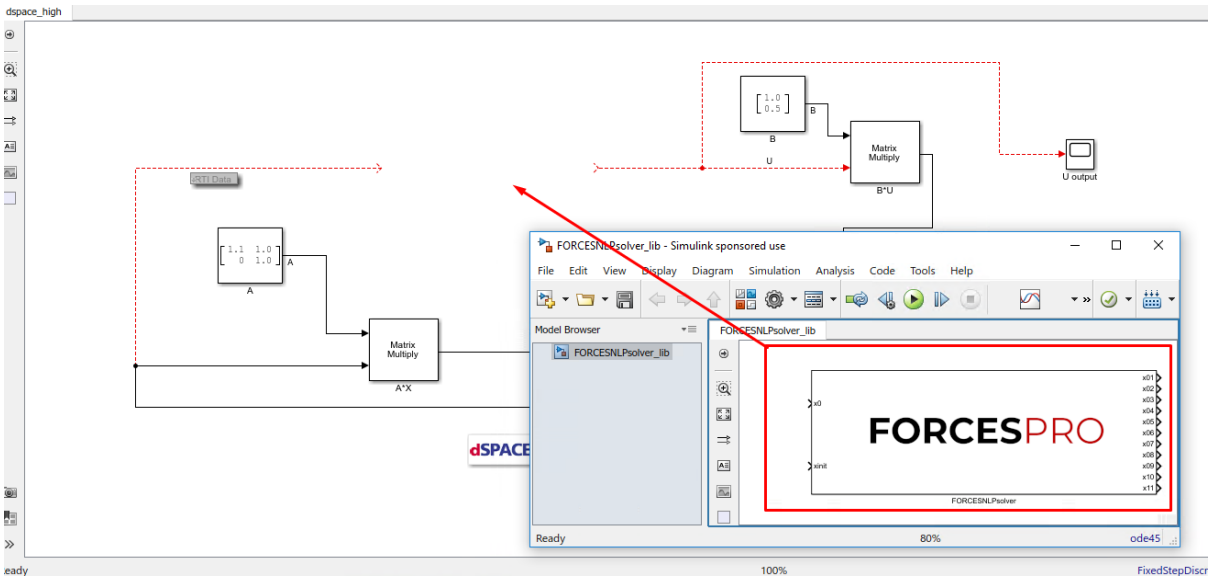


Figure 14.6: Copy-paste and connect the FORCESPRO block.

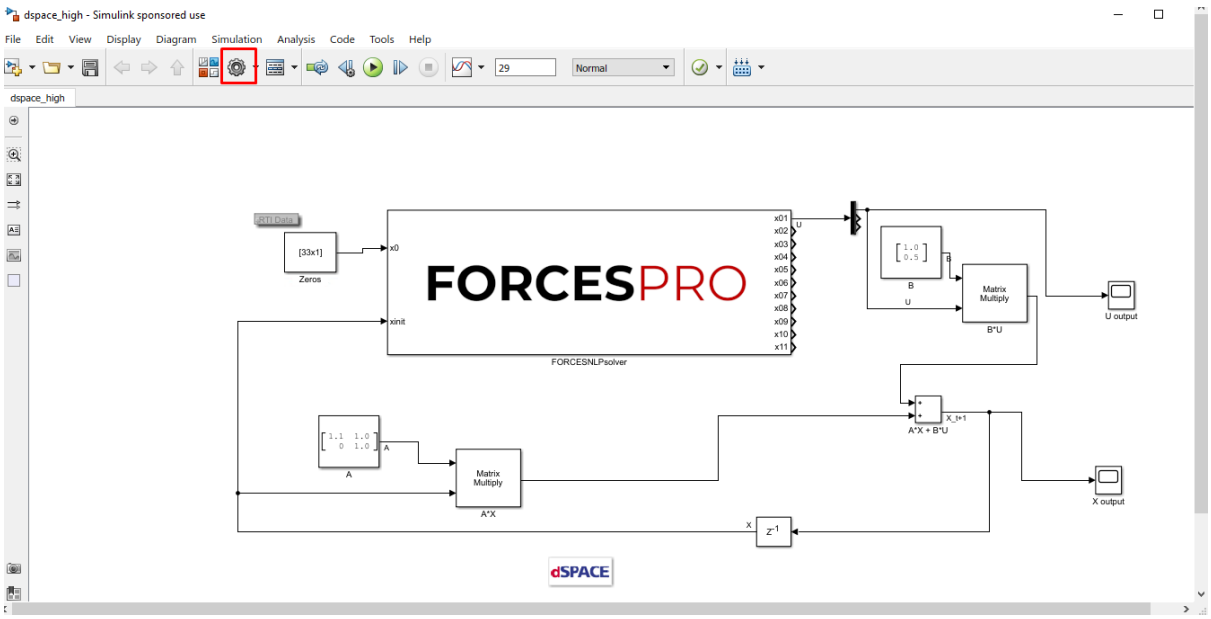


Figure 14.7: Open the Simulink model options.

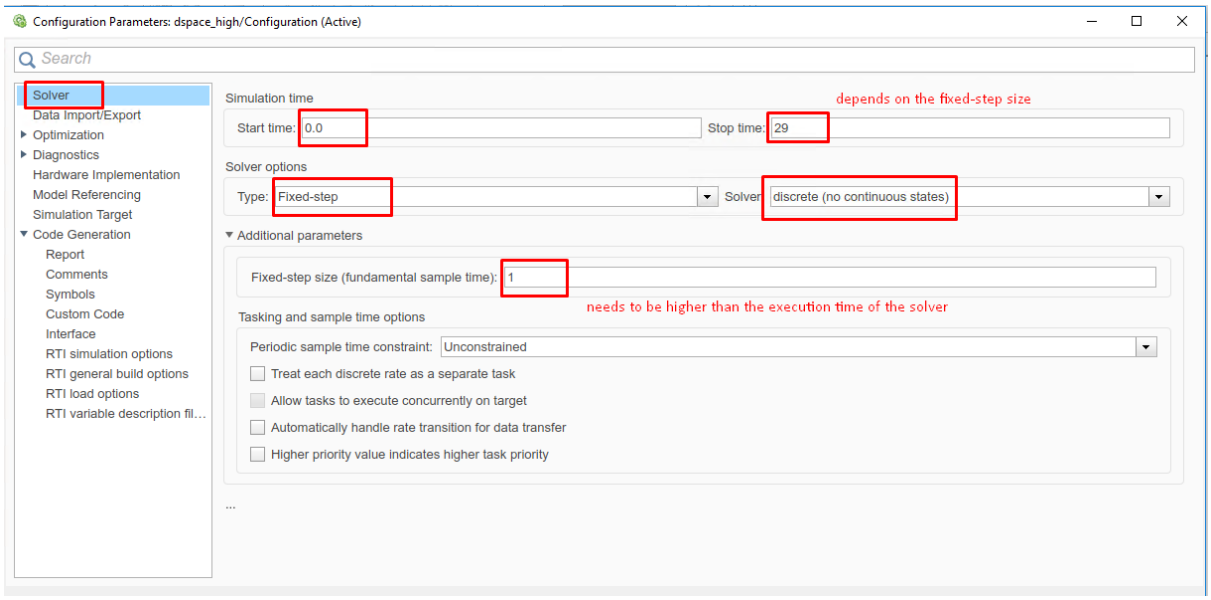


Figure 14.8: Set the Simulink solver options.

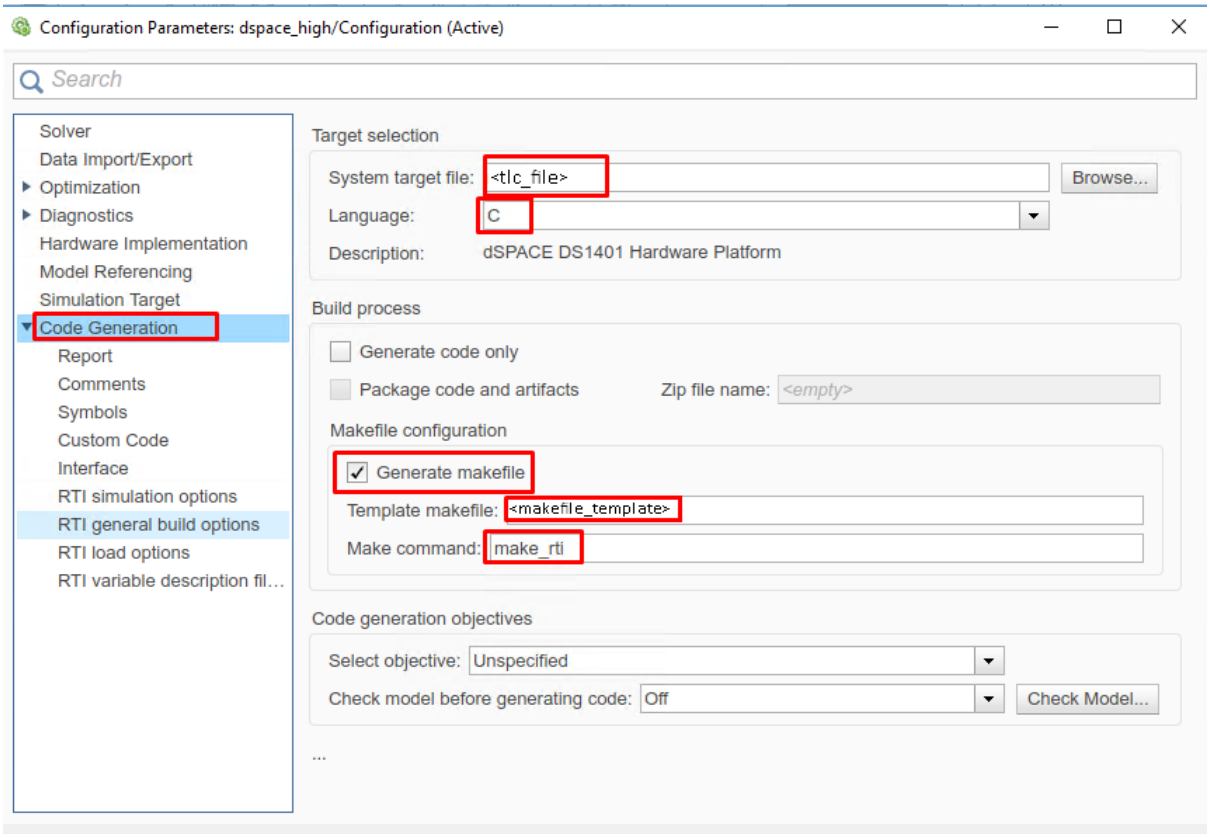


Figure 14.9: Set the Simulink code generation options.

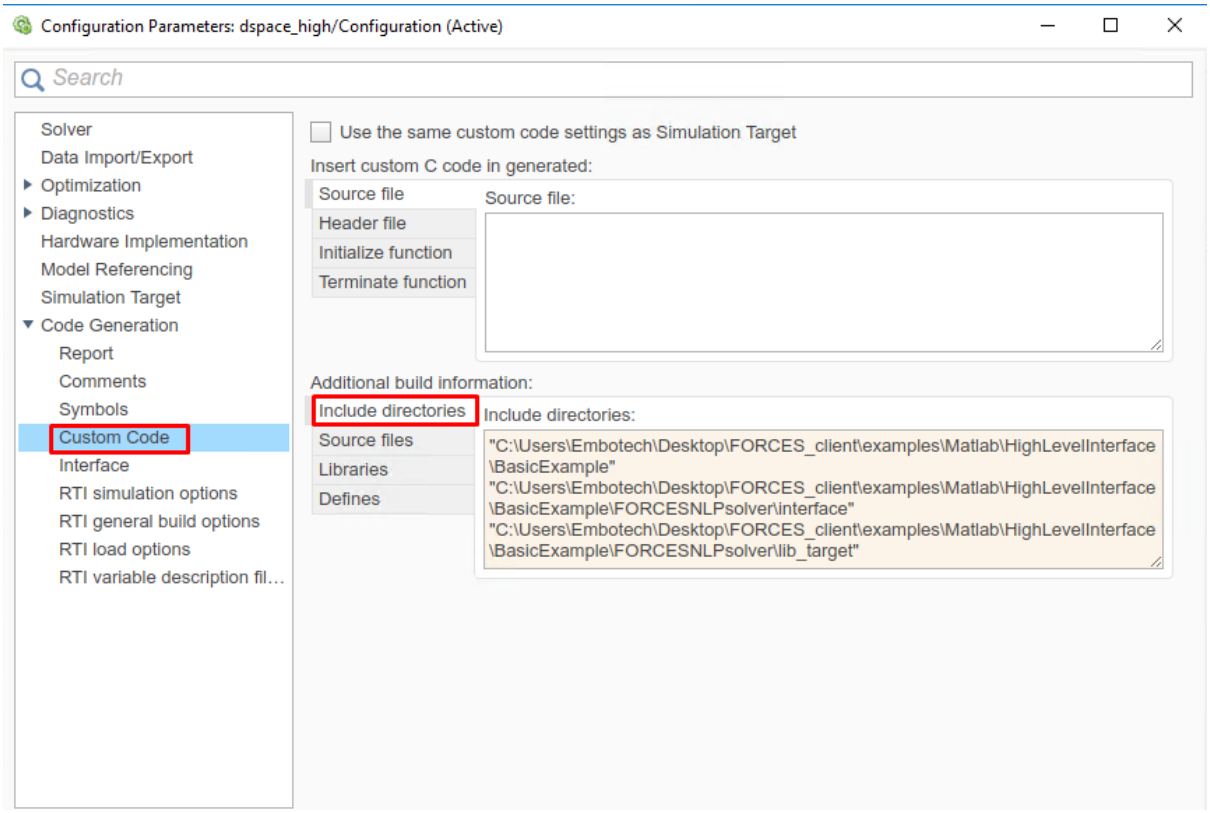


Figure 14.10: Add the directories included for the code generation.

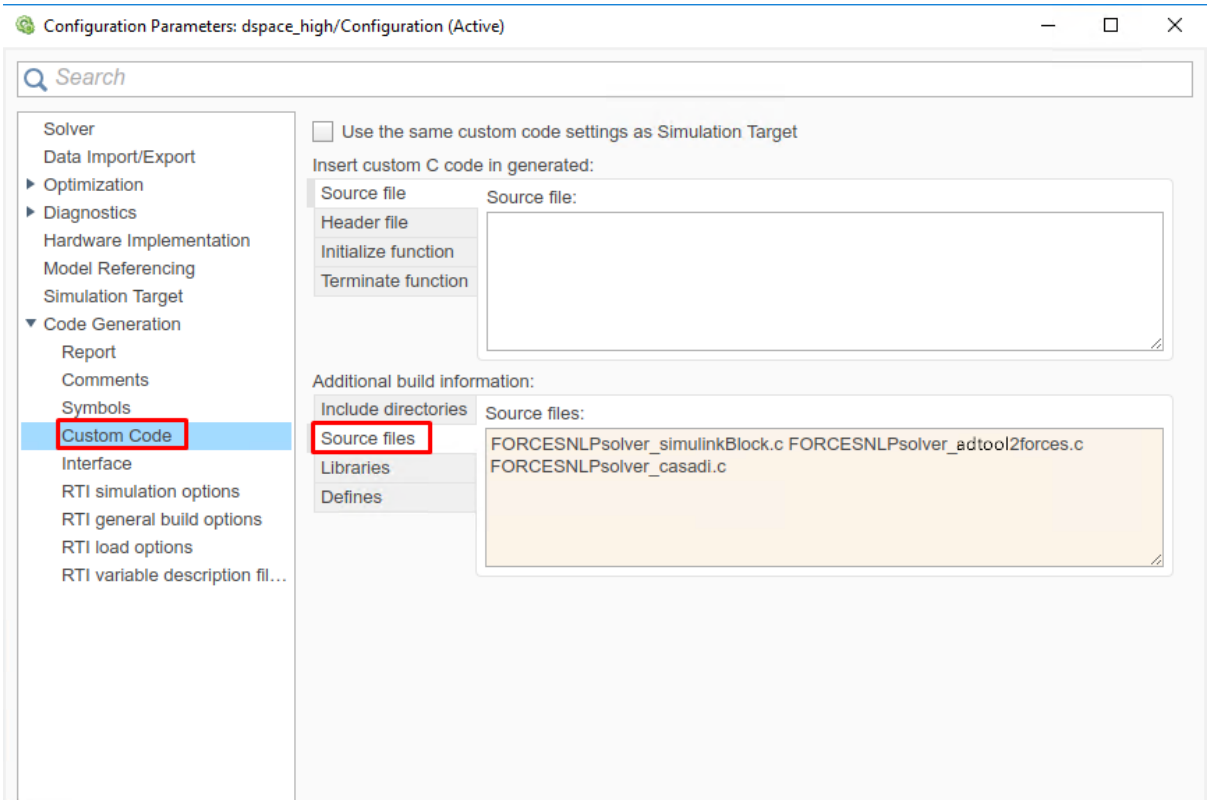


Figure 14.11: Add the source files used for the code generation.

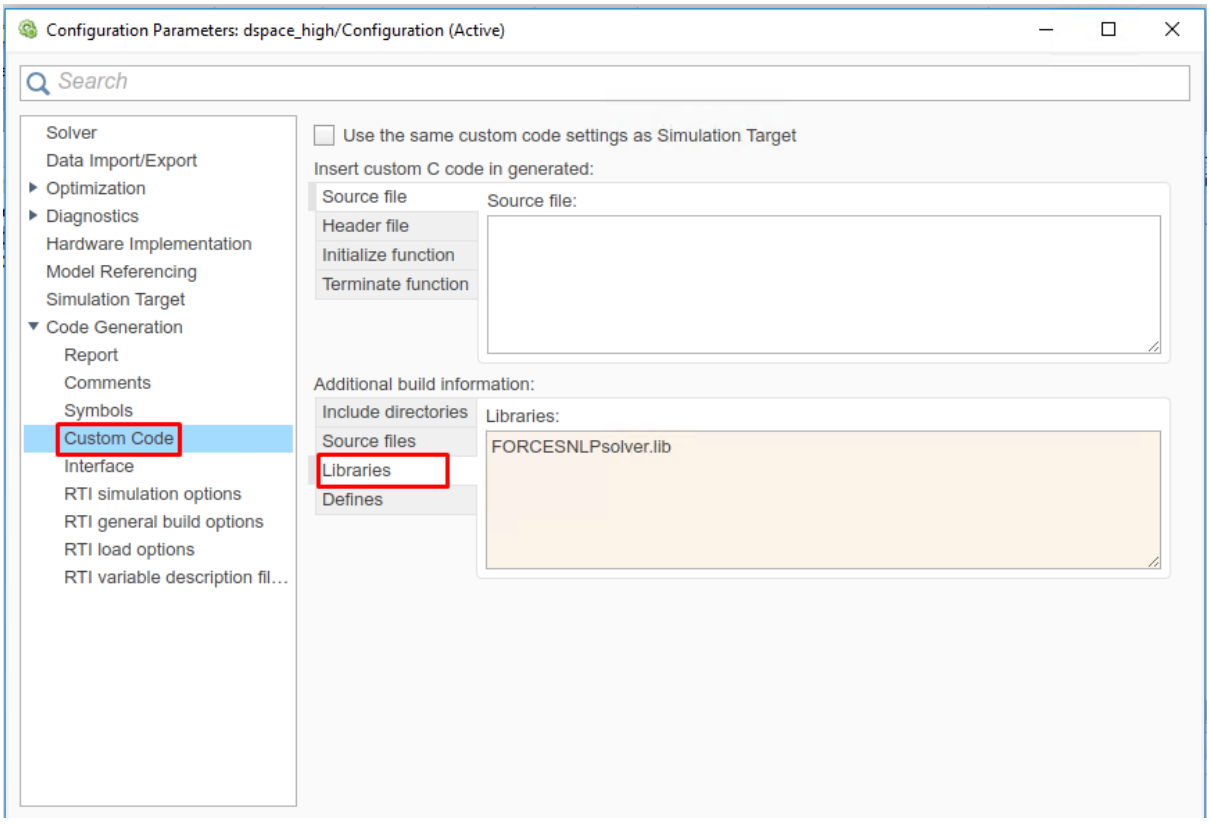


Figure 14.12: Add the libraries used for the code generation.

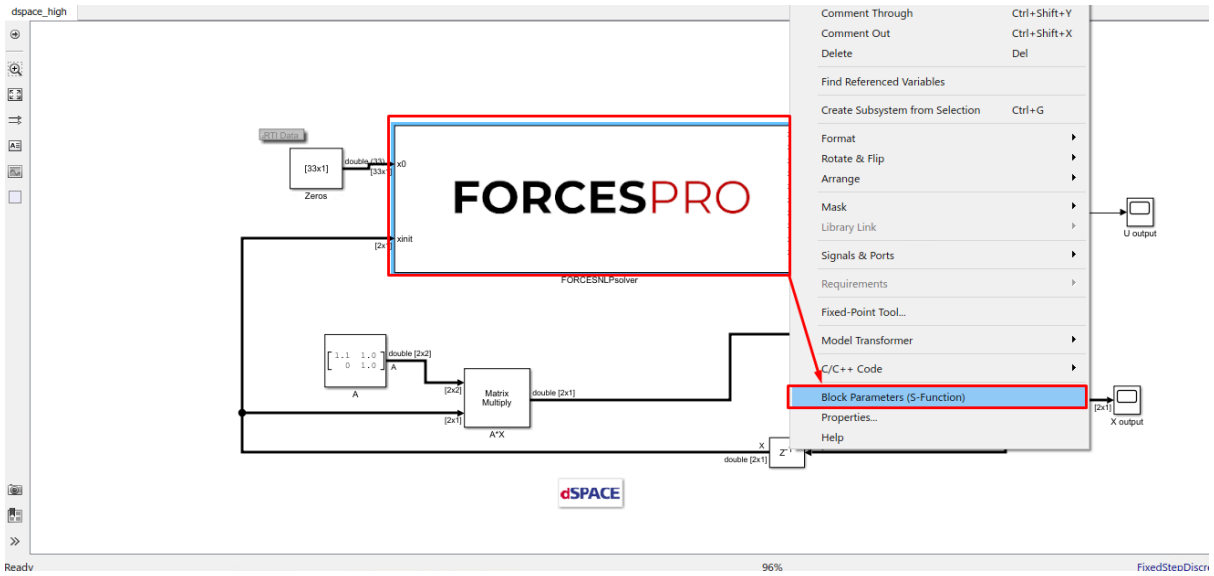


Figure 14.13: Open the FORCESPRO block's parameters.

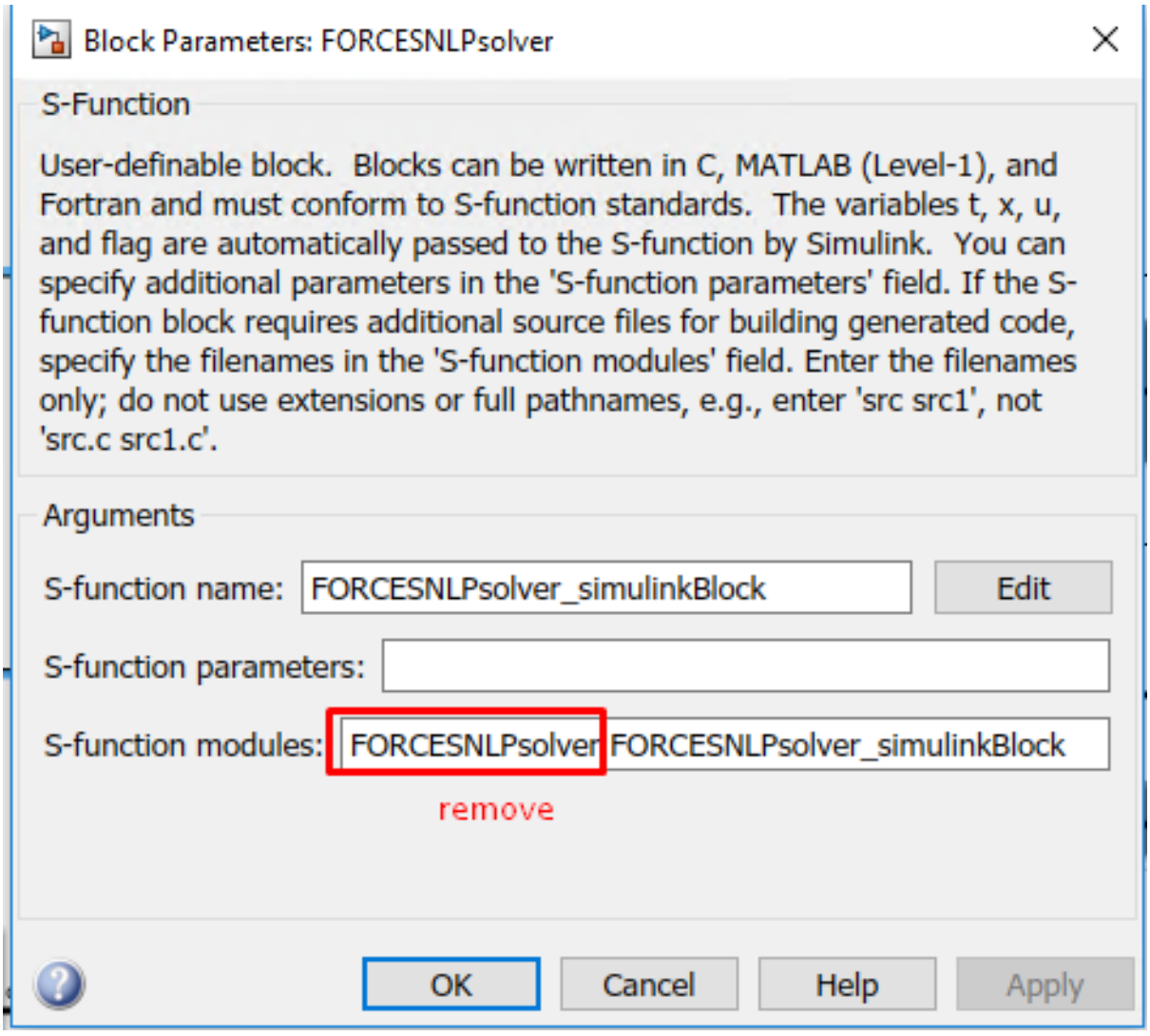


Figure 14.14: Remove the leading solver name from the S-function module.

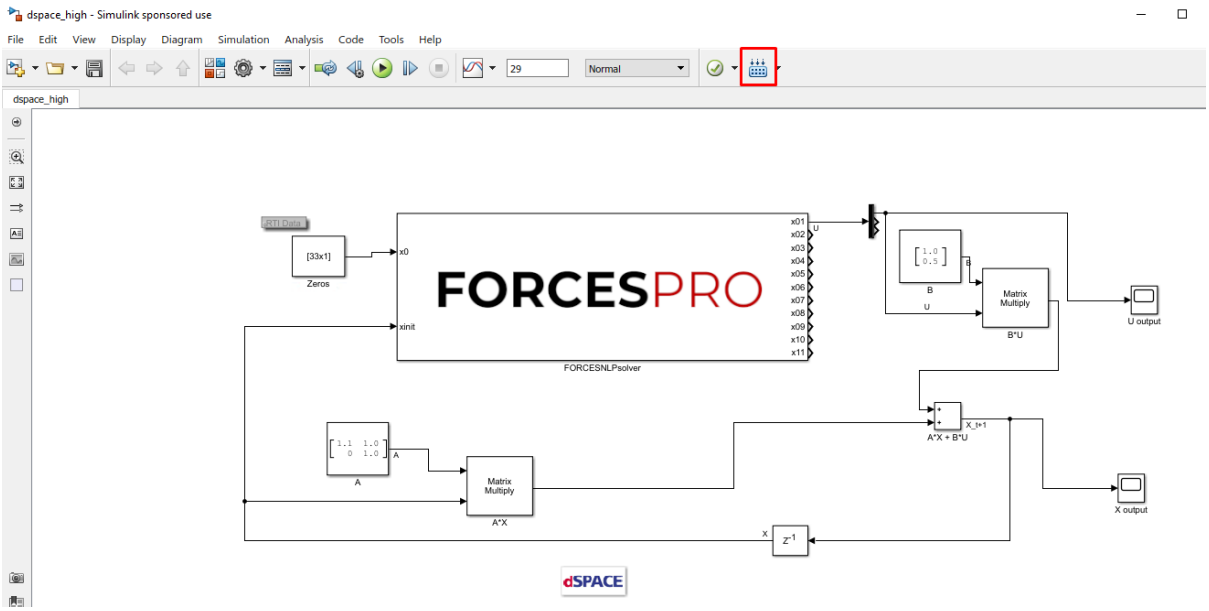


Figure 14.15: Compile the code of the Simulink model.

14.2.3 Y2F interface

Instructions

The steps to deploy and simulate a FORCESPRO controller on a dSPACE platform are detailed below.

1. (Figure 14.16) Set the code generation options (for <platform_name> see *Platform name codeoption*):

```
codeoptions.platform = '<platform_name>'; % to specify the platform
codeoptions.printlevel = 0; % on some platforms printing is not supported
```

and then generate the code for your solver (henceforth referred to as "simplempc_solver", placed in the folder "Y2F") using the Y2F interface.

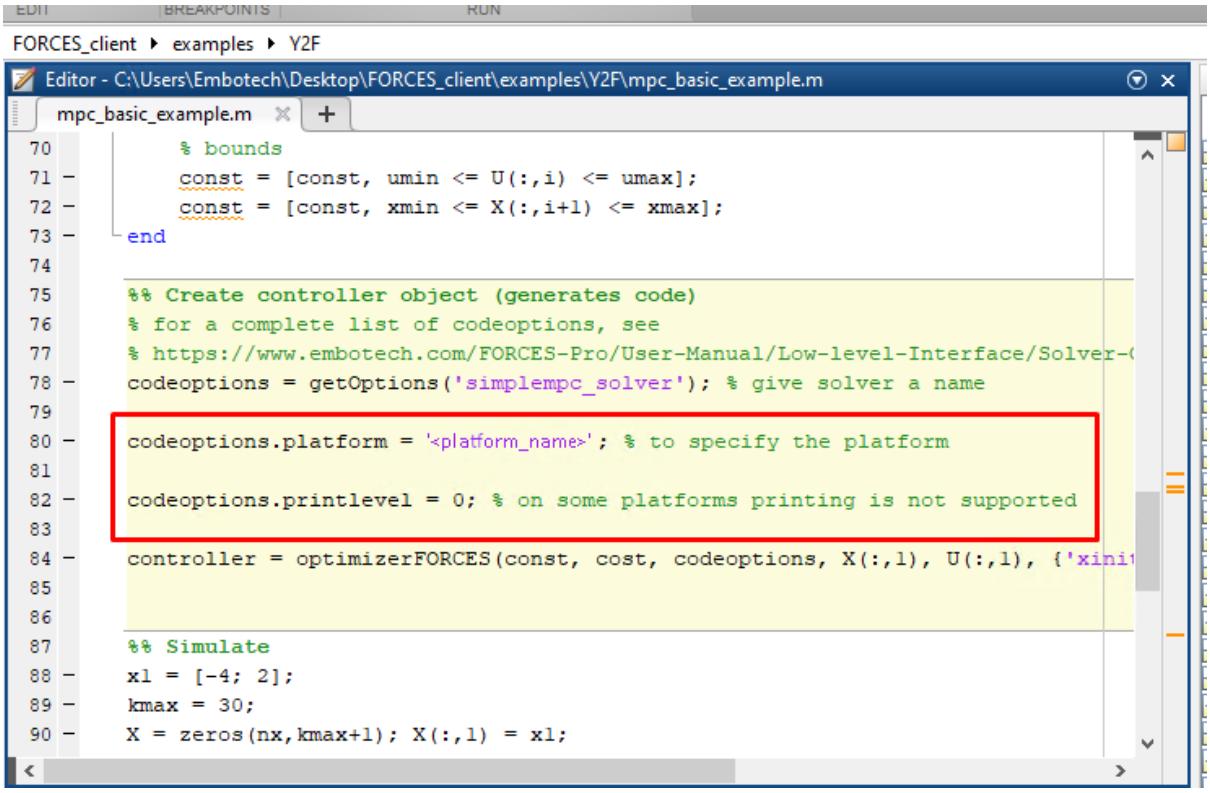


Figure 14.16: Set the appropriate code generation options.

2. (Figure 14.17) Create a new Simulink model using the Simulink model template provided by dSPACE (for <simulink_model_template> see *Simulink Model HW Target Configuration*).
3. (Figure 14.18) Populate the Simulink model with the system you want to control.
4. (Figure 14.19) Make sure the `simplempc_solver_simulinkBlock.mexw64` file (created during code generation) is on the MATLAB path.
5. (Figure 14.20) Copy-paste the FORCESPRO Simulink block, contained in the created `y2f_simulink_lib.slx` Simulink model file, into your simulation model and connect its inputs and outputs appropriately.
6. (Figure 14.21) Access the Simulink model's options.
7. (Figure 14.22) In the "Solver" tab, set the options:

- Simulation start/stop time: Depending on the simulation wanted.

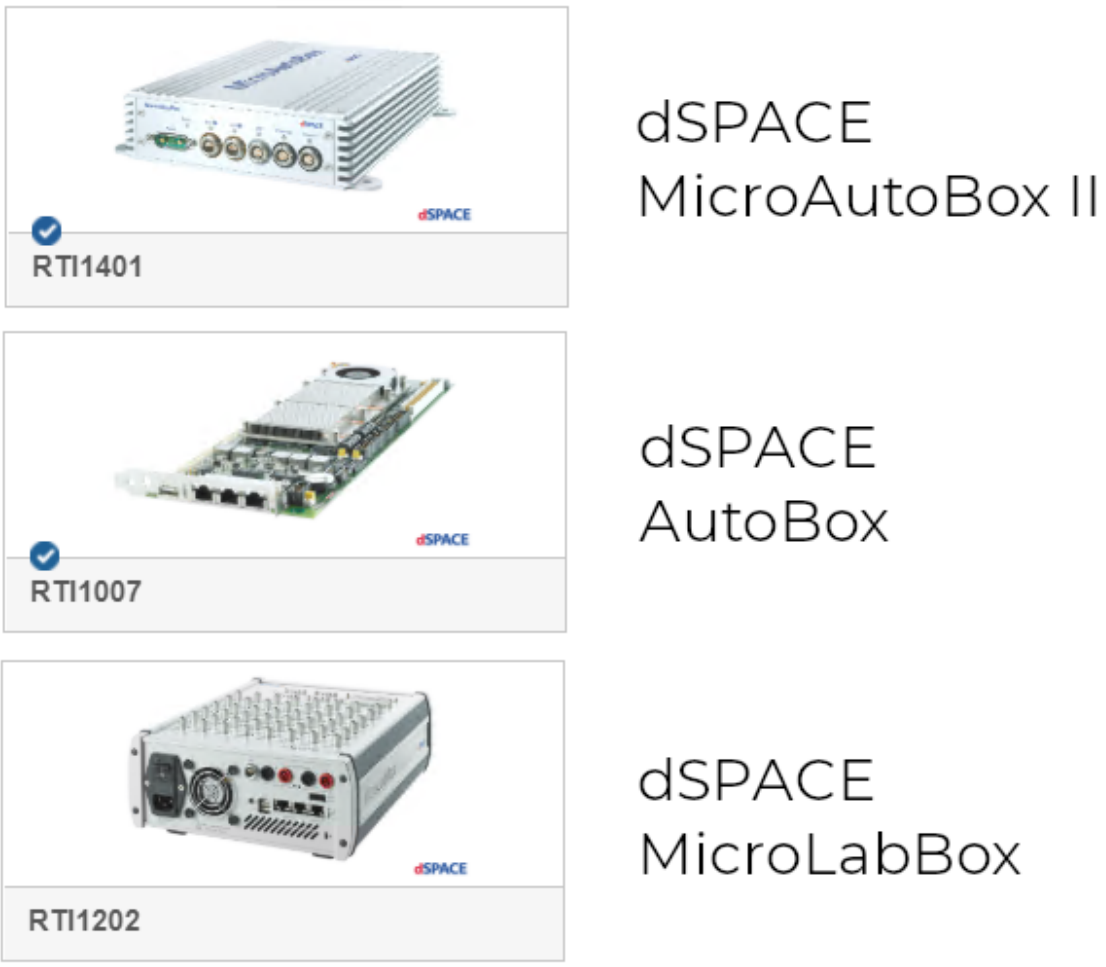


Figure 14.17: Create a Simulink model.

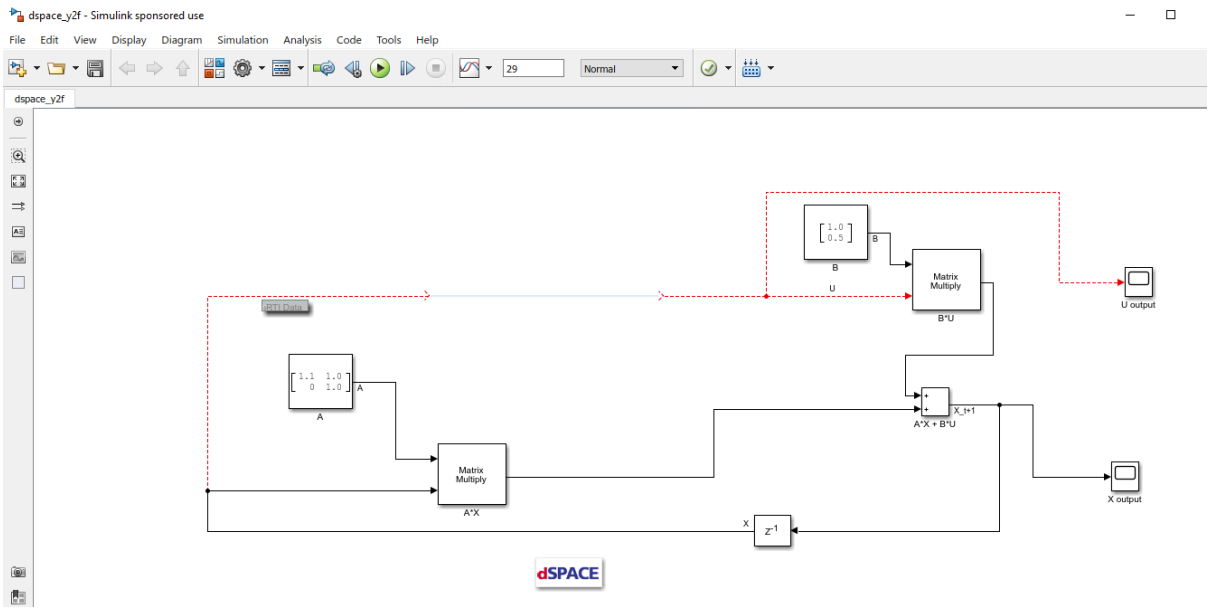


Figure 14.18: Populate the Simulink model.

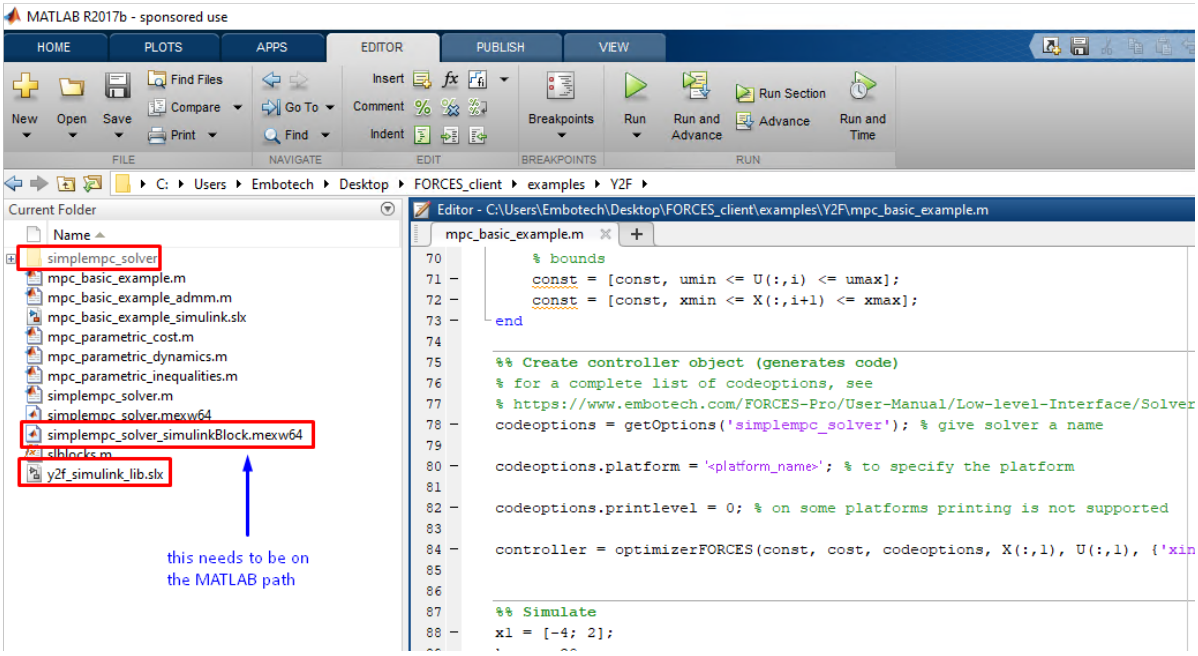


Figure 14.19: Add the folder containing the `.mexw64` solver file to the MATLAB path.

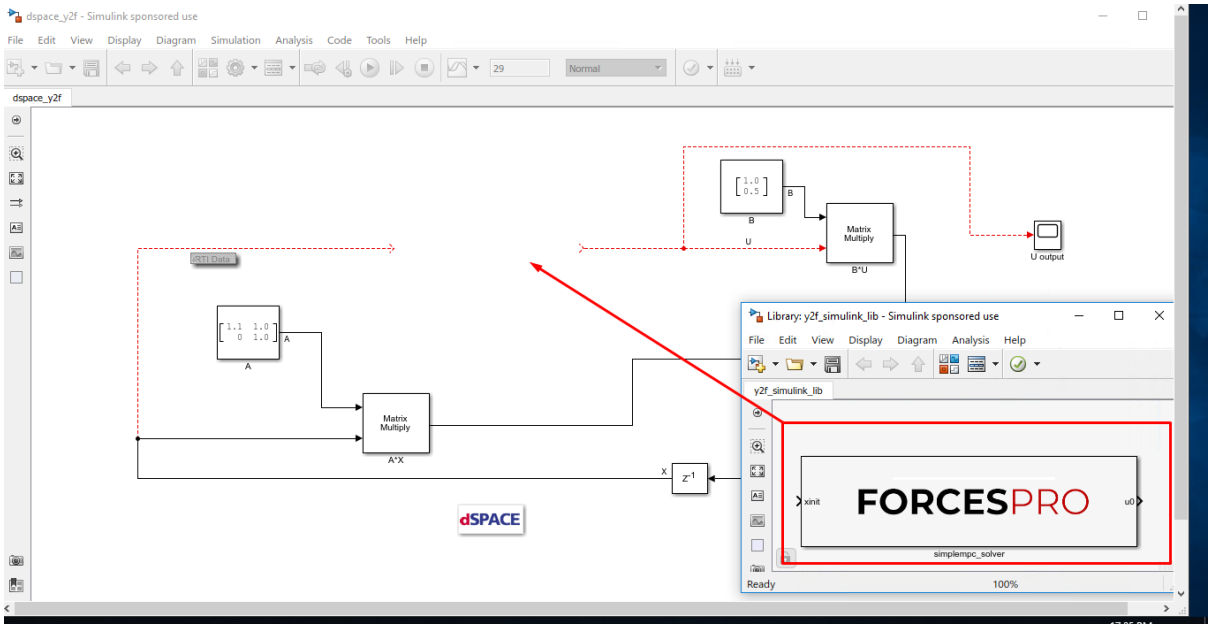


Figure 14.20: Copy-paste and connect the FORCESPRO block.

- Solver type: Discrete or fixed-step.
- Fixed-step size: Needs to be higher than the execution time of the solver.

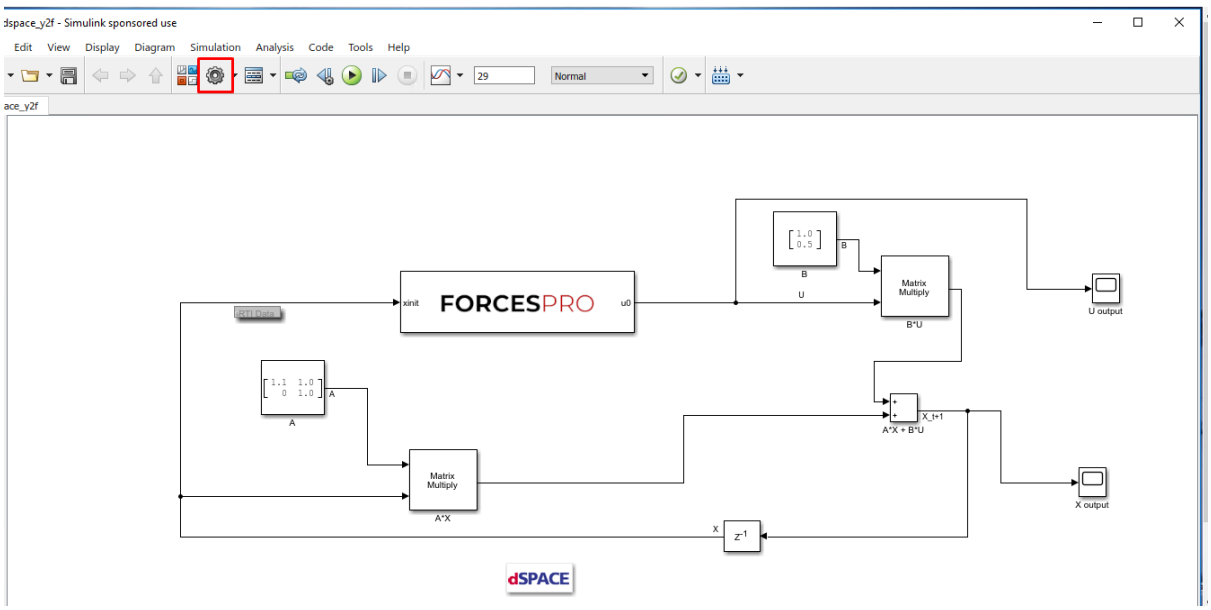


Figure 14.21: Open the Simulink model options.

8. (Figure 14.23) In the "Code Generation/RTI general build options" tab, set the options (for <`tlc_file`> and <`makefile_template`> see *Simulink Model HW Target Configuration*):
   - System target file: <`tlc_file`>
   - Language: C
   - Generate makefile: On
   - Template makefile: <`makefile_template`>
   - Make command: `make_rti`
9. (Figure 14.24) In the "Code Generation/Custom Code" tab, include the directories:
   - `Y2F`
   - `Y2F\simplempc_solver\interface`
   - `Y2F\simplempc_solver\lib_target`
10. (Figure 14.25) In the "Code Generation/Custom Code" tab, add the source files:
    - `simplempc_solver_simulinkBlock.c`
    - `simplempc_solver.c`
11. (Figure 14.26) In the "Code Generation/Custom Code" tab, add the library files:
    - `i_simplempc_solver_1.lib`
12. (Figure 14.27) Compile the code of the Simulink model. This will also automatically load the model to the connected dSPACE platform.
13. Deployment is complete and simulations can now be run on the dSPACE platform.

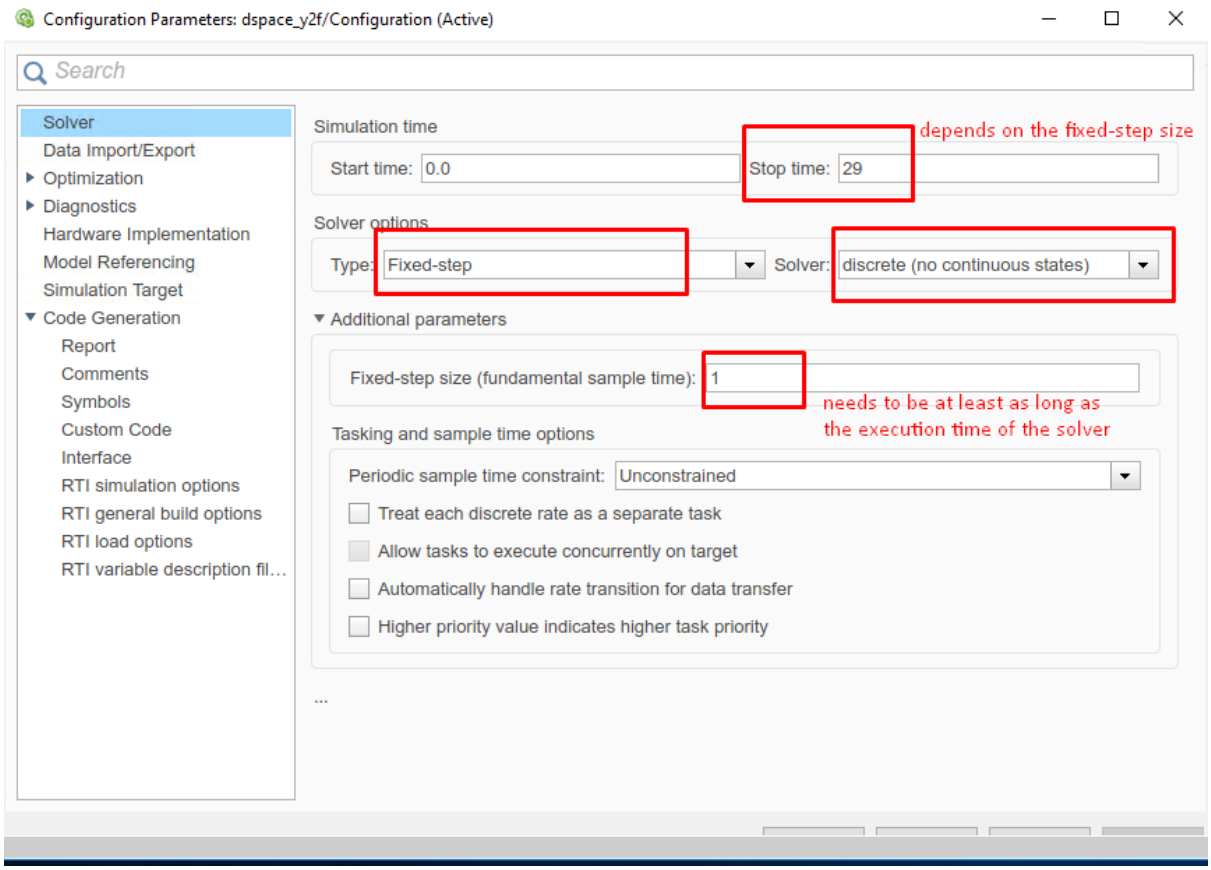


Figure 14.22: Set the Simulink solver options.

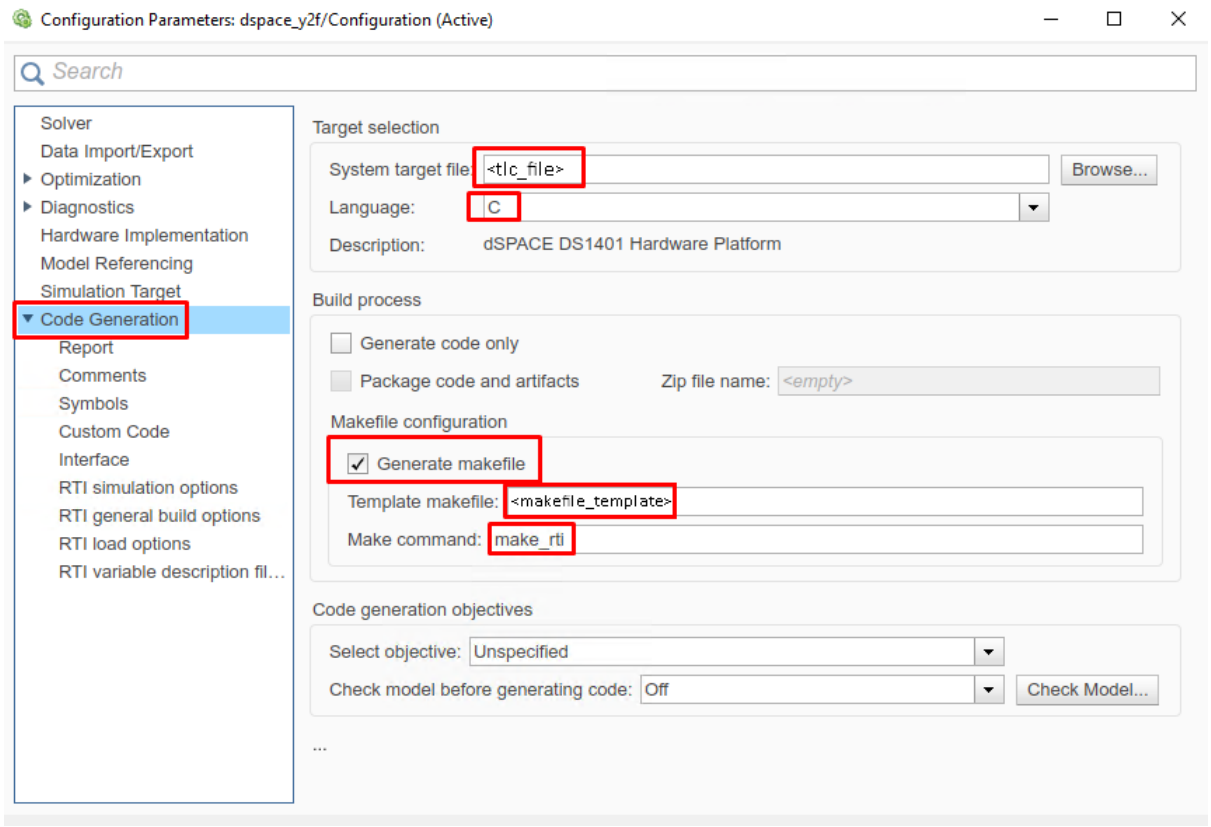


Figure 14.23: Set the Simulink code generation options.

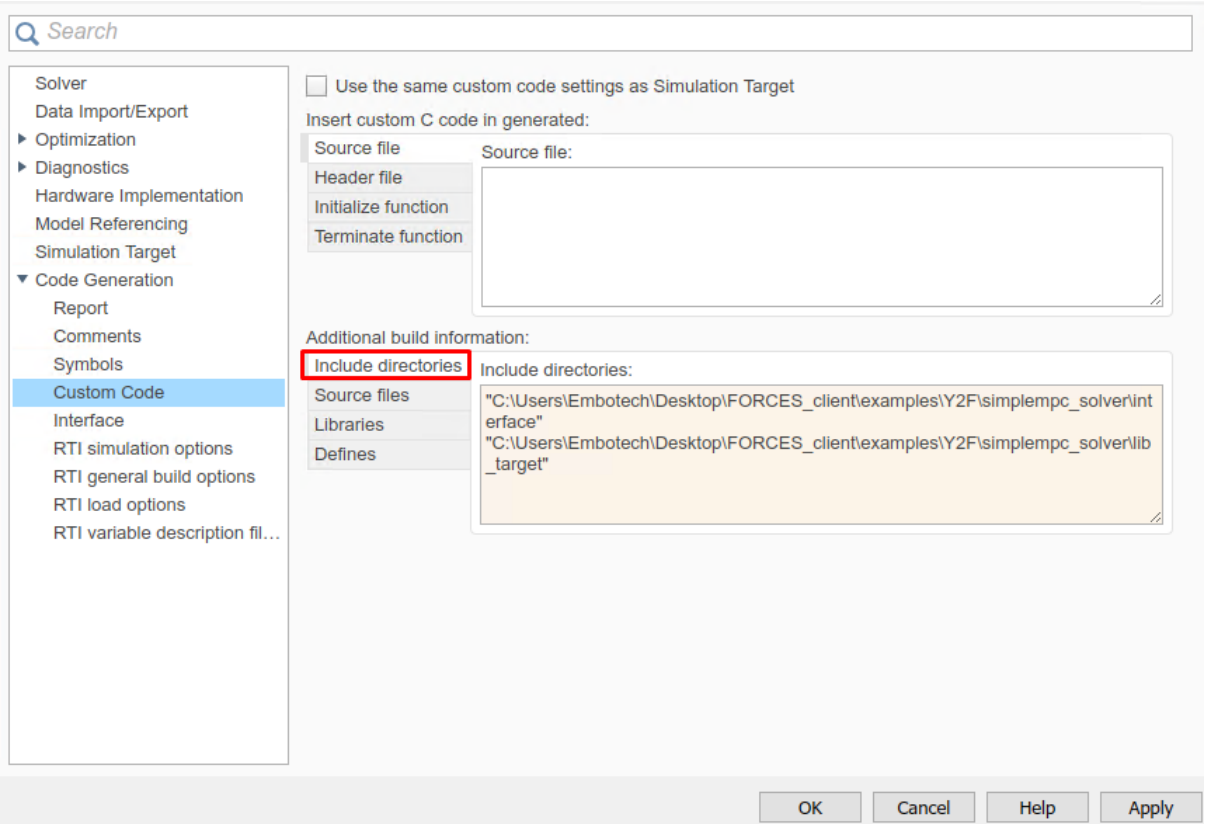


Figure 14.24: Add the directories included for the code generation.

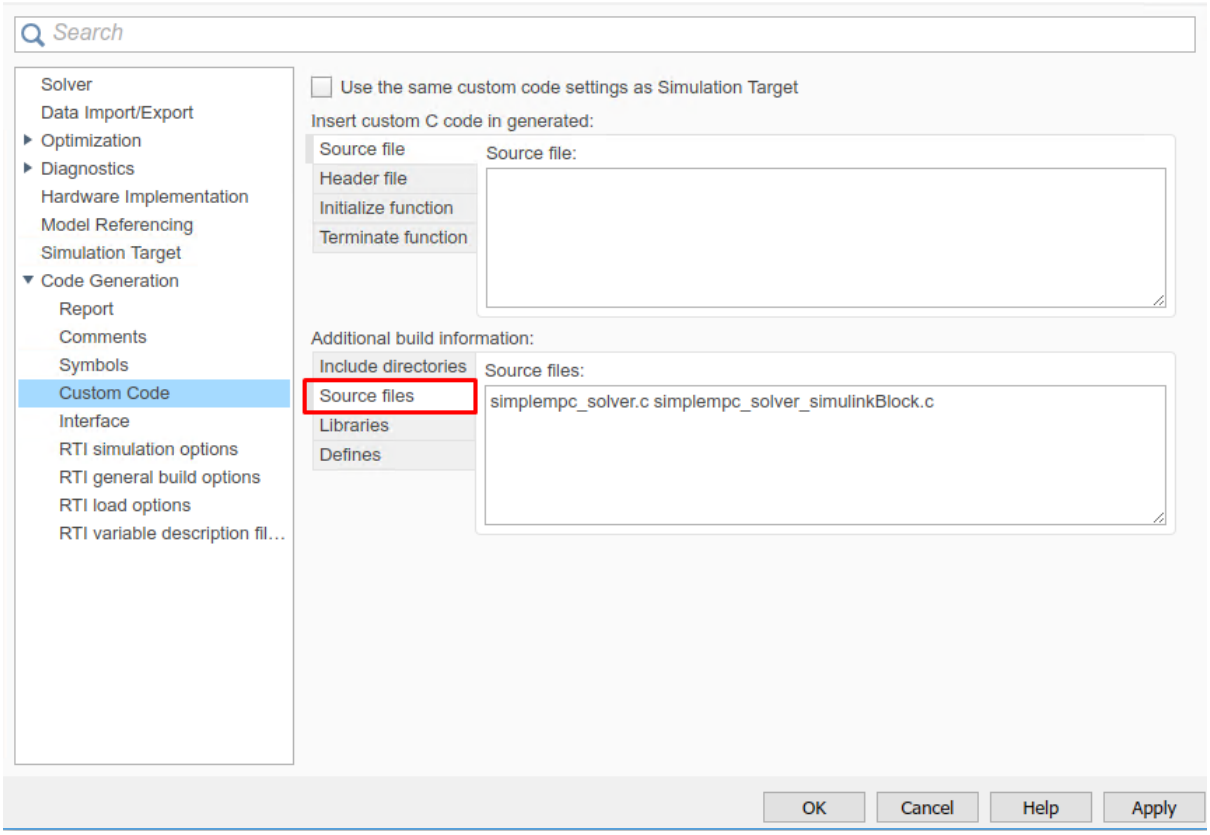


Figure 14.25: Add the source files used for the code generation.

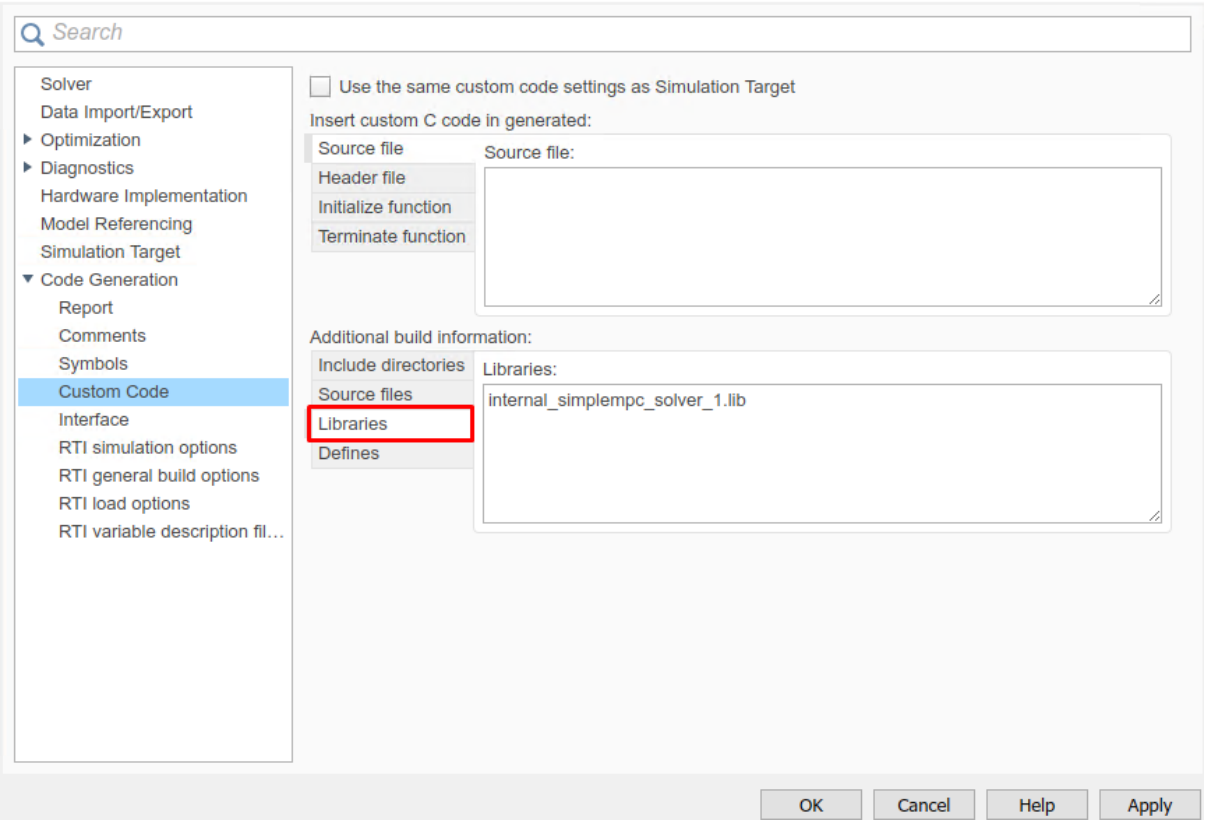


Figure 14.26: Add the libraries used for the code generation.

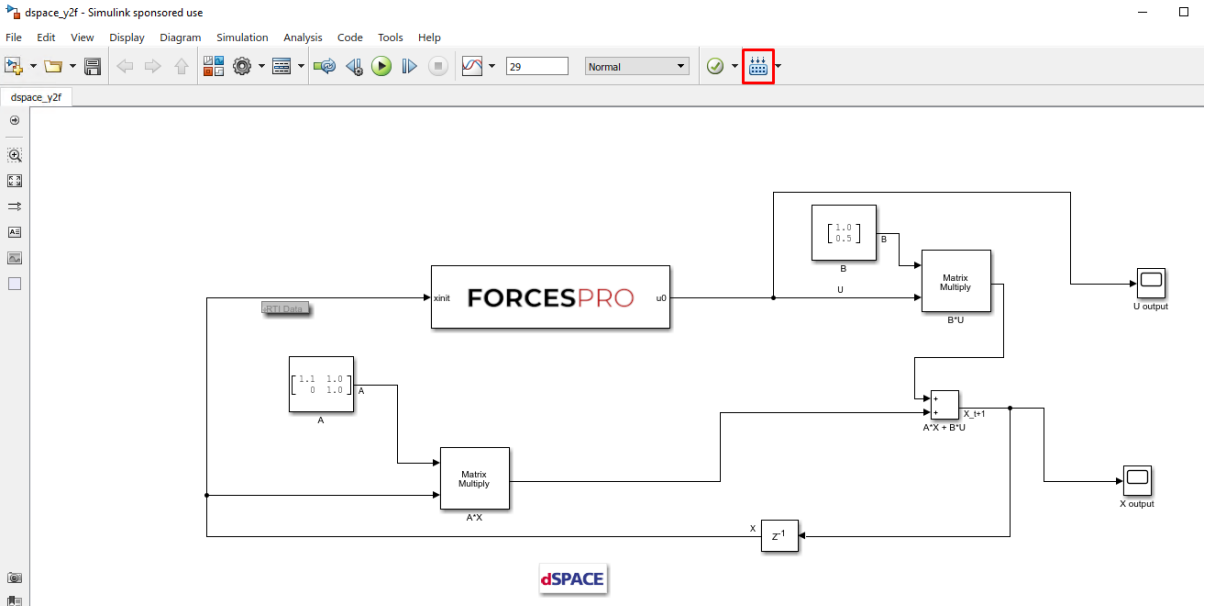


Figure 14.27: Compile the code of the Simulink model.

14.3 dSPACE deployment through ConfigurationDesk

- *Code Generation*
- *Solver Execution*

This process applies to the following dSPACE platforms

- **dSPACE MicroAutoBox III**
- **dSPACE SCALEXIO**

Important: When deploying to a target hardware platform, the library included in the **lib_target** directory of the generated solver should be used instead of the library in the **lib** directory.

14.3.1 Code Generation

The steps to deploy a FORCESPRO controller on a dSPACE platform are detailed below.

1) (Figure 14.28) Set the code generation options:

When generating code for HW target platforms, `codeoptions.platform` needs to be set.

- **dSPACE MicroAutoBox III**: `'dSPACE-MABXIII'`
- **dSPACE SCALEXIO**: `'dSPACE-SCALEXIO'`

```
codeoptions.platform = '<platform_name>'; % to generate code for the dSPACE platform
codeoptions.printlevel = 0; % printing should be disabled on target HW
codeoptions.cleanup = 0; % to keep necessary files for target compile
```

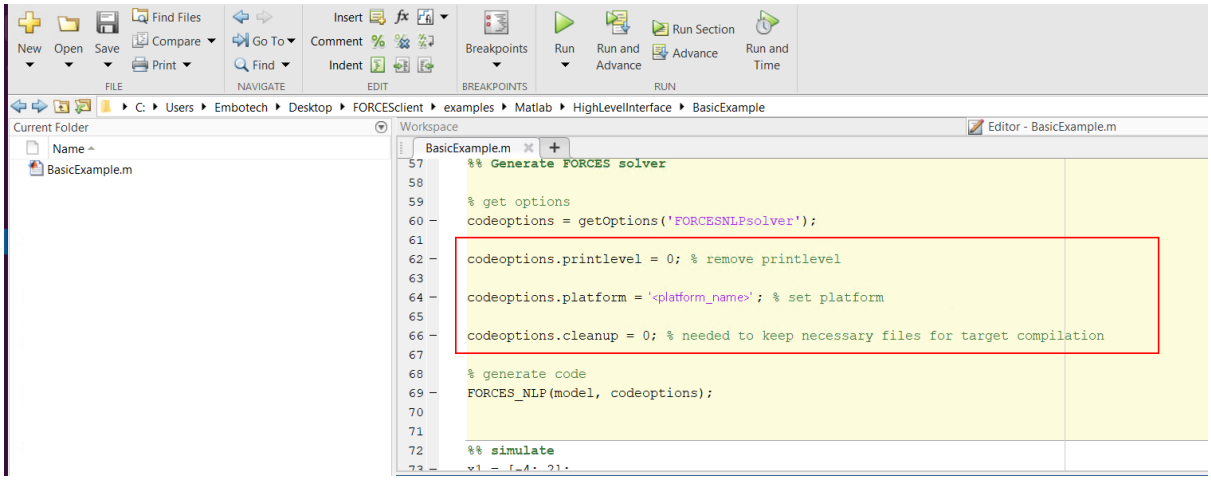


Figure 14.28: Set the appropriate code generation options.

1) Create a new Simulink model (henceforth referred to as `dSPACE.slx`) using the `dSPACE Run-Time Target` template provided by dSPACE and save it in the `BasicExample` folder (see Figure 14.29).
2) Populate the Simulink model with the system you want to control (see Figure 14.30).

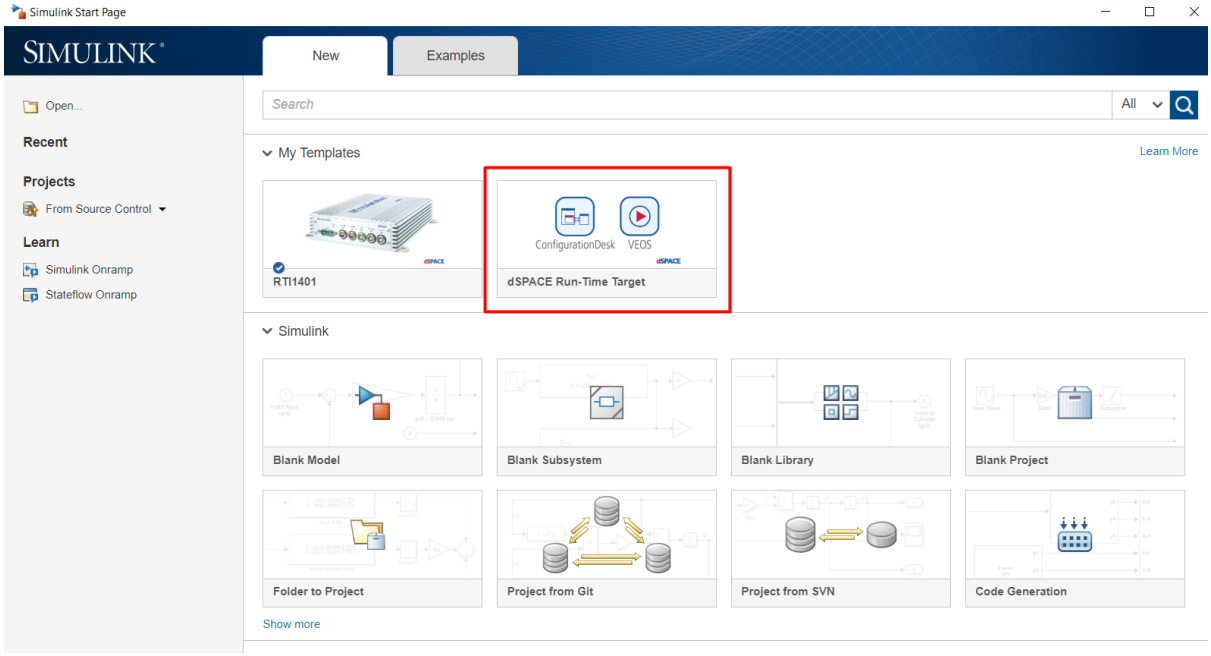


Figure 14.29: Create a Simulink model.

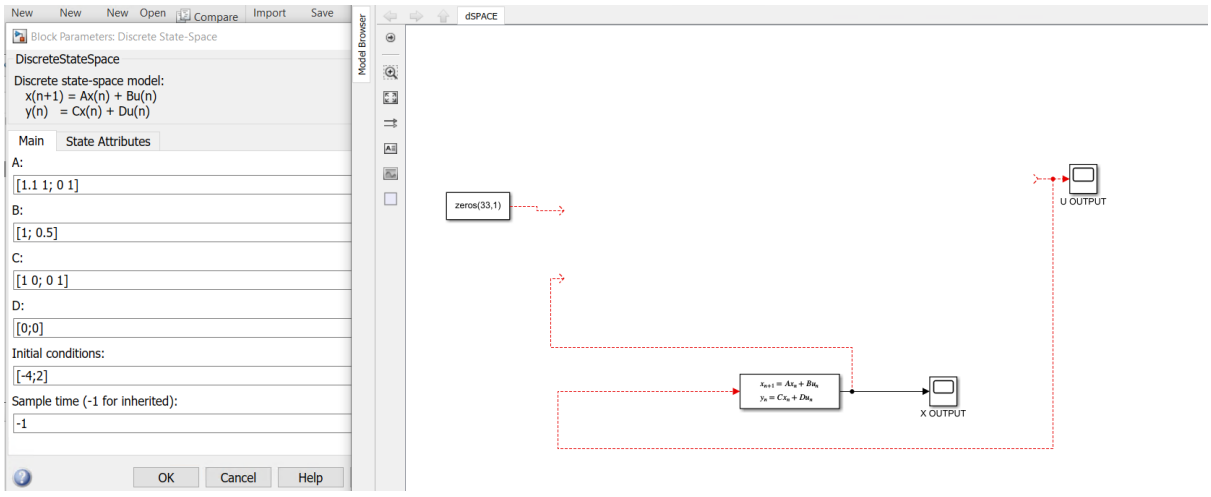


Figure 14.30: Populate the Simulink model.

4) Run the BasicExample.m script to perform code generation for your solver (henceforth referred to as `FORCESNLPsolver`, placed in the folder "BasicExample"). This will create the necessary files for your building (see Figure 14.31 , Figure 14.32 and Figure 14.33).
5) The `FORCESNLPsolver_simulinkBlock.<mex_extension>` file (created during code generation) needs to be in the same path as your model (see Figure 14.34).
6) Open the `FORCESNLPsolver_lib.mdl` Simulink model file, contained in the `interface` folder of the `FORCESNLPsolver` folder created during code generation (see Figure 14.35).
7) Copy-paste the FORCESPRO Simulink block into your simulation model and connect its inputs and outputs appropriately (see Figure 14.36).
8) Access the Simulink model's options. In the "Solver" tab, set the options (see Figure 14.37):
   - Simulation start/stop time: Depending on the simulation wanted.
   - Solver type: Discrete or fixed-step.
   - Fixed-step size: Needs to be higher than the execution time of the solver.
9) In the "Code Generation" tab, set the options (see Figure 14.38):
   - System target file: `dsrt.tlc`
   - Language: `C` / `C++`
   - Generate makefile: Checked
   - Template makefile: `dsrt_default_tmf`
   - Make command: `make_dsrt`
10) In the "Code Generation/Custom Code" tab, include the directories (see Figure 14.39):
   - `.\FORCESNLPsolver\include`
   - `.\FORCESNLPsolver\interface`
   - `.\FORCESNLPsolver\lib_target`
11) In the "Code Generation/Custom Code" tab, add the source files (see Figure 14.40):
   - `FORCESNLPsolver_simulinkBlock.c`
   - `FORCESNLPsolver_adtool2forces.c`
   - `FORCESNLPsolver_casadi.c`
12) In the "Code Generation/Custom Code" tab, add the library file (see Figure 14.41):
   - `libFORCESNLPsolver.a`
13) Access the FORCESPRO block's parameters (see Figure 14.42).
14) Remove the "FORCESNLPsolver" prefix from the S-function module (see Figure 14.43).
15) Create a new Project and Application in ConfigurationDesk. Select directory of project, name of project and application, the model `dSPACE.slx` as the application process and connected dSPACE platform to deploy to (see Figure 14.44).
16) Go to the tasks tab and make sure the period of the Periodic Task matches the fixed step size selected in the Simulink model options (see Figure 14.45).
17) Go to the build tab and start the building process. After building is complete the application will be loaded automatically in the dSPACE platform (see Figure 14.46).

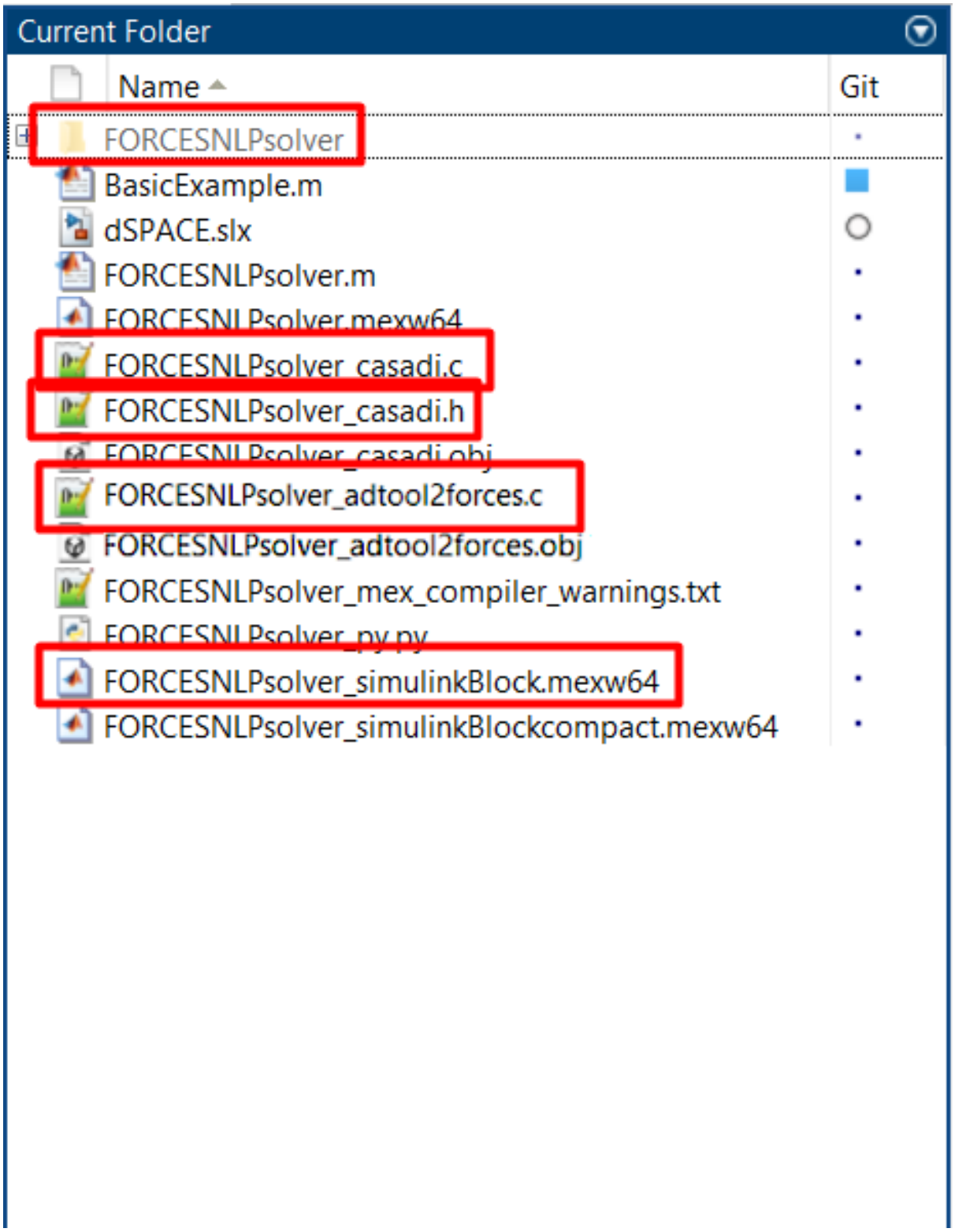


Figure 14.31: Generated files.

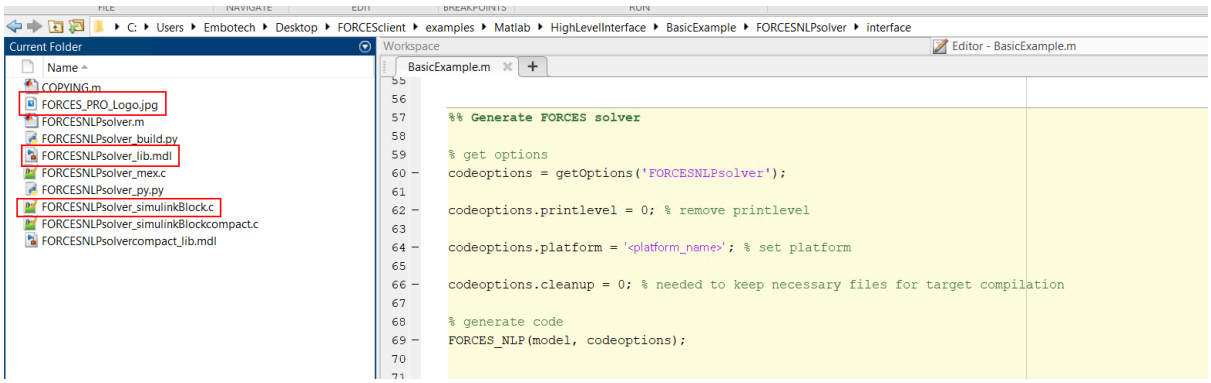


Figure 14.32: Solver interface files.

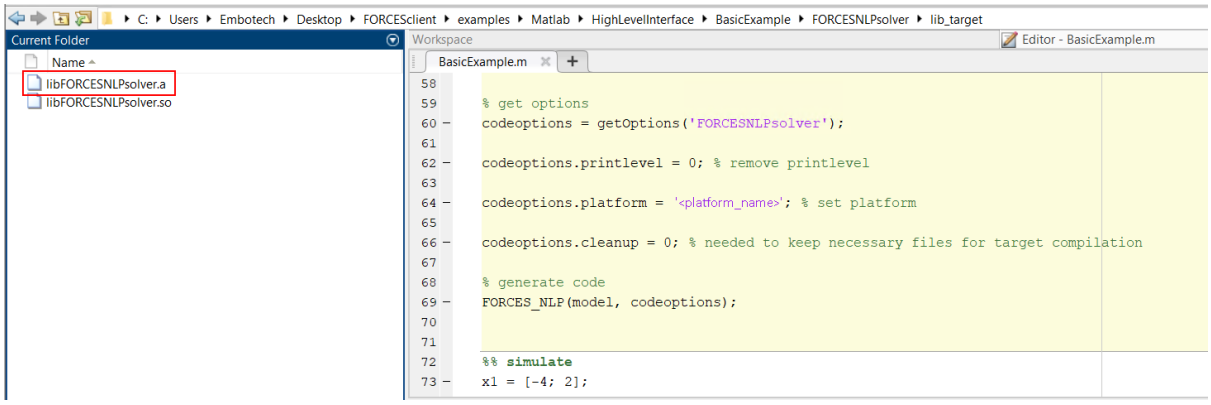


Figure 14.33: Solver libraries.

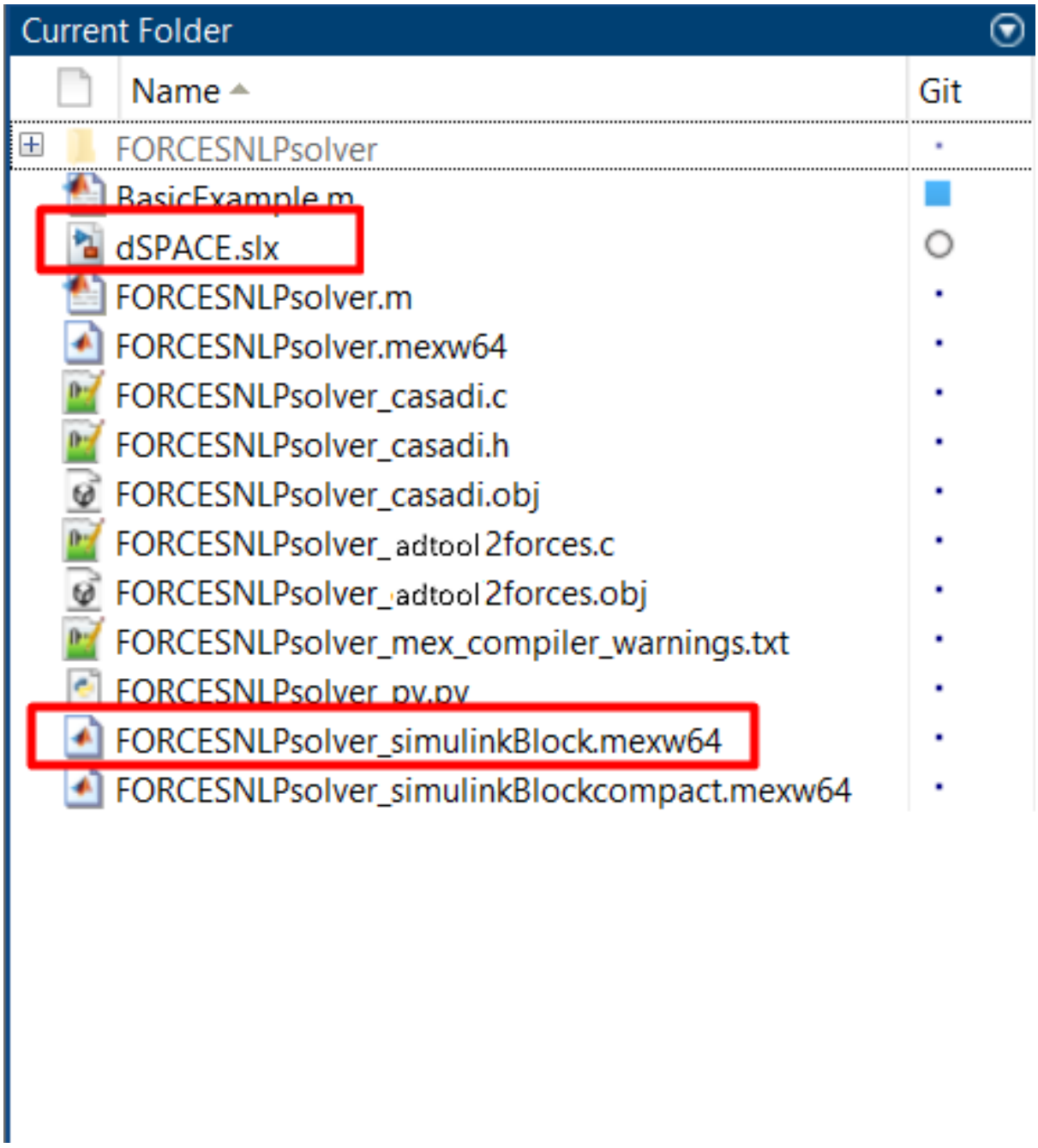


Figure 14.34: The `.<mex_extension>` solver file is in the same path as the model.

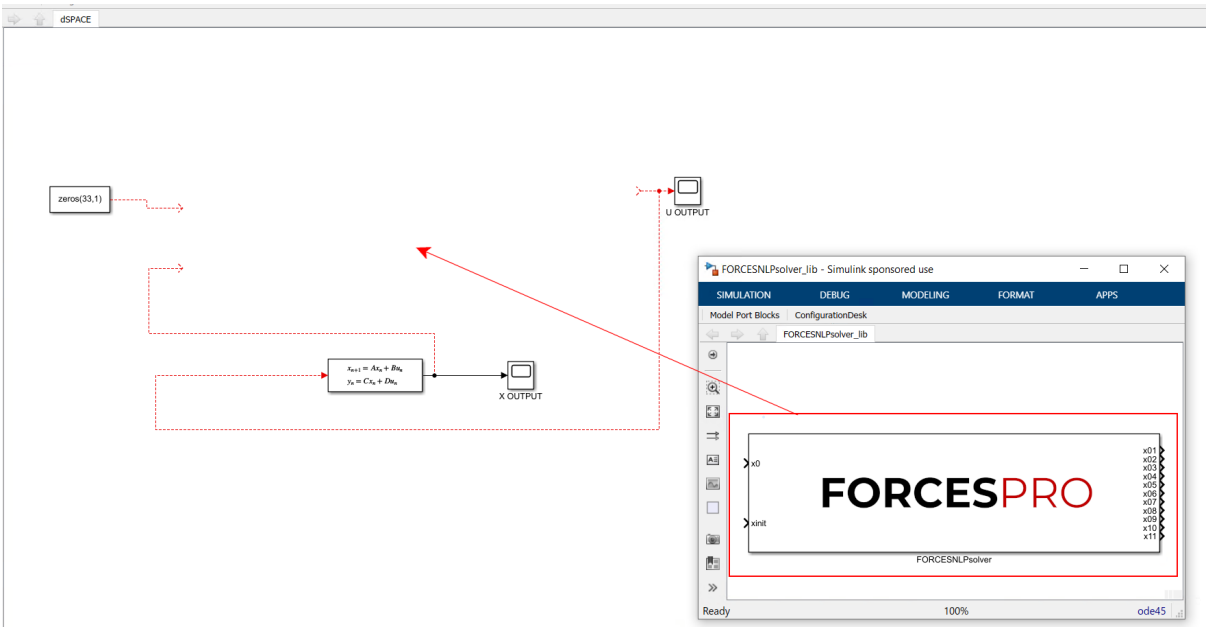


Figure 14.35: Open the generated Simulink solver model.

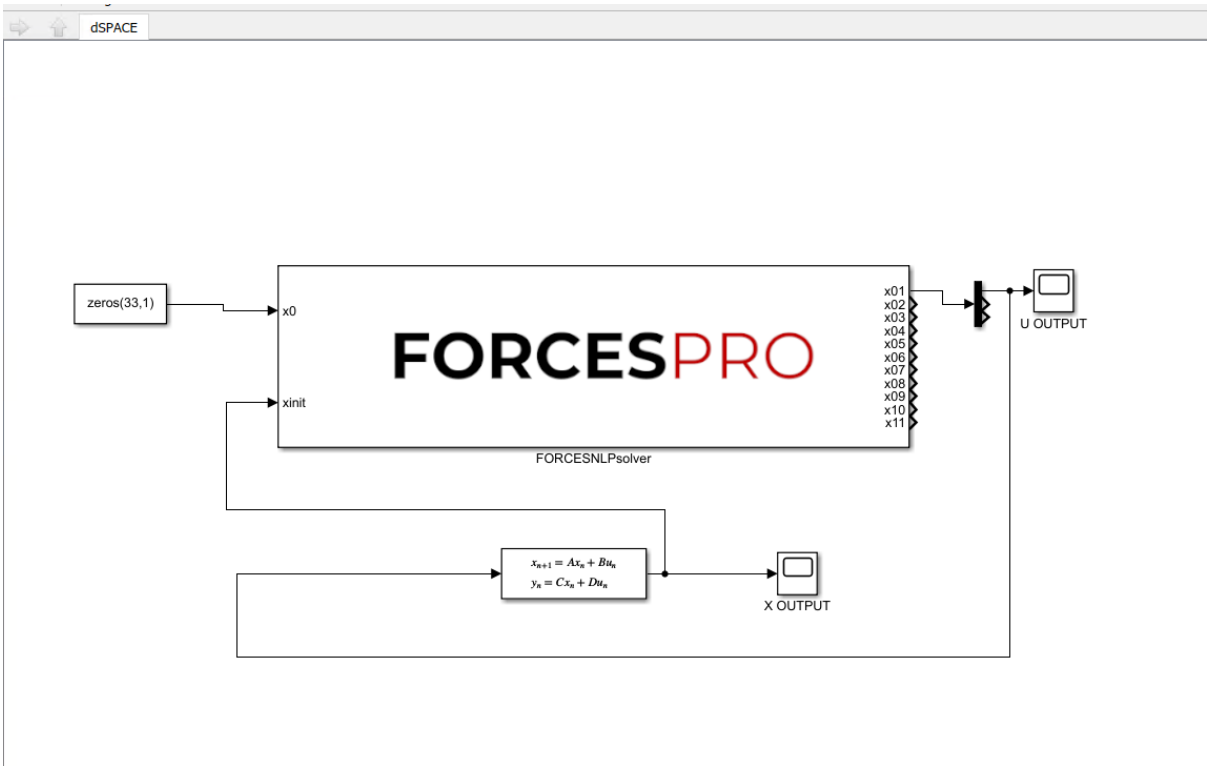


Figure 14.36: Copy-paste and connect the FORCESPRO block.

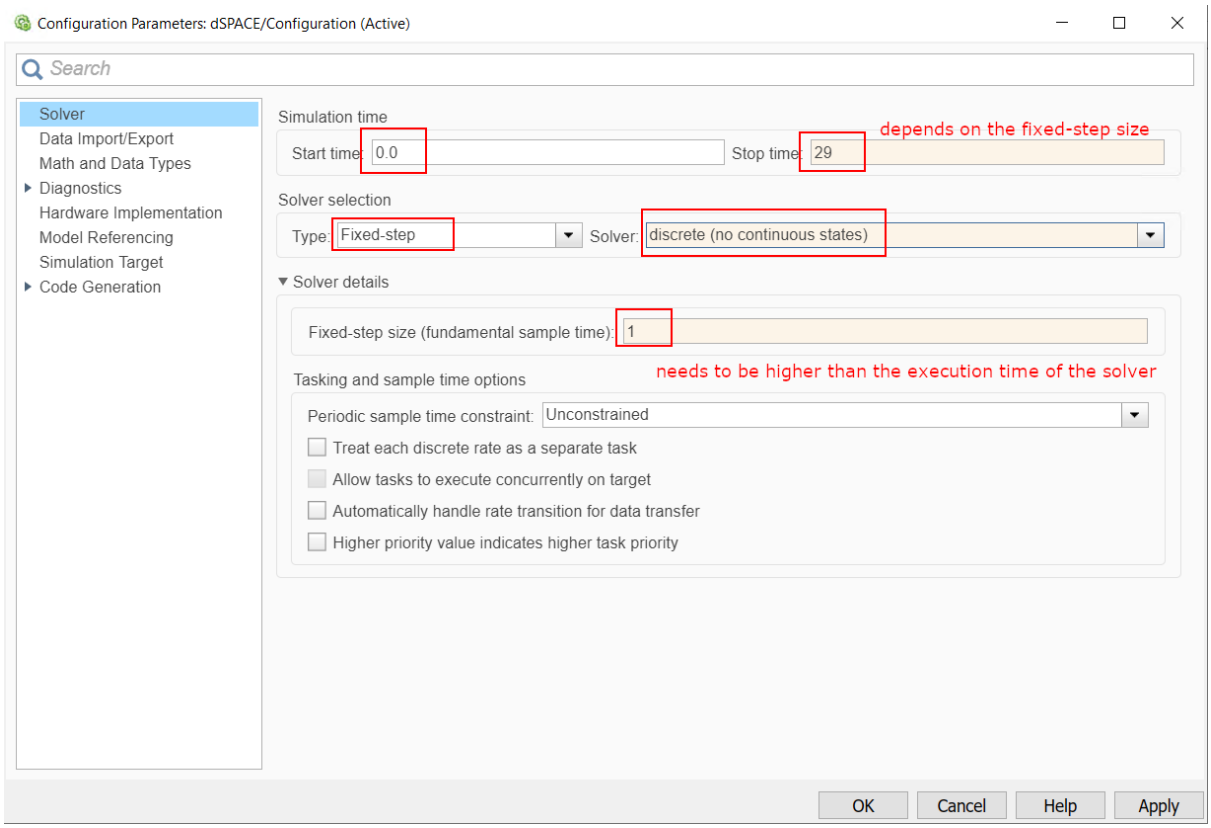


Figure 14.37: Set the Simulink solver options.

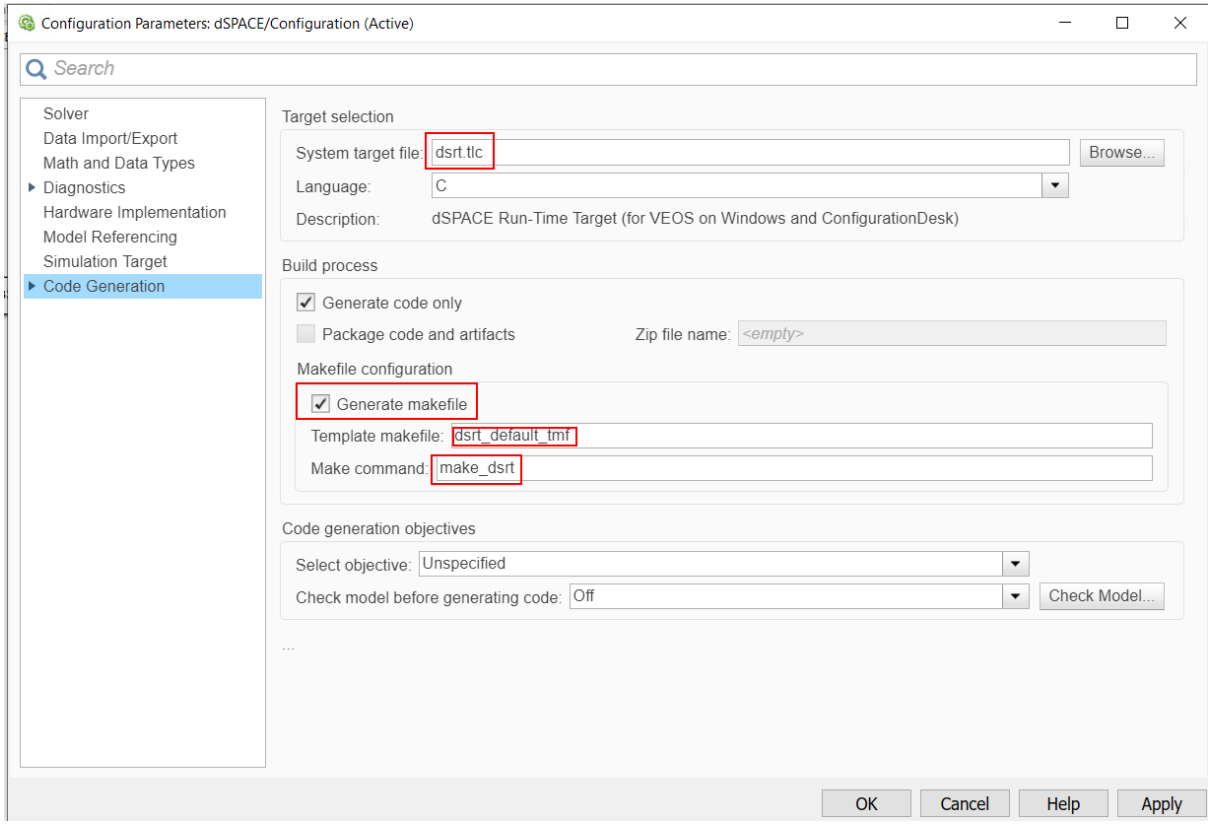


Figure 14.38: Set the Simulink code generation options.

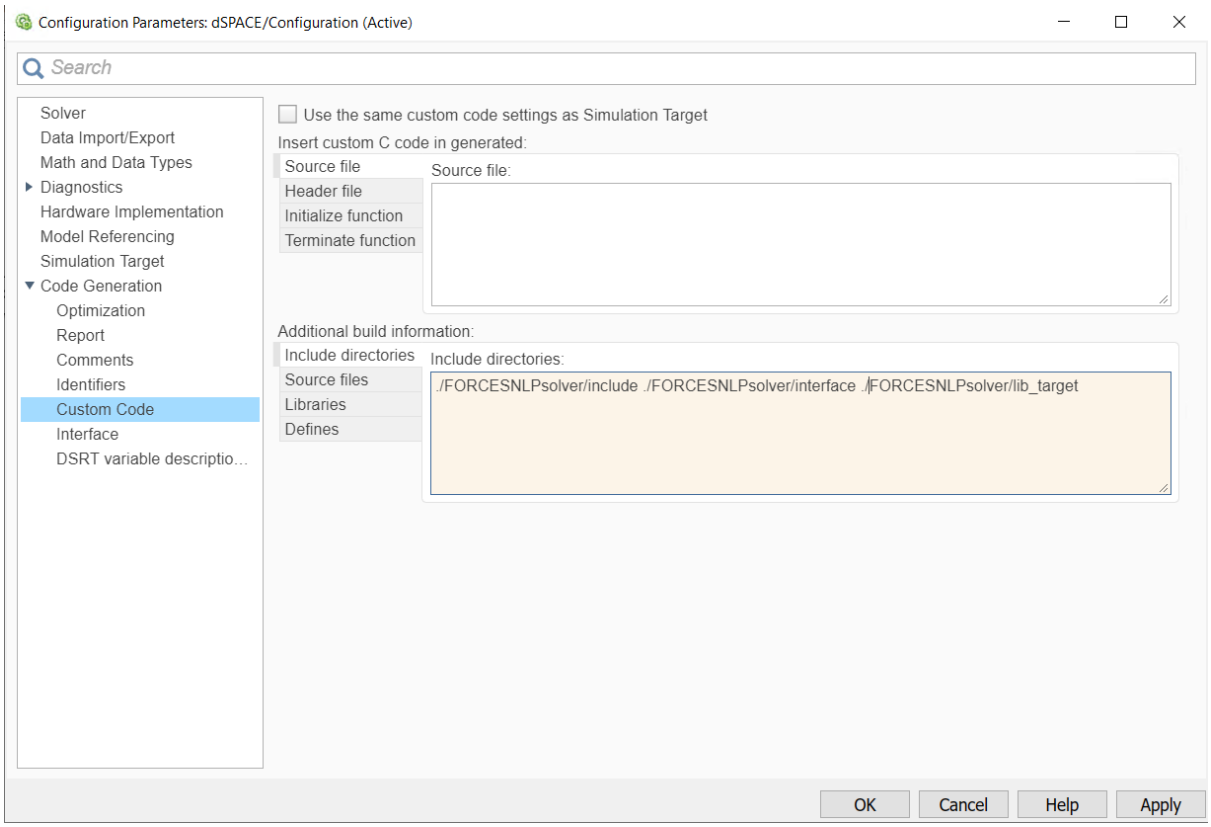


Figure 14.39: Add the directories included for the code generation.

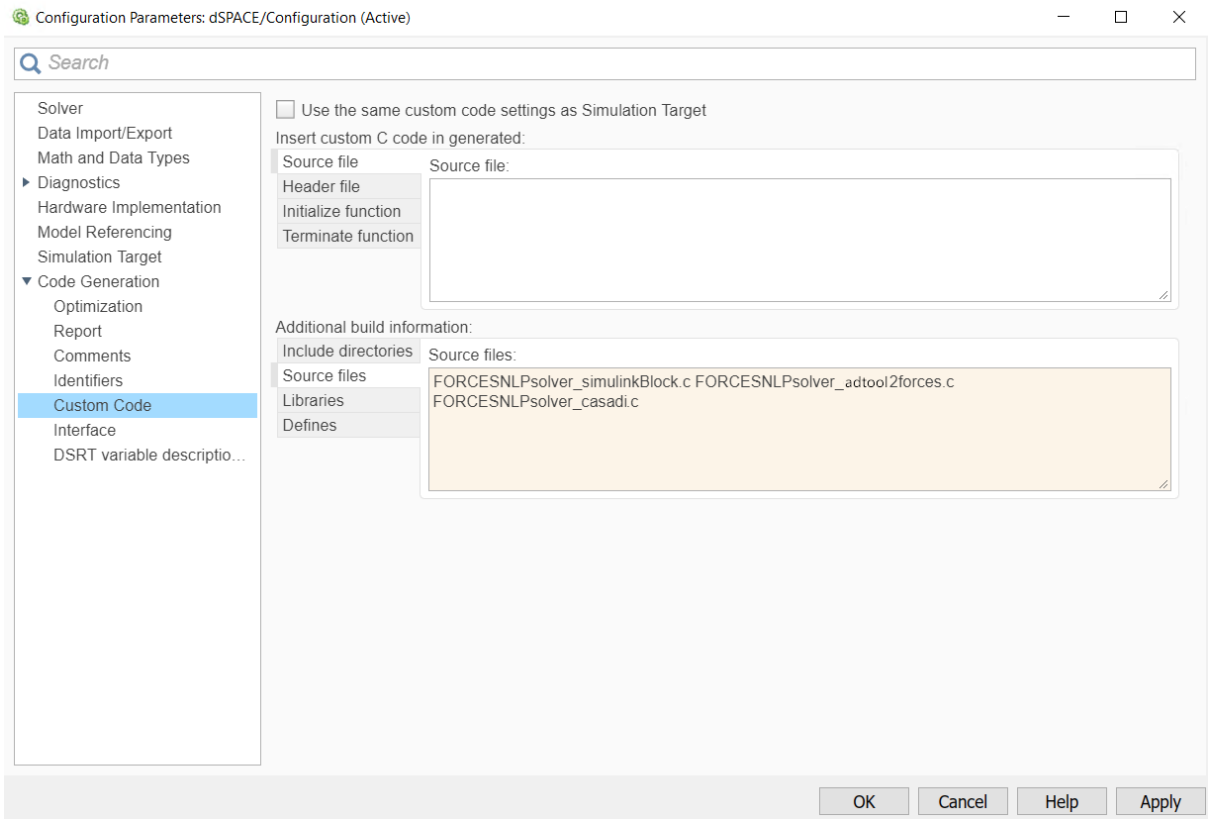


Figure 14.40: Add the source files used for the code generation.

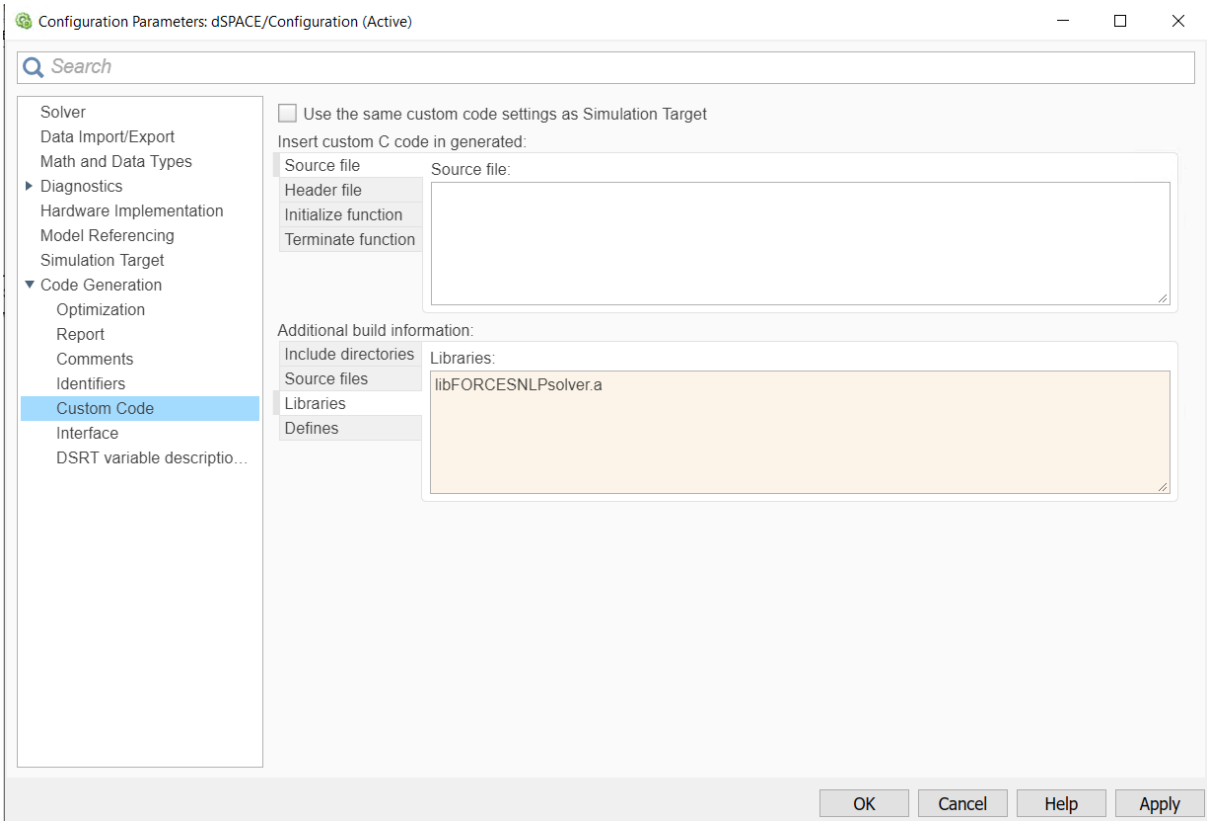


Figure 14.41: Add the libraries used for the code generation.

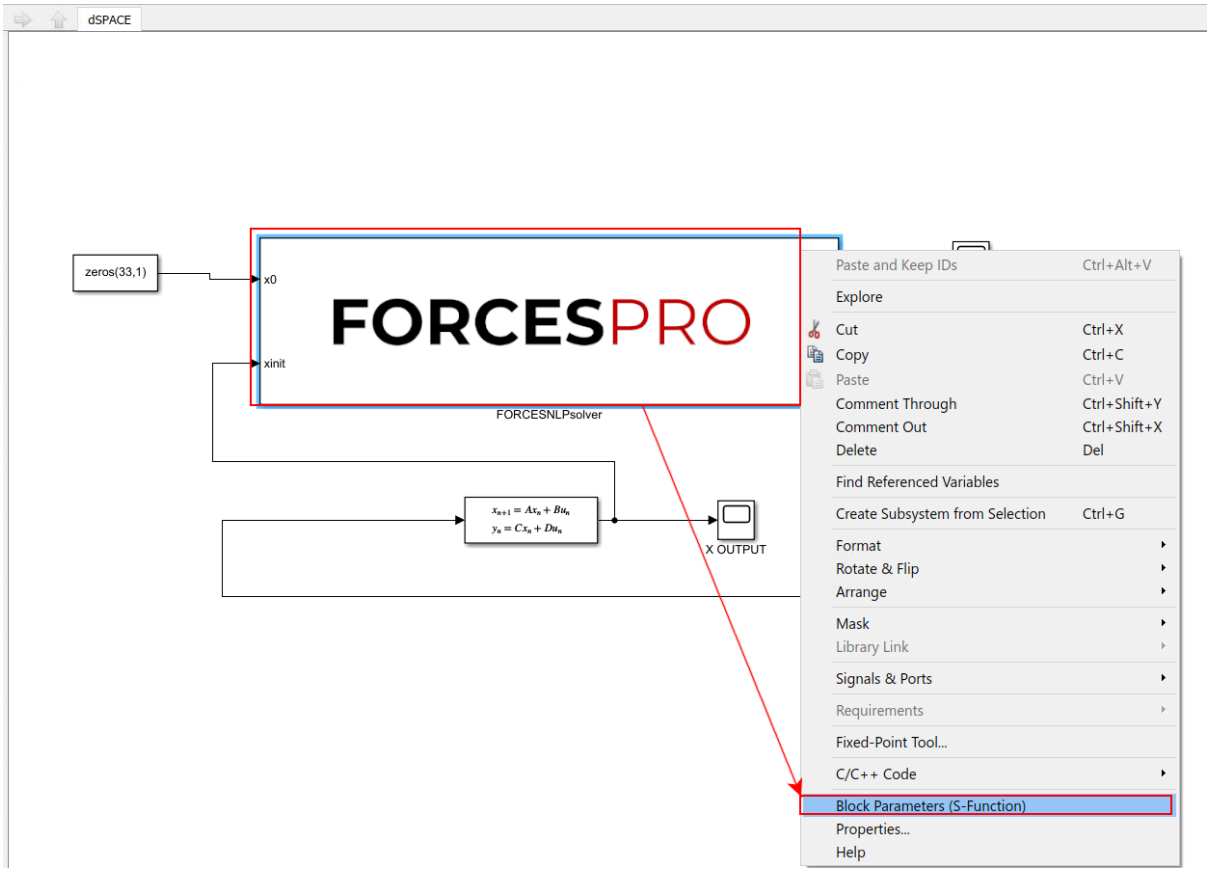


Figure 14.42: Open the FORCESPRO block's parameters.

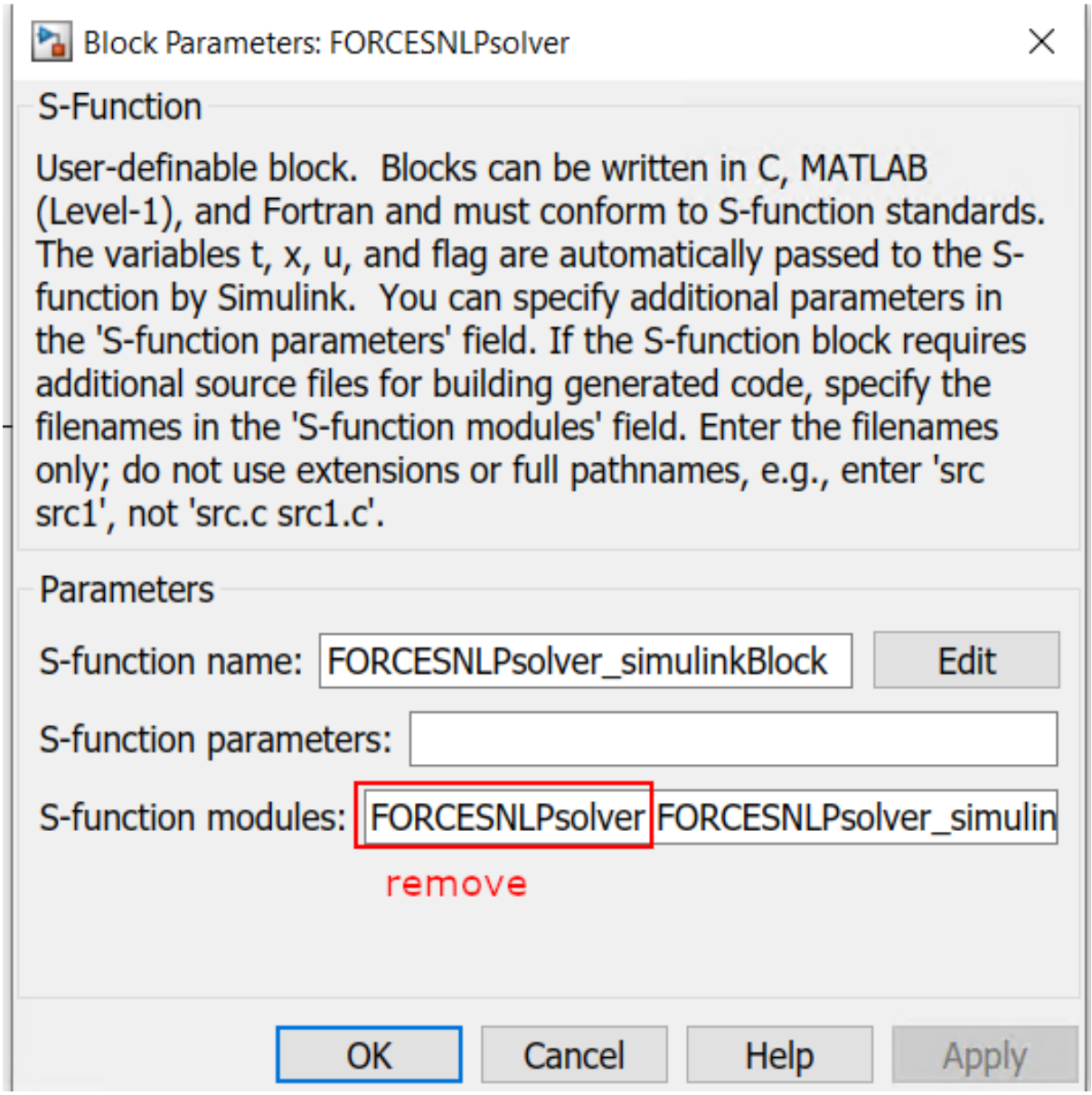


Figure 14.43: Remove the leading solver name from the S-function module.

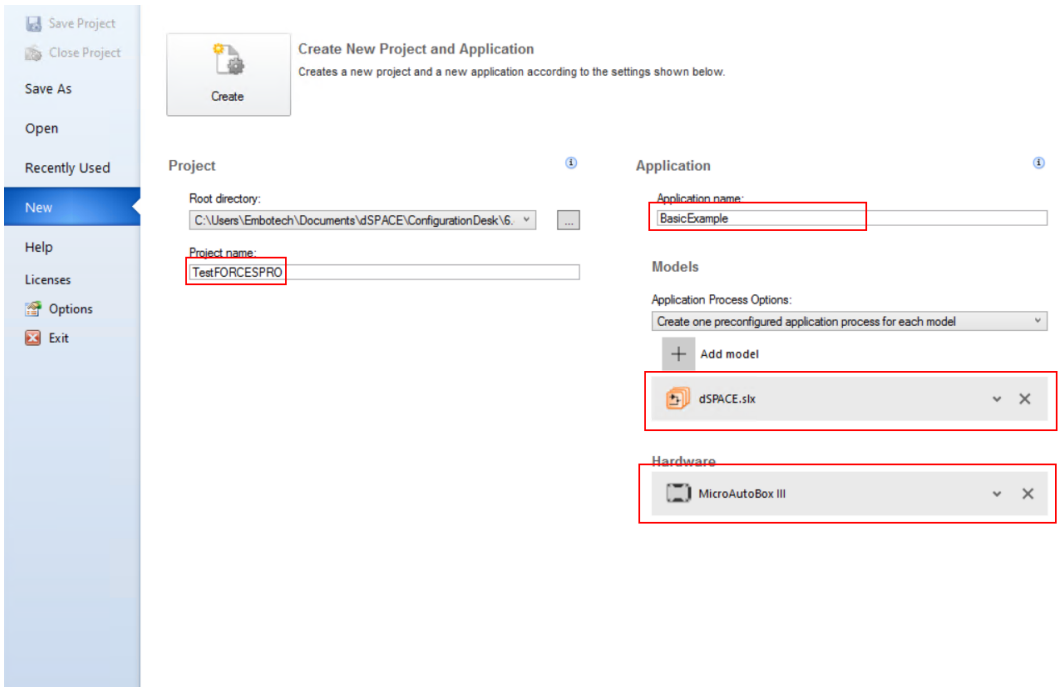


Figure 14.44: Create project and application in ConfigurationDesk.

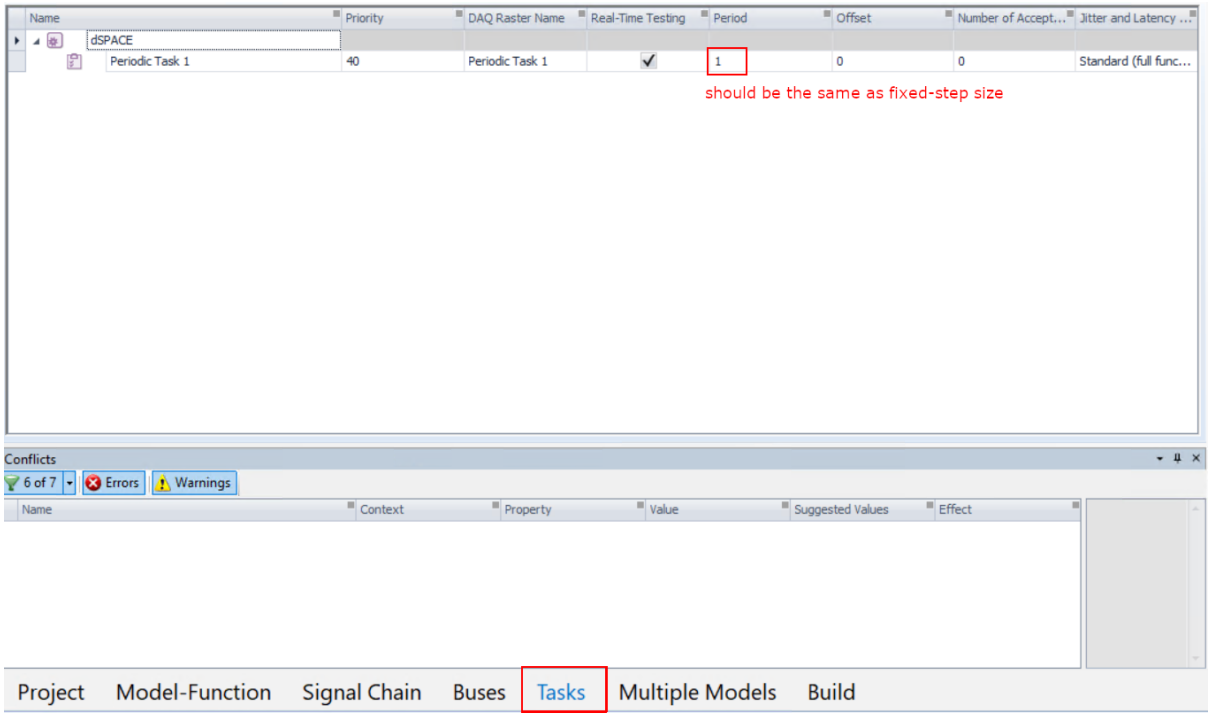


Figure 14.45: Set period of Periodic Task.

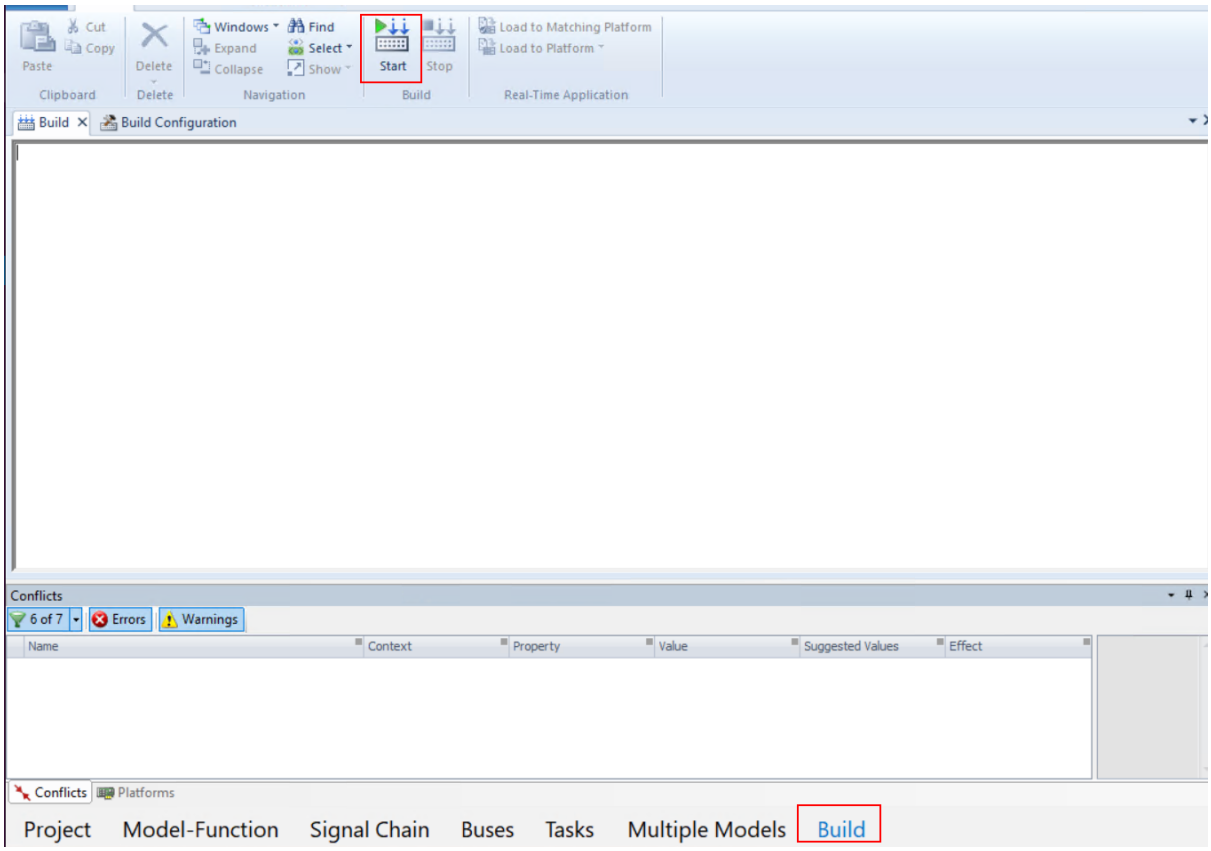


Figure 14.46: Build the project.

14.3.2 Solver Execution

The steps to simulate a FORCESPRO controller on a dSPACE platform are detailed below.

1) After code generation with FORCESPRO and building with the ConfigurationDesk, the ConfigurationDesk project will have generated files to use to run your model on the dSPACE platform (see Figure 14.47 and Figure 14.48).

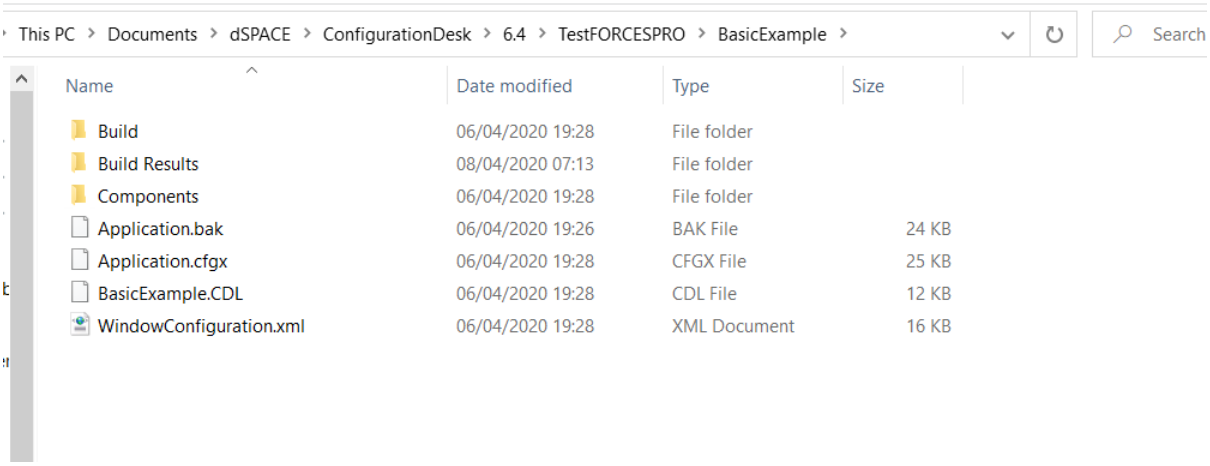


Figure 14.47: The generated files from the ConfigurationDesk building.

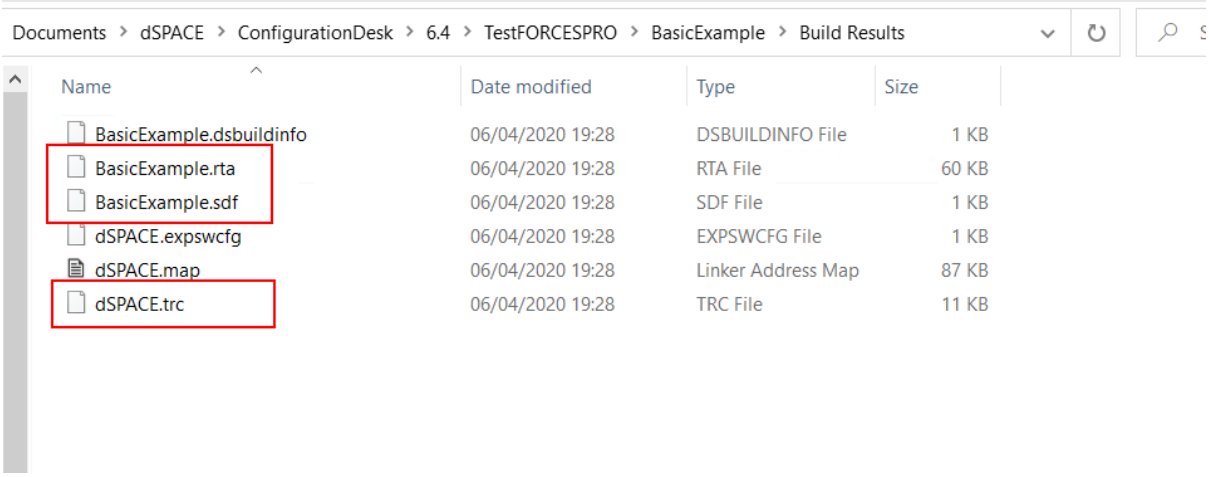


Figure 14.48: The files necessary for the simulation of the FORCESPRO controller.

2) Open dSPACE Control Desk and select create new project and name it (see Figure 14.49).
3) Name the experiment to execute (see Figure 14.50).
4) Select the platform to which you will deploy the generated executable (see Figure 14.51).
5) Import the variable description file `BasicExample.sdf` in order to have access to the model variables and see the results of the execution (see Figure 14.52).
6) On the project layout select the tab `Variables` and on the `BasicExample.sdf` category expand `Model Root`.
7) Select `U OUTPUT` and `X OUTPUT` and Drag & Drop all the input variables together to the Layout. In the opened menu select `Time Plotter` (see Figure 14.53 and Figure 14.54).
8) To see all the plots concurrently right-click on the left of the Y-axis and select `YAxes-view`> `Horizontal stacked` (see Figure 14.55).

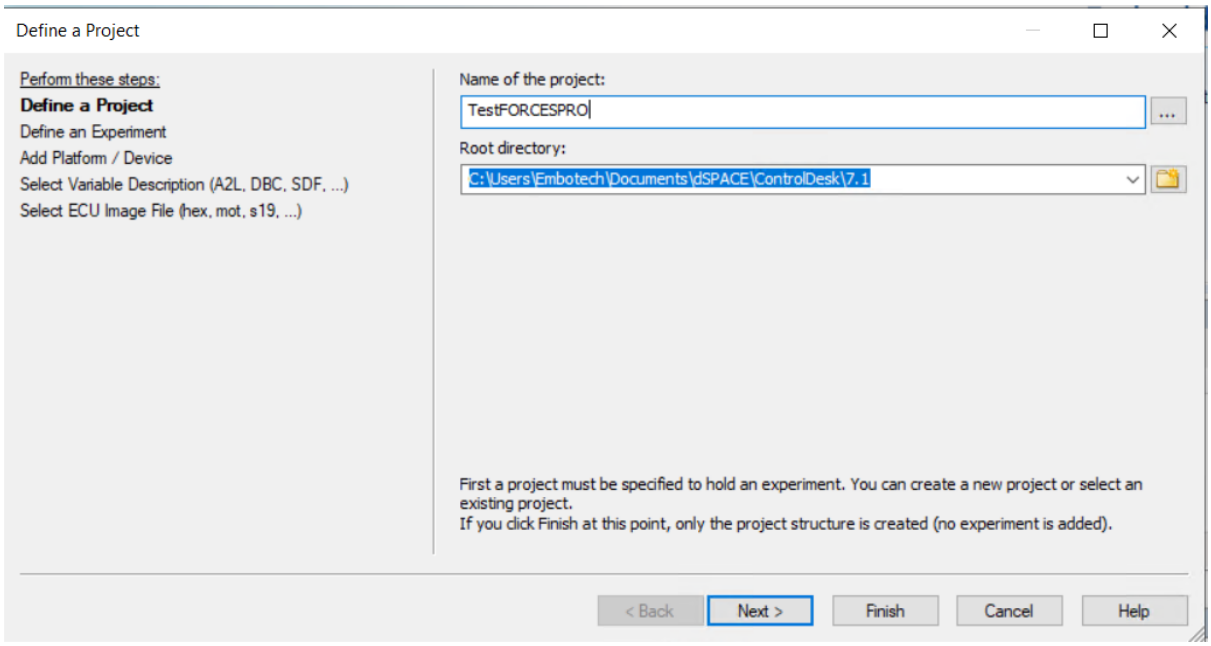


Figure 14.49: Start a new project and name it.

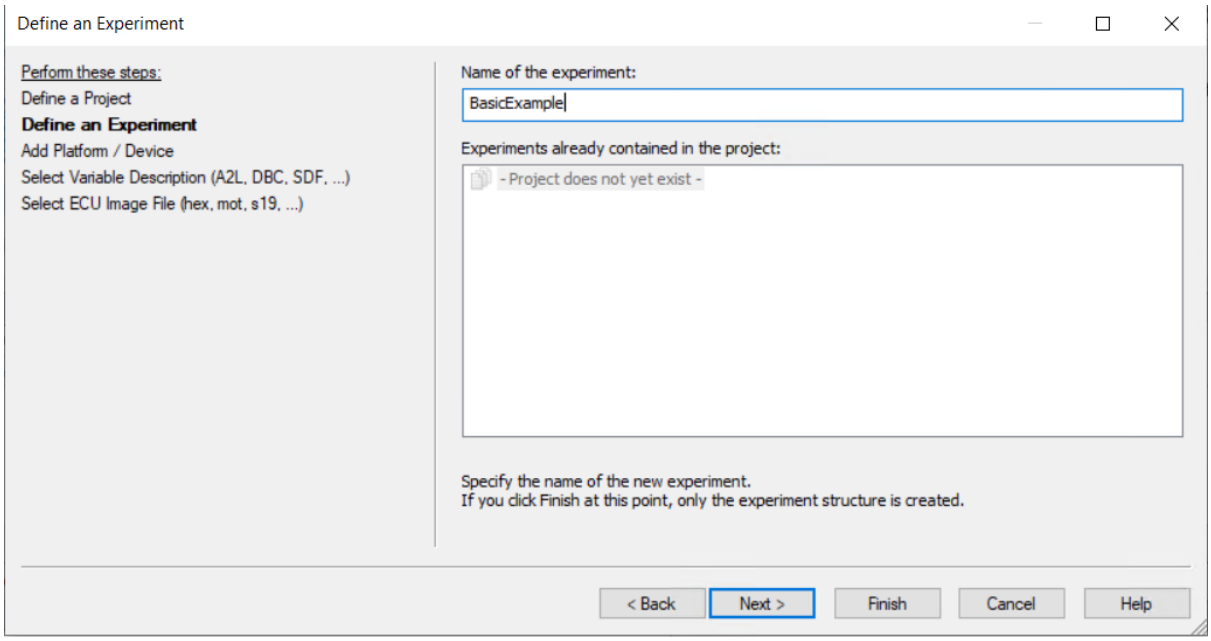


Figure 14.50: Name your experiment.

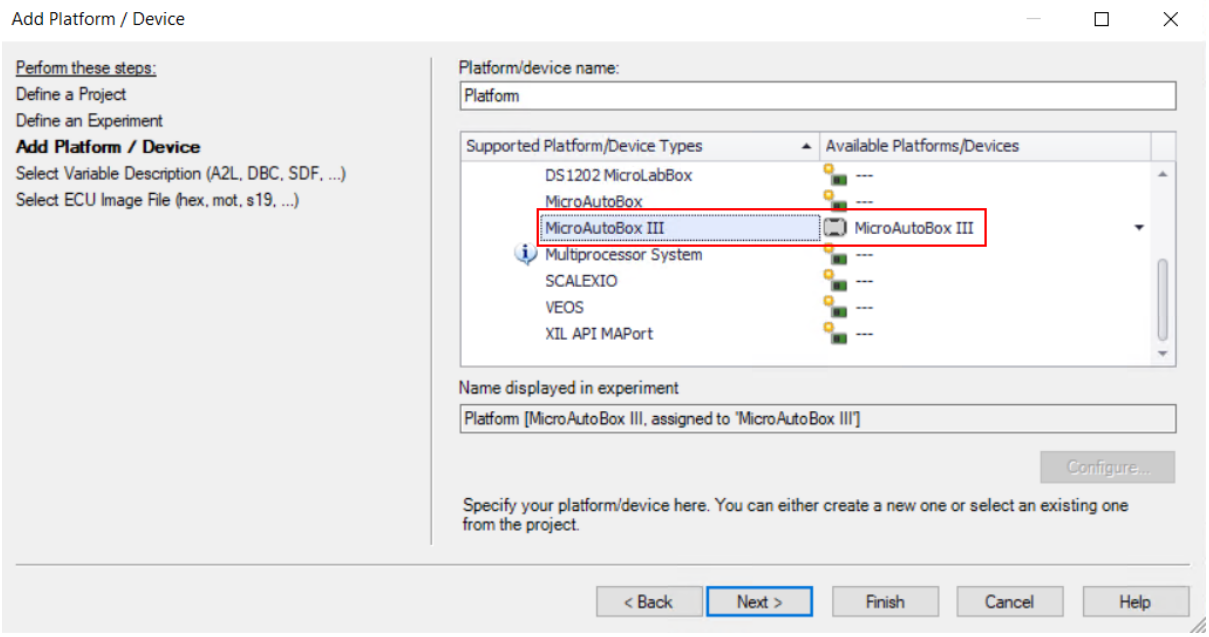


Figure 14.51: Select the dSPACE platform.

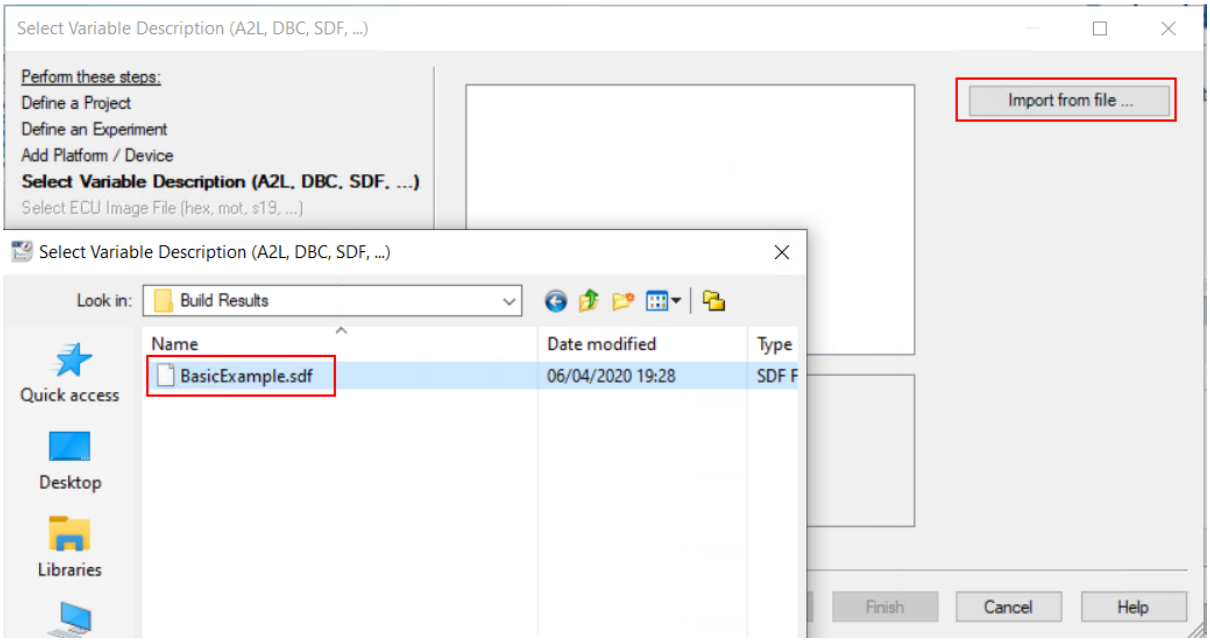


Figure 14.52: Import the variable description file.

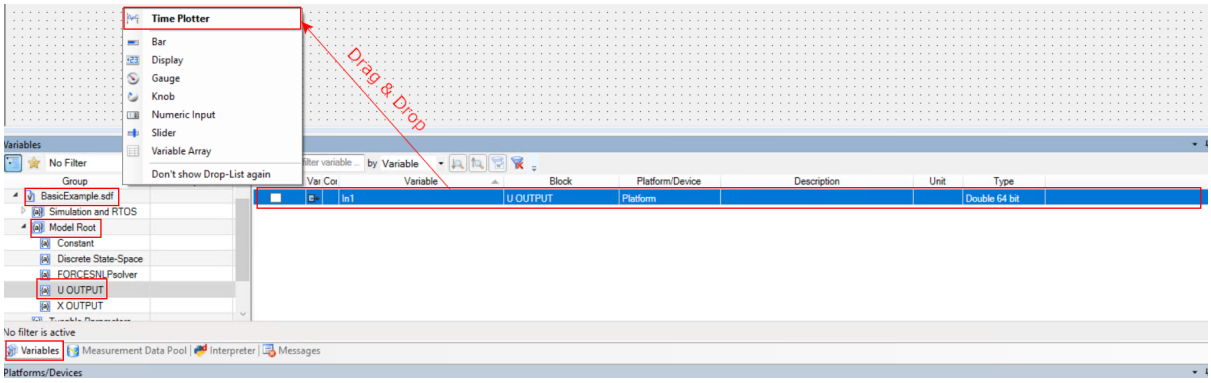


Figure 14.53: Add the inputs of U OUTPUT in a Time Plotter.

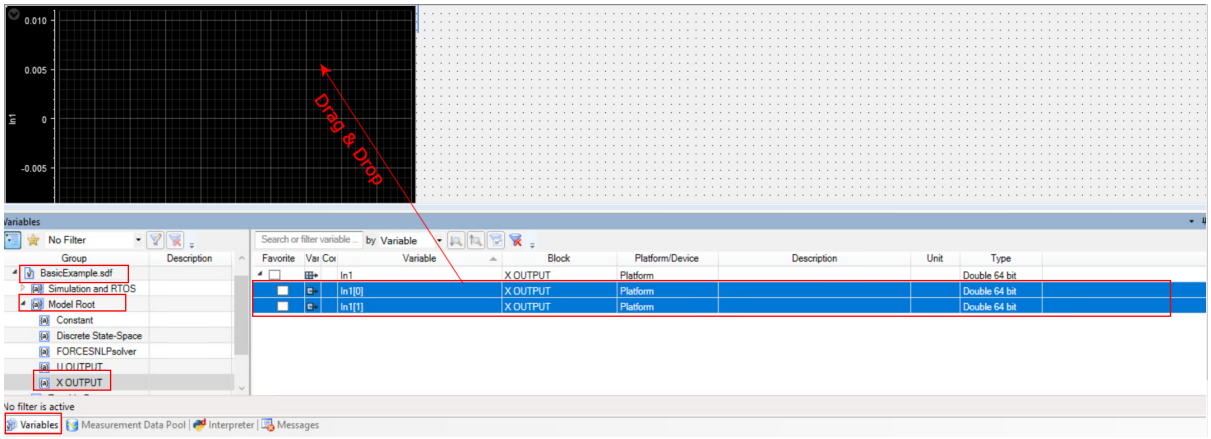


Figure 14.54: Add the inputs of X OUTPUT in the same Time Plotter.

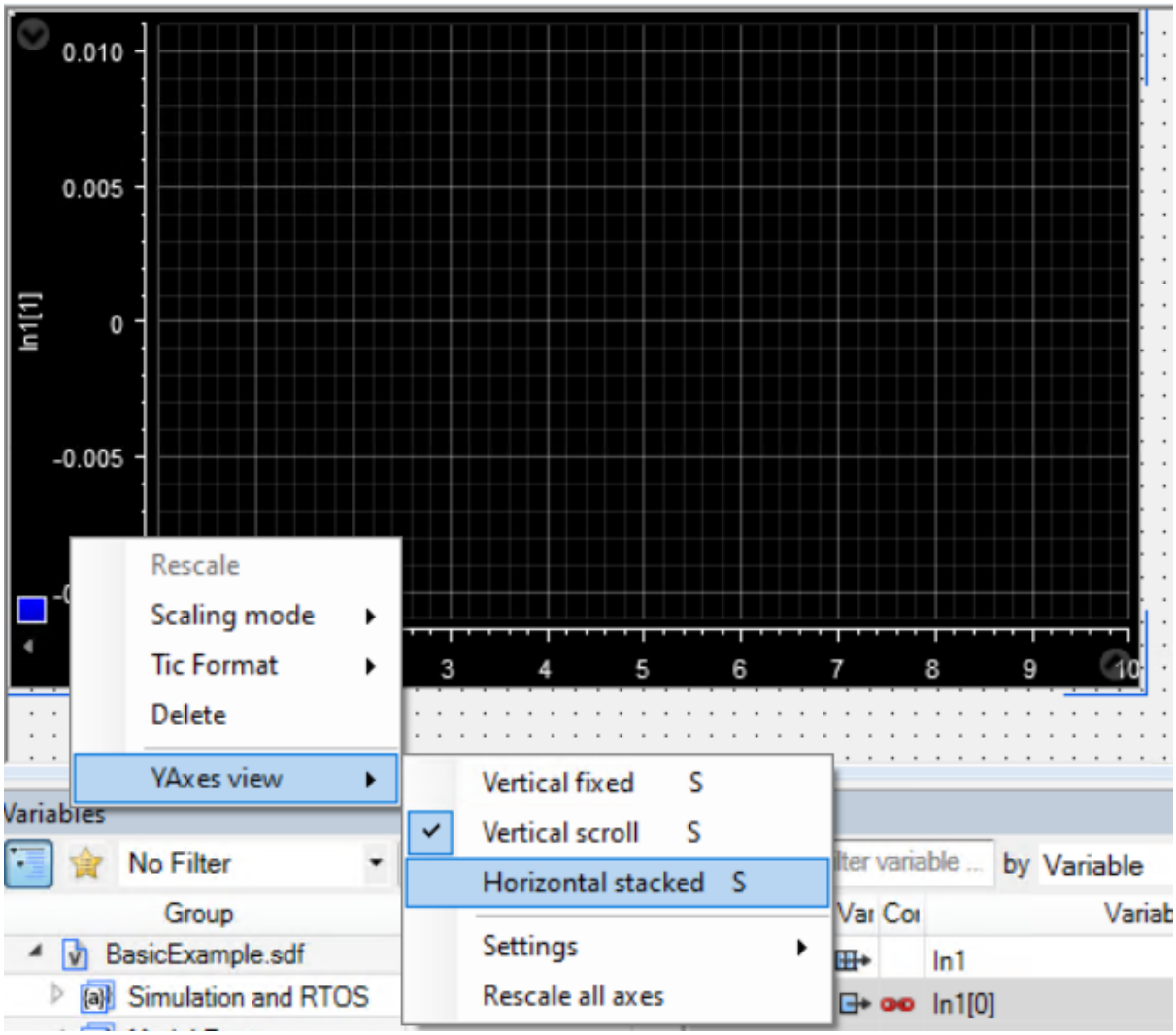


Figure 14.55: Select to show all the signals on the same plot with their own Y-axes

9) Application should have already been loaded from the building of ConfigurationDesk. Otherwise, select the **`Platforms/Devices`** tab. Right-Click on your platform and select **`Real-Time Application`**> **`Load`**. Choose the executable file **`BasicExample.rta`** (see Figure 14.56 and Figure 14.57).

10) Select **`Go Online`** and **`Start Measuring`** to see the results. (see Figure 14.58 and Figure 14.59).

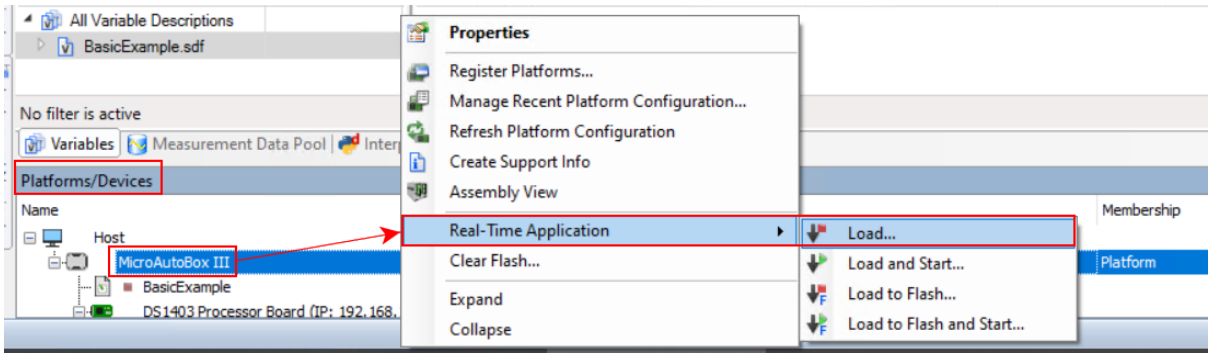


Figure 14.56: Load the application on the dSPACE platform.

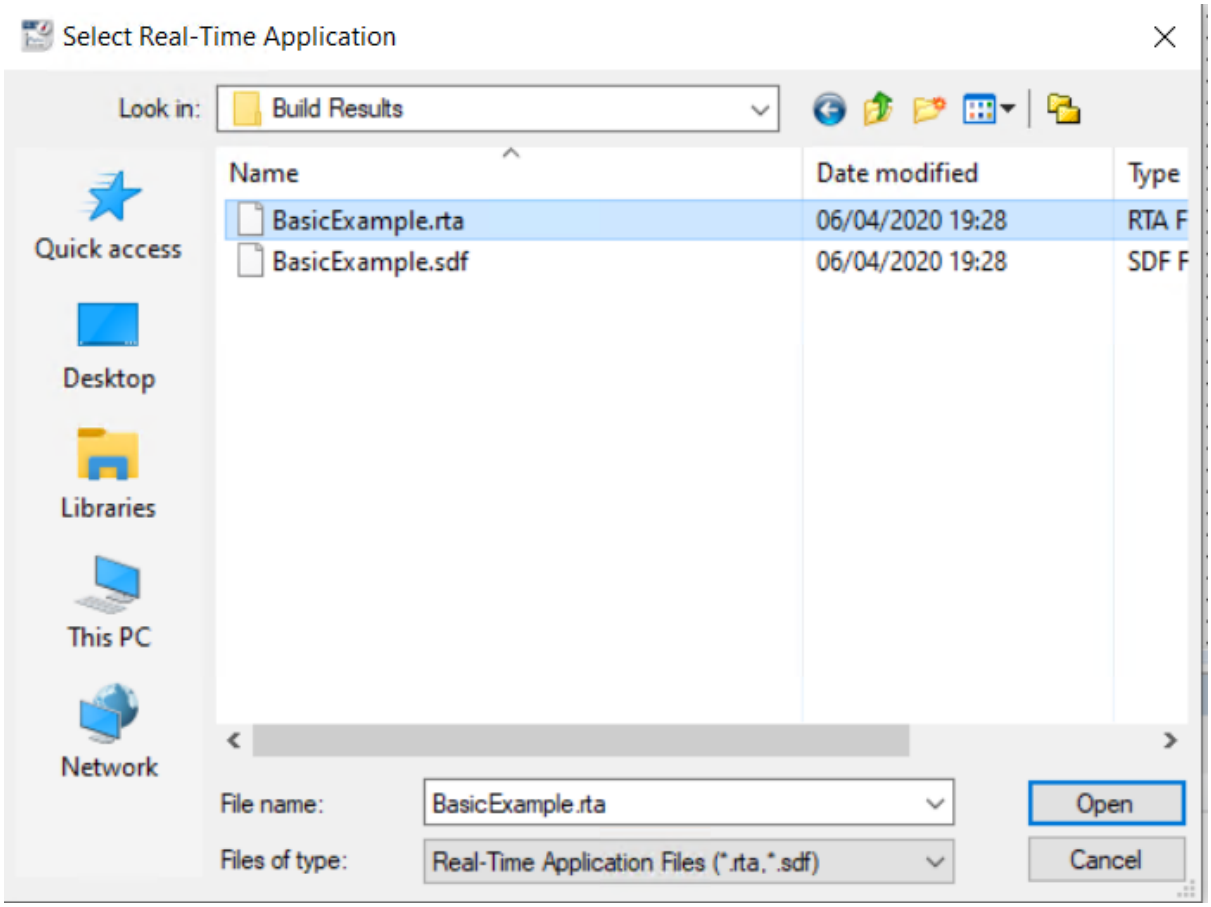


Figure 14.57: Select BasicExample.rta from the ConfigurationDesk project folder.

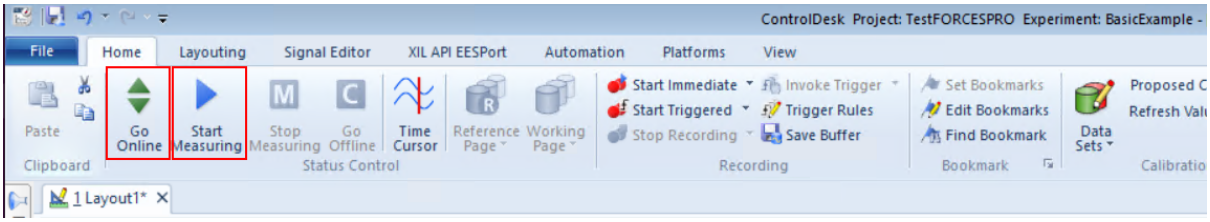


Figure 14.58: Buttons Go Online and Start Measuring to receive execution results.

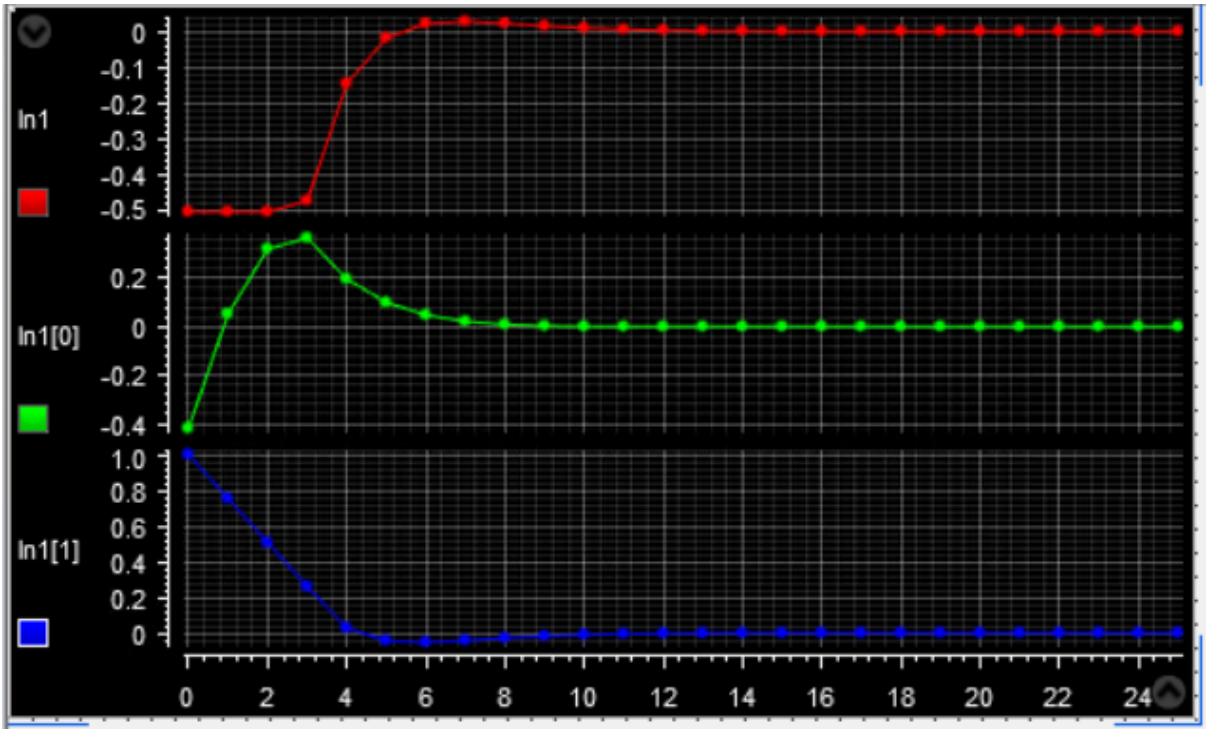


Figure 14.59: Plots and results from experiment on a dSPACE platform.

14.4 Speedgoat

- *High-level interface*
  - *Instructions*
  - *Figures*
- *Y2F interface*
  - *Instructions*
  - *Figures*

Important: When deploying to a target hardware platform, the library included in the **lib_target** directory of the generated solver should be used instead of the library in the **lib** directory.

14.4.1 High-level interface

Instructions

The steps to deploy and simulate a FORCESPRO controller on a Speedgoat platform are detailed below.

1. (Figure 14.60) Set the code generation options:

```
codeoptions.platform = 'Speedgoat-x86'; % to specify the platform
codeoptions.printlevel = 0; % on some platforms printing is not supported
codeoptions.cleanup = 0; % to keep necessary files for target compile
```

and then generate the code for your solver (henceforth referred to as "FORCESNLPsolver", placed in the folder "BasicExample") using the high-level interface.

2. (Figure 14.61) Create a new Simulink model using the blank model template.
3. (Figure 14.62) Populate the Simulink model with the system you want to control.
4. (Figure 14.63) Make sure the `FORCESNLPsolver_simulinkBlock.mexw64` file (created during code generation) is on the MATLAB path.
5. (Figure 14.64) Open the `FORCESNLPsolver_lib.mdl` Simulink model file, contained in the `interface` folder of the `FORCESNLPsolver` folder created during code generation.
6. (Figure 14.65) Copy-paste the FORCESPRO Simulink block into your simulation model and connect its inputs and outputs appropriately.
7. (Figure 14.66) Access the Simulink model's options.
8. (Figure 14.67) In the "Solver" tab, set the options:
   - Simulation start/stop time: Depending on the simulation wanted.
   - Solver type: Discrete or fixed-step.
   - Fixed-step size: Needs to be higher than the execution time of the solver.
9. (Figure 14.68) In the "Code Generation" tab, set the options:
   - System target file: `slrt.tlc`
   - Language: C

- Generate makefile: On
- Template makefile: `slrt_default_tmf`
- Make command: `make_rtw`

10. (Figure 14.69) In the "Code Generation/Custom Code" tab, include the directories:
    - `BasicExample`
    - `BasicExample\FORCESNLPsolver\interface`
    - `BasicExample\FORCESNLPsolver\lib_target`
11. (Figure 14.70) In the "Code Generation/Custom Code" tab, add the source files:
    - `FORCESNLPsolver_simulinkBlock.c`
    - `FORCESNLPsolver_adtool2forces.c`
    - `FORCESNLPsolver_casadi.c`
12. (Figure 14.71) In the "Code Generation/Custom Code" tab, add the library file:
    - `FORCESNLPsolver.lib`
13. (Figure 14.72) Access the FORCESPRO block's parameters.
14. (Figure 14.73) Remove "FORCESNLPsolver" and "FORCESNLPsolver_simulinkBlock" from the S-function module.
15. (Figure 14.74) Compile the code of the Simulink model. This will also automatically load the model to the connected Speedgoat platform.
16. Deployment is complete and simulations can now be run on the Speedgoat platform.
17. Run the simulation on the Speedgoat platform.

    You can find the MATLAB code of this simulation to try it out for yourself in the `examples` folder that comes with your client.

Figures

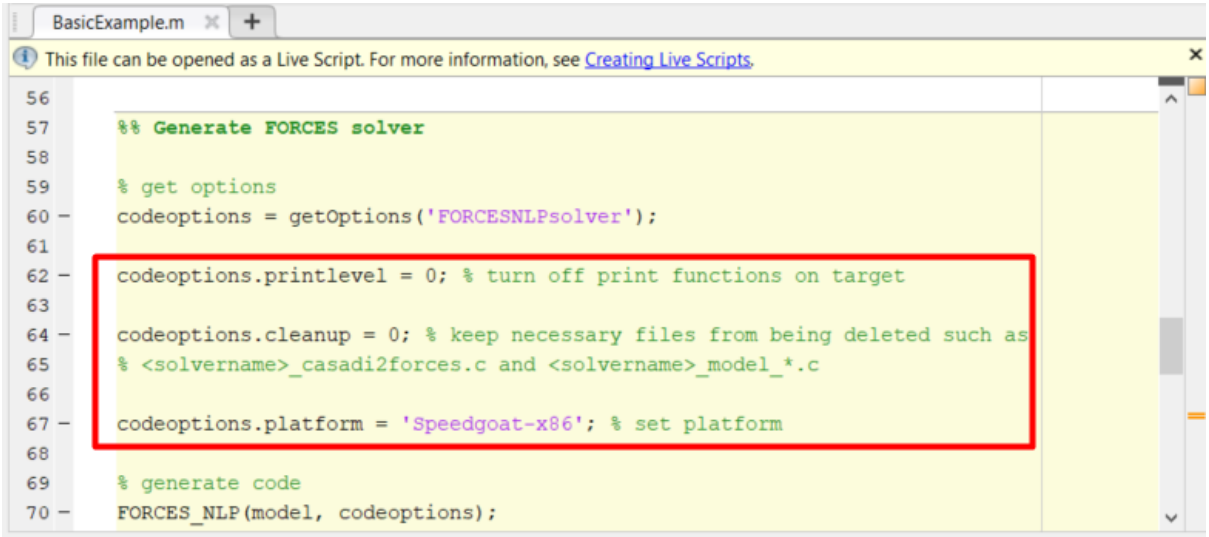


Figure 14.60: Set the appropriate code generation options.

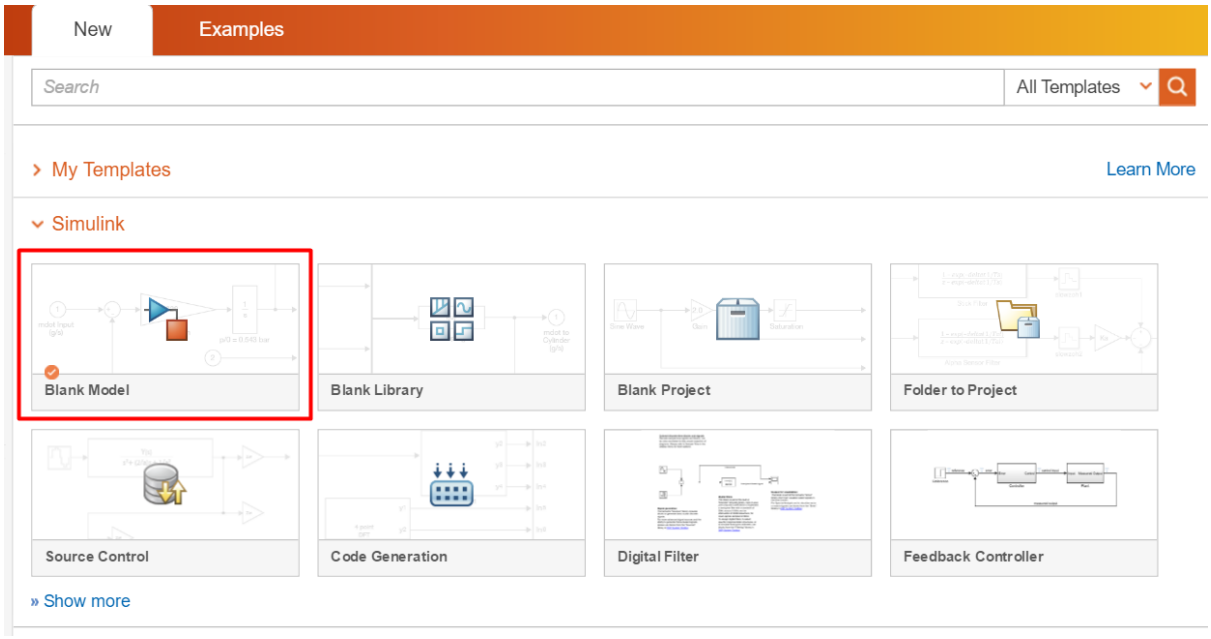


Figure 14.61: Create a Simulink model.

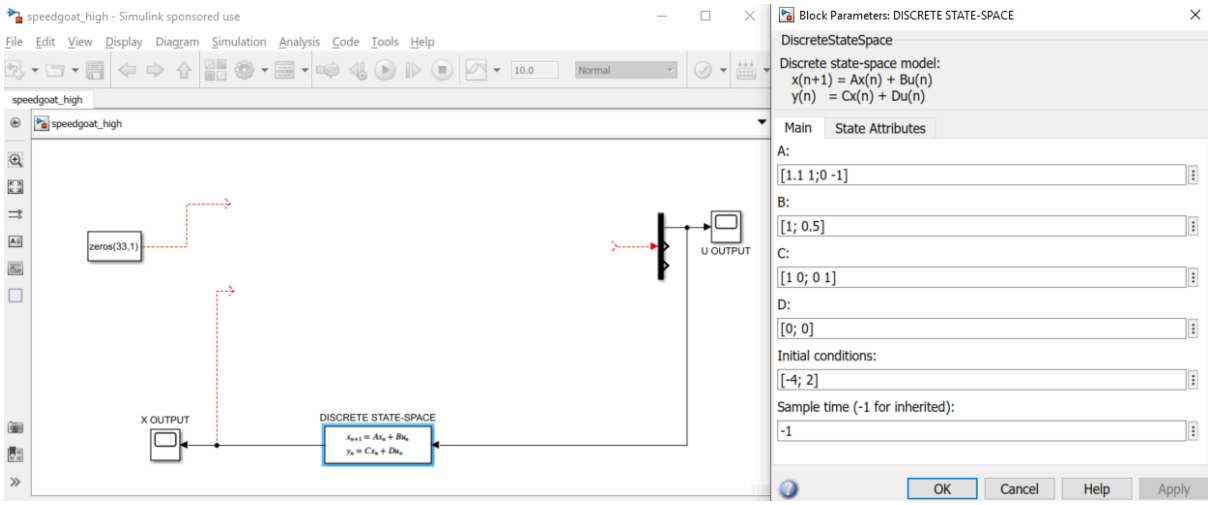


Figure 14.62: Populate the Simulink model.

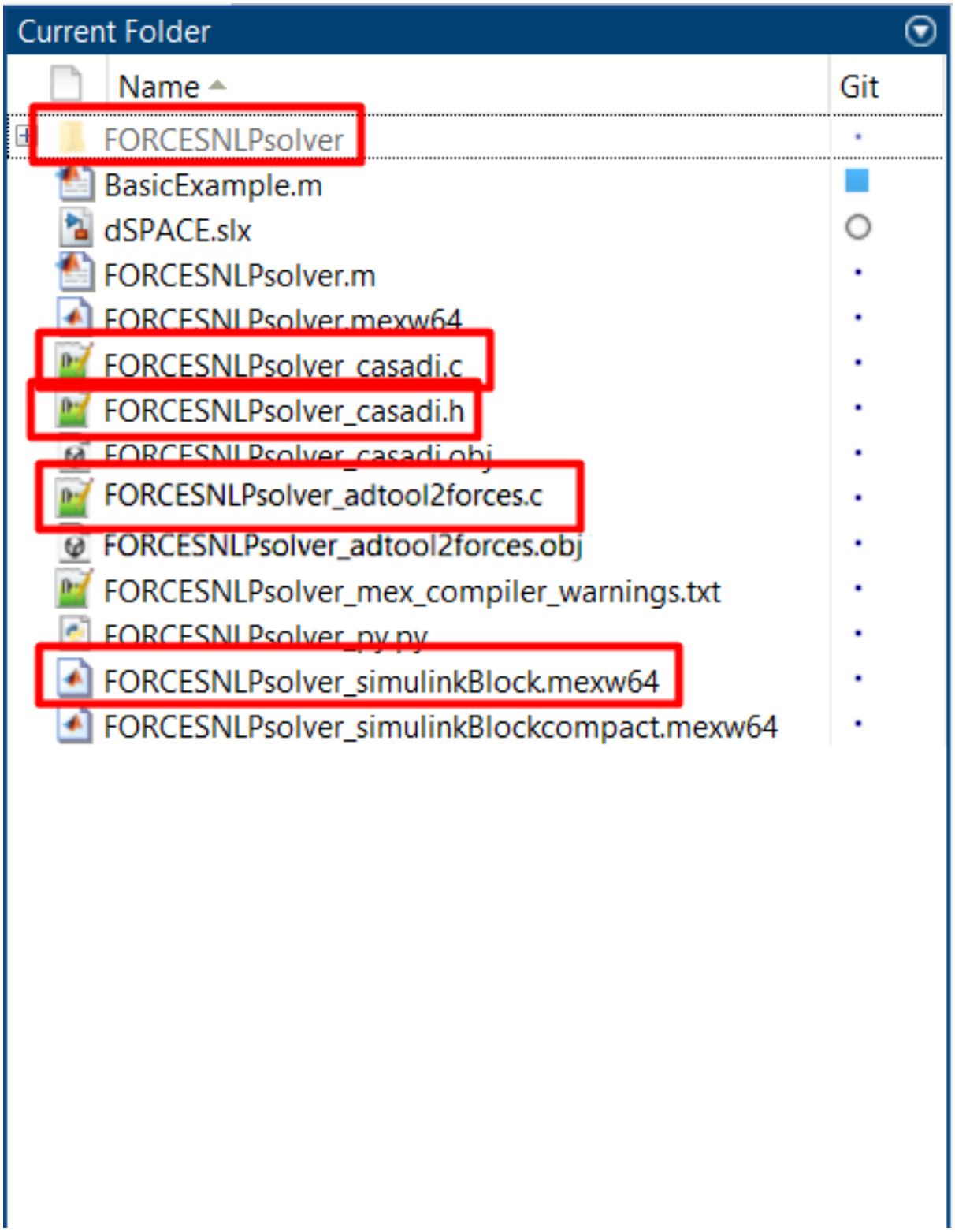


Figure 14.63: Add the folder containing the `.mexw64` solver file to the MATLAB path.

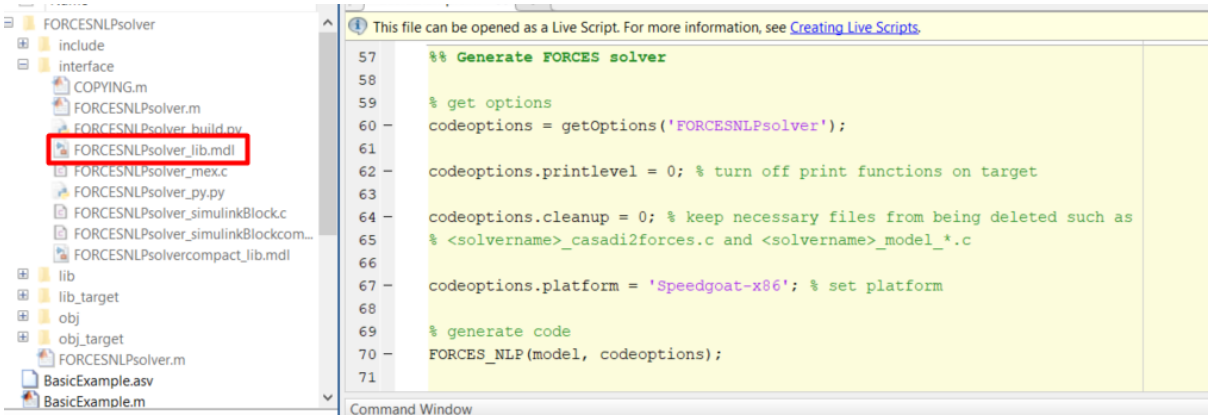


Figure 14.64: Open the generated Simulink solver model.

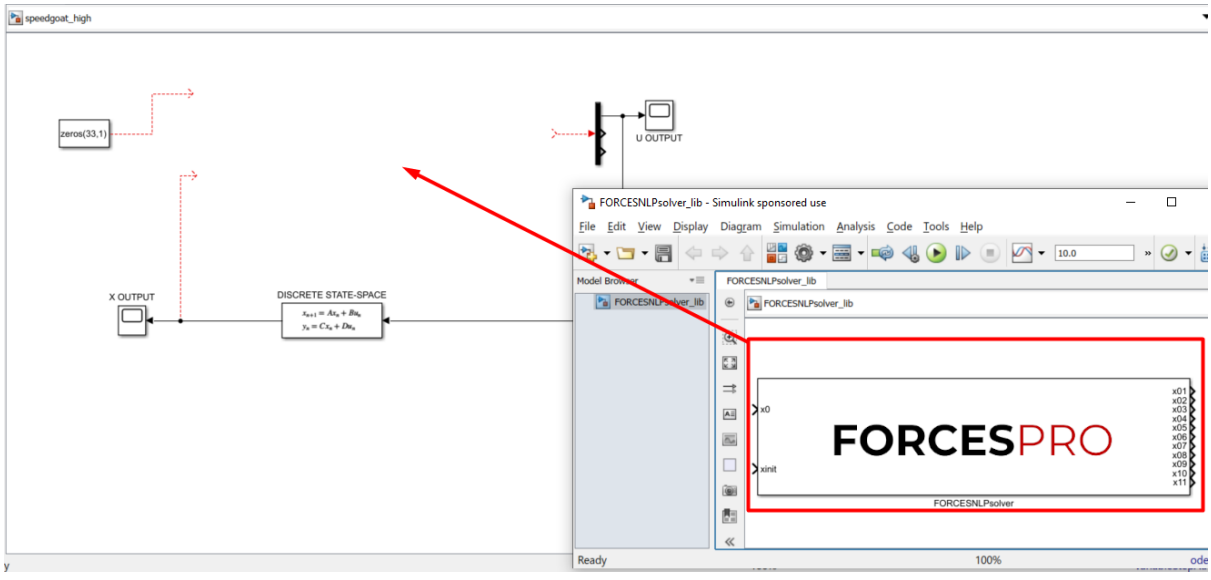


Figure 14.65: Copy-paste and connect the FORCESPRO block.

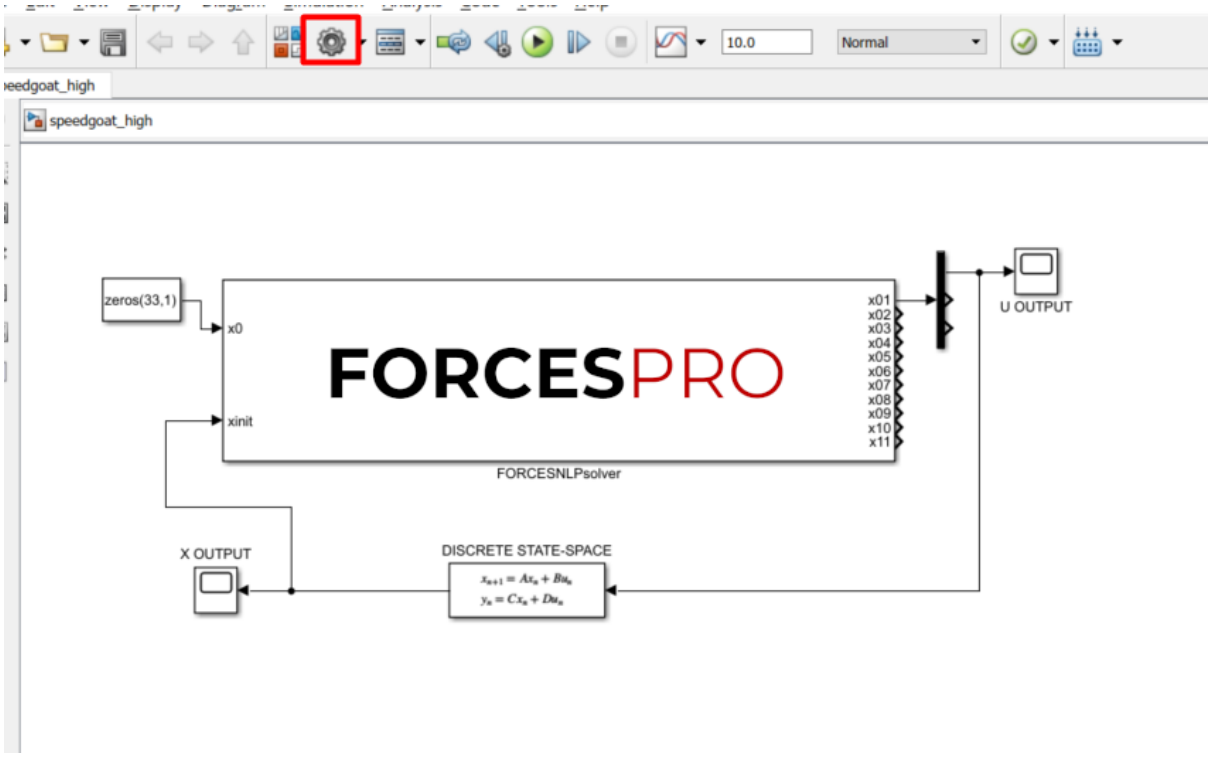


Figure 14.66: Open the Simulink model options.

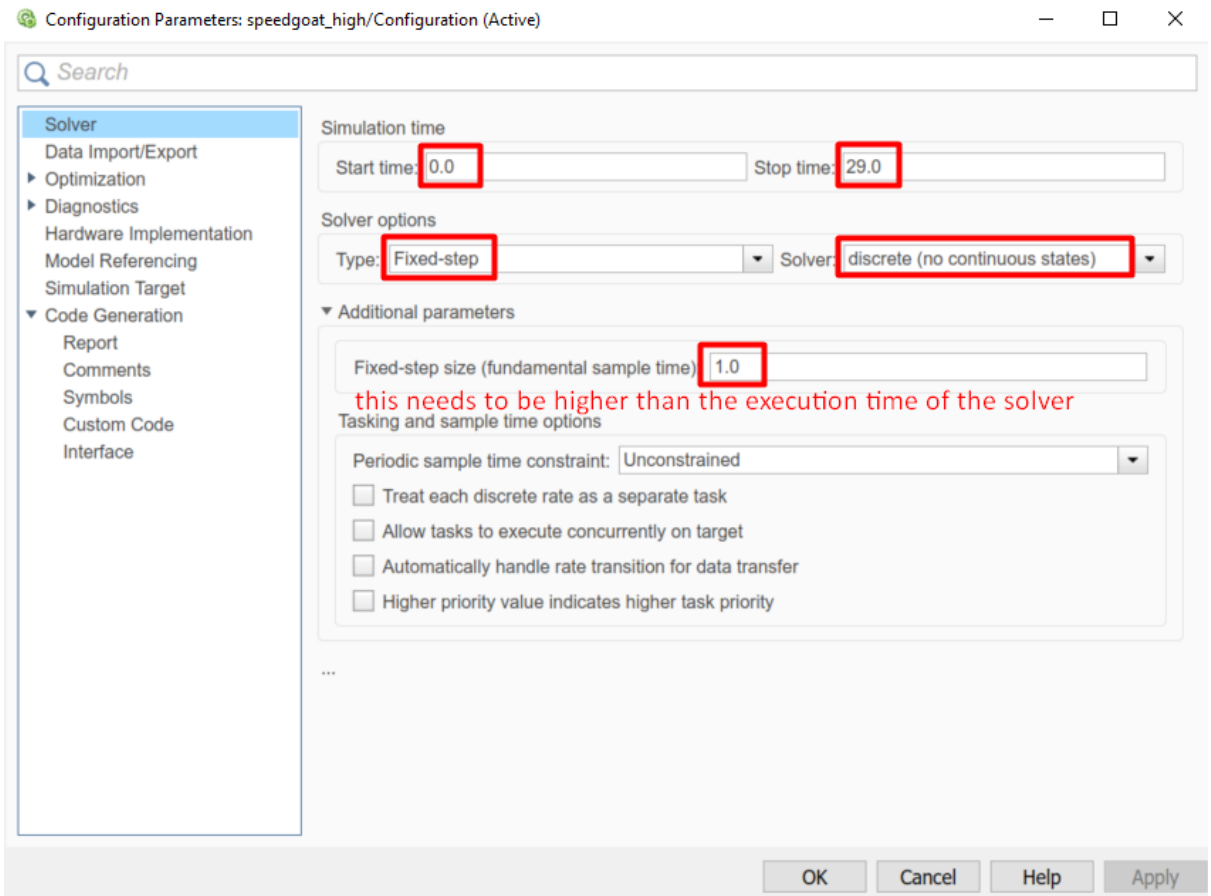


Figure 14.67: Set the Simulink solver options.

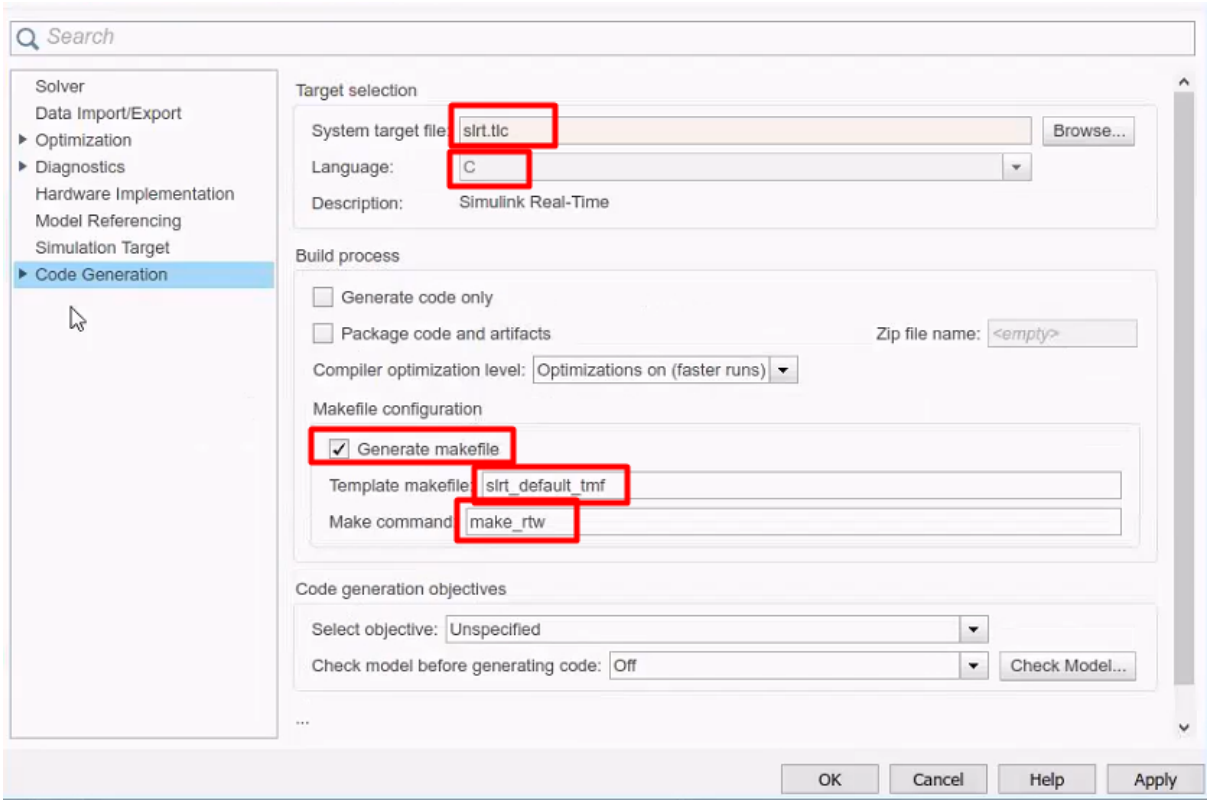


Figure 14.68: Set the Simulink code generation options.

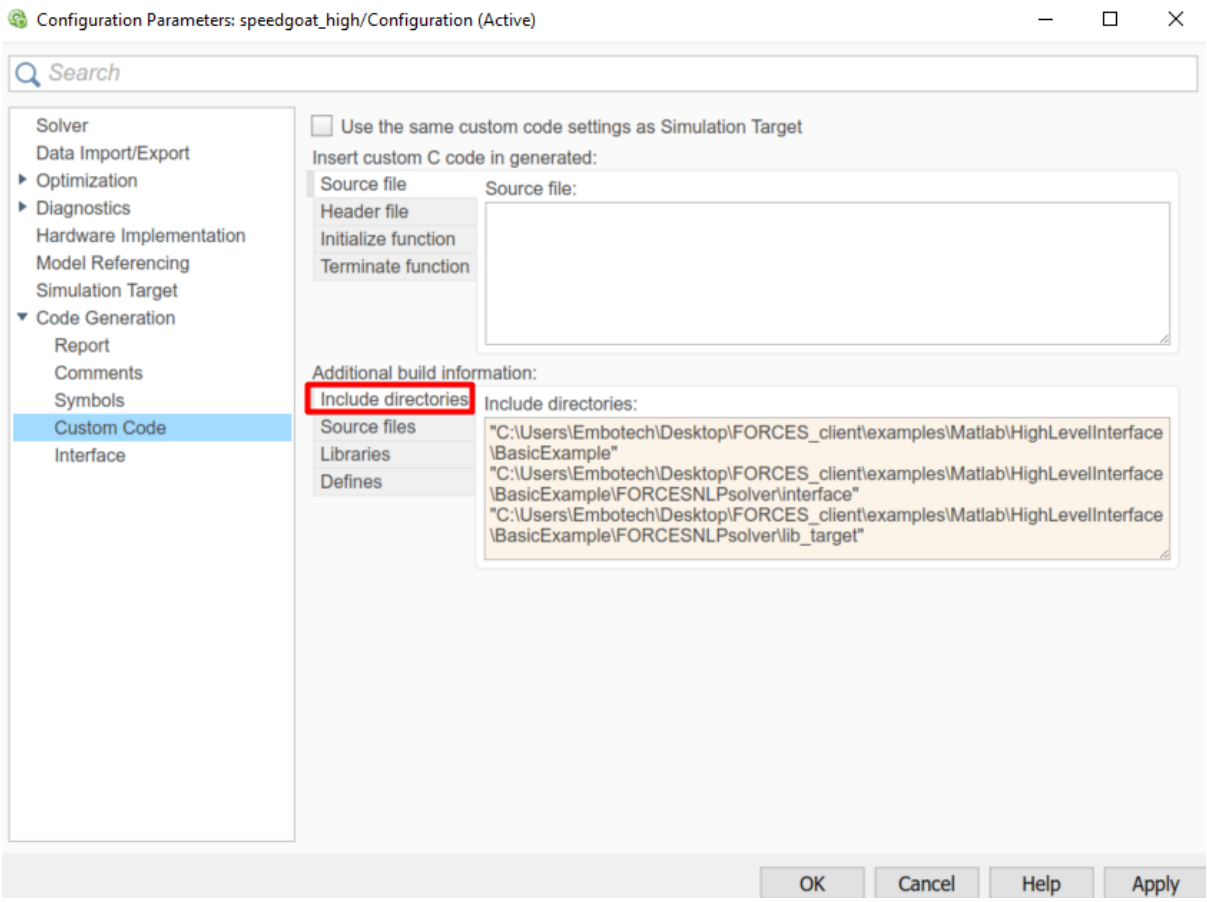


Figure 14.69: Add the directories included for the code generation.

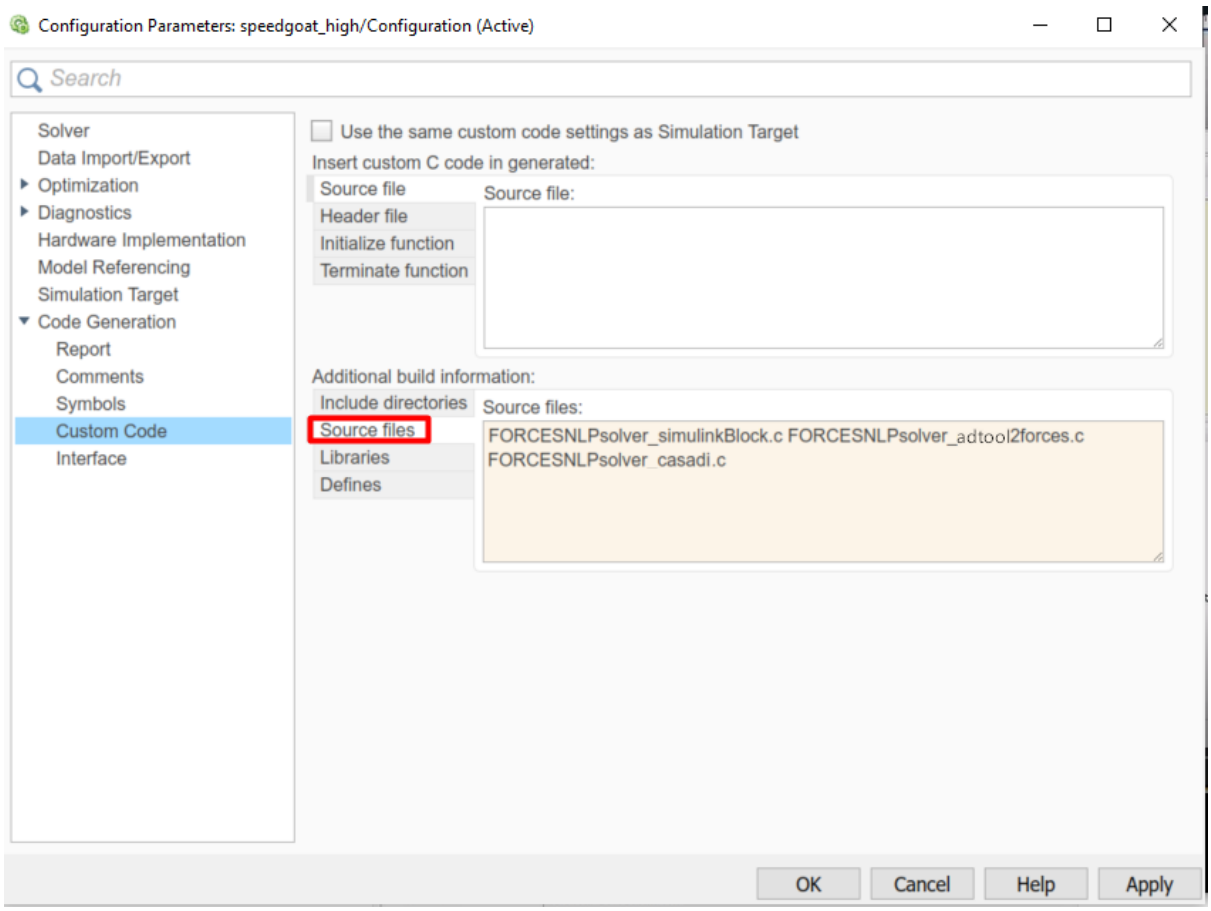


Figure 14.70: Add the source files used for the code generation.

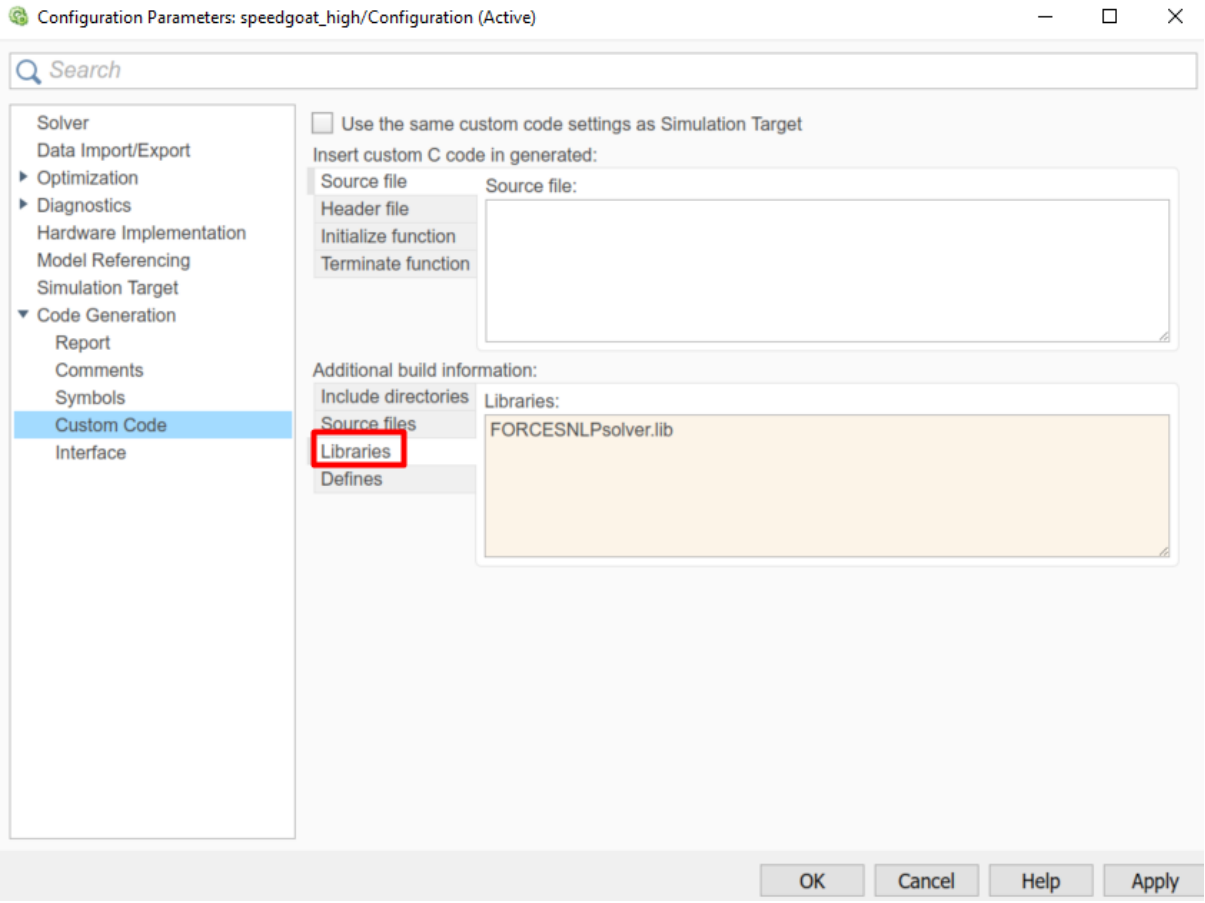


Figure 14.71: Add the libraries used for the code generation.

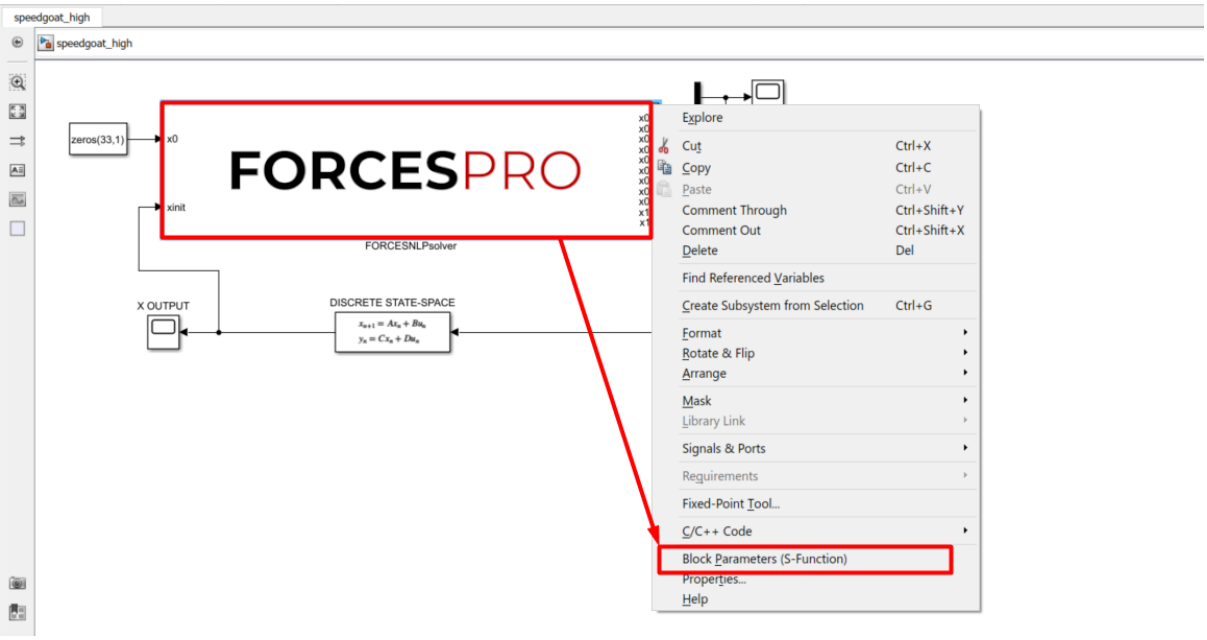


Figure 14.72: Open the FORCESPRO block's parameters.

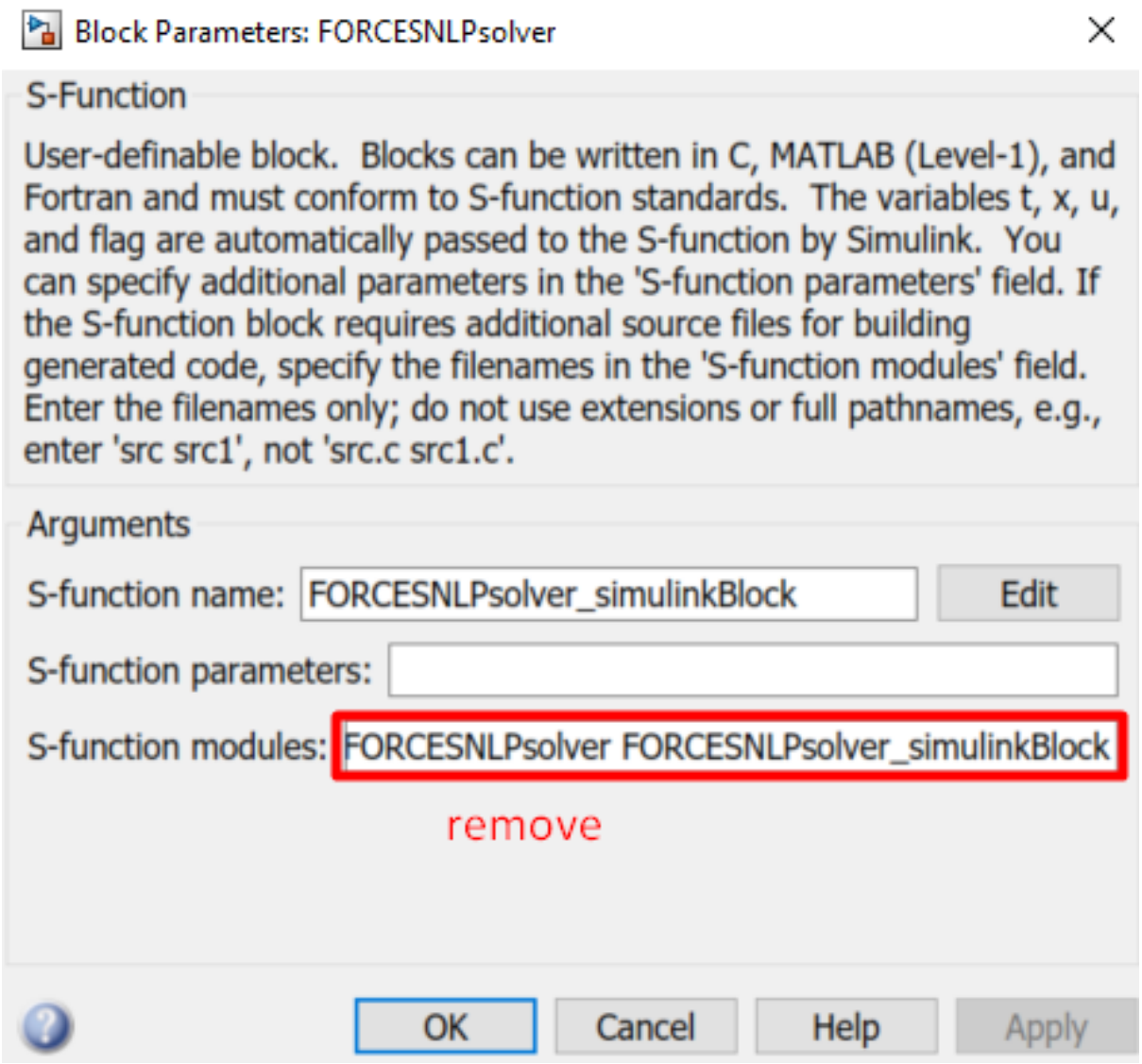


Figure 14.73: Remove the default data from the S-function module.

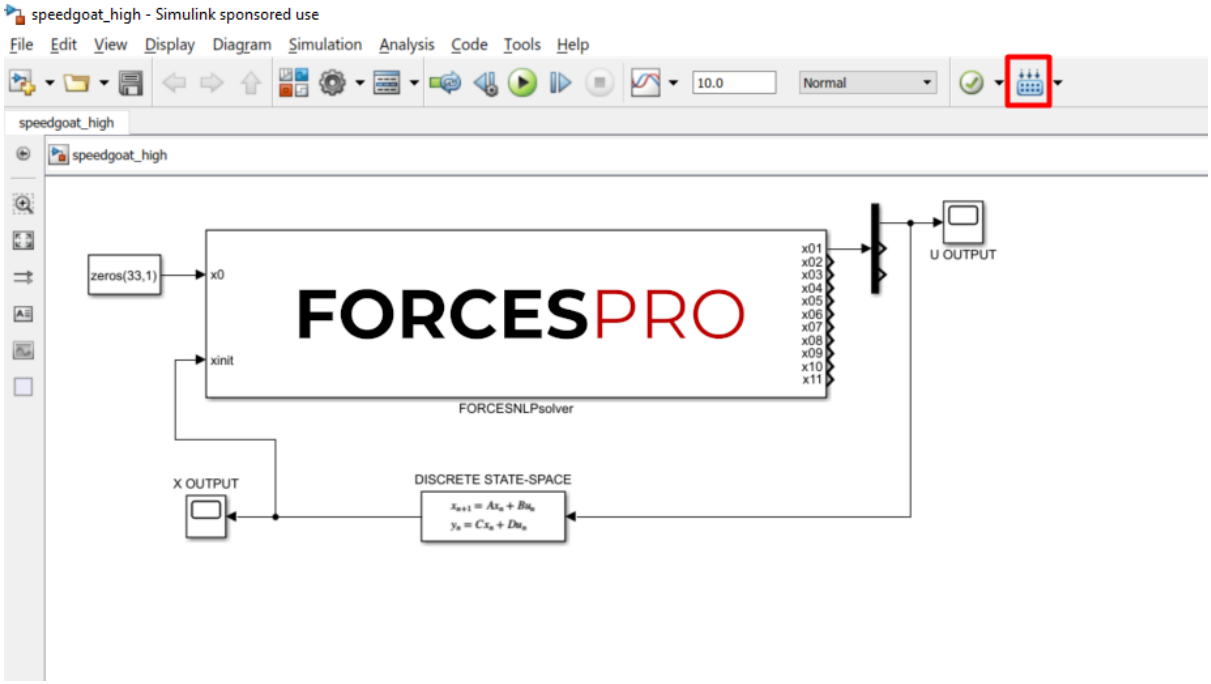


Figure 14.74: Compile the code of the Simulink model.

14.4.2 Y2F interface

Instructions

The steps to deploy and simulate a FORCESPRO controller on a Speedgoat platform are detailed below.

1. (Figure 14.75) Set the code generation options:

```
codeoptions.platform = 'Speedgoat-x86'; % to specify the platform
codeoptions.printlevel = 0; % on some platforms printing is not supported
```

and then generate the code for your solver (henceforth referred to as "simplempc_solver", placed in the folder "Y2F") using the Y2F interface.

2. (Figure 14.76) Create a new Simulink model using the blank model template.
3. (Figure 14.77) Populate the Simulink model with the system you want to control.
4. (Figure 14.78) Make sure the `simplempc_solver_simulinkBlock.mexw64` file (created during code generation) is on the MATLAB path.
5. (Figure 14.79) Copy-paste the FORCESPRO Simulink block, contained in the created `y2f_simulink_lib.slx` Simulink model file, into your simulation model and connect its inputs and outputs appropriately.
6. (Figure 14.80) Access the Simulink model's options.
7. (Figure 14.81) In the "Solver" tab, set the options:
   - Simulation start/stop time: Depending on the simulation wanted.
   - Solver type: Discrete or fixed-step.
   - Fixed-step size: Needs to be higher than the execution time of the solver.
8. (Figure 14.82) In the "Code Generation/RTI general build options" tab, set the options:
   - System target file: `slrt.tlc`
   - Language: C
   - Generate makefile: On
   - Template makefile: `slrt_default_tmf`
   - Make command: `make_rtw`
9. (Figure 14.83) In the "Code Generation/Custom Code" tab, include the directories:
   - `Y2F\simplempc_solver\interface`
   - `Y2F\simplempc_solver\lib_target`
10. (Figure 14.84) In the "Code Generation/Custom Code" tab, add the source files:
    - `simplempc_solver_simulinkBlock.c`
    - `simplempc_solver.c`
11. (Figure 14.85) In the "Code Generation/Custom Code" tab, add the library files:
    - `i_simplempc_solver_1.lib`
12. (Figure 14.86) Compile the code of the Simulink model. This will also automatically load the model to the connected Speedgoat platform.
13. Deployment is complete and simulations can now be run on the Speedgoat platform.

14. Run the simulation on the Speedgoat platform.

    You can find the MATLAB code of this simulation to try it out for yourself in the `examples` folder that comes with your client.

Figures

```
%% Create controller object (generates code)
% for a complete list of codeoptions, see
% https://www.embotech.com/FORCES-Pro/User-Manual/Low-level-Interface/Solver-Options
codeoptions = getOptions('simpleMPC_solver'); % give solver a name

codeoptions.printlevel = 0; % turn off print functions on target

codeoptions.platform = 'Speedgoat-x86'; % set platform

controller = optimizerFORCES(const, cost, codeoptions, X(:,1), U(:,1), {'xinit'}, {'
%% Simulate
```

Figure 14.75: Set the appropriate code generation options.

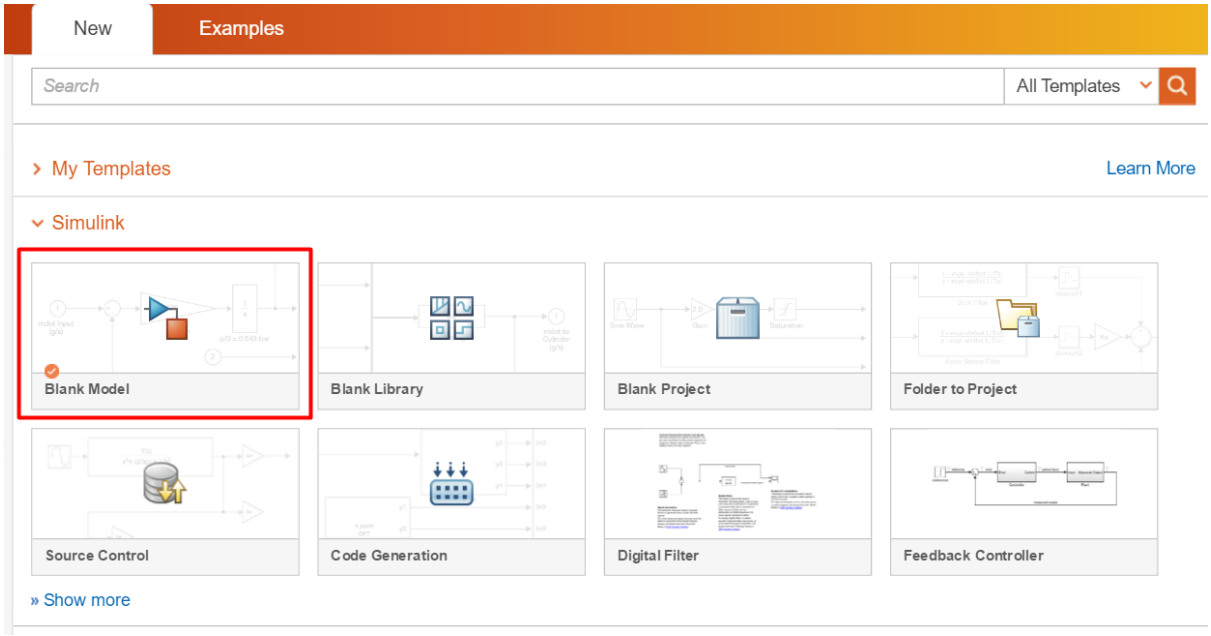


Figure 14.76: Create a Simulink model.

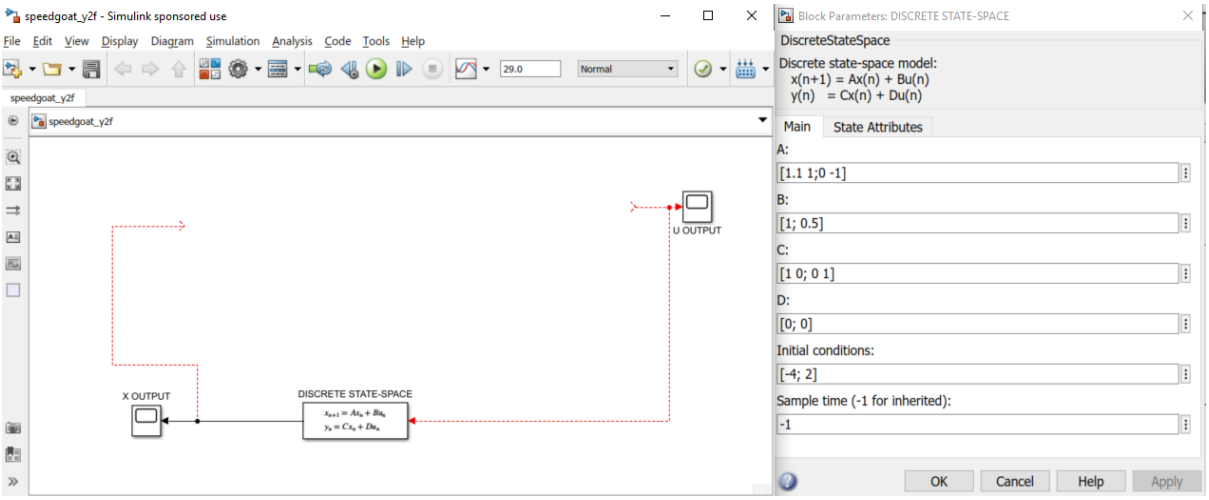


Figure 14.77: Populate the Simulink model.

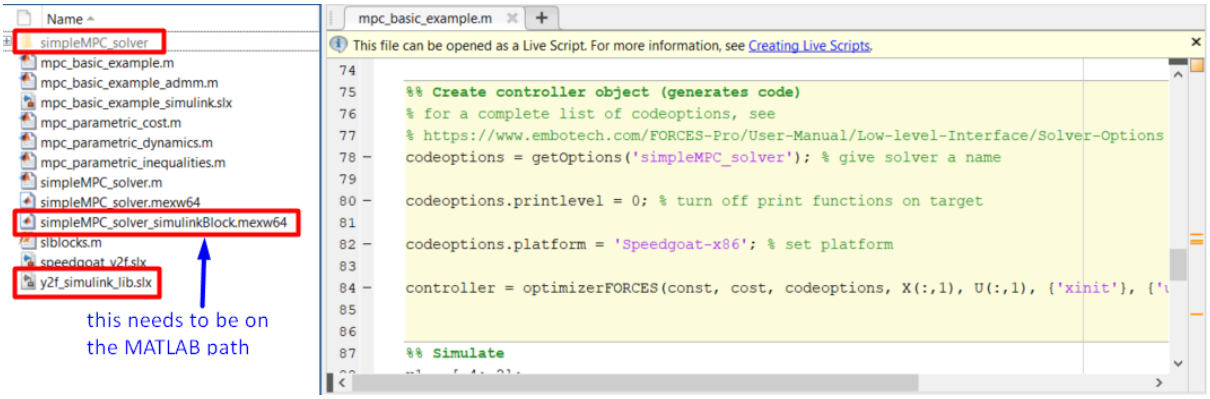


Figure 14.78: Add the folder containing the `.mexw64` solver file to the MATLAB path.

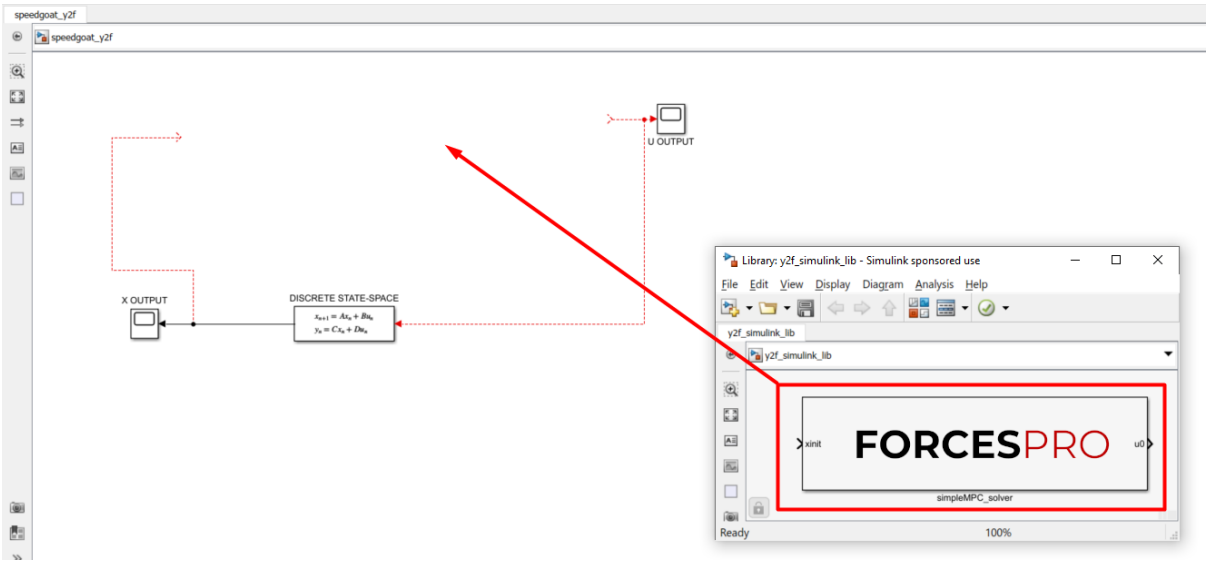


Figure 14.79: Copy-paste and connect the FORCESPRO block.

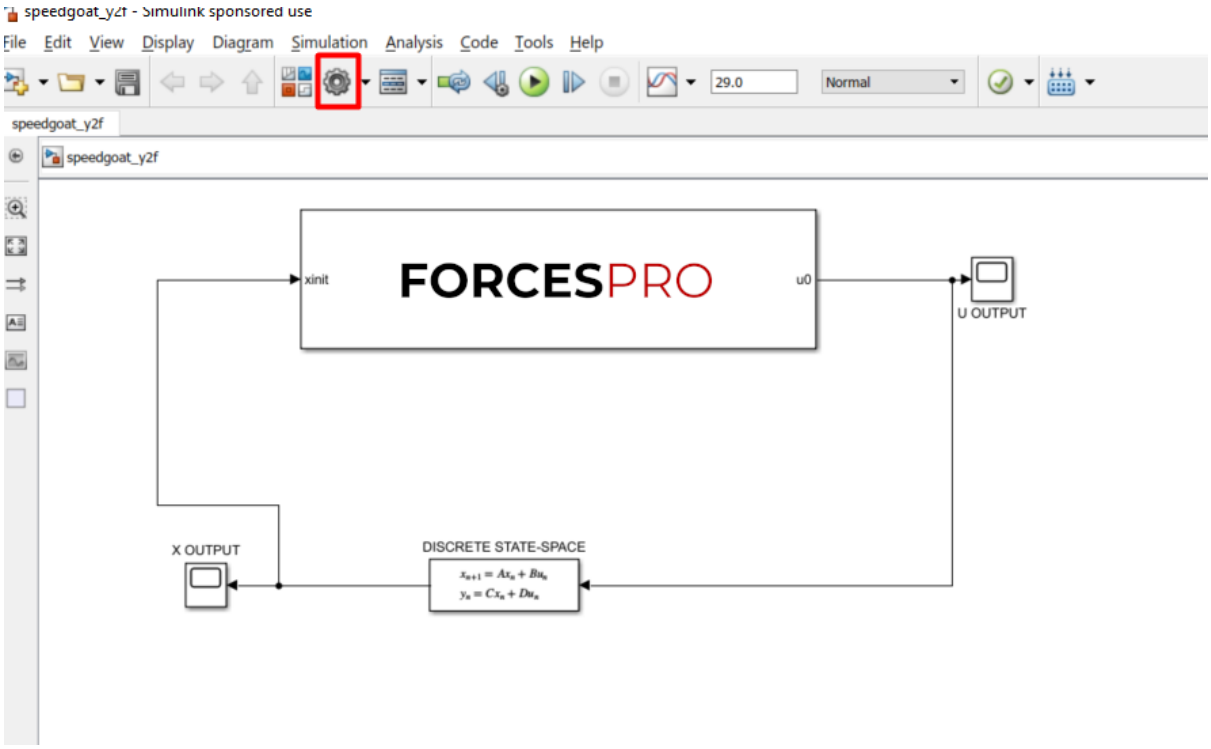


Figure 14.80: Open the Simulink model options.

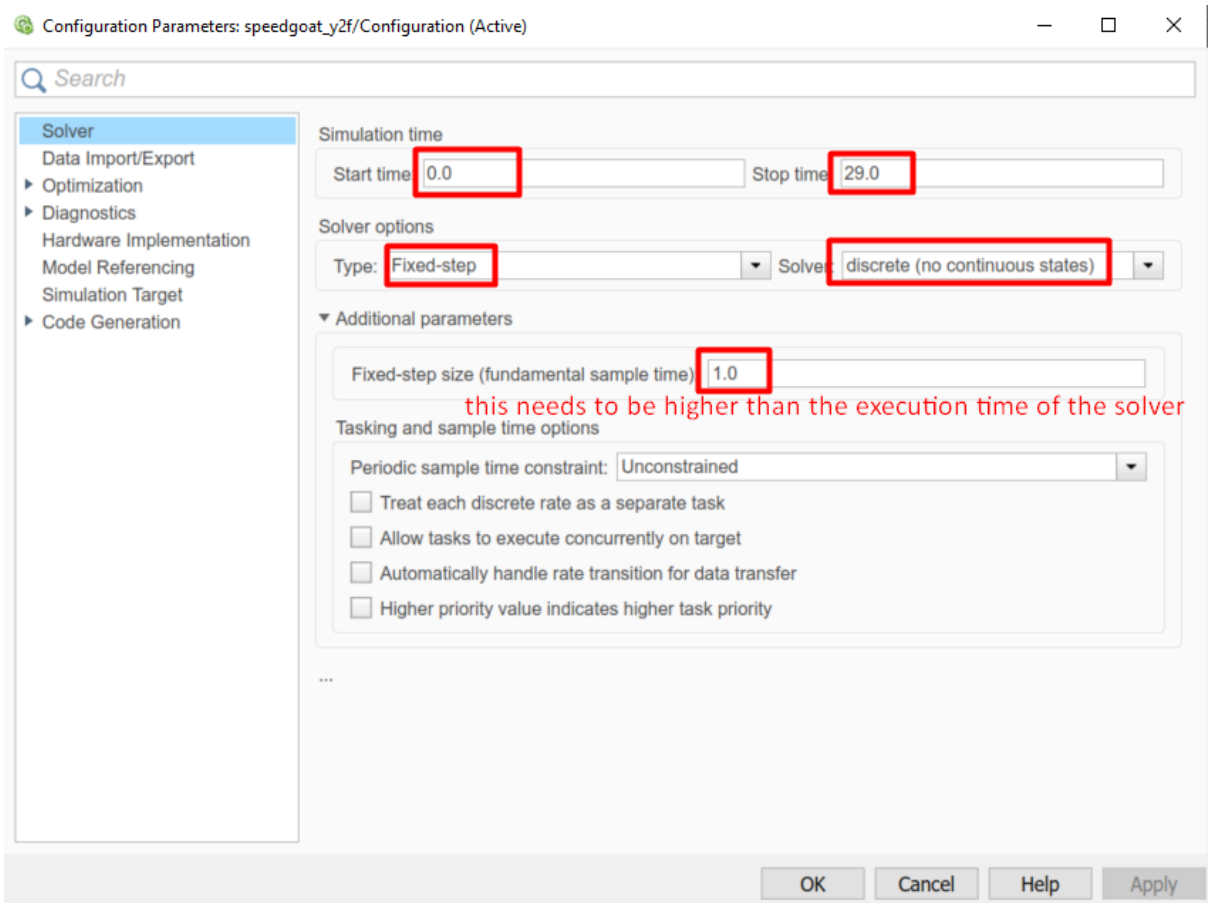


Figure 14.81: Set the Simulink solver options.

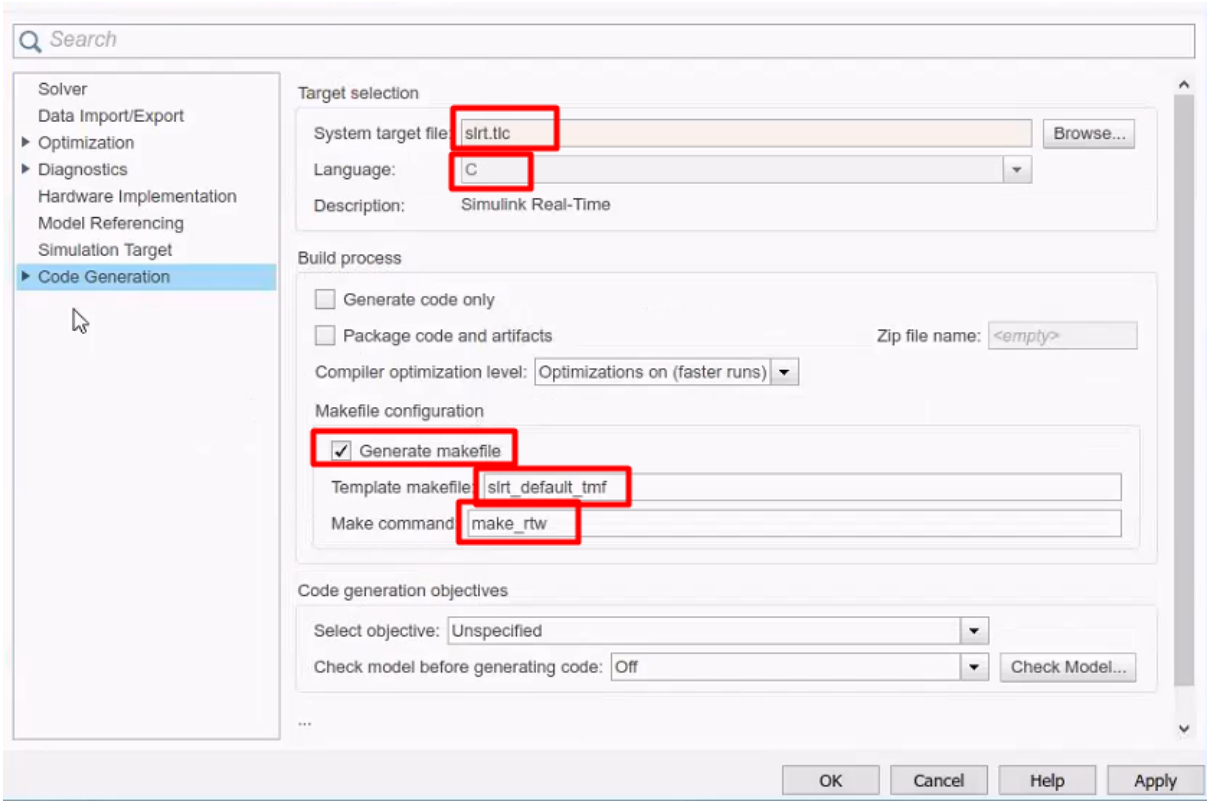


Figure 14.82: Set the Simulink code generation options.

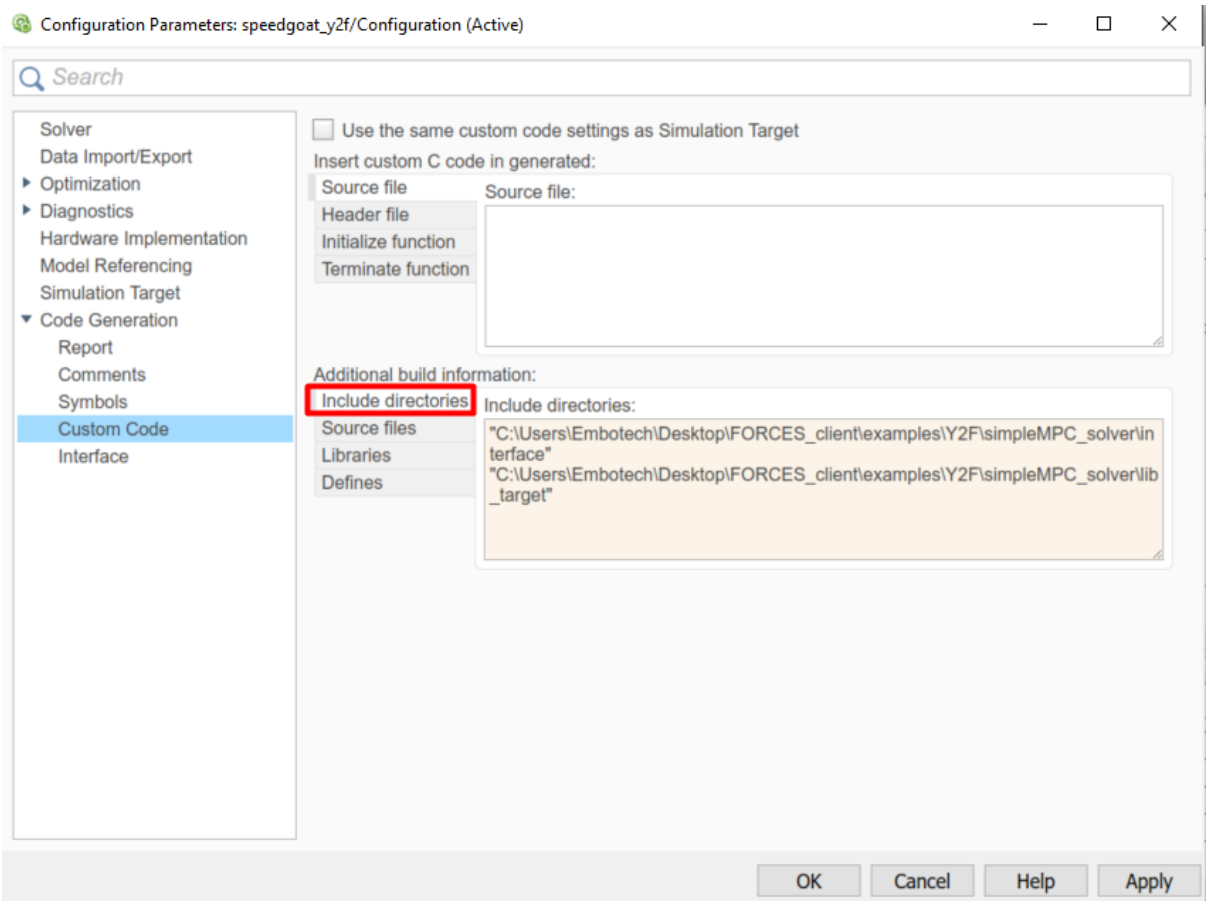


Figure 14.83: Add the directories included for the code generation.

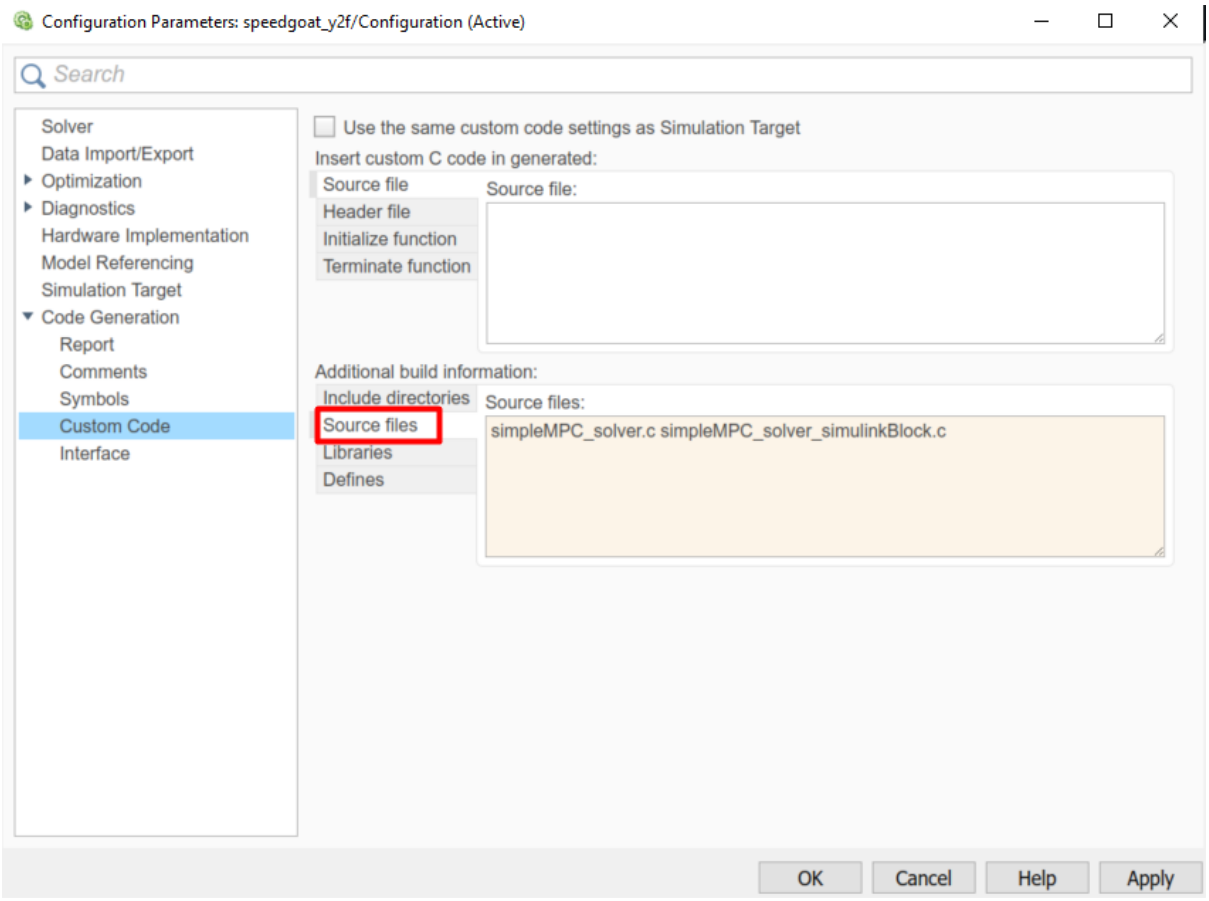


Figure 14.84: Add the source files used for the code generation.

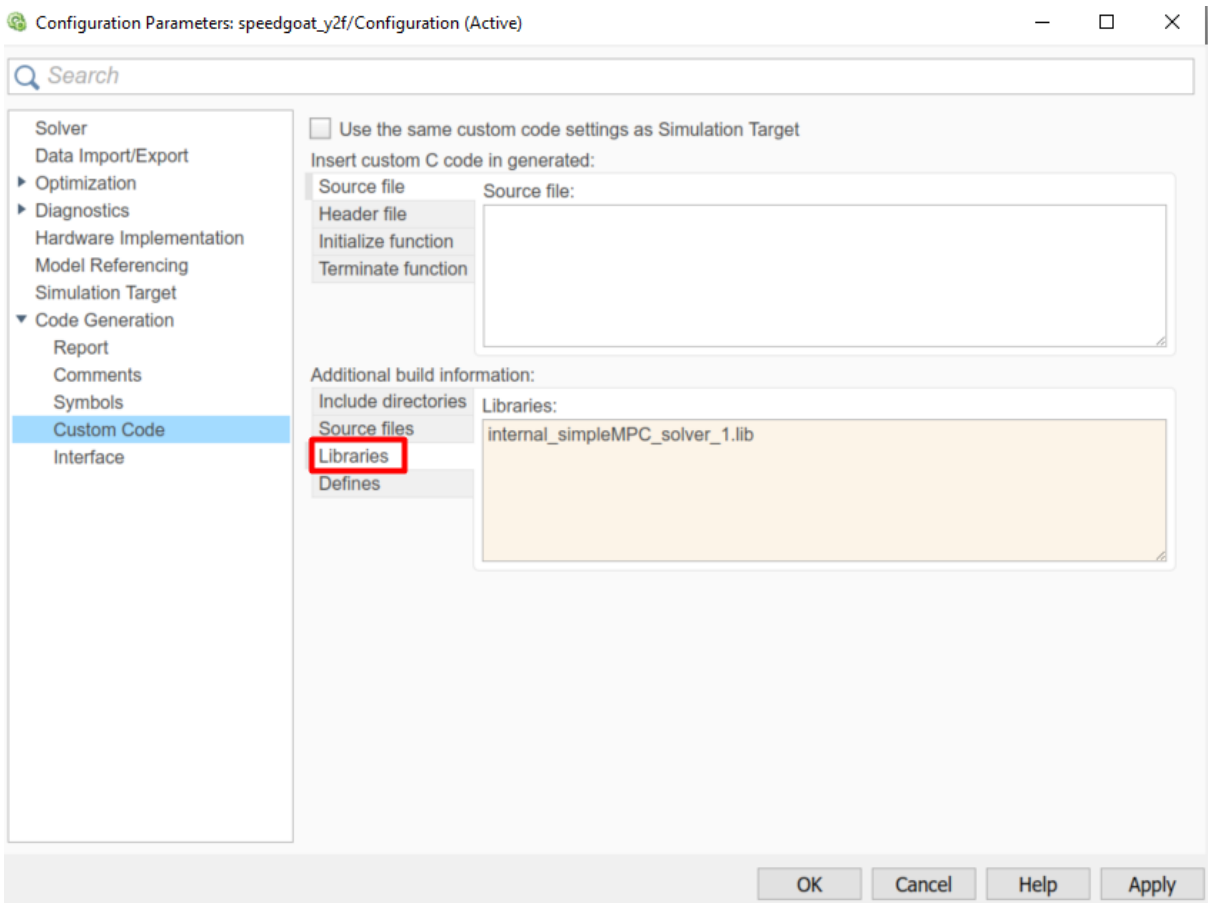


Figure 14.85: Add the libraries used for the code generation.

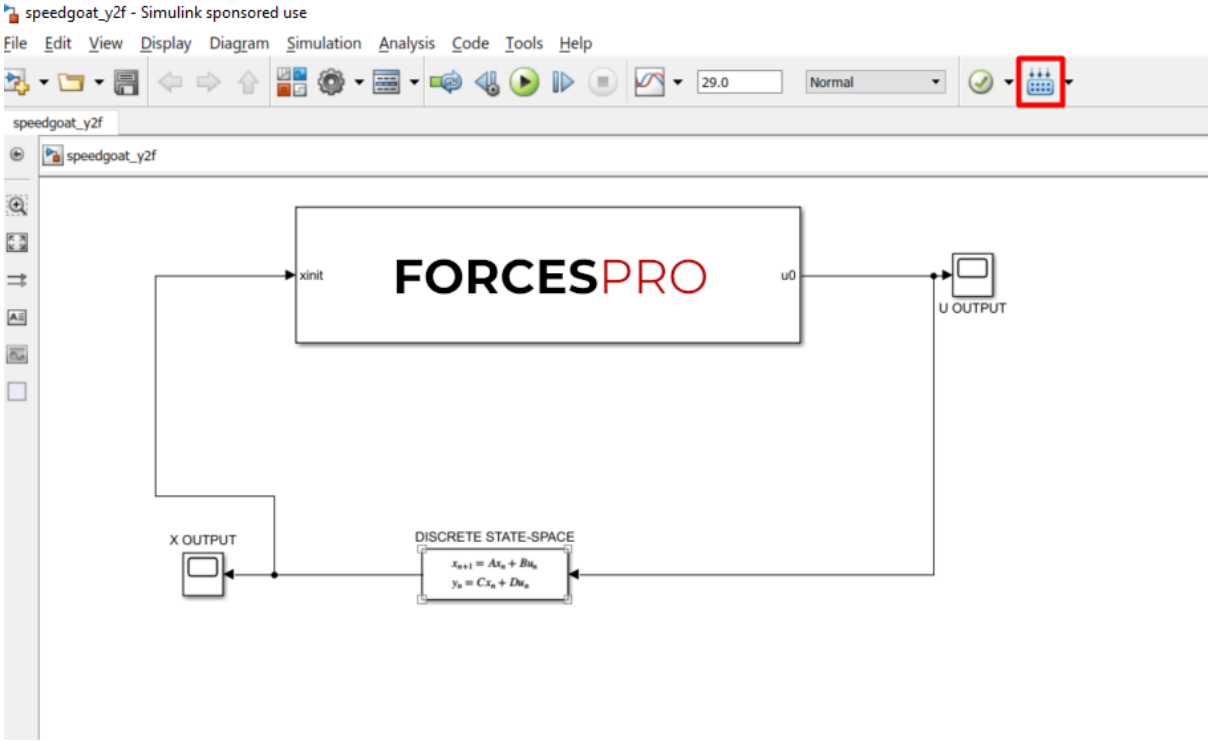


Figure 14.86: Compile the code of the Simulink model.

14.5 Speedgoat QNX

- *High-level interface*
  - *Instructions*
  - *Figures*
- *Y2F interface*
  - *Instructions*
  - *Figures*

Important: When deploying to a target hardware platform, the library included in the **lib_target** directory of the generated solver should be used instead of the library in the **lib** directory.

14.5.1 High-level interface

Instructions

The steps to deploy and simulate a FORCESPRO controller on a Speedgoat QNX platform are detailed below.

1. (Figure 14.87) Set the code generation options:

```
codeoptions.platform = 'Speedgoat-QNX'; % to specify the platform
codeoptions.printlevel = 0; % on some platforms printing is not supported
codeoptions.cleanup = 0; % to keep necessary files for target compile
```

and then generate the code for your solver (henceforth referred to as "FORCESNLPsolver", placed in the folder "BasicExample") using the high-level interface.

2. (Figure 14.88) Create a new Simulink model using the blank model template.
3. (Figure 14.89) Populate the Simulink model with the system you want to control.
4. (Figure 14.90) Make sure the `FORCESNLPsolver_simulinkBlock.mexw64` file (created during code generation) is on the MATLAB path.
5. (Figure 14.91) Open the `FORCESNLPsolver_lib.mdl` Simulink model file, contained in the `interface` folder of the `FORCESNLPsolver` folder created during code generation.
6. (Figure 14.92) Copy-paste the FORCESPRO Simulink block into your simulation model and connect its inputs and outputs appropriately.
7. (Figure 14.93) Access the Simulink Model's Settings.
8. (Figure 14.94) In the "Solver" tab, set the options:
   - Simulation start/stop time: Depending on the simulation wanted.
   - Solver type: Discrete or fixed-step.
   - Fixed-step size: Needs to be higher than the execution time of the solver.
9. (Figure 14.95) In the "Code Generation" tab, set the options:
   - System target file: `slrealtime.tlc`
   - Generate makefile: Off

10. (Figure 14.96) In the “Code Generation/Custom Code” tab, include the directories:
    - `BasicExample`
    - `BasicExample\FORCESNLPsolver\interface`
    - `BasicExample\FORCESNLPsolver\lib_target`
11. (Figure 14.97) In the “Code Generation/Custom Code” tab, add the source files:
    - `FORCESNLPsolver_simulinkBlock.c`
    - `FORCESNLPsolver_adtool2forces.c`
    - `FORCESNLPsolver_casadi.c`
12. (Figure 14.98) In the “Code Generation/Custom Code” tab, add the library file:
    - `libFORCESNLPsolver.a`
13. (Figure 14.99) Access the FORCESPRO block’s parameters.
14. (Figure 14.100) Remove “FORCESNLPsolver” and “FORCESNLPsolver_simulinkBlock” from the S-function module.
15. (Figure 14.101) Compile the code of the Simulink model. This will also automatically load the model to the connected Speedgoat platform.
16. Deployment is complete and simulations can now be run on the Speedgoat platform.

Figures

BasicExample.m

```
55
56
57    %% Generate FORCES solver
58
59    % get options
60 -  codeoptions = getOptions('FORCESNLPsolver');
61
62 -  codeoptions.platform = 'Speedgoat-QNX'; % to specify the platform
63
64 -  codeoptions.printlevel = 0; % on some platforms printing is not supported
65
66 -  codeoptions.cleanup = 0; % to keep necessary files for target compiling
67
68    % generate code
69 -  FORCES_NLP(model, codeoptions);
70
71
72    %% simulate
```

Figure 14.87: Set the appropriate code generation options.

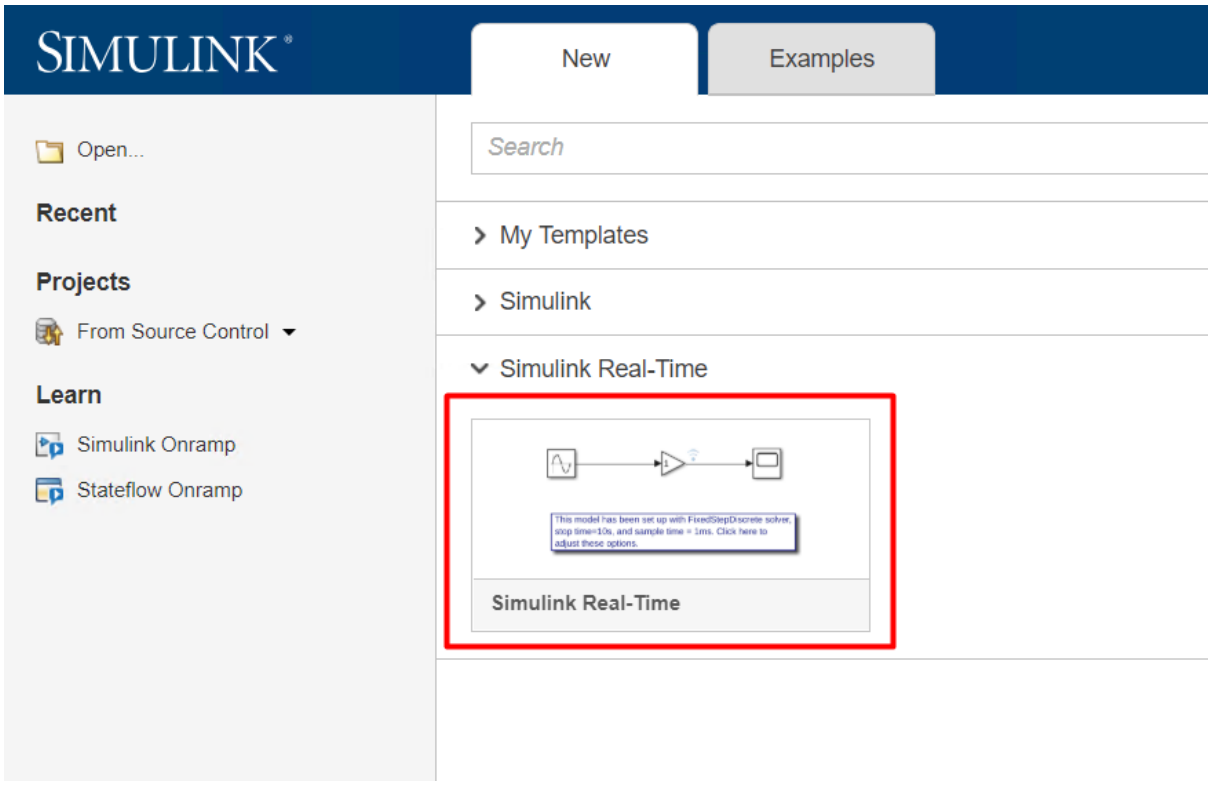


Figure 14.88: Create a Simulink model.

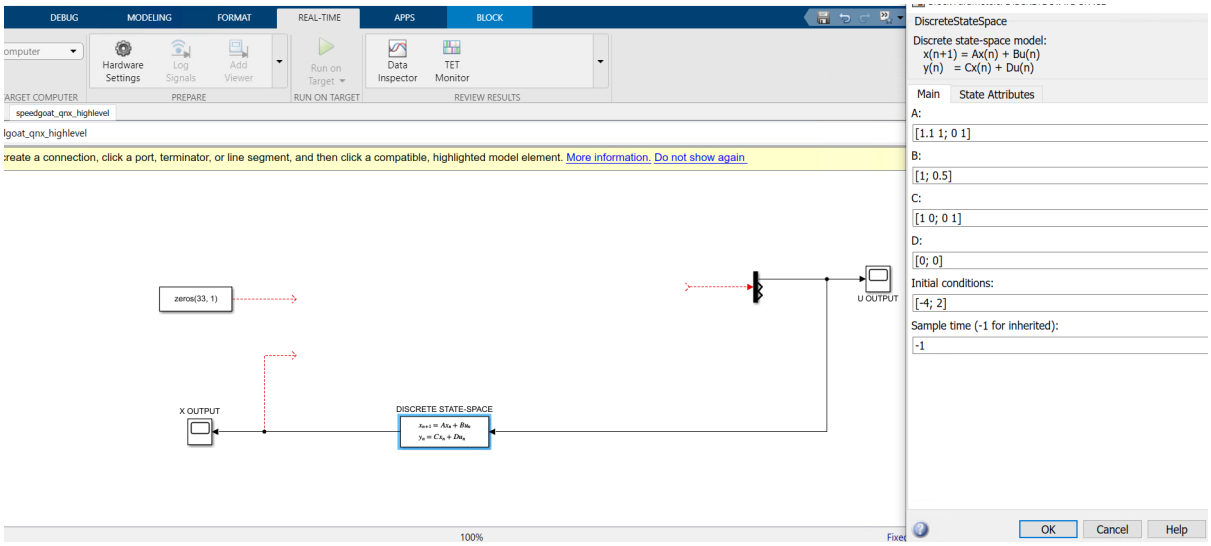


Figure 14.89: Populate the Simulink model.

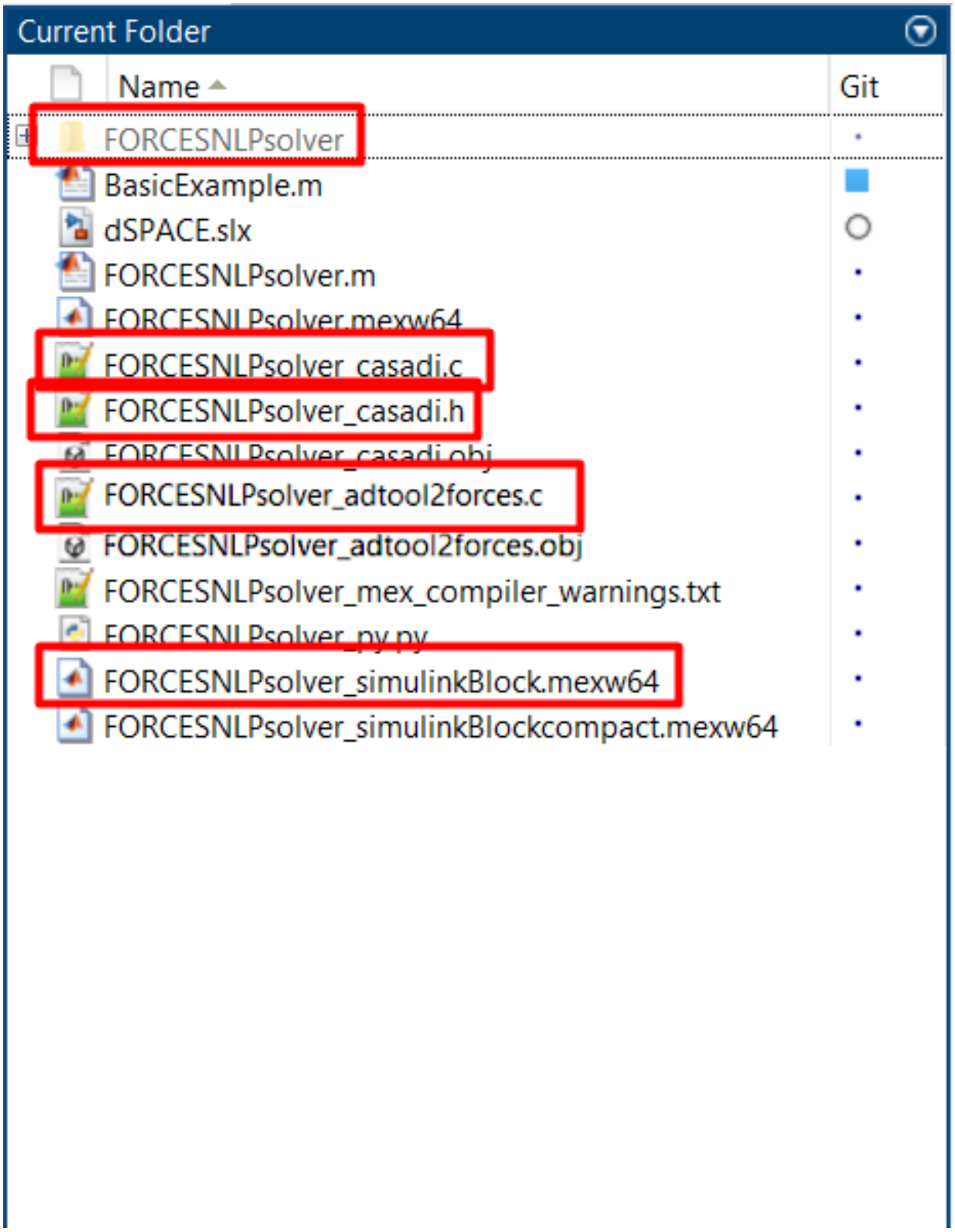


Figure 14.90: Add the folder containing the `.mexw64` solver file to the MATLAB path.

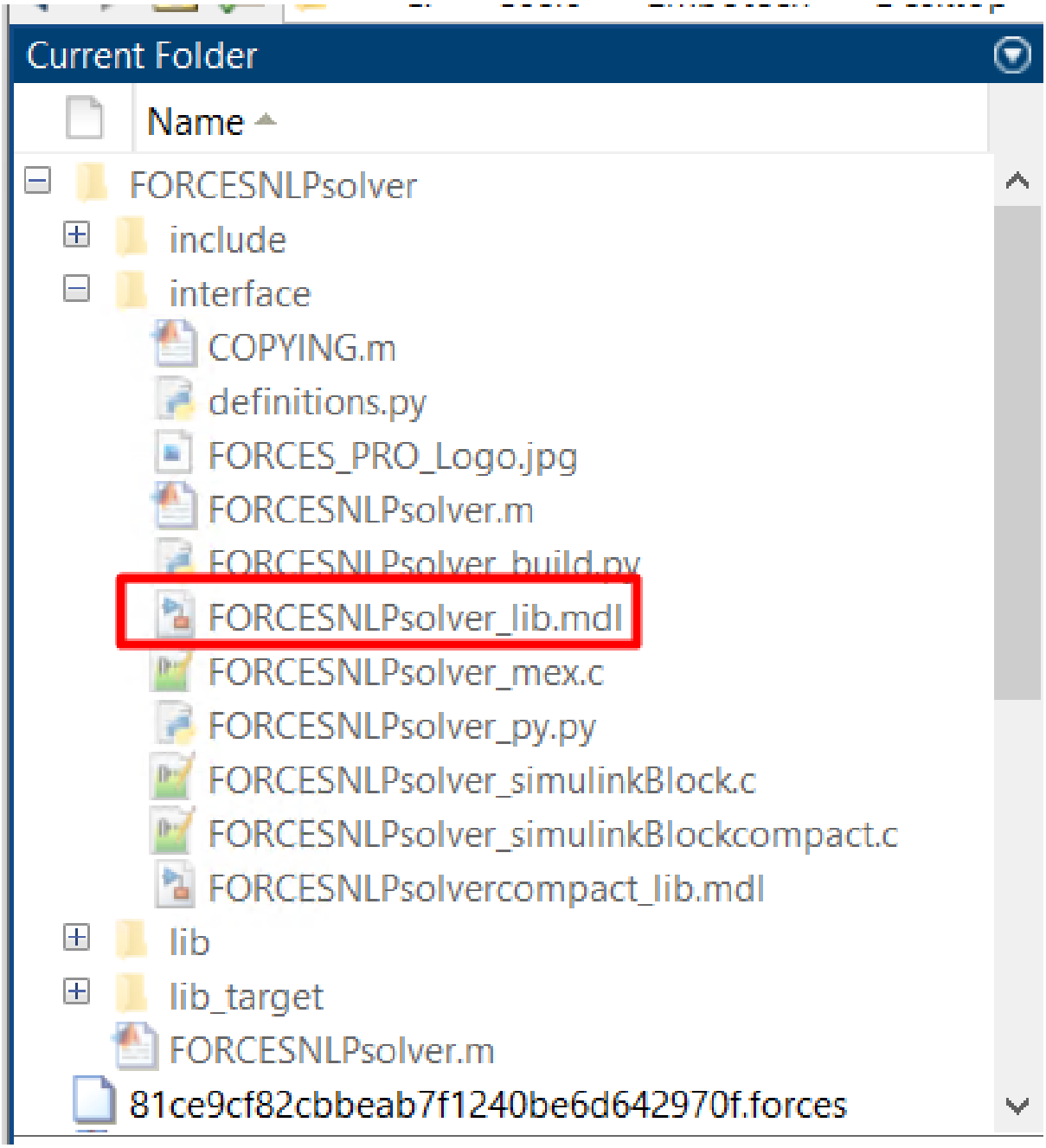


Figure 14.91: Open the generated Simulink solver model.

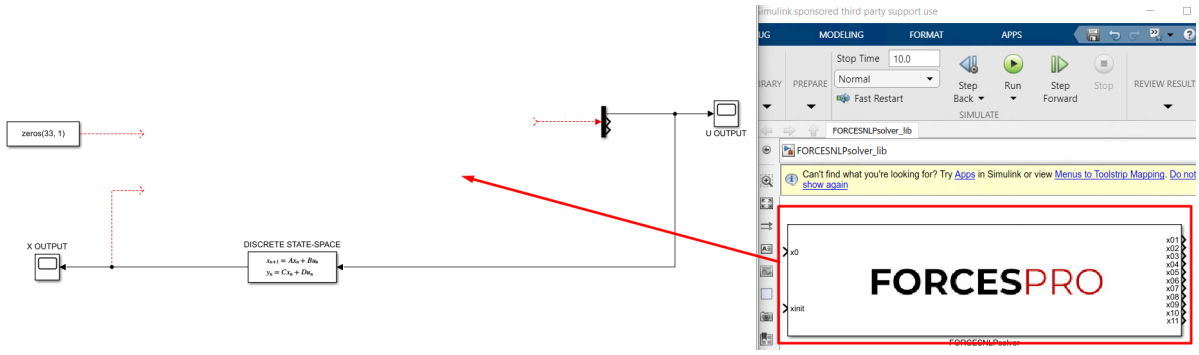


Figure 14.92: Copy-paste and connect the FORCESPRO block.

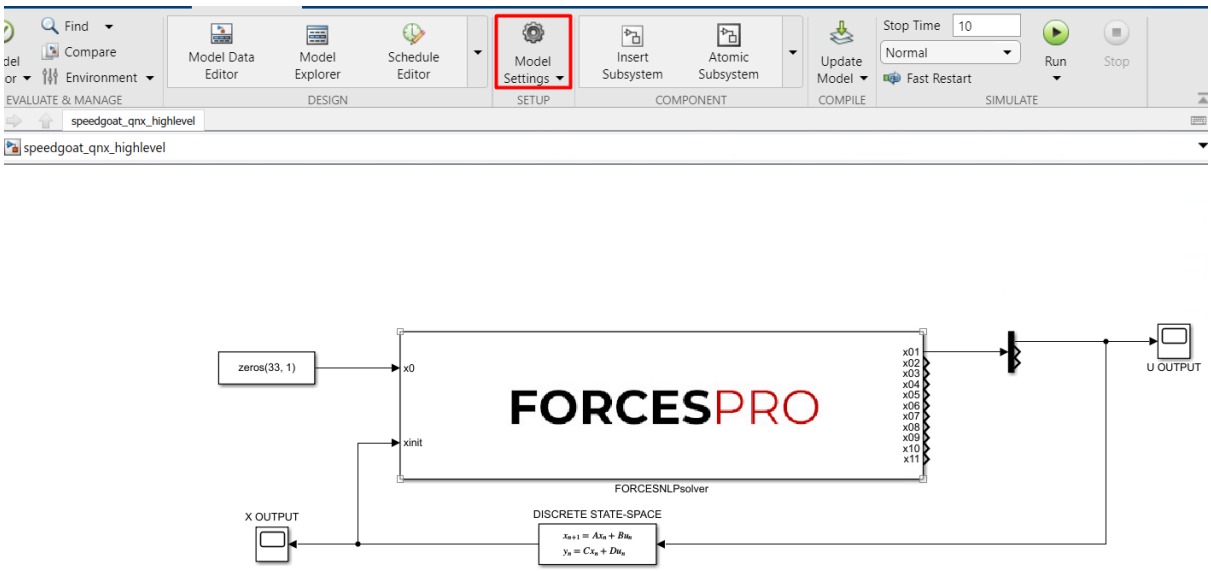


Figure 14.93: Open the Simulink Model Settings.

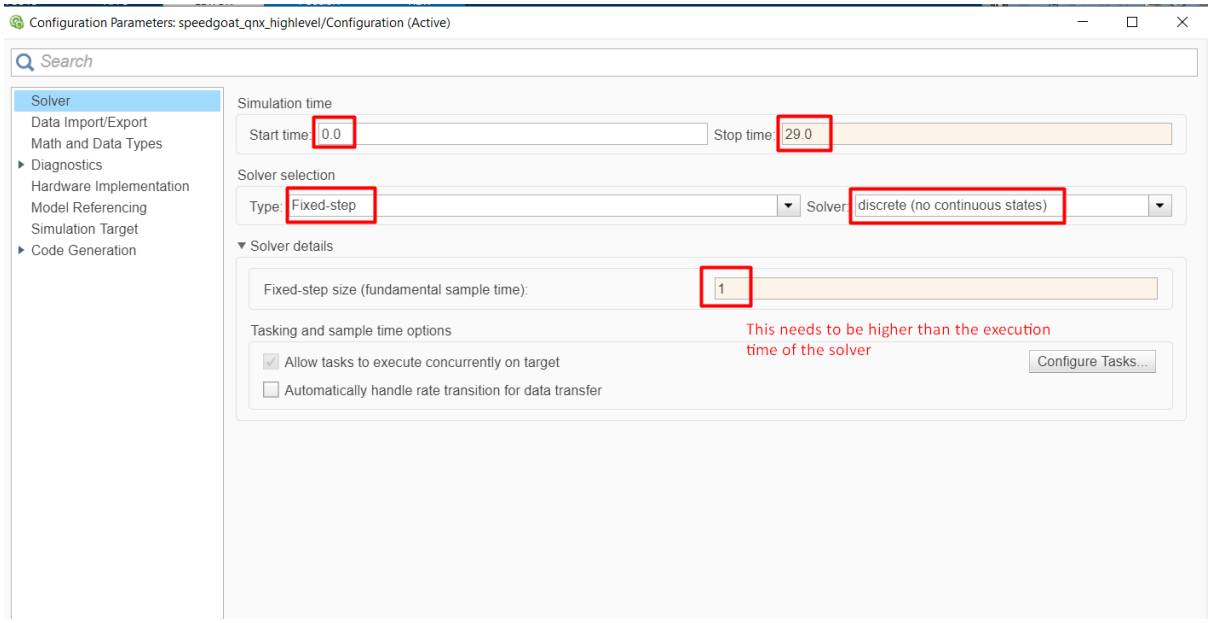


Figure 14.94: Set the Simulink solver options.

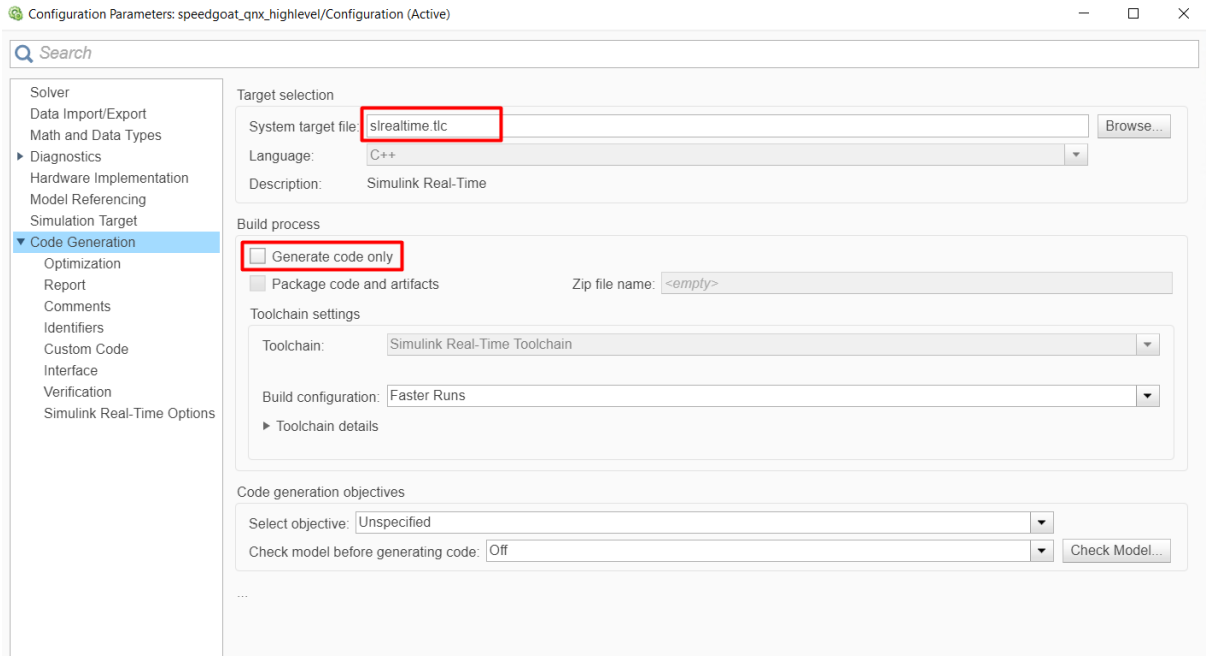


Figure 14.95: Set the Simulink code generation options.

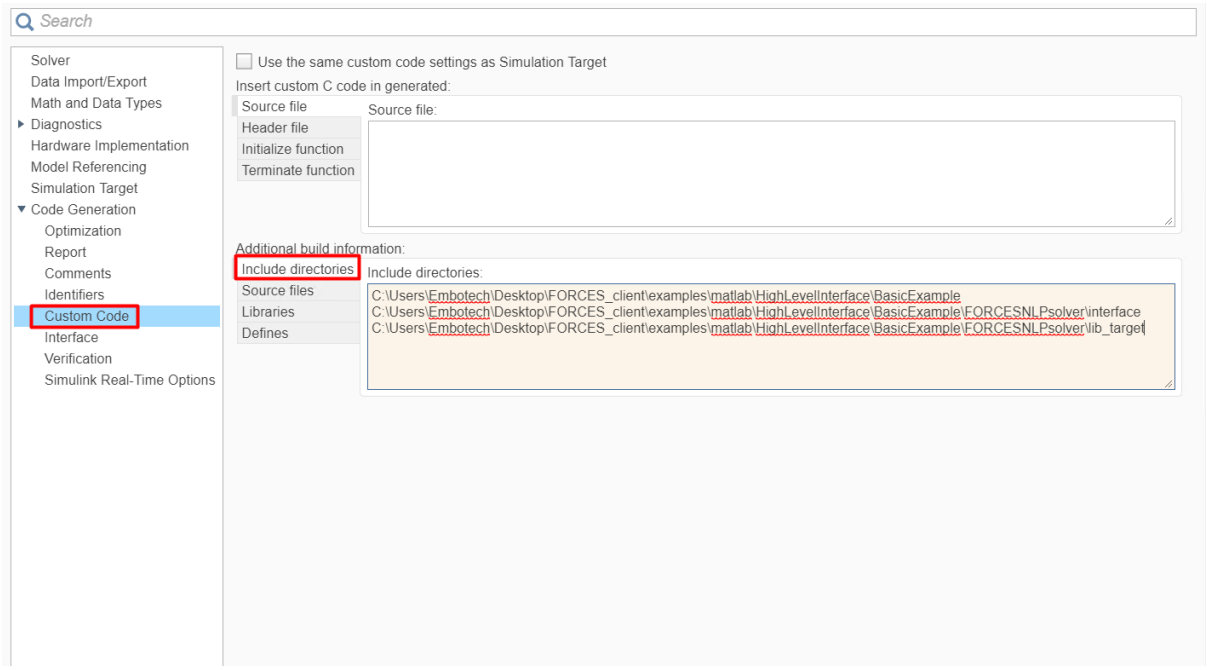


Figure 14.96: Add the directories included for the code generation.

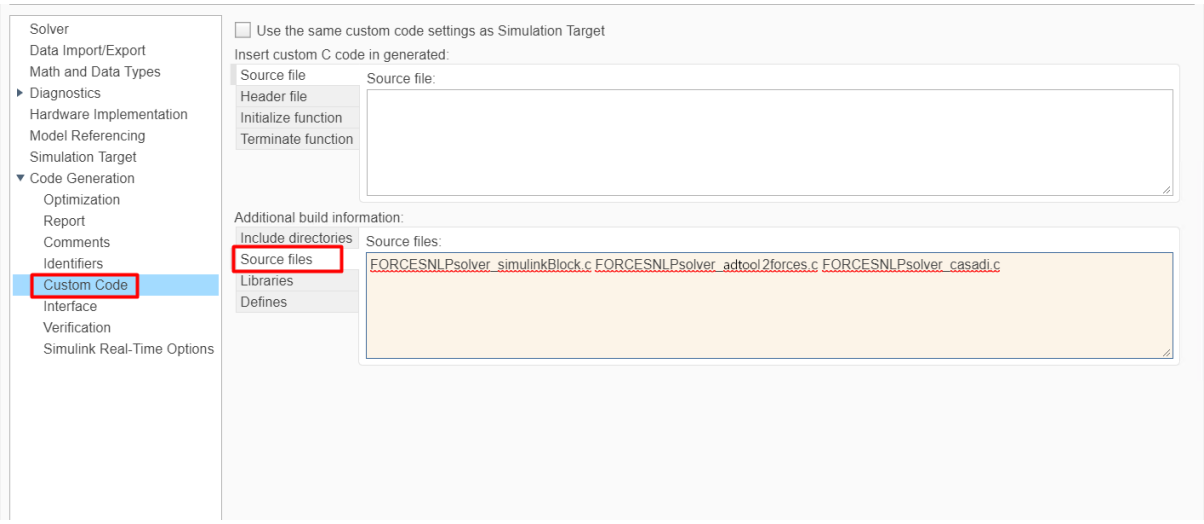


Figure 14.97: Add the source files used for the code generation.

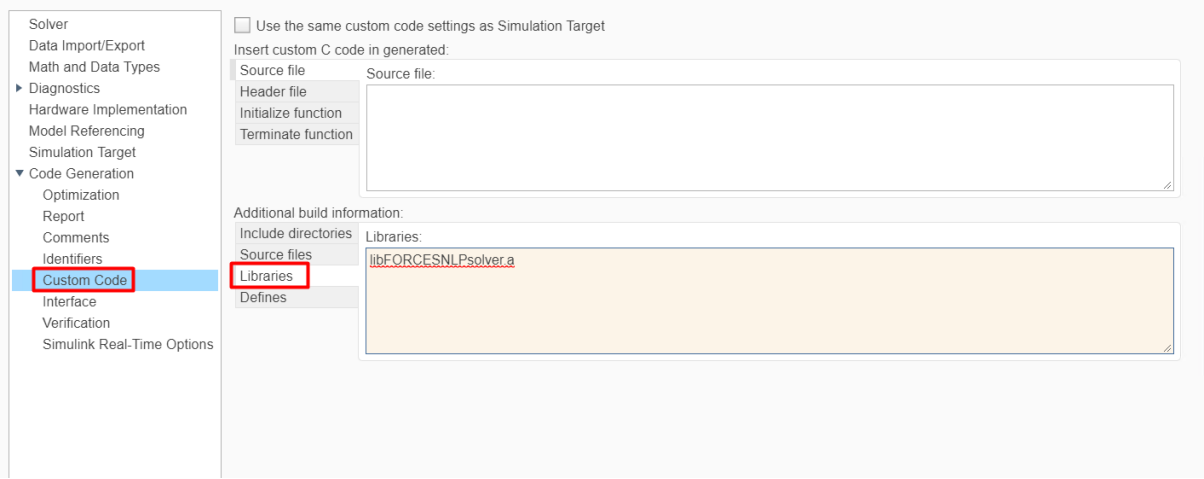


Figure 14.98: Add the libraries used for the code generation.

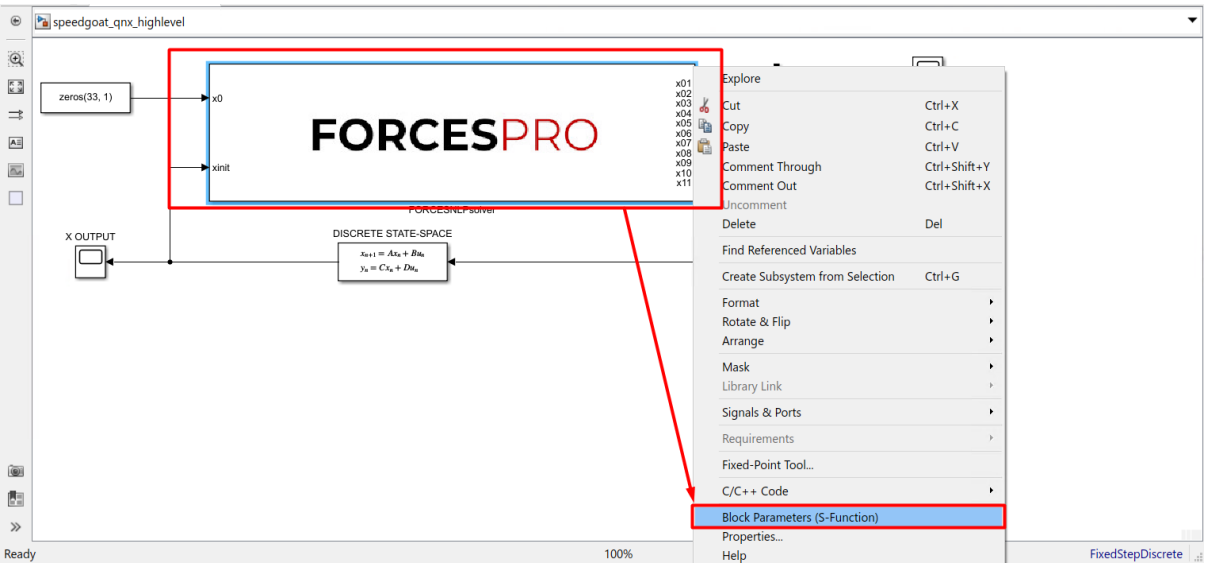


Figure 14.99: Open the FORCESPRO block's parameters.

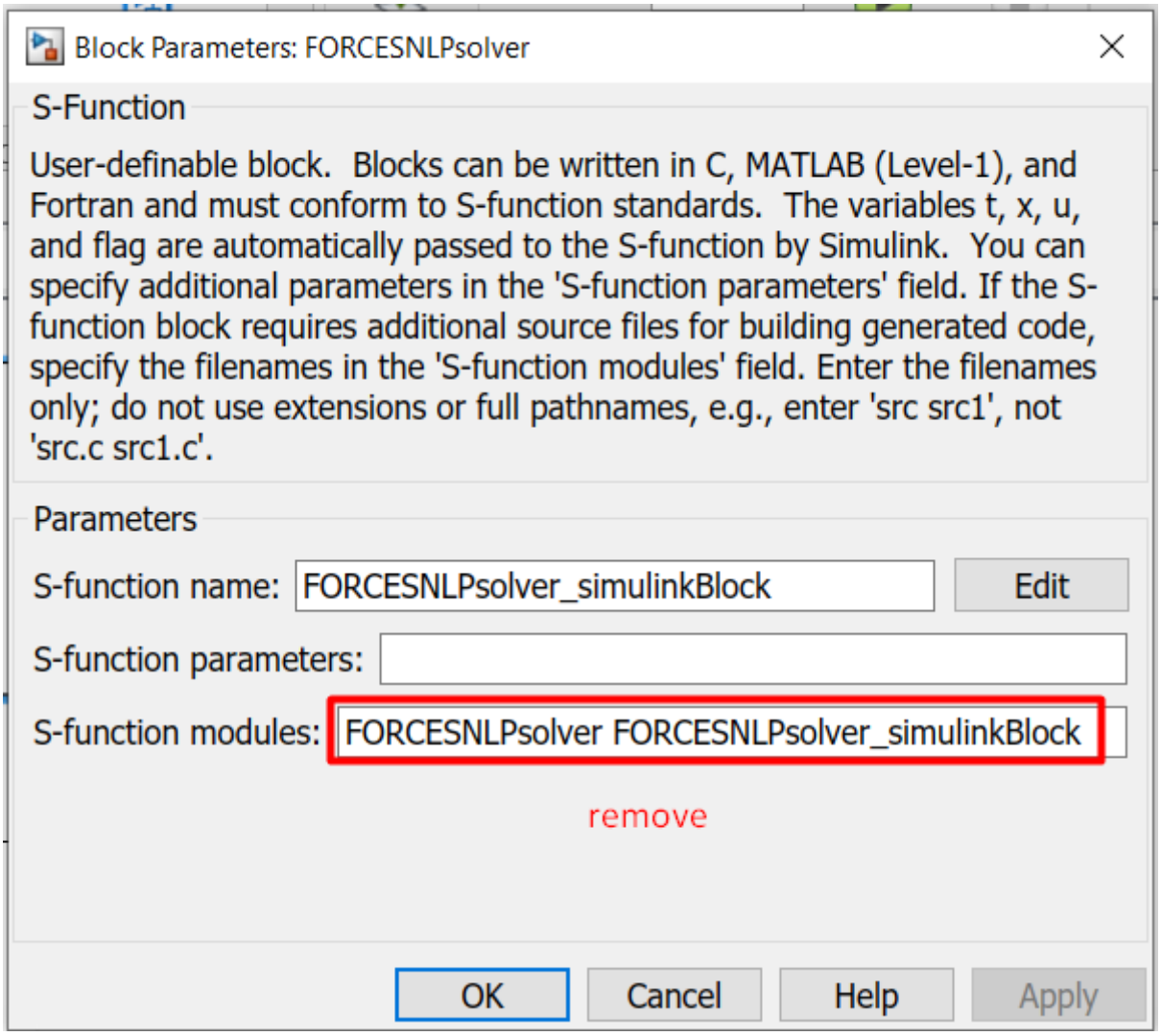


Figure 14.100: Remove the default data from the S-function module.

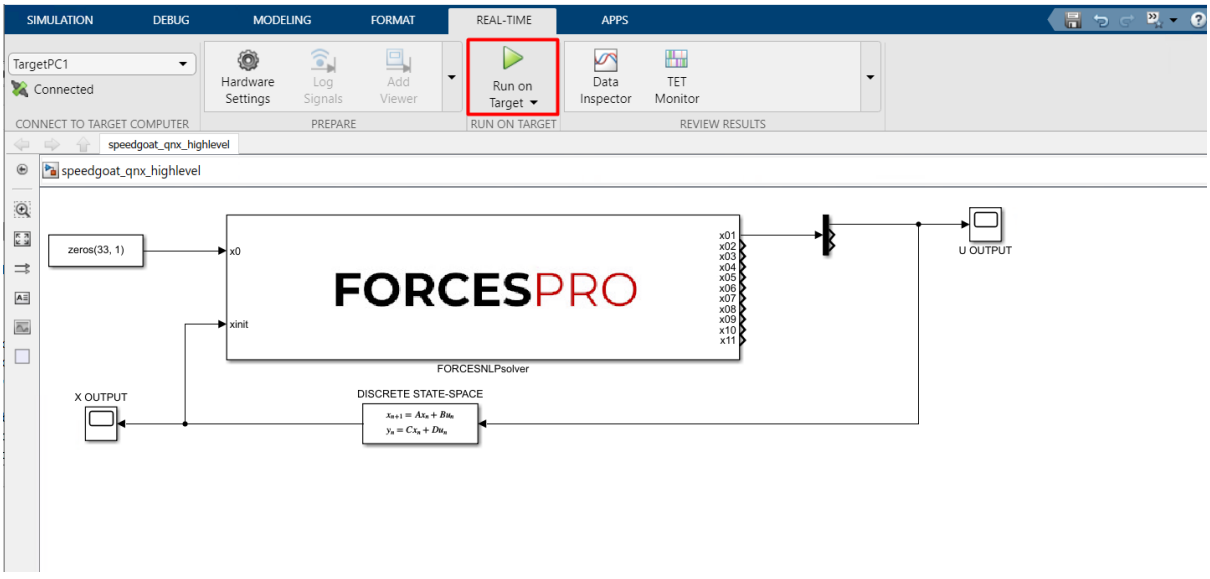


Figure 14.101: Compile the code of the Simulink model.

14.5.2 Y2F interface

Instructions

The steps to deploy and simulate a FORCESPRO controller on a Speedgoat QNX platform are detailed below.

1. (Figure 14.102) Set the code generation options:

```
codeoptions.platform = 'Speedgoat-QNX'; % to specify the platform
codeoptions.printlevel = 0; % on some platforms printing is not supported
```

and then generate the code for your solver (henceforth referred to as "simplempc_solver", placed in the folder "Y2F") using the Y2F interface.

2. (Figure 14.103) Create a new Simulink model using the blank model template.
3. (Figure 14.104) Populate the Simulink model with the system you want to control.
4. (Figure 14.105) Make sure the `simplempc_solver_simulinkBlock.mexw64` file (created during code generation) is on the MATLAB path.
5. (Figure 14.106) Copy-paste the FORCESPRO Simulink block, contained in the created `y2f_simulink_lib.slx` Simulink model file, into your simulation model and connect its inputs and outputs appropriately.
6. (Figure 14.107) Access the Simulink Model's Settings.
7. (Figure 14.108) In the "Solver" tab, set the options:
   - Simulation start/stop time: Depending on the simulation wanted.
   - Solver type: Discrete or fixed-step.
   - Fixed-step size: Needs to be higher than the execution time of the solver.
8. (Figure 14.109) In the "Code Generation/RTI general build options" tab, set the options:
   - System target file: `slrealtime.tlc`
   - Generate makefile: Off
9. (Figure 14.110) In the "Code Generation/Custom Code" tab, include the directories:
   - `Y2F`
   - `Y2F\simplempc_solver\interface`
   - `Y2F\simplempc_solver\lib_target`
10. (Figure 14.111) In the "Code Generation/Custom Code" tab, add the source files:
    - `simplempc_solver_simulinkBlock.c`
    - `simplempc_solver.c`
11. (Figure 14.112) In the "Code Generation/Custom Code" tab, add the library file:
    - `libi_simplempc_solver_1.a`
12. (Figure 14.113) Compile the code of the Simulink model. This will also automatically load the model to the connected Speedgoat platform.
13. Deployment is complete and simulations can now be run on the Speedgoat platform.

Figures

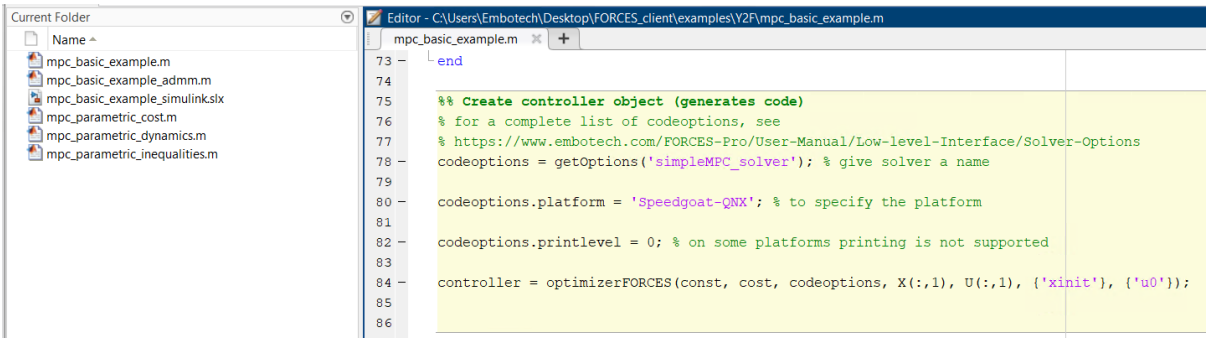


Figure 14.102: Set the appropriate code generation options.

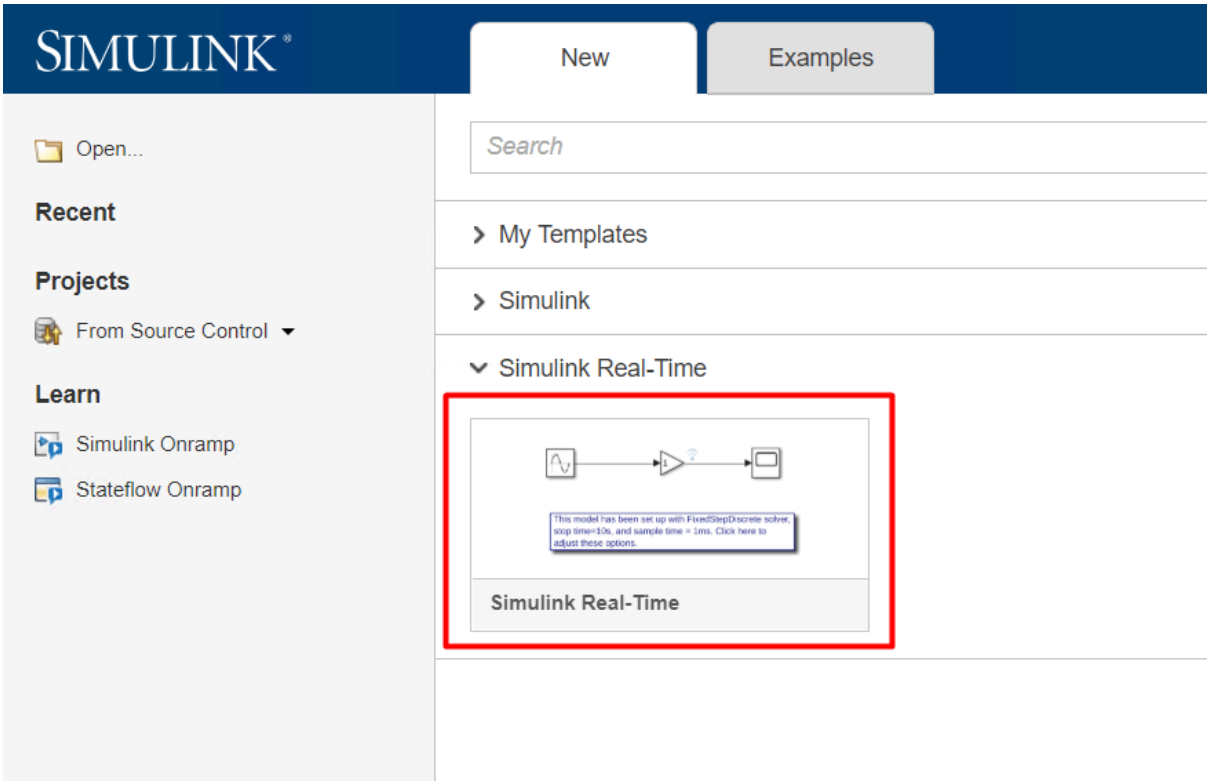


Figure 14.103: Create a Simulink model.

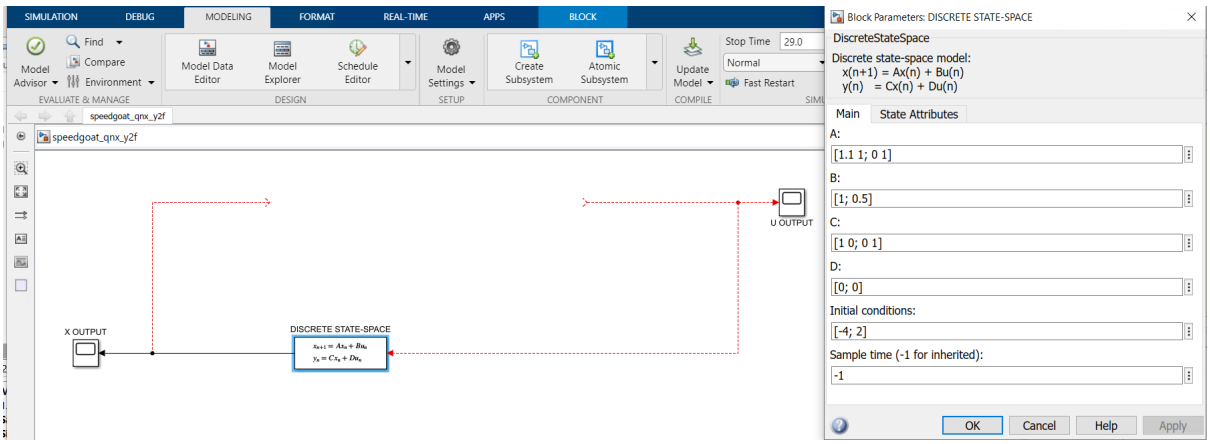


Figure 14.104: Populate the Simulink model.

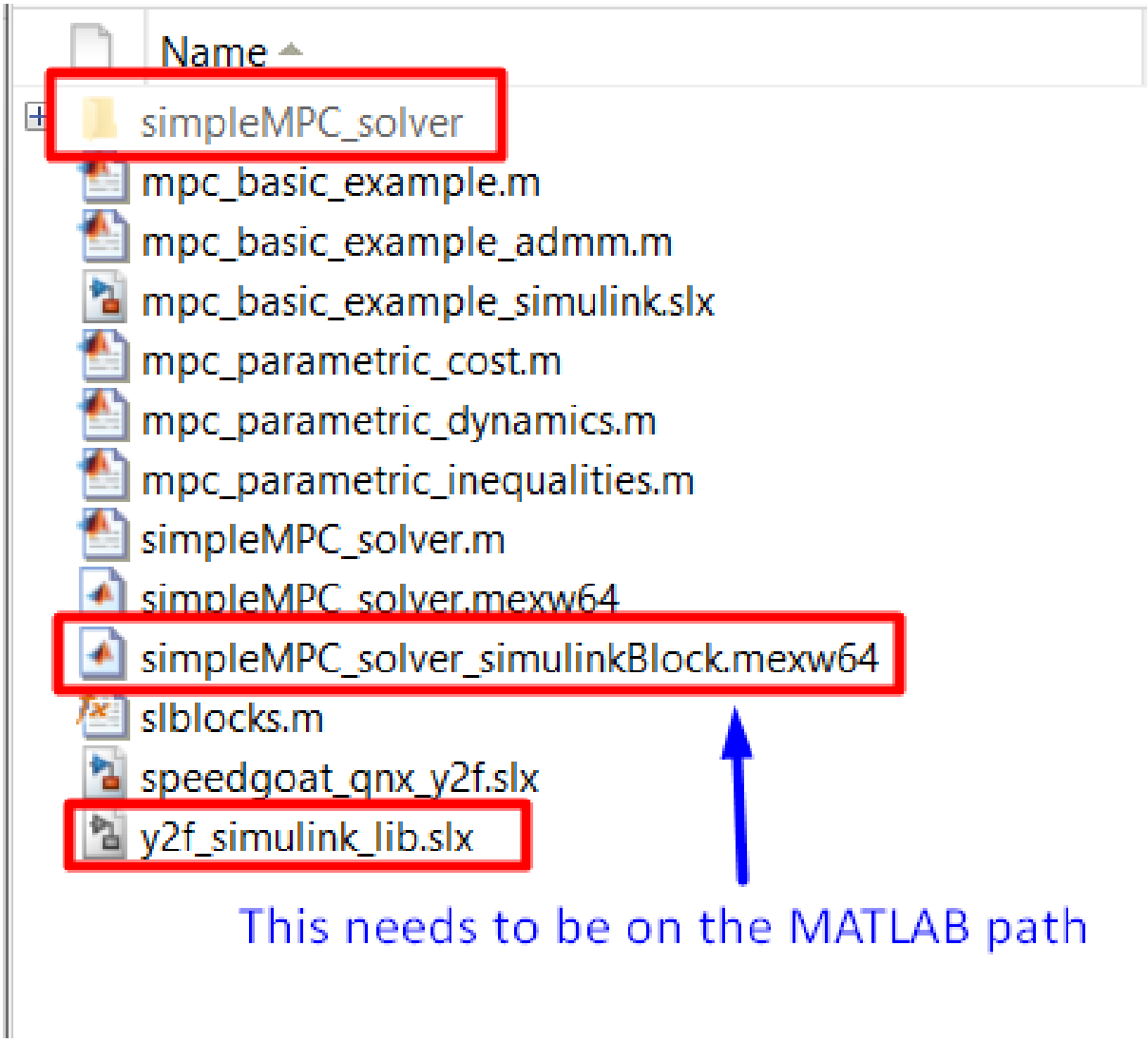


Figure 14.105: Add the folder containing the `.mexw64` solver file to the MATLAB path.

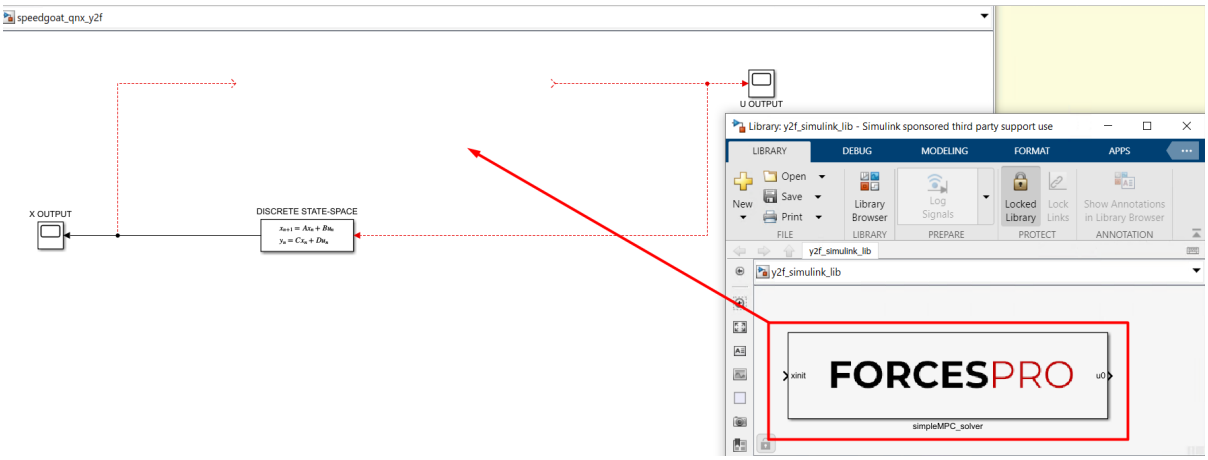


Figure 14.106: Copy-paste and connect the FORCESPRO block.

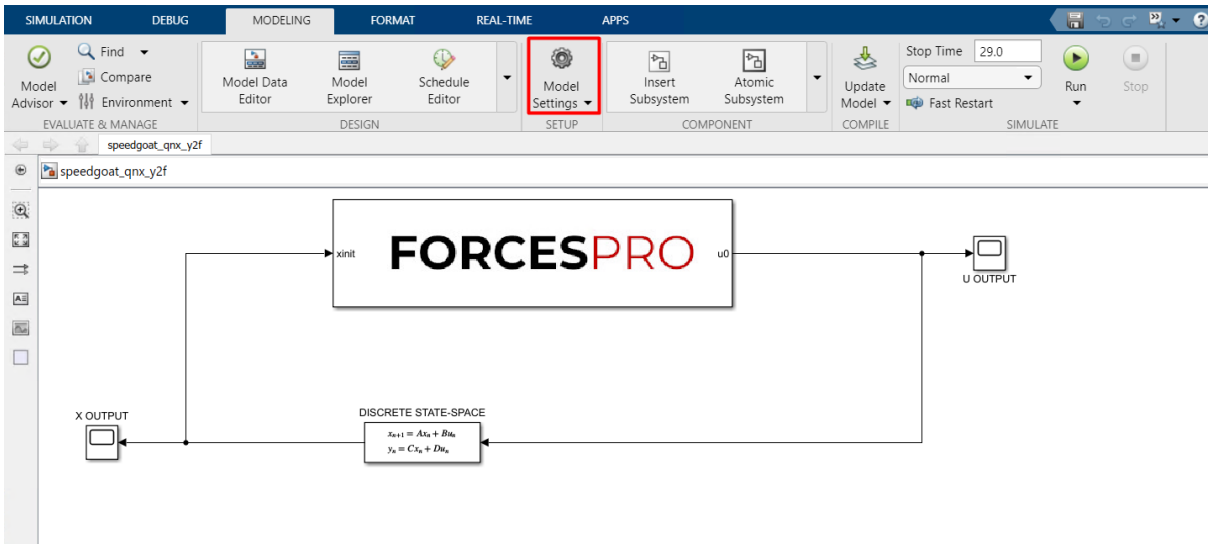


Figure 14.107: Open the Simulink Model Settings.

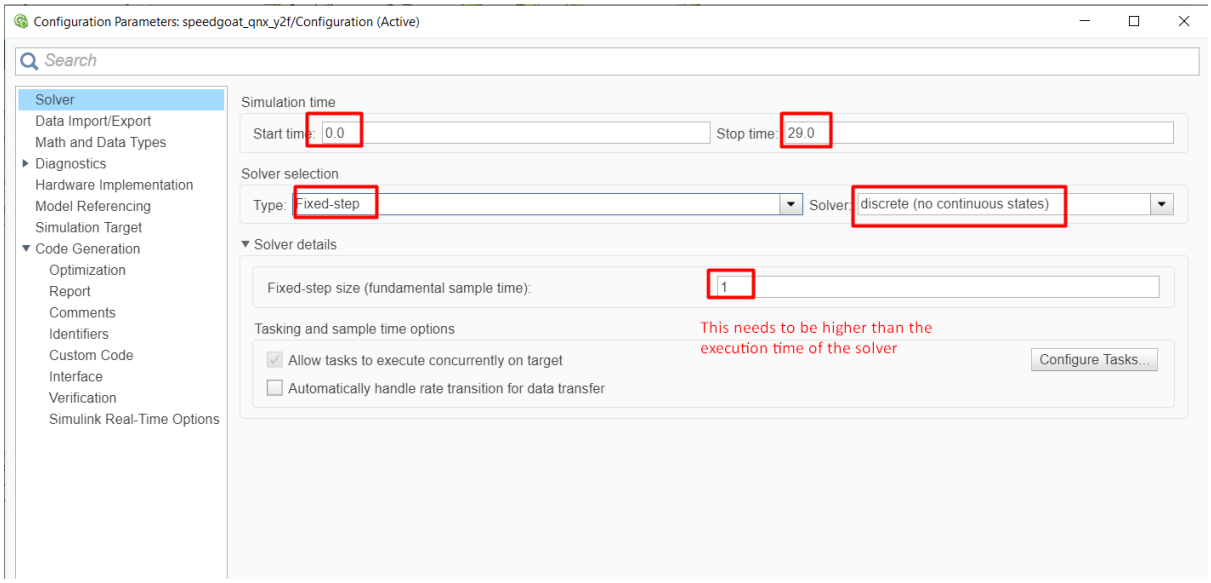


Figure 14.108: Set the Simulink solver options.

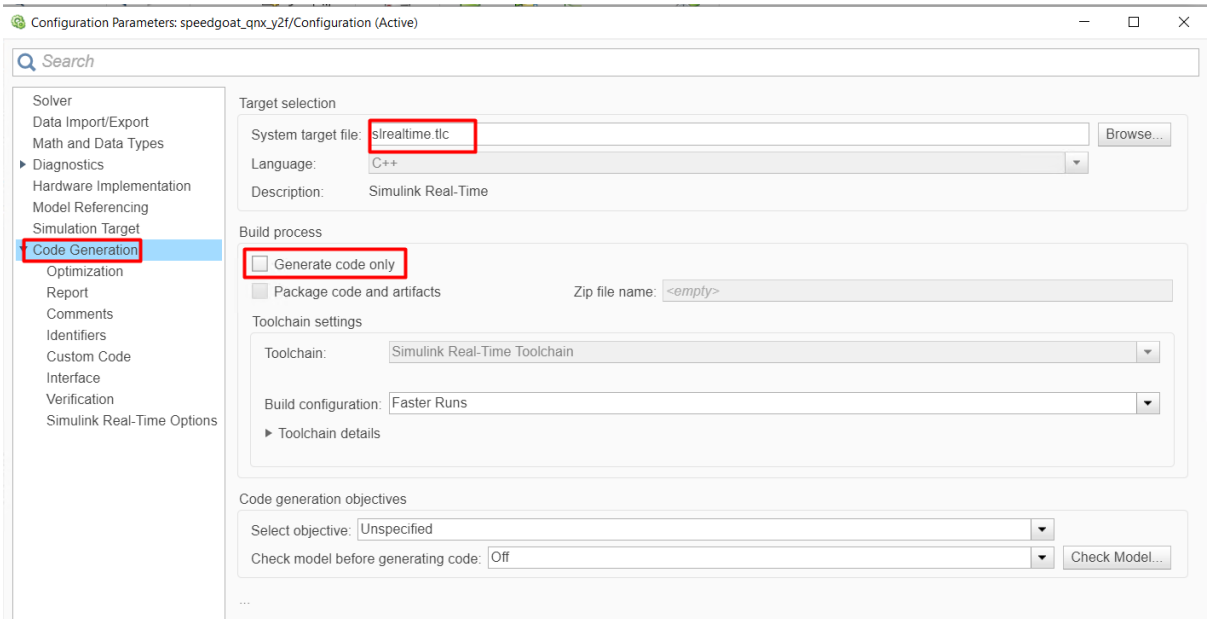


Figure 14.109: Set the Simulink code generation options.

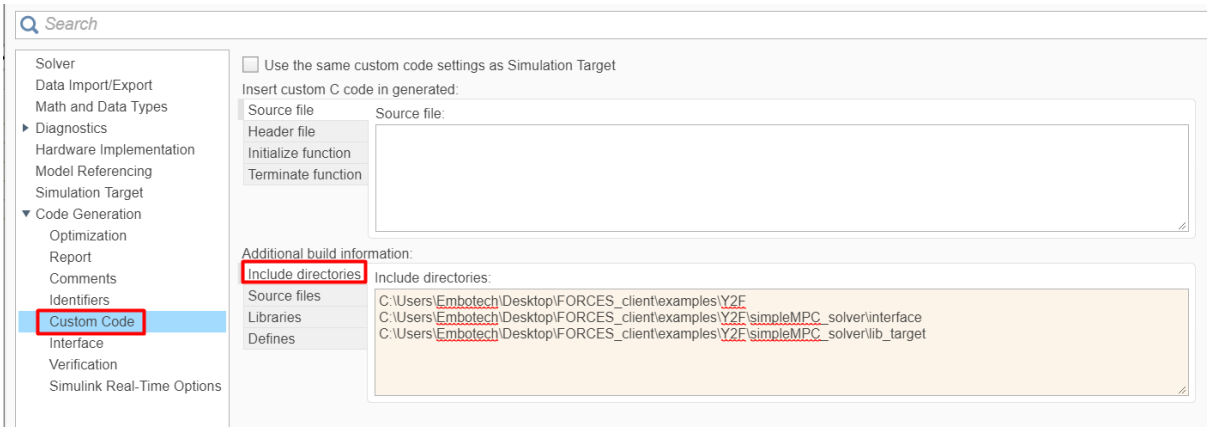


Figure 14.110: Add the directories included for the code generation.

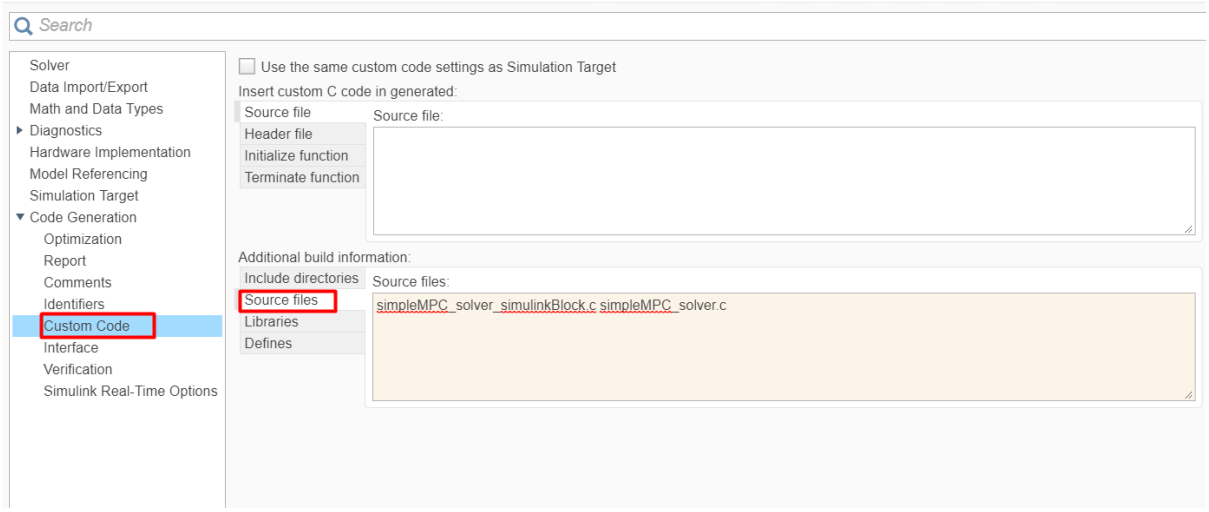


Figure 14.111: Add the source files used for the code generation.

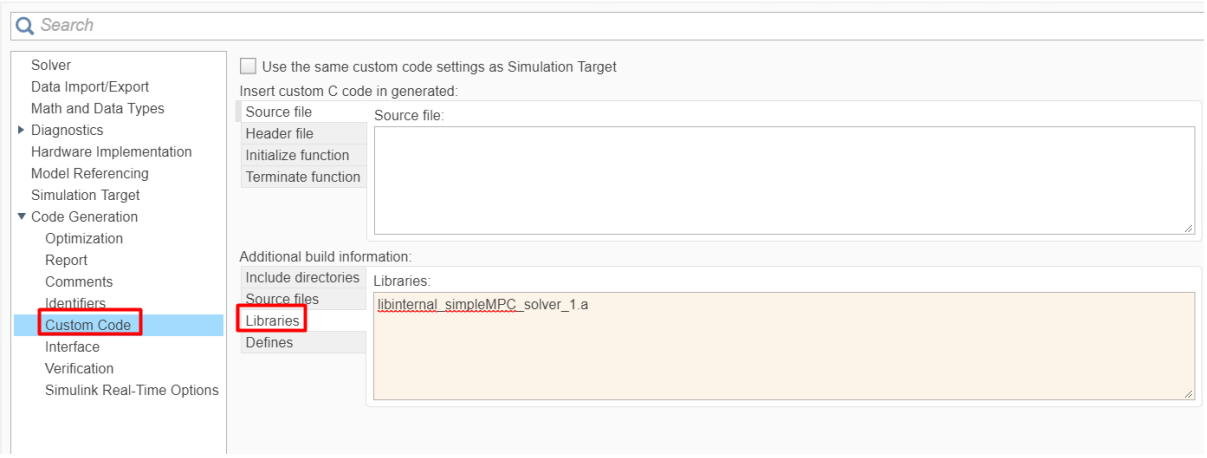


Figure 14.112: Add the libraries used for the code generation.

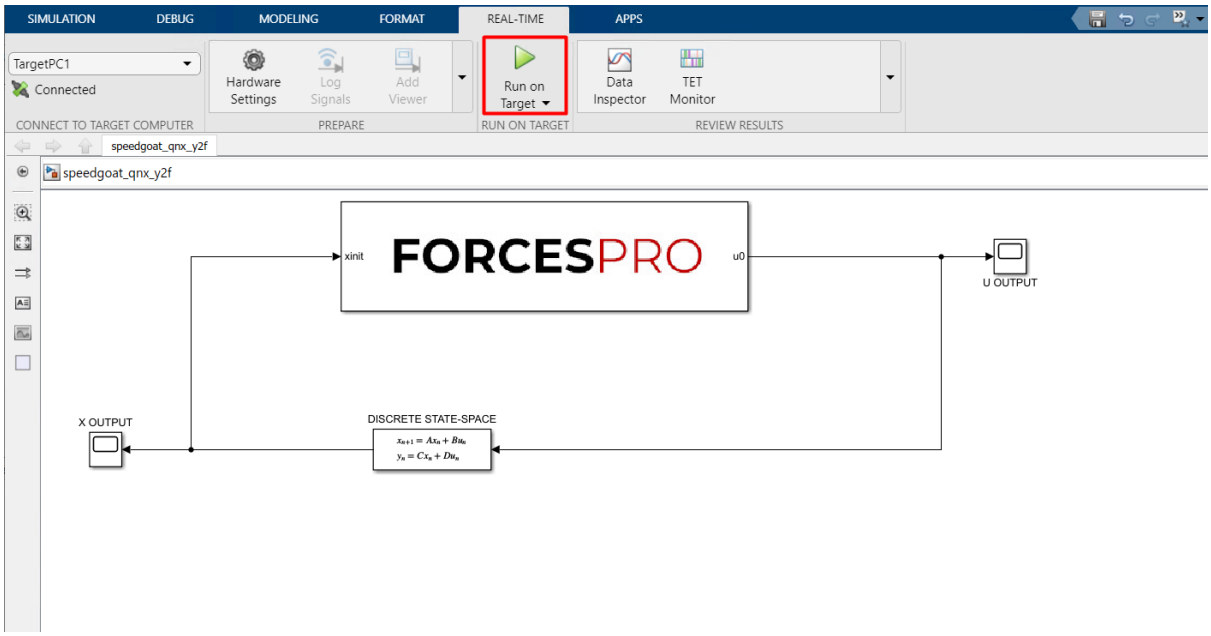


Figure 14.113: Compile the code of the Simulink model.

Chapter 15

Multicore parallelization

FORCESPRO supports two levels of multicore parallelism:

- Internal parallelism (*Internal parallelism*): the work for a solver is distributed over multiple cores
- External parallelism (*External parallelism*): a solver is run multiple times on multiple cores with different inputs

For combining both levels of parallelism, see section *Combining external and internal parallelism*.

15.1 Internal parallelism

FORCESPRO supports the computation on multiple cores, which is particularly useful for large problems and long horizons (the workload is split along the horizon to multiple cores). This is implemented by the use of OpenMP and can be switched on by using

Matlab

Python

```
codeoptions.parallel = 1;
```

```
codeoptions.parallel = 1
```

By default multicore computation is switched off.

When the parallel option is enabled with 1 (`codeoptions.parallel = 1`), the maximum number of threads to be used is set as the maximum number of threads available to OpenMP (`max_number_of_threads = omp_get_max_threads()`). Additionally, a runtime parameter `num_of_threads` is created to control the number of threads in runtime. The allowed range of values for the runtime parameter is `[1, max_number_of_threads]`. Leaving the parameter unset or setting a value outside the allowed range will lead in execution with the maximum number of threads (`max_number_of_threads`).

The maximum number of threads can also be set manually during code generation by setting:

Matlab

Python

```
% <max_number_of_threads> larger than 1
codeoptions.parallel = <max_number_of_threads>;
```

```
# <max_number_of_threads> larger than 1
codeoptions.parallel = <max_number_of_threads>
```

15.2 External parallelism

External parallelism can be enabled only on the level of the C interface (see *C interface* and *Memory allocations*). In order to execute multiple calls to the same generated solver in parallel, the solver is required to be thread-safe. Thread-safety can be ensured by setting the option

Matlab

Python

```
codeoptions.threadSafeStorage = 1;
```

```
codeoptions.threadSafeStorage = 1
```

The solver can be called in parallel by assigning an independent memory buffer to each thread as in the following code snippet:

```
/* each of the NUM_THREADS threads must be assigned its own memory buffer */
char * mem[NUM_THREADS];
FORCESNLPsolver_mem * mem_handle[NUM_THREADS];

/* input & output for each of the NUM_SOLVERS solvers */
FORCESNLPsolver_params params[NUM_SOLVERS];
FORCESNLPsolver_info info[NUM_SOLVERS];
FORCESNLPsolver_output output[NUM_SOLVERS];
int exit_code[NUM_SOLVERS];

/* create memory buffer for each thread */
for (i=0; i<NUM_THREADS; i++)
{
    mem[i] = malloc(mem_size);
    mem_handle[i] = FORCESNLPsolver_external_mem(mem[i], i, mem_size);
}

/* parallel call to the solver using OpenMP */
#pragma omp parallel for
for (i_solver=0; i_solver<NUM_SOLVERS; i_solver++)
{
    int i_thread = omp_get_thread_num();
    exit_code[i_solver] = FORCESNLPsolver_solve(&params[i_solver], &output[i_solver],␣
↪&info[i_solver], mem_handle[i_thread], ...);
}

/* free user-allocated memory */
for (i = 0; i < NUM_THREADS; i++)
{
    free(mem[i]);
}
```

If you run multiple concurrent solvers and if you're interested only in one solution, you can use the real-time parameter `solver_exit_external` to terminate all other solvers from the thread that converges fastest. See section *Early-terminate solver* for more information.

Important: Special care has to be taken when using `solvemethod = 'SQP_NLP'` with `problem.reinitialize = 0` or `solvemethod = 'QP_FAST'` with `problem.warmstart = 1` in parallel because the memory buffer stores the current state of the solver between consecutive calls. Therefore, the memory buffers have to be allocated per solver and not per thread, so that there are NUM_SOLVERS buffers.

Important: When using the code-generated integrators (see section *Code-generated integrators*) within a multithreaded environment, you will have to specify via the option `nlp.max_num_threads` the maximum number of threads on which you wish to run the solver in parallel. For instance, if running the solver on a maximum of 5 threads in parallel one would set

Matlab

Python

```
codeoptions.nlp.max_num_threads = 5;
```

```
codeoptions.nlp.max_num_threads = 5
```

Note: Solvers with binary (*Binary constraints*) or integer variables (*Mixed-integer nonlinear solver*) are not yet thread safe.

Alternatively, the internal memory interface (see *Internal memory*) also supports thread safety, but with less flexibility and with a hard limit on the number of memory buffers (see Table 13.1). This functionality is not covered here but you can get started by consulting the example `BasicExample_internal_mem_multithreading.c` in the `examples\StandaloneExecution\C` folder that comes with your client.

15.3 Combining external and internal parallelism

In order to combine external and internal parallelism on `m` internal threads and `n` external threads (so that `m*n` threads are employed in total), you need to set the following code options:

Matlab

Python

```
codeoptions.parallel = m;
codeoptions.max_num_mem = n; % only for internal memory interface
codeoptions.nlp.max_num_threads = m*n; % only for code-generated integrators
```

```
codeoptions.parallel = m
codeoptions.max_num_mem = n # only for internal memory interface
codeoptions.nlp.max_num_threads = m*n # only for code-generated integrators
```

If these options are set inconsistently with the number of threads, the solver will exit with exitflags `-101` (for insufficient `max_num_mem`) or `-102` (for insufficient `max_num_threads`).

In OpenMP, nested parallelism needs to be enabled. Depending on the compiler and OpenMP version, one or both of the following library calls are required before calling the solver:

```
omp_set_nested(1);
omp_set_max_active_levels(2);
```

Additionally, dynamic adjustment of the number of threads needs to be disabled by

```
omp_set_dynamic(0);
```

Chapter 16

Licensing

- *Machine Identification*
  - *Client Identification*
  - *Solver Identification*
- *Static License*
  - *System requirements for static license*
  - *Generating solvers with static license*
  - *Running solvers with static license*
- *License Files*
  - *System requirements for license files*
  - *Generating solvers with license files*
  - *Generating license files*
  - *Running solvers with license files*
- *Floating Licenses*
  - *Floating Licenses Proxy*
  - *System requirements for floating licenses*
  - *Using the Floating Licenses Proxy*
  - *Floating License Attributes*
  - *Generating solvers that use floating licenses*
  - *Configure floating licenses during code generation*
  - *Running solvers with floating licenses*

16.1 Machine Identification

The FORCESPRO licensing system works by receiving unique identifiers from the machines the software runs on and enabling the machines by activating the corresponding unique identifiers. Activation of machines can be done by receiving the unique identifiers of the machines using fingerprinting executables provided in the portal and adding those unique identifiers on the portal.

For more information on machine activation see: https://my.embotech.com/readme

16.1.1 Client Identification

Machines running FORCESPRO clients are licensed using the machine's username and the machine's unique identifier.

16.1.2 Solver Identification

Machines running FORCESPRO solvers are licensed using the machine's unique identifier.

16.2 Static License

When generating a solver the license's state on the portal (enabled machines and expiration) is saved in the solver so that the solver can run on the enabled machines.

16.2.1 System requirements for static license

The requirement for static license checking is to have correct system clock settings (accurately showing current time, compliant to UTC time).

16.2.2 Generating solvers with static license

Static license checking is automatically enabled on a generated solver.

16.2.3 Running solvers with static license

After generating a solver, you can move it to the running platform and build it with the rest of your project.

16.3 License Files

License Files are used in order to enable solvers to run in machines that were not enabled during the time of code generation or to enable solvers to run after a license renewal (that happened after solver code generation).

16.3.1 System requirements for license files

The requirements for using license files are:

- A platform supporting I/O operations
- A platform with access to file system
- Correct system clock settings (accurately showing current time, compliant to UTC time)
- Using the MATLAB interface of FORCESPRO

16.3.2 Generating solvers with license files

License file checking is automatically enabled on a generated solver (supposing the platform supports it). The user has the option to select the name of the license file using the following codeoption:

Matlab

Python

```
% MATLAB
codeoptions.license.static_license_file_name = '<filename_without_extension>'; % no␣
↪paths, only filename
```

```
% Python
codeoptions.license.static_license_file_name = '<filename_without_extension>' # no␣
↪paths, only filename
```

Important: The license file name must be a valid variable name

Note: Until **FORCESPRO 5.0.0**, the name of the license file could be set using the option **codeoptions.license_file_name**. This codeoption has been kept for backwards compatibility, however, setting the new option will override it.

16.3.3 Generating license files

License files can be created by using the MATLAB function `ForcesGetLicenseFile`. This function can be called with the following (optional) arguments:

- `license file name`: Name to be given to created license file (without extension). Default value: `FORCESPRO`
- `server`: FORCESPRO server to use to generate the license file. Default value: default server used by client

For more information on function usage run: `help ForcesGetLicenseFile` in the MATLAB Command Window.

16.3.4 Running solvers with license files

After generating a solver, you can move it to the running platform and build it with the rest of your project. After generating a license file, you can move it to your project folder.

When running a solver:

- The solver will read the license file and validate the license
- The license file needs to be in the working directory in which you execute your project (usually the directory that contains the executable)

16.4 Floating Licenses

Floating Licenses are used when the system that is enabled for running solvers needs to frequently change or is a virtualized environment (such as Docker or Virtualbox). The licensing

works by getting a temporary local lease from the floating license server in order to be able to run a solver on a machine. The connection to the floating license server is performed via a proxy server which receives the requests from the solvers, contacts the online floating license server to receive the floating license and provides it to the solver.

16.4.1 Floating Licenses Proxy

The deployment scenarios running FORCESPRO solvers do not always include the provision of an internet connection to the deployment machines. Instead, the deployment machines are managed by a centralized mechanism. The Floating Licenses Proxy allows to use this functionality for the provision of the FORCESPRO floating licenses. The Proxy can be set up either as a standalone application or as a web server in a machine which has network access with the deployment machines which will run the FORCESPRO solvers (without the need of an internet connection). This machine will listen for requests from the FORCESPRO solvers running in the deployment machines, perform the request on the online floating license server and finally return the acquired license leases to the FORCESPRO solvers. This way, the only machine requiring an internet connection will be the one running the Proxy. The solvers can be configured *during code generation* to which machine to connect to, in order to access the Floating Licenses Proxy. The packages for the Floating Licenses Proxy can be acquired in the Customer Portal in the Engineering Nodes view.

16.4.2 System requirements for floating licenses

The requirements for running the floating licenses proxy:

- A `x86/x86_64` Windows or Linux platform
- An internet connection
- Correct system clock settings (accurately showing current time, compliant to UTC time)
- OS specific system or packages requirements can be found in the manual of the Floating Licenses Proxy

The requirements for enabling solvers with floating licenses are:

- A `x86/x86_64` Linux platform
- A network connection of the running platform to the machine running the floating licenses proxy
- Correct system clock settings (accurately showing current time, compliant to UTC time)

16.4.3 Using the Floating Licenses Proxy

The Floating Licenses Proxy can be run as a standalone application or as a web server. Once run, no further actions are required on the Proxy's side. The FORCESPRO solvers will connect to it and receive the license leases to be validated for execution.

16.4.4 Floating License Attributes

Floating licenses are defined by the following two fields:

- **Number of Licenses**: The number of machines that can run solvers concurrently using a floating license for a FORCESPRO user.
- **Lease Time**: The time for which a local lease is valid after it has been granted. Default lease time is 10 minutes. Please contact support@embotech.com to change this.

16.4.5 Generating solvers that use floating licenses

To enable floating licenses on a generated solver use the following codeoption:

Matlab

Python

```
% MATLAB
codeoptions.license.use_floating_license = 1;
```

```
# Python
codeoptions.license.use_floating_license = 1
```

And select the platform to use

Matlab

Python

```
% MATLAB
codeoptions.platform = 'platform_name';
```

```
# Python
codeoptions.platform = "platform_name"
```

Available platform options are:

- **Gnu-x86**
- **Gnu-x86_64**
- **Docker-Gnu-x86**
- **Docker-Gnu-x86_64**

Note: Until **FORCESPRO 5.0.0**, enabling floating licenses was done using the codeoption **codeoptions.useFloatingLicense**. This codeoption has been kept for backwards compatibility, however, setting the new option will override it.

16.4.6 Configure floating licenses during code generation

During code generation, the following codeoptions are available to configure the floating license check:

- **codeoptions.license.floating_license_server**: Selects the hostname or ip address of the machine which is running the floating license proxy for the solver to connect to. The floating license server must be a valid uri. Default value is `localhost`
- **codeoptions.license.floating_license_port**: Selects the port of the machine which is running the floating license proxy for the solver to connect to. The floating license server must be a valid port number. Default value is `5000`
- **codeoptions.license.floating_license_file_name**: Sets the name for the local file created for the floating license. The floating license file name must be a valid variable name. Default value is the solvername
- **codeoptions.license.floating_license_retries**: Sets the number of retries to connect to the floating license server in case a connection fails. Default value is `3`

Note: If the FORCESPRO solver is run in a docker container and the Floating Licenses Proxy is run in the host of the same machine, in order for the solver to access the proxy **codeoptions.license.floating_license_server** should be set as **'host.docker.internal'**. In a Linux host, the docker container needs to provide access to the host for **host.docker.internal** by using the runtime option **–add-host=**. Depending on the docker version and the setup, the option value would be: **host.docker.internal:host-gateway** (i.e. **–add-host=host.docker.internal:host-gateway**), **host.docker.internal:172.17.0.1** or **host.docker.internal:<bridge-gateway-ip>** where **<bridge-gateway-ip>** is the IP of the gateway of the bridge network to the host. In a Dockerfile, the value for **host.docker.internal** can be added under the entry **extra_hosts:**.

16.4.7 Running solvers with floating licenses

After generating a solver, you can move it to the running platform and build it with the rest of your project.

When running a solver:

- The solver will communicate with the Floating Licenses Proxy
- The Proxy will perform the request on the online floating license server
- If the number of enabled machines has not exceeded the license limits, a license lease will be returned
- If a lease had already been granted for a machine (and is still valid) the license will be extended to match the set lease time and will be returned
- The Proxy will return the accepted license lease to the FORCESPRO solver
- The solver will save the lease locally and run
- If a valid local lease already exists the solver will run without communicating with the server

When the floating license check is enabled, the run-time parameter **receive_floating_license** is available. Setting it to `1` will cause the solver to run only the license check, acquire the floating license and return. This can be used to acquire the floating license early and avoid connecting to the floating license server in subsequent executions when running the solver initialized with the problem.

Chapter 17

Autotuner

- *Autotuner Options*
- *Collecting Tuning Data*
- *Validation*

The FORCESPRO autotuner is a tool for "tuning" a FORCESPRO solver in an automated way, in order to ensure both robustness and fast convergence on a specific application. Often this means choosing a set of so-called "hyper-parameters" which tailor the algorithm to a specific application.

Important: The FORCESPRO autotuner is only compatible with the SQP Fast algorithm (see *Different SQP variants*) and the QP Fast algorithm (see *The QP_FAST algorithm*).

Autotuning a FORCESPRO solver requires setting several tuning options. The tuning options object is constructed by calling `ForcesAutotuneOptions` which takes as an optional input a cell array of problems instances (here called `problems`) which are compatible with the generated solver. I.e. each element of the cell array is a struct containing the run-time parameters for a solver:

Matlab

Python

```
tuningoptions = ForcesAutotuneOptions(problems);
```

```
# not supported
```

See *Collecting Tuning Data* for an explanation of how to collect tuning data (`problems`) and see section *Autotuner Options* for which options can be specified through the `ForcesAutotuneOptions` object. Once the tuning options have been specified a *QP_FAST* solver (see *The QP_FAST algorithm*) can be generated by calling:

Matlab

Python

```
ForcesGenerateQpFastSolver(stages,params,codeoptions,tuningoptions,outputs);
```

```
# not supported
```

and a SQP solver using the *fast* QP solver (see *Different SQP variants*) can be generated as follows:

Matlab

Python

```
ForcesGenerateSqpFastSolver(model, codeoptions, tuningoptions, outputs);
```

```
# not supported
```

17.1 Autotuner Options

Via the `ForcesAutotuneOptions` object, the following options can be set:

- `TuningSeed`: Set equal to a non-negative integer to fix the seed in the random number generator used for the FORCESPRO autotuner.
- `TuningMinutes`: Number of minutes the tuning should last.
- `TuningIterations`: Number of iterations the autotuner should last (only relevant if `TuningMinutes = 0`). Must be a positive integer.
- `FailureTolerance`: Denotes the percentage of problems which are allowed to fail to converge for a tuning to be acceptable. E.g. if `FailureTolerance = 0.05` then 5% of problems are allowed to fail to converge. Normally, this value should be kept to 0.
- `ComparisonOutputs`: If non-empty, the autotuner will attempt to find a tuning for which the solver produces a solution which is as close to the ones specified here. I.e. the `ComparisonOutputs` field must be a cell-array of the same length as the cell-array `problems` passed in the constructor and each element must be of the same type and size as the output which the solver produces.
- `ComparisonObjectives`: If non-empty, the autotuner will attempt to find a tuning for which the objective value at the solution produced by the solver matches the one assigned here. In particular, if non-empty, `ComparisonObjectives` must be a cell-array of the same length as the cell-array `problems` passed in the constructor.

Several of these options can also be set via the following "setter" methods

- `setTuningGoal`: Choose what to tune the solver for. The input to this function must be `"speed"`, `"control"` or `"balanced"`.
- `setTuningSeed`: Set `TuningSeed`.
- `setTuningMinutes`: Set `TuningMinutes`.
- `setTuningIterations`: Set `TuningIterations`.

17.2 Collecting Tuning Data

The main step in tuning a solver is to collect the tuning data/problems which must be specified directly in the constructor to the `ForcesAutotuneOptions` class. The standard way to do this is by running one or more simulations with a solver which does not require tuning. In order to see the how to generate such a solver, consult section *The QP_FAST algorithm* for the QP Fast solver and *Tuning the SQP Fast solver* for the SQP Fast solver.

17.3 Validation

The FORCESPRO autotuning tool also allows the user to automatically validate the results the autotuner produces. The purpose of the validation is to check that the tuning found by the autotuner provides a solver which meets the requirements (be it speed or control performance) for a given application. The validation has a default configuration, but can also be configured by setting the following `ForcesAutotuneOptions` members:

- `ValidationMaxObjTol`: Specifies the maximum relative deviation in the optimal objective function value allowed to pass validation of the tuned solver. `0.1` means 10%. Must be a non-negative floating point number. The default value is `0.0`, which deactivates the check.
- `ValidationAvgObjTol`: Specifies the average relative deviation in the optimal objective function value allowed to pass validation of the tuned solver. `0.1` means 10%. Must be a non-negative floating point number. The default value is `0.1`.
- `ValidationMaxControlTol`: Specifies the maximum relative deviation in the first optimal control input (2-norm) allowed to pass validation of the tuned solver. `0.1` means 10%. Must be a non-negative floating point number. The default value is `0.0`, which deactivates the check.
- `ValidationAvgControlTol`: Specifies the average relative deviation in the first optimal control input (2-norm) allowed to pass validation of the tuned solver. `0.1` means 10%. Must be a non-negative floating point number. The default value is `0.1`, which deactivates the check.
- `ValidationControlOutputName`: Specifies the name of the output at the first stage used to validate the control input accuracy.
- `ValidationControlOutputIdx`: Specifies the indices of the variables corresponding to the control inputs at the first stage used to validate the control input accuracy.

Instead of setting these validations directly, one can configure them through the following methods:

- `setValidationControlOutput`: Specify the control inputs of the first stage. This method takes `ValidationControlOutputName` as the first input and `ValidationControlOutputIdx` as a second input.
- `disableObjValidation`: Completely disable validation of the objective function. This method takes no inputs.
- `disableControlValidation`: Completely disable the validation of the control performance. This method takes no inputs.
- `disableValidation`. Completely disable validation. This method takes no inputs.

Chapter 18

Solver Options

- *Default options*
- *General options*
  - *Solver name*
  - *Solve method*
  - *Print level*
  - *Maximum number of iterations*
  - *Parametric number of iterations*
  - *Compiler optimization level*
  - *Measure Computation time*
  - *Solver Timeout*
  - *Early-terminate solver*
  - *Options related to multicore parallelization*
  - *Datatypes*
  - *Code generation server*
  - *Enforcing solver regeneration*
  - *Overwriting existing solvers*
  - *Skipping automatic cleanup*
  - *Skipping the Build of Simulink S-function*
  - *Solver info in Simulink block*
  - *Identifying FORCESPRO solver*
  - *Target platform*
  - *Optimizing code size*
  - *Optimizing Linear Algebra Operations*
  - *MISRA 2012 compliance*
  - *MATLAB network communications*
  - *Python network communications*
  - *Dump problem formulation and data*

- *Low-level interface options*
  - *(Convex) Primal-Dual Interior-Point Method*
  - *Convex branch-and-bound solver*
  - *QP_FAST method*
  - *Optimizations for convex problems with short horizon*
  - *Alternating Directions Method of Multipliers*
  - *Dual Fast Gradient Method*
  - *Primal Fast Gradient Method*
- *High-level interface options*
  - *Integrators*
  - *Accuracy requirements*
  - *Barrier strategy*
  - *Hessian approximation*
  - *Line search settings*
  - *Regularization*
  - *Linear system solver*
  - *Automatic differentiation tool*
  - *Automatic differentiation expression class*
  - *Re-use of callback code*
  - *Safety checks*
  - *Reducing NLP code size*

FORCESPRO expects a set of solver options that help to customize the generated solver code. Section *Default options* describes how to obtain a set of default options, while further customizations are described in the remaining sections.

In the documentation below, we assume that you have created this struct and named it `codeoptions`.

Note: For the low-level interface in Python, the `codeoptions` struct has to be an element of the `stages` struct.

18.1 Default options

Default solver options can be loaded by giving a name to the solver with the following command

Matlab

Python

```
codeoptions = getOptions('solvername');
```

```
stages.codeoptions = forcespro.CodeOptions('solvername') # for the low-level interface
codeoptions = forcespro.CodeOptions('solvername') # for the high-level interface
```

This function is very generic and does not tailor the default options to the algorithm used. It is therefore recommended to use the following function instead (which has been introduce with release 6.0.0):

Matlab

Python

```
codeoptions = ForcesGetDefaultOptions( 'solvername','algorithm','floattype' );
```

```
stages.codeoptions = forcespro.CodeOptions() # for the low-level interface
stages.codeoptions.use_default('solvername', 'algorithm', 'floattype')
codeoptions = forcespro.CodeOptions() # for the high-level interface
codeoptions.use_default('solvername', 'algorithm', 'floattype')
```

In addition to the solver name, this function expects two more input arguments:

- a string specifying the `algorithm`, which can take any value listed in Table 18.1 or `'SQP_NLP_fast'` or `'QP_FAST'` to denote the SQP Fast (see *Different SQP variants*) or QP Fast algorithm (see *The QP_FAST algorithm*), respectively;
- a string `'floattype'`, which currently can be set to either `'double'` or `'float'` (see Table 18.6).

18.2 General options

We will first discuss how to change several options that are valid for all the FORCESPRO interfaces.

18.2.1 Solver name

The name of the solver will be used to name variables, functions, but also the MEX file and associated help file. This helps you to use multiple solvers generated by FORCESPRO within the same software project or Simulink model. To set the name of the solver use:

Matlab

Python

```
codeoptions.name = 'solvername';
```

```
codeoptions.name = 'solvername'
```

Alternatively, you can directly name the solver when generating the options struct by calling:

Matlab

Python

```
codeoptions = getOptions('solvername');
```

```
stages.codeoptions = forcespro.CodeOptions('solvername') # for the low-level interface
codeoptions = forcespro.CodeOptions('solvername') # for the high-level interface
```

18.2.2 Solve method

As a default optimization method the primal-dual interior-point method is used. Several other methods are available. To change the solve method set the `solvemethod` field as outlined in Table 18.1.

Table 18.1: Solve methods

`solvemethod`	Method	Description
`'PDIP_NLP'`	Nonlinear Primal-Dual Interior-Point method	The Nonlinear Primal-Dual Interior-Point method is the most stable and robust method for most nonlinear problems.
`'SQP_NLP'`	Sequential Quadratic Programming method	The Sequential Quadratic Programming method may be more efficient on mildly nonlinear problems.
`'PDIP'`	Primal-Dual Interior-Point method	The Primal-Dual Interior-Point method is the most stable and robust method for most convex problems.
`'QP_FAST'`	Fast QP method	The QP_FAST method for convex problems is typically faster (but possibly less stable) than the PDIP method and particularly well-suited for low-level hardware.
`'ADMM'`	Alternating Direction Methods of Multipliers	For some problems, ADMM may be faster. The method variant and several algorithm parameters can be tuned in order to improve performance.
`'DFG'`	Dual Fast Gradient method	For some problems with simple constraints, our implementation of the dual fast gradient method can be the fastest option. No parameters need to be tuned in this method.
`'FG'`	Fast Gradient method	For problems with no equality constraints (only one stage) and simple constraints, the primal fast gradient method can give medium accuracy solutions extremely quickly. The method has several tuning parameters that can significantly affect the performance.

18.2.3 Print level

To control the amount of information the generated solver prints to the console, set the field `printlevel` as outlined in Table 18.2.

Table 18.2: Print level options

`printlevel`	Result
`0`	No output will be written.
`1`	Summary line after each solve.
`2` (default)	Summary after each iteration of solver.

Important: `printlevel` should always be set to `0` when recording performance timings or when deploying the code on an autonomous embedded system.

18.2.4 Maximum number of iterations

To set the maximum number of iterations of the generated solver, use:

Matlab

Python

```
codeoptions.maxit = 200;
```

```
codeoptions.maxit = 200
```

The default maximum number of iterations for all solvers provided by FORCESPRO is set to `200`.

18.2.5 Parametric number of iterations

The maximum number of iterations of the generated solver can be made parametric with the following codeoption:

Matlab

Python

```
codeoptions.parametric_iterations = 1;
```

```
codeoptions.parametric_iterations = 1
```

This will generate the runtime parameter `maxit` which can be provided as input to the solver. The runtime parameter needs to be initialized with a value in the range `(0, codeoptions.maxit]` (i.e. the runtime iterations parameter needs to be set lower or equal to the hardcoded maximum number of iterations).

Note: The parametric number of iterations feature is currently not available for the `SQP` solvers.

18.2.6 Compiler optimization level

The compiler optimization level can be varied by changing the field `optlevel` from `0` to `3` (default):

Matlab

Python

```
codeoptions.optlevel = 0;
```

```
codeoptions.optlevel = 0
```

Important: It is recommended to set `optlevel` to `0` or `1` during prototyping to evaluate the functionality of the solver without long compilation times. Then set it back to `3` when generating code for deployment or timing measurements.

Note: In very rare cases, when the solver has a large amount of parameters or the problem is relatively large, compiling with *codeoptions.optlevel = 0;* has produced a stack overflow in the past. If you encounter such a situation, please try to increase the value of *codeoptions.optlevel* and submit a bug report to support@embotech.com.

18.2.7 Measure Computation time

You can measure the time used for executing the generated code by using:

Matlab

Python

```
codeoptions.timing = 1;
```

```
codeoptions.timing = 1
```

By default the execution time is measured. The execution time can be accessed in the field `solvetime` of the information structure returned by the solver. In addition, the execution time is printed in the console if the flag `printlevel` is greater than `0`.

Important: Setting timing on will introduce a dependency on libraries used for accessing the system clock. Timing should be turned off when deploying the code on an autonomous embedded system.

By default when choosing to generate solvers for target platforms, timing is disabled. You can manually enable timing on embedded platforms by using:

Matlab

Python

```
codeoptions.embedded_timing = 1;
```

```
codeoptions.embedded_timing = 1
```

18.2.8 Solver Timeout

Introduction

If you have a critical application which needs to run in a specific timeframe then it's useful to set a timeout for the solver in order to control its execution time.

The timeout works by checking the execution time of each iteration of the solver and making an estimate for next iterations as:

```
next_iteration_time = timeout_estimate_coeff * max_iteration_time
```

where:

- `max_iteration_time` is the execution time of the currently slowest iteration
- `timeout_estimate_coeff` is a coefficient used to make the estimate more conservative or forgiving. Its default value is `1.20`

Usage

To enable the solver timeout you can use the following codeoption:

Matlab

Python

```
% solver_timeout can take values 0-2
codeoptions.solver_timeout = 1;
```

```
# solver_timeout can take values 0-2
codeoptions.solver_timeout = 1
```

Setting the option to `1` will enable the timeout and provide the floating point variable `solver_timeout` as a runtime parameter. Setting the option to `2` will additionally provide the floating point variable `timeout_estimate_coeff` as a runtime parameter.

Important: For MINLP solvers a timeout is automatically enabled therefore there's no need to use the above codeoptions. For more details on how to use it please check section `Mixed-integer nonlinear solver`.

Not setting the runtime parameters after enabling them with code generation will result in them taking their default values. The default values for the runtime parameters are:

- For `solver_timeout` it's `-1.0` which results in timeout being disabled
- For `timeout_estimate_coeff` it's `1.20`

Important: Since an estimation is required for the timeout, the solvers will always perform the first iteration (only exception are SQP methods, check the following section `SQP inner QP timeout`).

SQP inner QP timeout

With the SQP_NLP solve method the QP solved as part of the SQP iteration is also set to timeout based on the remaining time available to the SQP solver. The QP timeout can be useful in cases where the inner QP takes longer time to execute than expected and could

otherwise cause the SQP solver to miss the timeout mark (in which case the SQP solver would time out at the start of the next iteration). If the QP times out, the SQP solver will return with the solution from the previous iteration.

If it is deemed more important to solve the whole QP and get a more updated solution rather than having a strict timeout, the inner qp timeout can be disabled with the following codeoption:

Matlab

Python

```
% this option is relevant only if codeoptions.solver_timeout is enabled
codeoptions.sqp_nlp.qp_timeout = 0;
```

```
# this option is relevant only if codeoptions.solver_timeout is enabled
codeoptions.sqp_nlp.qp_timeout = 0
```

Return Value

When solver timeout is enabled, two additional exitflags are available for the user:

Table 18.3: Timeout exitflags

Exitflag Name	Value	Description
TIMEOUT_<SOLVERNAME>	2	The solver timed out and returned the solution found up to the executed iteration
INVALID_TIMEOUT_<SOLVERNAME>	-12	The timeout provided was too small to even start a single iteration

If a normal timeout is returned, the outputs of the solver will contain the solution found up to the executed iteration. If an invalid timeout is returned, the outputs of the solver will contain the initialization of the solver (or the previous solution if it exists for SQPs).

18.2.9 Early-terminate solver

You can terminate a solver during execution from C based on an external event by setting the option

Matlab

Python

```
codeoptions.solver_exit_external = 1;
```

```
codeoptions.solver_exit_external = 1
```

Setting the option to `1` will enable the external termination by providing a runtime parameter `solver_exit_external` which can take 2 values:

Table 18.4: solver_exit_external parameter

Value of solver_exit_external	Description
0	solver runs normally
1	solver terminates at the end of the current iteration

When a solver is early-terminated, it always finishes the current iteration and returns its solution.

A solver that was early-terminated before convergence returns a dedicated exitflag:

Table 18.5: Early-termination exitflag

Exitflag Name	Value	Description
EXIT_EXTERNAL_<SOLVERNAME>	3	The solver was terminated because solver_exit_external = 1 was s

Note: To enable external and asynchronous update of the variable `solver_exit_external`, it is declared as `volatile` (ensuring visibility) and of type `sig_atomic_t` (ensuring atomicity). If you use this parameter within a multithreaded environment to terminate a solver on another thread, please note that this assumes a cache coherent architecture to work reliably as the `volatile` qualifier itself does not guarantee thread safety.

18.2.10 Options related to multicore parallelization

Internal parallelization of the solver using OpenMP can be switched on by using

Matlab

Python

```
codeoptions.parallel = 1;
```

```
codeoptions.parallel = 1
```

For how to control the number of threads, please refer to section *Internal parallelism*.

You can also parallelize over multiple solver calls (external parallelism). For that to work, ensure that the generated solver is thread safe by setting the option

Matlab

Python

```
codeoptions.threadSafeStorage = 1;
```

```
codeoptions.threadSafeStorage = 1
```

The use of external parallelism is described in more detail in section *External parallelism*.

18.2.11 Datatypes

The type of variables can be changed by setting the field `floattype` as outlined in Table 18.6. This will effect all floating point variables used inside the solver and the callbacks generated by the AD tool.

Table 18.6: Data type options

floattype	Decimation	Width (bits)	Supported algorithms
'double' (default)	64 bit	Floating point	PDIP_NLP, SQP_NLP, PDIP, QP_FAST, ADMM, DFG, FG
'float'	32 bit	Floating point	SQP_NLP, PDIP, QP_FAST, ADMM, DFG, FG
'int'	32 bit	Fixed point	ADMM, DFG, FG
'short'	16 bit	Fixed point	ADMM, DFG, FG

Important: Unless running on a resource-constrained platform, we recommend using double precision floating point arithmetic to avoid problems in the solver. If single precision floating point has to be used, reduce the required tolerances on the solver accordingly by a power of two (i.e. from `1E-6` to `1E-3`).

When running the solver in double precision arithmetic, it is possible to only use single precision arithmetic for evaluating the AD tool callbacks. This can be done by setting the field `callback_floattype`; see Table 18.7 and section *Single precision callbacks* for details.

Table 18.7: Callback data type options

`floattype`	Decimation	Width (bits)	Supported algorithms
`'double'` (default)	64 bit	Floating point	PDIP_NLP, SQP_NLP
`'float'`	32 bit	Floating point	PDIP_NLP, SQP_NLP

18.2.12 Code generation server

By default, code generation requests are routed to embotech's main FORCESPRO server (https://forces.embotech.com) which always provides the most up-to-date release of FORCESPRO. To send a code generation request to a custom server, for example when FORCESPRO is used in an enterprise setting, set the following field to an appropriate value:

Matlab

Python

```
codeoptions.server = 'https://yourforcesserver.com:1234';
```

```
codeoptions.server = 'https://yourforcesserver.com:1234'
```

18.2.13 Enforcing solver regeneration

In order to avoid unnecessary calls to the code-generation server, FORCESPRO internally computes a hash of your problem formulation and codeoptions. If this hash is identical to that of an already generated solver, the existing one is reused. In situations where this is not desired, hashing can be disabled as follows:

Matlab

Python

```
codeoptions.nohash = 1;
```

```
codeoptions.nohash = 1
```

In that case, the codegen server is always contacted to re-generate a new solver.

18.2.14 Overwriting existing solvers

When a new solver is generated with the same name as an existing solver one can control the overwriting behavior by setting the field `overwrite` as outlined in Table 18.8.

Table 18.8: Overwrite existing solver options

`overwrite`	Result
`0`	Never overwrite.
`1`	Always overwrite.
`2` (default)	Ask to overwrite.

18.2.15 Skipping automatic cleanup

FORCESPRO automatically cleans up some of the files that it generates during the code generation, but which are usually not needed any more after building the MEX file. In particular, some intermediate CasADi generated files are deleted. If you would like to prevent any cleanup by the FORCESPRO client, set the option:

Matlab

Python

```
codeoptions.cleanup = 0;
```

```
codeoptions.cleanup = 0
```

The default value is `1` (true).

Important: The library or object files generated by FORCESPRO contain only the solver itself. To retain the CasADi generated files for function evaluations, switch off automatic cleanup as shown above. This is needed if you want to use the solver within another software project, and need to link to it.

18.2.16 Skipping the Build of Simulink S-function

By default, after code generation, the Simulink block is compiled, which may take a very long time for large problems on Windows systems. If you will not use the Simulink block, or want to build it later yourself, you can disable automatic builds by using the following option:

Matlab

Python

```
codeoptions.BuildSimulinkBlock = 0;
```

```
# does not take effect in Python
```

18.2.17 Solver info in Simulink block

FORCESPRO always generates a Simulink block encapsulating the generated solver. You can add output ports to the Simulink block to obtain the solver exit flag and other solver information (number of iterations, solve time in seconds, value of the objective function) by setting:

Matlab

Python

```
codeoptions.showinfo = 1;
```

```
codeoptions.showinfo = 1
```

By default these ports are not present in the Simulink block.

18.2.18 Identifying FORCESPRO solver

In order to be able to identity and distinguish the different FORCESPRO solvers, each solver is assigned a unique GUID (32 hex characters) called `Solver Id`. The `Solver Id` is available for all solvers in the header file as a C comment in the format:

```
/* Solver Id: xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx */
```

This `Solver Id` is also available during runtime in the different FORCESPRO interfaces. It is returned from the solver as an unsigned integer array of length 8 each element of which represents 4 hex characters of the full GUID. These elements can be used to generate the GUID as:

```
solver_id[0] /* hex: a1a2a3a4 */
solver_id[1] /* hex: b1b2b3b4 */
solver_id[2] /* hex: c1c2c3c4 */
solver_id[3] /* hex: d1d2d3d4 */
solver_id[4] /* hex: e1e2e3e4 */
solver_id[5] /* hex: g1g2g3g4 */
solver_id[6] /* hex: h1h2h3h4 */
solver_id[7] /* hex: i1i2i3i4 */

/* Solver Id: a1a2a3a4 b1b2b3b4 - c1c2c3c4 - d1d2d3d4 - e1e2e3e4 - g1g2g3g4 h1h2h3h4␣
↪i1i2i3i4 */
```

The `Solver Id` is available during runtime through the info struct when using the following interfaces: * C Interface * MATLAB MEX Interface * Python Interface * MPC Toolbox Plugin MATLAB Interface (with the exception of the dense QPs)

The `Solver Id` is available during runtime as a Simulink Block output port when using the following interfaces: * Simulink Interface (when solver is generated with `codeoptions.showinfo = 1`) * MPC Toolbox Plugin Simulink Interface (the output port for the `Solver Id` can be enabled via the Block Parameters of the MPC Toolbox Plugin FORCESPRO Simulink Block – with the exception of the dense QPs)

18.2.19 Target platform

As a default option, FORCESPRO generates code for simulation on the host platform. To obtain code for deployment on a target embedded platform, set the field `platform` to the appropriate value. The platforms currently supported by FORCESPRO are given in Table 18.9. In order to acquire licenses to use a specific `platform`, licenses can be requested on the portal by selecting the platform naming stated in the `Portal Selection`.

Table 18.9: Target platforms supported by FORCESPRO

`platform`	Description	Portal Selection
`'Generic'` (default)	For the architecture of the host platform.	`'x86_64'` (Engineering Node)
`'x86_64'`	For x86_64 based 64-bit platforms (detected OS).	`'x86_64'`
`'x86'`	For x86 based 32-bit platforms (detected OS).	`'x86'`

continues on next page

Table 18.9 – continued from previous page

platform	Description	Portal Selection
'Win-x86_64'	For Windows x86_64 based 64-bit platforms (supports Microsoft/Intel compiler).	'x86_64'
'Win-x86'	For Windows x86 based 32-bit platforms (supports Microsoft/Intel compiler).	'x86'
'Win-MinGW-x86_64'	For Windows x86_64 based 64-bit platforms (supports MinGW compiler).	'x86_64'
'Win-MinGW-x86'	For Windows x86 based 32-bit platforms (supports MinGW compiler).	'x86'
'Mac-x86_64'	For Mac x86_64 based 64-bit platforms (supports GCC/Clang compiler).	'x86_64'
'Gnu-x86_64'	For Linux x86_64 based 64-bit platforms (supports GCC compiler).	'x86_64'
'Gnu-x86'	For Linux x86 based 32-bit platforms (supports GCC compiler).	'x86'
'Docker-Gnu-x86_64'	For Linux x86_64 based 64-bit platforms on Docker (supports GCC compiler).	'Docker-Gnu-x86_64'
'Docker-Gnu-x86'	For Linux x86 based 32-bit platforms on Docker (supports GCC compiler).	'Docker-Gnu-x86'
'ARM-Generic'	For ARM Cortex 32-bit processors (Gnueabih machine type).	'ARM-Generic-Gnu'
'ARM-Generic64'	For ARM Cortex 64-bit processors (Aarch machine type).	'ARM-Generic64-Gnu'
'Integrity-ARM32'	For ARM Cortex 32-bit processors using the Integrity toolchain.	'Integrity-ARM32'
'Integrity-ARM64'	For ARM Cortex 64-bit processors using the Integrity toolchain.	'Integrity-ARM64'
'ARM Cortex-M3'	For ARM Cortex M3 32-bit processors.	'ARM-Cortex-M3'
'ARM-Cortex-M4-NOFPU'	For the ARM Cortex M4 32-bit processors without a floating-point unit.	'ARM-Cortex-M4'
'ARM-Cortex-M4'	For the ARM Cortex M4 32-bit processors with a floating-point unit.	'ARM-Cortex-M4'
'ARM-Cortex-A7'	For the ARM Cortex A7 32-bit processors (Gnueabih machine type).	'ARM-Cortex-A7'
'ARM-Cortex-A8'	For the ARM Cortex A8 32-bit processors (Gnueabih machine type).	'ARM-Cortex-A8'

continues on next page

Table 18.9 – continued from previous page

platform	Description	Portal Selection
'ARM-Cortex-A9'	For the ARM Cortex A9 32-bit processors (Gnueabih machine type).	'ARM-Cortex-A9'
'ARM-Cortex-A15'	For the ARM Cortex A15 32-bit processors (Gnueabih machine type).	'ARM-Cortex-A15'
'ARM-Cortex-A53'	For the ARM Cortex A53 64-bit processors (Gnueabih machine type).	'ARM-Cortex-A53'
'ARM-Cortex-A72'	For the ARM Cortex A72 64-bit processors (Gnueabih machine type).	'ARM-Cortex-A72'
'TI-Cortex-A15'	For the ARM Cortex A15 32-bit processors (Gnueabih machine type).	'TI-Cortex-A15'
'NVIDIA-Cortex-A57'	For the NVIDIA Cortex A57 64-bit processors (Aarch machine type).	'NVIDIA-Cortex-A57'
'AARCH-Cortex-A53'	For the ARM Cortex A53 64-bit processors (Aarch machine type).	'AARCH-Cortex-A53'
'AARCH-Cortex-A57'	For the ARM Cortex A57 64-bit processors (Aarch machine type).	'AARCH-Cortex-A57'
'AARCH-Cortex-A72'	For the ARM Cortex A72 64-bit processors (Aarch machine type).	'AARCH-Cortex-A72'
'PowerPC'	For 32-bit PowerPC based platforms (supports GCC compiler).	'PowerPC-Gnu'
'PowerPC64'	For 64-bit PowerPC based platforms (supports GCC compiler).	'PowerPC64-Gnu'
'MinGW32'	For Windows x86 based 32-bit platforms (supports MinGW compiler).	'x86'
'MinGW64'	For Windows x86_64 based 64-bit platforms (supports MinGW compiler).	'x86_64'
'dSPACE-MABIII'	For the dSPACE MicroAutoBox III real-time system (supports Gcc compiler).	'dSPACE-MABIII-Gcc'
'dSPACE-MABXIII'	For the dSPACE MicroAutoBox III real-time system (supports Gcc compiler).	'dSPACE-MABIII-Gcc'
'dSPACE-AutoBox'	For the dSPACE AutoBox real-time system (supports Gcc compiler).	'dSPACE-AutoBox-Gcc'
'dSPACE-MicroLabBox'	For the dSPACE MicroLabBox real-time system (supports Gcc compiler).	'dSPACE-MicroLabBox-Gcc'
'dSPACE-SCALEXIO'	For the dSPACE SCALEXIO real-time system (supports Gcc compiler).	'dSPACE-SCALEXIO-Gcc'

continues on next page

Table 18.9 – continued from previous page

platform	Description	Portal Selection
'Speedgoat-x86'	For Speedgoat 32-bit real-time platforms (supports Microsoft compiler and mainly MATLAB Releases 2018b up to R2020a).	'Speedgoat-x86'
'Speedgoat-x64'	For Speedgoat 64-bit real-time platforms (supports Microsoft compiler and mainly MATLAB Releases 2018b up to R2020a).	'Speedgoat-x64'
'Speedgoat-QNX'	For Speedgoat 64-bit real-time platforms (supports MATLAB Releases 2020b onwards).	'Speedgoat-QNX'
'Speedgoat-Legacy-x86'	For Speedgoat Mobile 32-bit real-time platforms (supports Microsoft compiler and MATLAB Releases 2018a and earlier).	'Speedgoat-x86'
'NI-cRIO'	For National Instruments compactRIO Linux RTOS platforms (supports NILRT Gcc compiler).	'NI-cRIO'
'AURIX'	For Infineon AURIX platforms.	on special request
'IAtomE680_Bachmann'	For Bachmann PLC platforms (supports VxWorks compiler).	'IAtomE680-VxWorks'

Note: If a solver for another platform is requested, FORCESPRO will still provide the simulation interfaces for the `'Generic'` host platform to enable users to run simulations.

Cross compilation

The host OS for the client is automatically detected. To generate code for other operating systems different from the host platform, the **host_os** codeoption can be set. The OS selection for the interfaces is set by default based on the host OS. To select a different OS for the interfaces, the **interface_os** codeoption can be set:

Matlab

Python

```
% available options: 'win', 'win-mingw', 'mac', 'gnu'
codeoptions.host_os = 'win';
codeoptions.interface_os = 'gnu';
```

```
# available options: 'win', 'win-mingw', 'mac', 'gnu'
codeoptions.host_os = 'win'
codeoptions.interface_os = 'gnu'
```

Mac compilation

When compiling for mac platforms it's possible to select the compiler to be used for the web compilation. Select from the available values `gcc` (default) and `clang` with the following codeoption:

Matlab

Python

```
codeoptions.maccompiler = 'gcc'; % or 'clang'
```

```
codeoptions.maccompiler = 'gcc' # or 'clang'
```

SIMD instructions

On x86-based host platforms, one can enable the `sse` field to accelerate the execution of the solver

Matlab

Python

```
codeoptions.sse = 1;
```

```
codeoptions.sse = 1
```

On x86-based host platforms, one can also add the `avx` field to significantly accelerate the compilation and execution of the convex solver, from version `1.9.0`,

Matlab

Python

```
codeoptions.avx = 1;
```

```
codeoptions.avx = 1
```

Note: Currently when options `avx` and `blckMatrices` are enabled simultaneously, `blckMatrices` is automatically disabled.

Note: When sparse parameters are present in the model, the options `avx` and `neon` are automatically set to zero.

Depending on the host platform, `avx` may be automatically enabled. If the machine on which the solver is to be run does not support AVX and the message "Illegal Instruction" is returned at run-time, one can explicitly disable `avx` by setting:

Matlab

Python

```
codeoptions.avx = -1;
```

```
codeoptions.avx = -1
```

If the host platform supports AVX, but the user prefers not to have AVX intrinsics in the generated code, one can also keep the default option value of the solver:

Matlab

Python

```
codeoptions.avx = 0;
```

```
codeoptions.avx = 0
```

On 'NVIDIA-Cortex-A57', 'AARCH-Cortex-A53', 'AARCH-Cortex-A57' and 'AARCH-Cortex-A72' target platforms, one can also enable the field `neon` in order to accelerate the execution of the convex solver. From version `1.9.0`, the typical behavior is that the host platform gets vectorized code based on AVX intrinsics when `avx = 1`, and the target platform gets AVX vectorized code if it supports it when `avx = 1` and NEON vectorized code if it is one of the above Cortex platforms and `neon = 1`.

For single precision, the options are

Matlab

Python

```
codeoptions.floattype = 'float';
codeoptions.neon = 1;
```

```
codeoptions.floattype = 'float'
codeoptions.neon = 1
```

For double precision, the options are

Matlab

Python

```
codeoptions.floattype = 'double';
codeoptions.neon = 2;
```

```
codeoptions.floattype = 'double'
codeoptions.neon = 2
```

In case one wants to disable NEON intrinsics in the generated target code, the default value of the `neon` option is

Matlab

Python

```
codeoptions.neon = 0;
```

```
codeoptions.neon = 0
```

If NEON vectorization is being used and there is a mismatch between float precision and the value of the neon option, the solver is automatically generated with the following options:

Matlab

Python

```
codeoptions.floattype = 'double';
codeoptions.neon = 2;
```

```
codeoptions.floattype = 'double'
codeoptions.neon = 2
```

and a warning message is raised by the MATLAB client.

Note: From version `1.9.0`, ARMv8-A NEON instructions are generated. Hence, target platforms based on ARMv7 and previous versions are currently not supported.

C Standard

FORCESPRO generates solver code in C, which is intended to adhere to the C99 standard. If you are using a compiler that is meant to compile C90 code and fails to compile your FORCESPRO solver, try enabling the following option:

Matlab

Python

```
codeoptions.c90 = 1;
```

```
codeoptions.c90 = 1
```

This does *not* guarantee the solver code to stick to the C90 standard, but enables a few adjustments that may render compilation with those legacy compilers possible.

18.2.20 Optimizing code size

If your problem formulation contains sparse matrices, the size of sparse linear algebra routines in the generated code can be reduced by changing the option `compactSparse` from `0` to `1`:

Matlab

Python

```
codeoptions.compactSparse = 1;
```

```
codeoptions.compactSparse = 1
```

When using `codeoptions.solvemethod = 'PDIP_NLP'`, an additional reduction in code size is available through the option `nlp.compact_code` as described in Section *Reducing NLP code size*.

18.2.21 Optimizing Linear Algebra Operations

Some linear algebra routines in the generated code have available optimizations that can be enabled by changing the options `optimize_<optimization>` from `0` to `1`. These optimizations change the code in order to make better use of some embedded architectures in which hardware is more limited compared to host PC architectures. Therefore, these optimizations show better results in embedded platforms such as ARM targets rather than during simulations on host PCs. The available optimizations are:

- **Cholesky Division:** This option performs the divisions included in the Cholesky factorization more efficiently to reduce its computation time.

- **Registers:** This option attempts to use the architecture's registers in order to reduce memory operations which can take significant time.
- **Use Locals:** These options (which are separated into `simple`/`heavy`/`all` in ascending complexity) make better use of data locality in order to reduce memory jumps
- **Operations Rearrange:** This option rearranges operations in order to make more efficient use of data and reduce memory jumps
- **Loop Unrolling:** This option unrolls some of the included loops in order to remove their overhead.
- **Enable Offset:** This option allows the rest of the optimizations to take place in cases where the matrix contains offsets.

Matlab

Python

```
codeoptions.optimize_choleskydivision = 1;
codeoptions.optimize_registers = 1;
codeoptions.optimize_uselocalsall = 1;
codeoptions.optimize_uselocalsheavy = 1; % overridden if uselocalsall is enabled
codeoptions.optimize_uselocalssimple = 1; % overridden if uselocalsheavy is enabled
codeoptions.optimize_operationsrearrange = 1;
codeoptions.optimize_loopunrolling = 1;
codeoptions.optimize_enableoffset = 1;
```

```
codeoptions.optimize_choleskydivision = 1
codeoptions.optimize_registers = 1
codeoptions.optimize_uselocalsall = 1
codeoptions.optimize_uselocalsheavy = 1 # overridden if uselocalsall is enabled
codeoptions.optimize_uselocalssimple = 1 # overridden if uselocalsheavy is enabled
codeoptions.optimize_operationsrearrange = 1
codeoptions.optimize_loopunrolling = 1
codeoptions.optimize_enableoffset = 1
```

18.2.22 MISRA 2012 compliance

If your license allows it, add the following field to generate C code that is compliant with (a suitable subset of) the MISRA 2012 rules:

Matlab

Python

```
codeoptions.misra2012_check = 1;
```

```
codeoptions.misra2012_check = 1
```

This option makes the generated solver code MISRA compliant. After compilation, the client also downloads a folder whose name ends with `_misra2012_analysis`. The folder contains a summary of any remaining MISRA violations for source and header files of the core solver (i.e. without callback functions).

18.2.23 MATLAB network communications

From version `5.0.0`, the MATLAB client will perform connections to a REST interface for communicating with the FORCESPRO codegen server.

The selection of the server connection can be performed with the below option:

```
% Rest API connection
codeoptions.server_connection = ForcesWeb.ServerConnections.RestAPI;
% WSDL legacy connection
codeoptions.server_connection = ForcesWeb.ServerConnections.WSDL_legacy;
```

or change it by editing the FORCESPRO client. To do so, please edit the `+ForcesWeb/defaultServerConnection.m` function so that it returns the selected `ForcesWeb.ServerConnections` value.

Important: Setting the `codeoptions.server_connection` option will override the value in `+ForcesWeb/defaultServerConnection.m`

The connections are performed over HTTPS. For the server verification, a check via SSL certificates is performed. To perform this verification, a client certificate is needed on the client side. This certificate, if missing, can be acquired by accessing the FORCESPRO server via a browser (the browser will have the option to download the client certificate – the certificate with the entire chain of certificates should be selected). MATLAB stores the client certificates to be used for this verification in an installation specific file. In case this file cannot be accessed/changed due to admin rights, it is possible to manually select the files that will be checked for the certificates for the FORCESPRO connections. This can be done by editing the file: `+ForcesWeb/SSLCertificateFiles.m` in the FORCESPRO client. This file contains the entries:

```
% path to certificates file that contains client certificate to authenticate for␣
↪codegen server. To use default set to empty
codegen = '';
% path to certificates file that contains client certificate to authenticate for␣
↪storage server. To use default set to empty
storage = '';
```

The user can download the certificates with the entire chain from the codegen server (by default https://forces.embotech.com but also depends on the server the user has selected) and from the storage server (https://forcesblob.embotech.com) and then assign the path to those certificate files to the file above. This way, MATLAB will use those files for authentication instead of the default file.

Note: In most systems, the default certificate file contains the most common certificates. Therefore, for most users these certificate changes are not required. The above file is provided for cases of strict IT configurations in which certificates need to be specifically enabled for a successful verification.

18.2.24 Python network communications

From version `5.0.0`, the Python client will perform connections to a REST interface for communicating with the FORCESPRO codegen server.

From version `4.3.1`, the Python client supports connections to the FORCESPRO codegen server through a proxy.

The file `forcespro_proxy.py` exists in the FORCESPRO client folder in order to set the configuration for the proxy. The format of the file is as follows:

```
# host of the proxy. Can be an IP address ("x.x.x.x") or a DNS record. Set to empty␣
↪to not use a proxy
host=""
# port number of proxy to connect to. To use default set to 0
port=8888
# Protocol to connect to the proxy (http or https). To use default set to empty
protocol="http"
# Username with which to connect to the proxy. To not use a username set to empty
username="user"
# Password with which to connect to the proxy. To not use a password set to empty
password="pass"
```

Note: By default the file `forcespro_proxy.py` has an empty host entry so that no proxy is used unless set.

The connections are performed over HTTPS. For the server verification, a check via SSL certificates is performed. To perform this verification, a client certificate is needed on the client side. This certificate, if missing, can be acquired by accessing the FORCESPRO server via a browser (the browser will have the option to download the client certificate – the certificate with the entire chain of certificates should be selected). Python reads a system specific file to select the client certificates to be used for this verification. In case this file cannot be accessed/changed due to admin rights, it is possible to manually select the files that will be checked for the certificates for the FORCESPRO connections. This can be done by editing the file: `forcespro_certificates.py` in the FORCESPRO client. This file contains the entries:

```
# path to certificates file that contains client certificate to authenticate for␣
↪codegen server (path separator on Windows is "\\"). To use default set to empty
codegen=""
# path to certificates file that contains client certificate to authenticate for␣
↪storage (path separator on Windows is "\\"). To use default set to empty
storage=""
```

The user can download the certificates with the entire chain from the codegen server (by default https://forces.embotech.com but also depends on the server the user has selected) and from the storage server (https://forcesblob.embotech.com) and then assign the path to those certificate files to the file above. This way, Python will use those files for authentication instead of the default file.

Note: In most systems, the default certificate file contains the most common certificates. Therefore, for most users these certificate changes are not required. The above file is provided for cases of strict IT configurations in which certificates need to be specifically enabled for a successful verification.

18.2.25 Dump problem formulation and data

The FORCESPRO client provides a built-in tool to dump the problem formulation to reproduce the exact same solver for future reference. This tool is explained in detail in Section 21. In case you want to use the so-called *Legacy dumps* (which are only supported by the MATLAB client), you can turn on those dumps the setting:

Matlab

Python

```
codeoptions.dump_formulation = 1;
```

```
# not supported
```

Furthermore, you can dump problem structs containing the runtime parameters from C as described in Section 21. This tool is enabled for the host or/and the target platform by setting:

Matlab

Python

```
codeoptions.serializeCParamsHost = 1;
codeoptions.serializeCParamsTarget = 1;
```

```
codeoptions.serializeCParamsHost = 1
codeoptions.serializeCParamsTarget = 1
```

Additionally, you can ask FORCESPRO to generate a MEX function for performing the above-mentioned serialization of runtime parameters by setting:

Matlab

Python

```
codeoptions.serializeCParamsMex = 1;
```

```
# not yet supported
```

18.3 Low-level interface options

This sections describes a number of additional codeoptions that only take effect when using specific solve methods, but are ignored or not available for any other method.

18.3.1 (Convex) Primal-Dual Interior-Point Method

The Primal-Dual Interior-Point method is the default optimization method for both convex and nonlinear/nonconvex problems. It is a stable and robust method for most of the problems. Unless stated otherwise, the options described in this section are only available with solving convex problems, i.e. when using `codeoptions.solvemethod = 'PDIP'`.

Solver Initialization

The performance of the solver can be influenced by the way the variables are initialized. The default method (cold start) should work in most cases extremely reliably, so there should be no need in general to try other methods, unless you are experiencing problems with the default initialization scheme. To change the method of initialization in FORCESPRO set the field `init` to one of the values in Table 18.10.

Table 18.10: PDIP solver initialization

`init`	Method	Initialization method
`0` (default)	Cold start	Set all primal variables to 0, and all dual variables to the square root of the initial complementarity gap μ_0 : $z_i = 0, s_i = \sqrt{\mu_0}, \lambda_i = \sqrt{\mu_0}$. The default value is $\mu_0 = 10^6$.
`1`	Centered start	Set all primal variables to zero, the slacks to the RHS of the corresponding inequality, and the Lagrange multipliers associated with the inequalities such that the pairwise product between slacks and multipliers is equal to the parameter μ_0 : $z_i = 0, s_i = b_{\text{ineq}}$ and $s_i\lambda_i = \mu_0$.
`2`	Primal warm start	Set all primal variables as provided by the user. Calculate the residuals and set the slacks to the residuals if they are sufficiently positive (larger than 10^{-4}), or to one otherwise. Compute the associated Lagrange multipliers such that $s_i\lambda_i = \mu_0$.

Initial Complementary Slackness

The default value for μ_0 is 10^6. To use a different value, use:

Matlab

Python

```
codeoptions.mu0 = 10;
```

```
codeoptions.mu0 = 10;
```

Accuracy Requirements

The accuracy required for convergence (exitflag `1`) can be set as follows:

Matlab

Python

```
codeoptions.accuracy.ineq = 1e-6;  % infinity norm of residual for inequalities
codeoptions.accuracy.eq = 1e-6;    % infinity norm of residual for equalities
codeoptions.accuracy.mu = 1e-6;    % absolute duality gap
codeoptions.accuracy.rdgap = 1e-4; % relative duality gap := (pobj-dobj)/pobj
```

```
codeoptions.accuracy.ineq = 1e-6  # infinity norm of residual for inequalities
codeoptions.accuracy.eq = 1e-6    # infinity norm of residual for equalities
codeoptions.accuracy.mu = 1e-6    # absolute duality gap
codeoptions.accuracy.rdgap = 1e-4 # relative duality gap := (pobj-dobj)/pobj
```

Line Search Settings

If FORCESPRO experiences convergence difficulties, you can try selecting different line search parameters. The first two parameters of `codeoptions.linesearch`, `factor_aff` and `factor_cc` are the backtracking factors for the line search (if the current step length is infeasible, then it is reduced by multiplication with these factors) for the affine and combined search direction, respectively.

Matlab

Python

```
codeoptions.linesearch.factor_aff = 0.9;
codeoptions.linesearch.factor_cc = 0.95;
```

```
codeoptions.linesearch.factor_aff = 0.9
codeoptions.linesearch.factor_cc = 0.95
```

The remaining two parameters of the field `linesearch` determine the minimum (`minstep`) and maximum step size (`maxstep`). Choosing `minstep` too high will cause the generated solver to quit with an exitcode saying that the line search has failed, i.e. no progress could be made along the computed search direction. Choosing `maxstep` too close to 1 is likely to cause numerical issues, but choosing it too conservatively (too low) is likely to increase the number of iterations needed to solve a problem.

Matlab

Python

```
codeoptions.linesearch.minstep = 1e-8;
codeoptions.linesearch.maxstep = 0.995;
```

```
codeoptions.linesearch.minstep = 1e-8
codeoptions.linesearch.maxstep = 0.995
```

Dynamic Regularization

During factorization of supposedly positive definite matrices, FORCESPRO applies a regularization to the i-th pivot element if it is smaller than ϵ. In this case, it is set to δ, which is the lower bound on the pivot element that FORCESPRO allows to occur. The default values for this dynamic regularization approach can be adapted as follows:

Matlab

Python

```
codeoptions.regularize.epsilon = 1e-13; % if pivot element < epsilon ...
codeoptions.regularize.delta = 1e-8;    % then set it to delta
```

```
codeoptions.regularize.epsilon = 1e-13 # if pivot element < epsilon ...
codeoptions.regularize.delta = 1e-8    # then set it to delta
```

Dynamic regularization is also applied (and thus can be adjusted with the settings above) when using `codeoptions.solvemethod = 'PDIP_NLP'`.

Suggestions for solving QPs in single precision

Solving QPs in single precision can be rather challenging, i.e. non-converging solves are likely to occur due to the lack of accuracy. In order to mitigate this undesirable behavior, several options can be tuned to make convergence more robust. They are shown and commented in the code snippet below.

Matlab

Python

```
codeoptions.floattype = 'float';

codeoptions.regularize.epsilon = 1e-5;  % Tolerance on pivot in factorization
codeoptions.regularize.delta = 5e-3;    % On-the-fly regularization coefficient in␣
↪factorization
codeoptions.regularize.epsilon2 = 1e-5; % Tolerance on pivot in factorization
codeoptions.regularize.delta2 = 5e-3;   % On-the-fly regularization coefficient in␣
↪factorization

codeoptions.accuracy.ineq = 1e-4;       % infinity norm of residual for inequalities
codeoptions.accuracy.eq = 1e-4;         % infinity norm of residual for equalities
codeoptions.accuracy.mu = 1e-6;         % absolute duality gap
codeoptions.accuracy.rdgap = 1e-4;      % relative duality gap := (pobj-dobj)/pobj

codeoptions.init = 1;
```

```
codeoptions.floattype = 'float'

codeoptions.regularize.epsilon = 1e-5  # Tolerance on pivot in factorization
codeoptions.regularize.delta = 5e-3    # On-the-fly regularization coefficient in␣
↪factorization
codeoptions.regularize.epsilon2 = 1e-5 # Tolerance on pivot in factorization
codeoptions.regularize.delta2 = 5e-3   # On-the-fly regularization coefficient in␣
↪factorization

codeoptions.accuracy.ineq = 1e-4       # infinity norm of residual for inequalities
codeoptions.accuracy.eq = 1e-4         # infinity norm of residual for equalities
codeoptions.accuracy.mu = 1e-6         # absolute duality gap
codeoptions.accuracy.rdgap = 1e-4      # relative duality gap := (pobj-dobj)/pobj

codeoptions.init = 1;
```

In general, the rationale behind this tuning is to make the tolerances looser and increase the regularization in the linear algebra. Note that these suggestions are only applicable to QP solvers. Solving NLPs in single precision is even more challenging and we currently do not offer a set of options to robustify convergence on this type of problems.

18.3.2 Convex branch-and-bound solver

The settings of the FORCESPRO mixed-integer branch-and-bound convex solver are accessed through the `codeoptions.mip` struct. It is worthwhile to explore different values for the settings in Table 18.11, as they might have a severe impact on the performance of the branch-and-bound procedure.

Note: All the options described below are currently not available with the FORCESPRO nonlinear solver. For mixed-integer nonlinear programs and the available options, please refer to Section *Mixed-integer nonlinear solver*.

Table 18.11: Branch-and-bound options

Setting	Values	Default
`mip.timeout`	Any value ≥ 0	`31536000` (1 year)
`mip.mipgap`	Any value ≥ 0	`0`
`mip.branchon`	`'mostAmbiguous'`, `'leastAmbiguous'`	`'mostAmbiguous'`
`mip.stageinorder`	`0` (OFF), `1` (ON)	`1` (ON)
`mip.explore`	`'bestFirst'`, `'depthFirst'`	`'bestFirst'`
`mip.inttol`	Any value > 0	`1E-5`
`mip.queuesize`	Any integer value ≥ 0	`1000`

A description of each setting is given below:

- `mip.timeout`: Timeout in seconds, after which the search is stopped and the best solution found so far is returned.
- `mip.mipgap`: Relative sub-optimality after which the search shall be terminated. For example, a value of `0.01` will search for a feasible solution that is at most 1%-suboptimal. Set to zero if the optimal solution is required.
- `mip.branchon`: Determines which variable to branch on after having solved the relaxed problem. Options are `'mostAmbiguous'` (i.e. the variable closest to 0.5) or `'leastAmbiguous'` (i.e. the variable closest to 0 or 1).
- `mip.stageinorder`: Stage-in-order heuristic: For the branching, determines whether to fix variables in order of the stage number, i.e. first all variables of stage i will be fixed before fixing any of the variables of stage $i+1$. This is often helpful in multistage problems, where a timeout is expected to occur, and where it is important to fix the early stages first (for example MPC problems). Options are `0` for OFF and `1` for ON.
- `mip.explore`: Determines the exploration strategy when selecting pending nodes. Options are `'bestFirst'`, which chooses the node with the lowest lower bound from all pending nodes, or `'depthFirst'`, which prioritizes nodes with the most number of fixed binaries first to quickly reach a node.
- `mip.inttol`: Integer tolerance for identifying binary solutions of relaxed problems. A solution of a relaxed problem with variable values that are below `inttol` away from binary will be declared to be binary.
- `mip.queuesize`: Maximum number of pending nodes that the branch and bound solver can store. If that number is exceeded during the search, the solver quits with an `exitflag` value of `-2` and returns the best solution found so far.

18.3.3 QP_FAST method

When using `codeoptions.solvemethod = 'QP_FAST'` as described in Section *The QP_FAST algorithm*, additional options specific to the QP_FAST algorithm are specified via the `codeoptions.qp_fast` field (e.g. `codeoptions.qp_fast.warmstart = 0;`). The following options can be set:

- `warmstart`: Set equal to `1` (default) in order to allow run-time warmstart. Set equal to `0` to not allow run-time warmstart. The run-time warmstart option creates a run-time parameter (`problem.warmstart`) which if set to `1` will use the solution of the previous call to the solver as initial guess. If set to `0` the solver will use a default initial guess.
- `tol_primal`: Set the tolerance for the primal residual (default is `1e-3`).
- `tol_dual`: Set the tolerance for the dual residual (default is `1e-3`).

18.3.4 Optimizations for convex problems with short horizon

For small convex problems with a short control horizon and/or a small number of controls, the sparse FORCESPRO multistage formulation might be disadvantageous compared to a standard QCQP form. For that situation, FORCESPRO offers the option

Matlab

Python

```
codeoptions.condense = 1;
```

```
codeoptions.condense = 1
```

which automatically maps the sparse problem to a dense problem with all dynamic states eliminated. Please consider the requirements described in *Condensing (automatic state elimination)* for being able to use this option with your formulation.

This option is available for either the PDIP or the QP_FAST algorithm.

18.3.5 Alternating Directions Method of Multipliers

FORCESPRO implements several optimization methods based on the ADMM framework. Different variants can handle different types of constraints and FORCESPRO will automatically choose an ADMM variant that can handle the constraints in a given problem. To manually choose a specific method in FORCESPRO, use the `ADMMvariant` field of `codeoptions`:

Matlab

Python

```
codeoptions.ADMMvariant = 1; % can be 1 or 2
```

```
codeoptions.ADMMvariant = 1 # can be 1 or 2
```

where variant `1` is as follows:

$$\begin{aligned} \text{minimize} \quad & \frac{1}{2} y^\top H y + f^\top y \\ \text{subject to} \quad & Dy = c \\ & \underline{z} \leq z \leq \bar{z} \\ & y = z \end{aligned}$$

and variant `2` is as follows:

$$\begin{aligned} \text{minimize} \quad & \frac{1}{2} y^\top H y + f^\top y \\ \text{subject to} \quad & Dy = c \\ & Ay = z \\ & z \leq b \end{aligned}$$

Accuracy requirements

The accuracy required for convergence (exitflag `1`) can be set as follows:

Matlab

Python

```
codeoptions.accuracy.consensus = 1e-3;  % infinity norm of the consensus equality
codeoptions.accuracy.dres = 1e-3;    % infinity norm of the dual residual
```

```
codeoptions.accuracy.consensus = 1e-3  # infinity norm of the consensus equality
codeoptions.accuracy.dres = 1e-3    # infinity norm of the dual residual
```

Note that, in contrast to primal-dual interior-point methods, the required number of ADMM iterations varies very significantly depending on the requested accuracy. ADMM typically requires few iterations to compute medium accuracy solutions, but many more iterations to achieve the same accuracy as interior-point methods. For feedback applications, medium accuracy solutions are typically sufficient. Also note that the ADMM accuracy requirements have to be changed depending on the problem scaling.

Method parameters

ADMM uses a regularization parameter ρ, which also acts as the step size in the gradient step. The convergence speed of ADMM is highly variable in the parameter ρ. Its value should satisfy $\rho > 0$. This parameter can be tuned using the following command:

Matlab

Python

```
codeoptions.ADMMrho = 1;
```

```
codeoptions.ADMMrho = 1
```

In some cases it may be possible to let FORCESPRO choose the value ρ automatically. To enable this feature set:

Matlab

Python

```
codeoptions.ADMMautorho = 1;
```

```
codeoptions.ADMMautorho = 1
```

Please note that this does not guarantee that the choice of ρ will be optimal.

ADMM can also include an 'over-relaxation' step that can improve the convergence speed. This step is typically useful for problems where ADMM exhibits very slow convergence and can be tuned using the parameter α. Its value should satisfy $1 \leq \alpha \leq 2$. This step using the following command:

Matlab

Python

```
codeoptions.ADMMalpha = 1;
```

```
codeoptions.ADMMalpha = 1
```

18.3.6 Dual Fast Gradient Method

For some problems with simple constraints, our implementation of the dual fast gradient method can be the fastest option. No parameters need to be tuned in this method.

18.3.7 Primal Fast Gradient Method

For problems with no equality constraints (only one stage) and simple constraints, the primal fast gradient method can give medium accuracy solutions extremely quickly. The method has several tuning parameters that can significantly affect the performance.

Accuracy requirements

The accuracy required for convergence (exitflag `1`) can be set as follows:

Matlab

Python

```
codeoptions.accuracy.gmap = 1e-5;  % infinity norm of the gradient map
```

```
codeoptions.accuracy.gmap = 1e-5  # infinity norm of the gradient map
```

The gradient map is related to the difference with respect to the optimal objective value. Just like with other first-order methods, the required number of FG iterations varies very significantly depending on the requested accuracy. Medium accuracy solutions can typically be computed very quickly, but many iterations are needed to achieve the same accuracy as with interior-point methods.

Method parameters

The user has to determine the step size in the fast gradient method. The convergence speed of FG is highly variable in this parameter, which should typically be set to be one over the maximum eigenvalue of the quadratic cost function. This parameter can be tuned using the following command:

Matlab

Python

```
codeoptions.FGstep = 1/1000;
```

```
codeoptions.FGstep = 1/1000
```

In some cases it may be possible to let FORCESPRO choose the step size automatically. To enable this feature set:

Matlab

Python

```
codeoptions.FGautostep = 1;
```

```
codeoptions.FGautostep = 1
```

Warm starting

The performance of the fast gradient method can be greatly influenced by the way the variables are initialized. Unlike with interior-point methods, fast gradient methods can be very efficiently warm started with a good guess for the optimal solution. To enable this feature set:

Matlab

Python

```
codeoptions.warmstart = 1;
```

```
codeoptions.warmstart = 1
```

When the user turns warm start on, a new parameter `z_init_0` is automatically added. The user should set it to be a good guess for the solution, which is typically available when solving a sequence of problems.

18.4 High-level interface options

The FORCESPRO NLP solver of the high-level interface implements a nonlinear barrier interior-point method. We will now discuss how to change several parameters in the solver.

18.4.1 Integrators

When providing the continuous dynamics the user must select a particular integrator by setting `nlp.integrator.type` as outlined in Table 18.12.

Table 18.12: Integrators options

`nlp.integrator.type`	Type	Order
`'ForwardEuler'`	Explicit Euler Method	1
`'ERK2'`	Explicit Runge-Kutta	2
`'ERK3'`	Explicit Runge-Kutta	3
`'ERK4'` (default)	Explicit Runge-Kutta	4
`'BackwardEuler'`	Implicit Euler Method	1
`'IRK2'`	Implicit Runge-Kutta	2
`'IRK4'`	Implicit Runge-Kutta	4

The user must also provide the discretization interval (in seconds) and the number of intermediate shooting nodes per interval. For instance:

Matlab

Python

```
codeoptions.nlp.integrator.type = 'ERK2';
codeoptions.nlp.integrator.Ts = 0.01;
codeoptions.nlp.integrator.nodes = 10;
```

```
codeoptions.nlp.integrator.type = 'ERK2'
codeoptions.nlp.integrator.Ts = 0.01
codeoptions.nlp.integrator.nodes = 10
```

Tip: Usually an explicit integrator such as RK4 should suffice for most applications. If you have stiff systems, or suspect inaccurate integration to be the cause of convergence failure of the NLP solver, consider using implicit integrators from the table above.

Note: Note that the implicit integrators `BackwardEuler`, `IRK2` and `IRK4` currently rely on the CasADi AD tool and their legacy variant cannot be used in combination with CasADi MX expressions (see *Automatic differentiation expression class*).

Expert options for implicit integrators

The implicit integrators `BackwardEuler`, `IRK2` and `IRK4` do not just evaluate the differential equation, but actually solve a nonlinear equation to obtain the state trajectory. This is done by means of Newton iterations, with default values of `10` iterations for `BackwardEuler` and `5` iterations for `IRK2` and `IRK4`. These default values can be overwritten by using the following option:

Matlab

Python

```
codeoptions.nlp.integrator.newtonIter = 3;
```

```
codeoptions.nlp.integrator.newtonIter = 3
```

In order to reduce computational effort, the Jacobian of the nonlinear equation is only computed once by default. If your differential equations are highly nonlinear, it may be worth the effort to recompute it at every Newton iteration. This is achieved by means of the following option:

Matlab

Python

```
codeoptions.nlp.integrator.reuseNewtonJacobian = 0;
```

```
codeoptions.nlp.integrator.reuseNewtonJacobian = 0
```

Code-generated integrators

From FORCESPRO `4.1.0`, integrators generated on the server are available when using explicit integrators `ForwardEuler`, `RK2`, `RK3` and `RK4`, and the field *continuous_dynamics* is set in the *model* structure. From FORCESPRO `4.4.0` the implicit integration scheme `IRK2` was added to the list of supported codegenerated integration schemes. These integrators result in much smaller code size than previously. They also often result in faster run times on embedded targets.

Two different methods are used to compute sensitivities associated to these integrators:

- `chainrule`, which is the default option, can also be triggered by setting

Matlab

Python

```
codeoptions.nlp.integrator.differentiation_method = 'chainrule';
```

```
codeoptions.nlp.integrator.differentiation_method = 'chainrule'
```

- `vde`, which can be triggered by settings the following option:

Matlab

Python

```
codeoptions.nlp.integrator.differentiation_method = 'vde';
```

```
codeoptions.nlp.integrator.differentiation_method = 'vde'
```

When using the **vde** option, the following options also need to be set

Matlab

Python

```
codeoptions.nlp.sensitivity.Ts = codeoptions.nlp.integrator.Ts;
codeoptions.nlp.sensitivity.nodes = codeoptions.nlp.integrator.nodes / 2;
    % When using 'ERK2' or 'ERK4' for the sensitivity computation, the number of␣
↪nodes for the sensitivity
    % needs to be twice the number of nodes for the integrator
codeoptions.nlp.sensitivity.type = 'ERK4'; % Can also be 'ForwardEuler', 'RK2'␣
↪depending on the application
```

```
codeoptions.nlp.sensitivity.Ts = codeoptions.nlp.integrator.Ts
codeoptions.nlp.sensitivity.nodes = codeoptions.nlp.integrator.nodes / 2
    # When using 'ERK2' or 'ERK4' for the sensitivity computation, the number of␣
↪nodes for the sensitivity
    # needs to be twice the number of nodes for the integrator
codeoptions.nlp.sensitivity.type = 'ERK4' # Can also be 'ForwardEuler', 'RK2'␣
↪depending on the application
```

The **vde** option is likely to change the numerical behavior of the solver but can help for reducing the solve time in some cases, typically by having a looser integration on sensitivity.

Note: The **vde** option currently is still in an experimental state and we are working to fully robustify it. You may give it a try, but be prepared for unexpected behavior. Also, the **RK3** integration method is currently not supported with the **vde** option.

Linear subsystem exploitation

Often nonlinear optimal control problems contain linear subsystems, meaning part of the differential equation describing the dynamics of the system is linear while another part is nonlinear. By this we mean that the state x of the system can be split into two parts $x = (x_1, x_2)$ such that the differential equation $\dot{x} = c(x, u)$ (u denoting the control input) governing the dynamics of the system can be written as

$$\dot{x}_1 = A_1 x_1 + B_1 u \tag{18.1}$$

$$\dot{x}_2 = c_2(x_1, x_2, u). \tag{18.2}$$

Here A_1 and B_1 denote constant matrices and c_2 denotes a function. Since FORCESPRO **4.4.0** it is possible to exploit such subsystems for performance by performing parts of the numerical integration of the system offline. Currently this is supported only for the codegenerated **ERK4** integration scheme (see *Code-generated integrators*). FORCESPRO can automatically detect a linear subsystem if it exists. One can activate the detection of linear subsystems by enabling the **codeoptions.nlp.integrator.attempt_subsystem_exploitation** option as follows:

Matlab

Python

```
codeoptions.nlp.integrator.attempt_subsystem_exploitation = 1;
```

```
codeoptions.nlp.integrator.attempt_subsystem_exploitation = 1
```

Optionally, in combination with setting this option, one can also specify the state indices of the linear subsystem manually. These indices are specified as a numpy array of integers in Python and a vector of indices in MATLAB via the field **model.linInIdx**. For example, in a case

when $x \in \mathbb{R}^2$, $u \in \mathbb{R}$ and the right-hand-side c of the ODE describing the dynamics of the system is given by

$$c(x,u) = \begin{pmatrix} x_2 \\ x_1 \\ \cos(x_1) + \sin(x_2) + x_3 + u \end{pmatrix},$$

one would have to specify

Matlab

Python

```
model.linInIdx = [1, 2];
```

```
model.linInIdx = np.array([0, 1], dtype=np.int)
```

Note the 1-based indexing in MATLAB and the 0-based indexing in Python. For further details on how to exploit linear subsystems using FORCESPRO, see *Controlling a crane using a FORCESPRO NLP solver*.

Note: For large systems (more than about 16 states) there might be a considerable overhead in determining the indices of the linear subsystem automatically. In case you encounter such an overhead, you can avoid it by manually specifying `model.linInIdx` as shown above.

Moreover, note that linear subsystems currently cannot be exploited when using CasADi MX expressions (see *Automatic differentiation expression class*).

18.4.2 Accuracy requirements

One can modify the termination criteria by altering the KKT tolerance with respect to stationarity, equality constraints, inequality constraints and complementarity conditions, respectively, using the following fields:

Matlab

Python

```
% default tolerances
codeoptions.nlp.TolStat = 1e-5; % inf norm tol. on stationarity
codeoptions.nlp.TolEq = 1e-6;   % tol. on equality constraints
codeoptions.nlp.TolIneq = 1e-6; % tol. on inequality constraints
codeoptions.nlp.TolComp = 1e-6; % tol. on complementarity
```

```
# default tolerances
codeoptions.nlp.TolStat = 1e-5 # inf norm tol. on stationarity
codeoptions.nlp.TolEq = 1e-6   # tol. on equality constraints
codeoptions.nlp.TolIneq = 1e-6 # tol. on inequality constraints
codeoptions.nlp.TolComp = 1e-6 # tol. on complementarity
```

All tolerances are computed using the infinity norm $\|\cdot\|_\infty$.

18.4.3 Barrier strategy

The strategy for updating the barrier parameter is set using the field:

Matlab

Python

```
codeoptions.nlp.BarrStrat = 'loqo';
```

```
codeoptions.nlp.BarrStrat = 'loqo'
```

It can be set to `'loqo'` (default) or to `'monotone'`. The default settings often leads to faster convergence, while `'monotone'` may help convergence for difficult problems.

18.4.4 Hessian approximation

The way the Hessian of the Lagrangian function is computed can be set using the field:

Matlab

Python

```
codeoptions.nlp.hessian_approximation = 'bfgs';
```

```
codeoptions.nlp.hessian_approximation = 'bfgs'
```

FORCESPRO currently supports BFGS updates (`'bfgs'`) (default) and Gauss-Newton approximation (`'gauss-newton'`). Exact Hessians will be supported in a future version. Read the subsequent sections for the corresponding Hessian approximation method of your choice.

BFGS options

When the Hessian is approximated using BFGS updates, the initialization of the estimates can play a critical role in the convergence of the method. The default value is the identity matrix, but the user can modify it using e.g.:

Matlab

Python

```
model.bfgs_init = diag([0.1, 10, 4]);
```

```
model.bfgs_init = np.diag(np.array([0.1, 10, 4]))
```

Note that BFGS updates are carried out individually per stage in the FORCESPRO NLP solver, so the size of this matrix is the size of the stage variable. Also note that this matrix must be positive definite. When the cost function is positive definite, it often helps to initialize BFGS with the Hessian of the cost function. It's also possible to specify the initialization of the BFGS per stage (especially when the dimensions of the problem vary per stage):

Matlab

Python

```
model.bfgs_init = cell(1, model.N);
for i = 1:model.N
  model.bfgs_init{i} = diag([0.1, 10, 4]);
end
```

```
model.bfgs_init = list([] for _ in range(model.N))
for i in range(model.N):
  model.bfgs_init[i] = np.diag(np.array([0.1, 10, 4]))
```

Finally, it's possible to specify a initialization separately for the final stage:

Matlab

Python

```
model.bfgs_initN = diag([0.1, 10, 4]); % overrides model.bfgs_init{model.N}
```

```
model.bfgs_initN = np.diag(np.array([0.1, 10, 4])) % overrides model.bfgs_init[model.
↪N - 1]
```

This matrix is also used to restart the BFGS estimates whenever the BFGS updates are skipped several times in a row. The maximum number of updates skipped before the approximation is re-initialized is set using:

Matlab

Python

```
codeoptions.nlp.max_update_skip = 2;
```

```
codeoptions.nlp.max_update_skip = 2
```

The default value for `max_update_skip` is 2.

In order to set the BFGS initialization through the `bfgs_init` codeoption one must first come up with a guess for a good BFGS initialization. One way to do so is to first run the solver without any user-defined BFGS initialization (i.e. not setting `model.bfgs_init`) and using the BFGS matrix reached upon convergence as an initialization. One can export the BFGS matrix by setting

Matlab

Python

```
% diagonal of BFGS
codeoptions.exportBFGS = 1;
% lower triangular of BFGS
codeoptions.exportBFGS = 2;
```

```
# diagonal of BFGS
codeoptions.exportBFGS = 1
# lower triangular of BFGS
codeoptions.exportBFGS = 2
```

Instead of specifying the BFGS initialization at codegen one can also specify it at run-time. In order to do this one should set

Matlab

Python

```
codeoptions.nlp.parametricBFGSinit = 1;
```

```
codeoptions.nlp.parametricBFGSinit = 1
```

before generating the FORCESPRO solver. Having done this, the generated solver will expect an input `problem.BFGSinitLower<stage number>` for every stage. This is a vector which specifies the BFGS hessian initialization in LOWER TRIANGULAR ROW MAJOR format. Thus, in order

to specify e.g. the matrix

$$\begin{pmatrix} a_1 & 0 & 0 \\ 0 & a_2 & 0 \\ 0 & 0 & a_3 \end{pmatrix}$$

for constants $a_1, a_2, a_3 > 0$ as the BFGS initialization at stage 6 out of 50 stages in total, one would specify

Matlab

Python

```
problem.BFGSinitLower06 = [a_1, 0, a_2, 0, 0, a_3];
```

```
problem["BFGSinitLower06"] = np.array([a_1, 0, a_2, 0, 0, a_3])
```

Gauss-Newton options

For problems that have a least squares objective, i.e. the cost function can be expressed by a vector-valued function $r_k : \mathbb{R}^n \to \mathbb{R}^m$ which implicitly defines the objective function as:

$$f_k(z_k, p_k) = \frac{1}{2}\|r_k(z_k, p_k)\|_2^2,$$

the Gauss-Newton approximation of the Hessian is given by:

$$\nabla^2_{xx} L_k \approx \nabla r_k(z_k, p_k) \nabla r_k(z_k, p_k)^\top$$

and can lead to faster convergence and a more reliable method. When this option is selected, the functions r_k have to be provided by the user in the field `LSobjective`. For example if $r(z) = \sqrt{100}z_1^2 + \sqrt{6}z_2^2$, i.e. $f(z) = 50z_1^2 + 3z_2^2$, then the following code defines the least-squares objective (note that r is a vector-valued function):

Matlab

Python

```
model.objective = @(z) 0.1* z(1)^2 + 0.01*z(2)^2;
model.LSobjective = @(z) [sqrt(0.2)*z(1); sqrt(0.02)*z(2)];
```

```
# not yet implemented
```

Important: The field `LSobjective` will have precedence over `objective`, which need not be defined in this case.

When providing your own function evaluations in C, you must populate the Hessian argument with a positive definite Hessian.

18.4.5 Line search settings

The line search first computes the maximum step that can be taken while maintaining the iterates inside the feasible region (with respect to the inequality constraints). The maximum distance is then scaled back using the following setting:

Matlab

Python

```
% default fraction-to-boundary scaling
codeoptions.nlp.ftbr_scaling = 0.9900;
```

```
# default fraction-to-boundary scaling
codeoptions.nlp.ftbr_scaling = 0.9900;
```

18.4.6 Regularization

To avoid ill-conditioned saddle point systems, FORCESPRO employs two different types of regularization, static and dynamic regularization.

Static regularization

Static regularization of the augmented Hessian by $\delta_w I$, and of the multipliers corresponding to the equality constraints by $-\delta_c I$ helps avoid problems with rank deficiency. The constants δ_w and δ_c vary at each iteration according to the following heuristic rule:

$$\delta_w = \eta_w \min(\mu, \|c(x)\|))^{\beta_w} \cdot (i+1)^{-\gamma_w} + \delta_{w,\min}$$
$$\delta_c = \eta_c \min(\mu, \|c(x)\|))^{\beta_c} \cdot (i+1)^{-\gamma_c} + \delta_{c,\min}$$

where μ is the barrier parameter and i is the number of iterations.

This rule has been chosen to accommodate two goals: First, make the regularization dependent on the progress of the algorithm - the closer we are to the optimum, the smaller the regularization should be in order not to affect the search directions generated close to the solution, promoting superlinear convergence properties. Second, the amount of regularization employed should decrease with the number of iterations to a certain minimum level, at a certain sublinear rate, in order to prevent stalling due to too large regularization. FORCESPRO NLP does not employ an inertia-correcting linear system solver, and so relies heavily on the parameters of this regularization to be chosen carefully.

You can change these parameters by using the following settings:

Matlab

Python

```
% default static regularization parameters
codeoptions.nlp.reg_eta_dw = 1e-4;
codeoptions.nlp.reg_beta_dw = 0.8;
codeoptions.nlp.reg_min_dw = 1e-9;
codeoptions.nlp.reg_gamma_dw = 1.0/3.0;

codeoptions.nlp.reg_eta_dc = 1e-4;
codeoptions.nlp.reg_beta_dc = 0.8;
codeoptions.nlp.reg_min_dc = 1e-9;
codeoptions.nlp.reg_gamma_dc = 1.0/3.0;
```

```
# default static regularization parameters
codeoptions.nlp.reg_eta_dw = 1e-4
codeoptions.nlp.reg_beta_dw = 0.8
codeoptions.nlp.reg_min_dw = 1e-9
codeoptions.nlp.reg_gamma_dw = 1.0/3.0

codeoptions.nlp.reg_eta_dc = 1e-4
codeoptions.nlp.reg_beta_dc = 0.8
```

(continues on next page)

(continued from previous page)

```
codeoptions.nlp.reg_min_dc = 1e-9
codeoptions.nlp.reg_gamma_dc = 1.0/3.0
```

Note that by choosing $\delta_w = 0$ and $\delta_c = 0$, you can turn off the progress and iteration dependent regularization, and rely on a completely static regularization by $\delta_{w,\min}$ and $\delta_{c,\min}$, respectively.

Dynamic regularization

Dynamic regularization regularizes the matrix on-the-fly to avoid instabilities due to numerical errors. During the factorization of the saddle point matrix, whenever it encounters a pivot smaller than ϵ, it is replaced by δ. There are two parameter pairs: (ϵ, δ) affects the augmented Hessian and (ϵ_2, δ_2) affects the search direction computation. You can set these parameters by:

Matlab

Python

```
% default dynamic regularization parameters
codeoptions.regularize.epsilon = 1e-12; % (for Hessian approx.)
codeoptions.regularize.delta = 4e-6; % (for Hessian approx.)
codeoptions.regularize.epsilon2 = 1e-14; % (for Normal eqs.)
codeoptions.regularize.delta2 = 1e-14; % (for Normal eqs.)
```

```
# default dynamic regularization parameters
codeoptions.regularize.epsilon = 1e-12 # (for Hessian approx.)
codeoptions.regularize.delta = 4e-6 # (for Hessian approx.)
codeoptions.regularize.epsilon2 = 1e-14 # (for Normal eqs.)
codeoptions.regularize.delta2 = 1e-14 # (for Normal eqs.)
```

18.4.7 Linear system solver

The interior-point method solves a linear system to find a search direction at every iteration. FORCESPRO NLP offers the following four linear solvers:

- `'normal_eqs'` (default): Solving the KKT system in normal equations form.
- `'symm_indefinite'`: improved variant of `'symm_indefinite_legacy'` introduced in FORCESPRO version 5.0.0; roughly as efficient as `normal_eqs` but more robust.
- `'symm_indefinite_fast'`: Solving the KKT system in augmented / symmetric indefinite form, using regularization and positive definite Cholesky factorizations only. This is often the fastest solver but may be less numerical stable than `symm_indefinite`.
- `'symm_indefinite_legacy'`: Solving the KKT system in augmented / symmetric indefinite form; may be removed in a future release

The linear system solver can be selected by setting the following field:

Matlab

Python

```
codeoptions.nlp.linear_solver = 'symm_indefinite';
```

```
codeoptions.nlp.linear_solver = 'symm_indefinite'
```

It is recommended to try different linear solvers as the robustness and speed of the solvers are problem-dependent. The overall most robust method is `symm_indefinite`, which is also very efficient. For certain problems `normal_eqs` and `'symm_indefinite_fast'` may be slightly faster than `symm_indefinite` but possibly also slightly less numerically stable.

Note: Independent of the linear system solver choice, the generated code is always library-free and statically allocated, i.e. it can be embedded anywhere.

Note: From FORCESPRO version 5.0.0 onwards, the option *symm_indefinite* refers to an improved version; use *symm_indefinite_legacy* to restore the previous default.

The `'normal_eqs'` solver is the standard FORCESPRO linear system solver based on a full reduction of the KKT system (the so-called normal equations form). It works well for standard problems, especially convex problems or nonlinear problems where the BFGS or Gauss-Newton approximations of the Hessian are numerically sufficiently well conditioned.

The `'symm_indefinite'` solver is numerically more robust than `'normal_eqs'` and `symm_indefinite_fast` and typically similarly efficient. It is an improved variant of the `'symm_indefinite_legacy'`. Furthermore, it implements iterative refinement which further improves numerical stability (see *Iterative refinement*).

The `'symm_indefinite_fast'` solver is typically the fastest solver. Currently only used for receding-horizon/MPC-like problems where dimensions of all stages are equal (except for the first and last stage, those are handled separately).

The `'symm_indefinite_legacy'` solver is the most robust one, but currently replaced by an at least equally robust improved variant.

Iterative refinement

The linear solver `'symm_indefinite'` supports iterative refinement to further improve numerical stability. Iterative refinement is recommended for problems that don't converge due to numerical issues but can be safely disabled (default) for most problems. In order to enable iterative refinement, set `codeoptions.nlp.refinement_steps` to the desired number of steps > 0. Two types of iterative refinement are implemented which can be selected by setting `codeoptions.nlp.refinement_type` as outlined in Table 18.13.

Table 18.13: Options for setting iterative refinement type

`nlp.refinement_type`	Description
`0`	Includes additional modification (default)
`1`	Strictly based on the original linear system

18.4.8 Automatic differentiation tool

If external functions and derivatives are not provided directly as C code by the user, FORCESPRO makes use of an automatic differentiation (AD) tool to generate efficient C code for all the functions (and their derivatives) inside the problem formulation. Currently, two different AD tools (or four different AD tool versions) are supported that can be chosen by means of the setting `nlp.ad_tool` as summarized in Table 18.14.

Table 18.14: Automatic differentiation tool options

nlp.ad_tool	Tool	URL
'casadi'	CasADi (as in path or v3.5.5)	CasADi
'casadi-2.4.2'	CasADi v2.4.2	CasADi
'casadi-3.5.1'	CasADi v3.5.1	CasADi
'casadi-3.5.5'	CasADi v3.5.5	CasADi
'casadi-3.6.3'	CasADi v3.6.3	CasADi
'casadi-3.6.7'	CasADi v3.6.7	CasADi
'symbolic-math-tbx'	MathWorks Symbolic Math Toolbox	MathWorks

Note that MathWorks Symbolic Math Toolbox requires an additional license, which is why the default option is set to

Matlab

Python

```
codeoptions.nlp.ad_tool = 'casadi';
```

```
codeoptions.nlp.ad_tool = 'casadi'
```

Also note that the use of implicit integrators **BackwardEuler**, **IRK2** and **IRK4** (see Section 18.4.1) currently still rely on using the CasADi AD tool.

18.4.9 Automatic differentiation expression class

The AD tool CasADi supports two different types of symbolic expressions, called **SX** and **MX**. While SX expressions tend to be leaner and offer best performance for mathematical "operations that are naturally written as a sequence of scalar operations" (see CasADi's symbolic framework)), MX expressions offer advantages for matrix operations.

FORCESPRO only supported SX expressions prior to version 6.1.0 and still uses them by default. However, starting with version 6.1.0, FORCESPRO offers to make use of MX expressions by setting the following codeoption:

Matlab

Python

```
codeoptions.nlp.ad_expression_class = 'MX';
```

```
codeoptions.nlp.ad_expression_class = 'MX'
```

All supported values of that codeoption **nlp.ad_expression_class** are summarized in Table 18.15. Note that this option is only supported when using CasADi as AD Tool and that MX expressions are only supported for CasADi versions 3.5.1, 3.5.5, 3.6.3 and 3.6.7.

Table 18.15: Automatic differentiation expression classes

nlp.ad_expression_class	Supported AD Tools
'SX'	CasADi v2.4.2, v3.5.1, v3.5.5, v3.6.3, v3.6.7
'MX'	CasADi v3.5.1, v3.5.5, v3.6.3, v3.6.7

Note: Use CasADi MX expressions with care. In particular, when switching your FORCESPRO formulation between using SX and MX expressions, minor numerical differences may arise that can lead to (usually minor) differences in number of iterations or optimal solutions.

Note: Use of CasADi MX expressions is currently *not supported* in combination with any of the following FORCESPRO features: legacy and symbolic dump tool, implicit legacy integrators, linear subsystem exploitation, MINLP formulations and inside the MathWorks MPC Plugins.

18.4.10 Re-use of callback code

When defining your NLP problem formulation using an AD tool, you may specify objective functions, dynamic equations and constraints separately on each stage. In order to reduce the size of the generated callback code, FORCESPRO will perform a check whether all these callbacks are identical on any two or more stages and if so, only generates the callback code for those stages once. However, checking for exact identity can be tricky and may sometimes lead to false results. By default, FORCESPRO performs a less strict check for identity resulting in less duplicated callback code. If you observe that two stages are wrongly identified as identical, you can enable a more strict check by using the following codeoption:

Matlab

Python

```
codeoptions.nlp.strictCheckDistinctStages = 1;
```

```
# not yet supported
```

Note that using this option may be overly conservative and lead to duplicated callback code for different stages that are actually identical.

18.4.11 Safety checks

By default, the output of the function evaluations is checked for the presence of `NaN`s or `INF`s in order to diagnose potential initialization problems. In order to speed up the solver one can remove these checks by setting:

Matlab

Python

```
codeoptions.nlp.checkFunctions = 0;
```

```
codeoptions.nlp.checkFunctions = 0
```

18.4.12 Reducing NLP code size

The size of the solver libraries generated with `codeoptions.solvemethod = 'PDIP_NLP'` can be reduced by means of the option `nlp.compact_code`. By setting

Matlab

Python

```
codeoptions.nlp.compact_code = 1;
```

```
codeoptions.nlp.compact_code = 1
```

the user enables the FORCESPRO server to generate smaller code, which results in shorter compilation time and slightly better solve time in some cases. This feature is especially effective on long horizon problems.

Note: The `compact_code` option is currently only supported when using the linear systems solver `codeoptions.nlp.linear_solver = 'normal_eqs'` (which is the default choice).

Chapter 19

Exitflags

- *Exitflags and quality of the result*
- *Mixed integer Nonlinear Programming exitflags*

The FORCESPRO solver can identify issues that occur during its execution (whether they are of technical or of numerical nature) and notify the user of these issues. The solver is reporting these issues in the form of exit status codes called *exitflags*. Usages of the *exitflags*:

- The user can identify and fix configuration errors and mistakes with calling the solver.
- Certain *exitflags* can provide hints about issues to the problem formulation. By consulting the returned *exitflags* the user can perform improvements to the problem formulation which can lead to a more robust problem formulation.
- The user can identify issues occurring during execution on the fly. This can be used to prepare a strategy with exitflag-based decisions in order to achieve a more robust execution (handling both normal scenarios and edge cases).

19.1 Exitflags and quality of the result

The FORCESPRO solver returns 3 output elements in order to report the status and result of the execution:

- `output`: contains the solution to the problem given to the solver to be solved
- `exitflag`: contains the exitflag status code returned by the solver
- `info`: contains further information about the execution and solution of the problem

The standard format of an execution of the FORCESPRO solver is shown below:

Matlab

Python

```
[output, exitflag, info] = FORCESNLPsolver(problem);
```

```
output, exitflag, info = solver.solve(problem)
```

The possible exitflags are documented in Table 19.1. The exitflag should always be checked before continuing with program execution to avoid using spurious solutions later in the code. Check whether the solver has exited without an error before using the solution. For example, we suggest to use an assert statement:

Matlab

Python

```
assert(exitflag == 1, 'Some issue with FORCESPRO solver');
```

```
assert exitflag == 1, "Some issue with FORCESPRO solver"
```

Table 19.1: Exitflag values

Exitflag	Description
1	Locally optimal solution found (i.e. the point satisfies the KKT optimality conditions to the requested accuracy).
2	(for binary branch-and-bound) A feasible point has been identified for which the objective value is no more than `codeoptions.mip.mipgap*100` per cent worse than the global optimum. (for all other solvers) Specified timeout set for the solver execution has been reached. Best non-optimal point found during execution will be returned. You can examine the value of optimality conditions returned inside the `info` struct by FORCESPRO to decide whether the point returned is acceptable.
3	Solver execution has been terminated as the termination parameter has been set by the user. Best non-optimal point found during execution will be returned. You can examine the value of optimality conditions returned inside the `info` struct by FORCESPRO to decide whether the point returned is acceptable.
11	This exitflag is not returned for a solver call. It is returned when calling the external callbacks of the solver (see *Calling the nonlinear functions from MATLAB or Python*). This exitflag indicates that the execution of the integrator in the external callbacks was successful.
12	This exitflag is not returned for a solver call. It is returned when calling the external callbacks of the solver (see *Calling the nonlinear functions from MATLAB or Python*). This exitflag indicates that the number of steps set by the user for the integrator exceeded the maximum number of steps allowed.
0	(for binary branch-and-bound) maximum computation time of `codeoptions.mip.timeout` reached. The returned solution is the best one found so far. (for all other solvers) Maximum number of iterations reached. You can examine the value of optimality conditions returned inside the `info` struct by FORCESPRO to decide whether the point returned is acceptable.
-1	(only binary branch-and-bound) Infeasible problem (issues solving the root relaxation to desired accuracy).
-2	(only binary branch-and-bound) Out of memory – cannot fit branch and bound nodes into pre-allocated memory.
-3	Deprecated.
-4	Wrong number of inequalities input to solver.
-5	Error occurred during matrix factorization.
-6	`NaN` or `INF` occurred during functions evaluations in external callbacks or due to an internal error of the nonlinear solver.
-7	The solver could not proceed. Most likely cause is that the problem is infeasible.Try formulating a problem with slack variables (soft constraints) to avoid this error. For low-level formulations, see also *Debugging a formulation*.
-8	The internal QP solver could not proceed. This exitflag can only occur when using the *Sequential quadratic programming algorithm*. The most likely cause is that an infeasible QP or a numerical unstable QP was encountered. Try increasing the Hessian regularization parameter `reg_hessian` (see *SQP specific codeoptions*).
-9	Regularization has been set to an invalid (i.e. negative) value. This exitflag can only occur when using `codeoptions.nlp.linear_solver = 'symm_indefinite'` (see *Linear system solver*).
-10	`NaN` or `INF` occurred during evaluation of functions and derivatives or due to an internal error of the convex solver. The problem might be infeasible. If this occurs at iteration zero, try changing the initial point. For example, for a cost function $1/\sqrt{x}$ with an initialization $x_0 = 0$, this error would occur. For low-level formulations, see also *Debugging a formulation*.
-11	Invalid values in problem parameters.
-12	Specified timeout set for the solver was too small to find any point (op-

Note: Certain exitflags might not apply for all solve methods that can be selected for FORCESPRO. The available exitflags for a solver can also be found in the generated solver files, in the header file (under the *include* folder) as well as in the help section of the solver for the available interfaces (e.g. MATLAB, Simulink or Python).

19.2 Mixed integer Nonlinear Programming exitflags

The FORCESPRO MINLP solver returns a different set of status code exitflags which are available in the `info` output element of the FORCESPRO solver. The available exitflags for the MINLP solver can found below in Table 19.2.:

Table 19.2: MINLP exitflag values

Exitflag	Description
`0`	Local optimum found. Final tolerance on integrality gap satisfied.
`1`	Integer feasible solution found.
`2`	No integer feasible found.
`3`	Relaxation solve failed at root node.
`4`	Unexpected error occurred.
`5`	Search failed to converge.

Chapter 20

Modelling Utilities

- *Interpolations 1D (e.g. splines)*
  - *Polynomial Parameterization*
  - *Automatic Fit from Data*
  - *Application Example*
- *Interpolations 2D (B-splines)*
  - *B-Spline Representation*
  - *Parametric B-Spline Representation*
  - *Fitting B-Spline on Gridded Data*
  - *Fitting B-Spline on Arbitrary Data*
  - *Application Example*
- *Smooth Approximations*
  - *Smooth Minimum*
  - *Smooth Maximum*

Like any derivative-based optimization solver, FORCESPRO works best if all functions defining the optimization problem are sufficiently smooth (i.e. at least continuously differentiable once). Both the MATLAB and the Python client of FORCESPRO come along with a couple of utility functions to assist the user with setting up such smooth problem formulations. This chapter provides details on those utility functions for modelling.

20.1 Interpolations 1D (e.g. splines)

If a given function is based on measurement data (or any other set of discrete data points), one can interpolate between those data points to yield a continuous function. FORCESPRO can either create such a function directly from the data points or allows you to provide a polynomial parameterization that can be used inside your symbolic problem formulation.

20.1.1 Polynomial Parameterization

A polynomial parameterization can be obtained by providing a vector containing `M+1` break points (defining `M` interpolation intervals or pieces) as well as an array defining `M` sets of `N+1`

polynomial coefficients, each set defining a local polynomial of order `N` for each of those pieces.

Calling the line

Matlab

Python

```
f = ForcesInterpolation(breaks, coefs);
```

```
f = forcespro.modelling.Interpolation(breaks, coefs)
```

will yield a symbolic representation of a polynomial in standard form

$$f(x) = \sum_{j=0}^{N} c_{ij} x^j \quad \forall\, b_{i-1} \leq x \leq b_i \quad \forall\, i \in \{1, \ldots, M\}\,,$$

where *b* denotes the break points **breaks** and *c* denotes the coefficients **coefs**. *f(x)* can be a scalar or a `K`-dimensional function, i.e. **coefs** may be given for a multi-valued interpolation. For more details on how to pass those input parameters, we refer to the respective help function as the format differs slightly between the MATLAB and the Python client to follow domain-specific conventions.

In case your coefficients are defined relative to beginning of each piece, you can call

Matlab

Python

```
f = ForcesInterpolation(breaks, coefs, 'pp');
```

```
f = forcespro.modelling.Interpolation(breaks, coefs, 'pp')
```

to yield a symbolic representation of a polynomial in "piecewise polynomial" form

$$f(x) = \sum_{j=0}^{N} c_{ij} (x - b_i)^j \quad \forall\, b_{i-1} \leq x \leq b_i \quad \forall\, i \in \{1, \ldots, M\}\,.$$

Note: In addition to providing fixed numerical values for break points and coefficients, you may also pass symbolic quantities for some or all of those! This will allow you to change the parameterization of your interpolation on the fly, e.g. by means of real-time parameters that are passed to the FORCESPRO solver.

The symbolic interpolation `f` can now be used inside your problem formulation by evaluating it, either at a fixed value or at any symbolic quantity, e.g.

Matlab

Python

```
% assuming a state vector x and a control input u
y = f(x(1)) + u(1);
```

```
% assuming a state vector x and a control input u
y = f(x[0]) + u[0]
```

Important: Symbolic interpolations are currently only supported when using CasADi as AD tool.

20.1.2 Automatic Fit from Data

In case you do not want to specify break points and coefficients yourself, you can fit data points directly by calling:

Matlab

Python

```
f = ForcesInterpolationFit(X, Y, method);
```

```
f = forcespro.modelling.InterpolationFit(X, Y, kind)
```

Here, `X` and `Y` are vectors (say, of dimension `L`) of data points to yield an interpolation that satisfies

$$f(X_i) = Y_i \quad \forall i \in \{1, \ldots, L\}\,.$$

The third argument `method`/`kind` specifies the method to be used to obtain that fit using built-in functionality of either MATLAB (see Table Table 20.1) or Python (see Table Table 20.2).

The symbolic interpolation `f` can be used the same way as described in section Section 20.1.1.

Table 20.1: Interpolation Method for MATLAB (see MATLAB's `interp1` for more details)

`method`	Description
`'linear'`	Piecewise linear
`'nearest'`	Piecewise constant, value from nearest data point
`'next'`	Piecewise constant, value from next data point
`'previous'`	Piecewise constant, value from previous data point
`'spline'` (default)	Piecewise cubic spline
`'pchip'`	Shape-preserving piecewise cubic spline

Table 20.2: Interpolation Method for Python (see SciPy's `interpolation` class for more details)

`kind`	Description
`'cubic'` (default)	Piecewise cubic spline
`'pchip'`	Shape-preserving piecewise cubic spline

20.1.3 Application Example

A full example on how to use interpolations inside your problem formulation can be found in the `examples` folder that comes with your client. See the files `ObstacleAvoidance/ObstacleAvoidance_splines.m` (MATLAB) and `ObstacleAvoidance/obstacle_avoidance_splines.py` (Python), respectively. Therein, both road limits are defined as splines and are enforced as inequality constraints.

20.2 Interpolations 2D (B-splines)

In many industrial applications, 2D lookup tables/maps are commonly used. These have to be approximated by continuous function representations in order to use gradient-based solvers. Such lookup tables can be approximated using two-dimensional B-spline representations. A B-spline is a piecewise polynomial function.

In the following we provide several functionalities for fitting B-splines as well as capabilities for supplying coefficients of already existing B-splines directly. Importantly, all methods require the use of CasADi as the automatic differentiation tool. Furthermore, the symbolic variables have to be set to be CasADi MX expressions, which can be done by `codeoptions.nlp.ad_expression_class = 'MX'`.

20.2.1 B-Spline Representation

If you have already generated your own 2D spline, which can be represented as a general B-spline, you can directly provide the coefficients, knots and degrees. The coefficient matrix is matrix-indexed, meaning the x-inputs correspond to the rows and the y-inputs correspond to the columns. The knots positions `knots_x` and `knots_y` entail the internal knots as well as the additional ones on the boundaries/endpoints.

Matlab

Python

```
f = ForcesInterpolation2D(knots_x, knots_y, coefs, degree_x, degree_y, m=1,␣
↪casadiOptions=struct());
```

```
f = forcespro.modelling.Interpolation2D(knots_x, knots_y, coefs, degree_x, degree_y,␣
↪m=1, casadiOptions={})
```

This function is a wrapper for `casadi.Function.bspline`, where `m` and `casadiOptions` are optional CasADi-function specific options. The returned `f` is a function handle taking as input the two inputs to the lookup table and returning the lookup table value: `z=f(x, y)`.

20.2.2 Parametric B-Spline Representation

It is also possible to supply the coefficients online as runtime parameters. However, the knots and degrees still need to remain fixed, i.e. only the coefficient matrix can be changed online. The `numel_coefs` is the number of elements of the matrix-indexed coefficients matrix. The knots entail the internal as well as the additional ones on the boundaries/endpoints.

Matlab

Python

```
f = ForcesInterpolation2DParametric(numel_coefs, knots_x, knots_y, degree_x, degree_y,
↪ m=1, casadiOptions=struct());
```

```
f = forcespro.modelling.Interpolation2DParametric(numel_coefs, knots_x, knots_y,␣
↪degree_x, degree_y, m=1, casadiOptions={})
```

This function is a wrapper for `casadi.bspline`, where `m` and `casadiOptions` are optional CasADi-function specific options. The returned `f` is a function handle taking as inputs the two inputs to the lookup table as well as the coefficient matrix *flattened in column-major order* and returning the lookup table value: `z=f(x, y, coefs)`.

20.2.3 Fitting B-Spline on Gridded Data

For fitting on grid data (i.e. rectangular mesh), we can directly use CasADi's spline interpolation functionality.

Matlab

Python

```
f = ForcesInterpolation2DGridded(xgrid, ygrid, Z, scaling=[1, 1, 1],␣
↪casadiOptions=struct());
```

```
f = forcespro.modelling.Interpolation2DGridded(xgrid, ygrid, Z, scaling=[1, 1, 1],␣
↪casadiOptions={})
```

The `xgrid` and `ygrid` are the one-dimensional grid points in strictly ascending order and the `Z` is a two-dimensional matrix-indexed meshgrid of z values corresponding to `xgrid` and `ygrid` data sequences. Hence, `Z` has the size [length_x_grid, length_y_grid].

It additionally supports scaling of the spline inputs/outputs, which affects the numerical stability of the spline fitting routine. The scaling is automatically taken care of, so that the inputs/outputs to the resulting spline are still the physical quantities.

This function is a wrapper for `casadi.interpolant`, where `casadiOptions` are optional CasADi-function specific options. The returned `f` is a function handle taking as input the two inputs to the lookup table and returning the lookup table value: `z=f(x, y)`.

The documentation for CasADi lookup tables can be found here: here (CasADi).

20.2.4 Fitting B-Spline on Arbitrary Data

For fitting on arbitrarily aligned data, we provide functionality which makes use of SciPy's B-spline fitting routines: `SmoothBivariateSpline` and `LSQBivariateSpline` respectively. Hence, if utilizing these functions in MATLAB, a valid Python3 installation compatible with our MATLAB version needs to be present in our system. For a compatibility list of Python3 with different MATLAB versions, see here (Compatibility).

Note: Additionally, the Python3 package SciPy needs to be present in our system as well as the FORCESPRO client must be in our system's Python path. For installing SciPy, follow the instructions in here (SciPy) and for instructions on how to add the FORCESPRO client to the Python path see here (Python Path). In particular, to add the FORCESPRO client to the Python path within your current Matlab session, use the following commands:

```
py_path = py.sys.path;
py_path.insert(py.int(0), '/path/to/forcespro');
```

The first method is the `SmoothBivariateSpline` of SciPy:

Matlab

Python

```
f = ForcesInterpolationFit_SmoothBivariateSpline(x, y, z, w=ones(1, length(x)),␣
↪bbox=[min(x), max(x), min(y), max(y)], kx=3, ky=3, s=length(w), rankTol=1e-16,␣
↪scaling=[1, 1, 1], casadiOptions=struct());
```

```
f = forcespro.modelling.InterpolationFit_SmoothBivariateSpline(x, y, z, w=np.ones((1,␣
↪len(x))), bbox=[min(x), max(x), min(y), max(y)], kx=3, ky=3, s=len(w), eps=1e-16,␣
↪scaling=[1, 1, 1], casadiOptions={})
```

All the arguments up to `scaling` are SciPy specific arguments, which can be found here: here (SmoothBivariateSpline).

It additionally supports scaling of the spline inputs/outputs, which affects the numerical stability of the spline fitting routine. The scaling is automatically taken care of, so that the inputs/outputs to the resulting spline are still the physical quantities.

The fitted coefficients of SciPy are provided to a wrapper for `casadi.Function.bspline`, where `casadiOptions` are optional CasADi-function specific options. The returned `f` is a function handle taking as input the two inputs to the lookup table and returning the lookup table value: `z=f(x, y)`.

The second method is the `LSQBivariateSpline` of SciPy:

Matlab

Python

```
f = ForcesInterpolationFit_LSQBivariateSpline(x, y, z, tx, ty, w=ones(1, length(x)),␣
↪bbox=[min(x, tx), max(x, tx), min(y, ty), max(y, ty)], kx=3, ky=3, rankTol=1e-16,␣
↪scaling=[1, 1, 1], casadiOptions=struct());
```

```
f = forcespro.modelling.InterpolationFit_LSQBivariateSpline(x, y, z, tx, ty, w=np.
↪ones((1, len(x))), bbox=[min(x, tx), max(x, tx), min(y, ty), max(y, ty)], kx=3,␣
↪ky=3, eps=1e-16, scaling=[1, 1, 1], casadiOptions={})
```

All the arguments up to `scaling` are SciPy specific arguments, which can be found here: here (LSQBivariateSpline).

It additionally supports scaling of the spline inputs/outputs, which affects the numerical stability of the spline fitting routine. The scaling is automatically taken care of, so that the inputs/outputs to the resulting spline are still the physical quantities.

The fitted coefficients of SciPy are provided to `forcespro.modelling.Interpolation2D`, which is a wrapper for `casadi.Function.bspline`, where `casadiOptions` are optional CasADi-function specific options. The returned `f` is a function handle taking as input the two inputs to the lookup table and returning the lookup table value: `z=f(x, y)`.

20.2.5 Application Example

This example showcases how to generate splines from a 2D lookup table using the different provided functionalities.

Importantly, only the `SMOOTH` method was tuned properly, while `LSQ` and `GRIDDED` are tuned more roughly. Moreover, in this example, the coefficients provided to `COEFF` and `PARAMETRIC` are taken from the `SMOOTH` fitting solution.

The utilized 2D lookup represents the electric motor efficiency and is generated from figures in [JiaJibGor20_2D_modelling]. The 2D lookup table is stored as a matrix, where the rows correspond to the traction force, the columns to the velocity and each matrix entry represents the corresponding electric motor efficiency:

Matlab

Python

```
spline_method = 'SMOOTH'; % 'SMOOTH', 'LSQ', 'COEFF', 'GRIDDED', 'PARAMETRIC'

vMax = 50;      % maximum velocity in the lookup table [m/s]
FtMax = 5e3;    % maximum traction force in the lookup table [N]

vSampleOrig = 0:vMax/10:vMax;
FtSampleOrig = 0:FtMax/5:FtMax;

vSampleOrig = [vSampleOrig(1), mean(vSampleOrig(1:2)), vSampleOrig(2:end)];
FtSampleOrig = [FtSampleOrig(1), mean(FtSampleOrig(1:2)), FtSampleOrig(2:end)];

% The data is taken from "Jia, Y.; Jibrin, R.; Goerges, D.: Energy-Optimal
```

(continues on next page)

(continued from previous page)

```
% Adaptive Cruise Control for Electric Vehicles Based on Linear and
% Nonlinear Model Predictive Control. In: IEEE Transactions on Vehicular
% Technology, vol. 69, no. 12, pp. 14173-14187, Dec. 2020."
%     v [m/s] =        0,  2.5,    5,   10,   15,   20,   25,   30,   35,   40,   45,␣
↪  50
etaSampleOrig = [
                    0.50, 0.50, 0.50, 0.50, 0.50, 0.50, 0.50, 0.50, 0.50, 0.50, 0.50,␣
↪0.50; ... % F [N] = 0
                    0.50, 0.68, 0.80, 0.85, 0.85, 0.85, 0.85, 0.85, 0.85, 0.85, 0.83,␣
↪0.81; ... % F [N] = 500
                    0.50, 0.78, 0.81, 0.88, 0.88, 0.88, 0.88, 0.88, 0.85, 0.83, 0.81,␣
↪0.80; ... % F [N] = 1000
                    0.50, 0.75, 0.81, 0.85, 0.88, 0.85, 0.83, 0.81, 0.78, 0.68, 0.65,␣
↪0.63; ... % F [N] = 2000
                    0.50, 0.68, 0.80, 0.83, 0.83, 0.81, 0.78, 0.68, 0.65, 0.63, 0.60,␣
↪0.57; ... % F [N] = 3000
                    0.50, 0.67, 0.78, 0.81, 0.80, 0.78, 0.65, 0.63, 0.60, 0.57, 0.55,␣
↪0.55; ... % F [N] = 4000
                    0.50, 0.67, 0.75, 0.78, 0.75, 0.65, 0.63, 0.60, 0.57, 0.55, 0.52,␣
↪0.52; ... % F [N] = 5000
                ];

% convert to 1-D sequences of data points
[vTemp, FtTemp] = meshgrid(vSampleOrig, FtSampleOrig);
vSample = reshape(vTemp.', 1, []);
FtSample = reshape(FtTemp.', 1, []);
etaSample = reshape(etaSampleOrig.', [1, numel(etaSampleOrig)]);

% spline
if strcmp(spline_method, 'SMOOTH')
    s = 0.025;
    w = ones(1, length(vSample));
    bbox = [min(vSample), max(vSample), min(FtSample), max(FtSample)];
    kx = 3;
    ky = 3;
    rankTol = 1e-8;
    scaling = [1e1, 1e3, 1];
    etaMotorSpline = ForcesInterpolationFit_SmoothBivariateSpline(vSample, FtSample,␣
↪etaSample, w, bbox, kx, ky, s, rankTol, scaling);
elseif strcmp(spline_method, 'LSQ')
    tx = 30;
    ty = 3000;
    w = ones(1, length(vSample));
    bbox = [min([vSample, tx]), max([vSample, tx]), min([FtSample, ty]),␣
↪max([FtSample, ty])];
    kx = 3;
    ky = 3;
    rankTol = 1e-8;
    scaling = [1e1, 1e3, 1];
    etaMotorSpline = ForcesInterpolationFit_LSQBivariateSpline(vSample, FtSample,␣
↪etaSample, tx, ty, w, bbox, kx, ky, rankTol, scaling);
elseif strcmp(spline_method, 'COEFF')
    kx = 3;
    ky = 3;
    tx = [0, 0, 0, 0, 6.3993742001266, 24.1805094335686, 50, 50, 50, 50];
    ty = [0, 0, 0, 0, 760.320551795358, 1503.23831410745, 5000, 5000, 5000, 5000];
```

(continues on next page)

(continued from previous page)

```
    % matrix-indexed coefficient matrix
    coeffs = [0.498727471092637   0.511037494098402   0.524901875945660   0.
↪548511907792580   0.475607487652389   0.513135686437586
              0.498465143841756   0.654046675851965   0.783006359203673   0.
↪713858506960411   0.676705972846789   0.681154627267599
              0.510442836827170   0.854158083414314   1.007125597517271   0.
↪847628255554849   1.018658592375758   0.878826758082995
              0.495093430686428   0.735511289747710   0.857756122056489   0.
↪863078750613390   0.548595365620131   0.497927393614425
              0.510240152112442   0.835399001169469   0.952683895958243   0.
↪536982511482952   0.563982968586042   0.577416237725016
              0.501474795696226   0.773879473939183   0.878143062979889   0.
↪444534437467682   0.615960539904494   0.508404545245928];

    etaMotorSpline = ForcesInterpolation2D(tx, ty, coeffs, kx, ky);
elseif strcmp(spline_method, 'GRIDDED')
    xgrid = vSampleOrig;
    ygrid = FtSampleOrig;
    Z = etaSampleOrig.'; % switch indexing: cartesian -> matrix
    scaling = [1e1, 1e3, 1];
    kx = 3;
    ky = 3;
    casadiOptions = struct('degree', [kx, ky]);

    etaMotorSpline = ForcesInterpolation2DGridded(xgrid, ygrid, Z, scaling,␣
↪casadiOptions);
elseif strcmp(spline_method, 'PARAMETRIC')
    kx = 3;
    ky = 3;
    tx = [0, 0, 0, 0, 6.3993742001266, 24.1805094335686, 50, 50, 50, 50];
    ty = [0, 0, 0, 0, 760.320551795358, 1503.23831410745, 5000, 5000, 5000, 5000];
    % matrix-indexed coefficient matrix
    coeffs = [0.498727471092637   0.511037494098402   0.524901875945660   0.
↪548511907792580   0.475607487652389   0.513135686437586
              0.498465143841756   0.654046675851965   0.783006359203673   0.
↪713858506960411   0.676705972846789   0.681154627267599
              0.510442836827170   0.854158083414314   1.007125597517271   0.
↪847628255554849   1.018658592375758   0.878826758082995
              0.495093430686428   0.735511289747710   0.857756122056489   0.
↪863078750613390   0.548595365620131   0.497927393614425
              0.510240152112442   0.835399001169469   0.952683895958243   0.
↪536982511482952   0.563982968586042   0.577416237725016
              0.501474795696226   0.773879473939183   0.878143062979889   0.
↪444534437467682   0.615960539904494   0.508404545245928];

    numel_coeffs = numel(coeffs);
    etaMotorSpline = ForcesInterpolation2DParametric(numel_coeffs, tx, ty, kx, ky);
else
    error('Chosen spline_method is not valid');
end

% plotting
vMesh = 0:1:vMax;
FtMesh = 0:100:FtMax;
[vMesh, FtMesh] = meshgrid(vMesh, FtMesh);
```

(continues on next page)

(continued from previous page)

```
if strcmp(spline_method, 'PARAMETRIC')
    etaMesh = full(etaMotorSpline(vMesh(:).', FtMesh(:).', coeffs(:))); % X/Y must be␣
↪row-vectors
else
    etaMesh = full(etaMotorSpline(vMesh(:), FtMesh(:))); % sparse -> dense
end
etaMesh = reshape(etaMesh, size(vMesh, 1), size(vMesh, 2));

figure;
surf(vMesh, FtMesh, etaMesh);
xlabel("v [m/s]")
ylabel("Ft [N]")
title("Motor Efficiency")

figure;
contourf(vMesh, FtMesh, etaMesh, unique(etaSampleOrig)', 'ShowText',true)
xlabel("v [m/s]")
ylabel("Ft [N]")
title("Motor Efficiency")
```

```
import numpy as np
import matplotlib.pyplot as plt
from enum import Enum, auto
from forcespro.modelling import InterpolationFit_SmoothBivariateSpline,␣
↪InterpolationFit_LSQBivariateSpline, Interpolation2D, Interpolation2DGridded,␣
↪Interpolation2DParametric

class SplineMethod(Enum):
    SMOOTH = auto()
    LSQ = auto()
    COEFF = auto()
    GRIDDED = auto()
    PARAMETRIC = auto()

spline_method = SplineMethod.SMOOTH

vMax = 50           # maximum velocity in the lookup table [m/s]
FtMax = 5e3         # maximum traction force in the lookup table [N]

vSampleOrig = np.linspace(0, vMax, 11)
FtSampleOrig = np.linspace(0, FtMax, 6)

vSampleOrig = np.insert(vSampleOrig, 1, np.mean(vSampleOrig[0:2]))
FtSampleOrig = np.insert(FtSampleOrig, 1, np.mean(FtSampleOrig[0:2]))

# The data is taken from "Jia, Y.; Jibrin, R.; Goerges, D.: Energy-Optimal
# Adaptive Cruise Control for Electric Vehicles Based on Linear and
# Nonlinear Model Predictive Control. In: IEEE Transactions on Vehicular
# Technology, vol. 69, no. 12, pp. 14173-14187, Dec. 2020."
#     v [m/s] =         0,  2.5,    5,   10,   15,   20,   25,   30,   35,   40,   45,
↪   50
etaSampleOrig = [
                    [0.50, 0.50, 0.50, 0.50, 0.50, 0.50, 0.50, 0.50, 0.50, 0.50, 0.50,
↪ 0.50], # F [N] = 0
                    [0.50, 0.68, 0.80, 0.85, 0.85, 0.85, 0.85, 0.85, 0.85, 0.85, 0.83,
↪ 0.81], # F [N] = 500
```

(continues on next page)

(continued from previous page)

```
                    [0.50, 0.78, 0.81, 0.88, 0.88, 0.88, 0.88, 0.88, 0.85, 0.83, 0.81,
↪ 0.80], # F [N] = 1000
                    [0.50, 0.75, 0.81, 0.85, 0.88, 0.85, 0.83, 0.81, 0.78, 0.68, 0.65,
↪ 0.63], # F [N] = 2000
                    [0.50, 0.68, 0.80, 0.83, 0.83, 0.81, 0.78, 0.68, 0.65, 0.63, 0.60,
↪ 0.57], # F [N] = 3000
                    [0.50, 0.67, 0.78, 0.81, 0.80, 0.78, 0.65, 0.63, 0.60, 0.57, 0.55,
↪ 0.55], # F [N] = 4000
                    [0.50, 0.67, 0.75, 0.78, 0.75, 0.65, 0.63, 0.60, 0.57, 0.55, 0.52,
↪ 0.52]  # F [N] = 5000
                  ]
etaSampleOrig = np.array(etaSampleOrig)

# convert to 1d sequences of data points
[vTemp, FtTemp] = np.meshgrid(vSampleOrig, FtSampleOrig)
vSample = vTemp.flatten()
FtSample = FtTemp.flatten()
etaSample = etaSampleOrig.flatten()

# spline
if spline_method == SplineMethod.SMOOTH:
    s = 0.025
    w = np.ones((1, len(vSample)))
    bbox = [min(vSample), max(vSample), min(FtSample), max(FtSample)]
    kx = 3
    ky = 3
    eps = 1e-8
    scaling = [1e1, 1e3, 1]
    etaMotorSpline = InterpolationFit_SmoothBivariateSpline(x=vSample, y=FtSample,␣
↪z=etaSample, w=w, bbox=bbox, kx=kx, ky=ky, s=s, eps=eps, scaling=scaling)
elif spline_method == SplineMethod.LSQ:
    tx = [30.]
    ty = [3000.]
    w = np.ones((1, len(vSample)))
    bbox = [min(np.append(vSample, tx)), max(np.append(vSample, tx)), min(np.
↪append(FtSample, ty)), max(np.append(FtSample, ty))]
    kx = 3
    ky = 3
    eps = 1e-8
    scaling = [1e1, 1e3, 1]
    etaMotorSpline = InterpolationFit_LSQBivariateSpline(x=vSample, y=FtSample,␣
↪z=etaSample, tx=tx, ty=ty, w=w, bbox=bbox, kx=kx, ky=ky, eps=eps, scaling=scaling)
elif spline_method == SplineMethod.COEFF:
    kx = 3
    ky = 3
    tx = np.array([0, 0, 0, 0, 6.3993742001266, 24.1805094335686, 50, 50, 50, 50])
    ty = np.array([0, 0, 0, 0, 760.320551795358, 1503.23831410745, 5000, 5000, 5000,␣
↪5000])
    coeffs = np.array([
                        [0.498727471092637,   0.511037494098402,   0.524901875945660,␣
↪  0.548511907792580,   0.475607487652389,   0.513135686437586],
                        [0.498465143841756,   0.654046675851965,   0.783006359203673,␣
↪  0.713858506960411,   0.676705972846789,   0.681154627267599],
                        [0.510442836827170,   0.854158083414314,   1.007125597517271,␣
↪  0.847628255554849,   1.018658592375758,   0.878826758082995],
                        [0.495093430686428,   0.735511289747710,   0.857756122056489,␣
```

(continues on next page)

(continued from previous page)

```
↪  0.863078750613390,   0.548595365620131,   0.497927393614425],
                      [0.510240152112442,   0.835399001169469,   0.952683895958243,␣
↪  0.536982511482952,   0.563982968586042,   0.577416237725016],
                      [0.501474795696226,   0.773879473939183,   0.878143062979889,␣
↪  0.444534437467682,   0.615960539904494,   0.508404545245928]
                     ])

    etaMotorSpline = Interpolation2D(tx, ty, coeffs, kx, ky)
elif spline_method == SplineMethod.GRIDDED:
    xgrid = vSampleOrig
    ygrid = FtSampleOrig
    Z = etaSampleOrig.T # switch indexing: cartesian -> matrix
    scaling = [1e1, 1e3, 1]
    kx = 3
    ky = 3
    casadiOptions = {'degree': (kx, ky)}

    etaMotorSpline = Interpolation2DGridded(xgrid, ygrid, Z, scaling, casadiOptions)
elif spline_method == SplineMethod.PARAMETRIC:
    kx = 3
    ky = 3
    tx = np.array([0, 0, 0, 0, 6.3993742001266, 24.1805094335686, 50, 50, 50, 50])
    ty = np.array([0, 0, 0, 0, 760.320551795358, 1503.23831410745, 5000, 5000, 5000,␣
↪5000])
    coeffs = np.array([
                      [0.498727471092637,   0.511037494098402,   0.524901875945660,␣
↪  0.548511907792580,   0.475607487652389,   0.513135686437586],
                      [0.498465143841756,   0.654046675851965,   0.783006359203673,␣
↪  0.713858506960411,   0.676705972846789,   0.681154627267599],
                      [0.510442836827170,   0.854158083414314,   1.007125597517271,␣
↪  0.847628255554849,   1.018658592375758,   0.878826758082995],
                      [0.495093430686428,   0.735511289747710,   0.857756122056489,␣
↪  0.863078750613390,   0.548595365620131,   0.497927393614425],
                      [0.510240152112442,   0.835399001169469,   0.952683895958243,␣
↪  0.536982511482952,   0.563982968586042,   0.577416237725016],
                      [0.501474795696226,   0.773879473939183,   0.878143062979889,␣
↪  0.444534437467682,   0.615960539904494,   0.508404545245928]
                     ])


    numel_coeffs = coeffs.size
    etaMotorSpline = Interpolation2DParametric(numel_coeffs=numel_coeffs, tx=tx,␣
↪ty=ty, kx=kx, ky=ky)
else:
    raise ValueError("Chosen spline_method is not valid")

# plotting
vMesh = np.linspace(0, vMax, 51)
FtMesh = np.linspace(0, FtMax, 51)
vMesh, FtMesh = np.meshgrid(vMesh, FtMesh)

if spline_method == SplineMethod.PARAMETRIC:
    etaMesh = np.array(etaMotorSpline(np.expand_dims(vMesh.flatten(), 0), np.expand_
↪dims(FtMesh.flatten(), 0), coeffs.flatten('F'))).squeeze() # X/Y must be row-vectors
else:
    etaMesh = np.array(etaMotorSpline(vMesh.flatten(), FtMesh.flatten())).squeeze()
```

(continues on next page)

(continued from previous page)

```
etaMesh = etaMesh.reshape(vMesh.shape)

fig = plt.figure(figsize=(8, 6))
ax = plt.axes(projection="3d")
surf = ax.plot_surface(vMesh, FtMesh, etaMesh, cmap='viridis', edgecolor='black',␣
↪linewidth=0.5)
ax.set_xlabel("v [m/s]")
ax.set_ylabel("Ft [N]")
ax.set_title("Motor Efficiency")

fig = plt.figure(figsize=(8, 6))
cnt = plt.contour(vMesh, FtMesh, etaMesh, levels=np.unique(etaSampleOrig), colors=
↪"black")
plt.clabel(cnt, colors="black")
plt.contourf(vMesh, FtMesh, etaMesh, levels=np.unique(etaSampleOrig))
plt.xlabel("v [m/s]")
plt.ylabel("Ft [N]")
plt.title("Motor Efficiency")

plt.show()
```

20.3 Smooth Approximations

There are a number of useful basic functions that are not differentiable everywhere. For some of them FORCESPRO provides a built-in smooth approximation and we plan to add more in an upcoming release.

20.3.1 Smooth Minimum

The minimum value of two scalars is not differentiable at the points where both values are identical. You can use the following smooth approximation instead:

Matlab

Python

```
c = ForcesMin(a, b);
```

```
c = forcespro.modelling.smooth_min(a, b)
```

This function accepts an optional third argument to trade-off smoothness and approximation quality. The default value is set to `1e-8`; higher values make the function smoother but less accurate.

20.3.2 Smooth Maximum

The maximum value of two scalars is not differentiable at the points where both values are identical. You can use the following smooth approximation instead:

Matlab

Python

```
c = ForcesMax(a, b);
```

```
c = forcespro.modelling.smooth_max(a, b)
```

This function accepts an optional third argument to trade-off smoothness and approximation quality. The default value is set to `1e-8`; higher values make the function smoother but less accurate.

Chapter 21

Dumping Problem Formulation and Data

- *Why to use the dump tool?*
- *How to use the dump tool?*
  - *Quick reference*
  - *Symbolic dumps*
  - *Legacy dumps*
  - *Problem dumps from C*

21.1 Why to use the dump tool?

Along with its clients, FORCESPRO provides a tool that allows the user to dump the formulation and actual data of an optimization problem. This information allows to exactly reproduce the same solver for a given formulation and to feed it with exactly the same data to yield exactly the same results (provided it is run on the very same target hardware). The problem formulation and data is stored in "stand-alone" `mat` or `json` files, and the problem data can also be saved in binary format. This means there is no need to keep copies of other files that may be used to specify the formulation (such as the dynamic equations), except for formulations relying on external callbacks provided as C code (see *External function evaluations in C*).

The dump tool may be helpful for a couple of use cases such as:

- *Debugging:* a dumped problem allows you to re-run single solver calls without the need to have your full simulation environment up and running.
- *External support:* you may send a dumped problem to whomever is in charge of providing support and it will enable that person to exactly reproduce your issue.
- *Testing:* keeping dumps of problems that performed as expected can be used to run regression tests to ensure they work as expected after future changes.

Note that, depending on the dump type you choose (see *How to use the dump tool?*), the dump tool either stores your problem formulation on a symbolic level or keeps a copy of the *C code generated by the automatic differentiation tool*. Thus, keep the following in mind:

Important: A dumped problem will contain complete information about the solver that

you have setup. In particular, it may be used to reverse-engineer your problem formulation (including dynamic model, objective function, constraints etc.). Thus, only share a dumped problem with persons that have a right to obtain this information.

21.2 How to use the dump tool?

The dump tool currently provides three different dump types. Section *Symbolic dumps* describes the default *symbolic dump* tool which stores the full symbolic formulation, but requires CasADi v3.5.x to work. Section *Legacy dumps* describes the so-called *legacy dump* that is available in the MATLAB client only and does not store the full symbolic formulation. In section *Problem dumps from C*, you learn how to dump a problem struct (containing the runtime parameters) from C. This is useful when you work on the embedded system and you don't want to use the MATLAB and Python interface for dumping.

Note: In the Python client, code generation and dumping are not interchangeable, as code generation mutates the *CodeOptions* and *SymbolicModel* objects.

To get around this issue, use Python's *copy.deepcopy* function to make a copy which will not affected by code generation.

Note, that calling *copy.copy* makes a shallow copy and will contain references to the original object and will thus also mutate.

21.2.1 Quick reference

This section serves as a quick reference for all client functions related to the dump tool. Please read further below (sections *Symbolic dumps* and *Legacy dumps*) to find documentation on how to use the dump tool in context.

Matlab

In *Matlab*, the following functions may be used to dump and load a formulation:

- High-level interface:
  - `ForcesDumpFormulation`: dump formulation (symbolic & legacy dumps)
  - `ForcesDumpProblem`: dump problem data (symbolic & legacy dumps)
  - `ForcesDumpAll`: dump formulation and problem data (symbolic & legacy dumps)
  - `ForcesLoadSymbolicDump`: load formulation and problem data (symbolic dumps)
  - `ForcesFindDumpedProblems` find dumped formulation and problem data (symbolic & legacy dumps)
- Low-level interface (symbolic dumps only):
  - `ForcesDumpFormulationLowLevel`: dump formulation
  - `ForcesDumpProblemLowLevel`: dump problem data
  - `ForcesDumpAllLowLevel`: dump formulation and problem data
  - `ForcesLoadLowLevelDump`: load formulation and problem data
  - `ForcesFindDumpedProblems`: find dumped formulation and problem data

Please refer to the `help` text of the respective function (by e.g. typing `help(ForcesDumpFormulation)`) for specific documentation on how to use these functions.

Python

In Python, the following functions may be used to dump and load a formulation:

- `forcespro.dump.save_formulation`: dump formulation
- `forcespro.dump.save_problem`: dump problem data
- `forcespro.dump.save_all`: dump formulation and problem data
- `forcespro.dump.load`: load formulation and problem data
- `forcespro.dump.find_problems`: find dumped formulation and problem data

Please refer to the docstring text of the respective function (by e.g. typing `help(forcespro.dump.save_formulation)`) for specific documentation on how to use these functions.

21.2.2 Symbolic dumps

Symbolic dumps directly store symbolic expressions of your problem formulation and codeoptions after converting both into the text-based JSON format. From FORCESPRO v6.1.0 onwards, this is the default dump tool if not otherwise specified. Note that this variant reveals your complete problem formulation to anybody with whom you share those JSON files! While you should thus handle those symbolic dumps with care, they offer more flexibility than the legacy dumps and are also available via the Python client of FORCESPRO.

Creating a symbolic dump of a problem consists of two steps:

1. *Dumping the problem formulation:* you need to store your `model` or `stages` struct, the `codeoptions` struct and optionally the `outputs` struct, which can be done even before generating the actual solver code.
2. *Dumping problem data:* for each problem instance, the problem `params` struct needs to be stored. It is possible to store data of multiple problem instances for the same problem formulation (in either a single file or multiple files).

Both steps may also be performed at once.

Dumping the problem formulation

For dumping the problem formulation in a symbolic way, just call the following function:

Matlab

Python

```
% For the high-level interface:
[tag, fullFilename] = ForcesDumpFormulation( model,codeoptions,outputs,...
                              label,dumpDirectory );

% For the low-level interface:
[tag, fullFilename] = ForcesDumpFormulationLowLevel( stages,codeoptions,outputs,....
                              params,label,dumpDirectory );
```

```
# For the high-level interface:
tag,full_filename = forcespro.dump.save_formulation( modelOrStages, codeoptions,␣
↪outputs,
                                                    label=label, path=dump_directory )

# For the low-level interface:
tag,full_filename = forcespro.dump.save_formulation( modelOrStages, label=label,␣
↪path=dump_directory )
```

In MATLAB, the last argument enables the use of a symbolic dump. This parameter is not required in Python since there is only the symbolic way to dump problems. In Python, the argument `modelOrStages` refers to the `model` or `stages` object, for the high-level interface or the low-level interface, respectively. Pass `outputs` if your problem formulation contains outputs. Moreover, you may pass an additional `label` used inside the filenames (or pass an empty string) and `dumpDirectory` (keyword `path` in Python) for storing the dumped file (the default is the current working directory). When called this way, the function `ForcesDumpFormulation` will create a `json` file in the specified directory containing the passed information. The filename is automatically chosen and will contain the name of your solver, your label, a unique tag, a timestamp as well as the suffix `_F`, e.g. `myFORCESsolver_ABC3DEFGHIJ_20200101120000000_F.json`. The returned string `fullFilename` consists of the dump directory and the filename of the dump.

Note that this function returns a `tag` that is unique for a given formulation and code options. It is strongly recommended to use it when dumping corresponding problem data. However, the MATLAB and the Python client generate different tags for the same mathematical formulation.

Note that for Python low-level dumps, `options` and `outputs` must be set to `None` (default), as these should be included in the `stages` object.

Note: For Python low-level dumps, code generation and dumping are not interchangeable, as code generation mutates the *CodeOptions* object.

To get around this issue, use Python's *copy.deepcopy* function to make a copy which will not be affected by code generation.

Note: Also note that formulations making use of CasADi MX expressions (see *Automatic differentiation expression class*) currently cannot be dumped.

Dumping problem data

Assuming your generated FORCESPRO solver is called `myFORCESsolver` and you are calling it with the following command

Matlab

Python

```
[output, exitflag, info] = myFORCESsolver( problem );
```

```
output, exitflag, info = my_forces_solver.solve(problem);
```

then dumping the problem data of any problem instance is as simple as calling

Matlab

Python

```
% For the high-level interface:
fullFilename = ForcesDumpProblem( problems,tag,dumpDirectory );

% For the low-level interface:
fullFilename = ForcesDumpProblemLowLevel( problems,tag,dumpDirectory );
```

```
full_filename = forcespro.dump.save_problem(problems, tag, dump_directory)
```

You can pass as *problems* either a single problem parameter struct (dictionary in Python) or a cell array (list in Python) of problem parameter structs. Besides, the unique `tag` that has been generated when dumping the problem formulation is required. The third argument `dumpDirectory` for storing the dumped problem data is optional (with the default being the current working directory). The functions `ForcesDumpProblem` or `forcespro.dump.save_formulation` will create a `json` file in the specified directory containing the passed information. The filename is automatically chosen and will contain the name of your solver, the unique tag (including any label passed when dumping the formulation), a timestamp as well as the suffix `_P`, e.g. `myFORCESsolver_ABC3DEFGHIJ_20200101120001000_P.json`. The returned string `fullFilename` consists of the dump directory and the filename of the dump.

There is no limit on the number of problem instances that you may dump that way.

Dumping the problem formulation and data at once

For dumping both the problem formulation and all problem data at once in a symbolic way to a single `json` file, just call the following function:

Matlab

Python

```
% For the high-level interface:
[tag, fullFilename] = ForcesDumpAll( model,codeoptions,outputs,...
                      label,dumpDirectory,problems );

% For the low-level interface:
[tag, fullFilename] = ForcesDumpAllLowLevel( stages,codeoptions,outputs,...
                     params,label,dumpDirectory,problems );
```

```
# For the high-level interface:
tag, full_filename = forcespro.dump.save_all( modelOrStages, codeoptions, outputs, \
                       label=label, path=dump_directory, problems=problems )

# For the low-level interface:
tag, full_filename = forcespro.dump.save_all( modelOrStages, \
                       label=label, path=dump_directory, problems=problems )
```

The arguments and behavior are the same as for the functions used to dump formulation and problem separately (see *Dumping the problem formulation* and *Dumping problem data*). The only difference is that `formulation` and `problems` are dumped to a single file with suffix `_A`, e.g. `myFORCESsolver_ABC3DEFGHIJ_20200101120000000_A.json`

Running a dumped problem

After you have dumped a problem formulation and at least one set of problem data, you can use either a matching pair of `_F`/`_P` files or a single `_A` file in JSON format to exactly reproduce your solver and problem instances. To do so, you need to perform the following two steps:

1. *Load problem formulation and data from JSON file or files* by calling:

   Matlab

   Python

```
% For the high-level interface:
[model, codeoptions, outputs, additionalData, problems] = ...
              ForcesLoadSymbolicDump( formulationFilename,problemFilenames␣
↪);

% For the low-level interface:
[stages, codeoptions, outputs, params, problems] = ForcesLoadLowLevelDump(␣
↪formulationFilename, problemFilenames );
```

```
modelOrStages, options, outputs, additional, problems = \
              forcespro.dump.load(formulation_filename, problem_filename)
```

   `problemFilenames` may either be a single file name or a cell array (list in Python) containing all the problem data set that you want to load. In case you have dumped both formulation and problem data set(s) at once within a single file, just pass that one as `formulationFilename` and do not specify `problemFilenames`. The returned `problems` variable is an array (list in Python) containing all problem sets found in the dumps.

2. *Re-generating and running the solver with dumped information* by simply calling:

   Matlab

   Python

```
% For the high-level interface:
FORCES_NLP( model,codeoptions,outputs );

% For the low-level interface:
generateCode( stages,params,codeoptions,outputs );

myFORCESsolver( problems(1) );
% and more problem instances if present
```

```
# For the high-level interface:
solver = model.generate_solver(codeoptions, outputs)
result, exitflag, info = solver.solve(problems[1])
# and more problem instances if present

# For the low-level interface:
stages.generateCode(get_userid.userid)
import my_forces_solver_py
result, exitflag, info = my_forces_solver_py.my_forces_solver_
↪solve(problems[1])
# and more problem instances if present
```

Tip: To get the filenames of dumped problems and formulations in a directory, simply use the function:

Matlab

Python

```
[formulationFilename, problemFilename] = ForcesFindDumpedProblems( tag,dumpDirectory␣
↪);
```

```
formulation_filename, problem_filenames = forcespro.dump.find_problems(tag, dump_
↪directory)
```

Both arguments are optional. If `tag` is not given, the function returns any dumped filenames regardless of their tag. The default `dumpDirectory` is the current working directory. `problemFilenames` is a cell array (list in Python) of problem filenames.

Important: MATLAB support for low-level dumps has been added in FORCESPRO version `6.1.0`, and low-level dumps created with an earlier version (from Python) can't be loaded in MATLAB.

Limitations of symbolic dumps

Symbolic dumps only work with CasADi v3.5.x for reasons beyond our control, which is why we currently do not plan to extend support to CasADi v2.4.2 or MathWorks' Symbolic Math Toolbox.

21.2.3 Legacy dumps

Legacy dumps are available for the MATLAB client only and store a pre-processed problem formulation including C code generated by the automatic differentiation tool. This variant is somewhat less explicit and is supposed to work with all supported AD tools.

Creating a legacy dump of a problem consists of two steps:

1. *Dumping the problem formulation:* once a new solver has been generated, a `formulation` struct, the `codeoptions` struct and optionally the `outputs` struct need to be stored.
2. *Dumping problem data:* for each problem instance, the problem `params` struct needs to be stored. It is possible to store data of multiple problem instances for the same problem formulation.

Dumping the problem formulation

For dumping the problem formulation, the following three steps need to be taken:

1. *Enabling creation of a formulation dump:* This is done by using the option

   ```
   codeoptions.dump_formulation = 1;
   ```

2. *Obtaining the dumped formulation:* Calling `FORCES_NLP` with the before-mentioned code option enabled will make it return a `formulation` struct as *third* output argument

   ```
   [stages, codeoptions, formulation] = FORCES_NLP( model, codeoptions, outputs );
   ```

3. *Storing the necessary structs into a file:* After calling `FORCES_NLP`, you should use the following function to store both the `formulation` and `codeoptions` struct

   ```
   [tag, fullFilename] = ForcesDumpFormulation( formulation,codeoptions,outputs,
   ↪label,dumpDirectory,ForcesDumpType.LegacyDumpGeneratedC );
   ```

Pass `outputs` if your problem formulation contains outputs. Moreover, you may pass an additional `label` used inside the filenames (or pass an empty string) and `dumpDirectory` for storing the dumped formulation (the default is the current working directory). The function `ForcesDumpFormulation` will create a `mat` file in the specified directory containing the passed information. The filename is automatically chosen and will contain the name of your solver, your label, a unique tag, a timestamp as well as the suffix `_F`, e.g. `myFORCESsolver_ABC3DEFGHIJ_20200101120000000_F.mat`. The returned `fullFilename` is a string consisting of the directory and the filename of the dump.

Note that this function returns a `tag` that is unique for a given formulation and code options. It is strongly recommended to use it when dumping corresponding problem data. Note that the MATLAB and the Python client generate different tags for the same mathematical formulation.

Dumping problem data

Assuming your generated FORCESPRO solver is called `myFORCESsolver` and you are calling it with the following command

```
[output, exitflag, info] = myFORCESsolver( problem );
```

then dumping the problem data of any problem instance is as simple as calling

```
fullFilename = ForcesDumpProblem( problem,tag,dumpDirectory,ForcesDumpType.
↪LegacyDumpGeneratedC );
```

Here, you need to provide both the problem parameter struct as well as the unique `tag` that has been generated when dumping the problem formulation. The third argument `dumpDirectory` for storing the dumped problem data is optional (with the default being the current working directory). The function `ForcesDumpProblem` will create a `mat` file in the specified directory containing the passed information. The filename is automatically chosen and will contain the name of your solver, the unique tag (including any label passed when dumping the formulation), a timestamp as well as the suffix `_P`, e.g. `myFORCESsolver_ABC3DEFGHIJ_20200101120001000_P.mat`. The returned `fullFilename` is a string consisting of the directory and the filename of the dump.

There is no limit on the number of problem instances that you may dump that way.

Running a dumped problem

After you have dumped a problem formulation and at least one set of problem data, you can use those `mat` files to exactly reproduce your solver and problem instances. To do so, you need to perform the following two steps (where we assume you have stored the two files `myFORCESsolver_ABC3DEFGHIJ_20200101120000000_F.mat` and `myFORCESsolver_ABC3DEFGHIJ_20200101120001000_P.mat` at a location in your MATLAB path):

1. *Re-generate the FORCESPRO solver* by loading the formulation `mat` file and using its content to call the code generation:

   ```
   F = load('myFORCESsolver_ABC3DEFGHIJ_20200101120000000_F.mat');
   FORCES_NLP( F.formulation,F.codeoptions,F.outputs );
   ```

   This will re-create the solver MEX function `myFORCESsolver`. Note that the third input struct containing the `outputs` is only available if you included it into your dump.

2. *Running the solver with dumped problem data* by loading the data `mat` file and using its content to call the generated solver:

```
P = load('myFORCESsolver_ABC3DEFGHIJ_20200101120001000_P.mat');
myFORCESsolver( P.problem );
```

You may repeat this step for as many problem instances as you have dumped.

Tip: To get the filenames of dumped problems and formulations in a directory, simply use the function:

```
[formulationFilename, problemFilename] = ForcesFindDumpedProblems( tag,dumpDirectory␣
↪);
```

Both arguments are optional. If `tag` is not given, the function returns any dumped filenames regardless of their tag. The default `dumpDirectory` is the current working directory. `problemFilenames` is a cell array of problem filenames.

Limitations of legacy dumps

Legacy dumps have the following limitations:

- They are only available via the MATLAB client of FORCESPRO.
- They cannot be used if you pass external functions in form of C code.
- They cannot be used in combination with CasADi MX expressions (see *Automatic differentiation expression class*).

These limitations can be overcome by using a symbolic dump (see *Symbolic dumps*).

21.2.4 Problem dumps from C

Problem dumps from C consider only the `params` (or `problem`) struct containing the runtime parameters that are passed to the solver in each solver call. In order to save or load `params` (or `problem`) structs, you can call the dump tool from any C script. This offers the opportunity to work directly on the embedded system. The data is stored in binary format. The problem dumps from C use the *msgpack-c* library.

Important: The problem dumps from C require dynamic memory allocation. This is because of the dependency on the *msgpack-c* library. Please check if your embedded platform supports dynamic memory allocation.

Before dumping problems from C for the first time, you need to install msgpack as described in *Download and install msgpack for C*. Then, the dumping procedure consists of the following three steps:

1. *Enabling the generation of the dump functions:* When generating your solver, you need to set a codeoption in order to enable the generation of the dump functions.
2. *Dumping problem data:* Call the generated *serialize* function in order to dump a `params` struct. Exactly one `params` struct can be stored at a time.
3. *Loading problem data:* Call the generated *deserialize* function in order to load a `params` struct. Exactly one `params` struct can be stored at a time.

Download and install msgpack for C

Since the dump tool for problems from C requires msgpack, make sure you installed the library. You can either clone the github repository or you can simply download msgpack as a zip file. For the installation of the library, you need `gcc >= 4.1.0` and `cmake >= 2.8.1`.

How to install:

- **Windows:**
  1. Run `cmake .` in your terminal from the *msgpack-c-c_master* folder.
  2. Open the generated *msgpack.sln* file (located in the same folder) in Visual Studio and click *Build*. The generated library is located in *msgpack-c-c_master/Debug*.
- **Other platforms:**
  1. Run `cmake .` in your terminal from the *msgpack-c-c_master* folder.
  2. Run `make`. The generated library is located in *msgpack-c-c_master*.

For more information about the installation of msgpack, see the *README.md* on github.

Enable the Generation of the Dump Functions

Before generating your solver you need to enable the generation of the dump functions. You can specify whether you want to dump from your host platform or/and from your target platform. For dumping from the host platform set:

Matlab

Python

```
codeoptions.serializeCParamsHost = 1;
```

```
codeoptions.serializeCParamsHost = 1
```

and for dumping from your specified target platform set:

Matlab

Python

```
codeoptions.serializeCParamsTarget = 1;
```

```
codeoptions.serializeCParamsTarget = 1
```

Dumping Problem Data

This section explains how to write and run a C script that dumps your problem data based on the high-level basic example (see *High-level interface: Basic example*). We assume your C script is in the same folder as your generated solver.

You can find the code of this example script to try it out for yourself in the `examples` folder that comes with your client.

1. Include your solver header called `<solvername>.h`. In the solver header, the `<solvername>_params` struct and the dumping routines `<solvername>_serialize` and `<solvername>_deserialize` are defined. Note that the `params` struct for binary problems is called `<solvername>_binaryparams` and for mixed integer problems it is `<solvername>_integerparams`.

```
#include "FORCESNLPsolver/include/FORCESNLPsolver.h"
```

2. Create a `params` struct and fill it with your data:

```
/* create params struct */
FORCESNLPsolver_params params;

/* fill params struct with data */
params.xinit[0] = -4.;
params.xinit[1] = 2.;
for (int i = 0; i < 33; i++)
{
    params.x0[i] = 0.0;
}
```

3. Choose a filename for the dump and call the serialization routine:

```
const char filename[] = "dump.msgpack";
int returnvalueSerialize = FORCESNLPsolver_serialize(&params, filename);
```

4. Compile your script:

```
$ <Compiler_exec> my_C_dump_script.c <compiled_solver> -L<msgpack_lib_path>␣
↪-l<msgpack_lib> <additional_libs>
```

Where:

- `<Compiler_exec>` is your compiler (for example `gcc`)
- `my_C_dump_script.c` is your script that calls the `serialize` function (for example `serializationCParams_HighLevel_BasicExample.c`)
- `<compiled_solver>` is your compiled solver library (static or shared):
  - For Linux/MacOS/MinGW it is `libFORCESNLPsolver.a` or `libFORCESNLPsolver.so` in the `lib` or `lib_target` directory
  - For Windows it is `FORCESNLPsolver_static.lib` or `FORCESNLPsolver.lib` in the `lib` or `lib_target` directory
- `<msgpack_lib_path>` specifies your path to the compiled msgpack library
  - For Linux/MacOS/MinGW it is the path to your `msgpack-c-c_master` folder
  - For Windows it is the path to your `msgpack-c-c_master/Debug` folder
- `<msgpack_lib>` specifies the name of the compiled msgpack library
  - For the static library on Windows set it to `msgpackc_import`
  - Otherwise, it is `msgpackc`
- `<additional_libs>` are possible libraries that need to be linked to resolve existing dependencies.
  - For Linux/MacOS it's usually necessary to link the math library (`-lm`)
  - For Windows you usually need to link the `iphlpapi.lib` library (it's distributed with the Intel Compiler, MinGW as well as MATLAB) and sometimes some additional intel libraries (those are included in the FORCESPRO client under the folder `libs_Intel` – if missing they are downloaded after code generation)

The following shows how to compile the dump example script on Linux:

```
$ gcc serializationCParams_HighLevel_BasicExample.c FORCESNLPsolver/lib/
↪libFORCESNLPsolver.so -L/path/to/msgpack-c -lmsgpackc -lm
```

Loading Problem Data

This section explains how to write and run a C script that loads a dumped C `params` struct based on the high-level basic example (see *High-level interface: Basic example*). We assume your C script is in the same folder as your generated solver.

You can find the code of this example script to try it out for yourself in the `examples` folder that comes with your client.

1. Include your solver header called `<solvername>.h`.

```
#include "FORCESNLPsolver/include/FORCESNLPsolver.h"
```

2. Create an empty `params` struct:

```
FORCESNLPsolver_params dumped_params;
```

3. Call the deserialization routine and pass the filename you chose when dumping the problem data:

```
int returnvalueDeserialize = FORCESNLPsolver_deserialize(&dumped_params,␣
↪filename);
```

4. Compile your script:

```
$ <Compiler_exec> my_C_dump_script.c <compiled_solver> -L<msgpack_lib_path>␣
↪-l<msgpack_lib> <additional_libs>
```

   Where:

   - `<Compiler_exec>` is your compiler (for example `gcc`)
   - `my_C_dump_script.c` is your script that calls the `deserialize` function (for example `serializationCParams_HighLevel_BasicExample.c`)
   - `<compiled_solver>` is your compiled solver library (static or shared):
     - For Linux/MacOS/MinGW it is `libFORCESNLPsolver.a` or `libFORCESNLPsolver.so` in the `lib` or `lib_target` directory
     - For Windows it is `FORCESNLPsolver_static.lib` or `FORCESNLPsolver.lib` in the `lib` or `lib_target` directory
   - `<msgpack_lib_path>` specifies your path to the compiled msgpack library
     - For Linux/MacOS/MinGW it is the path to your `msgpack-c-c_master` folder
     - For Windows it is the path to your `msgpack-c-c_master/Debug` folder
   - `<msgpack_lib>` specifies the name of the compiled msgpack library
     - For the static library on Windows set it to `msgpackc_import`
     - Otherwise, it is `msgpackc`
   - `<additional_libs>` are possible libraries that need to be linked to resolve existing dependencies.
     - For Linux/MacOS it's usually necessary to link the math library (`-lm`)

- For Windows you usually need to link the `iphlpapi.lib` library (it's distributed with the Intel Compiler, MinGW as well as MATLAB) and sometimes some additional intel libraries (those are included in the FORCESPRO client under the folder `libs_Intel` – if missing they are downloaded after code generation)

The following shows how to compile the dump example script on Linux:

```
$ gcc serializationCParams_HighLevel_BasicExample.c FORCESNLPsolver/lib/
↪libFORCESNLPsolver.so -L/path/to/msgpack-c -lmsgpackc -lm
```

Dumping and Loading Problem Data from the MATLAB Client

It is possible to dump a `params` (or `problem`) struct into the *msgpack* format, so that it can be loaded from a FORCESPRO solver in C, directly from the MATLAB client. Likewise, a `params` struct dumped from C can be loaded from the MATLAB client by enabling the following codeoption:

Matlab

Python

```
codeoptions.serializeCParamsMex = 1;
% need to set codeoptions.serializeCParamsHost = 1 as well
```

```
# not yet supported
```

This enables generation of two additional MEX files `<solvername>_serialize_mex.c` and `<solvername>_deserialize_mex.c` within the `interface` folder of the generated solver. Additionally, the client tries to MEX compile those two files assuming a *msgpack-c* library is available. If this library cannot be found on your system, that auto-compilation will fail and you need to MEX compile those functions manually.

Note: Those two generated MEX files for (de)serialization are specific to a given FORCESPRO solver and *cannot* be used to (de)serialize `params` structs of any other FORCESPRO solver.

Assuming those two MEX functions have been successfully generated and compiled, you can dump a MATLAB `params` struct into a file in *msgpack* format named `filename` as follows:

```
success = <solvername>_serialize_mex(params, filename);
```

`success` will be `1` if the `params` struct has been successfully serialized, and `0` otherwise (which is different from the `returnvalue` returned by the corresponding plain C function).

Similarly, a file named `filename` containing a `params` struct in *msgpack* format can be loaded into a MATLAB struct `loadedParams` by calling

```
[success, loadedParams] = <solvername>_deserialize_mex(filename);
```

Note that `loadedParams` will contain meaningful data only if `success` is `1` (which is different from the `returnvalue` returned by the corresponding plain C function)!

Dumping and Loading Problem Data from the Python Client

It is possible to dump a `problem` (or `params`) dictionary into the *msgpack* format, so that it can be loaded from a FORCESPRO solver in C, directly from the Python client. This functionality is automatically included into any generated FORCESPRO solver and can be called as follows:

```
# assumes a solver object as returned by the function generate_solver
solver.serialize_c_params(problem, filename)
```

This line of code will store a `problem` dictionary into a file in *msgpack* format named `filename`. Rather than returning a success flag, the Python client would throw an exception in case of an error.

Likewise, a `problem` struct dumped from C can be loaded from the Python client by calling:

```
# assumes a solver object as returned by the function generate_solver
loaded_problem = solver.deserialize_c_params(filename)
```

This loads the struct into the dictionary `loaded_problem` assuming that the file called `filename` contains a `problem` struct in *msgpack* format.

Note: Serialization and deserialization of C parameters from Python requires the `msgpack` package to be available in your Python installation, otherwise an exception will be thrown.

Chapter 22

Webcompilers

- *Webcompilers in FORCESPRO*
- *Webcompilers versioning*
  - *Webcompilers with legacy infrastructure*
- *Ensuring use of webcompiler version*

22.1 Webcompilers in FORCESPRO

The online FORCESPRO code generation service is providing solver binaries which are configured to run in the platforms supported by FORCESPRO. To build these binaries and configure them properly, webcompiler services (developed and maintained as part of FORCESPRO and hosted in the FORCESPRO infrastructure) are used which contain configurations and compilers for each platform.

22.2 Webcompilers versioning

In general, the webcompiler services are aligned with the FORCESPRO versioning in order to ensure each FORCESPRO version can always generate the solvers required as well as to ensure that the same FORCESPRO version will continue producing binaries with the same configuration even when new webcompiler services or FORCESPRO versions are released.

Since FORCESPRO `6.1.0`, the generation of solver binaries for all platforms, except the ones specified *here*, is using a new infrastructure in order to achieve better scalability and service availability. The version of the webcompilers in the new infrastructure still continues to be aligned with the FORCESPRO version.

The webcompiler service version in use for a FORCESPRO version is always ensured to be compatible with the used FORCESPRO version. However, e.g. as a result of security maintenance, the webcompiler version used in the new infrastructure may change. While this will not affect compatibility with the used FORCESPRO version, a different webcompiler version could result in a different configuration for the generated binaries and as a result potentially different numerics. If this is critical for a specific application, the version to be used can be explicitly requested as shown here: *Ensuring use of webcompiler version*.

The webcompiler version used for the generation of a solver is directly accessible in the solver in its header file. The header file contains two comments:

```
/* Host Compiler Version: <compiler version> */
/* Target Compiler Version: <compiler version> */
```

These refer to the versions of the webcompiler services used for the compilation of the host solver and the target solver respectively and will point to the names of the versions used. For the mentioned platforms which are still using the legacy infrastructure, the value returned will be `unavailable`. For these platforms, the same version of the webcompiler service is ensured to be used by a FORCESPRO version throughout its lifetime and as such, explicit versioning is not necessary.

22.2.1 Webcompilers with legacy infrastructure

The platforms below continue using the legacy infrastructure for now:

- Host solver for `Windows with MSVC/Windows SDK compiler` on MATLAB client or Python client
- Target solver for `dSPACE platforms`
- Target solver for `Speedgoat platforms`
- Target solver for `NI-cRIO platforms`

22.3 Ensuring use of webcompiler version

As mentioned in the previous section, the version of the webcompiler service used to generate the binaries of a solver is available in the solver's header file. If it is critical for an application to ensure the same numerics across different generations of the used solver, the version of the webcompiler service can be explicitly requested using the entries retrieved from the header file and providing them during solver generation as shown below:

Matlab

Python

```
codeoptions.host_compiler_version = '<compiler_version>';
codeoptions.target_compiler_version = '<compiler_version>';
```

```
options.host_compiler_version = '<compiler_version>'
options.target_compiler_version = '<compiler_version>'
```

In case of questions please contact support at support@embotech.com to discuss the requirements of the application and decide on the appropriate setup.

Chapter 23

Frequently asked questions

- *Features of FORCESPRO*
- *Setting up a FORCESPRO problem formulation*
- *Issues during code generation*
- *Issues when running a generated solver*
- *Issues when using the Simulink interface*
- *Issues during code deployment*
- *Other topics*

23.1 Features of FORCESPRO

- **What problem classes are supported by FORCESPRO?**

  FORCESPRO is tailored to solve multistage optimization problems that are either convex (resulting in QP or QCQP problems, see *Low-level interface*) or non-convex (resulting in NLP problems, see *High-level Interface*) with continuous optimization variables. In addition, there is somewhat limited support for problems also containing binary or integer variables.

- **Can I use FORCESPRO for non-multistage problems?**

  Yes, FORCESPRO supports the case $N = 1$, i.e. a general QCQP of the form

$$\begin{aligned} \text{minimize} \quad & \frac{1}{2} z^\top H z + f^\top z \\ \text{subject to} \quad & Dz = c \\ & \underline{z} \leq z \leq \bar{z} \\ & Az \leq b \\ & z^\top Q z + q^\top z \leq r \end{aligned}$$

  In order to use this feature, simply call

  Matlab

  Python

```
stages = MultistageProblem(1);
```

```
stages = forcespro.MultistageProblem(1)
```

and fill in the matrices as described in *Low-level interface*. The same holds true for single-stage NLP problems.

- **I need to re-linearize the model of my plant at each sampling time. Does FORCESPRO support changing problem matrices during runtime?**

  Yes, basically all problem data can be made a real-time parameter and actual values provided during runtime (see section *Parametric problems* for convex problems and section *High-level Interface* for passing real-time parameters for a general nonlinear problem).

  When re-linearizing nonlinear dynamics, you obtain in each sampling time a different matrix A, B and also a new affine part g:

  $$x_{k+1} = Ax_k + Bu_k + g_k$$

  FORCESPRO supports changing these variables at run-time by defining them as parameters.

  Basically the only numerical quantities that need to be fixed during solver generation are the problem dimensions (such as horizon length and number of states/inputs etc.).

- **Which optimization algorithm is used by FORCESPRO?**

  FORCESPRO is a code generator that generates optimization solvers that are customized to a specific problem formulation. As the most efficient algorithm typically depends on the exact nature of the problem, various optimization algorithms are provided by FORCESPRO. In particular, implementations of interior-point and first-order methods for convex problems as well as primal-dual interior-point (PDIP) and sequential quadratic programming (SQP) methods for nonlinear problems are offered (see *Solve method*).

- **Can FORCESPRO target dSPACE hardware?**

  Yes, FORCESPRO can be seamlessly integrated in the dSPACE design flow using its Simulink interface. For more details see *dSPACE deployment through Simulink Coder* and *dSPACE deployment through ConfigurationDesk*.

- **Are FORCESPRO solvers thread-safe?**

  All solvers comprising only continuous optimization variables are thread-safe indeed (see section *Multicore parallelization*). Solvers comprising binary or integer variable are currently not thread-safe.

23.2 Setting up a FORCESPRO problem formulation

- **Does my problem formulation need to be smooth? What exactly does this mean?**

  Like any derivative-based optimization solver, FORCESPRO solvers work best if all functions defining the optimization problem are sufficiently smooth, i.e. at least continuously differentiable once. FORCESPRO provides a couple of modelling utilities (such as splines) to assist users to formulate smooth problems (see section *Modelling Utilities*).

  If your problem exhibits non-differentiabilities or even jumps, the solver will often fail to converge.

- **I need a lookup table inside my system dynamics, constraints or objective function. What is the best way to add it?**

  In order to yield a smooth formulation, lookup tables need to be approximated or interpolated by a smooth function. FORCESPRO provides dedicated functionality for 1D and

2D splines (see section *Modelling Utilities*) that can be used anywhere inside a nonlinear problem formulation.

- **Can I formulate convex-quadratic MPC problem using the high-level interface of FORCESPRO?**

  The *High-level Interface* has been designed to formulate general nonlinear optimization problems. If you use it for a formulation that happens to be convex-quadratic, FORCESPRO will typically detect this and will generate a convex solver instead of a nonlinear one. However, this convex solver will be less efficient than the same formulation provided directly via the *Low-level interface* as certain structure-exploiting features of convex solvers are not yet supported via the high-level interface.

  Finally, note that convexity detection will be bypassed in case the Hessian is provided as a real-time parameter: Since it is not guaranteed to be positive definite in this case, a nonlinear solver is generated, even though the formulation seems convex.

- **When accessing arrays within a problem formulation using the FORCESPRO client, do indices start from 0 or 1?**

  All indices for arrays inside the MATLAB client are 1-based. The Python client uses 0-based indices for the high-level interface and, for legacy reasons, 1-based indices for the low-level interface. Further details can be found in section *Differences between the MATLAB and the Python client*.

23.3 Issues during code generation

- **I get the following error when generating code: Internal error 0x<something> in FORCESPRO. Please contact support@embotech.com.**

  We are sorry for the inconvenience. While client errors and also most server errors provide meaningful error messages to the user, certain server errors only return an internal error code. In order to help you with this, please send an e-mail to support@embotech.com containing the exact message as well as some information on your problem formulation that caused this error.

- **The code generation process gets stuck displaying Generating and compiling code... and sometimes it returns an error after 10 minutes.**

  By default, the code is compiled with all compiler optimizations turned on (e.g. *-O3*). When the size of your code is large, typically when you have a long prediction horizon, it can take a very long time to compile the code with all optimizations turned on. If this process takes too long, the server times out and returns a compilation error. You can reduce the compilation time by reducing the compiler optimization level (see section *Compiler optimization level*).

- **I am not able to generate a FORCESPRO solver receiving the error This machine and user do not have a valid license.**

  This means your FORCESPRO license is either not valid or has not been setup correctly. Please make sure that:

  - Your FORCESPRO engineering license has not expired.
  - You have assigned the correct fingerprint of your host machine to your engineering license on the license portal (and this fingerprint has not changed in the meantime).
  - You have entered the correct username in your engineering license on the license portal.

  For further help see https://my.embotech.com/help or write an e-mail to support@embotech.com.

- **I get the following error when generating code: Server was unable to process request. —> There is no parameter that maps to c of stage 1. However, according to the multistage formulation, my D_1 is empty in my problem, so c_1 should also be empty.**

  We recommend to reformulate the optimization variables for each stage so that D_1 is not empty for performance reasons.

  If this is not possible and D_1 must remain empty, then the inter-stage equality constraint equations become

  $$C_{i-1}z_{i-1} + D_i z_i = c_{i-1}$$

  instead of

  $$C_{i-1}z_{i-1} + D_i z_i = c_i.$$

- **I get the following error message when generating code: Error downloading URL. Your network connection may be down or your proxy settings improperly configured.**

  Your current MATLAB configuration is not accepting our website's SSL certificate. Please follow this link to add our certificate to MATLAB's list of certificates manually. You can download the embotech certificate using your browser.

- **I get the following error message when generating code: Invalid MEX-file. The specified module could not be found.**

  Please install the Visual Studio redistributable libraries from here.

- **I get the following error when generating code: java.io.IOException: Server is not responding, it might not support the current protocol. Missing ServerHello.**

  Some MATLAB versions and some Java installations give problems when communicating using HTTPS from MATLAB. Please edit the file callSoapService.m. Search for the line

  ```
  url = URL(endpoint);
  ```

  and replace it with

  ```
  url = URL([], endpoint, sun.net.www.protocol.https.Handler)
  ```

- **I get the following error when generating code: java.io.IOException: The issuer can not be found in the trusted CA list.**

  Some MATLAB versions and some Java installations give problems when communicating using HTTPS from MATLAB. Please edit the file callSoapService.m. Search for the line

  ```
  url = URL(endpoint);
  ```

  and replace it with

  ```
  url = URL([], endpoint, sun.net.www.protocol.https.Handler)
  ```

- **I get the following error when generating code: javax.net.ssl.SSLException: Unrecognized SSL message, plaintext connection?**

  If you are using the enterprise version of FORCESPRO (separate server in your company network), had previously altered the file callSoapService.m to accept secure HTTP connections and the enterprise server is listening on an HTTP port, you receive this error. To fix: Please edit the file callSoapService.m. Search for the line

```
url = URL([], endpoint, sun.net.www.protocol.https.Handler)
```

and replace it by the default

```
url = URL(endpoint);
```

- **I get the following error message when using the MATLAB interface: 'Unable to cast object of type 'csmatio.types.MLDouble' to type 'csmatio.types.MLStructure'.'**

  Please check that you have your MEX compiler correctly set up. If the problem persists please send your MATLAB and platform settings to support@embotech.com.

23.4 Issues when running a generated solver

- **I am getting exitflag -6, -8 or -10 when running my solver.**

  There may be different of reasons depending on your exact formulation, but in general those exitflags point to numerical instabilities. Make sure your problem is reasonably scaled and see the other FAQs.

- **My solver returns exitflag -7.**

  Exitflag -7 means that the solver could not proceed. A common cause is the problem being infeasible. FORCESPRO does not have infeasibility detection to speed up the solution time. However, one can

  - in case of a convex problem, use the function *stages2qcqp* to convert the FORCESPRO problem into a standard (QC)QP that can be given to another QP solver to check for infeasibility (see *Debugging a formulation*);
  - in case of a general NLP, try formulating the problem with slack variables to make constraints soft to ensure feasibility.

  Another cause for exitflag -7 may be bad scaling of your optimization variables, including unreasonably large bounds.

- **I am getting exitflag -8 when running my SQP solver.**

  The underlying QP problem was found to be infeasible. Make sure your problem is reasonably scaled and try increasing the Hessian regularization `reg_hessian` (see *SQP specific codeoptions*).

- **I am getting exitflag -6 or -10 when running my nonlinear solver.**

  In this case it is a good idea to check that the nonlinear functions provided to the solver are well-defined and that they don't produce `NaN`s or `Inf`s. See section *Calling the nonlinear functions from MATLAB or Python* for how to call the nonlinear functions along with their derivatives directly in MATLAB or Python. If the solver returns exitflag -6, -8 or -10 in the first iteration one needs to check that

  Matlab

  Python

```
problem.x0
```

```
problem['x0']
```

  does not yield `NaN` or `Inf` when the nonlinear functions are evaluated on it. E.g. if one has generated a solver named `FORCESsolver` which does not use any real-time parameters, one needs to check that the following code does not produce an error

  Matlab

Python

```
jj = 1;
for ss = 1:model.N
    z = problem.x0[jj:(jj+model.nvar)];
    c, jacc = FORCESsolver_dynamics(z, [], ss);
    ineq, jacineq = FORCESsolver_inequalities(z, [], ss);
    obj, gradobj = FORCESsolver_objective(z, [], ss);
    assert( any(isnan(c) | isinf(c), 'all'), ['Encountered NaNs or Infs in c at␣
↪stage ', num2str(ss) ]);
    assert( any(isnan(jacc) | isinf(jacc), 'all'), ['Encountered NaNs or Infs in␣
↪jacc at stage ', num2str(ss) ]);
    assert( any(isnan(ineq) | isinf(ineq), 'all'), ['Encountered NaNs or Infs in␣
↪ineq at stage ', num2str(ss) ]);
    assert( any(isnan(jacineq) | isinf(jacineq), 'all'), ['Encountered NaNs or␣
↪Infs in jacineq at stage ', num2str(ss) ]);
    assert( any(isnan(obj) | isinf(obj), 'all'), ['Encountered NaNs or Infs in␣
↪obj at stage ', num2str(ss) ]);
    assert( any(isnan(gradobj) | isinf(gradobj), 'all'), ['Encountered NaNs or␣
↪Infs in gradobj at stage ', num2str(ss) ]);
    jj = jj + model.nvar;
end
disp('Did not encounter any NaNs or Infs');
```

```
import numpy as np

jj = 0
x0 = problem['x0']
for ss in range(model.N):
    z = x0[jj:model.nvar]
    c, jacc = solver.dynamics(z, stage=ss)
    ineq, jacineq = solver.ineq(z, stage=ss)
    bj, gradobj = solver.objective(z, stage=ss)
    assert any(np.isnan(c)) or any(np.isinf(c)), 'Encountered NaN in c at stage
↪' + str(ss)
    assert any(np.isnan(jacc)) or any(np.isinf(jacc)), 'Encountered NaN in jacc␣
↪at stage ' + str(ss)
    assert any(np.isnan(ineq)) or any(np.isinf(ineq)), 'Encountered NaN in ineq␣
↪at stage ' + str(ss)
    assert any(np.isnan(jacineq)) or any(np.isinf(jacineq)), 'Encountered NaN in␣
↪jacineq at stage ' + str(ss)
    assert any(np.isnan(obj)) or any(np.isinf(obj)), 'Encountered NaN in obj at␣
↪stage ' + str(ss)
    assert any(np.isnan(gradobj)) or any(np.isinf(gradobj)), 'Encountered NaN in␣
↪gradobj at stage ' + str(ss)
    jj = jj + model.nvar
print('Did not encounter NaNs')
```

Note: See also *Real-time SQP Solver: Robotic Arm Manipulator (MATLAB & Python)* for an example of how to simulate the dynamics of the system directly in MATLAB and Python.

- **My solver does not perform any iterations and returns exitflag -100 instead.**

  This means that the license check for your generated solver failed. Please make sure that:

  - You have assigned the correct fingerprint of the machine on which you are running

the solver to your software or hardware testing license on the license portal (and this fingerprint has not changed in the meantime).

- The license of the machine on which you are running the solver belongs to the same license group as the engineering license used to generate the solver.
- Your FORCESPRO licenses have not expired.
- The system clock of your target machine matches universal time (adjusted by correct timezone) quite precisely.

For further help see https://my.embotech.com/help or write an e-mail to support@embotech.com.

- **ADMM does not converge for my problem.**

  The convergence of ADMM depends heavily on the problem scaling. If the matrices for the problem data have very high condition numbers and norms, ADMM can converge extremely slowly regardless of the algorithm parameters. In some cases, ADMM might not converge at all due to severe accumulation of numerical errors. However, often the problem is choosing the right ADMM parameters ρ and α to obtain fast convergence of the algorithm.

  If rescaling does not help, try using an interior-point method instead, which is generally more robust and can address numerically more challenging formulations.

- **I am generating code from 32-bit MATLAB. When I run the code it produces a segfault. What is the problem?**

  By default, the code is compiled with all optimizations turned on (*-O3*). We have observed that sometimes there are problems when linking on 32-bit versions of MATLAB. This problem does not occur when the compiler optimization flags are set to *-O0*, *-O1*, or *-O2*. You can change the compiler optimization level by setting the following codeoptions to a value less than 3:

  Matlab

  Python

```
codeoptions.optlevel = 2;
```

```
codeoptions.optlevel = 2
```

- **When I run the solver in MATLAB I get the following error: ??? Error using ==> TestSolver freopen of stdout did not work.**

  This is a printing error that occurs in certain old versions of MATLAB because stdout is not defined inside MEX files. Supported versions of MATLAB should not produce this error. You can avoid this error by setting

  Matlab

  Python

```
codeoptions.printlevel = 0;
```

```
codeoptions.printlevel = 0
```

23.5 Issues when using the Simulink interface

- **When I have a long prediction horizon I have many input and output ports that I need to wire up in my Simulink interface. When I change my prediction horizon I need to re-wire them all again and this is a pain.**

FORCESPRO provides a 'compact' version of all Simulink interfaces that can be called with stacked parameters and has a small and constant number of input ports independent of the prediction horizon.

To check the dimensions of the new stacked parameters click on the 'Help' button in the dialogue of the 'compact' Simulink block.

23.6 Issues during code deployment

- **I get the following error message when deploying a solver on dSPACE hardware: OPUS MAKE: Don't know how to make ...**

  This is well-known deployment issue with compiled files. During building for target the compiler is looking for the source code of the solver. The resulting object file is added in the folder `<solvername>_<target_ext>` which is automatically generated by the compiler. Therefore, to use the object file you need to move it to that folder in order for the compiler to detect it and skip compilation. A possible workaround is to use the static library of the solver as specified in *dSPACE deployment through Simulink Coder*.

23.7 Other topics

- **Why am I being asked to update the FORCESPRO client software every now and then?**

  We have a development policy of continuous deployment, which unfortunately means that we have to ask users to update their clients every time there is a new release of FORCESPRO. To update your client simply type the following in the (MATLAB) command prompt:

  Matlab

  Python

```
>> updateClient
```

```
$ python updateClient.py
```

  For more details see *Installation of the MATLAB Client* or *Installation of the Python Client*.

- **How can I obtain information about the KKT conditions at the solution?**

  The *printlevel* solver option allows the user to control how much information is printed by the solver. See here for more information on how to define solver options.

  When *printlevel* is set to 2 the solver outputs information related to the KKT conditions at every iteration. In particular:

  - *res_eq* is the maximum $||C_{i-1}z_{i-1} + D_i z_i - c_i||_\infty$ for all i,
  - If we rewrite all inequality constraints as $Gz \leq g$ and s are slack variables for the same constraints, *res_ineq* is equal to $||Gz - g + s||_\infty$,
  - If λ are the Lagrange multipliers for the inequality constraints, μ is equal to $\lambda^\top s$ divided by the number of constraints, i.e. the average complementary slackness.

- **What system information am I sharing by using FORCESPRO?**

  When contacting the solver generation server, the FORCESPRO client sends the following system information:

- Machine username
- MAC address
- Fingerprints

The fingerprint is platform dependent. We create two fingerprints using different system information to create hashes and validate with either of them in order to have a more stable validation:

- For Windows, each fingerprint uses a subset of the below information:
  * Mac addresses
  * CPU ID (register with machine support)
  * Volume Serial Number
  * Volume GUID
- For MacOS, each fingerprint uses a subset of the below information:
  * CPU type and CPU subtype
  * Network node hostname
  * Mac addresses
- For Linux, each fingerprint uses a subset of the below information:
  * Network node hostname
  * /etc/machine-id
  * Mac addresses
  * Linux user uid

The above information is hashed to create the fingerprint which means that it cannot be recovered by using the fingerprint.

- **I have been using FORCES in the past. Why should I use FORCESPRO?**

  The development of the free version of FORCES by ETH (forces.ethz.ch) has been discontinued many years ago, and the code generation service is no longer available.

  The professional version of FORCESPRO comes with professional support, additional interfaces, and a large performance increase.

Bibliography

[GörSch] Göhrle, C.; Schindler, A.; Wagner, A.; Sawodny, O.: Design and Vehicle Implementation of Preview Active Suspension Controllers. IEEE Transactions on Control Systems Technology, pp.1135–1142, vol. 22, no. 3, May 2014

[HarMac14] Hartley, E. N.; Maciejowski, J. M.: Field programmable gate array based predictive control system for spacecraft rendezvous in elliptical orbits. In Optimal Control Applications and Methods, Mar 2014

[VukLoock] Vukov, Milan & Van Loock, Wannes & Houska, Boris & Ferreau, Joachim & Swevers, Jan & Diehl, Moritz. (2012). Experimental validation of nonlinear MPC on an overhead crane using automatic code generation. Proceedings of the American Control Conference. 6264-6269. 10.1109/ACC.2012.6315390.

[QuirDiehl] Quirynen, Rien & Gros, Sebastien & Diehl, Moritz. (2013). Efficient NMPC for nonlinear models with linear subsystems. Proceedings of the IEEE Conference on Decision and Control. 5101-5106. 10.1109/CDC.2013.6760690.

[SicSci09] Siciliano, B.; Sciavicco, L.; Villani, L.; Oriolo, G.. Robotics: Modelling, planning and control. Berlin: Springer, 2009.

[BerUnb] S. Daniel-Berhe; H. Unbehauen: Experimental physical parameter estimation of a thyristor driven DC-motor using the HMF-method. Control Engineering Practice, 6:615–626, 1998

[GarJor77] Garrard, W.L.; Jordan, J.M..: Design of Nonlinear Automatic Control Systems. In: Automatica 1977, vol. 13, 497-505.

[JiaJibItoGor19] Jia Y.; Jibrin, R.; Itoh Y.; Görges, D.: Energy-Optimal Adaptive Cruise Control for Electric Vehicles in Both Time and Space Domain based on Model Predictive Control. In: IFAC-PapersOnLine, vol. 50, no. 2, 2017, pp. 13-20, 2019.

[JiaJibGor20] Jia, Y.; Jibrin, R.; Görges, D.: Energy-Optimal Adaptive Cruise Control for Electric Vehicles Based on Linear and Nonlinear Model Predictive Control. In: IEEE Transactions on Vehicular Technology, vol. 69, no. 12, pp. 14173-14187, Dec. 2020.

[Fastned] Fastned: Charging with a BMW i3, https://support.fastned.nl/hc/en-gb/articles/204784718-Charging-with-a-BMW-i3

[xEngineer] X-engineer: Why do we need gears?, https://x-engineer.org/need-gears/

[BMWi3] BMW Group: Technical specifications of the BMW i3 (120 Ah), https://www.press.bmwgroup.com/global/article/detail/T0285608EN/

[JiaJibItoGor19_2D] Jia Y.; Jibrin, R.; Itoh Y.; Görges, D.: Energy-Optimal Adaptive Cruise Control for Electric Vehicles in Both Time and Space Domain based on Model Predictive Control. In: IFAC-PapersOnLine, vol. 50, no. 2, 2017, pp. 13-20, 2019.

[JiaJibGor20_2D] Jia, Y.; Jibrin, R.; Görges, D.: Energy-Optimal Adaptive Cruise Control for Electric Vehicles Based on Linear and Nonlinear Model Predictive Control. In: IEEE Transactions on Vehicular Technology, vol. 69, no. 12, pp. 14173-14187, Dec. 2020.

[Fastned_2D] Fastned: Charging with a BMW i3, https://support.fastned.nl/hc/en-gb/articles/204784718-Charging-with-a-BMW-i3

[xEngineer_2D] X-engineer: Why do we need gears?, https://x-engineer.org/need-gears/

[BMWi3_2D] BMW Group: Technical specifications of the BMW i3 (120 Ah), https://www.press.bmwgroup.com/global/article/detail/T0285608EN/

[JiaJibGor20_2D_modelling] Jia, Y.; Jibrin, R.; Goerges, D.: Energy-Optimal Adaptive Cruise Control for Electric Vehicles Based on Linear and Nonlinear Model Predictive Control. In: IEEE Transactions on Vehicular Technology, vol. 69, no. 12, pp. 14173-14187, Dec. 2020.