FORCES PRO

FORCES PRO

Version 3.1.0

FORCES PRO User Manual

Embotech AG

Technoparkstrasse 1

CH-8005 Zürich

www.embotech.com

info@embotech.com

Contents

1 Introduction **1**
1.1 Troubleshooting and support . . . 1
1.2 Licensing . . . 2
1.3 Citing FORCES PRO . . . 2
1.4 Release Notes . . . 3
1.5 Version history of manual . . . 6

2 Installation **9**
2.1 Obtaining FORCES PRO . . . 9
2.2 Installation of the MATLAB Client . . . 9
2.3 Installation of the Python Client . . . 10

3 Y2F Interface **15**
3.1 Installing Y2F . . . 15
3.2 Generating a solver . . . 15
3.3 Calling the solver . . . 16
3.4 Solver info . . . 16
3.5 Performance . . . 17
3.6 Examples . . . 19

4 MathWorks MPC Toolbox Plugin **21**
4.1 Different types of solvers . . . 22
4.2 Generating a QP solver from an MPC object . . . 22
4.3 Solving a QP from MPC online data . . . 25
4.4 Using the FORCES PRO MPC Simulink block . . . 26
4.5 Deploy to dSpace MicroAutoBox II using the FORCES PRO MPC Simulink block . 28
4.6 Examples . . . 31

5 Low-level interface **43**
5.1 Supported problem class . . . 43
5.2 Multistage struct . . . 44
5.3 Dimensions . . . 44
5.4 Cost function . . . 44
5.5 Equality constraints . . . 45
5.6 Lower and upper bounds . . . 45
5.7 Polytopic constraints . . . 45
5.8 Quadratic constraints . . . 46
5.9 Binary constraints . . . 47
5.10 Declaring Solver Outputs . . . 47
5.11 Generating the solver . . . 48
5.12 Calling the generated low-level solver . . . 48

6 High-level Interface **51**
6.1 Supported problems . . . 51
6.2 Expressing the optimization problem in code . . . 53
6.3 Generating a solver . . . 61
6.4 Calling the solver . . . 62

6.5 External function evaluations in C . . . 65
6.6 Mixed-integer nonlinear solver . . . 69
6.7 Sequential quadratic programming algorithm . . . 72
6.8 Differences between the MATLAB and the Python client . . . 75
6.9 Examples . . . 75

7 Simulating your custom controller in Simulink® 77
7.1 Configuration of a custom linear MPC controller using the FORCES PRO Simulink® GUI . . . 77
7.2 Getting Started - Basic MPC Regulation State Feedback Example . . . 80
7.3 Real-time control with the Simulink block . . . 82

8 Examples 87
8.1 How to . . . 87
8.2 Y2F interface: Basic example . . . 104
8.3 Low-level interface: Active Suspension Control . . . 110
8.4 Low-level interface Robust estimation (Kalman filter) . . . 114
8.5 Low-level interface: Spacecraft Rendezvous . . . 120
8.6 Low-level interface DC/DC converter . . . 122
8.7 High-level interface: Basic example . . . 129
8.8 High-level interface: Obstacle avoidance (MATLAB & Python) . . . 133
8.9 High-level interface: Indoor localization (MATLAB & Python) . . . 143
8.10 Real-time SQP Solver: Robotic Arm Manipulator (MATLAB & Python) . . . 147
8.11 Controlling a DC motor using a FORCES PRO SQP solver . . . 153
8.12 Mixed-integer nonlinear solver: F8 Crusader aircraft . . . 158

9 Parametric problems 165
9.1 Defining parameters . . . 165
9.2 Example . . . 166
9.3 Parametric Quadratic Constraints . . . 166
9.4 Diagonal Hessians . . . 167
9.5 Sparse Parameters . . . 167
9.6 Special Parameters . . . 167
9.7 Python: Column vs Row Major Storage Format . . . 168

10 Code Deployment 169
10.1 Main Targets . . . 169
10.2 dSPACE MicroAutoBox II . . . 173
10.3 dSPACE MicroAutoBox III . . . 183
10.4 Speedgoat . . . 201

11 Licensing 213
11.1 Machine Identification . . . 213
11.2 Static License . . . 213
11.3 License Files . . . 214
11.4 Floating Licenses . . . 215

12 Solver Options 217
12.1 General options . . . 217
12.2 High-level interface options . . . 225
12.3 Convex branch-and-bound options . . . 230
12.4 Solve methods . . . 231

13 Dumping Problem Formulation and Data 237
13.1 Why to use the dump tool? . . . 237
13.2 How to use the dump tool? . . . 237
13.3 Limitations . . . 239

14 Frequently asked questions 241

14.1 Quick links . . . 241
14.2 Features of FORCES PRO . . . 241
14.3 Issues during code generation . . . 242
14.4 Issues when running the solver . . . 243
14.5 Simulink interface . . . 244
14.6 Code deployment . . . 244
14.7 Other topics . . . 245

Bibliography **247**

Chapter 1

Introduction

This is a user manual for FORCES PRO, a commercial tool for generating highly customized optimization solvers that can be deployed on all embedded computers. FORCES PRO 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.

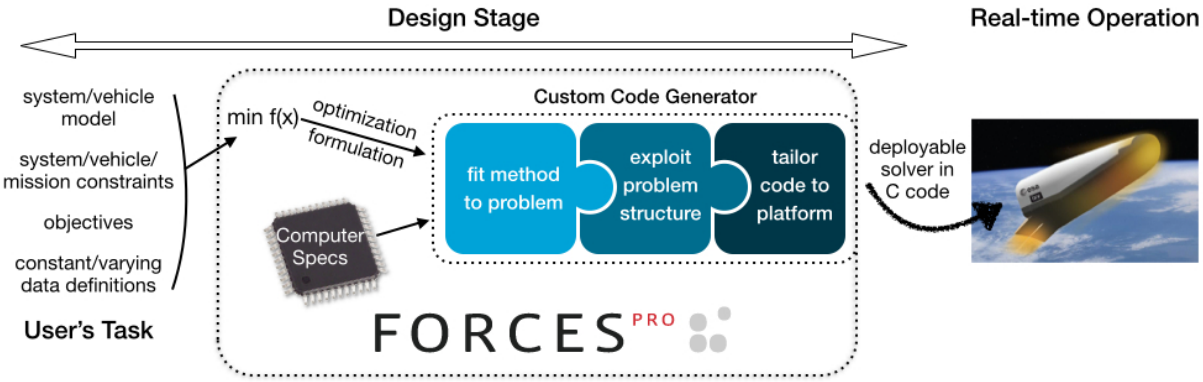


Figure 1.1: Overview of FORCES PRO.

The code generation engine in FORCES PRO 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 FORCES PRO. It is important to point out that FORCES PRO 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

FORCES PRO 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 FORCES PRO client and is also available on all FORCES PRO 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 Industrial licensing

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

- **Engineer License**: For generating FORCES PRO solvers. Charged per engineer computer.
- **Software Testing License (SiL/CI)**: For running FORCES PRO 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 FORCES PRO solvers on servers or virtualised 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 contact sales@embotech.com.

1.2.2 Academic licensing

Users at degree granting institutions can have access to the **Engineer License** version of FORCES PRO 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 FORCES PRO

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

```
@misc{FORCESPro,
      Author       = "Alexander Domahidi and Juan Jerez",
      Howpublished = "Embotech AG, url=https://embotech.com/FORCES-Pro",
      Title        = "FORCES Professional",
      Year         = "2014--2019"
}
```

```
@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 Release Notes

1.4.1 New features in FORCES PRO 3.1.0

- High-level Python interface for NLP solvers

1.4.2 Improvements in FORCES PRO 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 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

1.4.3 Bug Fixes in FORCES PRO 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

1.4.4 Improvements in FORCES PRO 3.0.1

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

1.4.5 Bug Fixes in FORCES PRO 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 SQP_NLP

1.4.6 New features in FORCES PRO 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

1.4.7 Improvements in FORCES PRO 3.0.0

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

1.4.8 Bug Fixes in FORCES PRO 3.0.0

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

1.4.9 New features in FORCES PRO 2.0.0

- Introduced support for FORCES PRO QP solvers in the MATLAB MPC Toolbox from MathWorks
- Created new examples for the MPC Toolbox plugin

1.4.10 Improvements in FORCES PRO 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

1.4.11 Bug Fixes in FORCES PRO 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

1.4.12 New features in FORCES PRO 1.9.1

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

1.4.13 Improvements in FORCES PRO 1.9.1

- Fixed numerical bug in NLP line-search

1.4.14 New features in FORCES PRO 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

1.4.15 Improvements in FORCES PRO 1.9.0

- Changed floating license communication to http
- Enabled user-defined outputs in MINLP solver
- Added codeoption c90 to add extra C definitions in casadi model files
- Added openmp flag to nvidia webcompiler
- Added support for python3.6
- Updated usysid files in client

1.4.16 Bug Fixes in FORCES PRO 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

1.4.17 New features in FORCES PRO 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

1.4.18 Improvements in FORCES PRO 1.8.0

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

1.4.19 Bug Fixes in FORCES PRO 1.8.0

- Fixed numerical bug in v1.7.0

1.4.20 New features in FORCES PRO 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 codeoption compactSparse for smaller code and faster compilation of sparse problems
- Adding threadSafeStorage option, enabling creation of thread-safe solvers (requires C11 compilers)

1.4.21 Improvements in FORCES PRO 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 FORCES PRO client
- Now passing iteration number to function evaluations
- Added new error code for invalid parameter initial values

1.4.22 Bug Fixes in FORCES PRO 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

1.5 Version history of manual

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

Table 1.1: Version history of FORCES PRO manual

Version	Revision	Date	Reason for change
1	0	04/10/2017	Initial version
2	0	09/27/2018	Overhaul of outdated manual
2	1	11/19/2018	Add dSPACE code deployment
3	0	02/20/2019	Updated manual for v1.7.0
4	0	06/04/2019	Updated manual for v1.8.0
4	1	08/29/2019	Updated manual for v1.9.0
5	0	10/10/2019	Updated manual for v1.9.1
6	0	12/09/2019	Updated manual for v2.0.0
7	0	04/07/2020	Updated manual for v3.0.0
8	0	05/26/2020	Updated manual for v3.0.1
9	0	07/13/2020	Updated manual for v3.1.0

Chapter 2

Installation

2.1 Obtaining FORCES PRO

FORCES PRO 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 FORCES PRO, follow the steps below:

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

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

2.2 Installation of the MATLAB Client

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

```
addpath /home/user/FORCES_PRO
```

on your MATLAB command prompt. Alternatively, you can add the path of the `FORCES_PRO` folder via the graphical user interface of MATLAB as seen in Figure 2.1.

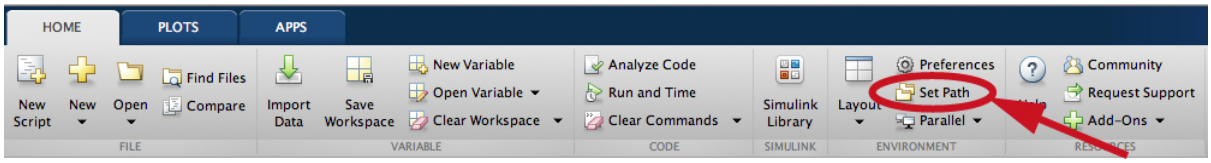


Figure 2.1: Adding the `FORCES_PRO` folder to the MATLAB path.

2.2.1 System requirements

FORCES PRO 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.

2.2.2 Keeping FORCES PRO up to date

FORCES PRO 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 FORCES PRO client may also be updated through Python, see *Keeping FORCES PRO up to date*.

2.3 Installation of the Python Client

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

2.3.1 Quick Guide

If you want to get up and running quickly, we have compiled the most common commands needed to go from a blank system to generating and executing the first solver in a example below. If you encounter issues, please have a look at the more detailed description of the required prerequisites below.

In the following, we assume you have obtained the FORCES PRO client as described in *Obtaining FORCES PRO*, and unzipped its files into the directory */path/to/forces/pro* on Unix platforms or *C:\path\to\forces\pro* 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 numpy scipy suds-jurko casadi matplotlib
$env:PYTHONPATH="C:\path\to\forces\pro"
C:\PythonXY\python.exe C:\path\to\forces\pro\examples\robot_sim.py
```

Linux Ubuntu

```
pip3 install numpy scipy suds-jurko casadi matplotlib
sudo apt-get install gcc libomp-dev
export PYTHONPATH="/path/to/forces/pro":$PYTHONPATH
python3 /path/to/forces/pro/examples/robot_sim.py
```

Mac

```
xcode-select --install
brew install python3 libomp
python3 -m pip install numpy scipy suds-jurko casadi matplotlib
export PYTHONPATH="/path/to/forces/pro":$PYTHONPATH
python3 /path/to/forces/pro/examples/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)"
```

2.3.2 Requirements

The Python client has been tested with the follwing 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 2.7 (low-level convex problems only)
- Python 3.6
- Python 3.7
- Python 3.8

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

For purposes of readibility, 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*, and Python 2.7 as *python*.

Python Packages

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

- **numpy** (Tested with version 1.18.3)
- **scipy** (Tested with version 1.4.1)
- **casadi** (Version 3.5.1 required only for high-level interface)
- **matplotlib** (Required only for plotting in the example code)

Additionally, Python 2.7 requires the following packages:

- **suds**

Additionally, Python versions 3.x require the following packages:

- **suds-jurko**

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

```
pip3 install numpy scipy casadi matplotlib suds-jurko
```

Or, on Windows:

```
C:\PythonXX\Scripts\pip.exe install numpy scipy casadi matplotlib suds-jurko
```

Available Compiler

Nonlinear symbolic problem formulations are translated into C code by the FORCES PRO 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 FORCES PRO 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 MINL-problems, the following package must be installed through Homebrew:

```
brew install libomp
```

2.3.3 Adding the FORCES PRO Python Client to your Python path

Once the FORCES PRO 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 FORCES PRO client interface in Python. Doing so will allow you to write *import forcespro* or *import forcespro.nlp* in your scripts to

import the FORCES PRO functionality. To make the FORCES PRO client available this way, you have several options:

Option A: Setting the *PYTHONPATH* environment variable

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

```
$env:PYTHONPATH="C:\path\to\forces\pro"
```

In Windows *cmd.exe*:

```
set PYTHONPATH=C:\path\to\forces\pro
```

On Unix (Linux and Mac):

```
export PYTHONPATH=/path/to/forces/pro
```

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

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

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

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

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

2.3.4 Keeping FORCES PRO up to date

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

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

```
$ cd /path/to/forces/pro
$ python updateClient.py
```

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

Chapter 3

Y2F Interface

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 FORCES PRO 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 FORCES PRO 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`.

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.

3.1 Installing Y2F

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

```
addpath /home/user/FORCES_PRO/Y2F/Y2F
```

on your MATLAB command prompt.

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

3.2 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 FORCES PRO solver:

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

where Options is a FORCES `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.

3.3 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).

3.4 Solver info

3.4.1 Exitflags

One should always check whether the solver has exited without an error before using the solution. Possible values of `exitflag` are presented in Table 3.1.

Table 3.1: Exitflag values

Exitflag	Description
1	Optimal solution found to the requested accuracy.
2	(only 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.
0	Timeout – maximum number of iterations or maximum computation time of `codeoptions.mip.timeout` (only branch-and-bound) reached. The returned solution is the best one found so far.
-1	(only branch-and-bound) Infeasible problem (problems solving the root relaxation to desired accuracy).
-2	(only branch-and-bound) Out of memory – cannot fit branch and bound nodes into pre-allocated memory.
-7	The convex solver could not proceed due to stalled line search. The problem might be infeasible. Otherwise, please submit a bug report to support@embotech.com including all data necessary to reproduce the problem. You can also run `FORCESdiagnostics` on your problem to check for most common errors in the formulation.
-10	The convex solver could not proceed due to an internal error. The problem might be infeasible. Otherwise, please submit a bug report to support@embotech.com including all data necessary to reproduce the problem. You can also run `FORCESdiagnostics` on your problem to check for most common errors in the formulation.
-100	License error. If you have generated code with a simulation license, it will run only on the machine from which the code has been generated. In some cases, e.g. when connected to a VPN network, the FORCES PRO license checker produces a false negative. Re-run the code generation script in this case to make sure licensing information is correctly set.

3.4.2 Additional diagnostics

The solver returns additional information to the optimizer in the `info` struct. Some of the fields are described in Table 3.2. Depending on the method used, there will also be other fields describing the quality of the returned result.

Table 3.2: 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.

3.5 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 FORCES PRO (2 states, 1 input, box constraints, varying horizon) is presented in Figure 3.1. 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 FORCES PRO.

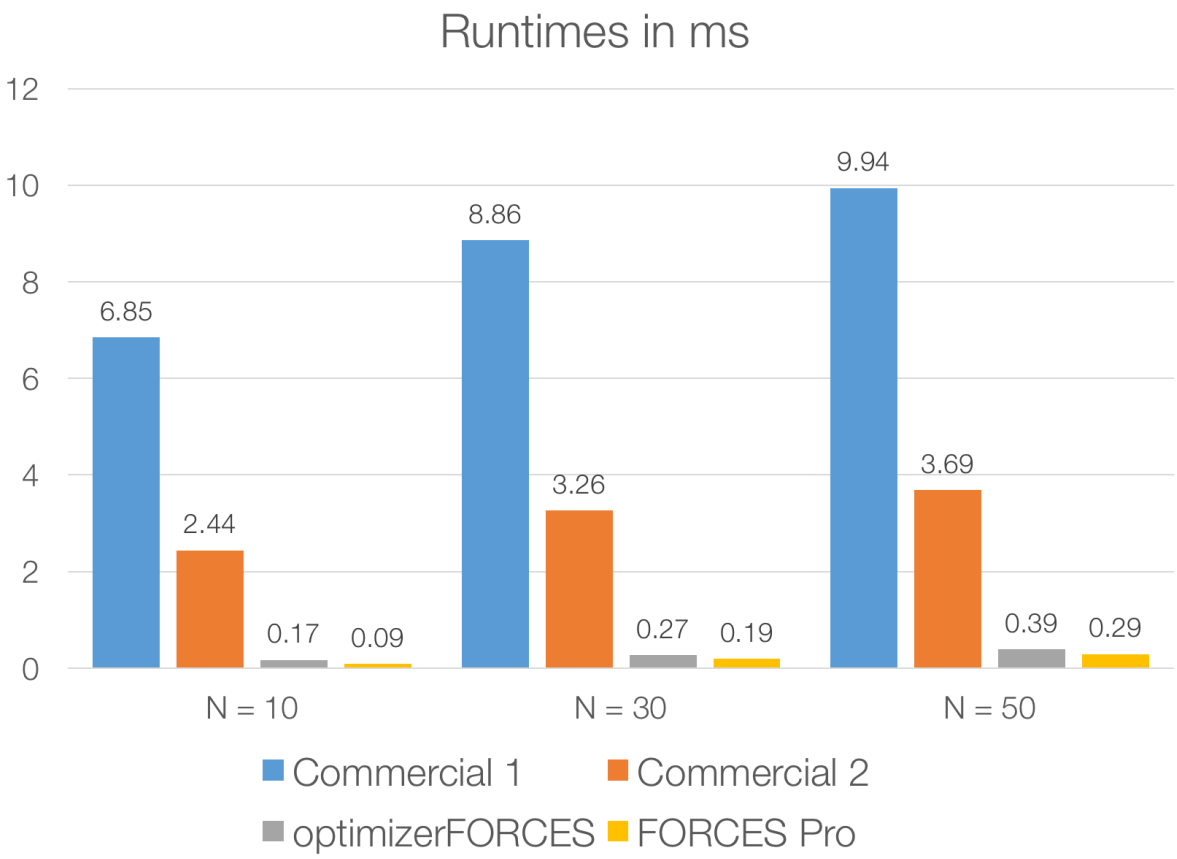


Figure 3.1: Performance comparison of the Y2F interface of FORCES PRO.

3.6 Examples

- *Y2F interface: Basic example*: Learn how to formulate problems in YALMIP easily, and then use the Y2F interface to generate code with FORCES PRO.

Chapter 4

MathWorks MPC Toolbox Plugin

As a result of a long-term collaboration, MathWorks Inc. and Embotech AG developed a MATLAB® plugin for FORCES PRO. Users are now able to use the FORCES PRO solver in MATLAB® and Simulink® from within the MATLAB® Model Predictive Control Toolbox. The plugin leverages the powerful design capabilities of the MPC Toolbox™ and the computational performance of FORCES PRO. With FORCES PRO 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 lets users 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 FORCES PRO, enables code deployment of the FORCES PRO solver on real-time hardware from within MATLAB® and Simulink®, in addition to the QP solvers shipped by MathWorks. The new FORCES PRO 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 allows users to run closed-loop simulations and evaluation of controller performance. User-friendly MPC design capabilities are combined with the powerful numerical algorithms of FORCES PRO. This combination of the MPC Toolbox™ and FORCES PRO enables code deployment on real-time hardware. The generated code is highly optimized for fast computations and low memory footprint.

The plugin mainly consists of the three following MATLAB commands which are described in details in this chapter:

- `mpcToForces` for generating a FORCES PRO solver from an MPC object designed by the MPC Toolbox
- `mpcmoveForces` for calling the generated solver on a specific MPC problem instance
- `mpcCustomSolver` for using the FORCES PRO 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 LTI 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 ECR
- Uniform or time-varying bounds on outputs, manipulated variables and manipulated variables rates
- Soft constraints
- Signal previewing on reference and measured disturbances
- Scale factor
- Nominal values
- Online updates of weights and constraints
- Built-in and custom state estimators

Currently, convex quadratic programs are supported by the MATLAB plugin. Extensions to adaptive, linear time-varying, nonlinear MPC are under development. The current limitations of the plugin are the following:

- Mixed input-output constraints are not covered
- Offdiagonal terms on the hessian of the objective cannot be implemented
- Unconstrained problems are not supported
- No single-precision solvers
- No suboptimal solutions

4.1 Different types of solvers

The plugin converts an MPC object (weights, bounds, horizons, references, output measurements) into a quadratic program (QP) formulated via the FORCES PRO 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 less 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.

4.2 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);
```

Two types of QP solvers can be generated via `mpcToForces`: a **sparse** solver that corresponds to a multi-stage formulation as in *Low-level interface* and a **dense** solver that corresponds to a one-stage QP with inequality constraints only.

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** as inputs.

Table 4.1: *mpcToForces* inputs

Input	Description
mpcobj	LTI MPC controller designed by MPC Toolbox
options	Object that provides solver generation options.

The outputs of `mpcToForces` consist of three structures **coredata**, **statedata** and **onlinedata**. The FORCES PRO server generates 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.
- *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.

The user is not allowed to change the generated solver name.

Table 4.2: *mpcToForces* outputs

Output	Type	Description
coredata	Structure	Store constant Store constant data needed to construct quadratic progam at run-time
statedata	Structure	Represent prediction model states and last optimal MV. The index k stands for the current simulation time. It contains 4 fields: When built-in state estimation is used: `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 $x_n[k\|k-1]$ `LastMove` is the optimal manipulated variables at the previous solve In this case, users should not manually change any field at run-time. When custom state estimation is used: `Plant` is the estimated plant state $x_p[k\|k]$ `Disturbance` is the estimated disturbance states $x_d[k\|k]$ `Noise` is the estimated noise states $x_n[k\|k]$ `LastMove` is the optimal manipulated variables at the previous solve In this case, user should manually change Plant, Disturbance (if used), Noise (if used) fields at run-time but leave LastMove alone.
onlinedata	Structure	Represent online signals It contains up to three fields: `signals`, a structure containing following fields: *ref* (references of Output Variables) *mvTarget* (references of Manipulated Variables) *md* (when Measured Disturbance is present) *ym* (when using the built-in estimator) *externalMV* (when UseExternalMV is true in the options object) `weights`, a structure containing the following fields: *y* (when UseOnlineWeightOV is enabled) *u* (when UseOnlineWeightMV is enabled) *du* (when UseOnlineWeightMVRate is enabled) *ecr* (when UseOnlineWeightECR is enabled) `constraints`, a structure containing the following fields: *vmin* (when UseOnlineConstraintOVMin is enabled) *vmax* (when UseOnlineConstraintOVMax is enabled) *umin* (when UseOnlineConstraintMVMin) *umax* (when UseOnlineConstraintMVMax) *dumin* (when UseOnlineConstraintMVRateMin) *dumax* (when UseOnlineConstraintMVRateMax)

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 a one-stage dense QP solvers
- "sparse" for generating a multi-stage QP solver.

The structures provided by the `mpcToForcesOptions` command have the following MPC related fields in common both in 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, MW weights should be positive for proper tracking. When false, the MV reference is the nominal value by default. In this case, MV weights should be zero to avoid unexpected behaviour.

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

- *ForcesServer* is the FORCES PRO server url. Default is forces.embotech.com.
- *ForcesMaxIteration* is the maximum number of iterations in a FORCES PRO solver. Default value is 50.
- *ForcesPrintLevel* is the logging level of the FORCES PRO 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 FORCES PRO interior point algorithm. If equal to 0, the solver is cold-started. If equal to 1, a centered start is computed.
- *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}.

4.3 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` as follows

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

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

The outputs of the *mpcmoveForces* command are described below. In the table 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.

Table 4.3: *mpcmoveForces* outputs

Output	Type	Description
mv	Vector of size	Optimal manipulated variables at current solve time instant
statedata	Structure	Initialized by `mpcToForces`
info	Structure	Information about the FORCES solve `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 FORCES PRO 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

4.4 Using the FORCES PRO MPC Simulink block

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

- Generate a custom dense FORCES PRO solver

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

- Set the following settings in the MPC object

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

The FORCES PRO sparse QP solver is also available via the MPC 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 4.1 below. Currently only the sparse QP solver can be used via the Simulink API.

In order to run an MPC simulation using the FORCES PRO block, a solver first needs to be generated via the following code for instance:

```
%% Generate FORCES PRO 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
```

(continues on next page)

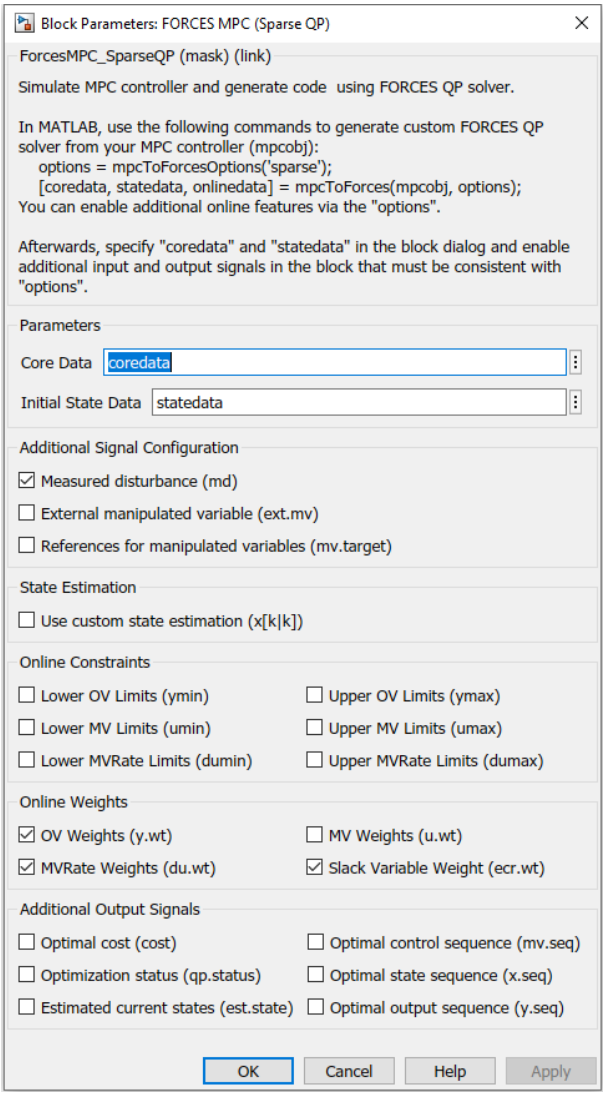


Figure 4.1: FORCES PRO MPC block configuration window

(continued from previous page)

```
options.UseOnlineWeightOV = true;
options.UseOnlineWeightMVRate = true;
options.UseOnlineWeightECR = true;
[coredata, statedata, onlinedata] = mpcToForces(mpcobj, options);
```

The structures *coredata* and *statedate* needed by the FORCES PRO solver are then provided to the Simulink block via the window shown in Figure 4.1.

- *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 FORCES PRO MPC block in a Simulink model is shown in Figure 4.2 below.

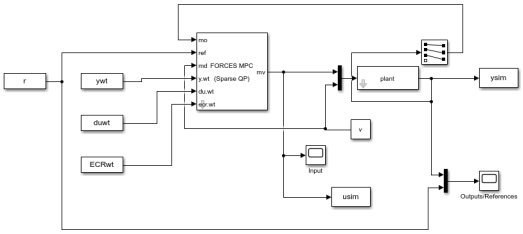


Figure 4.2: Simulink model illustrating the integration of the FORCES PRO MPC block

The Simulink model can be run from MATLAB in the usual way.

```
% Start simulation.
mdl = 'forcesmpc_onlinetuning';
open_system(mdl);                    % Open Simulink(R) Model
sim(mdl);                            % Start Simulation
```

Finally, the FORCES PRO MPC block is available via the Library browser once the user has updated his client to the latest version of FORCES, as shown in Figure 4.3 below.

4.5 Deploy to dSpace MicroAutoBox II using the FORCES PRO MPC Simulink block

The FORCES PRO 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 4.4 below.

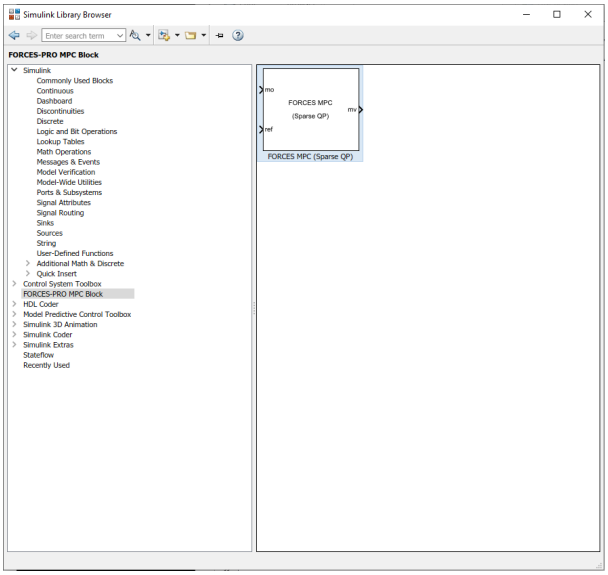


Figure 4.3: FORCES PRO MPC block in the library browser

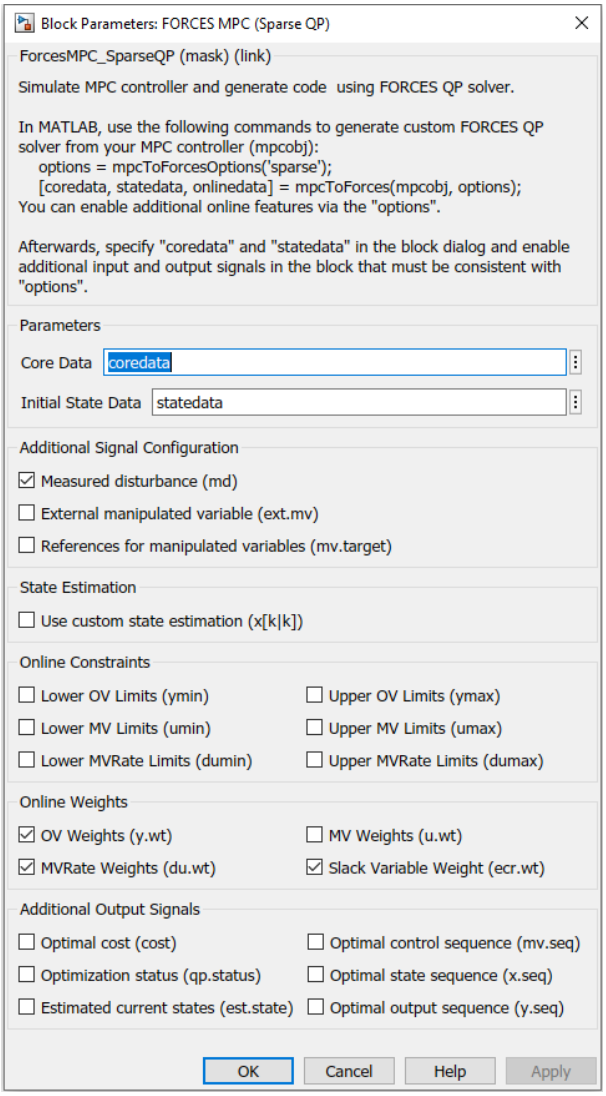


Figure 4.4: FORCES MPC block configuration

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

```
%% Generate FORCES PRO 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;
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 *statedate* needed by the FORCES PRO solver are then provided to the Simulink block via the window shown in Figure 4.4. The integration of the FORCES PRO MPC block in a Simulink model is shown in Figure 4.5 below.

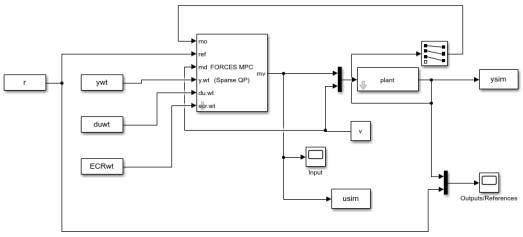


Figure 4.5: FORCES PRO MPC block integration in a Simulink model

3) When creating the Simulink Model, in the Configurations, in the "Code Generation" tab, set the options (see Figure 4.6 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 4.7).
6) Open dSpace Control Desk and select create new project (see Figure 4.8).
7) Name the project and the experiment (see Figure 4.9 and Figure 4.10).
8) Select the platform to which you will deploy the generated executable (see Figure 4.11).
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 4.12 and Figure 4.13).

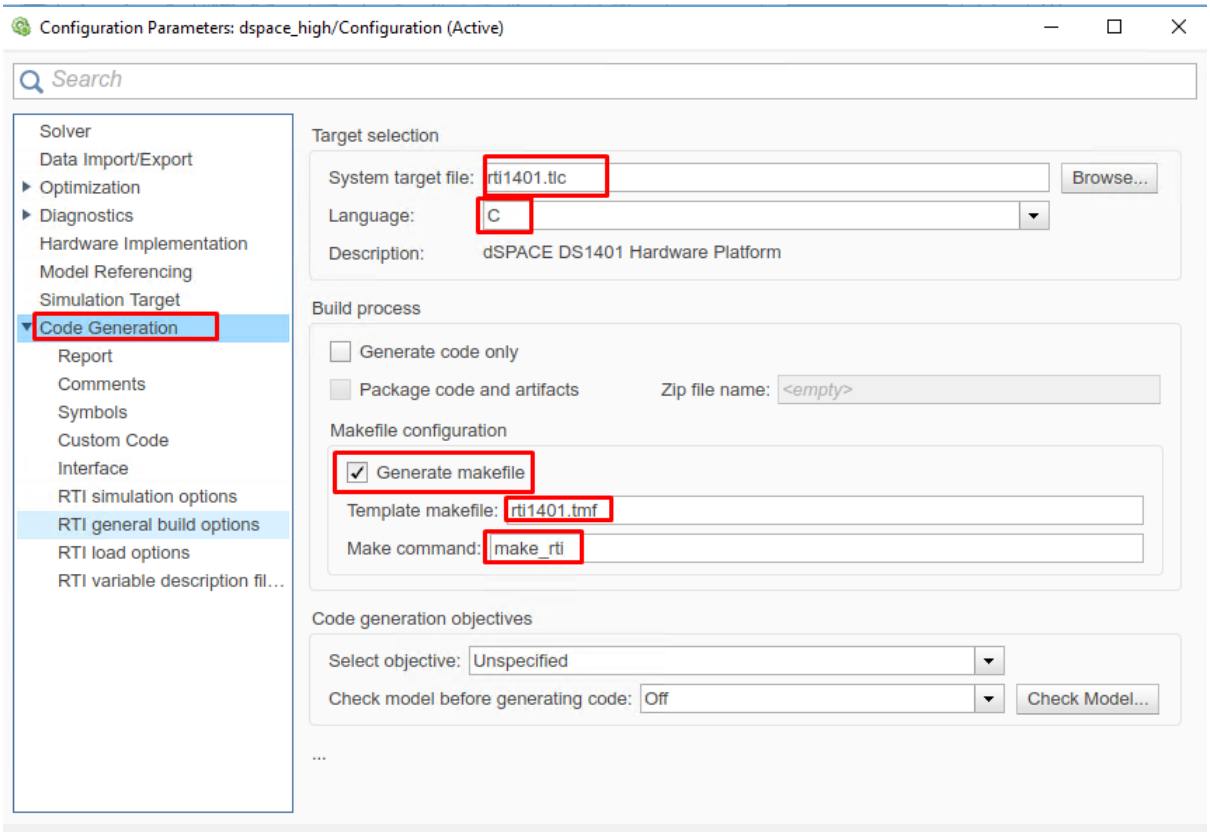


Figure 4.6: Configure Code Generation for dSPACE MicroAutoBox II

10) Click Finish to create the project (see Figure 4.14).
11) On the project layout select the tab `Variables` and on the `forcesmpc_onlinetuning_dSpace_MicroAutoBoxII` category expand `Model Root` (see Figure 4.15).
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 4.16).
13) Drag & Drop the output variables again and now choose `Display` (see Figure 4.17).
14) To see all the plots concurrently right-click on the left of the Y-axis and select `YAxes-view> Horizontal stacked` (see Figure 4.18).
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 4.19 and Figure 4.20).
16) Select `Go Online` and `Start Measuring` to see the results. (see Figure 4.21 and Figure 4.22).

4.6 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` folder that comes with your client.

The packaged examples are the following ones:

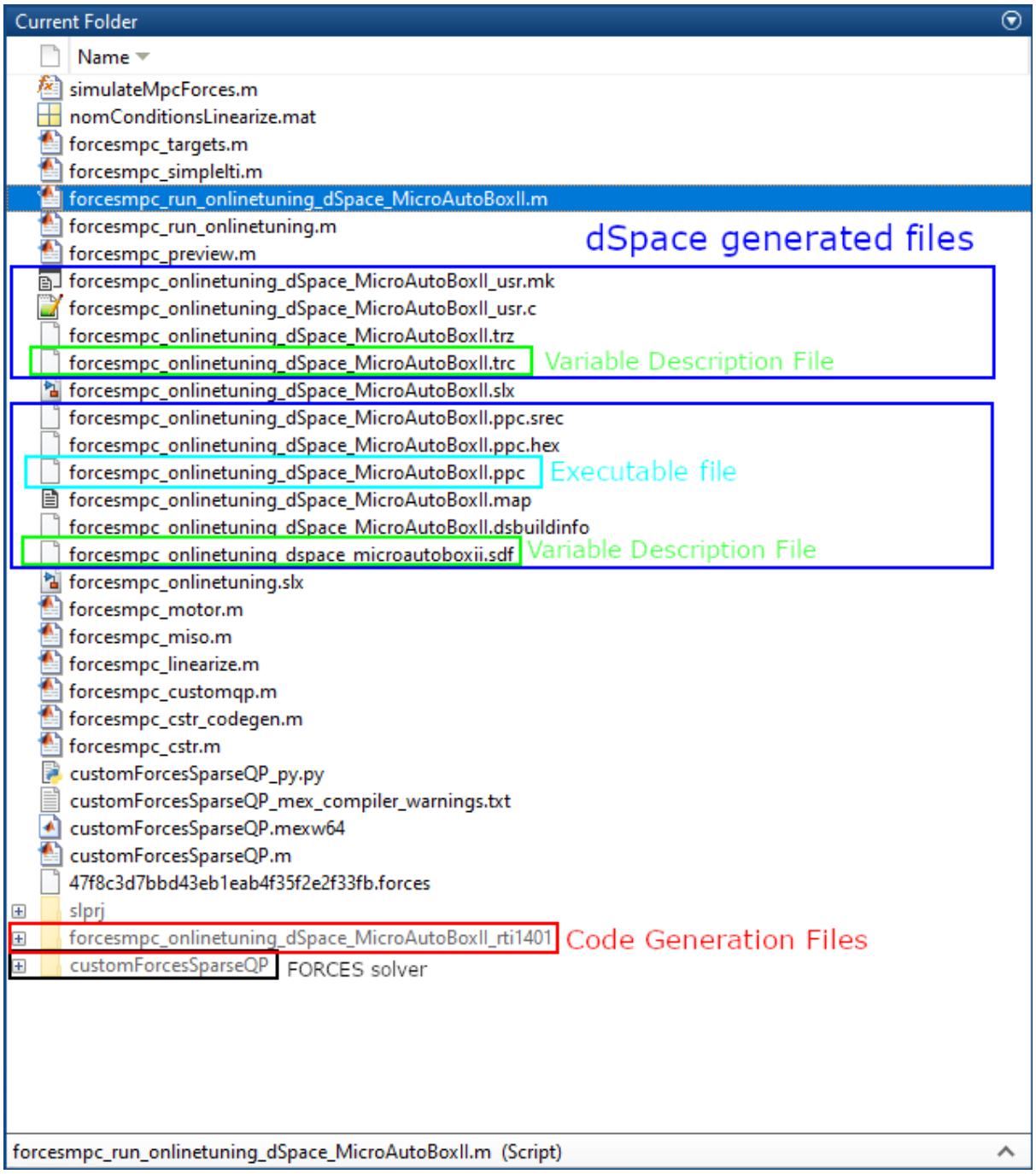


Figure 4.7: The generated files from the Simulink Code Generation

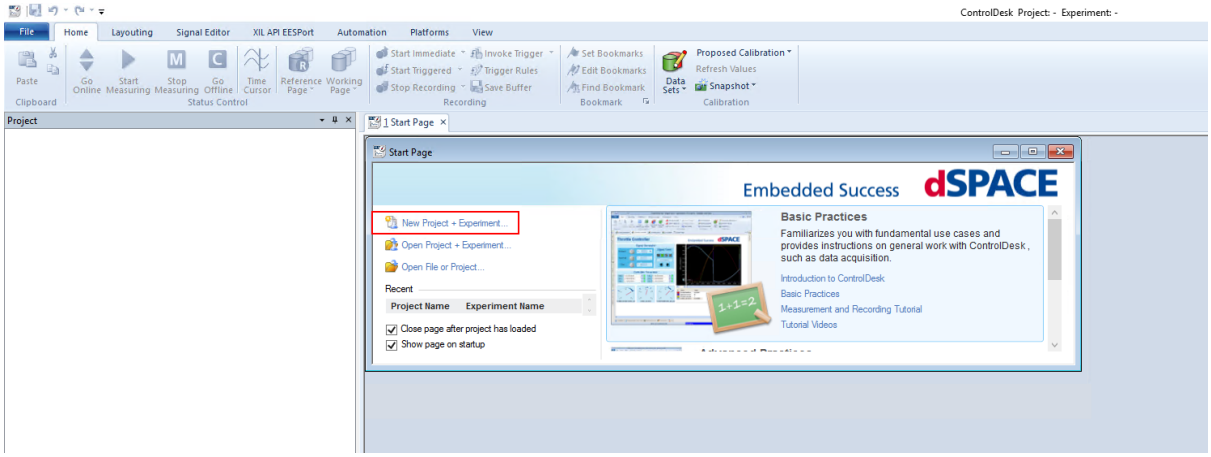


Figure 4.8: Start a new project

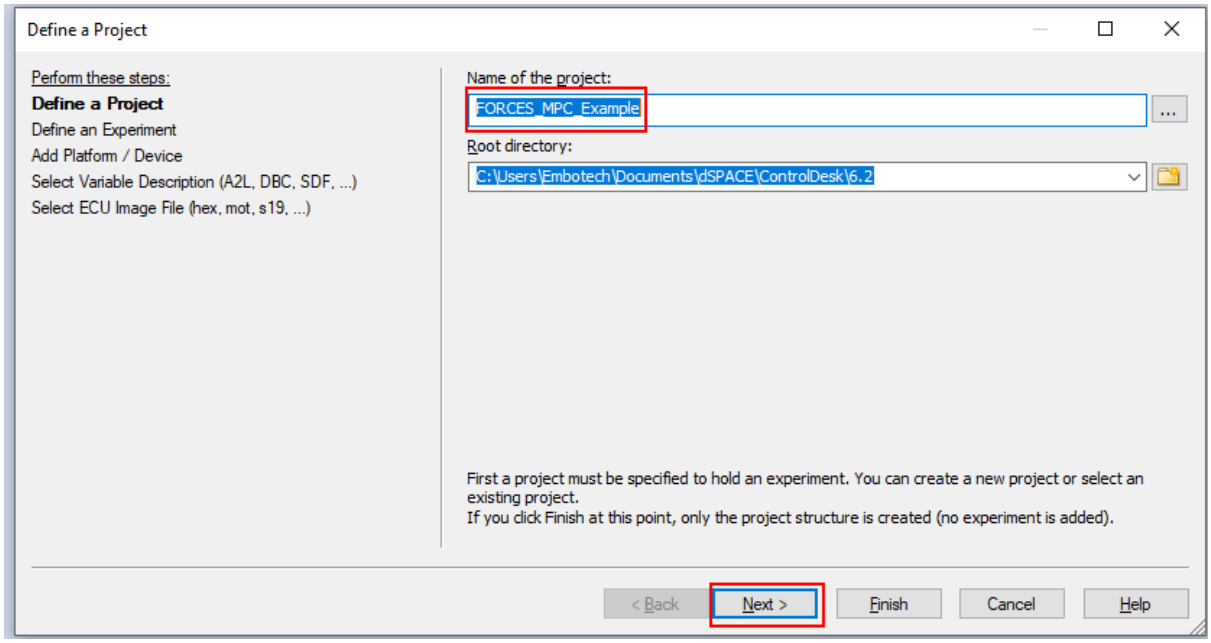


Figure 4.9: Name your project

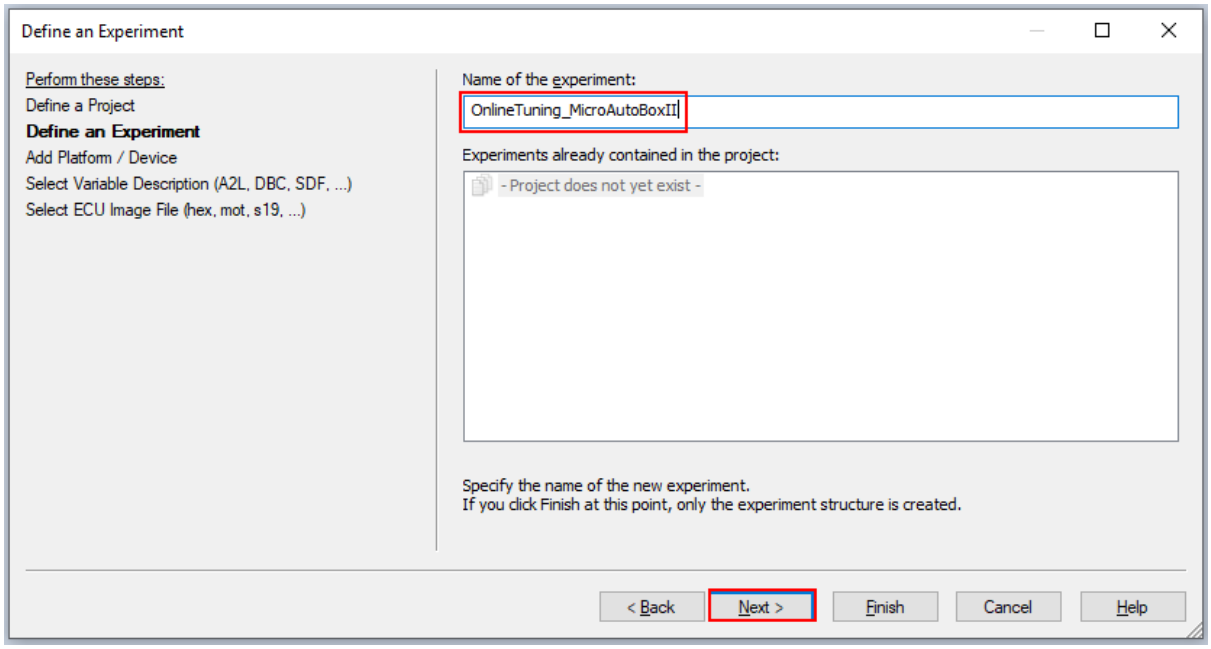


Figure 4.10: Name your experiment

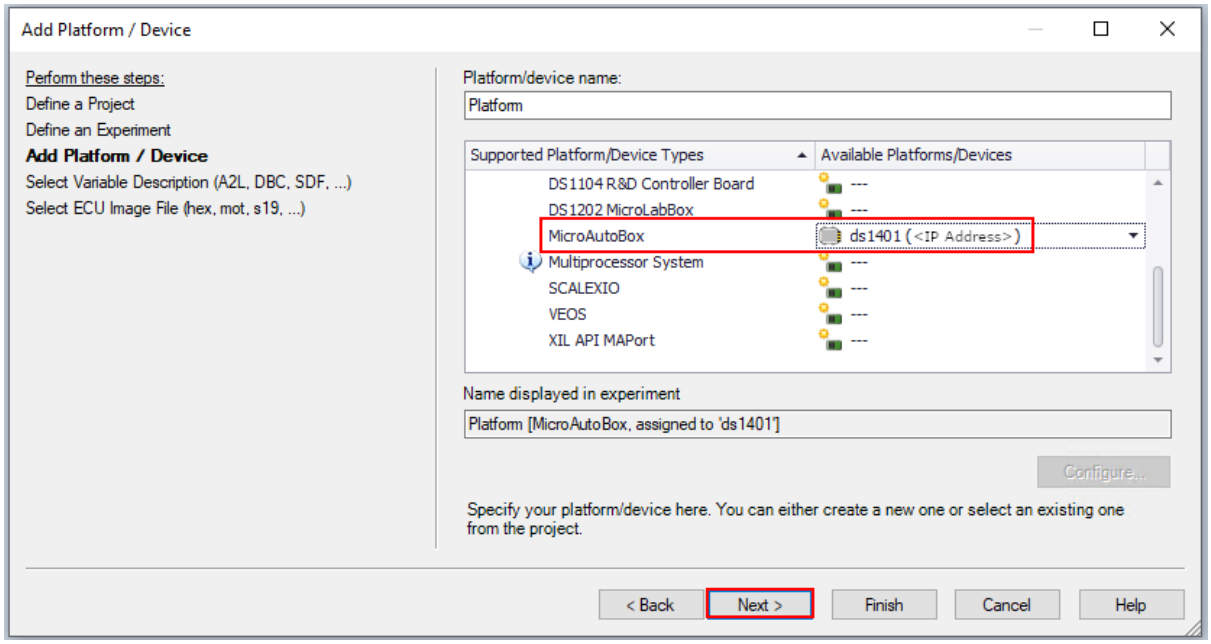


Figure 4.11: Select the MicroAutoBox platform

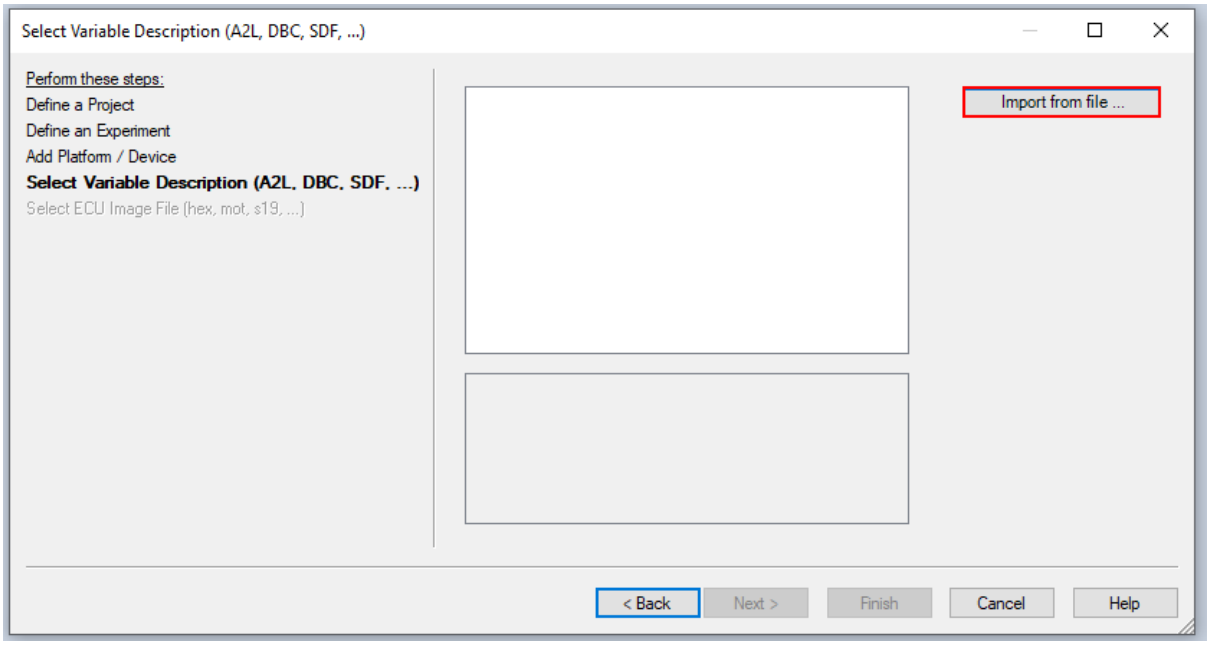


Figure 4.12: Import the variable description file

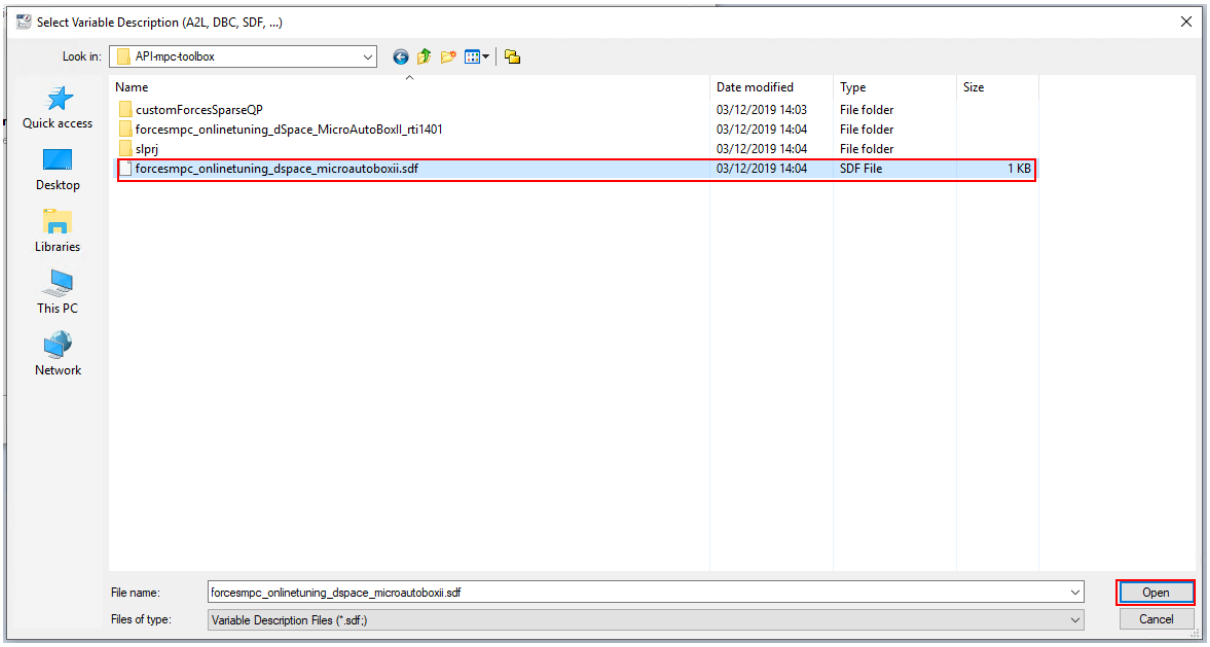


Figure 4.13: Select the sdf file with the variables description

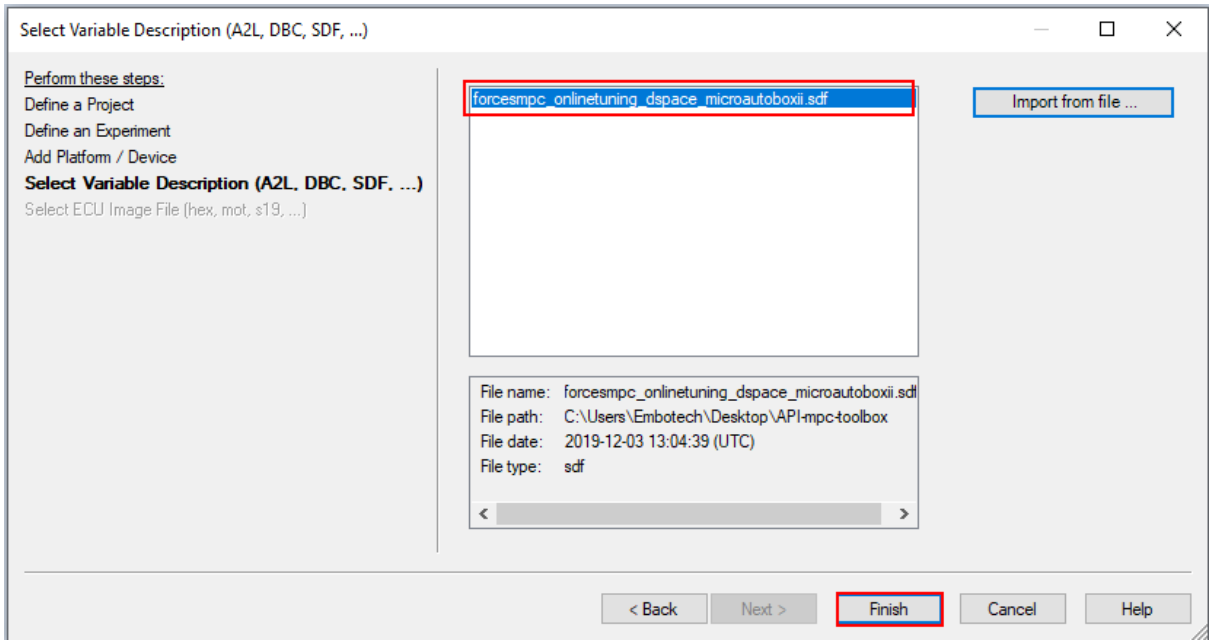


Figure 4.14: Click Finish to create the project

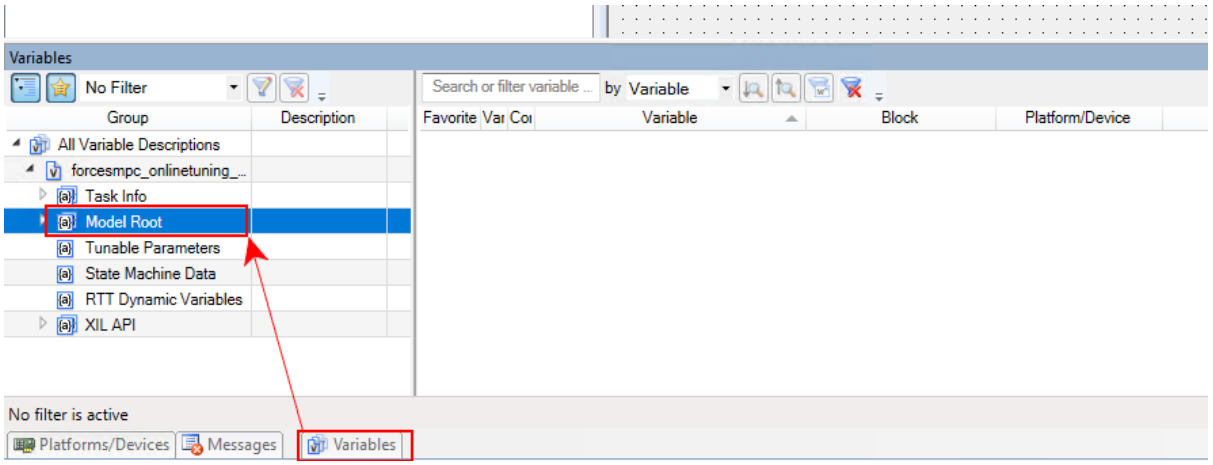


Figure 4.15: Find the model root in the variables tab

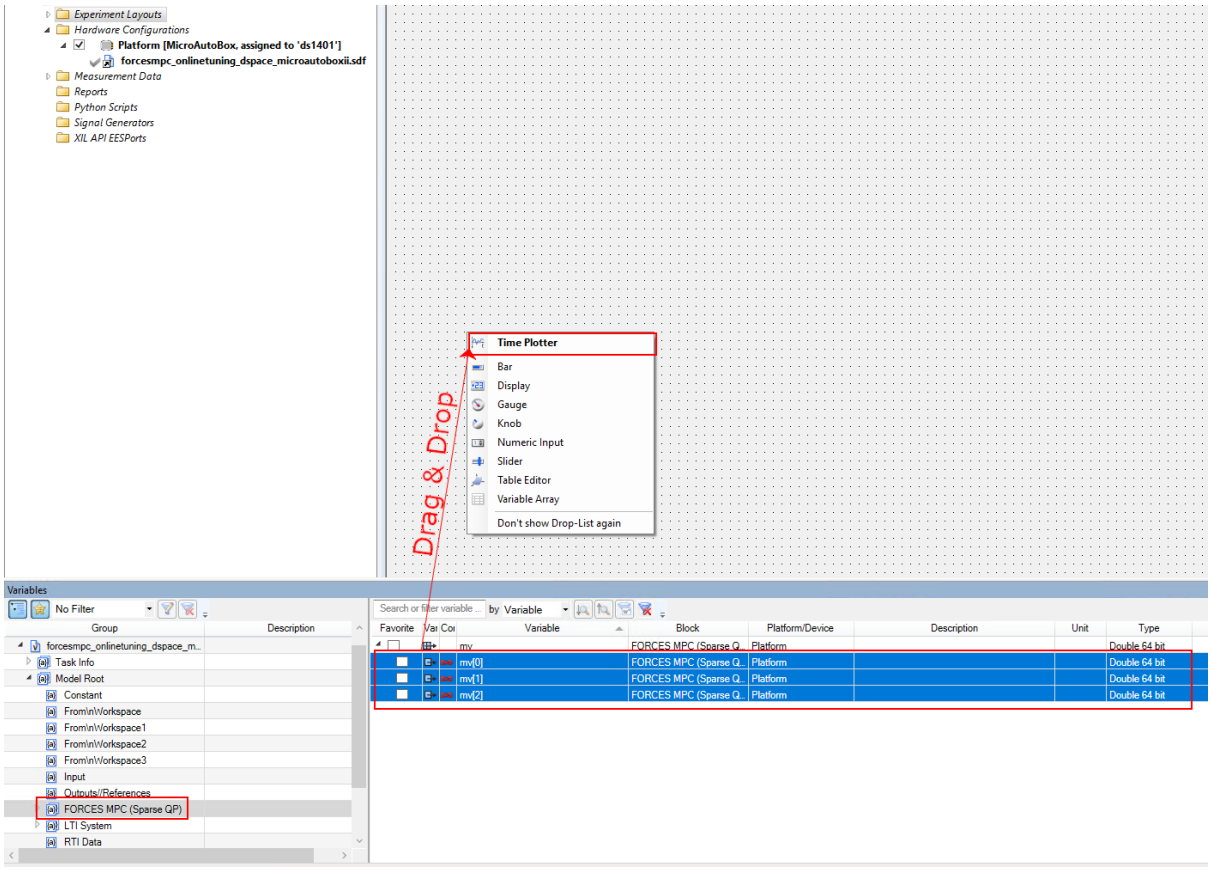


Figure 4.16: Add the variables as plots

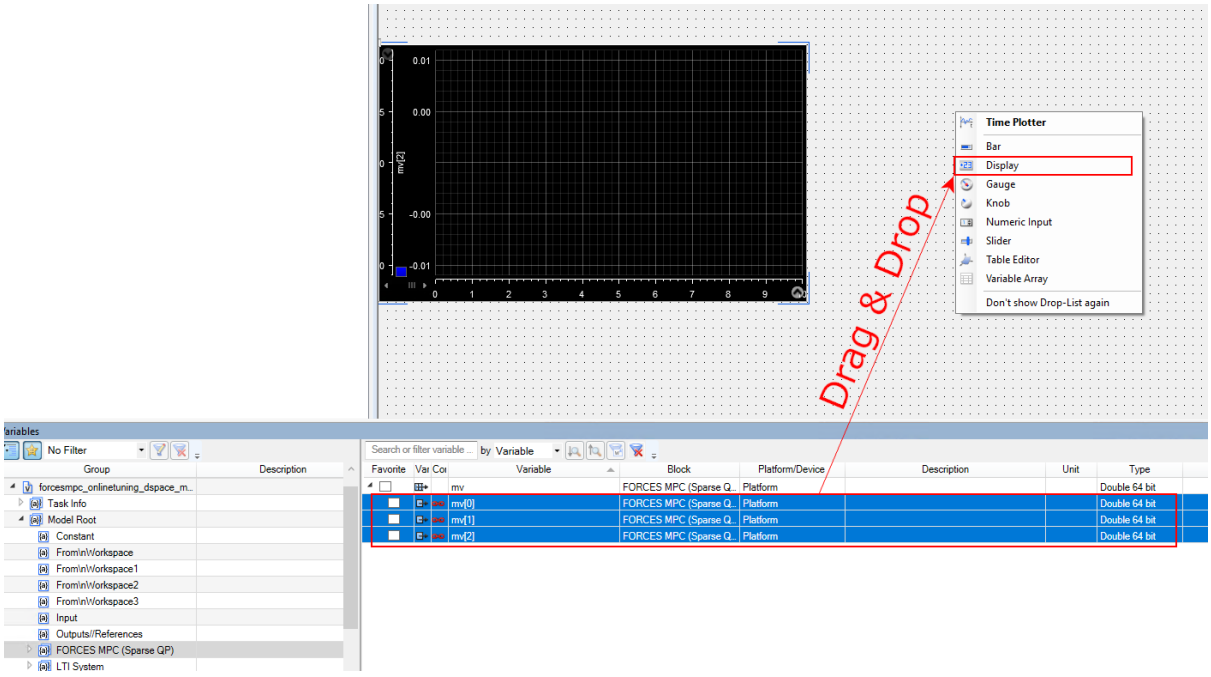


Figure 4.17: Add the variables as displays

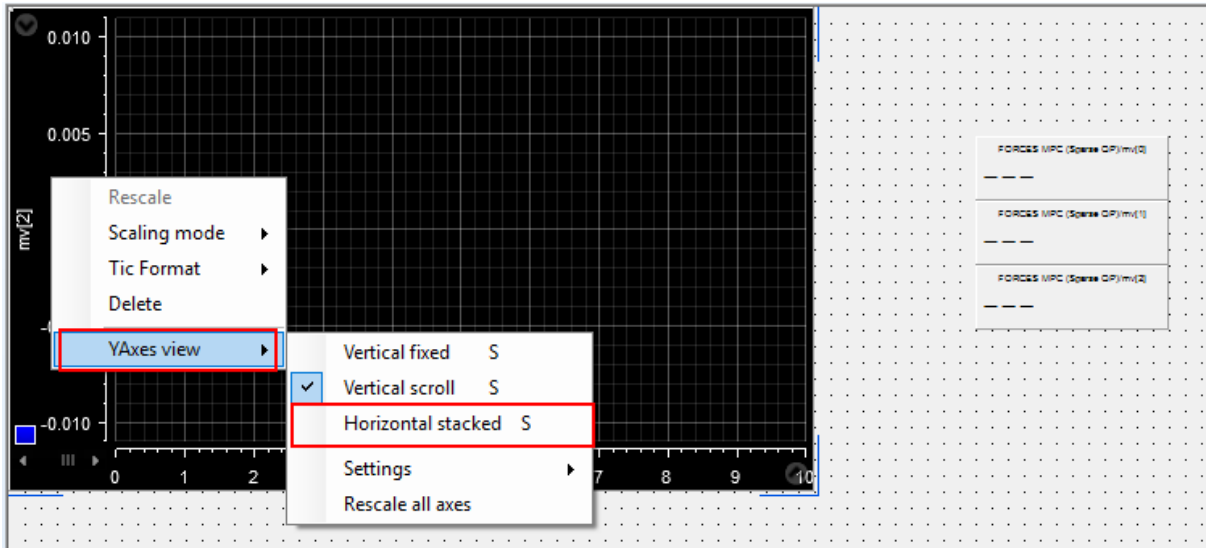


Figure 4.18: Select to show all the signals on the same plot with their own Y-axes

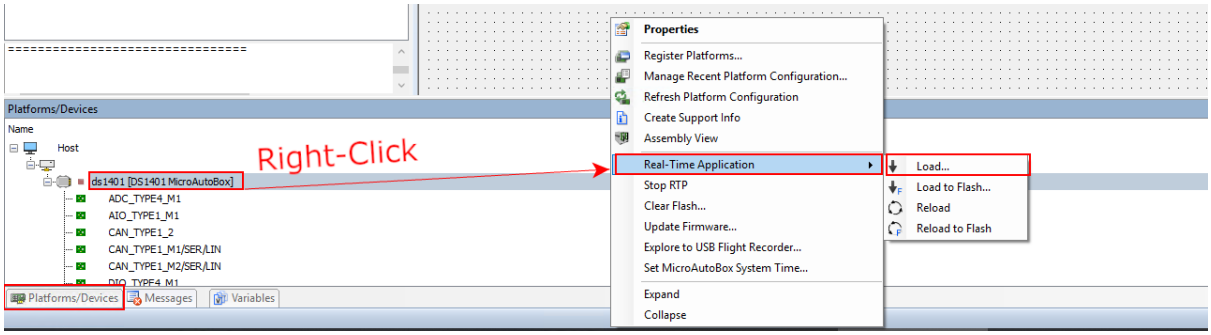


Figure 4.19: Load the application on the dSPACE MicroAutoBox II

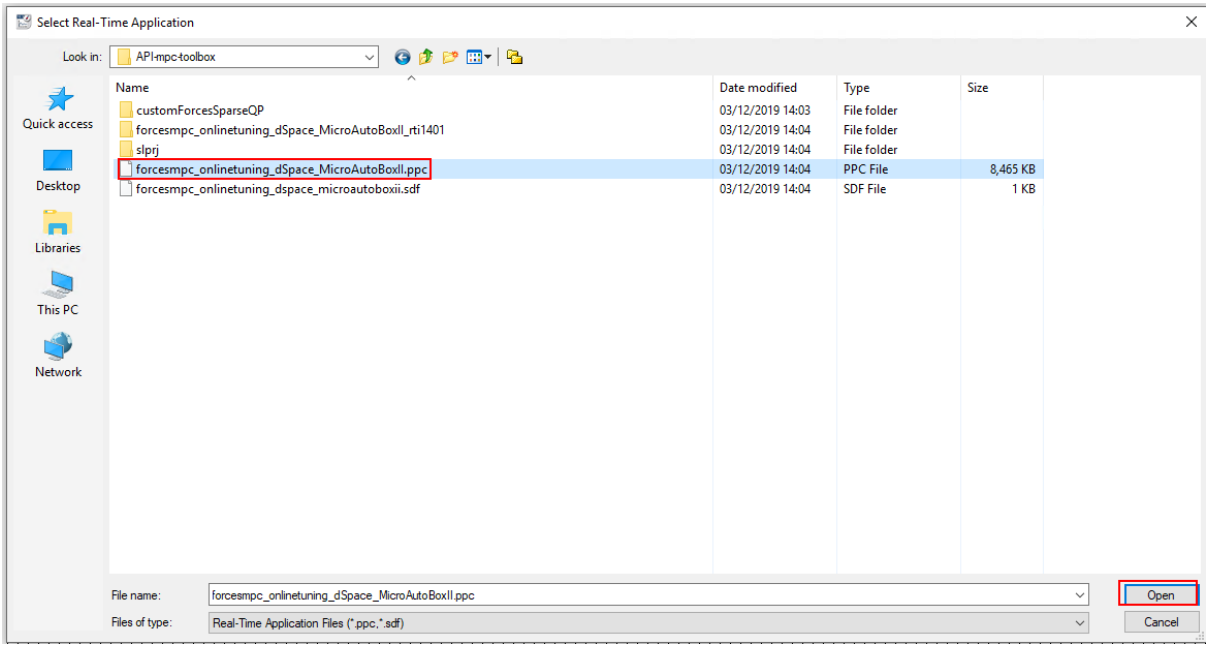


Figure 4.20: Select the executable to run the experiment

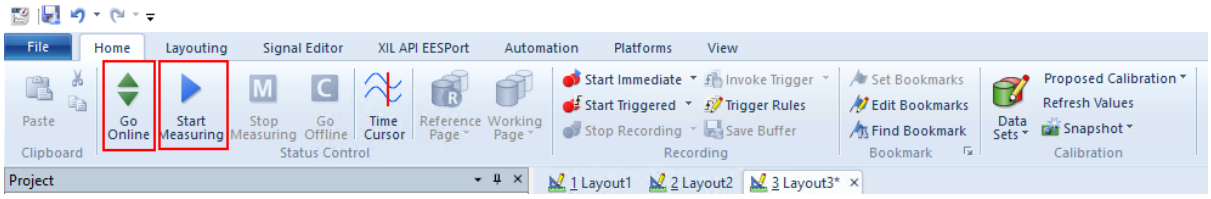


Figure 4.21: Buttons Go Online and Start Measuring to receive execution results

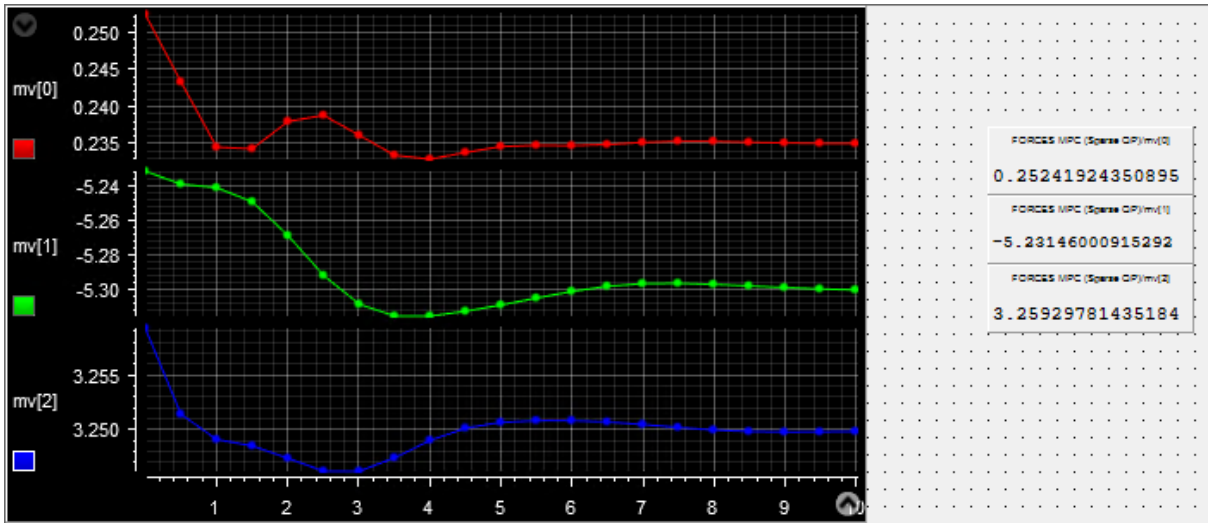


Figure 4.22: Plots and results from experiment on dSPACE MicroAutoBox II

- `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
- `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_simpleIti.m` demonstrates a simple LTI MPC designed
- `forcesmpc_linearize.m` is an example of linear MPC around an operating point of a nonlinear system.
- `forcesmpc_customqp.m` shows how to use the FORCES PRO dense QP solver as a custom solver in an MPC object
- `forcesmpc_run_onlinetuning.m` demonstrates how to run the MPC Simulink block.
- `forcesmpc_run_onlinetuning_dSpace_MicroAutoBoxII.m` demonstrates how to generate code for dSpace MicroAutoBox II using the MPC Simulink block.

The `forcesmpc_linearize.m` example is described in more details below. First, the linearized model and the operating point are defined.

```
%% 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 FORCES PRO solver can be generated.

```
% Create options structure for the FORCES PRO sparse QP solver
options = mpcToForcesOptions();
% Generates the FORCES PRO 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 FORCES PRO solver
  [mv, statedata, info] = mpcmoveForces(coredata, statedata, onlinedata);
  if info.ExitFlag < 0
    warning('Internal problem in FORCES PRO 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 4.23 and Figure Figure 4.24 respectively.

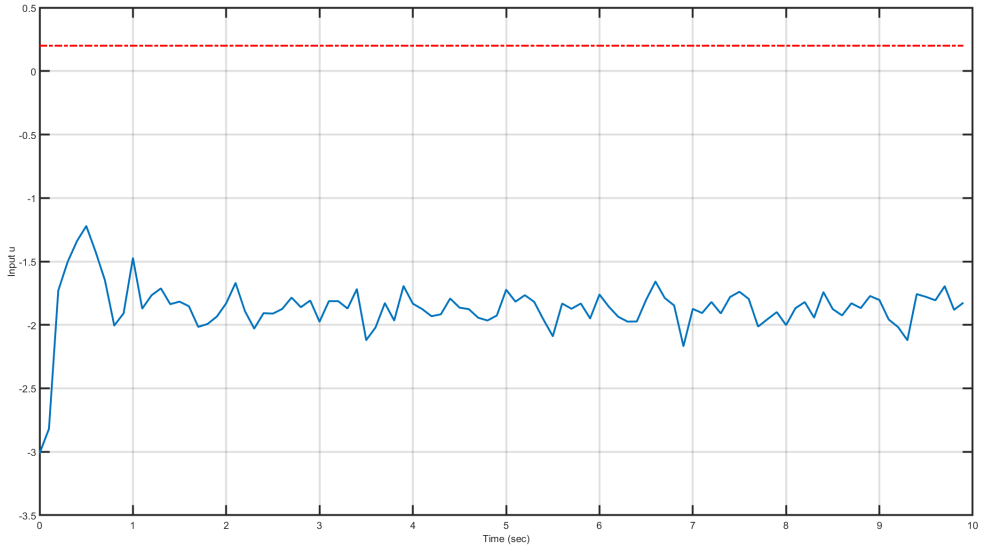


Figure 4.23: Manipulated variable computed by the FORCES PRO plugin.

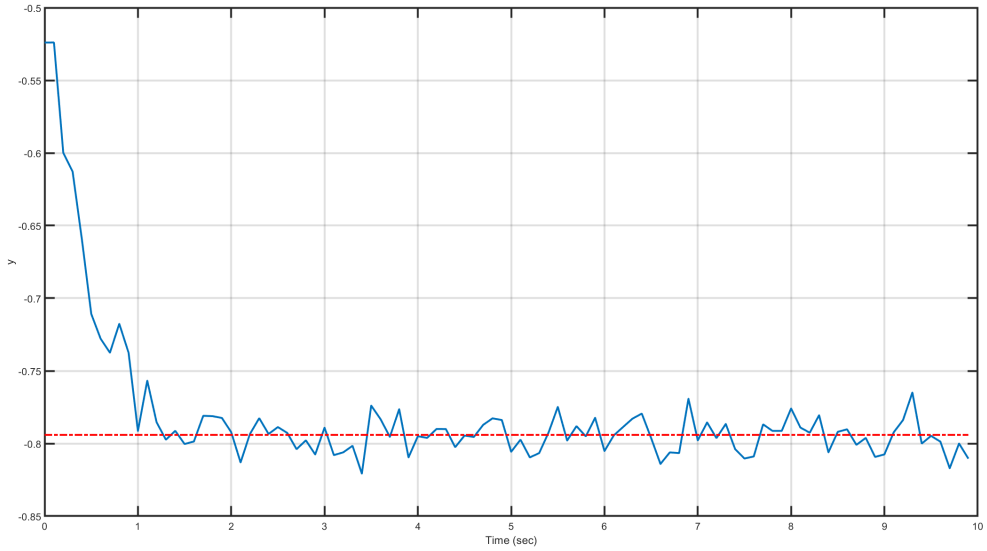


Figure 4.24: Output variable computed by the FORCES PRO plugin.

Chapter 5

Low-level interface

FORCES PRO 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.

5.1 Supported problem class

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

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

for $i = 1, ..., N$ and $k = 1, ..., M$. To obtain a solver for this optimization program using the FORCES PRO client, you need to define all data in the problem, that is the matrices H_i, A_i, $Q_{i,j}$, 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 $Q_{i,j}$ should all be positive definite.

Note: FORCES PRO supports all problem data to be parametric, i.e. to be unknown at code generation time. Read Section 9 to learn how to use parameters correctly.

In the following, we describe how to model a problem of the above form with FORCES PRO. First make sure that the FORCES PRO client is on the MATLAB/Python path. See Section 2 for more details on how to set up the MATLAB client and Section 2.3.

After the *PYTHONPATH* has been appropriately set up to include your FORCES PRO client directory (see Section 2.3.3), Python users have to import the FORCES PRO module:

```
import forcespro
```

5.2 Multistage struct

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

Matlab

Python

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

```
stages = forcespro.MultistagePoblem(N)
```

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.

5.3 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
```

```
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
```

5.4 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
```

```
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).

5.5 Equality constraints

The equality constraints for each stage, which are given by the matrices C_i, D_i and the vector c_i, have to be provided in the following form:

Matlab

Python

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

```
stages.eq[ i ]['C'] = ...
stages.eq[ i ]['c'] = ...
stages.eq[ i ]['D'] = ...
```

5.6 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*:

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
stages.ineq[ i ]['b']['lb'] = ...    # lower bounds
stages.ineq[ i ]['b']['ubidx'] = ... # index vector for upper bounds
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']*.

5.7 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
```

```
stages.ineq[ i ]['A'] = ... # Jacobian of linear inequality
stages.ineq[ i ]['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.

5.8 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
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 this 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.

5.8.1 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
```

5.9 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, FORCES PRO will add a branch-and-bound procedure to the standard convex solver it generates.

5.10 Declaring Solver Outputs

FORCES PRO 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 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 5.1: Possible string values for argument `maps2const`

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

5.10.1 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')
```

5.11 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 12.

Matlab

Python

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

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

5.12 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 accord-

ing 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.

Chapter 6

High-level Interface

The FORCES PRO 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.

Important: Starting with FORCES PRO `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.

6.1 Supported problems

6.1.1 Canonical problem for discrete-time dynamics

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

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

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 and final values, respectively. The initial and final values z_{init} and z_{final} can also be changed in real-time. At every stage k, the matrix F_k is a selection matrix that picks some coordinates in vector z_k.

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 FORCES PRO client, you need to define all functions involved (f_k, c_k, h_k) along with the corresponding dimensions.

6.1.2 Continuous-time dynamics

Instead of having discrete-time dynamics as can be seen in Section 6.1.1, 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 6.2.4 for more details.

6.1.3 Other variants

Not all elements in Section 6.1.1 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;
- that optimize over the initial value z_{init} and do not include the initial equality;
- that optimize over the final value z_{final} 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.

6.1.4 Function evaluations

The FORCES 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 consraints $h_k(z_k)$ and their Jacobians $\nabla h_k(z_k)$.

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

1. Automatic C-code generation of these functions from MATLAB using the automatic differentiation (AD) tool CasADi. This happens automatically in the background, as long as CasADi is found on the system. This process is hidden from the user, only standard MATLAB commands are needed to define the necessary functions. This is the recommended way of getting started with FORCES NLP. See Section 6.2 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 6.5 for details on how to provide own function evaluations and derivatives to FORCES PRO.

6.2 Expressing the optimization problem in code

When solving nonlinear programs of the type in Section 6.1.1, FORCES 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 our integration of FORCES with CasADi, an open-source package for generating derivatives. 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 6.5 to learn how to provide your own derivatives to FORCES NLP solvers. This section will describe the CasADi-based approach in detail, using either the MATLAB or the Python client of FORCES PRO. 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*.

6.2.1 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 seting 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** - FORCES PRO 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
```

6.2.2 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 6.2.7 for an example.

6.2.3 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. FORCES PRO 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 6.2.7 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]
    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*.

6.2.4 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 6.2.7 for an example.

Continuous-time

For continuous-time dynamics, FORCES PRO 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 intertia. 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 descretize the system for use with FORCES PRO we have to:

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

Matlab

Python

```
function xdot = continous_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

```
continous_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 FORCES PRO 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.

6.2.5 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
```

6.2.6 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])
```

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 FORCES PRO 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` and `model.ub` can be made parametric by leaving said fields empty and using the `model.lbidx` and `model.ubidx` fields to indicate on which variables lower and upper bounds are present. The numerical values will then be expected at runtime. 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)
```

(continues on next page)

(continued from previous page)

```
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["{: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 behaviour, 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,))
```

6.2.7 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(nlp.N) = 4;
model.objective{nlp.N} = @(z) -100*z(2);
model.lb{nlp.N} = [-3, 0, 0, 0];
model.ub{nlp.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 defining cell arrays as above for all stages, FORCES PRO 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 FORCES PRO 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.

6.3 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.nlp.Solver.from_directory('/path/to/solver')
```

6.3.1 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])
```

Important: When using the MINLP solver and defining outputs, all integer variables need to be specified as custom outputs.

6.4 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.nlp.Solver.from_directory("/path/to/generated/solver/")
solver.help()
```

6.4.1 Initial guess

The FORCES 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}
```

6.4.2 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.

6.4.3 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,))
```

6.4.4 Tolerances as real-time parameters

From FORCES *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-generatioon, 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.

6.4.5 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 Table 6.1. 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, in MATLAB, we suggest to use an assert statement:

Matlab

Python

```
assert(exitflag == 1, 'Some issue with FORCES solver');
```

```
assert exitflag == 1, "Some issue with FORCES solver"
```

Table 6.1: Exitflag values

Exitflag	Description
1	Local optimal solution found (i.e. the point satisfies the KKT optimality conditions to the requested accuracy).
0	Maximum number of iterations reached. You can examine the value of optimality conditions returned by FORCES to decide whether the point returned is acceptable.
-4	Wrong number of inequalities input to solver.
-5	Error occured during matrix factorization.
-6	`NaN` or `INF` occured during functions evaluations.
-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.
-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` if this exitflag is encountered (see *SQP specific codeoptions*).
-10	`NaN` or `INF` occured during evaluation of functions and derivatives. 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.
-11	Invalid values in problem parameters.
-100	License error. This typically happens if you are trying to execute code that has been generated with a Simulation license of FORCES PRO on another machine. Regenerate the solver using your machine.

6.5 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 FORCES NLP code generator will not check this.

6.5.1 Interface

Expected function signature

To obtain the necessary information, the FORCES NLP will automatically call a function supplied by the user. This function must have the following signature:

```
void myfunctions (
    double *x, /* primal vars */
    double *y, /* eq. constraint multiplers */
    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 */
)
```

In the MATLAB client, note that this function must have the same name as the file it is contained in, minus the file extension. In the above example, the function must therefore be stored in a file named *myfunctions.c*. Using the Python client, an arbitrary function name unrelated to the file name can be used.

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;
```

This will change the interface of the expected external function to:

```
void myfunctions (
    double *x, /* primal vars */
    double *y, /* eq. constraint multiplers */
    double *l, /* ineq . constraint multipliers */
    void *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 */
)
```

i.e. you can 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 */
```

(continues on next page)

(continued from previous page)

```
/* Set parameter pointer to your data structure */
mySolver_params params; /* Define solver parameters */
params.customParams = &p;

/* Call solver (assuming everything else is defined) */
mySolver_solv(&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.

Note: Using a custom parameters struct is currently not supported when using the Python client.

6.5.2 Supplying function evaluation information

In MATLAB, to let the code generator know about the path to the C files implementing the necessary function evaluations use:

```
model.extfuncs = 'C/myfunctions.c';
```

As noted above, the MATLAB client 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, 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", funcion="myfunctions")
```

Herein, the *add_auxiliary()* method is used to add any helper C source files or object files that should be compiled and liked 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.

6.5.3 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;
}
```

(continues on next page)

(continued from previous page)

```
/* eq constr */
if (c)
{
    vehicle_dyanmics (x, c);
}
/* jacobian equalities ( column major ) */
if ( nabla_c )
{
    vehicle_dyanmics_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 FORCES NLP solver to call the same interface with different pointers depending on the functions that it requires.

6.5.4 Matrix format

Matrices are assumed to be stored in dense column major format. However, only the non-zero components need to be populated, as FORCES NLP makes sure that the arrays are initialized to zero before calling this interface.

6.5.5 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')
```

6.5.6 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.

6.6 Mixed-integer nonlinear solver

From FORCES PRO `1.8.0`, mixed-integer nonlinear programs (MINLPs) are supported. This broad class of problems encompasses all nonlinear programs with some integer decision variables.

6.6.1 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. Forces Pro switches to the MINLP solver as soon as some variables are declared as integers in any stage. This information can be provided to FORCES PRO 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, FORCES PRO 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 FORCES PRO 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 FORCES PRO 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 6.2, 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 6.2.

Table 6.2: FORCES PRO 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`

- 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. FORCES PRO 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` settings 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.

Note: The MINLP solver is currently constrained to run on one thread on MacOS, meaning that *minlp.max_num_threads* is automatically set to *1* on MacOS.

Important: When generating a MINLP solver for MacOS the thread local feature (*codeoptions.threadSafeStorage*) is automatically set to *0* so if a dynamic library is used for a MINLP solver in a MacOS environment then one should not run at the same time more than one solvers linked to that library. A workaround for this would be to use the static library which is not bound by this restriction.

The FORCES PRO 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.timeOutBnB`, the maximum amount of time allowed for completing the search. Its default value is *1.0* seconds, if not set by the user.
- `minlp.parallelStrategy`, the method used for parallelizing the mixed-integer search (from FORCES-PRO 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.

6.6.2 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 FORCES PRO 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 FORCES PRO 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.

6.6.3 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 behaviour differs depending on the combinations of options. The different behaviours 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 behaviour.
- 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.

6.7 Sequential quadratic programming algorithm

The FORCES PRO 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 (6.1.1) must be *affine* functions of the variable z_k (not necessarily of p_k).

6.7.1 How to generate a SQP solver

To generate a FORCES PRO 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 FORCES PRO 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.

6.7.2 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 benefitial 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.

6.7.3 Controlling the initial guess at run-time

Upon the first call to the generated FORCES PRO SQP solver one needs to specify a primal initial guess (`problem.x0`, see also *Initial guess*). The default behaviour of the FORCES PRO

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`.

6.7.4 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 6.3: 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;
```

6.8 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 FORCES-PRO-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.

6.9 Examples

- *High-level interface: Basic example*: In this example, you learn the basics in how to use FORCES PRO 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 FORCES PRO 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 FORCES PRO 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 FORCES PRO SQP solver to control a robotic arm.

- *Controlling a DC motor using a FORCES PRO SQP solver*: This example describes how to apply the FORCES PRO SQP solver to control a DC motor.

Chapter 7

Simulating your custom controller in Simulink®

FORCES PRO provides a Simulink® interface for easy simulation of your custom controllers within existing Simulink® diagrams. Once code has been generated the block transforms into a new block with the appropriate number of ports for your specific configuration. 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 FORCES PRO block to other blocks in you Simulink diagram and run the simulation.

Watch an introductory video on how to use the FORCES PRO Simulink® interface here

7.1 Configuration of a custom linear MPC controller using the FORCES PRO Simulink® GUI

The Simulink® GUI for FORCES PRO is an easy and intuitive way to design model-based optimal controllers that can take decisions considering future information and system constraints. The general supported problem formulation is as follows:

Given a measurement or estimate of the current state of the system, x, and possibly:

- an estimate for an additive disturbance, w_k
- the previous control command, u_{prev},
- the output reference to track, $y_{ref,k}$

the controller decides the future control actions $u_0, u_1, \ldots, u_{N-1}$, and the resulting predicted state trajectory $x_1, x_2, \ldots, x_N$, over the prediction horizon, N, in order to optimize the control objectives

$$\sum_{k=0}^{N-1} (x_{k+1} - x_{ss,k})^T Q_k (x_{k+1} - x_{ss,k}) + (u_{k+1} - u_{ss,k})^T Q_k (u_{k+1} - u_{ss,k}) + \Delta u_k^T T_k \Delta u_k$$

where

$$\begin{pmatrix} A_k - I & B_k \\ C & 0 \end{pmatrix} \begin{pmatrix} x_{ss,k} \\ u_{ss,k} \end{pmatrix} = \begin{pmatrix} 0 \\ y_{ref,k} \end{pmatrix}$$

$$\Delta u_0 = u_0 - u_{prev},$$

$$\Delta u_0 = u_k - u_{k-1}, \quad k = 1, ..., N-1$$

subject to a linear mode of the system

$$x_1 = A_0 x + B_0 u_0 + w_0$$

$$\begin{aligned} x_{k+1} = A_k x_k + B_k u_k + w_k, &\quad \text{for all } k = 1, \ldots, N-1 \\ y_k = C x_k, &\quad \text{for all } k = 1, \ldots, N \end{aligned}$$

and satisfying the *system constraints*

$$y_k \in \mathbb{Y}_k, \quad \text{for all } k = 1, 2, \ldots, N$$

$$u_k \in \mathbb{U}_k, \quad \text{for all } k = 0, 1, \ldots, N-1$$

$$\Delta u_k \in \mathbb{V}_k, \quad \text{for all } k = 0, 1, \ldots, N-1$$

The settings for your particular controller can be specified by editing the mask of the FORCES PRO Simulink block. To start a new controller design copy the block in `LTI_MPC_lib.mdl` to your Simulink diagram and give a name to your controller. Double click on the new block and configure the different parameters as described here:

7.1.1 Model

Describe your linear state-space model of the system.

- *Time* : Choose whether your state-space model is described using differential equations or using a discrete update equation.
- *Type* : Choose whether your model has an extra affine term, i.e. is w present?
- *Sampling time* : If you are loading a continuous-time model specify the sampling time in seconds.
- *System matrices* : Specify the workspace variables describing the different system matrices A,B,C, and , if present, vector w.
- *Parameters* : Some variables are allowed to be parameters at design time, i.e. they can change dynamically during runtime. To allow this feature mark the appropriate check boxes to determine whether the parameter changes over the prediction horizon.
- *System dimensions* : If one or more system matrices are parameters you might need to specify any unresolved system dimensions.

7.1.2 Control Objectives

The control objectives are typically a trade-off between how well the controller tracks the output reference and how much input action it uses.

- *Tracking options*: Check if the controller is tracking an output reference or leave unchecked if the controller is regulating to the origin. If the controller is tracking a reference, specify whether the output reference y_{ref} will be provided, or whether the steady-state offset-free state and input references, x_{ss} and u_{ss}, have already been calculated. Also specify if the reference is changing over time and whether the changes are known ahead of time or not ($y_{ref,k} = y_{ref}$). If reference changes are known ahead of time, the controller can use this preview information to improve the control performance.
- *Input slew rate penalty* : Check if the controller should also attempt to minimize the actuator changes between control samples. If the checkbox is left unmarked, the weight matrix T is set to zero.

- *Terminal cost* : If the checkbox is marked the state penalty matrix for the last stage becomes $Q_N := P$, where P is the solution of the discrete-time Ricatti equation. Note that the matrix P can only be computed when matrices A, B, C, and R are known at design time, i.e. they are not runtime parameters. In general, having a terminal cost allows for a reduced prediction horizon but imposes certain restrictions on the optimization methods that can be used.
- *Control horizon* : Specify the number of samples that the controller looks into the future. In general, a longer control horizon can improve control performance but leads to longer computation times.
- *Weighting matrices* : Check if the weighting matrices on the outputs, Q, on the inputs, R, and on the input rates, T, are available and specify the corresponding workspace variables. If no weighting matrices are available specify the relative importance for tracking/regulation of the different outputs, inputs and slew rates. A high weight on an output tells the controller to focus on improving the tracking performance on that output. A high weight on an input tells the controller to use less of that input.
- *Parameters* : The penalty matrices can also be parameters at design time and change dynamically at runtime. To allow this feature mark the appropriate check boxes to determine whether the parameter changes over the prediction horizon.

7.1.3 System Constraints

Describe system limits that cannot be exceeded due to physical, safety, economic or regulatory reasons.

- *Constraint list* : Check which output, input, and slew rate constraints are present. For each constraint specify the upper and lower bounds. Note that an empty bound implies a one-sided constraint, e.g. $0 \leq u_1$.
- *Soft constraints* : Output constraints can be specified to be soft to prevent infeasible problems. In this case a slack variable, δ, is introduced resulting in the constraint.

$$\begin{aligned} -23\delta \leq & y_2 \leq 23 + \delta, \\ & \delta \geq 0. \end{aligned}$$

- *Parameters* : Upper and lower bounds can also be defined as runtime parameters. To allow this feature mark the appropriate check boxes to determine whether the parameter changes over the prediction horizon.

7.1.4 Estimator Settings

Describe additional characteristics for your customized solver.

- *Data type* : Choose the data type used by the solver. For some embedded platforms, floating-point computations (specially double precision) will incur significant computational delays. In the standard and premium versions of FORCES PRO fixed-point data types can lead to reduced computation times depending on the platform, but this imposes certain restrictions on the optimization methods that can be used.
- *Optimization method* : The basic version of FORCES PRO always uses a Primal-Dual Interior-Point (PDIP) method to implement the optimal controller. In the standard and premium versions of FORCES PRO one can select other alternative methods, such as ADMM and DFGM, that can lead to reduced computation times. One can also let FORCES PRO choose the most appropriate optimization method for your problem.
- *Number of iterations* : Specify the maximum number of iterations used in the optimization algorithm. One can also let FORCES PRO determine the number of iterations for your problem.

- *Method-specific options* : For some methods the user can choose values for certain parameters to tune the performance of the method.
- *Platform* : In the standard and premium versions of FORCES PRO one can choose the platform that the solver will run on to obtain customized code for the particular platform. For desktop based platforms, choose 'x86_64' for 64-bit platforms and 'x86' for 32-bit platforms. For embedded platforms, choose between 'x86', 'ARM Cortex M3 and M4', 'ARM Cortex A9', 'Tricore', 'PowerPC', or get a customized circuit design described in VHDL. Note that additional add-ons for FORCES PRO are required to generate code for different target platforms.
- *Description* : Add an optional description for your controller that can be used later to identify the settings for a particular controller instance in your web workspace.
- *Solve information* : Mark this check box to obtain runtime information from the solver that can be used to diagnose problems.

Once all the necessary solver options have been specified a custom solver for your controller can be built by executing the command `configure_block`. The command returns an error if any essential information is missing or if the license type is not valid. Note that this command transforms the block to make it ready for simulation. Once a controller has been generated you can change the configuration by double clicking the block and running `configure_block` again.

Several instance of the FORCES PRO block can exist in the same Simulink® diagram.

7.2 Getting Started - Basic MPC Regulation State Feedback Example

This example will show how to get started with the Simulink® interface of Forces Pro 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.

$$x_{k+1} = \begin{pmatrix} 0.7115 & -0.4345 \\ 0.4345 & 0.8853 \end{pmatrix} x_k + \begin{pmatrix} 0.2173 \\ 0.0573 \end{pmatrix} u_k$$

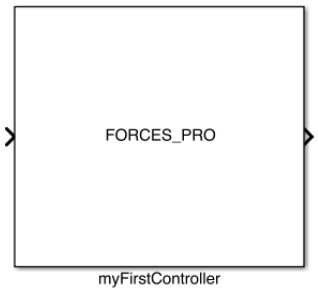
$$y_k = \begin{pmatrix} 0 & 1 \end{pmatrix} x_k$$

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. After downloading the files we can start with the design of the controller. First load the data from `myFirstController_data.mat` into the workspace and then open the Simulink® model `myFirstController_sim.slx`.

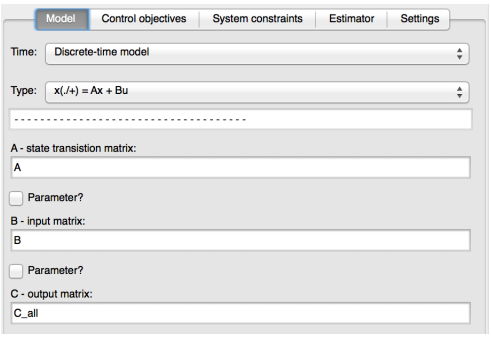
Then copy the FORCES PRO Simulink® block `MPC_lib_2012b.mdl` into your Simulink® diagram. Give the block a name. Here we will call it `myFirstController`.

We are now ready to configure the controller. Double-click on the block and go to the 'Model' tab to enter the details of the system that we want to control. The model described above has already been discretized with a sampling time of 0.1 seconds. We therefore choose 'Discrete-time model' and chose the type of state-space model (we have no additive term g in this example). Enter the state transition matrix A, the input matrix B and the output matrix C_{all}. Notice that we use C_{all}, which is just the identity matrix, instead of C, since we want to regulate both states, not just the output of the system.

We are now ready to configure the controller. Double-click on the block and go to the 'Model' tab to enter the details of the system that we want to control. The model described above has

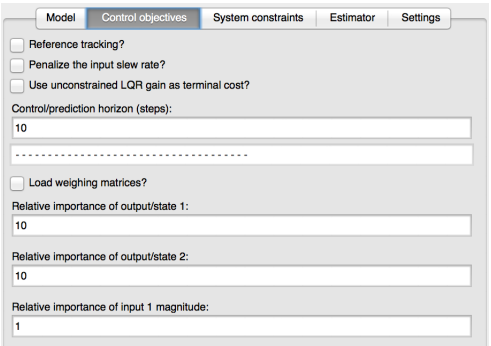


already been discretized with a sampling time of 0.1 seconds. We therefore choose 'Discrete-time model' and chose the type of state-space model (we have no additive term g in this example). Enter the state transition matrix A, the input matrix B and the output matrix C_{All}. Notice that we use C_{All}, which is just the identity matrix, instead of C, since we want to regulate both states, not just the output of the system.



In the 'Control Objective' tab we choose a prediction horizon of 10 steps, i. e. the controller looks 1 second into the future. We will input the relative weights manually. We weight the importance of regulating the states 10 times higher then reducing the use of the actuator.

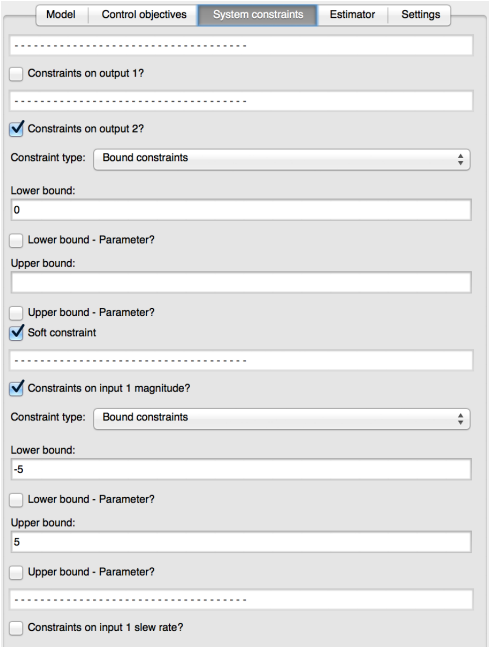
You are encouraged to change these weights and observe the effect on the control behaviour.



In the 'System Constraints' tab we 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. We enter the lower bound -5 and the upper bound 5. We can also check the option 'Soft Constraint' for the output constraint to prevent infeasibility problems in the solver.

Since we are designing a state feedback controller we will leave the only option in the 'Estimator' tab as 'State Feedback'. There will be no estimator built into the FORCES PRO block.

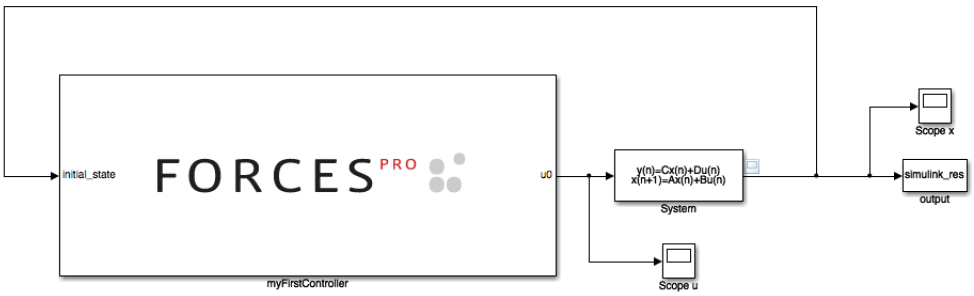
If we wish the controller to give information on the optimization process at each time step we check the option 'Get Solve Information' in the 'Settings' tab. The controller will have an additional output from which we can read this information.



We are now ready to configure the controller. Simply type

```
>> configure_block
```

in the MATLAB® command prompt. 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 FORCES PRO block is automatically updated and made ready for simulation on your Simulink® diagram. We can connect the ports of the controller to the rest of the system and run the simulation.

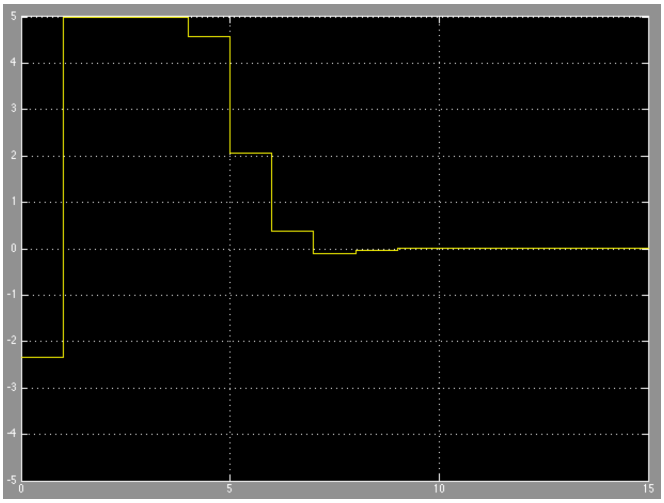


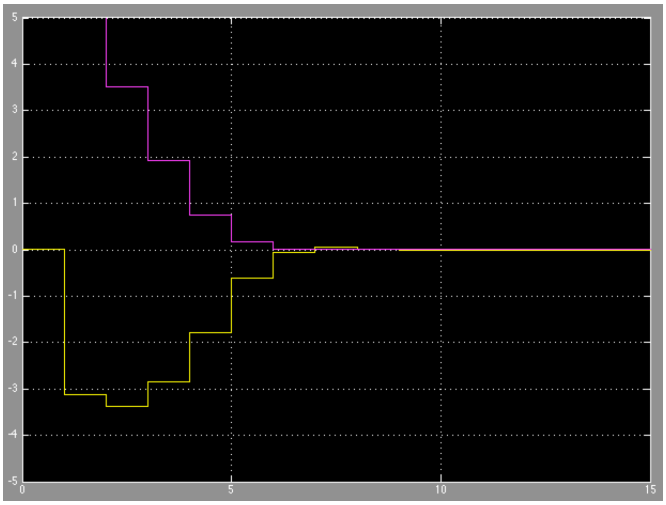
From the left plot we can see that the actuator remains in the allowed range. The right plot shows how the second state x_2 is always non-negative (purple graph in the right plot) and both states are regulated to zero.

7.3 Real-time control with the Simulink block

When a user generates a new solver from either the graphical Simulink interface, or the textual MATLAB or Python interfaces, several Simulink blocks are automatically created in the 'interfaces' folder. These blocks are useful to interface the solver with other Simulink models for simulation, or for deployment in embedded prototyping hardware using tools such as dSpace MicroAutobox or Simulink Coder.

In the following we describe the difference between the different available Simulink interfaces.


5
4
3
2
1
0
-1
-2
-3
-4
-5
0
5
10
15


5
4
3
2
1
0
-1
-2
-3
-4
-5
0
5
10
15

7.3.1 Input and Output Ports in the Compact Interface

For every solver, there are two Simulink interfaces generated: a standard interface; and a compact interface, which groups parameters and outputs. For problems with many parameters and outputs, the compact interface 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 FORCES PRO Simulink block can be checked by double-clicking the block and clicking the ‘Help’ button.

Chapter 8

Examples

8.1 How to

8.1.1 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 = [ui, xi+1] for i=1...N-1

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:

```
# FORCES multistage form
# assume variable ordering zi = [ui, xi+1] for i=1...N-1

stages = MultistageProblem(N) # get stages struct of length N

for i in range(N):

         # dimension
        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␣
↪these indices
        stages.ineq[ i ]['b']['ub'] = np.concatenate((umax,xmax),0) # upper bound␣
↪for this stage variable

        # equality constraints
```

(continues on next page)

(continued from previous page)

```
        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))
```

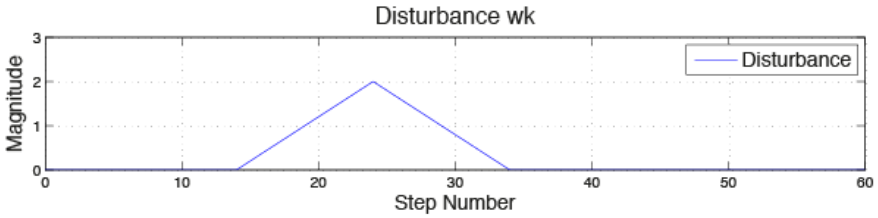
8.1.2 How to Incorporate Preview Information in the MPC Problem

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 FORCES PRO controller. At each time step k, a predicted disturbance for the next N steps is considered by the FORCES PRO 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. $x_0 = [0; 0]$.

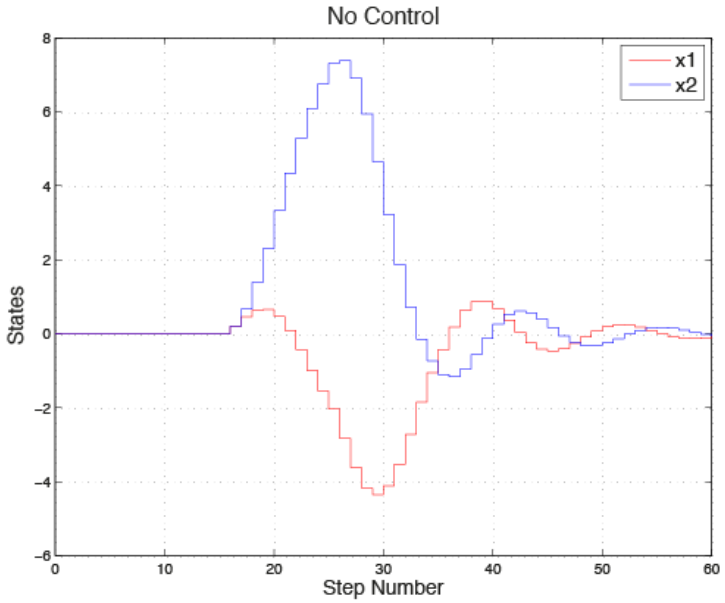


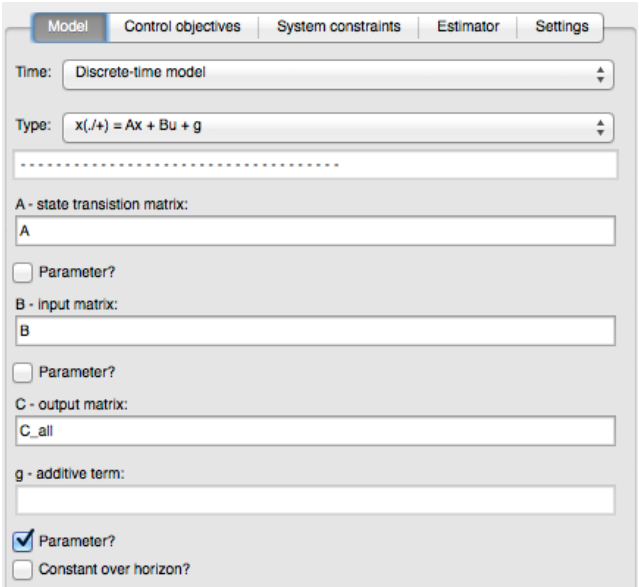
One can see that the disturbance drives the states far away from the desired value. In this example it is shown how FORCES PRO can significantly improve the dynamical behaviour by using the concept of 'preview' when such future information is available.

To implement a FORCES PRO controller with 'preview' one can either use the Simulink® interface or the MATLAB® interface. Here both options are presented. The result is the same.

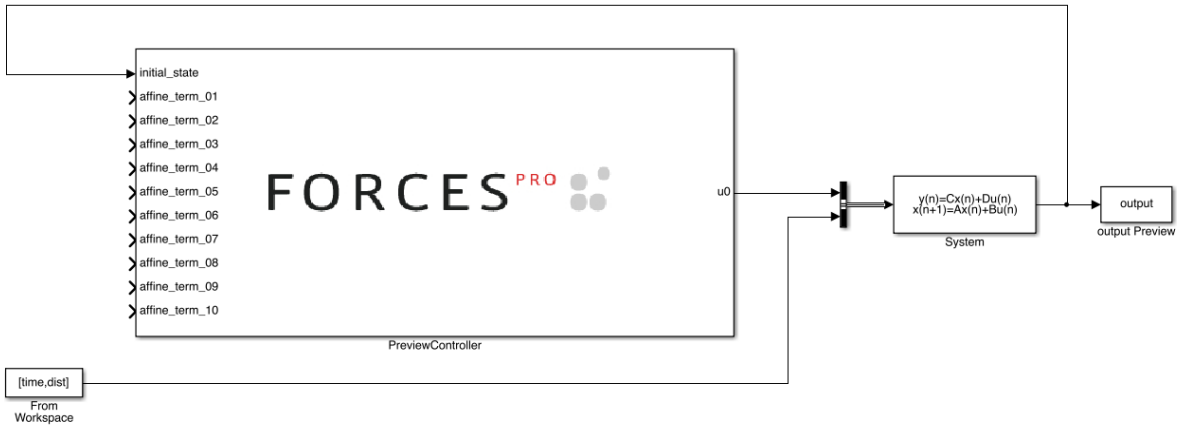
Use preview information in the Simulink® interface

To implement a FORCES PRO controller which makes use of preview information, drag the `LTI_MPC` block from the `LTI_MPC_lib` from the `FORCES_PRO` folder into the Simulink® model. After renaming the block, double click on it and chose in the tab `Model` the settings shown on the right side. In this example, the preview information comes through the additive term `g`. Check the option `parameter`. `g` is a parameter because at each time instant new disturbance predictions enter the controller. Also note that the additive term `g` is not constant over time, i.e. the disturbance prediction can vary over the prediction horizon.


No Control
x1
x2
States
Step Number
8
6
4
2
0
−2
−4
−6
0
10
20
30
40
50
60


Model
Control objectives
System constraints
Estimator
Settings
Time:
Discrete-time model
Type:
x(./+) = Ax + Bu + g
A - state transistion matrix:
A
Parameter?
B - input matrix:
B
Parameter?
C - output matrix:
C_all
g - additive term:
Parameter?
Constant over horizon?

The rest of the configuration of the FORCES PRO block is the same as for the design of a standard MPC regulator described *here*. After finishing the configuration, type `configure_block` to obtain a customized solver for your controller.



The controller is now configured and the number of inputs ports to the controller is determined by the length of the preview horizon.

Add the data of the disturbance and its preview from the workspace to model and start the simulation. To see the impact of using preview information see the section `Comparison of MPC with Preview and Standard MPC` 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. When running this example the code will automatically generate the Simulink block.

Use preview information in the MATLAB® interface

The same problem can be solved using the MATLAB® Interface. The multistage problem is constructed as shown in the simple example *here* and is then extended as shown below.

As in the Simulink® interface, 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 FORCES PRO, the system is now ready for simulation.

Comparison of MPC with Preview and Standard MPC

Figure 8.2 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 8.1 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.

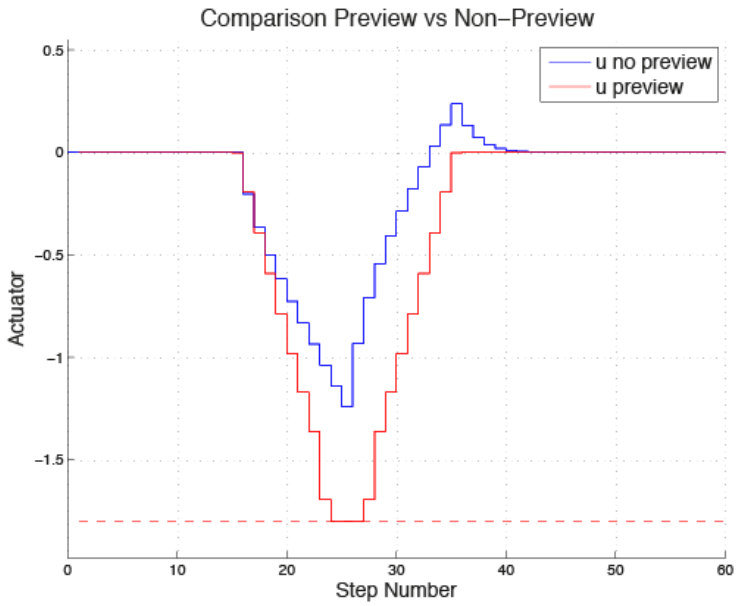


Figure 8.1: Comparison preview vs. non-preview

8.1.3 HOW TO: Implement an MPC Controller with a Time-Varying Model

Introduction

This 'HOW TO' explains how FORCES PRO 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 oscillator.

$$\text{System 1:}\quad x_{k+1} = \begin{pmatrix} 0.7115 & -0.6 \\ 0.6 & 0.8853 \end{pmatrix} x_k + \begin{pmatrix} 0.2173 \\ 0.0573 \end{pmatrix} u_k$$

$$\text{System 2:}\quad x_{k+1} = \begin{pmatrix} 0.9 & 0.5 \\ 0.5 & 1 \end{pmatrix} x_k + \begin{pmatrix} 0 \\ 0.0666 \end{pmatrix} u_k$$

$$\text{System 3:}\quad x_{k+1} = \begin{pmatrix} 0.7115 & -0.5 \\ 0.5 & 1 \end{pmatrix} x_k + \begin{pmatrix} 0.5 \\ 0.01 \end{pmatrix} u_k$$

$$\text{System 4:}\quad x_{k+1} = \begin{pmatrix} 0 & 0.9 \\ -1 & 0 \end{pmatrix} x_k + \begin{pmatrix} 0 \\ 0.2 \end{pmatrix} u_k$$

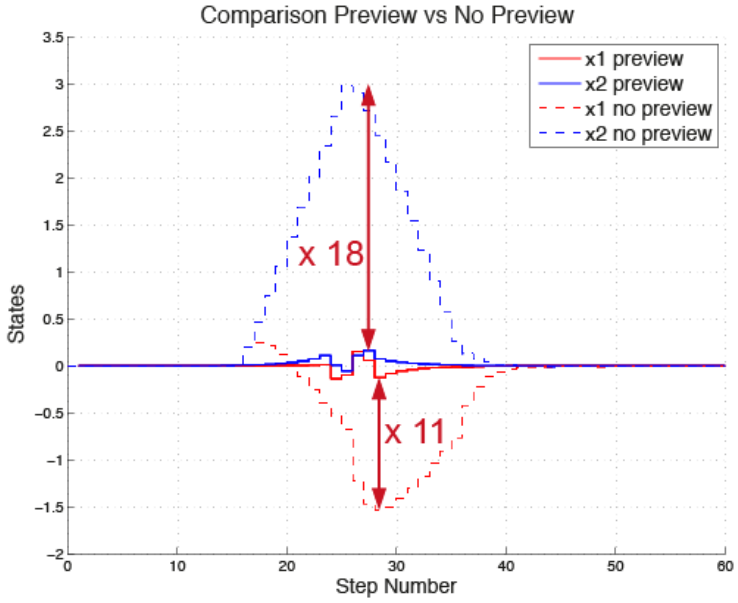
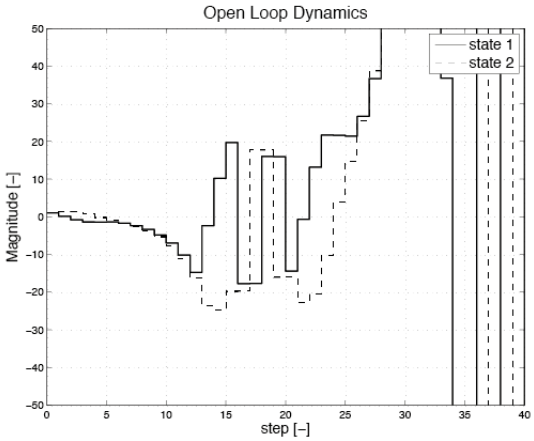


Figure 8.2: Comparison preview vs. no preview

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 FORCES PRO 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 FORCES PRO MATLAB® Interface.

Implementation

The FORCES PRO 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 FORCES PRO.

```
%% 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 Equlity
        % 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

% 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 8.3 and Figure 8.4 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 contraints. 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 FORCES PRO time-varying controller the costs is reduced to 457.5. This is a cost reduction of almost 80%.

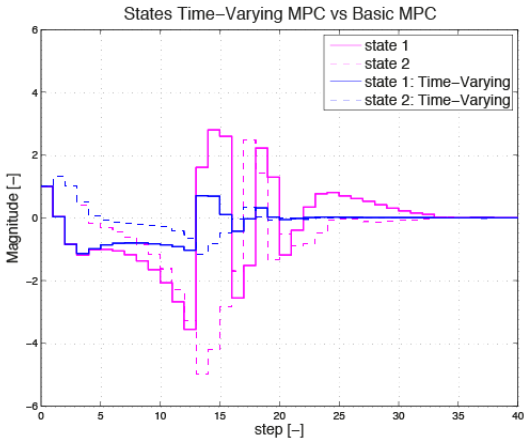


Figure 8.3: States Time-varying MPC vs. basic MPC

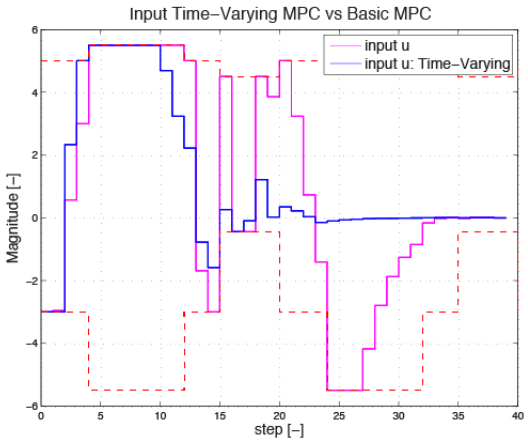


Figure 8.4: Input Time-varying MPC vs. basic MPC

8.1.4 How to Implement 1-Norm and Infinity-Norm Cost Functions

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 minimising 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 FORCES PRO. In particular, note the changes to the cost function and the introduction of polytopic constraints.

```
%% FORCES multistage form
% assume variable ordering zi = [ui, xi+1, ei] for i=1...N-1

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 )
```

(continues on next page)

(continued from previous page)

```
                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 minimise 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

$$\begin{aligned} &\text{minimize} && \|Qx_i\|_\infty \\ &\text{subject to} && \text{constraints} \end{aligned}$$

is transformed into the problem

$$\begin{aligned} &\text{minimize} && \epsilon \\ &\text{subject to} && \pm Qx_i \leq \mathbf{1}^T\epsilon \\ & && \text{constraints} \end{aligned}$$

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 FORCES PRO. 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 = [ui, xi+1, ei] for i=1...N-1

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)];
        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.

8.1.5 HOW TO: Implement Rate Constraints

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 FORCES PRO, 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, xhat_{i+1}] for i=1...N-1

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)
```

(continues on next page)

(continued from previous page)

```
        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
        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 8.5, Figure 8.6 and Figure 8.7 show how the controller regulates both states to zero while $\hat{u}$ and u remain in the required range.

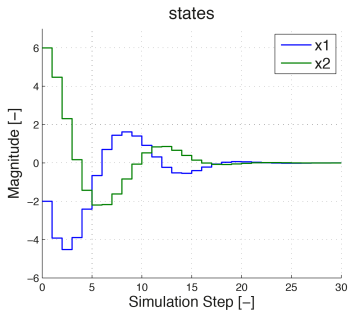


Figure 8.5: The states are both regulated to zero. No constraints are imposed on the states.

In Figure 8.6 and Figure 8.7 one sees how the input signal is maximally increased in the be-

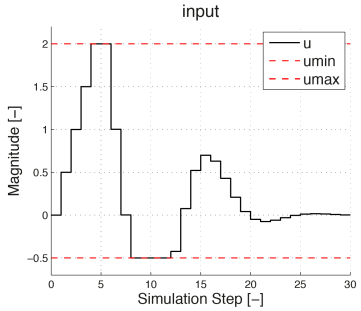


Figure 8.6: Plot of u

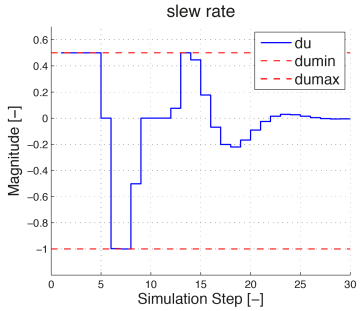


Figure 8.7: Plot of du

ginning with a slew rate of 0.5, until it reaches its upper bound of 2. In the figure on the right 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.

8.1.6 Binary MPC Example

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 FORCES PRO 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 FORCES PRO. 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;

% assume variable ordering zi = [delta_i; xi+1] for i=1...N-1
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.

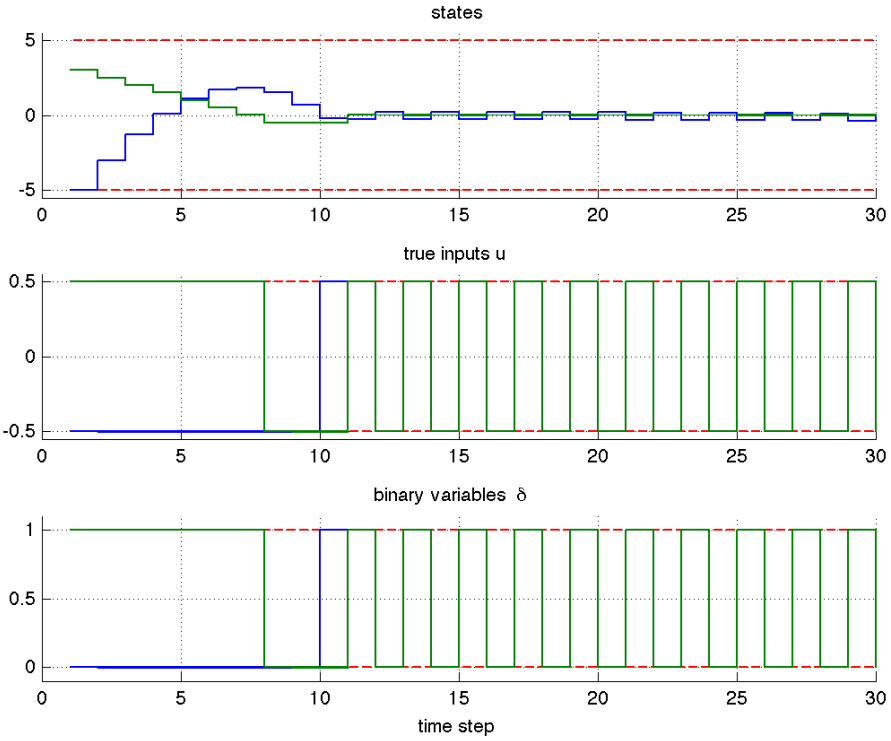
Simulation result

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

$$\begin{aligned} \tilde{R} &= \text{diag}(u_{max} - u_{min})R\text{diag}(u_{max} - u_{min}) \\ \tilde{f} &= 2R\text{diag}(u_{max} - u_{min})u_{min} \end{aligned}$$

8.2 Y2F interface: 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. Here we present the problem formulation with YALMIP, how you can use Y2F to easily generate a solver with FORCES PRO, 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).

8.2.1 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

% constraints
umin = -0.5;      umax = 0.5;
xmin = [-5; -5]; xmax = [5; 5];
```

8.2.2 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.

8.2.3 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.

8.2.4 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.

8.2.5 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
```

8.2.6 Results

The results of the simulation are presented in Figure 8.8. 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.

8.2.7 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 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 `optmizerFORCES` that they are parameters:

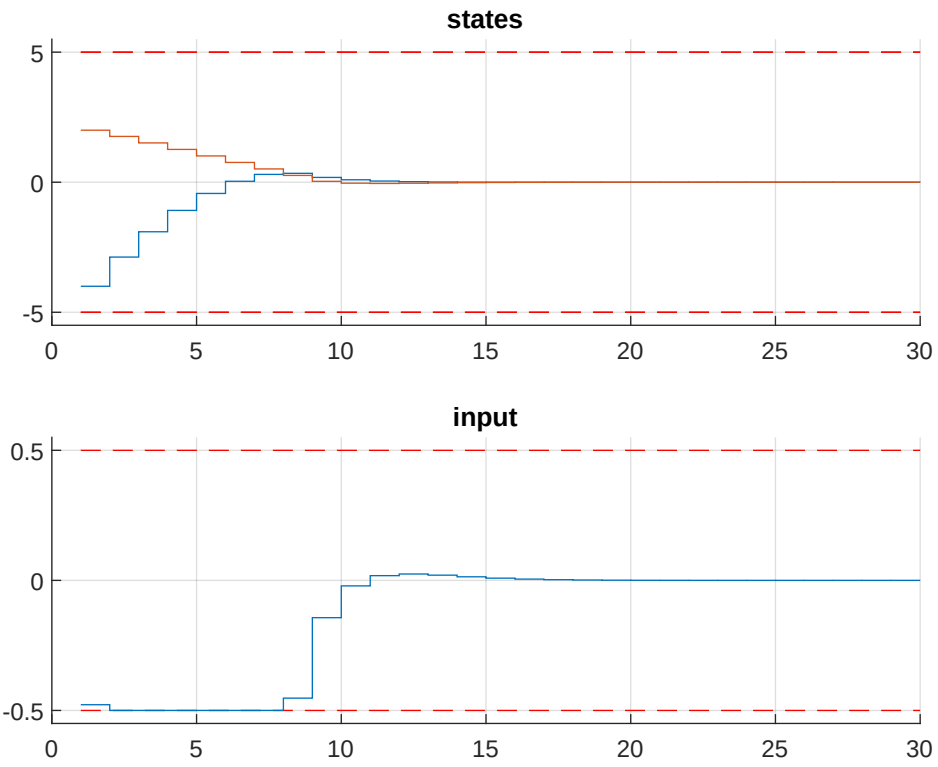


Figure 8.8: Simulation results of the states (top, in blue and red) and input (bottom, in blue) over time. The state and input constraints are plotted in red dashed lines.

```
% 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.

8.2.8 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 `optmizerFORCES` that they are parameters:

```
A = sdpvar(nx,nx,'full'); % system matrix - parameter
B = sdpvar(nx,nu,'full'); % input matrix - parameter

% [... 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.

8.2.9 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 `optmizerFORCES` 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.

8.3 Low-level interface: Active Suspension Control

8.3.1 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.

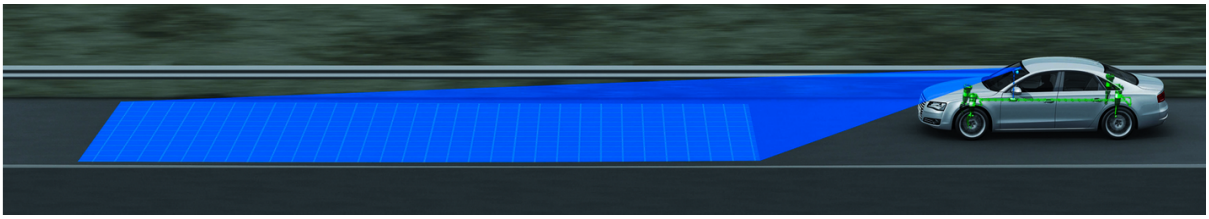


Figure 8.9: Figure borrowed from [GörSch]

The preview information can be used to improve the riding comfort, i. e. minimize the heave, pitch and roll accelerations, by actively controling 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 $\dot{w}$.

$$x := \begin{pmatrix} \text{heave displacement [m]} \\ \text{pitch angle [rads]} \\ \text{roll angle [rads]} \\ \text{heave velocity [m/s]} \\ \text{pitch rate [rads/s]} \\ \text{roll rate [rads/s]} \end{pmatrix}$$

$$u := (\text{active spring displacements [m]})$$

$$y := \begin{pmatrix} \text{heave acceleration [m/s}^2\text{]} \\ \text{pitch acceleration [rads/s}^2\text{]} \\ \text{roll acceleration [rads/s}^2\text{]} \end{pmatrix}$$

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:

$$\dot{x}(t) = Ax(t) + B_u u(t) + B_w \begin{pmatrix} w(t) \\ \dot{w}(t) \end{pmatrix}$$

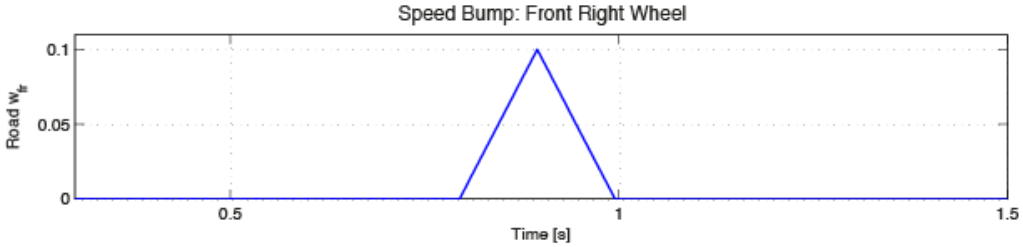
$$y(t) = Cx(t) + Du(t)$$

In the following it is shown how the FORCES PRO 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.

8.3.2 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.



8.3.3 Implementation of Preview Information

This is a linear MPC problem with lower and upper bounds on inputs and a terminal cost term:

$$\begin{aligned} \text{minimize} \quad & x_N^T P x_N + \sum_{i=0}^{N-1} x_i^T Q x_i + u_i^T R u_i \\ \text{subject to} \quad & x_0 = x \\ & x_{i+1} = Ax_i + Bu_i + B_w w_i + B_w \dot{w}_i \\ & \underline{u} \leq u_i \leq \overline{u} \end{aligned}$$

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');
% Paramteres: 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
% FORCES PRO multistage form
% assume variable ordering zi = [ui; xi+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
```

(continues on next page)

(continued from previous page)

```
    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 < N )
            parameter(i+1) = newParam(['pre',num2str(i+1),'_w'],i+1,'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.

8.3.4 Comparison of Passive Vehicle and Active Suspension Control Using Preview Information

In Figure 8.10, Figure 8.11 and Figure 8.12, the accelerations in the direction heave, pitch and roll respectively are depicted. The blue graphs represent the controlled outputs while the red ones show the uncontrolled accelerations. One can see that the vertical dynamics of the vehicle are reduced substantially. There are smaller maximal accelerations and also less time is required to regulate the accelerations back to zero.

Applying Model Predictive Control with Preview using FORCES PRO 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 8.13, Figure 8.14, Figure 8.15 and Figure 8.16 below show the input

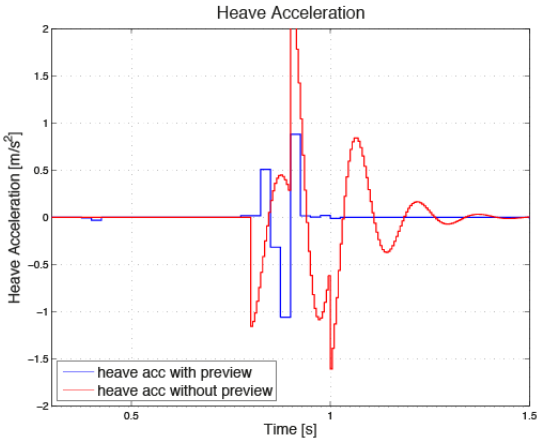


Figure 8.10: Acceleration in heave direction

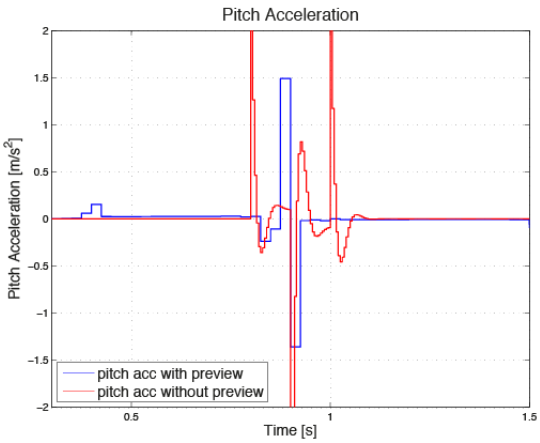


Figure 8.11: Acceleration in pitch direction

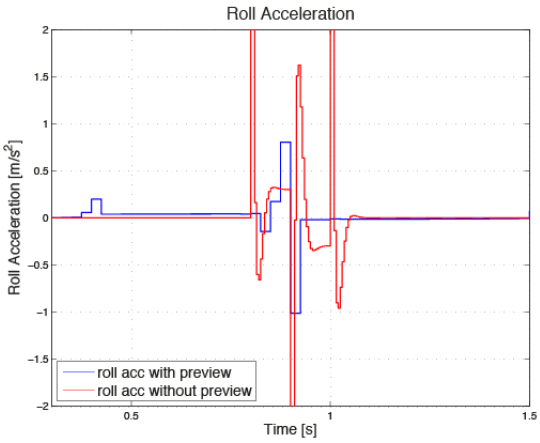


Figure 8.12: Acceleration in roll direction

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.

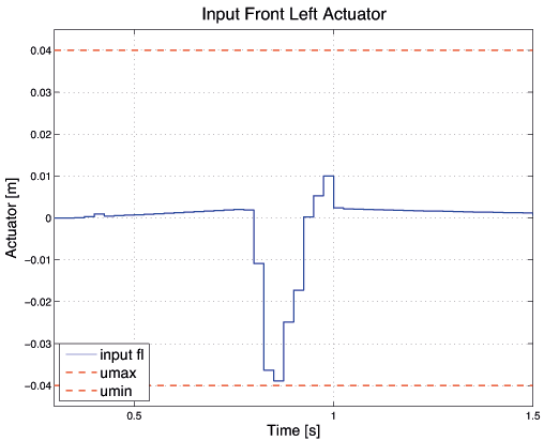


Figure 8.13: Input front left actuator

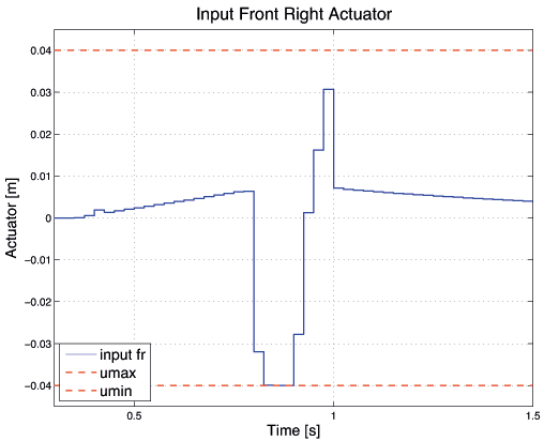


Figure 8.14: Input front right actuator

8.4 Low-level interface Robust estimation (Kalman filter)

8.4.1 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.

$$x^+ = Ax + Bu + v = \begin{pmatrix} 1-\alpha_1 & 0 & 0 \\ 0 & 1-\alpha_2 & 0 \\ \alpha_+ & \alpha_2 & 1-\alpha_3 \end{pmatrix} x + \begin{pmatrix} 0.5 \\ 0.5 \\ 0 \end{pmatrix} u + v$$

$$z = Hx + w + y = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} x + w + y$$

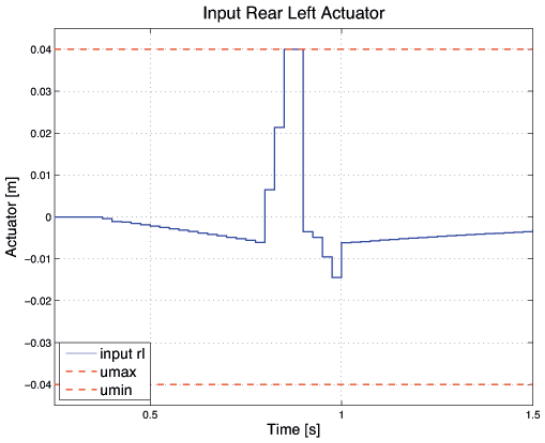


Figure 8.15: Input rear left actuator

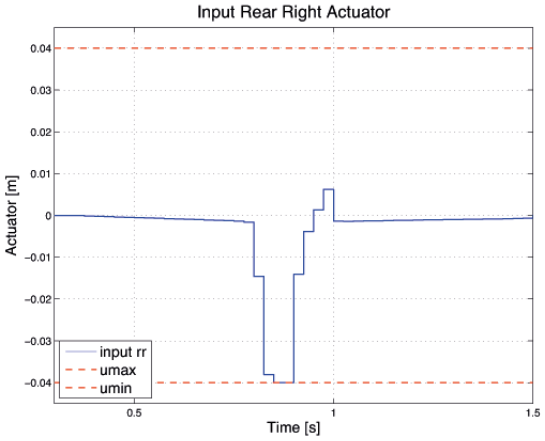
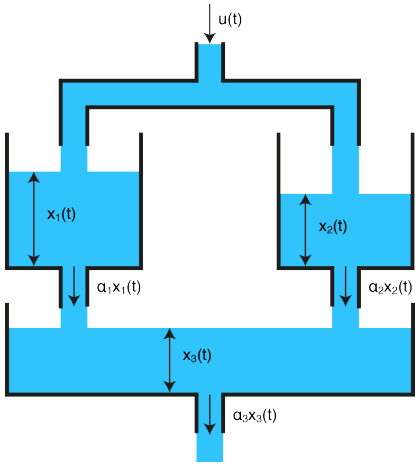


Figure 8.16: Input rear right actuator

The states of the system are $x = \begin{pmatrix} x_1 & x_2 & x_3 \end{pmatrix}^T$ 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 FORCES PRO, 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.



8.4.2 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 esimate $x^m(k)$. 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-penaltiy 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 FORCES PRO, 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 FORCES PRO. 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 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

% Ploytopic 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.

8.4.3 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 FORCES PRO.

```
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)*x_p_RKF);...
                                        zeros(nw,1);...
```

(continues on next page)

(continued from previous page)

```
                                        zeros(ny,1);...
                                        ones(ne,1)];
        problem.z_i = z(:,i);
        [solverout,exitflag,info] = Robust_KF(problem);
        solve_time(1,i-1) = info.solvetime;
        x_hat_RKF(:,i) = solverout.x_hat_RKF;
        P_hat_RKF(:,:,i) = P_p_RKF(:,:,i);
end
```

In the plots in Figure 8.17, Figure 8.18 and Figure 8.19 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.

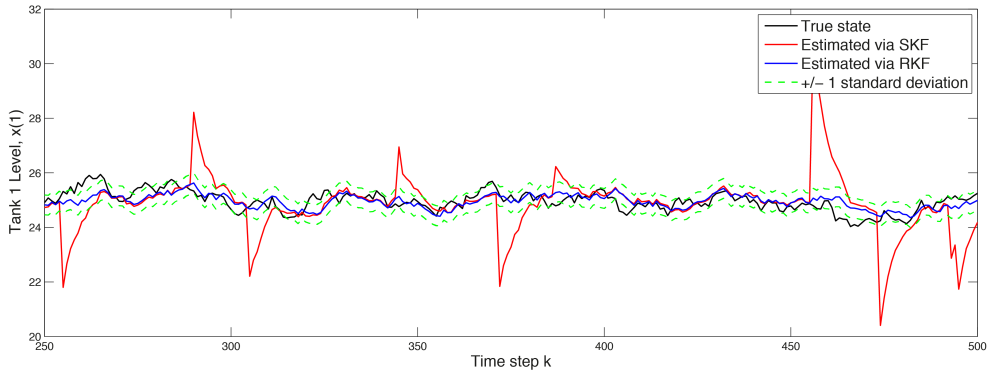


Figure 8.17: Estimated state $x(1)$

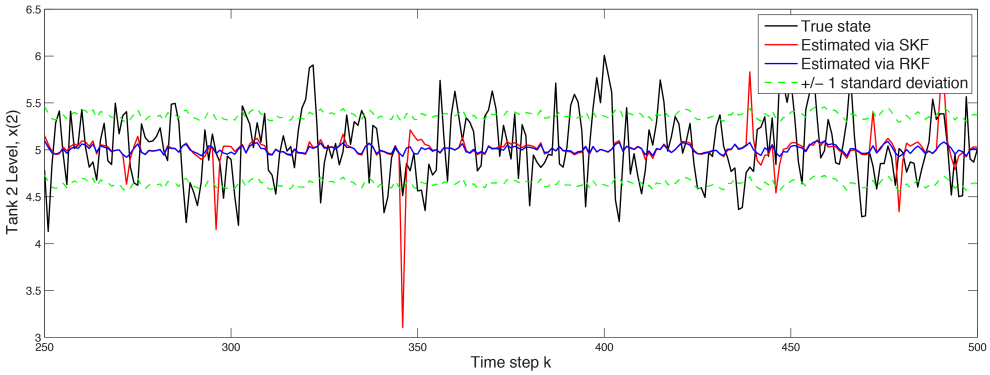


Figure 8.18: Estimated state $x(2)$

The impact on the RMS error magnitude of the robust Kalman Filter can be seen in the plots in Figure 8.20, Figure 8.21 and Figure 8.22. 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). Applying online optimization with FORCES PRO improves the quality of the state estimations significantly.

With FORCES PRO 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 $\sim 0.1ms$.

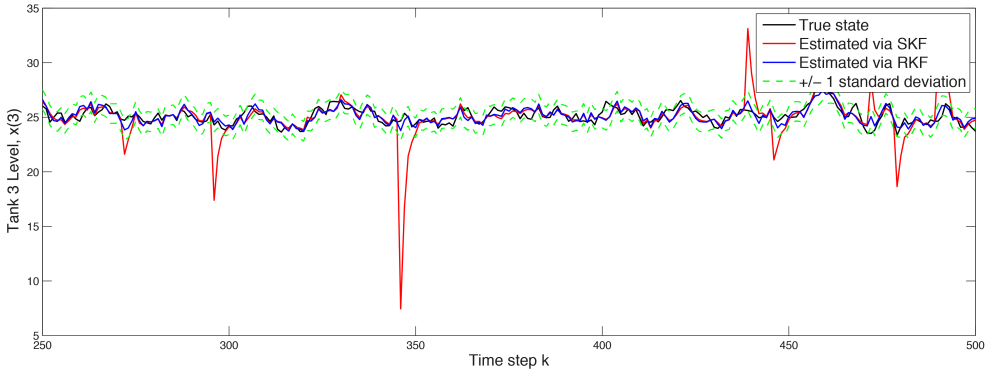


Figure 8.19: Estimated state $x(3)$

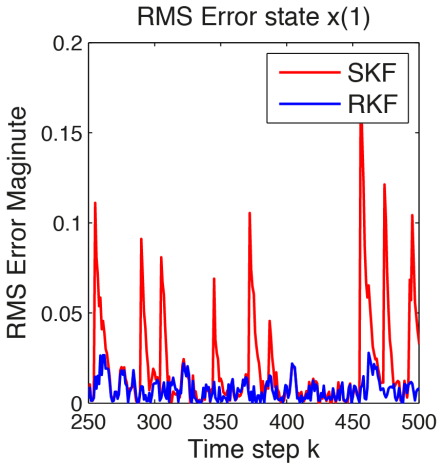


Figure 8.20: RMS error for $x(1)$

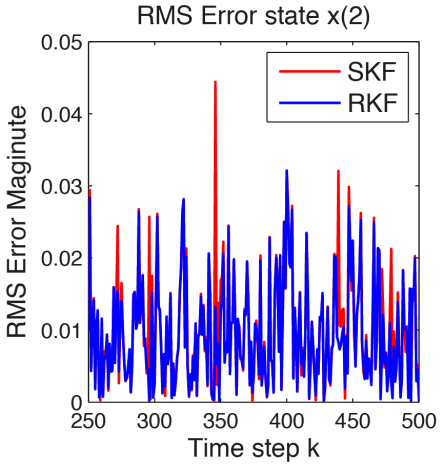


Figure 8.21: RMS error for $x(2)$

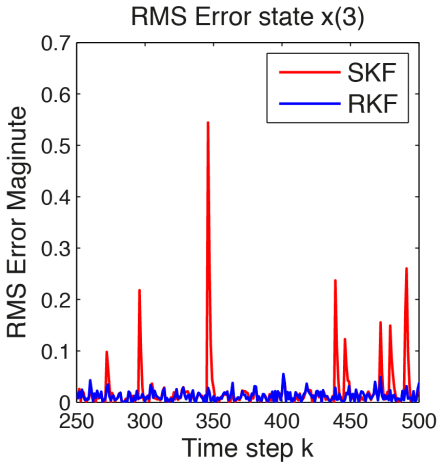


Figure 8.22: RMS error for $x(3)$

8.5 Low-level interface: Spacecraft Rendezvous

8.5.1 Introduction

This example uses the concepts described in the subsections *HOW TO: Implement an MPC Controller with a Time-Varying Model* 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 maneouvers 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.

8.5.2 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.

8.5.3 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.

8.5.4 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 penalised using the 1-norm, which better reflects the propellant consumption being directly proportional to delivered thrust and also attempts to minimise 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.

```
%% MPC with Preview
% FORCES PRO multistage form
% assume variable ordering zi = [ui; xi+1, eui] for i=1...N-1

% Parameters: First Eq. RHS
parameter(1) = newParam('minusA_times_x0',1,'eq.c');

stages = MultistageProblem(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 = 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(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

        % 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 < N )
                params(end+1) = newParam(['C_',num2str(i)],i,'eq.C');
        end
        params(end+1) = newParam(['D_',num2str(i)],i,'eq.D');
        if( i > 1 )
                params(end+1) = newParam(['pre',num2str(i+1),'_w'],i+1,'eq.c');
        end
```

(continues on next page)

(continued from previous page)

```
end
```

8.5.5 Spacecraft Rendezvous Manoeuvers with and without 1-Norm Cost

The simulation describes a rendezvous maneover 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 8.23 illustrates the behaviour of the controlled spacecraft with standard quadratic cost, while the plots in Figure 8.24 shows the behaviour 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.

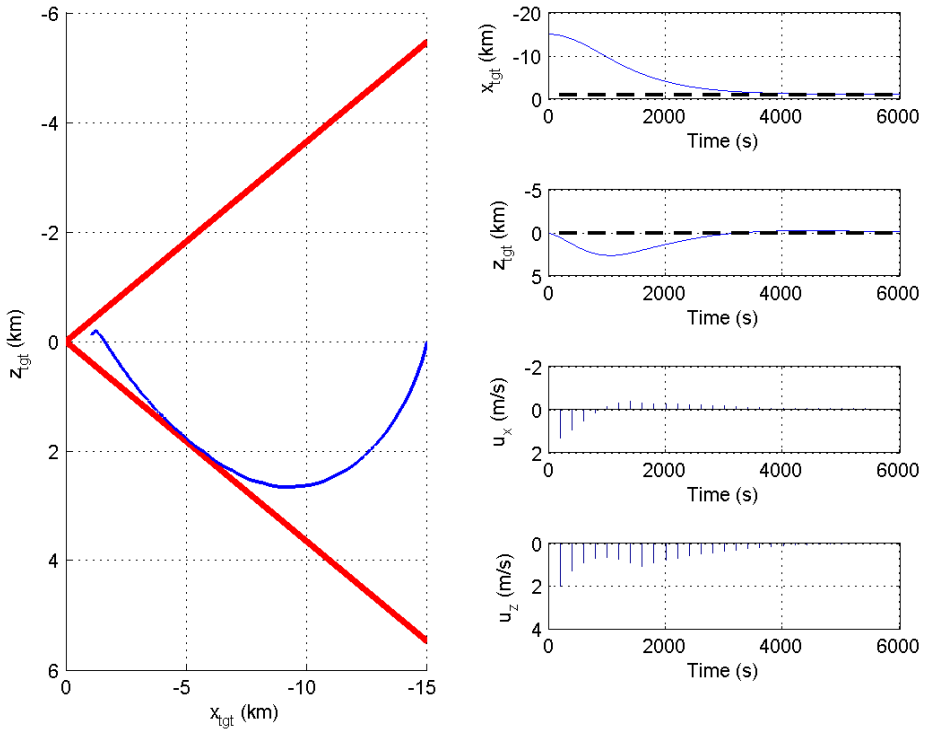


Figure 8.23: Behaviour with quadratic cost.

8.6 Low-level interface DC/DC converter

8.6.1 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 FORCES PRO is presented. Finally, the simulation setup is explained and the simulation results using PI and MPC are compared.

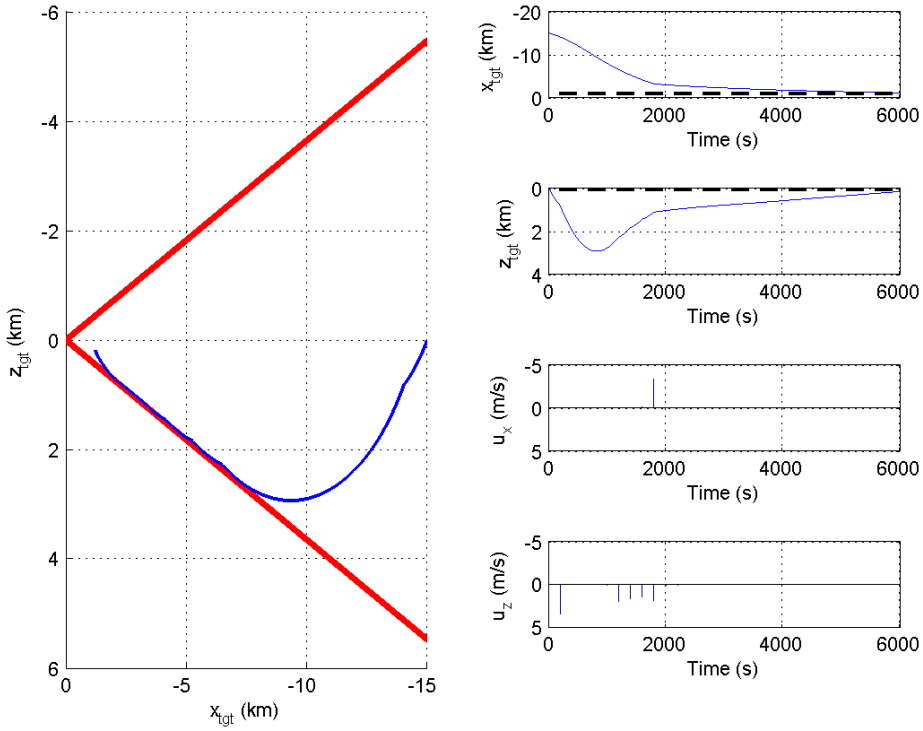


Figure 8.24: Behaviour with cost given by 1-norm.

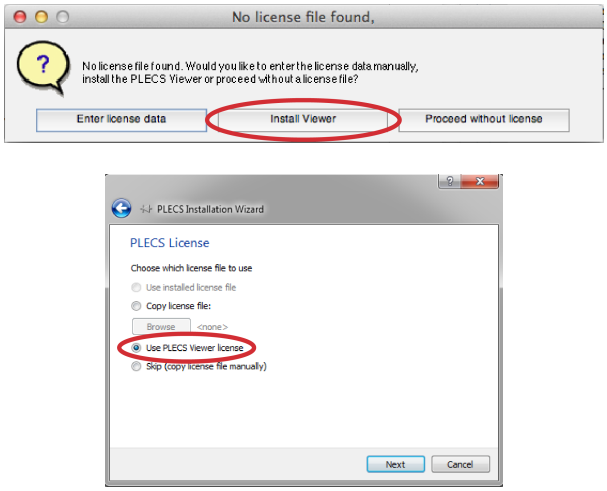
- *Introduction*: General introduction to the example.
- *Control Objective*: What can be gained by applying MPC with FORCES PRO.
- *MPC via FORCES PRO*: How to generate a solver with FORCES PRO for the power electronic converter.
- *Simulation*: Illustration on how to simulate the system with the generated controller.
- *Comparison*: Discussion of the results of the simulation.

8.6.2 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 licence 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.

8.6.3 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.

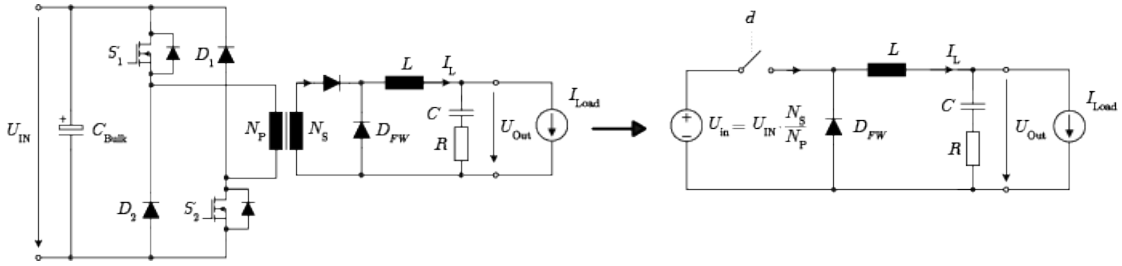


Figure 8.25: Based on the lecture material Power Electronic Systems II, Institute for Power Electronic Systems, ETH Zürich

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

model (small signals are marked with a hat) in state-space form reads as:

$$\frac{d}{dt}\hat{x} = \begin{pmatrix} -\frac{R}{L} & -\frac{1}{L} \\ \frac{1}{C} & 0 \end{pmatrix} \hat{x} + \begin{pmatrix} \frac{U_{in}}{L} \\ 0 \end{pmatrix} \hat{d} + \begin{pmatrix} \frac{D}{L} & -\frac{R}{L} \\ 0 & -\frac{1}{C} \end{pmatrix} \hat{w}$$

$$\hat{y} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ R & 1 \end{pmatrix} \hat{x} + \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 0 & -R \end{pmatrix} \hat{w}$$

$$\Updownarrow$$

$$\frac{d}{dt}\hat{x} = A \cdot \hat{x} + B1 \cdot u + B2 \cdot \hat{w}$$

$$\hat{y} = C \cdot \hat{x} + \begin{pmatrix} D2 \\ D4 \end{pmatrix} \cdot \hat{w}$$

8.6.4 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.

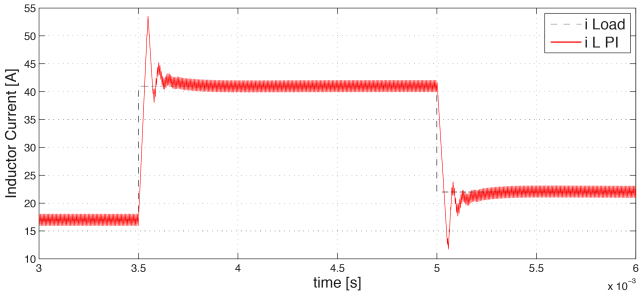


Figure 8.26: Inductor current vs. time

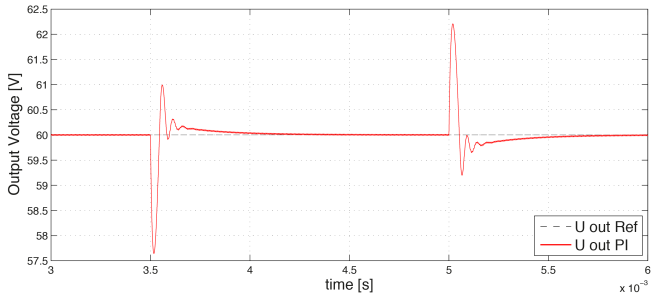


Figure 8.27: Output voltage vs. time

From Figure 8.26 and Figure 8.27 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 occures.

Important: Some of the potential benefits of model predictive cotrol are the following

- Below it is shown that the size of the converter can be reduced by using a MPC controller designed with FORCES PRO. 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 FORCES PRO will calculate the optimal input at every time step. The performance of the system is increased, i. e. less overshoot and faster settling time.

8.6.5 Model Predictive Control Design via FORCES PRO MATLAB® Interface

To design the FORCES PRO controller, the MPC setup has to be definded first. Below the requirements are shown. A prediction horizon of 25 is choosen. 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 multi-stage 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.

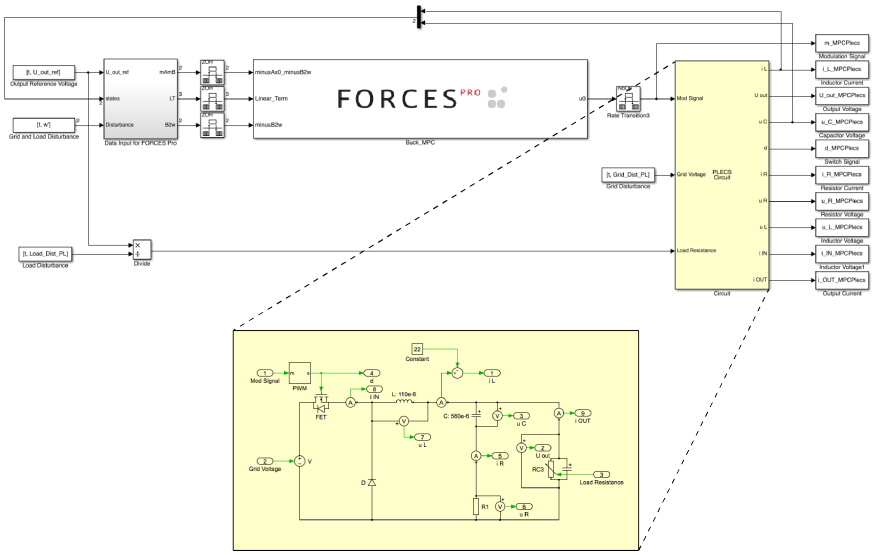
```
%% Multistage Problem
% get stages struct of length N
stages = MultistageProblem(N);

% RHS of first eq. constr. is a parameter: stages(1).eq.c = -A*x0 - B2*w
parameter(1) = newParam('minusAx0_minusB2w',1,'eq.c');

% Linear Term depends on x_ref and u_ref
parameter(2) = newParam('Linear_Term',1:N,'cost.f');

% RHS of equality constraints for remaining stages: stages(i).eq.c = - B2*w
parameter(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
   tages(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 u 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 u and state iL

   % equality constraints
   if( i < N )
          stages(i).eq.C = [zeros(nx,nu), Ad];
   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,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.

8.6.6 Simulation of the PLECS® Model with Model Predictive Control

After the code is generated, the FORCES PRO 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 FORCES PRO).

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 FORCES PRO controller is depicted.



In the grey box in the model depicted above, the three parameters which are the input to the FORCES PRO 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 $-Bd2 \cdot w$.

8.6.7 Comparison of Model Predictive Control and PI Control

In the Figure 8.28 and Figure 8.29 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 FORCES PRO. 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.

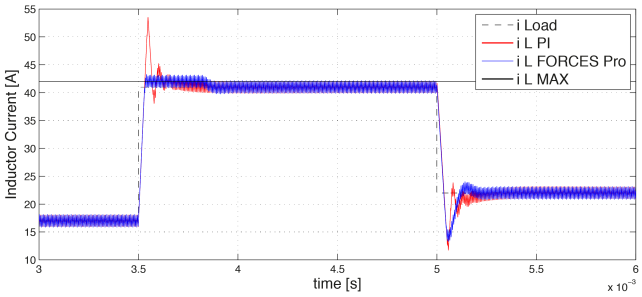


Figure 8.28: Inductor current vs. time

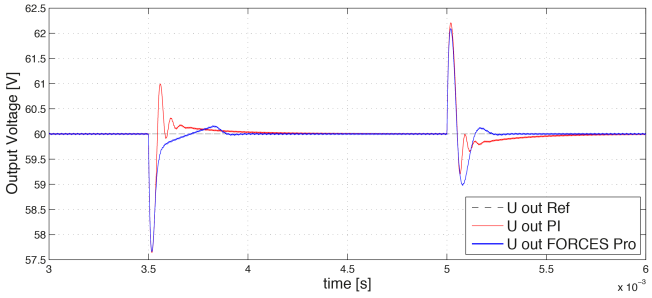


Figure 8.29: Output voltage vs. time

8.7 High-level interface: 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.

You can find the Matlab code of this example to try it out for yourself in the `examples` folder that comes with your client.

8.7.1 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
N = 10;
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];
```

8.7.2 Defining the MPC problem

Let's now dive in right into the problem formulation:

```
%% FORCES multistage form
% assume variable ordering zi = [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(1)*R*z(1) + [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  ];
```

8.7.3 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 FORCES solver

% get options
codeoptions = getOptions('FORCESNLPsolver');
codeoptions.printlevel = 2;

% generate code
FORCES_NLP(model, codeoptions);
```

8.7.4 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.

8.7.5 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);
        solvetime(k) = info.solvetime;
        iters(k) = info.it;
    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
```

8.7.6 Results

The results of the simulation are presented in Figure 8.8. 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.

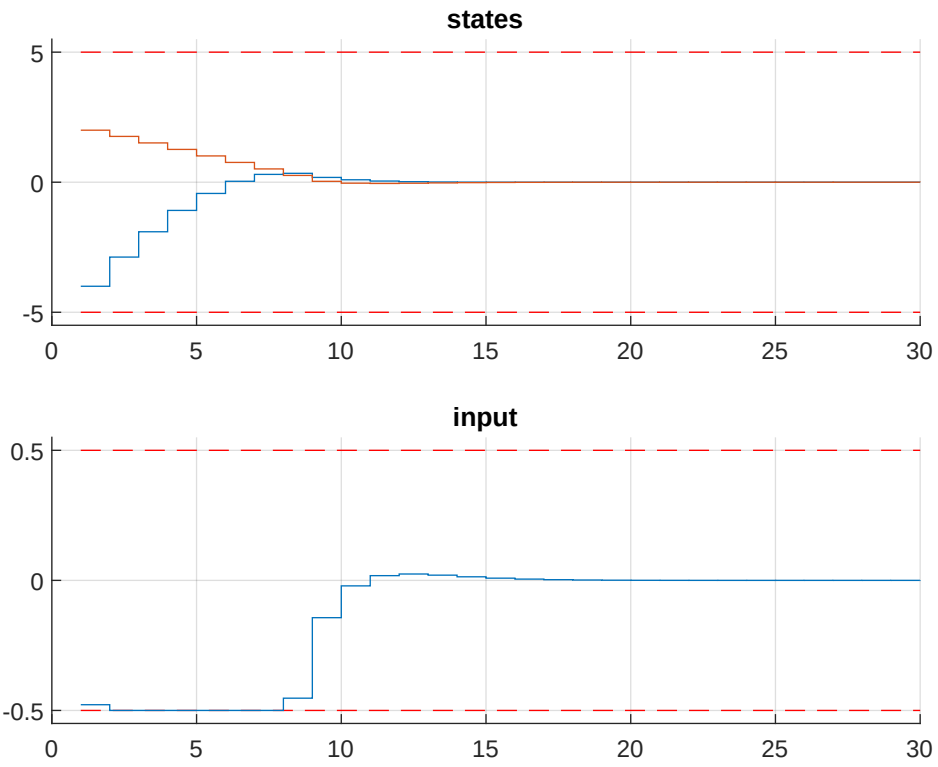


Figure 8.30: Simulation results of the states (top, in blue and red) and input (bottom, in blue) over time. The state and input constraints are plotted in red dashed lines.

8.8 High-level interface: Obstacle avoidance (MATLAB & Python)

In this example we illustrate the simplicity of the high-level user interface on a vehicle optimal trajectory generation problem. In particular, we use a simple vehicle model described by a set of ordinary differential equations (ODEs):

$$\begin{aligned}\dot{x} &= v\cos(\theta)\\ \dot{y} &= v\sin(\theta)\\ \dot{v} &= F/m\\ \dot{\theta} &= s/L\end{aligned}$$

The model consists of four differential states: x and y are the Cartesian coordinates of the car, v is the linear velocity and θ is the heading angle. Next, there are two control inputs to the model: the acceleration force F and the steering torque s. The two parameters are the car mass $m = 1\,\mathrm{kg}$ and the wheel base which we assume to be $L = 1\,\mathrm{m}$.

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:

$$z = [F, s, x, y, v, \theta]^\top$$

You can find the code of this example to try it out for yourself in the `examples` folder that comes with your client.

8.8.1 Defining the problem data

Objective

In this example the cost function is the same for all stages. We want to maximize progress in the y direction, with quadratic penalties on the inputs F and s, i.e.:

$$f(z) = -100z_4 + 0.1z_1^2 + 0.01z_2^2$$

The stage cost function is coded in MATLAB as the following function:

Matlab

Python

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

```
model.objective = lambda z: -100 * z[3] + 0.1 * z[0]**2 + 0.01 * z[1]**2
```

Matrix equality constraints

The matrix equality constraints in this example represent only the discretized dynamic equations of the vehicle using an explicit Runge-Kutta integrator of order 4. The vehicle dynamics defi ned above are represented by a function `continuous_dynamics` and the NLP constraint function $c(\cdot)$ as the function dynamics. Note that the function `RK4` is included in the FORCES PRO client software.

Matlab

Python

```
function xdot = continuous_dynamics(x, u)
    F = u(1);
    s = u(2);
    v = x(3);
    theta = x(4);
    m = 1;
    L = 1;
    xdot = [v * cos(theta);
            v * sin(theta);
            F / m;
            s / L];
end

function xnext = dynamics(z)
    x = z(3:6);
    u = z(1:2);
    % implements a RK4 integrator for the dynamics
    integrator_stepsize = 0.1;
    xnext = RK4(x, u, @continuous_dynamics, integrator_stepsize);
end
```

```
# Dynamics, i.e. equality constraints
def continuous_dynamics(x, u):
    m, I = 1, 1  # physical constants of the model
    return np.array([x[2] * casadi.cos(x[3]),  # v*cos(theta)
                     x[2] * casadi.sin(x[3]),  # v*sin(theta)
                     u[0] / m,  # F/m
                     u[1] / I])  # (v*s)/L

# We use an explicit RK4 integrator here to discretize continuous dynamics
integrator_stepsize = 0.1
model.eq = lambda z: forcespro.nlp.integrate(continuous_dynamics, z[2:6], z[0:2],
                                             integrator=forcespro.nlp.integrators.
↪RK4,
                                             stepsize=integrator_stepsize)
# Indices on LHS of dynamical constraint - for efficiency reasons, make
# sure the matrix E has structure [0 I] where I is the identity matrix.
model.E = np.concatenate([np.zeros((4,2)), np.eye(4)], axis=1)
```

Inequality constraints

The maneuver is subjected to a set of constraints, involving both the simple bounds:

$$\begin{aligned} -5\,\mathrm{N} \leq & F \leq 5\,\mathrm{N} \\ -1\,\mathrm{Nm} \leq & s \leq 1\,\mathrm{Nm} \\ -3\,\mathrm{m} \leq & x \leq 0\,\mathrm{m} \\ 0\,\mathrm{m} \leq & y \leq 3\,\mathrm{m} \\ 0\,\mathrm{m/s} \leq & v \leq 2\,\mathrm{m/s} \\ 0\,\mathrm{rad} \leq & \theta \leq \pi\,\mathrm{rad} \end{aligned}$$

as well the nonlinear nonconvex constraints:

$$\begin{aligned} 1\,\mathrm{m}^2 \leq & x^2 + y^2 \leq 9\,\mathrm{m}^2 \\ 0.9025\,\mathrm{m}^2 \leq & (x+2)^2 + (y-2.5)^2 \end{aligned}$$

In this case, the nonlinear constraint function $h(\cdot)$ can be coded in MATLAB/Python as follows:

Matlab

Python

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

```
# General (differentiable) nonlinear inequalities hl <= h(x) <= hu
model.ineq = lambda z:  np.array([z[2] ** 2 + z[3] ** 2,
                                  (z[2] + 2) ** 2 + (z[3] - 2.5) ** 2])

# Upper/lower bounds for inequalities
model.hu = np.array([9, +np.inf])
model.hl = np.array([1, 0.95 ** 2])
```

Initial and final conditions

The goal of the maneuver is to steer the vehicle from a set of initial conditions:

$$x_{\text{init}} = -2\,\text{m}, \quad y_{\text{init}} = 0\,\text{m}, \quad v_{\text{init}} = 0\,\text{m/s}, \quad \theta_{\text{init}} = 2.0944\,\text{rad}$$

to another point in the state-space subjected to the final conditions:

$$v_{\text{final}} = 0\,\text{m/s}, \quad \theta_{\text{final}} = 0\,\text{rad}$$

8.8.2 Defining the MPC problem

With the above de fined MALTAB functions for objective, matrix equality and inequality functions, we can completely define the NLP formulation in the next code snippet. For this example, we chose to use $N = 50$ stages in the NLP:

Matlab

Python

```
%% Problem dimensions
model.N = 50;             % horizon length
model.nvar = 6;           % number of variables
model.neq  = 4;           % number of equality constraints
model.nh = 2;             % number of inequality constraint functions

%% Objective function
model.objective = @objective;

%% Matrix equality constraints
model.eq = @dynamics;
model.E = [zeros(4, 2), eye( 4 )];

%% Inequality constraints
% upper/lower bounds lb <= z <= ub
model.lb = [-5, -1, -3, 0, 0, 0 ];
model.ub = [+5, +1, 0, 3, 2, +pi];
```

(continues on next page)

(continued from previous page)

```
% Nonlinear inequalities hl <= h(z) <= hu
model.ineq = @inequalities;
model.hu = [9, +inf]';
model.hl = [1, 0.95^2]';

%% Initial and final conditions
model.xinit = [-2, 0, 0, deg2rad(120)]';
model.xinitidx = 3:6;
model.xfinal = [0, deg2rad(0)]';
model.xfinalidx = 5:6;
```

```
# Problem dimensions
model = forcespro.nlp.SymbolicModel()
model.N = 50  # horizon length
model.nvar = 6  # number of variables
model.neq = 4  # number of equality constraints
model.nh = 2  # number of inequality constraint functions

# Inequality constraints
# Simple bounds
# upper/lower variable bounds lb <= x <= ub
#                     inputs  |  states
#                      F   s     x   y   v   theta
model.lb = np.array([ -5, -1,   -3,  0,  0,      0])
model.ub = np.array([ +5, +1,    0,  3,  2, +np.pi])

# Initial and final conditions
# Initial condition on vehicle states
xinit = np.array([-2, 0, 0, np.deg2rad(120)])  # x=-2, y=0, v=0 (standstill),␣
↪heading angle=120? # transposed
model.xinitidx = range(2,6)  # use this to specify on which variables initial␣
↪conditions are imposed

# Final condition on vehicle velocity and heading angle
xfinal = np.array([0, np.deg2rad(0)])  # v final=0 (standstill), heading angle␣
↪final=0? # transposed
model.xfinalidx = range(4, 6)  # use this to specify on which variables final␣
↪conditions are imposed
```

8.8.3 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` with the first state as a parameter:

Matlab

Python

```
%% Define 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;
```

(continues on next page)

(continued from previous page)

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

```
# Set solver options
codeoptions = forcespro.CodeOptions('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

# Creates code for symbolic model formulation given above, then contacts server to␣
↪generate new solver
solver = model.generate_solver(codeoptions)
```

8.8.4 Calling the generated solver

Once all parameters have been populated, the MEX interface of the solver can be used to invoke it:

Matlab

Python

```
%% Call solver
% Set initial guess to start solver from:
x0i=model.lb+(model.ub-model.lb)/2;
x0=repmat(x0i',model.N,1);
problem.x0=x0;

% Set initial and final conditions. This is usually changing from problem
% instance to problem instance:
problem.xinit = model.xinit;
problem.xfinal = model.xfinal;

% Time to solve the NLP!
[output,exitflag,info] = FORCESNLPsolver(problem);

% Make sure the solver has exited properly.
assert(exitflag == 1,'Some problem in FORCES solver');
fprintf('\nFORCES took %d iterations and %f seconds to solve the problem.\n',info.
↪it,info.solvetime);
```

```
# Set initial guess to start solver from:
x0i = (model.lb + model.ub) / 2.0
x0 = np.transpose(np.tile(x0i, (1, model.N)))

problem = {"x0": x0,
           "xinit": xinit,
           "xfinal": xfinal}

# 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))
```

8.8.5 Results

The goal is to find a trajectory that steers the vehicle from point A to another standstill point while avoiding obstacles and maximizing the progress on the y-direction along the way. The trajectory should also be feasible with respect to the vehicle dynamics and its safety and physical limitations. The vehicle's trajectory in 2D space is presented in Figure 8.31.

The vehicle's velocity and steering angle over time is presented in Figure 8.32, and the actuator commands in Figure 8.33. One can see that all constraints are respected.

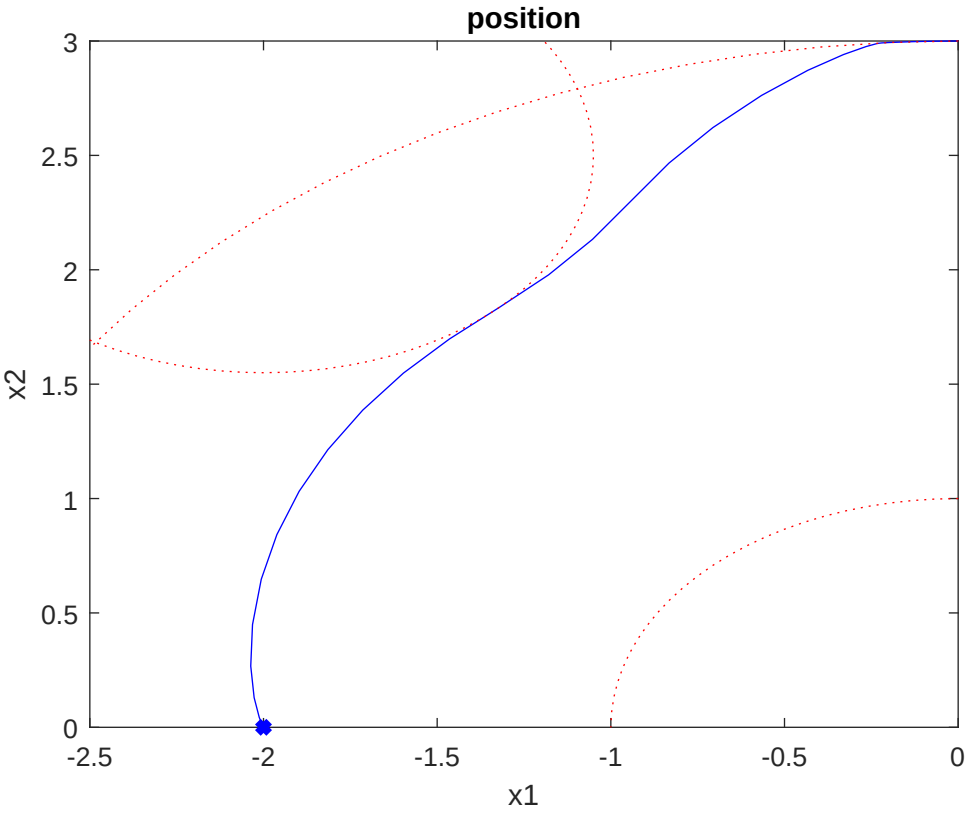


Figure 8.31: Vehicle's trajectory in 2D space.

8.8.6 Variation 1: Parameters

One possible variation is if we consider the mass m and wheel base L as parameters, so that we can tune them after the code generation. First we define the number of parameters:

Matlab

Python

```
for i=1:model.N-1
    model.npar(i) = 2; % number of parameters
end
model.npar(model.N) = 0; % no parameters in the last stage
```

```
model.npar = 2
model.nparN = 0
```

and then include them into our dynamics function handles:

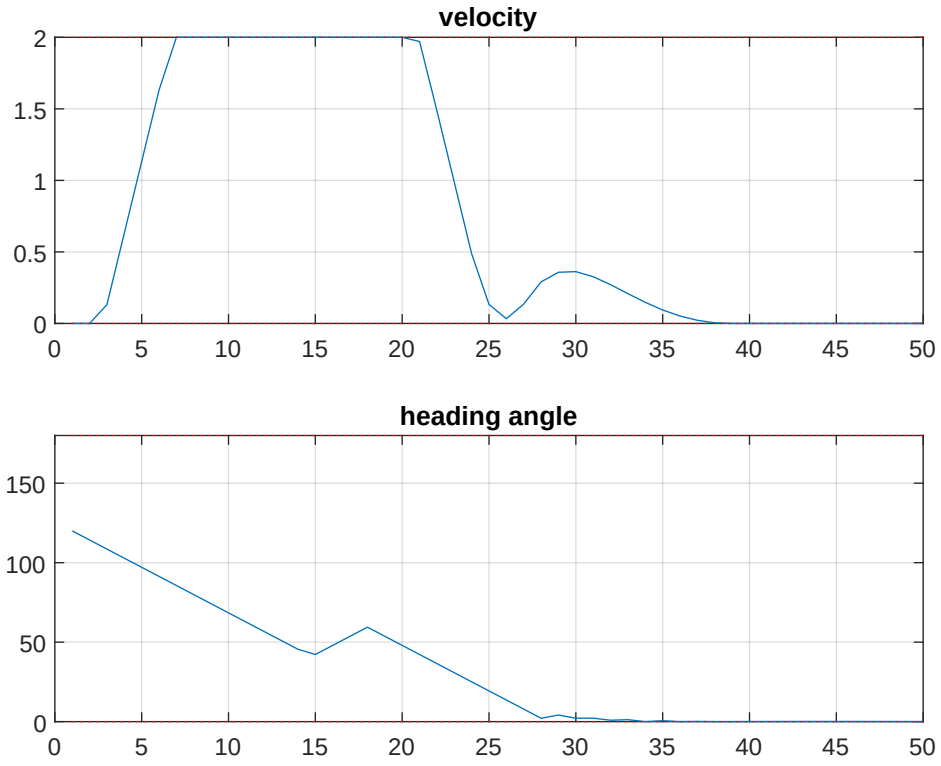


Figure 8.32: Vehicle's velocity and steering angle over time.

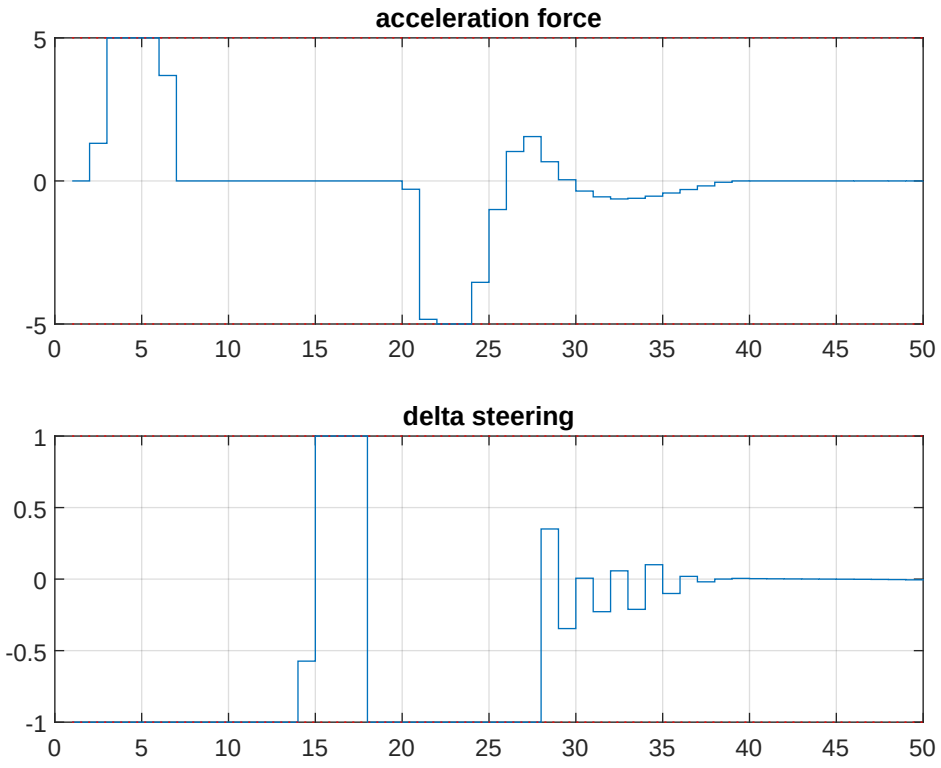


Figure 8.33: Vehicle's actuator commands over time.

Matlab

Python

```
function xdot = continuous_dynamics(x, u, p)
    F = u(1);
    s = u(2);
    v = x(3);
    theta = x(4);
    m = p(1);
    L = p(2);
    xdot = [v * cos(theta);
            v * sin(theta);
            F / m;
            s / L];
end

function xnext = dynamics(z, p)
    x = z(3:6);
    u = z(1:2);
    % implements a RK4 integrator for the dynamics
    integrator_stepsize = 0.1;
    xnext = RK4(x, u, @continuous_dynamics, integrator_stepsize, p);
end
```

```
# Dynamics, i.e. equality constraints
def continuous_dynamics(x, u, p):
    m = p[0]
    I = p[1]
    return np.array([x[2] * casadi.cos(x[3]),  # v*cos(theta)
                     x[2] * casadi.sin(x[3]),  # v*sin(theta)
                     u[0] / m,  # F/m
                     u[1] / I])  # (v*s)/L

# We use an explicit RK4 integrator here to discretize continuous dynamics
integrator_stepsize = 0.1
model.eq = lambda z, p: forcespro.nlp.integrate(continuous_dynamics, z[2:6],␣
↪z[0:2], p,
                                                integrator=forcespro.nlp.
↪integrators.RK4,
                                                stepsize=integrator_stepsize)
```

Note that we have to provide these parameters through the `problem` struct before calling the generated solver:

Matlab

Python

```
% Set parameters
problem.all_parameters = repmat([1 1]',model.N-1,1);
```

```
m = 1
I = 1
problem = {"x0": np.reshape(x0, -1),
           "xinit": xinit,
           "all_parameters": np.tile(np.array([m, I]), model.N-1)}
```

You can find the code of this example to try it out for yourself in the `examples` folder that comes with your client.

8.8.7 Variation 2: Different integrator

Another possible variation is if we want to change the integrator that is used to discretize the continuous-time dynamics. In the example above, we discretized our dynamics ourselves by using the supplied *RK4* function. It is also possible to give the continuous-time dynamics to the solver generator by using the *continuous_dynamics* field and varying the `codeoptions.nlp` fields:

Matlab

Python

```
% use continuous-time dynamics
codeoptions.continuous_dynamics = @continuous_dynamics;

% define integrator options
codeoptions.nlp.integrator.type = 'IRK4'; % can also be 'ForwardEuler', 'ERK2',
↪'ERK3', 'ERK4', 'BackwardEuler', or 'IRK2'
codeoptions.nlp.integrator.Ts = 0.1;
codeoptions.nlp.integrator.nodes = 10;
```

```
# continuous dynamics will be integrated according to codeoptions.integrator␣
↪settings
model.continuous_dynamics = continuous_dynamics

# Integration options
codeoptions.nlp.integrator.type = "IRK4"
codeoptions.nlp.integrator.Ts = 0.1
codeoptions.nlp.integrator.nodes = 10
```

For more information regarding the different integrators available, see *Integrators*.

You can find the code of this example to try it out for yourself in the `examples` folder that comes with your client.

8.8.8 Variation 3: Terminal cost

Another possible variation is if we want to have a terminal cost that is different than the stage costs of the horizon. To do that we provide each cost function handle in a cell array as follows:

Matlab

Python

```
%% Objective function
% In this example, we want to penalize the inputs F and s:
for i=1:model.N-1
    model.objective{i} = @(z) 0.1*z(1)^2 + 0.01*z(2)^2;
end

% and maximize the progress on the y direction, while ensuring a small
% velocity and heading angle at the end of the horizon.
% Terminal cost: -100*y 100*v^2 + 100*theta^2 to aim for max y, v=0 and theta=0
model.objective{model.N} = @(z) -100*z(4) + 10*(z(5)-0)^2 + 10*(z(6)-0)^2;
```

```
# Objective function
# In this example, we want to penalize the inputs F and s:
model.objective = lambda z: 0.1*z[0]**2 + 0.01*z[1]**2

# and maximize the progress in the y direction, while ensuring a small
```

(continues on next page)

(continued from previous page)

```
# velocity and heading angle at the end of the horizon:
model.objectiveN = lambda z: -100*z[3] + 10*(z[4]-0)**2 + 10*(z[5]-0)**2
```

You can find the code of this example to try it out for yourself in the `examples` folder that comes with your client.

8.8.9 Variation 4: External functions

One final variation is if we 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';
```

We also need to include the two `extern` functions `car_dyanmics` and `car_dyanmics_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
# the 'relative_to' argument specifies that the paths are 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', 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.

8.9 High-level interface: Indoor localization (MATLAB & Python)

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 FORCES PRO as well with only minor changes to the code below.

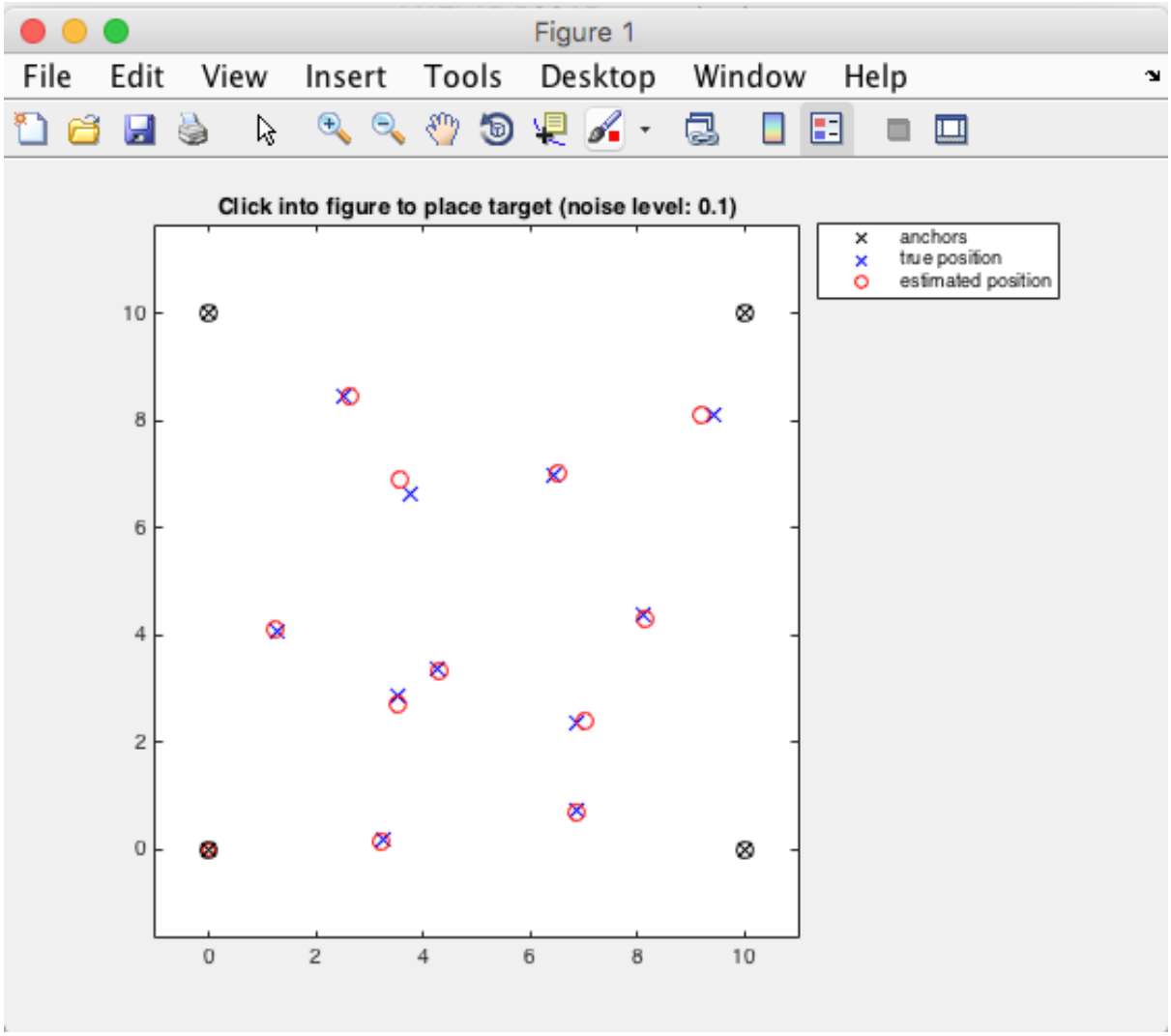


Figure 8.34: Indoor localization example GUI.

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 8.34.

8.9.1 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 $c = 299\,792\,458\,\mathrm{m/s}$

8.9.2 Estimation error

Instead of the real distance, we work with squared distances to define the estimation error:

$$e_i = (x - x_i^a)^2 + (y - y_i^a)^2 - d_i^2$$

8.9.3 Minimize the error

The objective is a least-squares error function:

$$\min_{x,y} \sum_{i=1}^{N} e_i^2$$

8.9.4 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 FORCES NLP
% na: number of anchors
  global na
  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 = @objective;
  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

  %% generate code
```

(continues on next page)

(continued from previous page)

```
  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 )
  global 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 FORCES PRO solver that esimates 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)

    # FORCES PRO 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:

     - 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
```

(continues on next page)

(continued from previous page)

```
def distance(xa, xtrue, ya, ytrue):
    return np.sqrt((xa - xtrue)**2 + (ya - ytrue)**2)
```

8.10 Real-time SQP Solver: Robotic Arm Manipulator (MATLAB & Python)

In this example we illustrate the use of the real-time Sequential Quadratic Programming (SQP) solver. 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.

8.10.1 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.objective = lambda z, p: (  1000 * (z[2] - p[0]*1.2)**2
                                +  0.1 * z[3]**2
                                + 1000 * (z[4] + p[0]*1.2)**2
                                + 0.10 * z[5]**2
                                + 0.01 * z[6]**2
                                + 0.01 * z[7]**2
                                + 0.01 * z[0]**2
                                + 0.01 * z[1]**2)
```

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)
```

8.10.2 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 timing)
codeoptions.optlevel = 3;
% Explicit Runge-Kutta 4 integrator
codeoptions.nlp.integrator.Ts = integrator_stepsize;
codeoptions.nlp.integrator.nodes = 5;
codeoptions.nlp.integrator.type = 'ERK4';
% Options for SQP solver
codeoptions.solvemethod = 'SQP_NLP';                        % SQP algorithm
codeoptions.nlp.hessian_approximation = 'gauss-newton';     % Gauss-Newton hessian␣
↪approximation of nonlinear least-squares objective
codeoptions.sqp_nlp.use_line_search = 0;                    % Disable line-search␣
↪for efficiency (only doable with Gauss-Newton approximation)

%% Generate real-time SQP solver
FORCES_NLP(model, codeoptions);
```

```
# 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

# Generate real-time SQP 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 convergence 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.

8.10.3 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.

8.10.4 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 8.35 and Figure Figure 8.36 respectively, while the input torque rates are plotted in Figure Figure 8.37. The closed-loop objective, which is a measure of the control performance is shown in Figure Figure 8.38.

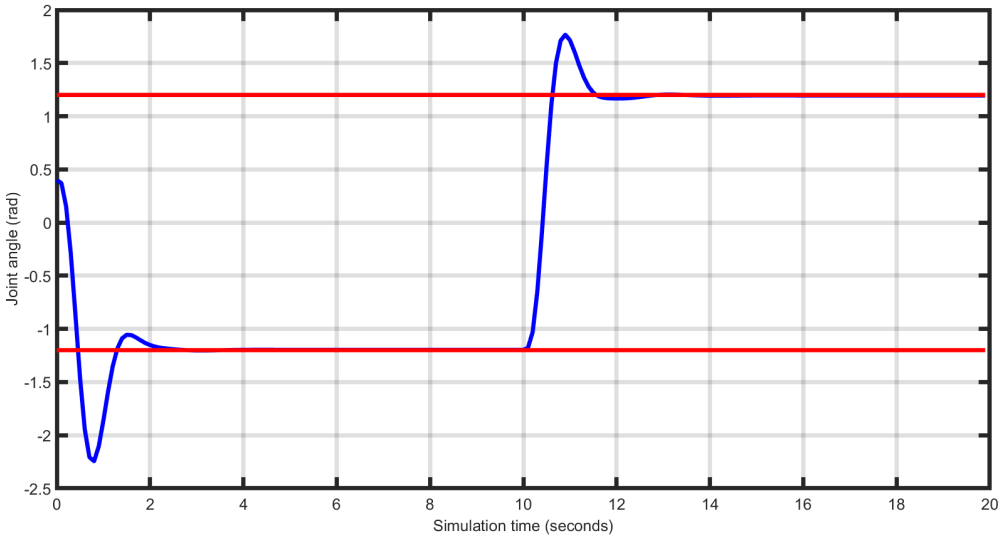


Figure 8.35: Manipulator's joint angle.

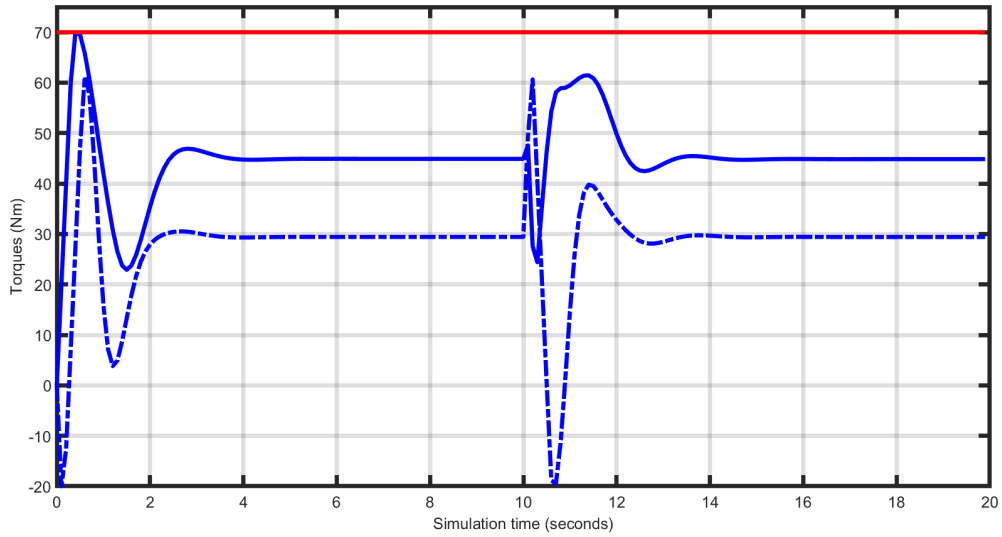


Figure 8.36: Manipulator's torques at joints.

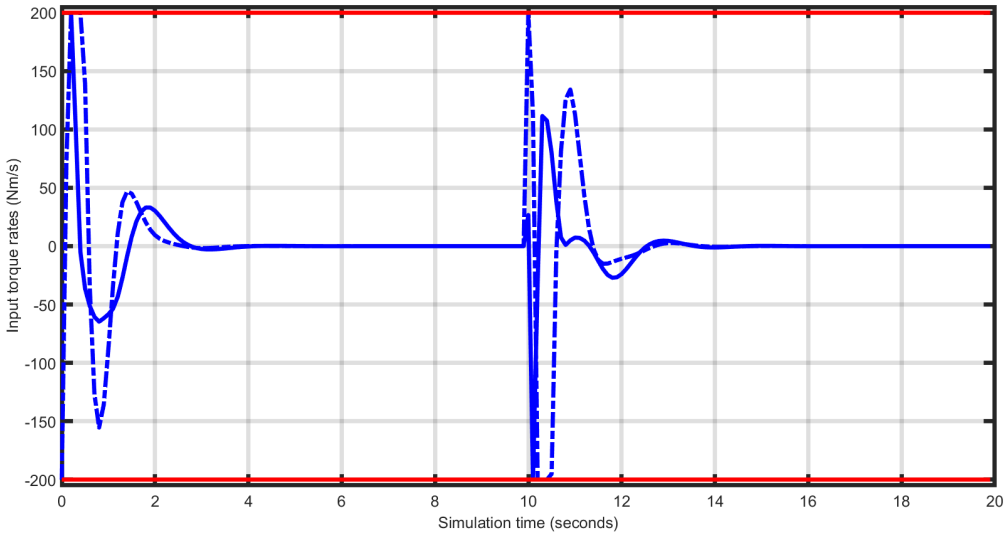


Figure 8.37: Manipulator's torque rates.

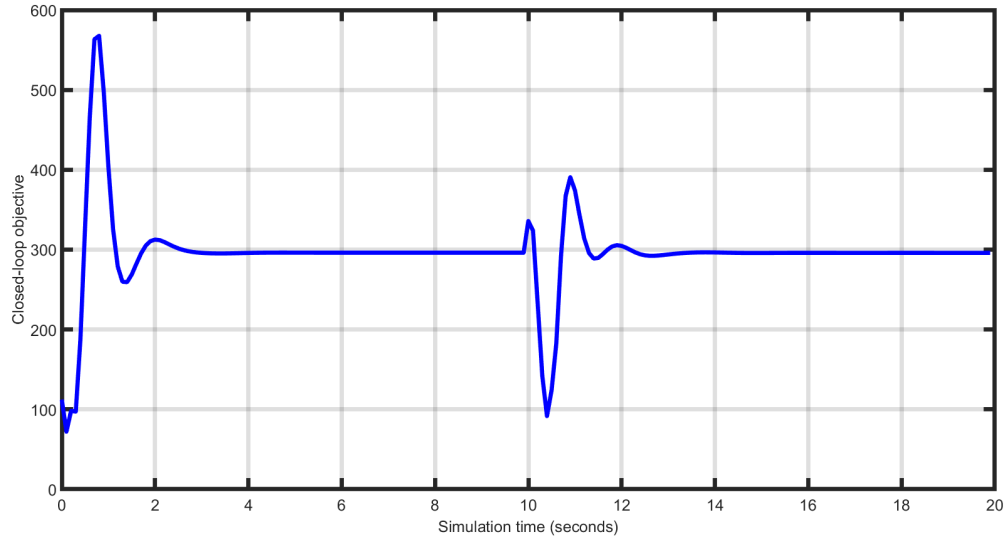


Figure 8.38: Manipulator's closed loop objective.

8.11 Controlling a DC motor using a FORCES PRO SQP solver

In this example our aim is to control a DC-motor using a FORCES PRO 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 viscuous 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.

8.11.1 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 (8.11) 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:

$$1\text{A} \leq u \leq 1.6\text{A}$$
$$-5\text{A} \leq x_1 \leq 5\text{A}$$
$$-10\tfrac{\text{rad}}{\text{sec}} \leq x_2 \leq 10\tfrac{\text{rad}}{\text{sec}}$$

This can be read into the FORCES PRO model as follows

```
model.lb = [1, -5, -10];
model.ub = [1.6, 5, 10];
```

Generating the FORCES PRO 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 code-snippet shows how this can be done:

```
%% set codeoptions
codeoptions = getOptions('FORCESPROSolver');
codeoptions.solvemethod = 'SQP_NLP'; % generate SQP solver
codeoptions.nlp.integrator.type = 'ERK4';
codeoptions.nlp.integrator.Ts = 0.01;
codeoptions.nlp.integrator.nodes = 1;
codeoptions.nlp.hessian_approximation = 'gauss-newton';
codeoptions.server = 'https://forces.embotech.com';

%% generate FORCES PRO 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 8.43 the controller completes this task. The control (u) is plotted in Figure 8.39, the first state (x_1) is plotted in Figure 8.40 and second state (x_2) in Figure 8.41. As a measure of control quality, the closed loop objective value is plotted in Figure 8.42.

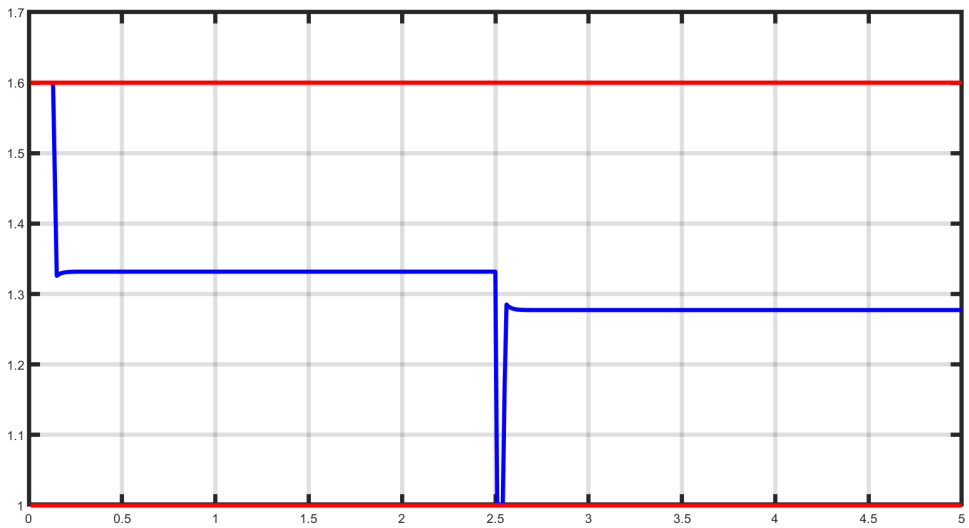


Figure 8.39: The control (u, in blue) as a function of simulation time (s). The control obeys its constraints (red) throughout the simulation.

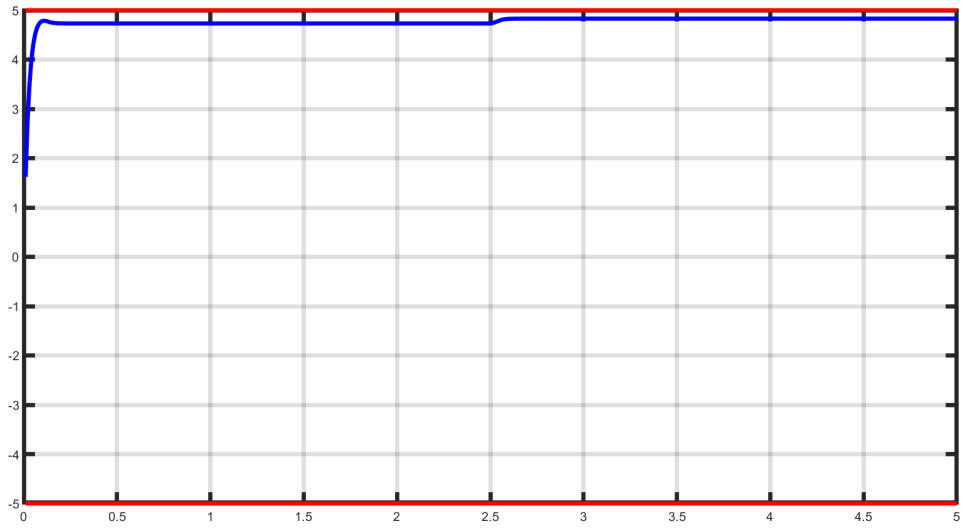


Figure 8.40: The first state (x_1, in blue) as a function of simulation time. It obeys its constraints (red) throughout the simulation.

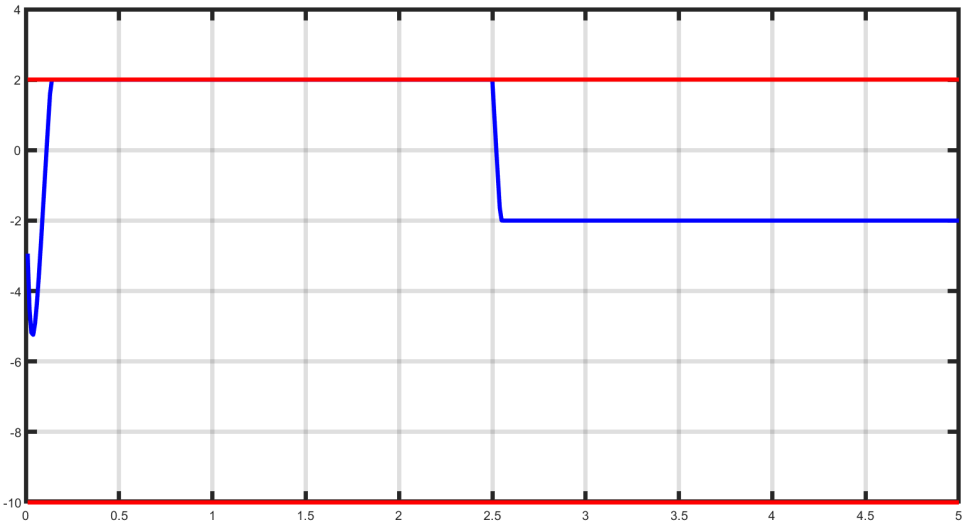


Figure 8.41: The second state (x_2, in blue) as a function of simulation time. It obeys its constraints (red) throughout the simulation.

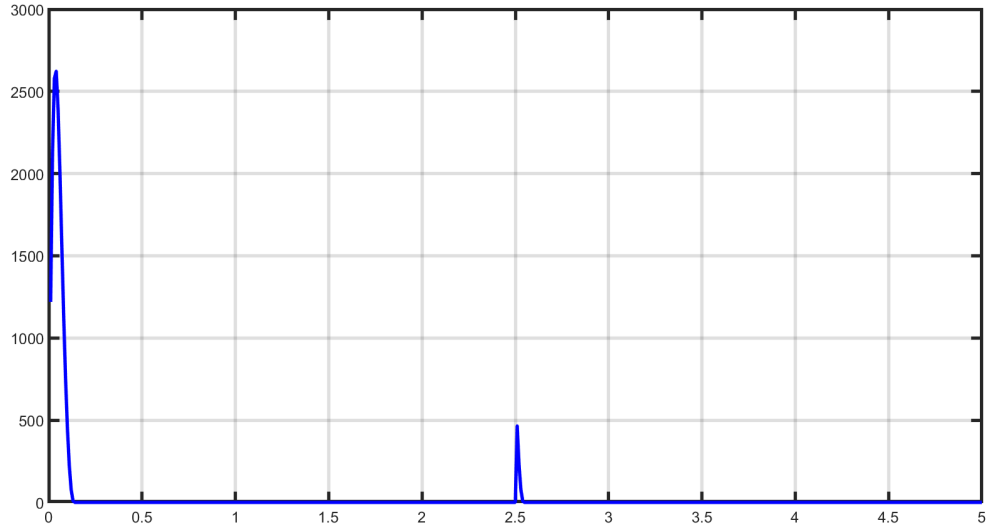


Figure 8.42: Closed-loop objective value as a function of time

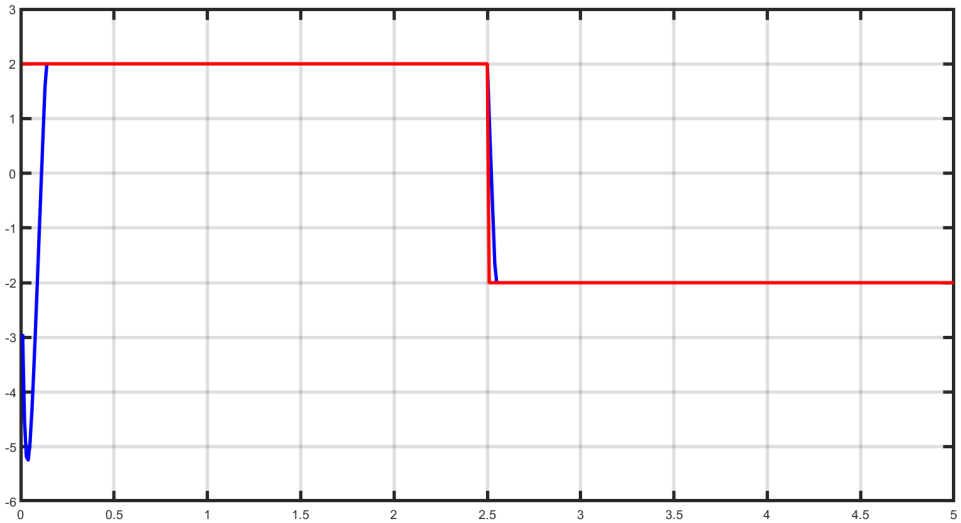


Figure 8.43: Angular speed (blue) and tracked reference (red) value as a function of time.

8.12 Mixed-integer nonlinear solver: F8 Crusader aircraft

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.

8.12.1 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) 0.1 * z(1)^2;
```

Equality constraints

In this example, the only equality constraints are related to the dynamics. They are provided to FORCES PRO in continuous form. The discretization is then computed internally by the FORCES PRO 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(3, 1), eye(3)];
```

Inequality constraints

The maneuver is subjected to a set of constraints, involving only the simple bounds:

$$\begin{aligned} 0\,\mathrm{rad} \leq & u \leq 1\,\mathrm{rad} \\ -10\,\mathrm{rad} \leq & x_0 \leq 10\,\mathrm{rad} \\ -10\,\mathrm{rad} \leq & x_1 \leq 10\,\mathrm{rad} \\ -10\,\mathrm{rad/sec} \leq & x_2 \leq 10\,\mathrm{rad/sec} \end{aligned}$$

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.

8.12.2 Defining the MPC problem

With the above de fined MALTAB 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.

```
%% Problem dimension
nx = 3;
nu = 1;
nz = nx + nu;
model.N = 100;
```

(continues on next page)

(continued from previous page)

```
model.nvar = nz;
model.neq = nx;

%% Indices of initial state in stage variable
model.xinitidx = nu+1:model.nvar;

%% Lower and upper bound need to be set as parametric for generating an MINLP␣
↪solver
model.lb = [];
model.ub = [];
model.lbidx{1} = 1 : nu;
model.ubidx{1} = 1 : nu;
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(nx, nu), eye(nx)];

%% Objective
mode.objective =  @(z) 0.1 * z(nu)^2;
model.objectiveN = @(z) 150 * z(nu+1)^2...
                      + 5 * z(nu+2)^2...
                      + 5 * z(nu+3)^2;

%% Indices of integer variables within every stage
for s = 1:model.N
  model.intidx{s} = [1];
end
```

8.12.3 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');
codeoptions.printlevel = 1;
codeoptions.misra2012_check = 1;
codeoptions.maxit = 2000;
codeoptions.timing = 0;
codeoptions.nlp.integrator.type = 'IRK2';
codeoptions.nlp.integrator.Ts = 0.05;
codeoptions.nlp.integrator.nodes = 20;

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

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`.

8.12.4 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
problem.(sprintf('lb%02d', 1)) = 0;
problem.(sprintf('ub%02d', 1)) = 1;
for s = 2:99
  problem.(sprintf('lb%02d', s)) = [0, -1e1 * ones(1, 3)]';
  problem.(sprintf('ub%02d', s)) = [1, 1e1 * ones(1, 3)]';
end
problem.(sprintf('lb%02d', 100)) = [0, -1e1 * ones(1, 3)]';
problem.(sprintf('ub%02d', Nstages)) = [1, 1e1 * ones(1, 3)]';

problem.x0 = repmat([0; zeros(3, 1)], 100, 1);
problem.xinit = zeros(3, 1);
problem.xinit(1) = 0.4655;

%% Call MINLP solver
[sol, exitflag, info] = F8aircraft(problem);
```

8.12.5 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
```

(continues on next page)

(continued from previous page)

```
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
```

8.12.6 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)
```

8.12.7 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 FORCES PRO MINLP solver running on one thread is shown in Figure Figure 8.44 and the input sequence is in Figure Figure 8.45. One can notice the bang-bang behaviour of the solution. When running on three threads the FORCES PRO MINLP solver provides a solution with lower final primal objective. Results are shown on Figures Figure 8.46 and Figure 8.47.

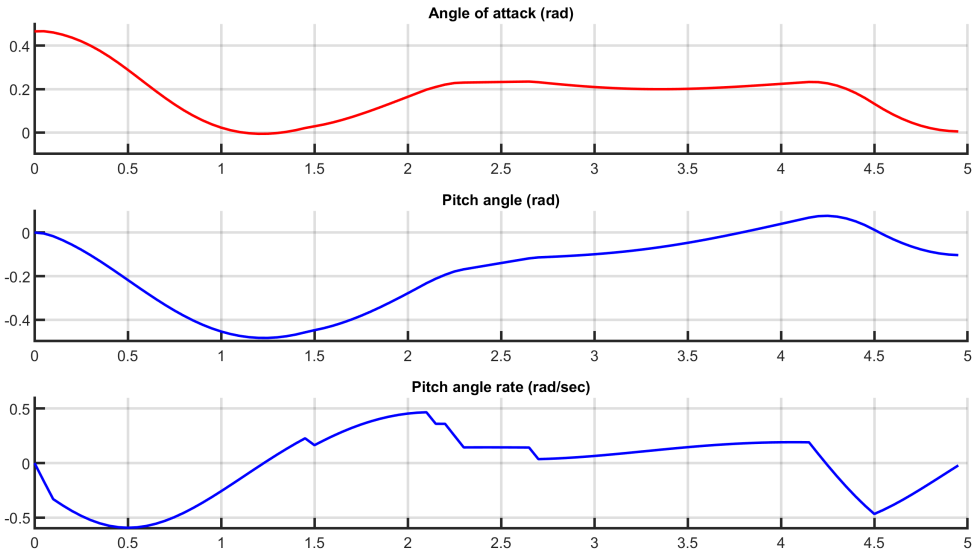


Figure 8.44: Aircraft's angle of attack over time computed with one thread.

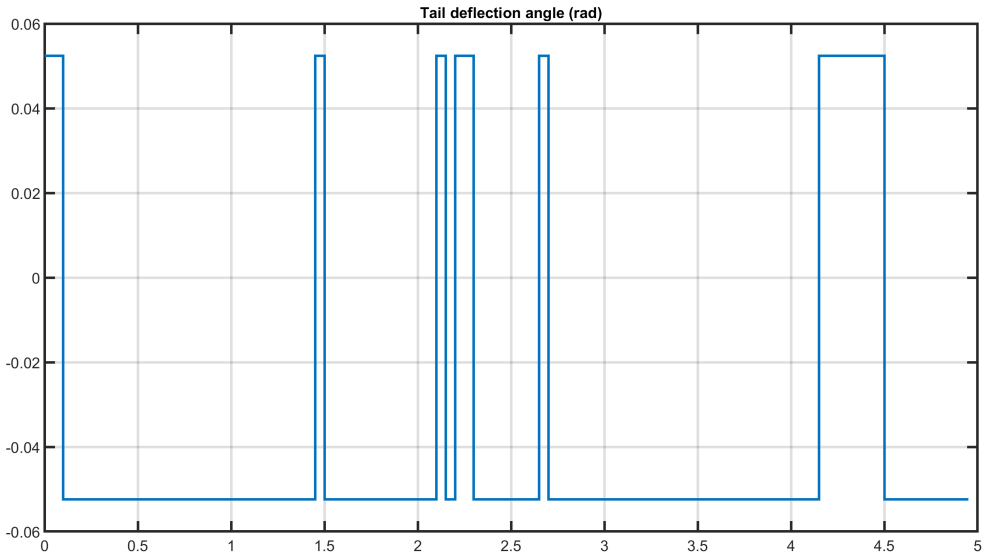


Figure 8.45: Aircraft's tail deflection angle over time with one thread.

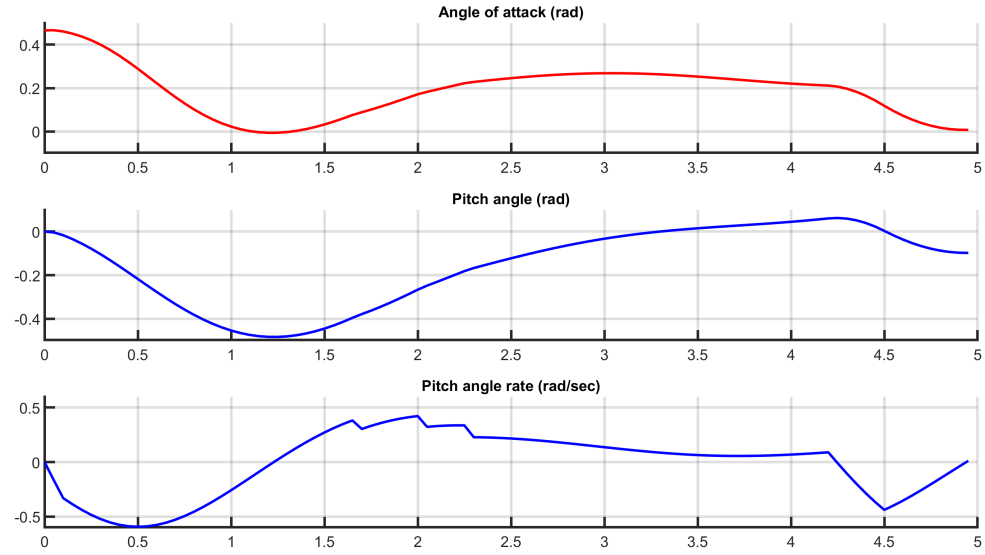


Figure 8.46: Aircraft's angle of attack over time computed with three threads.

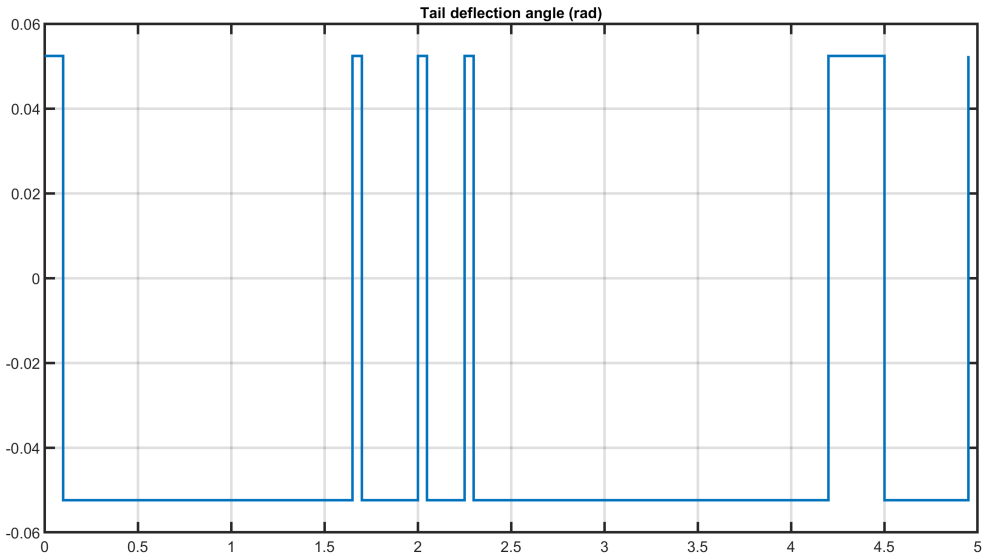


Figure 8.47: Aircraft's tail deflection angle over time with three threads.

Chapter 9

Parametric problems

Parameters (or real-time data) are a key concept in FORCES PRO. 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 non-linear 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.

9.1 Defining parameters

FORCES PRO 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 in MATLAB, use the function

```
parameter = newParam(name, maps2stage, maps2data);
```

and in Python, use

```
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 9.1: 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 FORCES PRO `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 or matrix before getting passed to the solver at runtime. FORCES PRO is notified about this by having

```
maps2stage = [];
```

For instance, in order to provide a parametric linear cost across all stages, one should use the following code at codegen.

```
parameter = newParam('linear_stage_cost', [], 'cost.f');
```

At runtime, the user is expected to provide the linear stage cost as follows.

```
problem.linear_stage_cost = repmat(rand(problem.nvar, 1), problem.horzLength, 1);
```

where `problem.horzLength` is the horizon length and `problem.nvar` is the number of stage variables.

Note: The stacked parameters feature is only available in MATLAB from Forces '1.8.0'.

9.2 Example

To define the linear term of the cost of stages 1 to 5 as a parameter, use the following command in MATLAB

```
parameter1 = newParam('linear_cost', 1:5, 'cost.f');
```

and in Python, use

```
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 differents in this case.

We can also have a second parameter. For instance, the right handside of the first equality constraints, which is a very common caes in MPC. In MATLAB

```
parameter2 = newParam('RHS_first_equality_constraint', 1, 'eq.c');
```

In Python

```
stages.newParam('RHS_first_equality_constraint', [1], 'eq.c')
```

9.3 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. In MATLAB

```
parameter = newParam(name, maps2stage, maps2data, idxWithinStage);
```

In Python

```
stages.newParam(name, maps2stage, maps2data, idxWithinStage)
```

where *idxWithinStage* denotes the index of the quadratic constraints to which this parameters applies.

9.4 Diagonal Hessians

In case your parametric Hessian is diagonal, you should use the fourth argument of `newParam` as shown below. In MATLAB

```
parameter1 = newParam('Hessians', 1:5, 'cost.H', 'diag');
```

In Python

```
stages.newParam('Hessians', range(1,6), 'cost.H', 'diag')
```

The FORCES PRO 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.

9.5 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 FORCES PRO know about the sparsity pattern. In MATLAB

```
parameter2 = newParam('Ai', 1:5, 'ineq.p.A', 'sparse', [zeros(5, 6) rand(5, 2)]);
```

In Python

```
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 FORCES PRO 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: In MATLAB

```
problem.Ci = nonzeros(sparse(B));
```

In Python

```
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.

9.6 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 FORCES PRO know about the location of the elements that will be supplied at runtime. In MATLAB

```
parameter2 = newParam('Ci', 1:5,'eq.C','sparse',Cstruc,Cvar)
```

In Python

```
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.

9.7 Python: Column vs Row Major Storage Format

Unlike Matlab, numpy stores arrays by default in row-major format internally. Since FORCES 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 10

Code Deployment

10.1 Main Targets

Main 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

You can check *here* to find the correct naming option for each platform.

10.1.1 High-level interface

The steps to deploy and simulate a FORCES PRO controller on most targets are detailed below.

1. In the High-level interface example BasicExample.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
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. Additionally to your solver you will receive the following files generated by CasADi:

- `FORCESNLPsolver_casadi2forces.c`
- `FORCESNLPsolver_model_1.c`
- `FORCESNLPsolver_model_11.c`

3. For most target platforms you will receive the following compiled files:

- For Linux/MacOS:
  - an object file `FORCESNLPsolver.o` inside the folder `obj_target`

  - a static library file `libFORCESNLPsolver.a` inside the folder `lib_target`
  - a shared library file `libFORCESNLPsolver.so` inside the folder `lib_target`
- For Windows:
  - an object file `FORCESNLPsolver.obj` inside the folder `obj_target`
  - a static library file `FORCESNLPsolver_static.lib` inside the folder `lib_target`
  - a dynamic library file `FORCESNLPsolver.dll` with its definition file for compilation `FORCESNLPsolver.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 `FORCESNLPsolver` folder
- The source files from step 2
- The interface from step 4

6. Compile the solver. The compilation command would be (supposing you are in the directory which contains the FORCESNLPsolver folder):

```
<Compiler_exec> HighLevel_BasicExample.c <compiled_solver> FORCESNLPsolver_
↪casadi2forces.c FORCESNLPsolver_model_1.c FORCESNLPsolver_model_11.c <additional_
↪libs>
```

Where:

- `<Compiler_exec>` would be the compiler used in the target
- `<compiled_solver>` would be one of the compiled files of step 3
- `<additional_libs>` 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 sometimes some additional intel libraries (those are included in the FORCES PRO client under the folder `libs_Intel` – if missing they are downloaded after code generation)

10.1.2 Y2F interface

The steps to deploy and simulate a FORCES PRO 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 Linux/MacOS:
  - an object file `internal_simpleMPC_solver_1.o` inside the folder `obj_target`
  - a static library file `libinternal_simpleMPC_solver_1.a` inside the folder `lib_target`
  - a shared library file `libinternal_simpleMPC_solver_1.so` inside the folder `lib_target`
- For Windows:
  - an object file `internal_simpleMPC_solver_1.obj` inside the folder `obj_target`
  - a static library file `internal_simpleMPC_solver_1_static.lib` inside the folder `lib_target`
  - a dynamic library file `internal_simpleMPC_solver_1.dll` with its definition file for compilation `internal_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_exec> Y2F_mpc_basic_example.c simpleMPC_solver/interface/simpleMPC_
↪solver.c <compiled_solver> <additional_libs>
```

Where:

- `<Compiler_exec>` would be the compiler used in the target
- `<compiled_solver>` would be one of the compiled files of step 3

- `<additional_libs>` 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 sometimes some additional intel libraries (those are included in the FORCES PRO client under `libs_Intel` – if missing they are downloaded after code generation)

10.2 dSPACE MicroAutoBox II

10.2.1 High-level interface

The steps to deploy and simulate a FORCES PRO controller on a dSPACE MicroAutoBox II are detailed below.

1. (Figure 10.1) Set the code generation options:

```
codeoptions.platform = 'dSPACE-MABII'; % 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 10.2) Create a new Simulink model using the RTI1401 template provided by dSPACE.
3. (Figure 10.3) Populate the Simulink model with the system you want to control.
4. (Figure 10.4) Make sure the `FORCESNLPsolver_simulinkBlock.mexw64` file (created during code generation) is on the Matlab path.
5. (Figure 10.5) Open the `FORCESNLPsolver_lib.mdl` Simulink model file, contained in the `interface` folder of the `FORCESNLPsolver` folder created during code generation.
6. (Figure 10.6) Copy-paste the FORCES PRO Simulink block into your simulation model and connect its inputs and outputs appropriately.
7. (Figure 10.7) Access the Simulink model's options.
8. (Figure 10.8) In the "Solver" tab, set the options:
   - Simulation start/stop time: Depending on the simulation wanted.
   - Solver type: Discrete or fixed-step (variable-step solvers are not supported by the MicroAutoBox).
   - Fixed-step size: Needs to be higher than the execution time of the solver.
9. (Figure 10.9) In the "Code Generation" tab, set the options:
   - System target file: `rti1401.tlc`
   - Language: C
   - Generate makefile: On
   - Template makefile: `rti1401.tmf`
   - Make command: `make_rti`
10. (Figure 10.10) In the "Code Generation/Custom Code" tab, include the directories:
    - `BasicExample`
    - `BasicExample\FORCESNLPsolver\interface`
    - `BasicExample\FORCESNLPsolver\lib_target`
11. (Figure 10.11) In the "Code Generation/Custom Code" tab, add the source files:
    - `FORCESNLPsolver_simulinkBlock.c`
    - `FORCESNLPsolver_casadi2forces.c`
    - `FORCESNLPsolver_model_1.c`
    - `FORCESNLPsolver_model_11.c`

12. (Figure 10.12) In the “Code Generation/Custom Code” tab, add the library files:
    - `FORCESNLPsolver.lib`
13. (Figure 10.13) Access the FORCES PRO block’s parameters.
14. (Figure 10.14) Remove the “FORCESNLPsolver” prefix from the S-function module.
15. (Figure 10.15) Compile the code of the Simulink model. This will also automatically load the model to the connected MicroAutoBox.

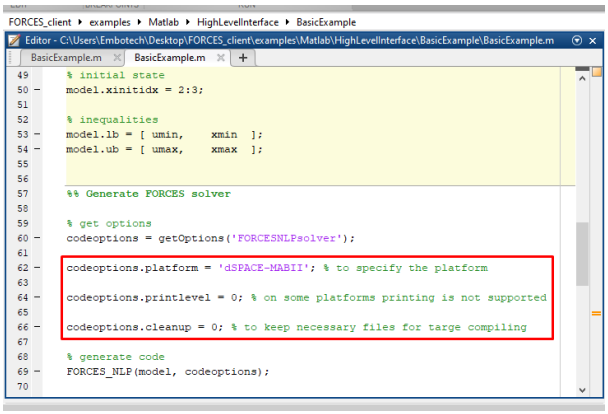


Figure 10.1: Set the appropriate code generation options.

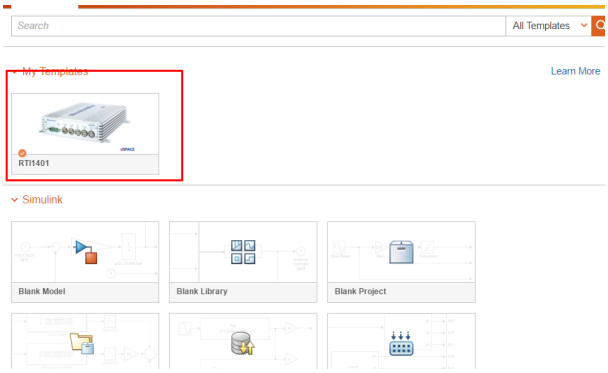


Figure 10.2: Create a Simulink model.

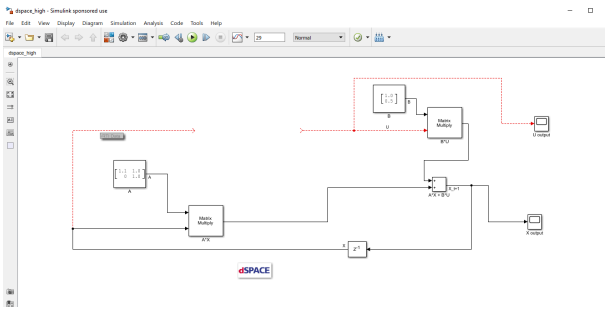


Figure 10.3: Populate the Simulink model.

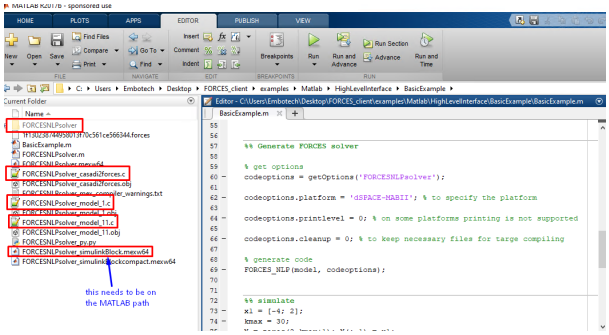


Figure 10.4: Add the folder containing the `.mexw64` solver file to the Matlab path.

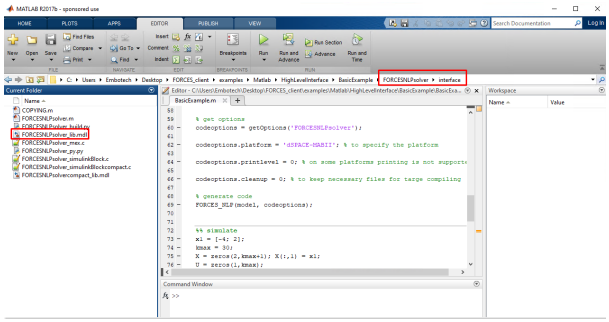


Figure 10.5: Open the generated Simulink solver model.

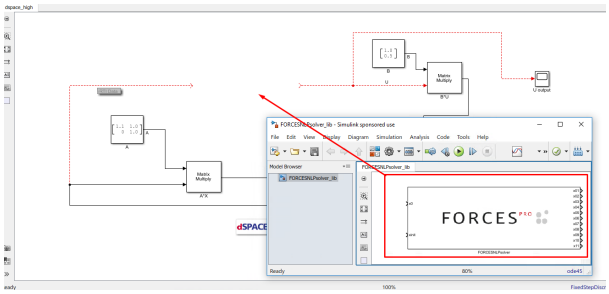


Figure 10.6: Copy-paste and connect the FORCES PRO block.

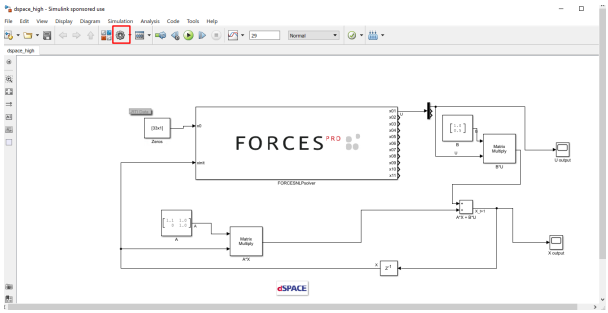


Figure 10.7: Open the Simulink model options.

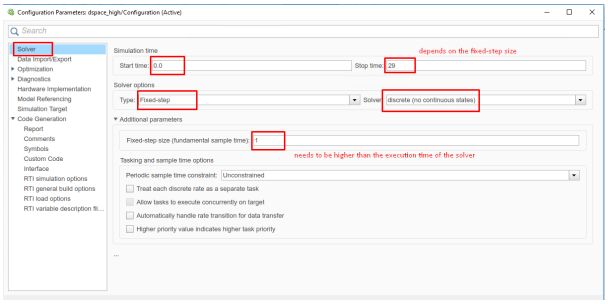


Figure 10.8: Set the Simulink solver options.

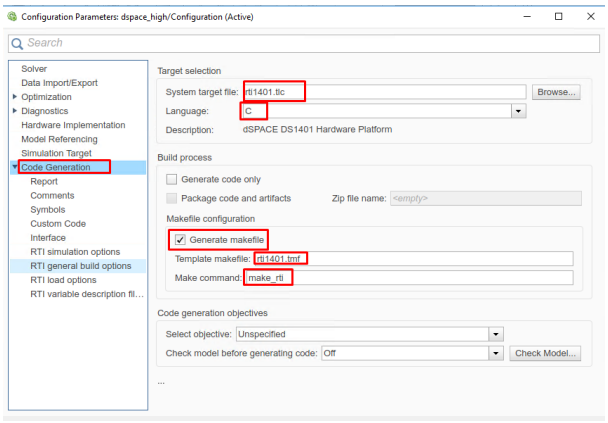


Figure 10.9: Set the Simulink code generation options.

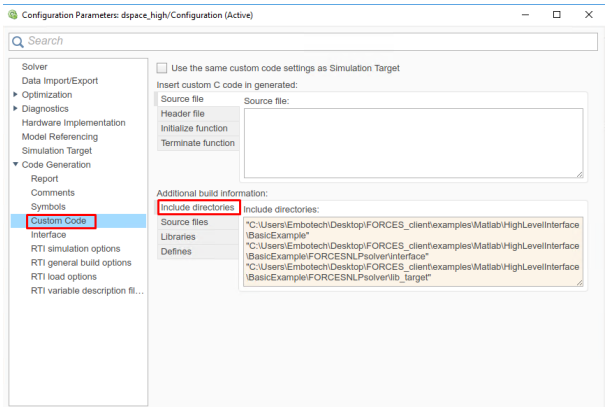


Figure 10.10: Add the directories included for the code generation.

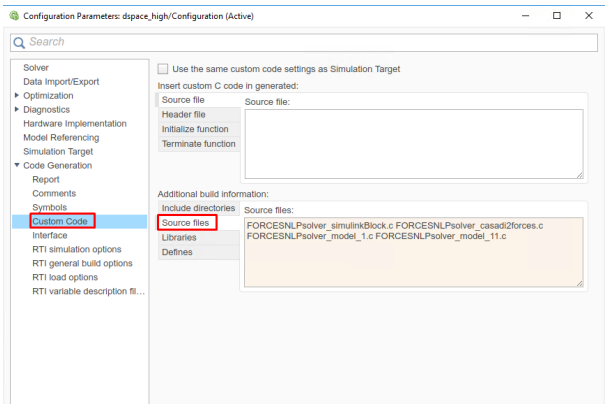


Figure 10.11: Add the source files used for the code generation.

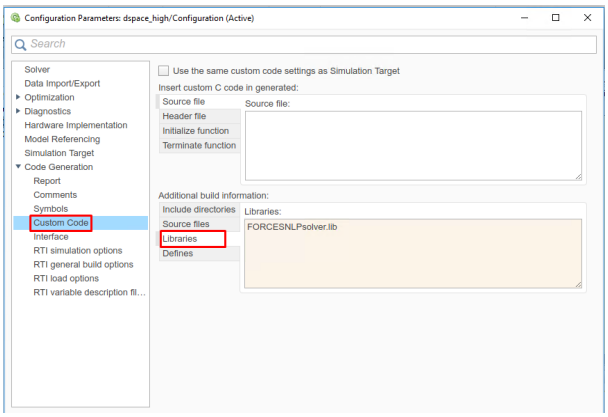


Figure 10.12: Add the libraries used for the code generation.

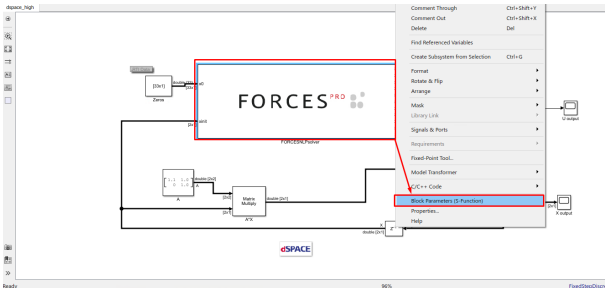


Figure 10.13: Open the FORCES PRO block's parameters.

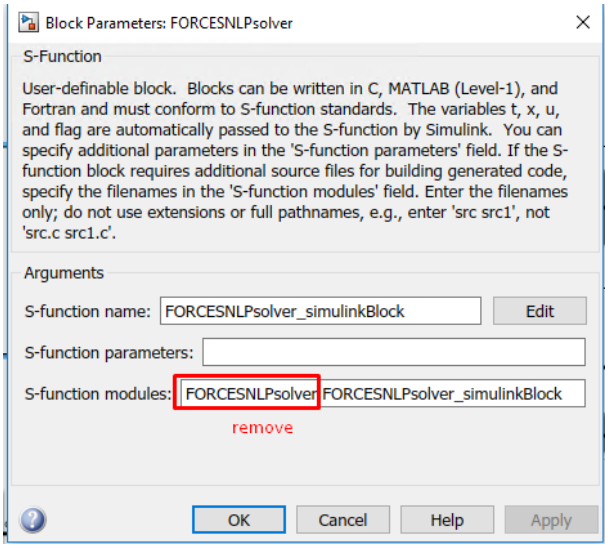


Figure 10.14: Remove the leading solver name from the S-function module.

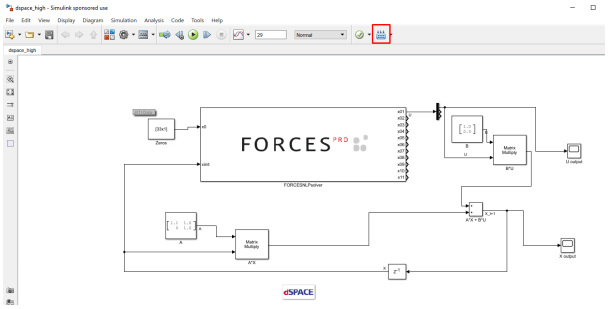


Figure 10.15: Compile the code of the Simulink model.

10.2.2 Y2F interface

The steps to deploy and simulate a FORCES PRO controller on a dSPACE MicroAutoBox II are detailed below.

1. (Figure 10.16) Set the code generation options:

```
codeoptions.platform = 'dSPACE-MABII'; % 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 10.17) Create a new Simulink model using the RTI1401 template provided by dSPACE.
3. (Figure 10.18) Populate the Simulink model with the system you want to control.
4. (Figure 10.19) Make sure the `simplempc_solver_simulinkBlock.mexw64` file (created during code generation) is on the Matlab path.
5. (Figure 10.20) Copy-paste the FORCES PRO 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 10.21) Access the Simulink model's options.
7. (Figure 10.22) In the "Solver" tab, set the options:
   - Simulation start/stop time: Depending on the simulation wanted.
   - Solver type: Discrete or fixed-step (variable-step solvers are not supported by the MicroAutoBox).
   - Fixed-step size: Needs to be higher than the execution time of the solver.
8. (Figure 10.23) In the "Code Generation/RTI general build options" tab, set the options:
   - System target file: `rti1401.tlc`
   - Language: C
   - Generate makefile: On
   - Template makefile: `rti1401.tmf`
   - Make command: `make_rti`
9. (Figure 10.24) In the "Code Generation/Custom Code" tab, include the directories:
   - `Y2F`
   - `Y2F\simplempc_solver\interface`
   - `Y2F\simplempc_solver\lib_target`
10. (Figure 10.25) In the "Code Generation/Custom Code" tab, add the source files:
    - `simplempc_solver_simulinkBlock.c`
    - `simplempc_solver.c`
11. (Figure 10.26) In the "Code Generation/Custom Code" tab, add the library files:
    - `internal_simplempc_solver_1.lib`
12. (Figure 10.27) Compile the code of the Simulink model. This will also automatically load the model to the connected MicroAutoBox.

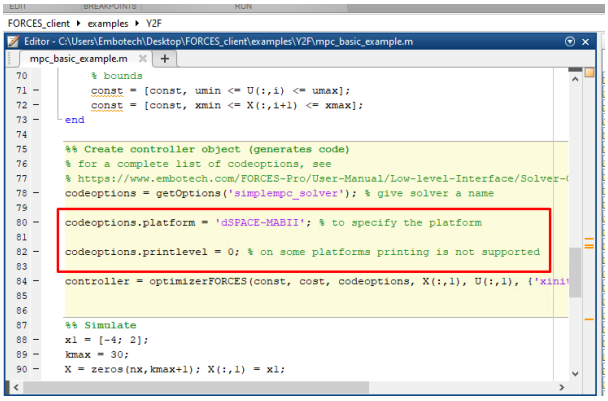


Figure 10.16: Set the appropriate code generation options.

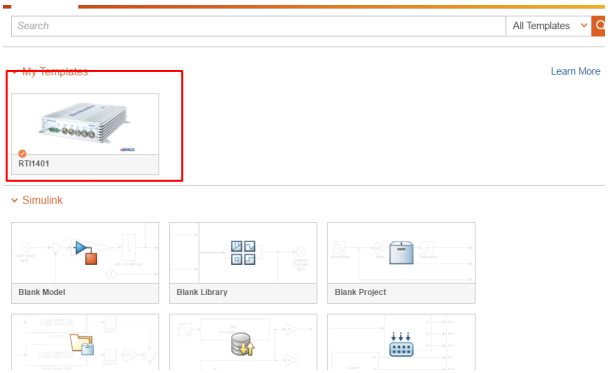


Figure 10.17: Create a Simulink model.

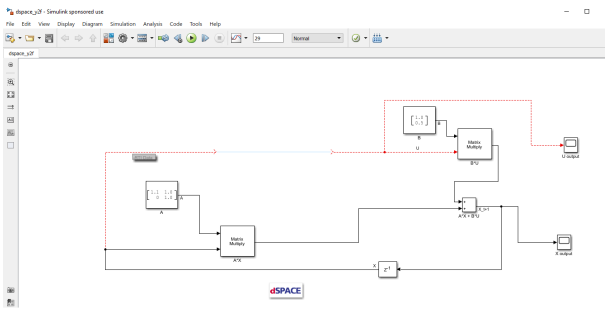


Figure 10.18: Populate the Simulink model.

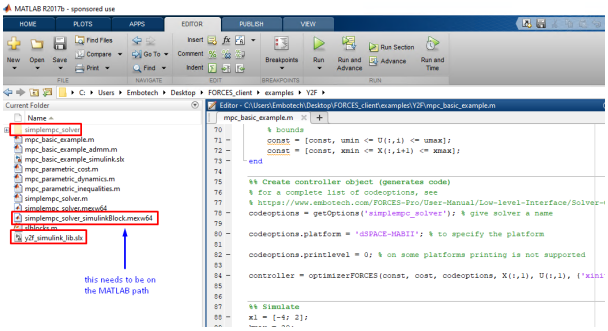


Figure 10.19: Add the folder containing the `.mexw64` solver file to the Matlab path.

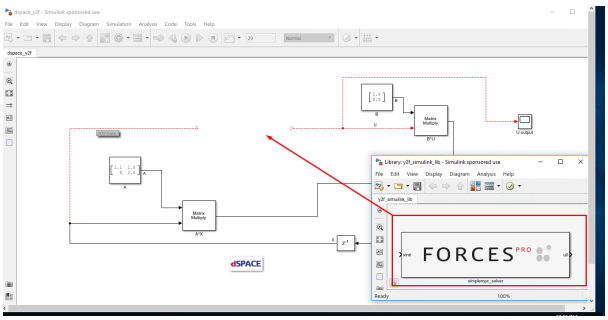


Figure 10.20: Copy-paste and connect the FORCES PRO block.

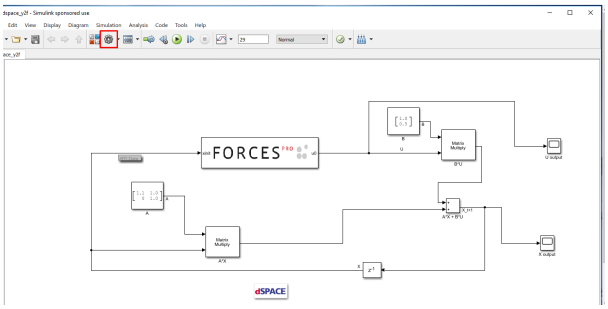


Figure 10.21: Open the Simulink model options.

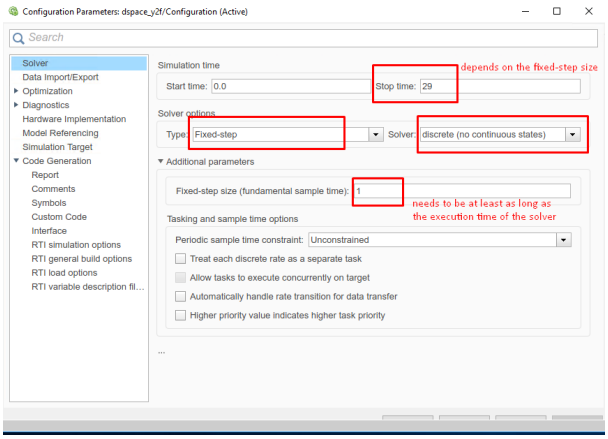


Figure 10.22: Set the Simulink solver options.

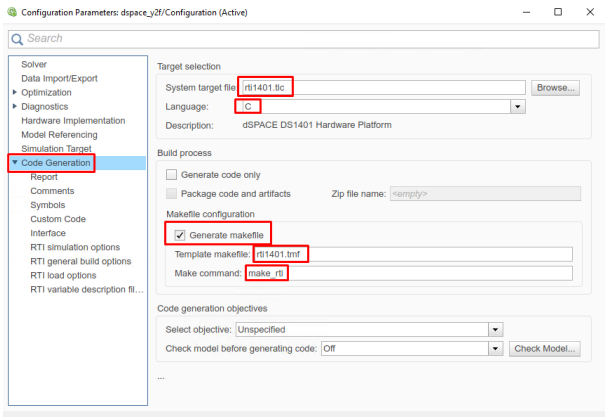


Figure 10.23: Set the Simulink code generation options.

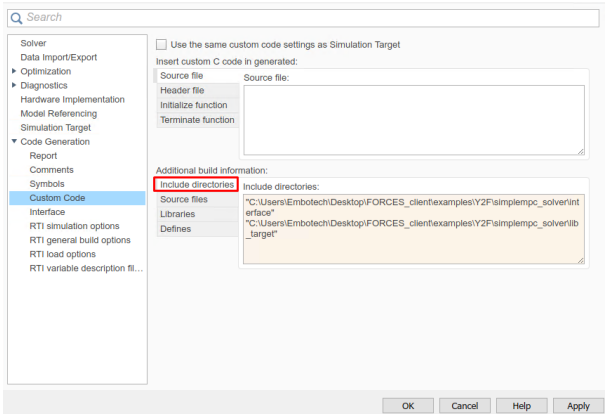


Figure 10.24: Add the directories included for the code generation.

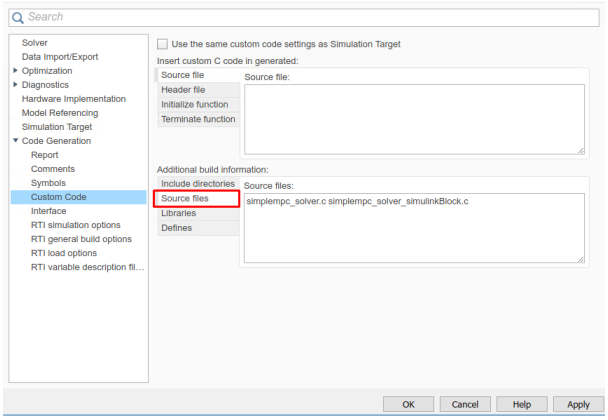


Figure 10.25: Add the source files used for the code generation.

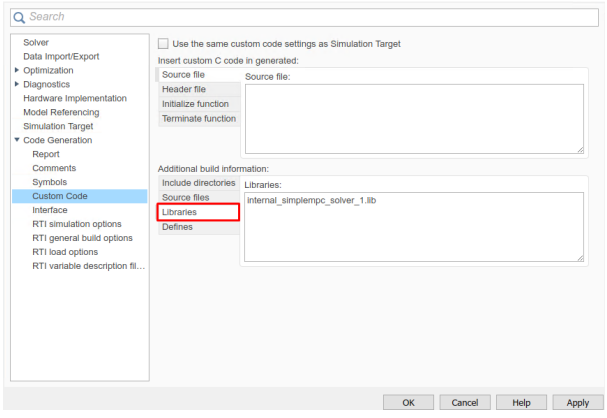


Figure 10.26: Add the libraries used for the code generation.

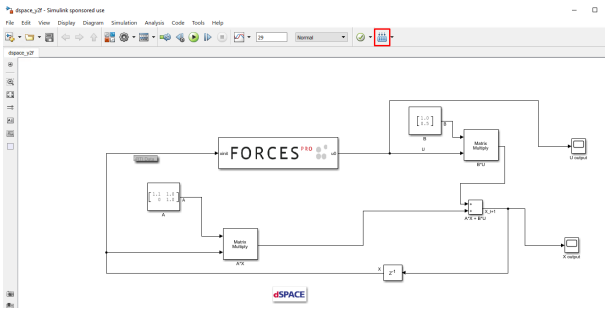


Figure 10.27: Compile the code of the Simulink model.

10.3 dSPACE MicroAutoBox III

10.3.1 Code Generation

The steps to deploy a FORCES PRO controller on a dSPACE MicroAutoBox III are detailed below.

1) Set the code generation options (see Figure 10.28):

```
codeoptions.platform = 'dSPACE-MABXIII'; % to generate code for the MicroAutoBox␣
↪III
codeoptions.printlevel = 0; % printing is not supported for the MicroAutoBox III
codeoptions.cleanup = 0; % to keep necessary files for target compile
```

Important: When generating code for the MicroAutoBox III, `codeoptions.optlevel` can take values 0-4 instead of 0-3 where

- 0: no optimization
- 1: optimize for size
- 2: optimize for speed
- 3: optimize for size and speed
- 4: optimize for size and speed with more precise numerics

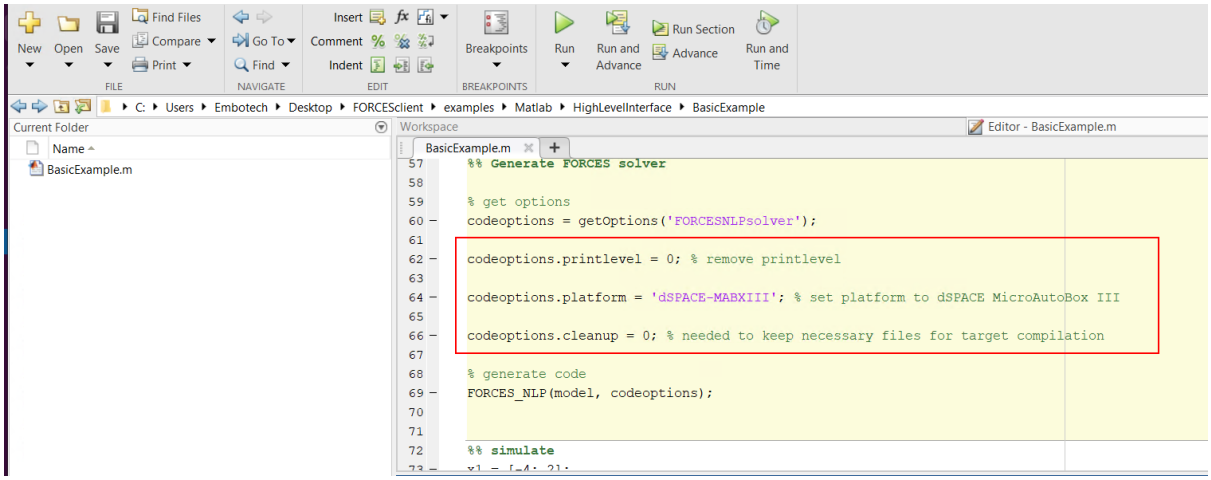


Figure 10.28: Set the appropriate code generation options.

2) Create a new Simulink model (henceforth referred to as `dSPACE-MABXIII.slx`) using the `dSPACE Run-Time Target` template provided by dSPACE and save it in the `BasicExample` folder (see Figure 10.29).

3) Populate the Simulink model with the system you want to control (see Figure 10.30).

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 10.31 , Figure 10.32 and Figure 10.33).

5) The `FORCESNLPsolver_simulinkBlock.<mex_extension>` file (created during code generation) needs to be in the same path as your model (see Figure 10.34).

6) Open the `FORCESNLPsolver_lib.mdl` Simulink model file, contained in the `interface` folder of the `FORCESNLPsolver` folder created during code generation (see Figure 10.35).

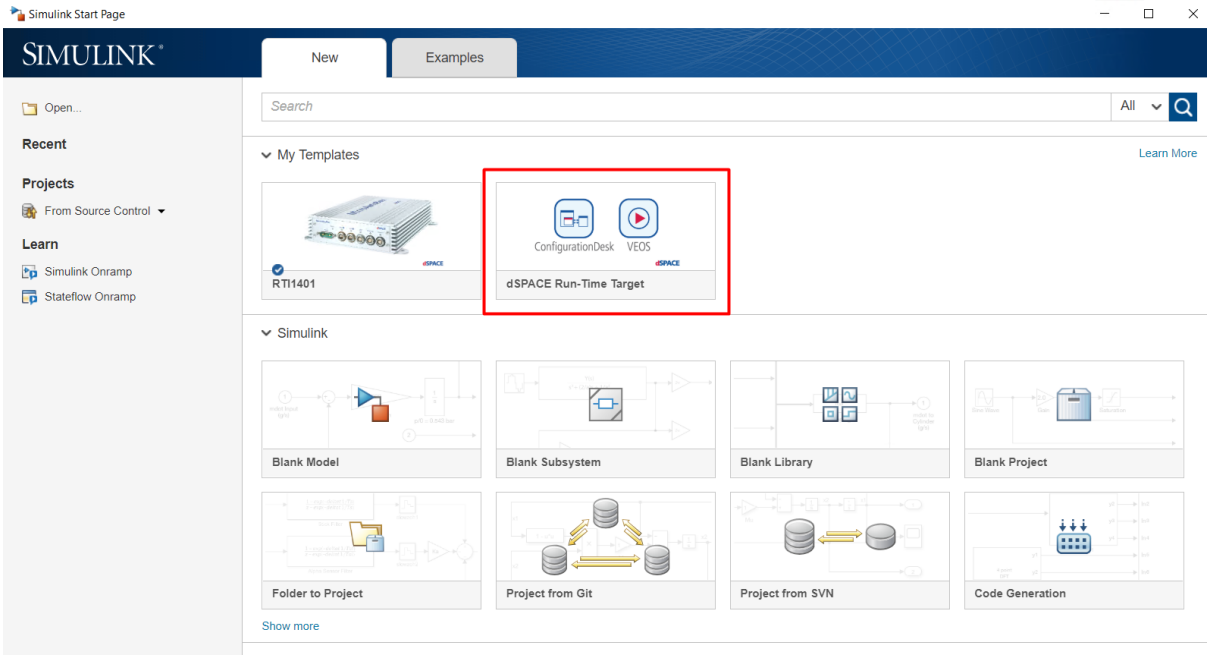


Figure 10.29: Create a Simulink model.

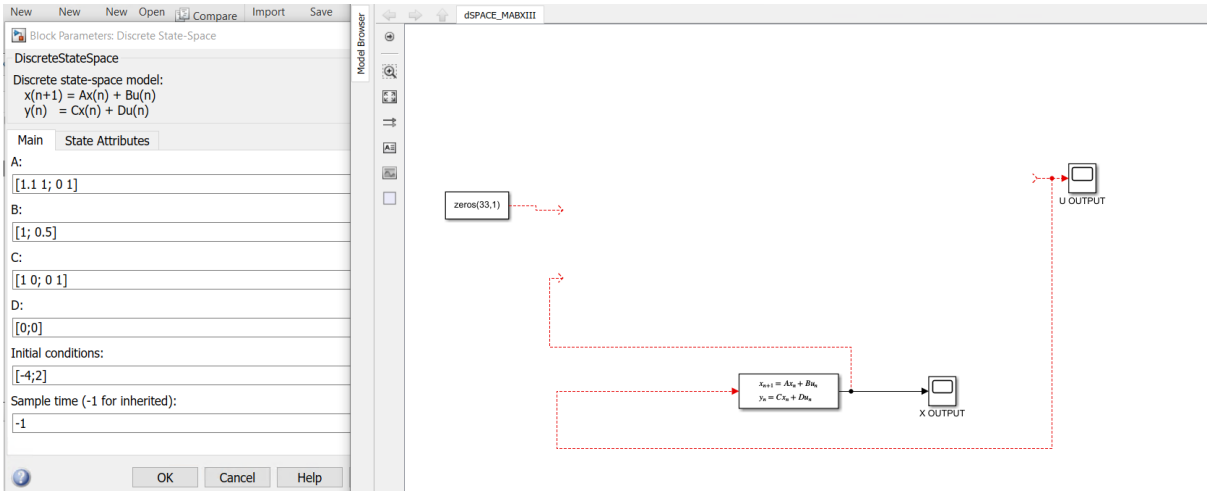


Figure 10.30: Populate the Simulink model.

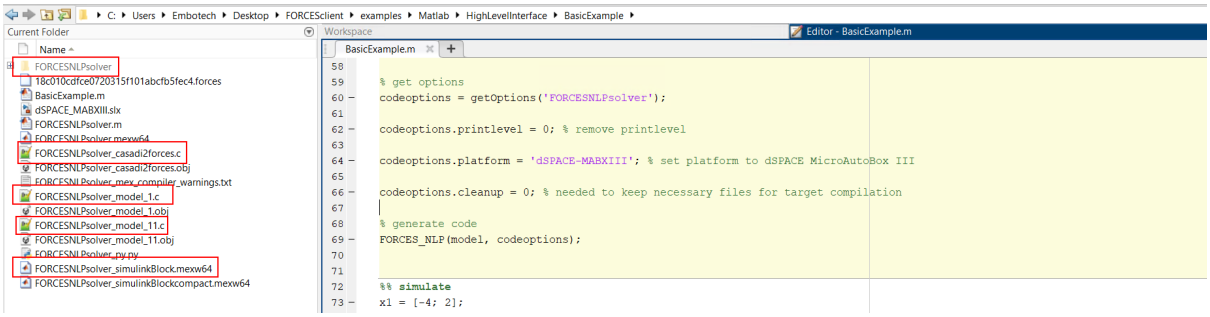


Figure 10.31: Generated files.

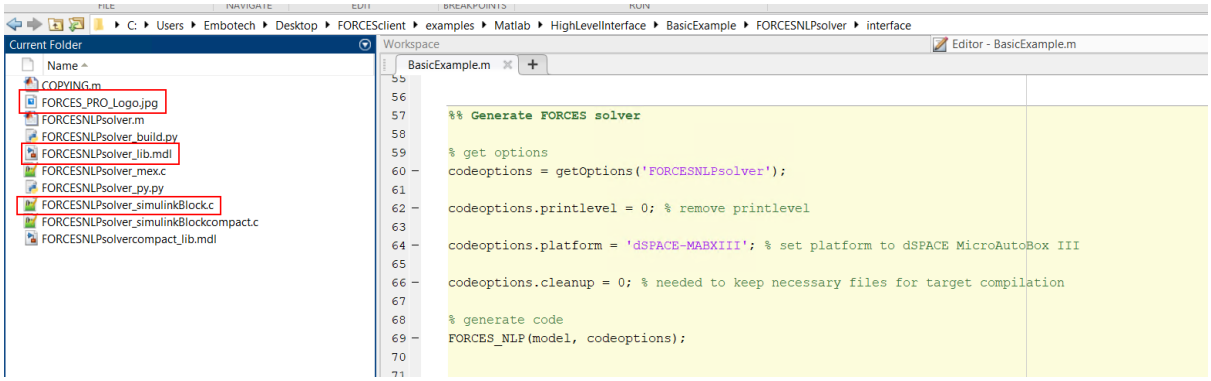


Figure 10.32: Solver interface files.

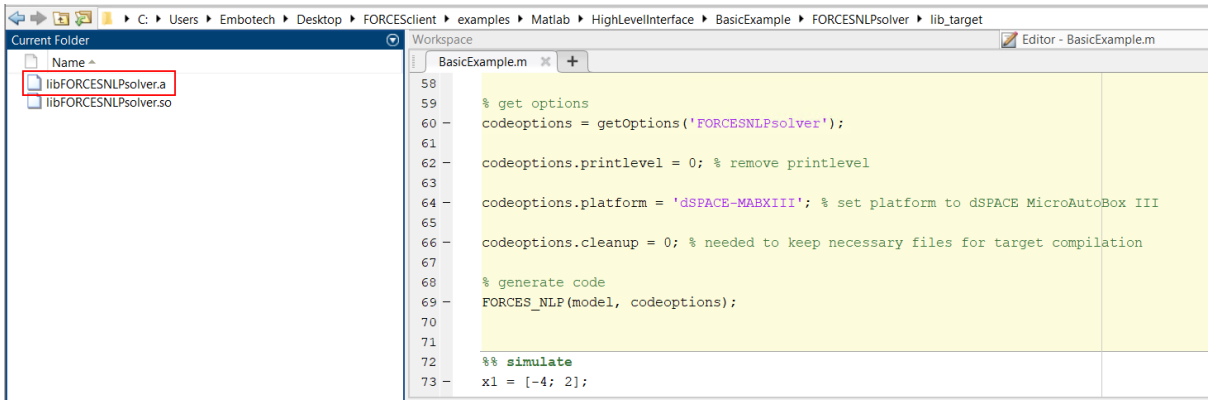


Figure 10.33: Solver libraries.

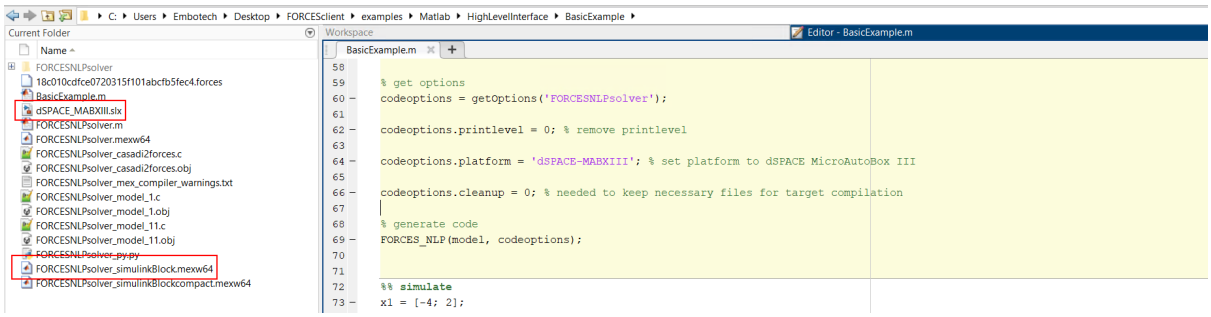


Figure 10.34: The `.<mex_extension>` solver file is in the same path as the model.

7) Copy-paste the FORCES PRO Simulink block into your simulation model and connect its inputs and outputs appropriately (see Figure 10.36).

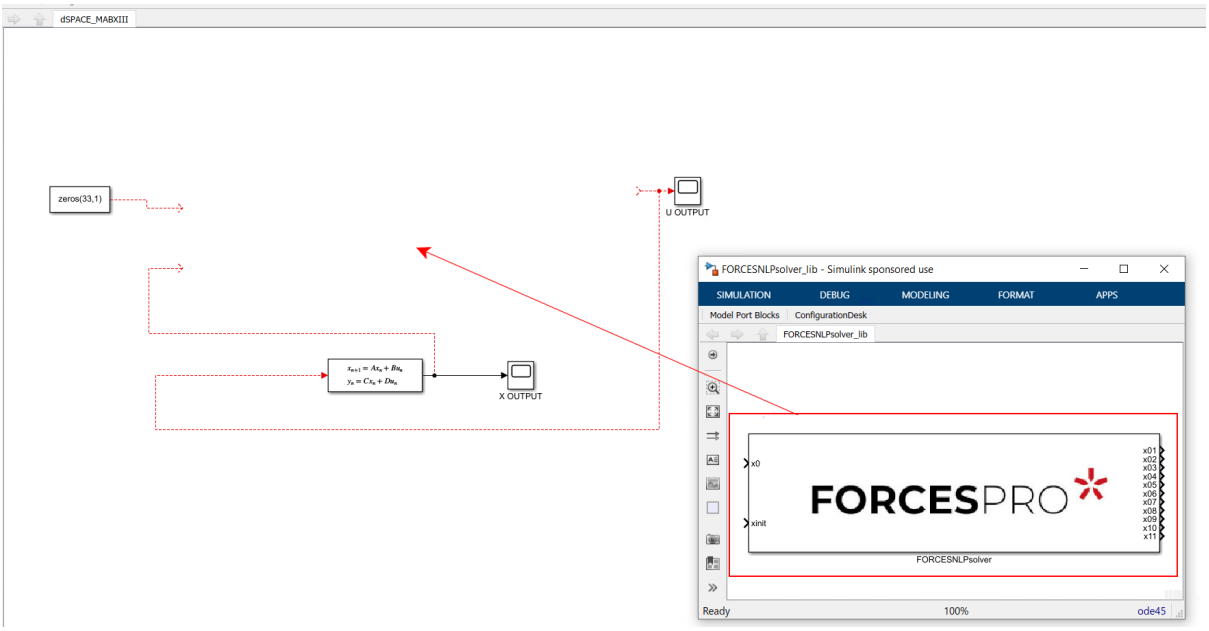


Figure 10.35: Open the generated Simulink solver model.

8) Access the Simulink model's options. In the "Solver" tab, set the options (see Figure 10.37):
   - 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 10.38):
   - System target file: `dsrt.tlc`
   - Language: `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 10.39):
   - `.\FORCESNLPsolver\include`
   - `.\FORCESNLPsolver\interface`
   - `.\FORCESNLPsolver\lib_target`
11) In the "Code Generation/Custom Code" tab, add the source files (see Figure 10.40):
   - `FORCESNLPsolver_simulinkBlock.c`
   - `FORCESNLPsolver_casadi2forces.c`
   - `FORCESNLPsolver_model_1.c`
   - `FORCESNLPsolver_model_11.c`
12) In the "Code Generation/Custom Code" tab, add the library files (see Figure 10.41):
   - `libFORCESNLPsolver.a`
13) Access the FORCES PRO block's parameters (see Figure 10.42).

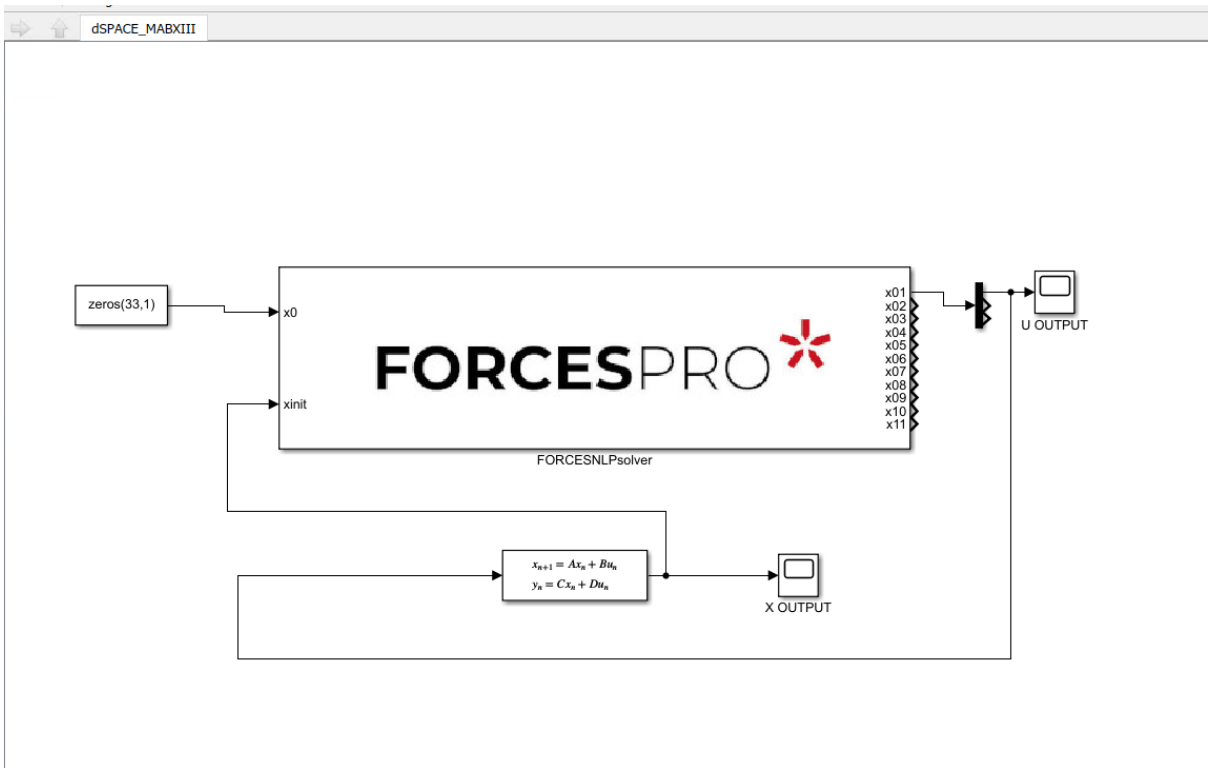


Figure 10.36: Copy-paste and connect the FORCES PRO block.

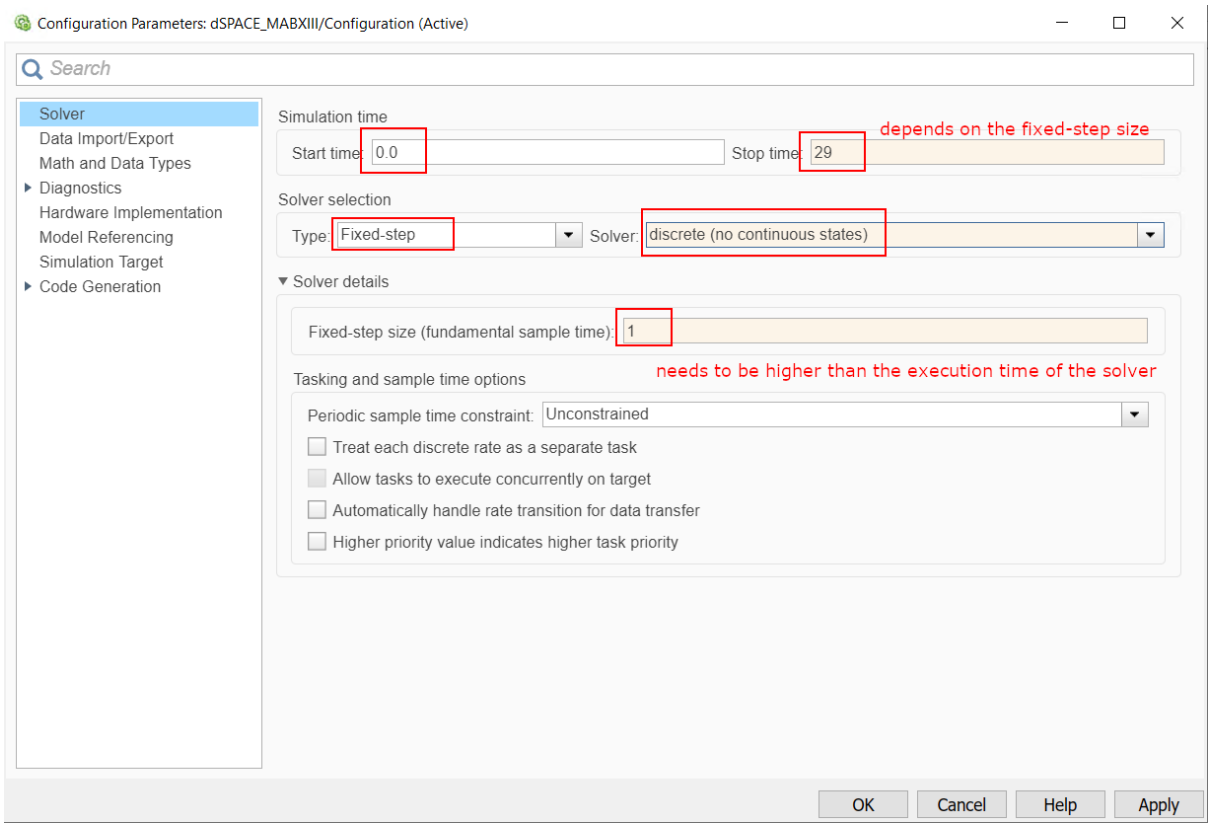


Figure 10.37: Set the Simulink solver options.

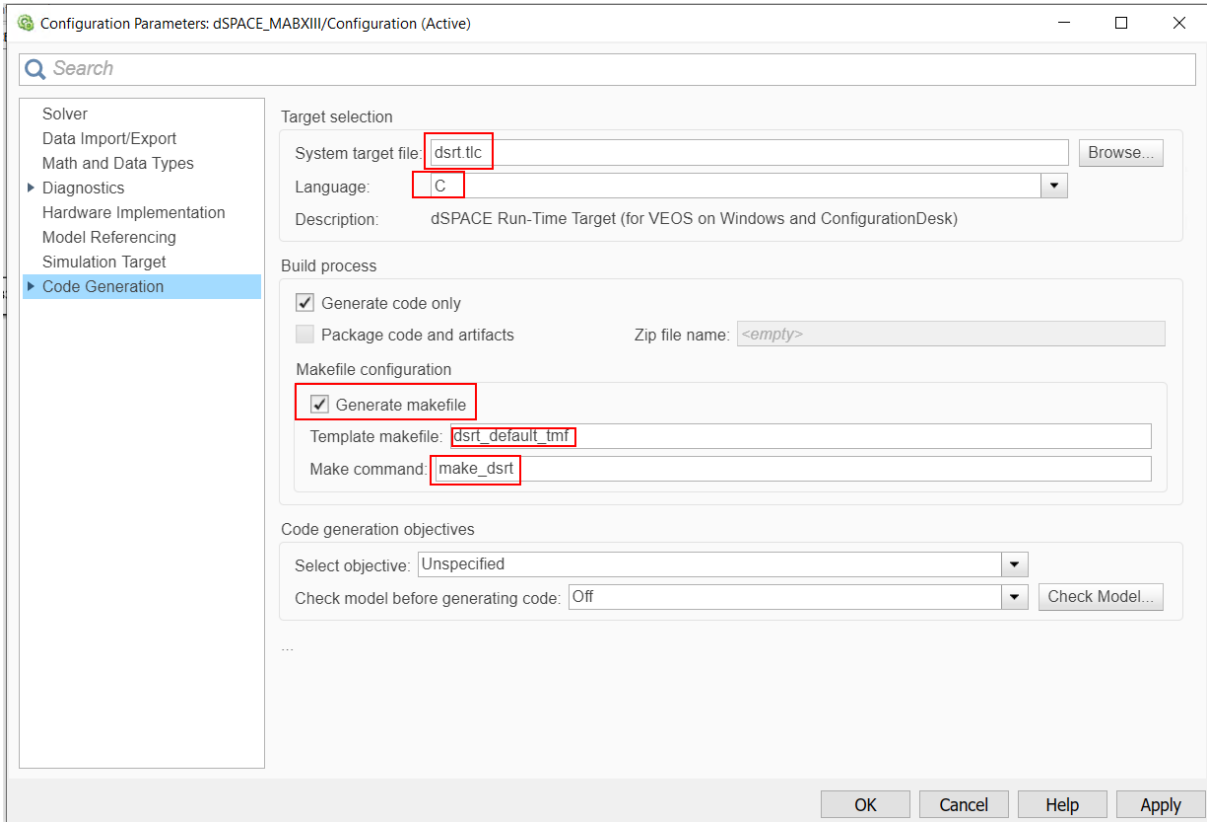


Figure 10.38: Set the Simulink code generation options.

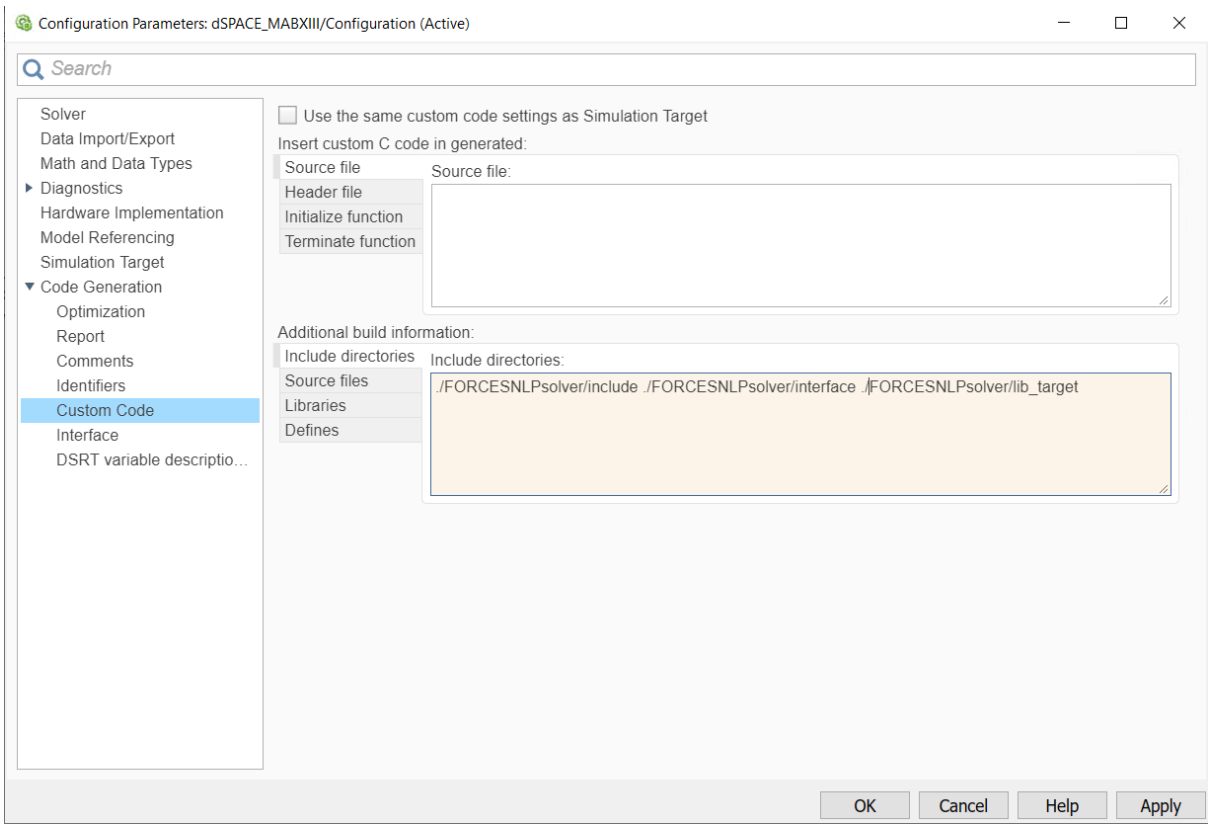


Figure 10.39: Add the directories included for the code generation.

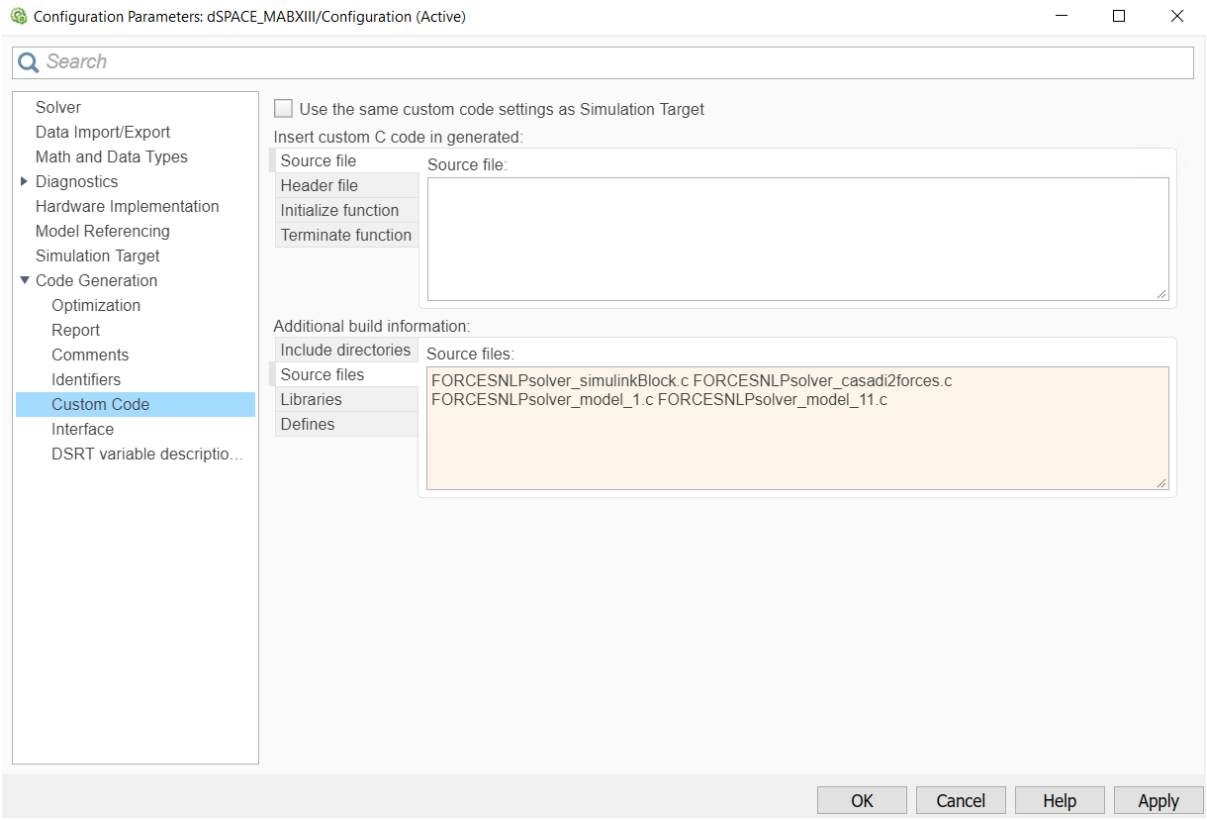


Figure 10.40: Add the source files used for the code generation.

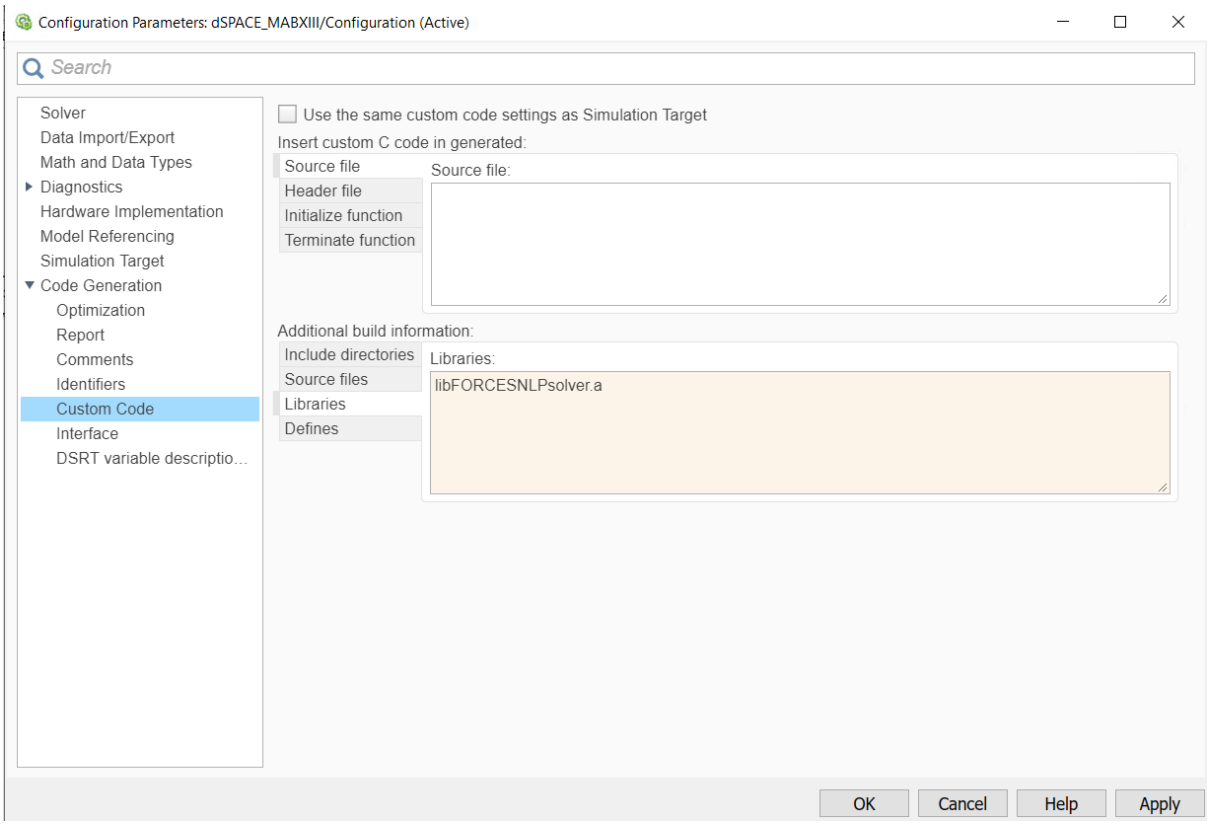


Figure 10.41: Add the libraries used for the code generation.

14) Remove the “FORCESNLPsolver” prefix from the S-function module (see Figure 10.43).

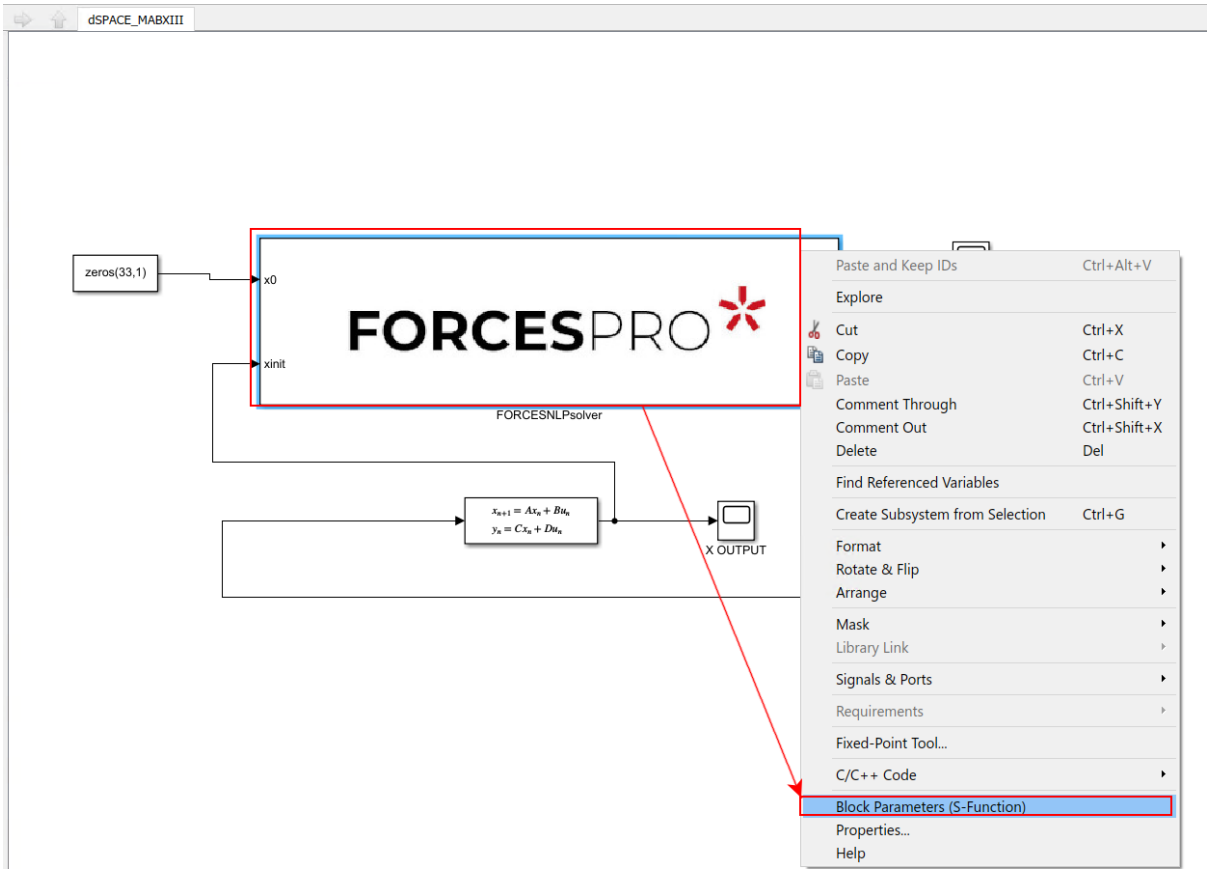


Figure 10.42: Open the FORCES PRO block’s parameters.

15) Create a new Project and Application in ConfigurationDesk. Select directory of project, name of project and application, the model `dSPACE_MABXIII.slx` as the application process and connected MicroAutoBox III to deploy to (see Figure 10.44).

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 10.45).

17) Go to the build tab and start the building process. After building is complete the application will be loaded automatically in the MicroAutoBox III (see Figure 10.46).

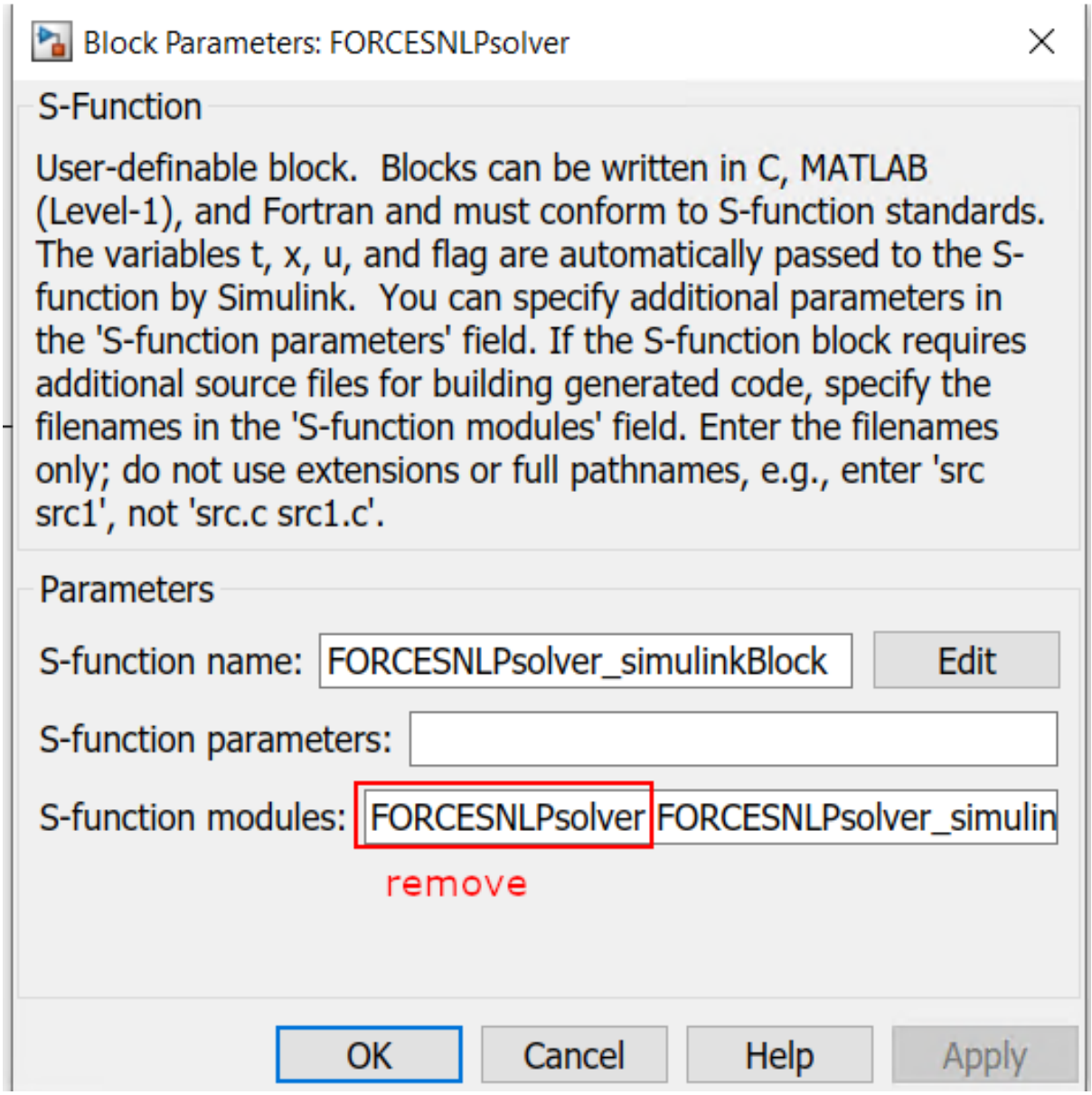


Figure 10.43: Remove the leading solver name from the S-function module.

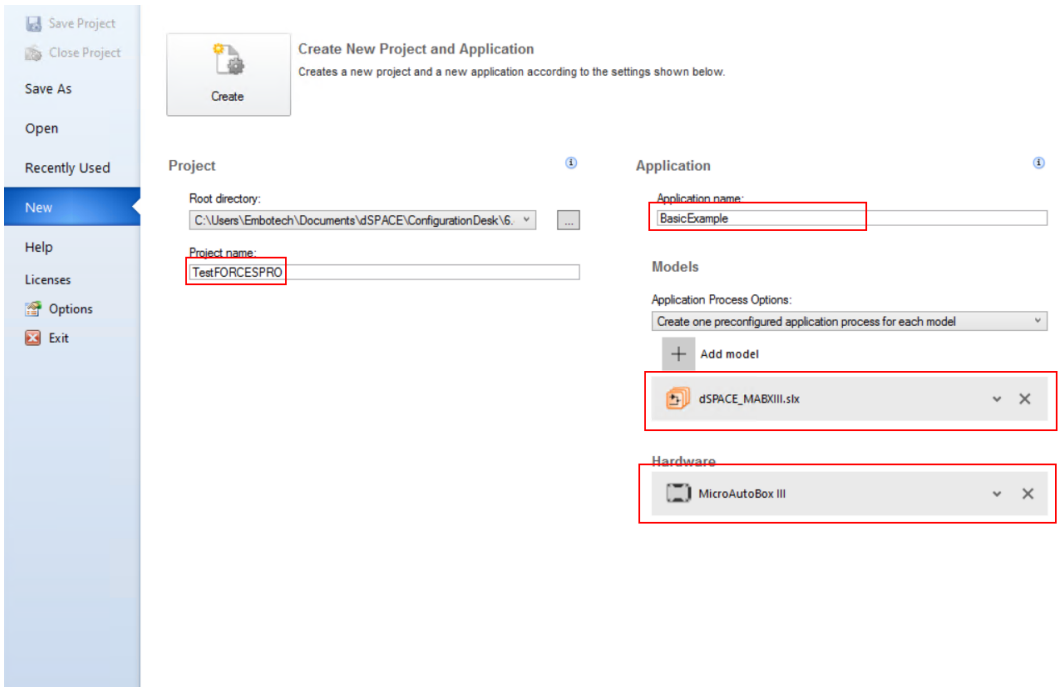


Figure 10.44: Create project and application in ConfigurationDesk.

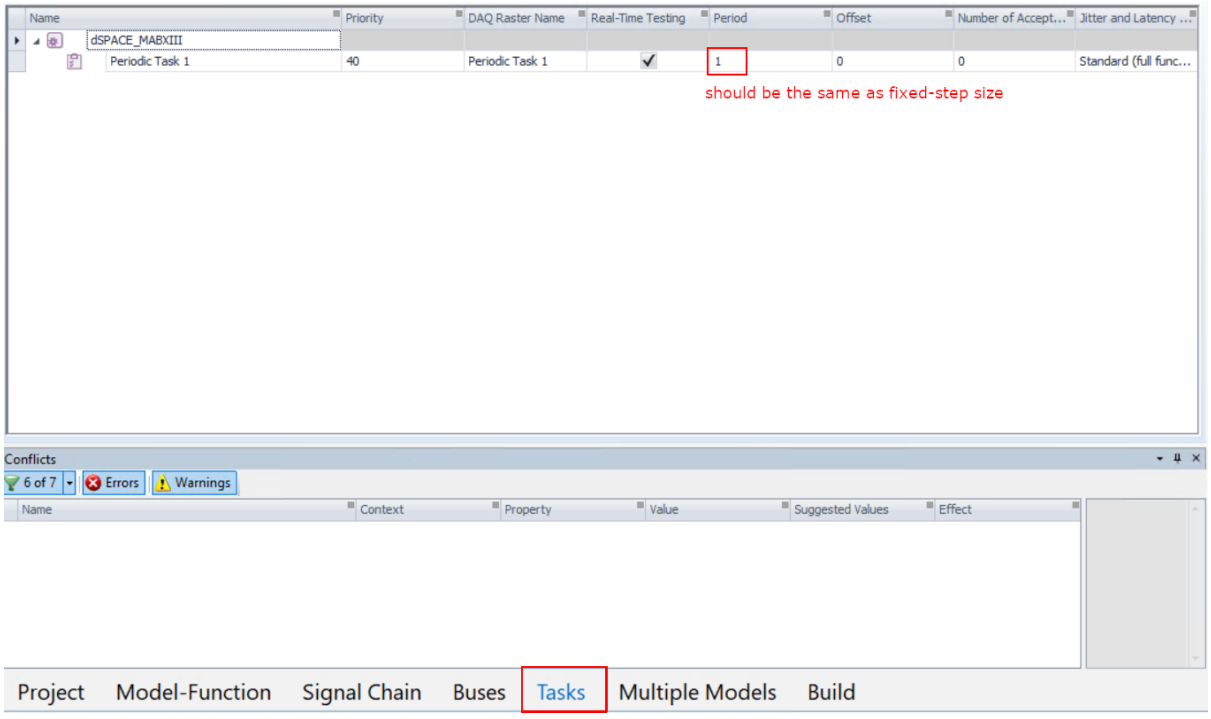


Figure 10.45: Set period of Periodic Task.

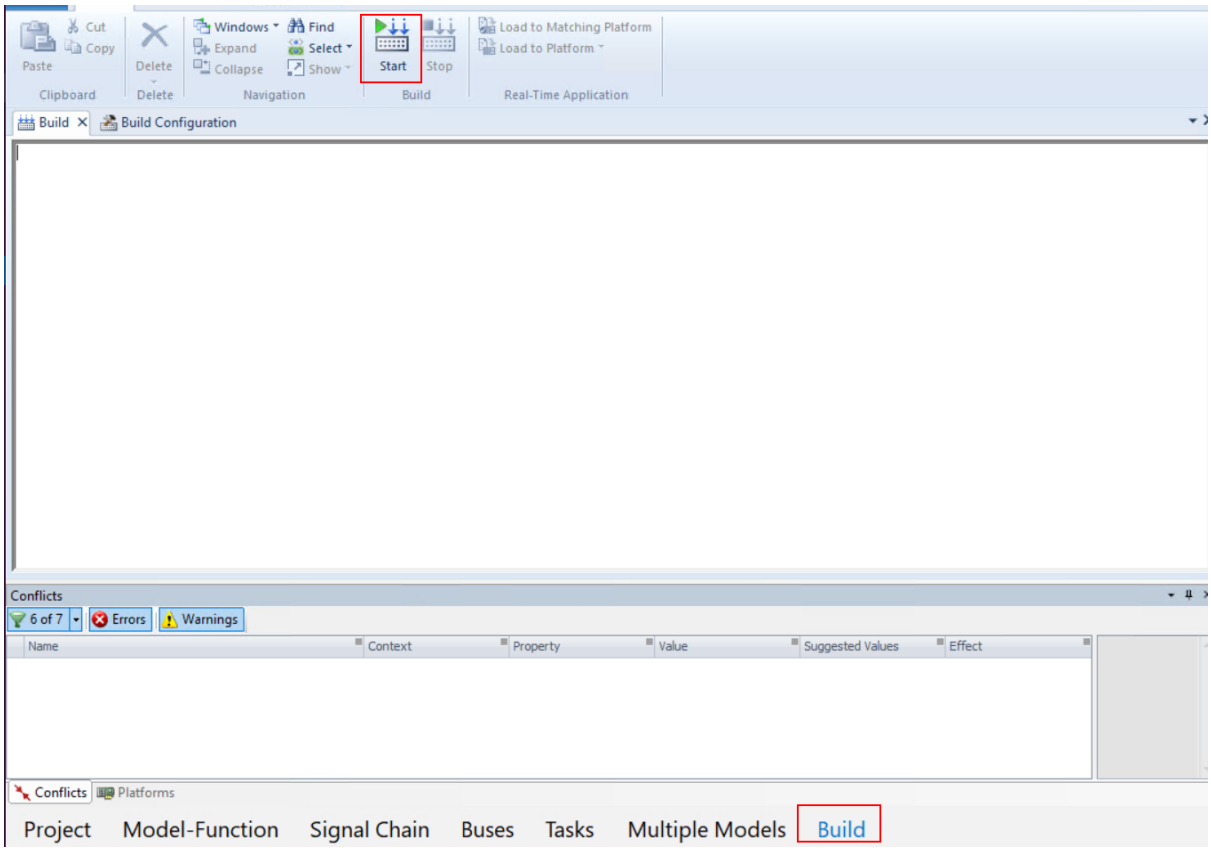


Figure 10.46: Build the project.

10.3.2 Solver Execution

The steps to simulate a FORCES PRO controller on a dSPACE MicroAutoBox III are detailed below.

1) After code generation with FORCES PRO and building with the ConfigurationDesk, the ConfigurationDesk project will have generated files to use to run your model on the MicroAutoBox III (see Figure 10.47 and Figure 10.48).

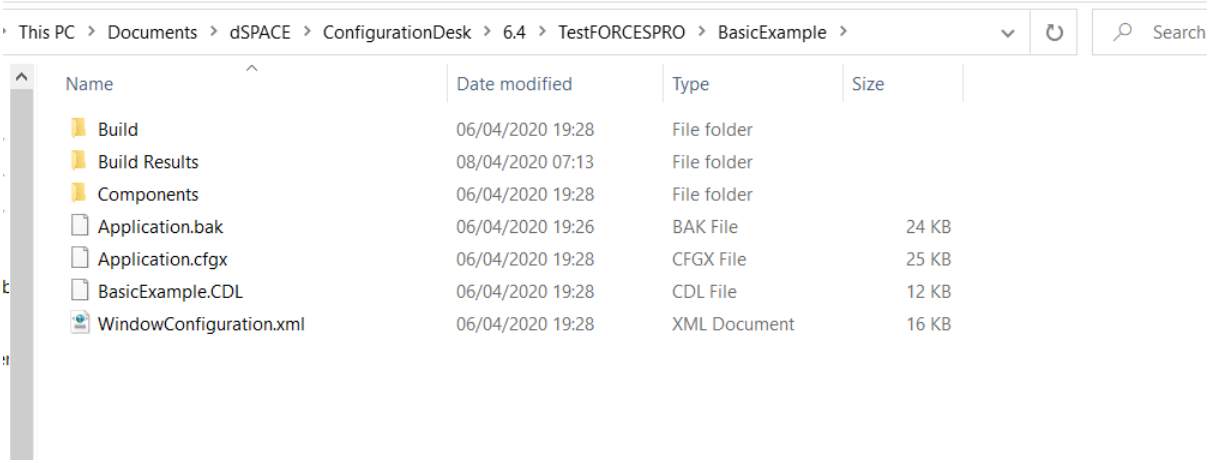


Figure 10.47: The generated files from the ConfigurationDesk building.

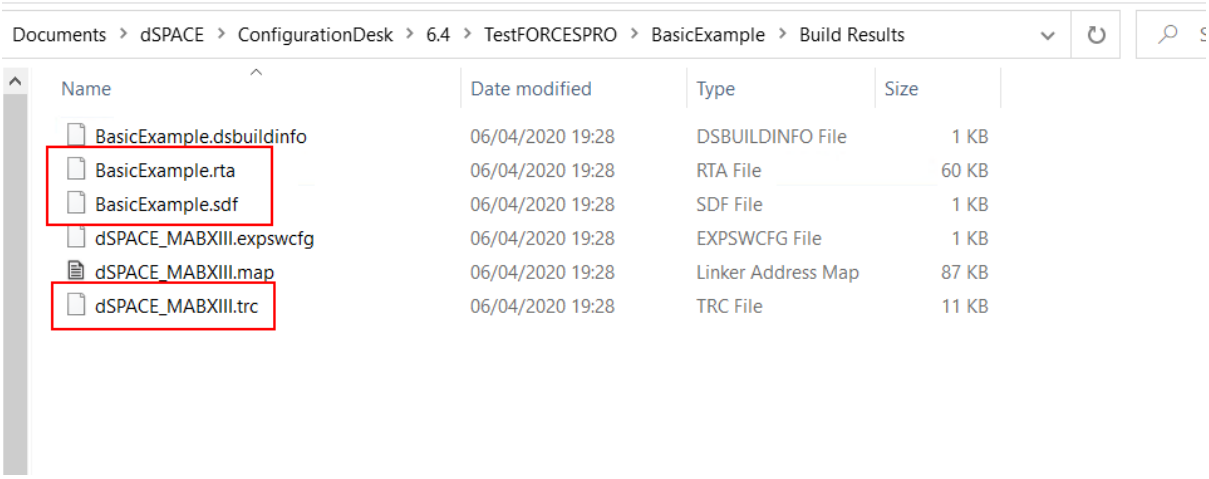


Figure 10.48: The files necessary for the simulation of the FORCES PRO controller.

2) Open dSpace Control Desk and select create new project and name it (see Figure 10.49).
3) Name the experiment to execute (see Figure 10.50).
4) Select the platform to which you will deploy the generated executable (see Figure 10.51).
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 10.52).
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 10.53 and Figure 10.54).
8) To see all the plots concurrently right-click on the left of the Y-axis and select `YAxes-view> Horizontal stacked` (see Figure 10.55).

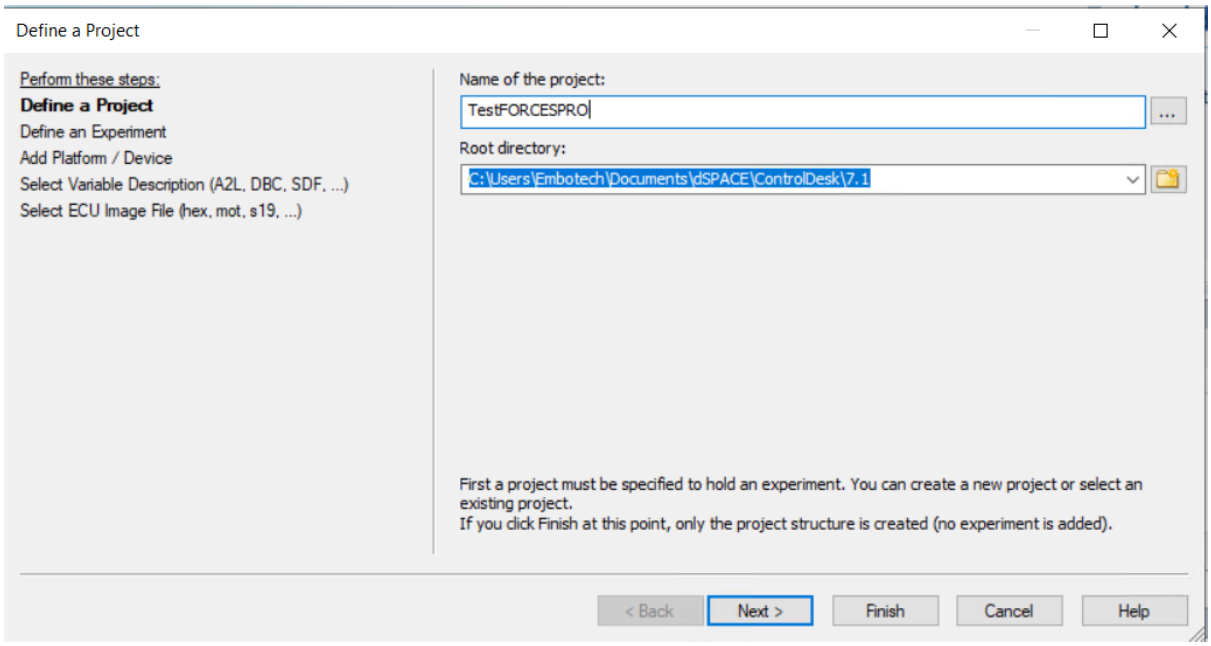


Figure 10.49: Start a new project and name it.

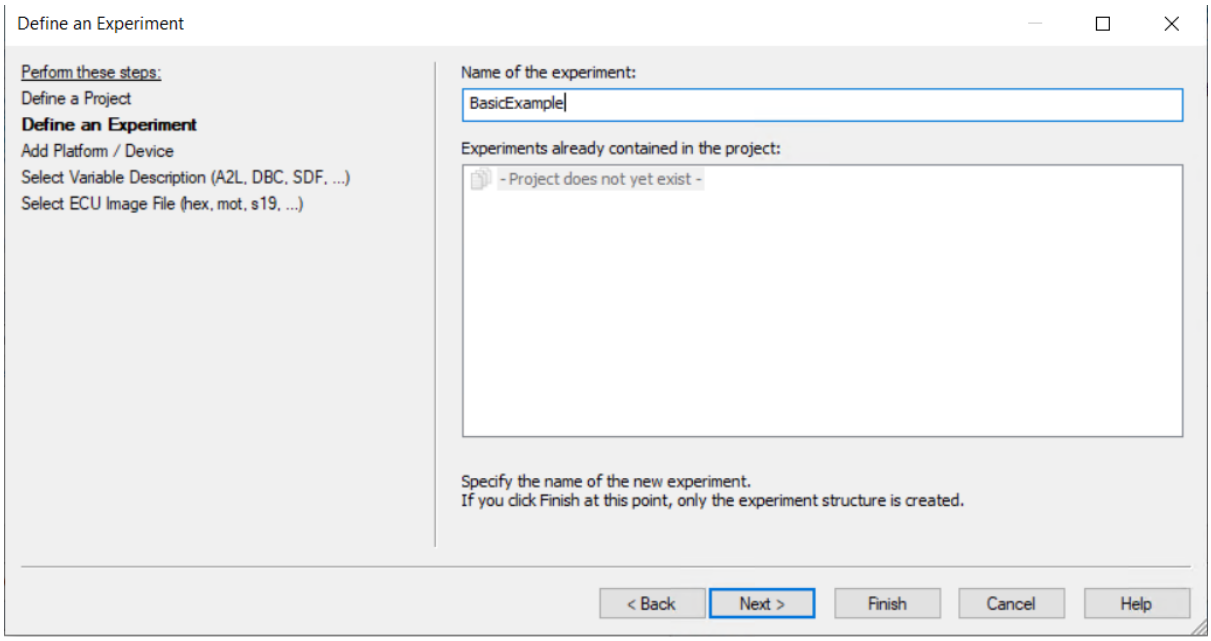


Figure 10.50: Name your experiment.

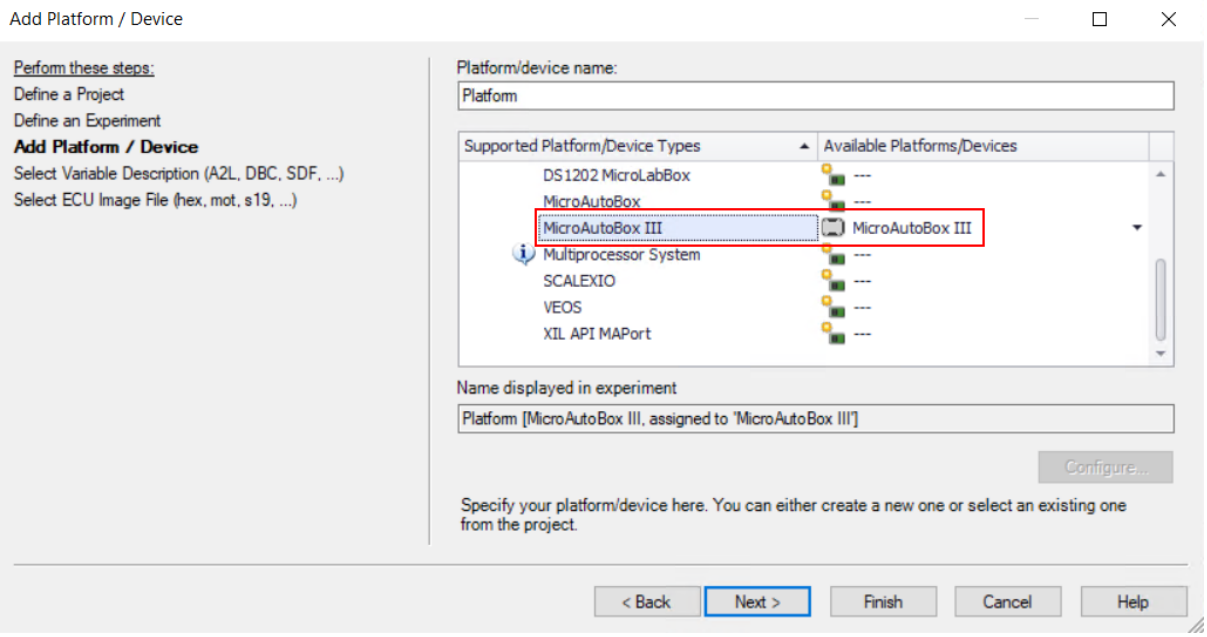


Figure 10.51: Select the MicroAutoBox III platform.

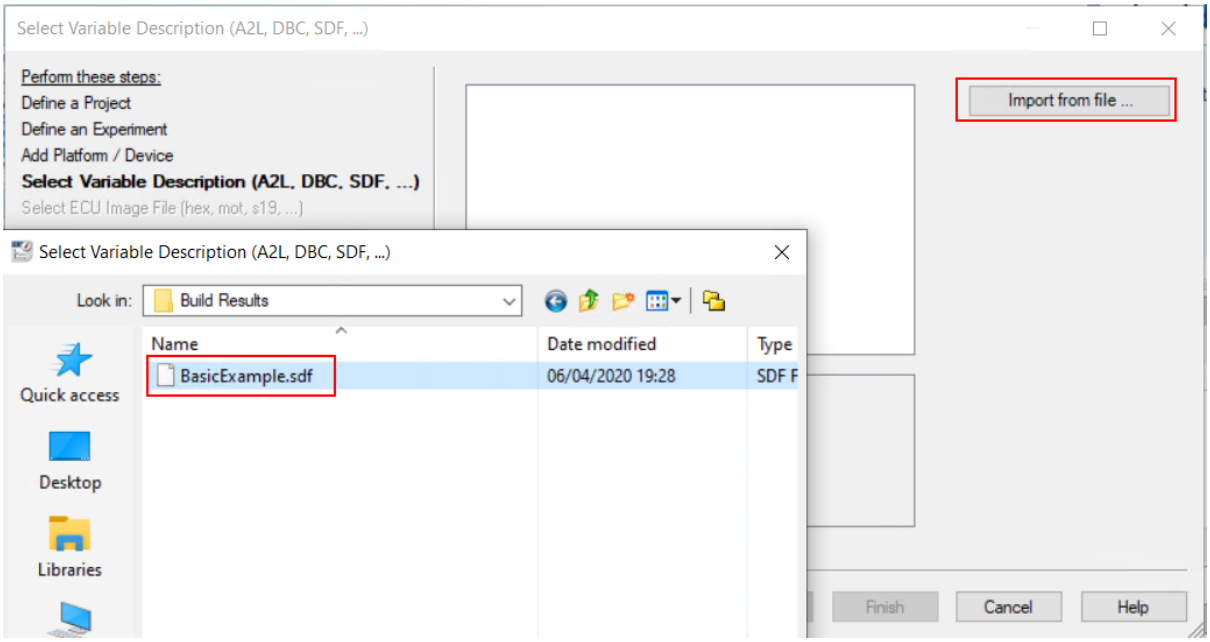


Figure 10.52: Import the variable description file.

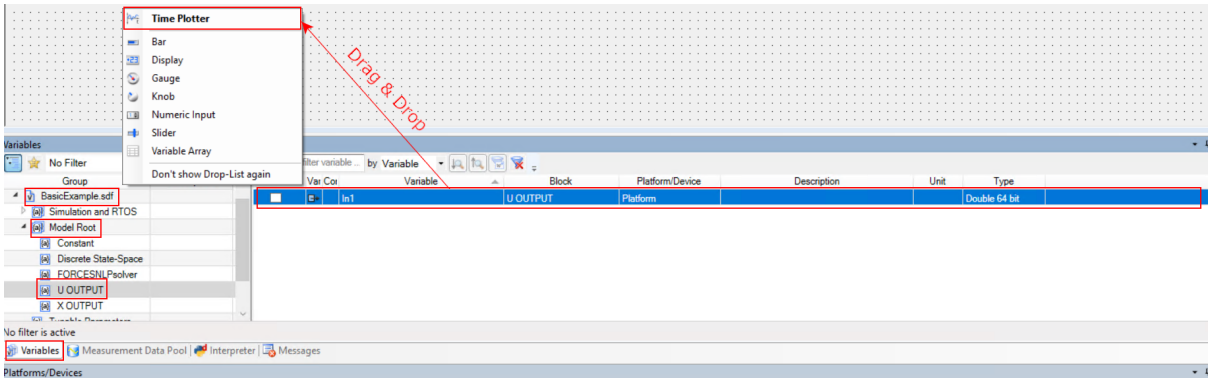


Figure 10.53: Add the inputs of U OUTPUT in a Time Plotter.

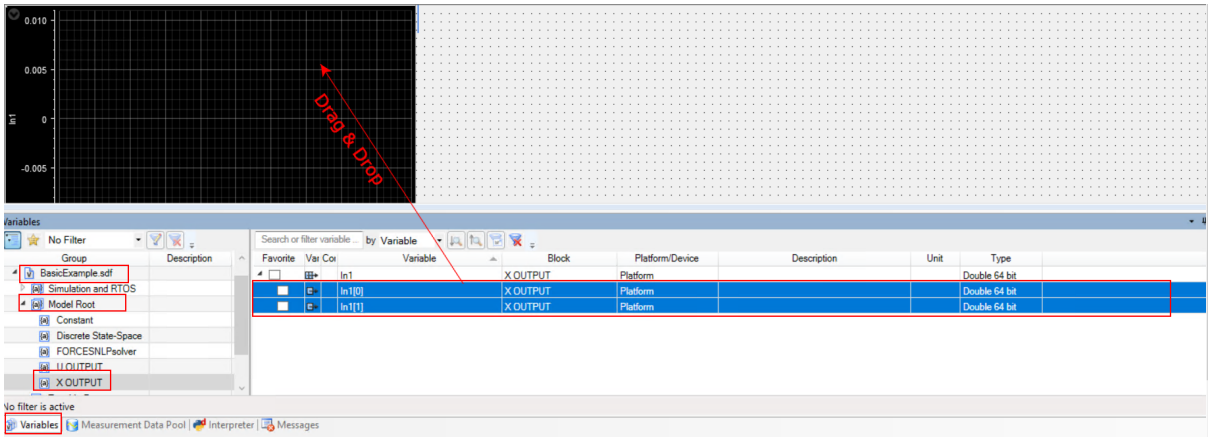


Figure 10.54: Add the inputs of X OUTPUT in the same Time Plotter.

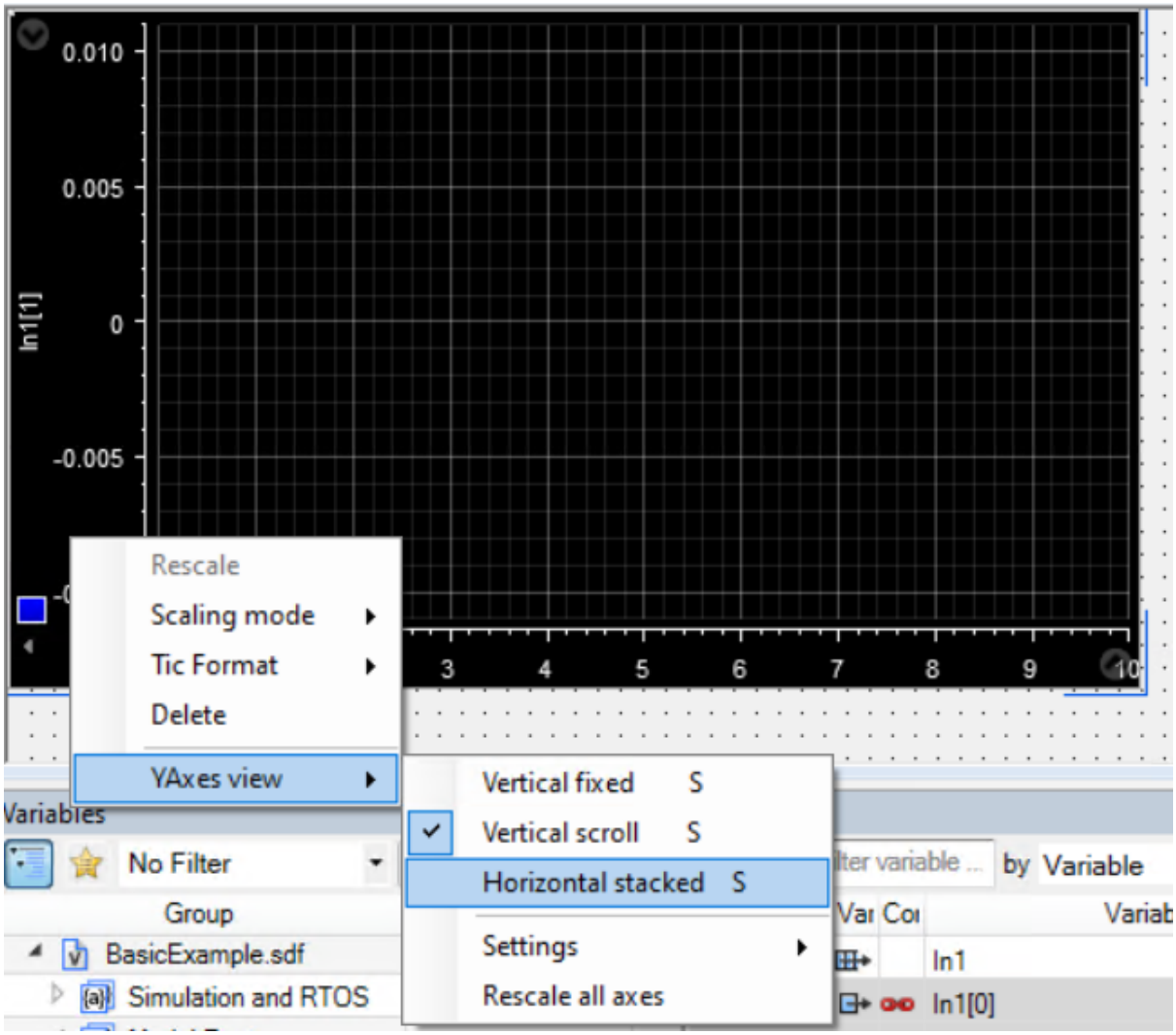


Figure 10.55: Select to show all the signals on the same plot with their own Y-axes

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 10.56 and Figure 10.57).

10) Select `Go Online` and `Start Measuring` to see the results. (see Figure 10.58 and Figure 10.59).

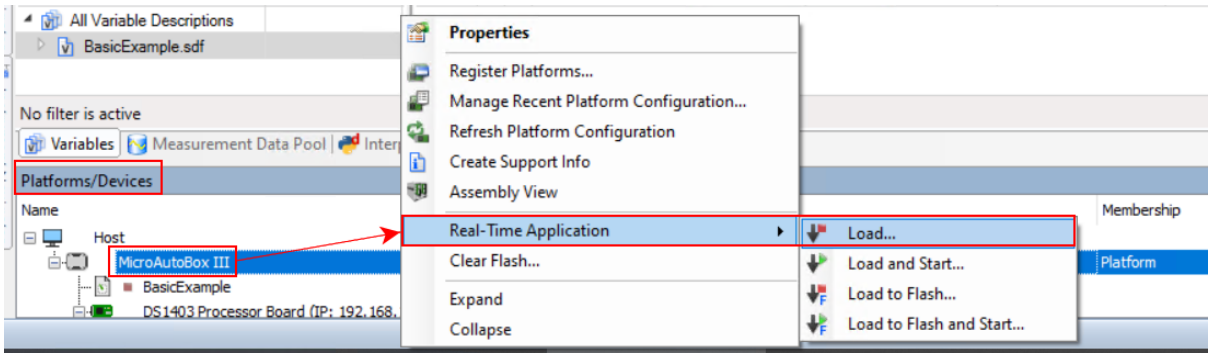


Figure 10.56: Load the application on the dSPACE MicroAutoBox III.

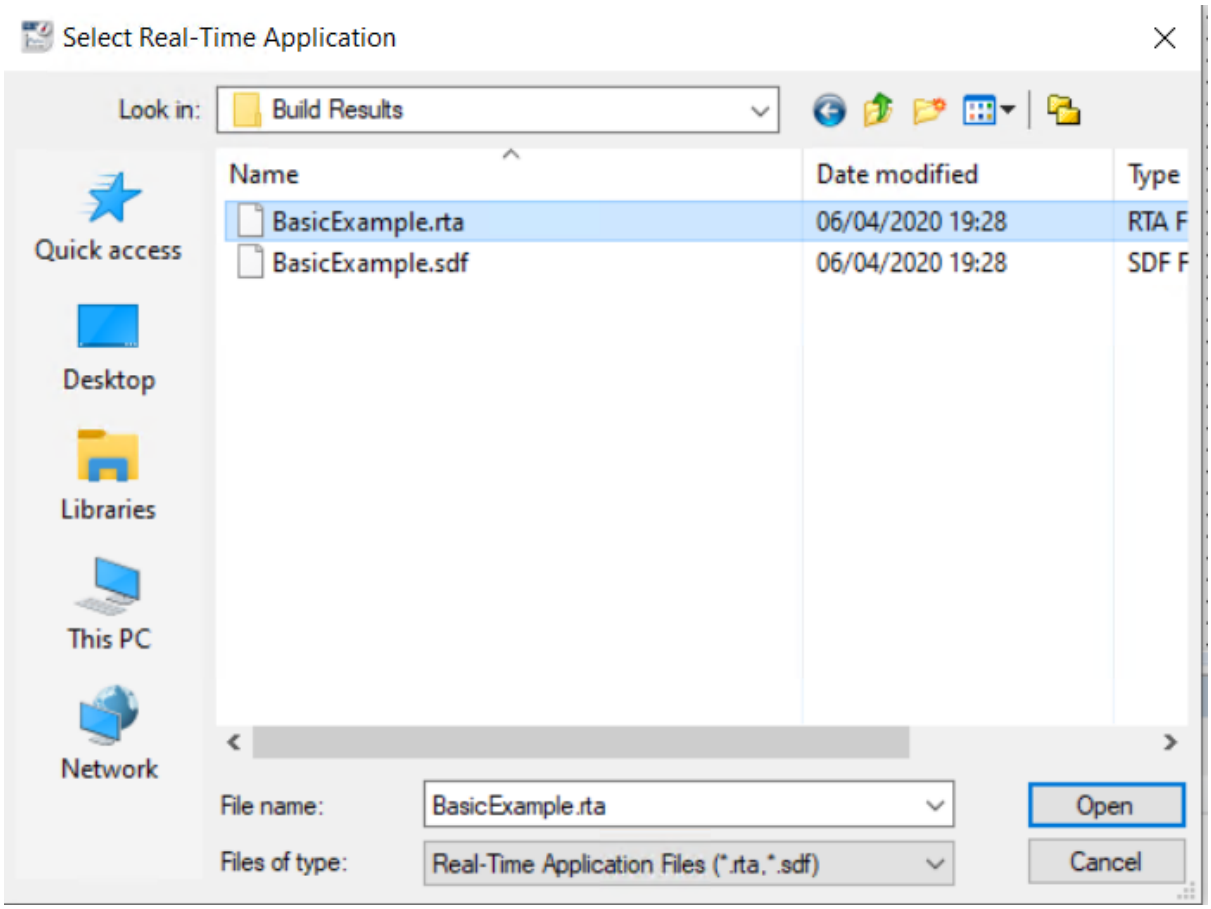


Figure 10.57: Select BasicExample.rta from the ConfigurationDesk project folder.

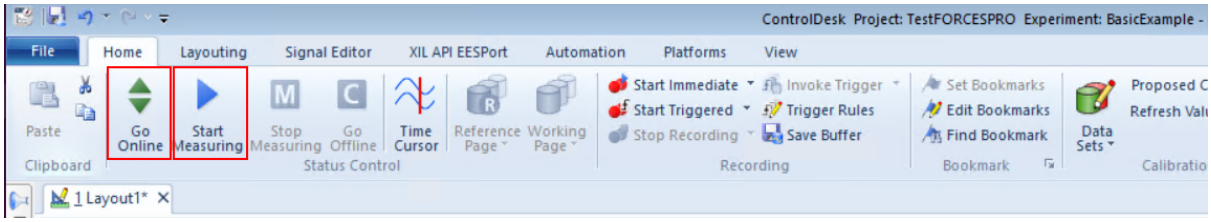


Figure 10.58: Buttons Go Online and Start Measuring to receive execution results.

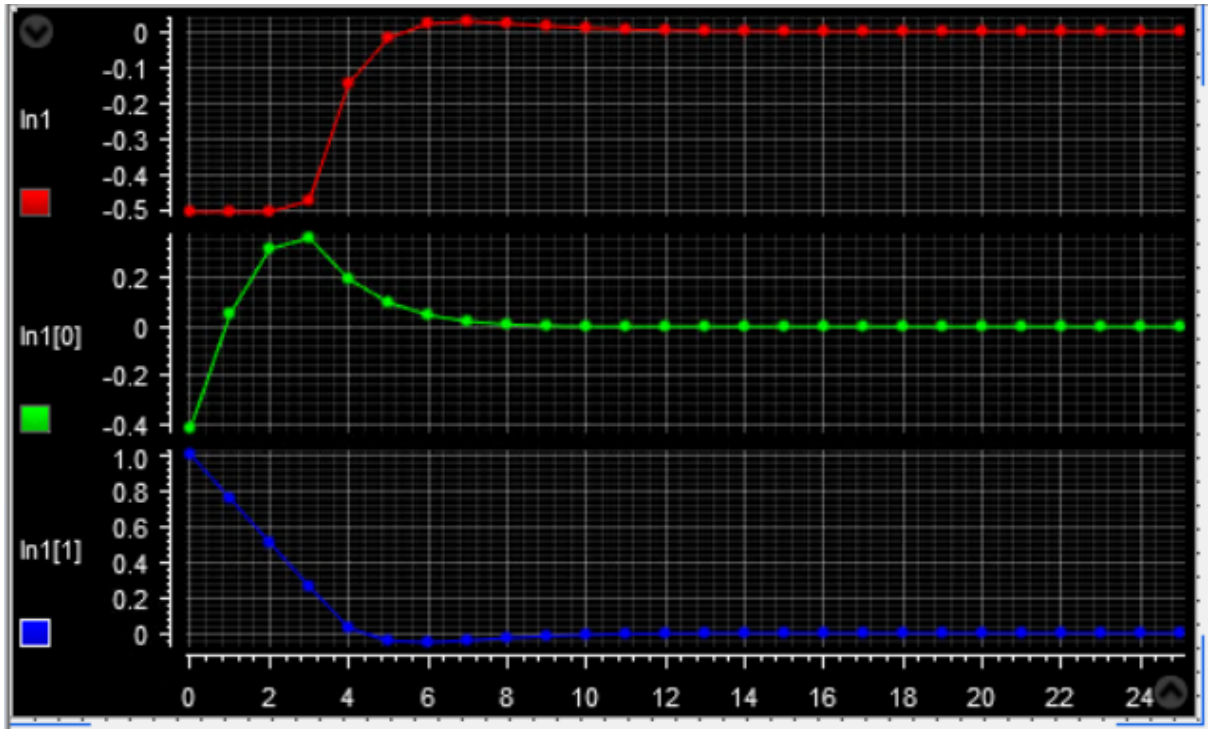


Figure 10.59: Plots and results from experiment on dSPACE MicroAutoBox III.

10.4 Speedgoat

10.4.1 High-level interface

The steps to deploy and simulate a FORCES PRO controller on a Speedgoat platform are detailed below.

1. (Figure 10.60) 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 10.61) Create a new Simulink model using the blank model template.
3. (Figure 10.62) Populate the Simulink model with the system you want to control.
4. (Figure 10.63) Make sure the `FORCESNLPsolver_simulinkBlock.mexw64` file (created during code generation) is on the Matlab path.
5. (Figure 10.64) Open the `FORCESNLPsolver_lib.mdl` Simulink model file, contained in the `interface` folder of the `FORCESNLPsolver` folder created during code generation.
6. (Figure 10.65) Copy-paste the FORCES PRO Simulink block into your simulation model and connect its inputs and outputs appropriately.
7. (Figure 10.66) Access the Simulink model's options.
8. (Figure 10.67) 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 10.68) 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 10.69) In the "Code Generation/Custom Code" tab, include the directories:
    - `BasicExample`
    - `BasicExample\FORCESNLPsolver\interface`
    - `BasicExample\FORCESNLPsolver\lib_target`
11. (Figure 10.70) In the "Code Generation/Custom Code" tab, add the source files:
    - `FORCESNLPsolver_simulinkBlock.c`
    - `FORCESNLPsolver_casadi2forces.c`
    - `FORCESNLPsolver_model_1.c`
    - `FORCESNLPsolver_model_11.c`
12. (Figure 10.71) In the "Code Generation/Custom Code" tab, add the library files:
    - `FORCESNLPsolver.lib`

13. (Figure 10.72) Access the FORCES PRO block's parameters.
14. (Figure 10.73) Remove "FORCESNLPsolver" and "FORCESNLPsolver_simulinkBlock" from the S-function module.
15. (Figure 10.74) Compile the code of the Simulink model. This will also automatically load the model to the connected Speedgoat platform.
16. 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.

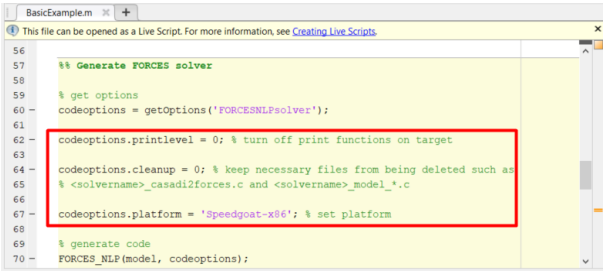


Figure 10.60: Set the appropriate code generation options.

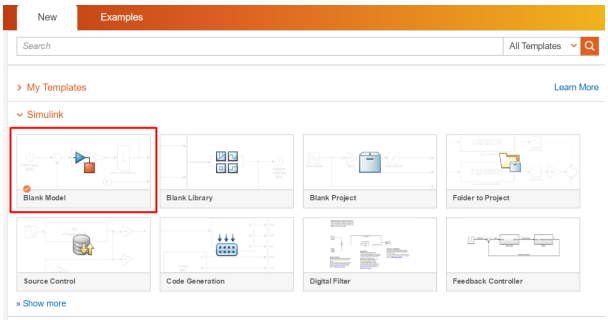


Figure 10.61: Create a Simulink model.

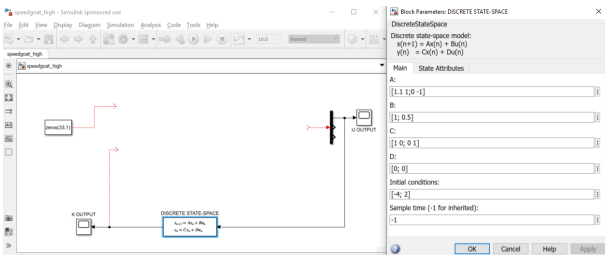


Figure 10.62: Populate the Simulink model.

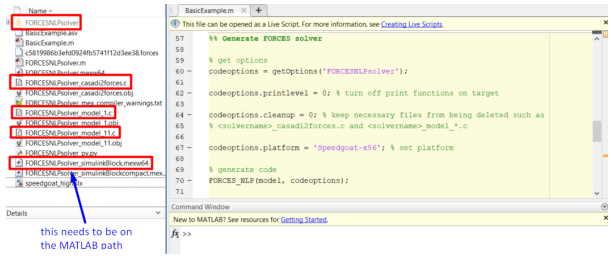


Figure 10.63: Add the folder containing the `.mexw64` solver file to the Matlab path.

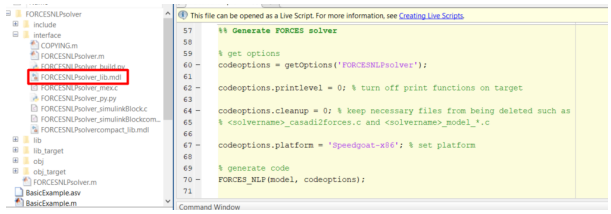


Figure 10.64: Open the generated Simulink solver model.

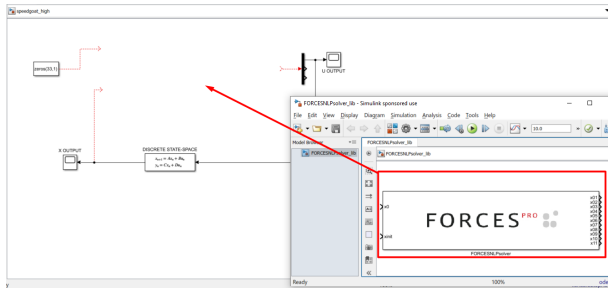


Figure 10.65: Copy-paste and connect the FORCES PRO block.

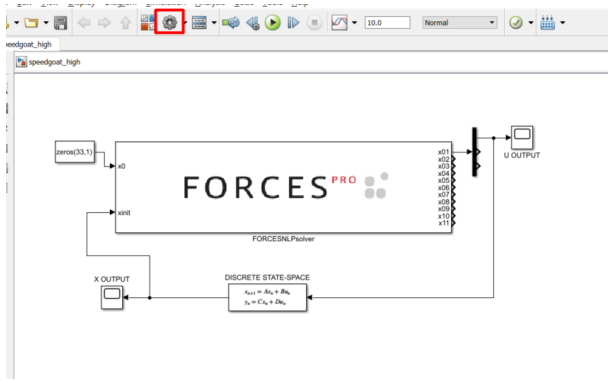


Figure 10.66: Open the Simulink model options.

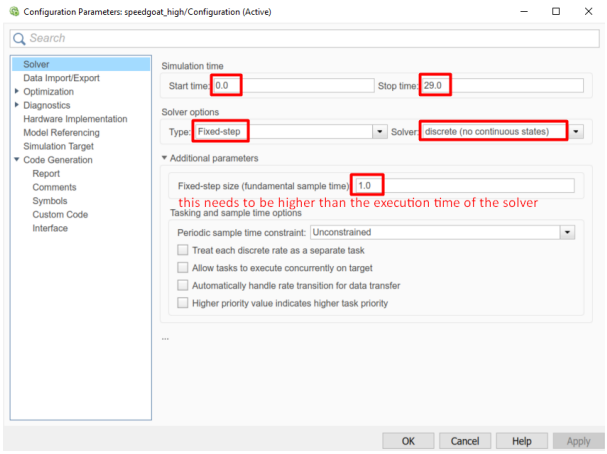


Figure 10.67: Set the Simulink solver options.

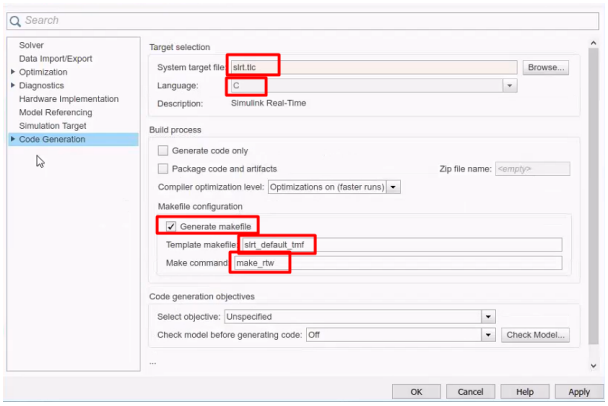


Figure 10.68: Set the Simulink code generation options.

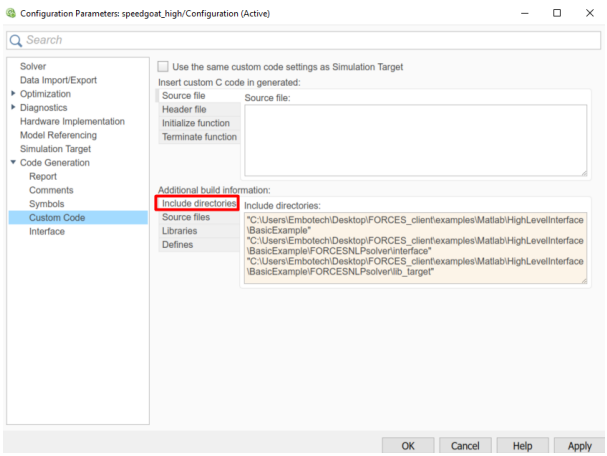


Figure 10.69: Add the directories included for the code generation.

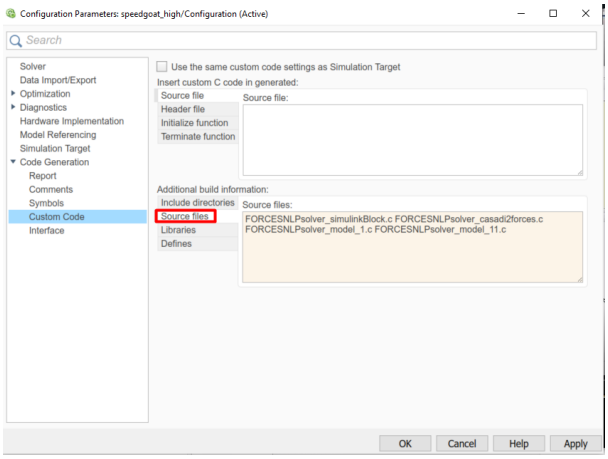


Figure 10.70: Add the source files used for the code generation.

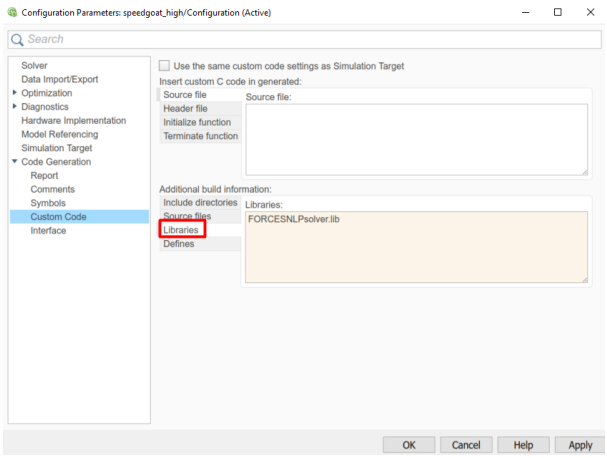


Figure 10.71: Add the libraries used for the code generation.

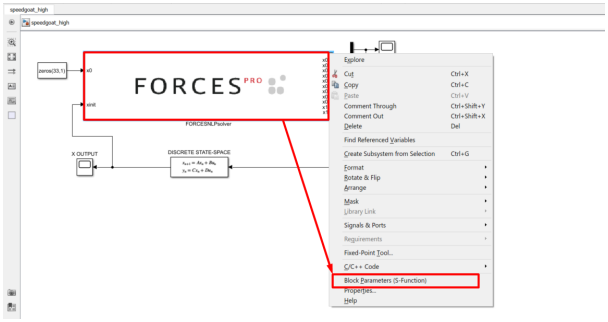


Figure 10.72: Open the FORCES PRO block's parameters.

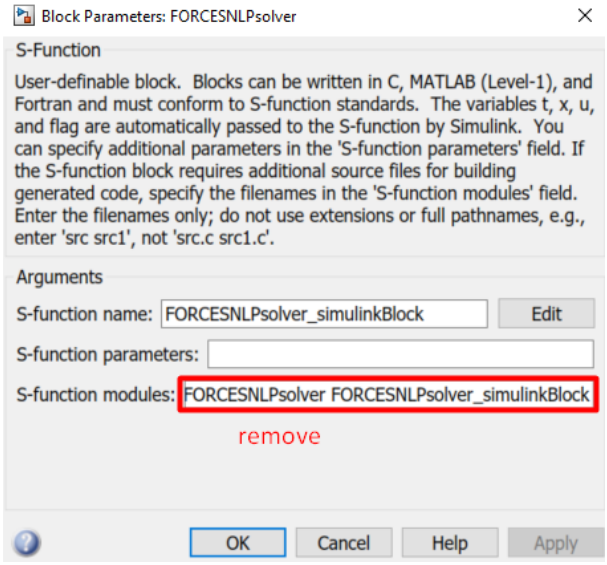


Figure 10.73: Remove the default data from the S-function module.

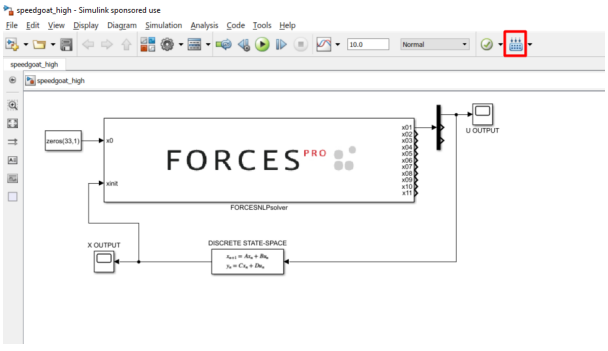


Figure 10.74: Compile the code of the Simulink model.

10.4.2 Y2F interface

The steps to deploy and simulate a FORCES PRO controller on a Speedgoat platform are detailed below.

1. (Figure 10.75) 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 10.76) Create a new Simulink model using the blank model template.
3. (Figure 10.77) Populate the Simulink model with the system you want to control.
4. (Figure 10.78) Make sure the `simplempc_solver_simulinkBlock.mexw64` file (created during code generation) is on the Matlab path.
5. (Figure 10.79) Copy-paste the FORCES PRO 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 10.80) Access the Simulink model's options.
7. (Figure 10.81) In the "Solver" tab, set the options:
   - Simulation start/stop time: Depending on the simulation wanted.
   - Solver type: Discrete or fixed-step (variable-step solvers are not supported by the MicroAutoBox).
   - Fixed-step size: Needs to be higher than the execution time of the solver.
8. (Figure 10.82) 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 10.83) In the "Code Generation/Custom Code" tab, include the directories:
   - `Y2F\simplempc_solver\interface`
   - `Y2F\simplempc_solver\lib_target`
10. (Figure 10.84) In the "Code Generation/Custom Code" tab, add the source files:
    - `simplempc_solver_simulinkBlock.c`
    - `simplempc_solver.c`
11. (Figure 10.85) In the "Code Generation/Custom Code" tab, add the library files:
    - `internal_simplempc_solver_1.lib`
12. (Figure 10.86) Compile the code of the Simulink model. This will also automatically load the model to the connected Speedgoat platform.
13. 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.

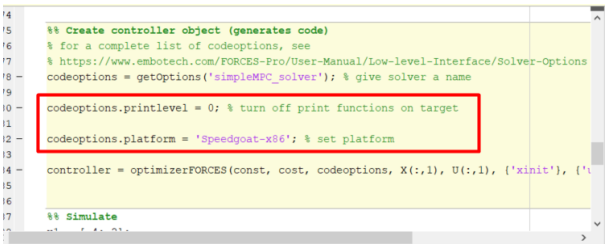


Figure 10.75: Set the appropriate code generation options.

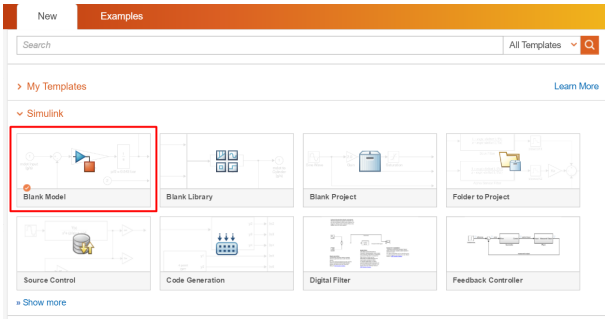


Figure 10.76: Create a Simulink model.

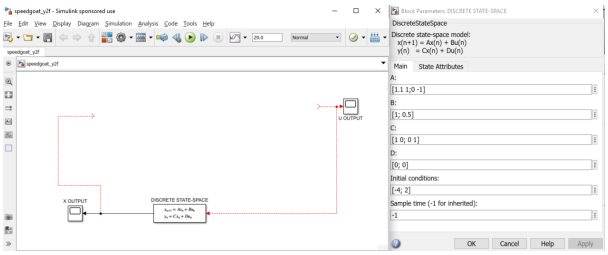


Figure 10.77: Populate the Simulink model.

Figure 10.78: Add the folder containing the `.mexw64` solver file to the Matlab path.

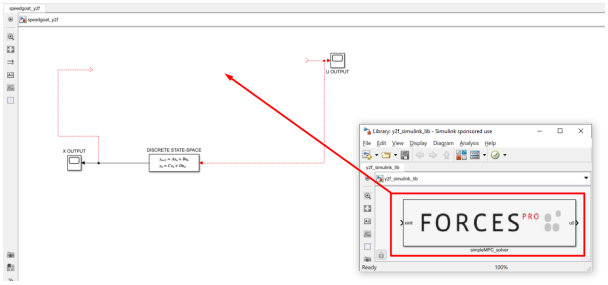


Figure 10.79: Copy-paste and connect the FORCES PRO block.

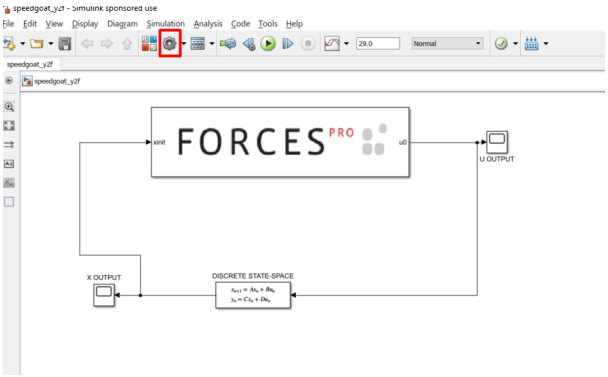


Figure 10.80: Open the Simulink model options.

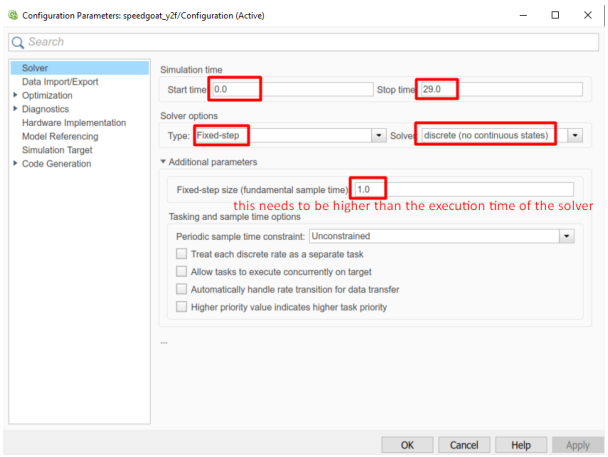


Figure 10.81: Set the Simulink solver options.

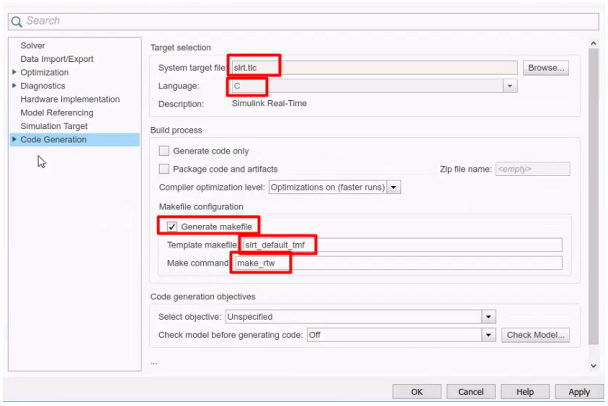


Figure 10.82: Set the Simulink code generation options.

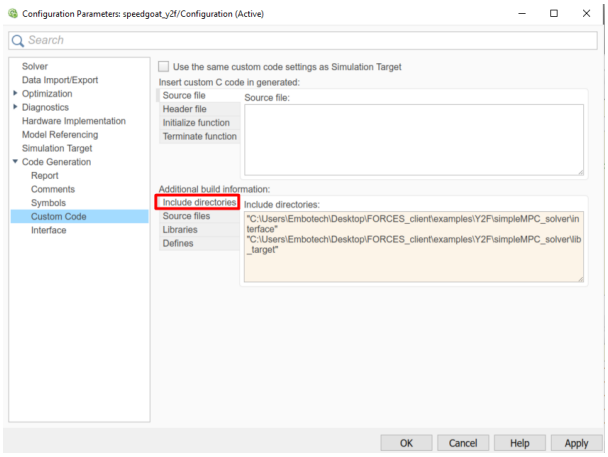


Figure 10.83: Add the directories included for the code generation.

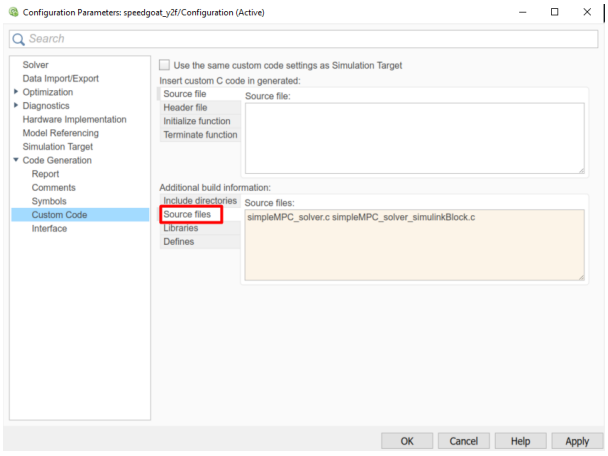


Figure 10.84: Add the source files used for the code generation.

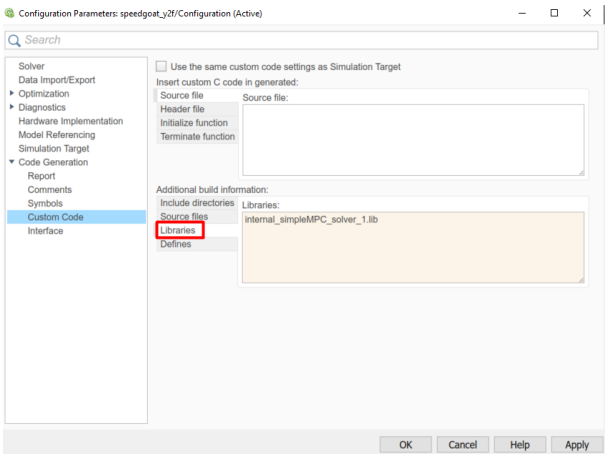


Figure 10.85: Add the libraries used for the code generation.

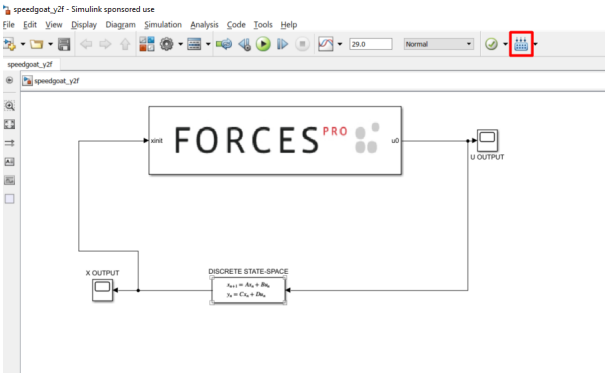


Figure 10.86: Compile the code of the Simulink model.

Chapter 11

Licensing

11.1 Machine Identification

The FORCES PRO 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

11.1.1 Client Identification

Machines running FORCES PRO clients are licensed using the machine's username and the machine's unique identifier.

11.1.2 Solver Identification

Machines running FORCES PRO solvers are licensed using the machine's unique identifier.

11.2 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.

11.2.1 System Requirements

The requirement for static license checking is to have correct system clock settings (accurately showing current time, compliant to UTC time).

11.2.2 Generating solvers

Static license checking is automatically enabled on a generated solver.

11.2.3 Running solvers

After generating a solver, you can move it to the running platform and build it with the rest of your project.

11.3 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).

11.3.1 System Requirements

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 FORCES PRO

11.3.2 Generating solvers

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
codeoptions.license_file_name = '<filename_without_extension>'; % no paths, only␣
↪filename
```

Important: The license file name must be a valid variable name

11.3.3 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: `FORCES_PRO`
- `server`: FORCES PRO 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.

11.3.4 Running solvers

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 need to be in the same folder as the executable of your project

11.4 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 in a machine.

11.4.1 System Requirements

The requirements for enabling solvers with floating licenses are:

- A `x86/x86_64` Linux platform
- An internet connection on the running platform
- Correct system clock settings (accurately showing current time, compliant to UTC time)

11.4.2 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 FORCES PRO 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.

11.4.3 Generating solvers

To enable floating licenses on a generated solver use the following codeoption:

```
% Matlab
codeoptions.useFloatingLicense = 1;
```

```
# Python
codeoptions["useFloatingLicense"] = 1
```

And select the platform to use

```
% 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**

11.4.4 Running solvers

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 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) this will be the one returned to the solver instead of granting a new one
- 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

Chapter 12

Solver Options

The default solver options can be loaded when giving a name to the solver with the following command

```
codeoptions = getOptions('solvername');
```

In the documentation below, we assume that you have created this struct and named it `codeoptions`.

12.1 General options

We will first discuss how to change several options that are valid for all the FORCES PRO interfaces.

12.1.1 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 FORCES within the same software project or Simulink model. To set the name of the solver use:

```
codeoptions.name = 'solvername';
```

Alternatively, you can directly name the solver when generating the options struct by calling:

```
codeoptions = getOptions('solvername');
```

12.1.2 Print level

To control the amount of information the generated solver prints to the console, set the field `printlevel` as outlined in Table 12.1.

Table 12.1: Print level options

`printlevel`	Result	Dependency
`0`	No output will be written.	(None)
`1`	Summary line after each solve.	`<stdio.h>`
`2` (default)	Summary after each iteration of solver.	`<stdio.h>`

Note: For `printlevel=0`, the generated solver has no dependency on any system library. Otherwise, there will be a dependency on `<stdio.h>`.

Important: `printlevel` should always be set to `0` when recording performance timings or when deploying the code on an autonomous embedded system.

12.1.3 Maximum number of iterations

To set the maximum number of iterations of the generated solver, use:

```
codeoptions.maxit = 200;
```

The default maximum number of iterations for all solvers provided by FORCES PRO is set to `200`.

12.1.4 Compiler optimization level

The compiler optimization level can be varied by changing the field `optlevel` from `0` to `3` (default):

```
codeoptions.optlevel = 0;
```

Important: It is recommended to set `optlevel` to `0` 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.

12.1.5 Running solvers in parallel

The generated solver can be run in parallel on different threads by changing the field `threadSafeStorage` from `false` to `true`:

```
codeoptions.threadSafeStorage = true;
```

12.1.6 Measure Computation time

You can measure the time used for executing the generated code by using:

```
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:

```
codeoptions.embedded_timing = 1;
```

12.1.7 Datatypes

The type of variables can be changed by setting the field `floattype` as outlined in Table 12.2.

Table 12.2: Data type options

`floattype`	Decimation	Width (bits)	Supported algorithms
`'double'` (default)	64 bit	Floating point	PDIP, PDIP_NLP, ADMM, DFG, FG
`'float'`	32 bit	Floating point	PDIP, PDIP_NLP, ADMM, DFG, FG
`'int'`	32 bit	Fixed point	PDIP, PDIP_NLP, ADMM, DFG, FG
`'short'`	16 bit	Fixed point	PDIP, PDIP_NLP, ADMM, DFG, FG

Important: Unless running on a resource-constrained platform, we recommend using double precision floating point arithmetics 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`).

12.1.8 Overwriting existing solvers

When a new solver is generated with the same name as an existing solver one can control the overwriting behaviour by setting the field `overwrite` as outlined in Table 12.3.

Table 12.3: Overwrite existing solver options

`overwrite`	Result
`0`	Never overwrite.
`1`	Always overwrite.
`2` (default)	Ask to overwrite.

12.1.9 Solver info in Simulink block

FORCES PRO 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:

```
codeoptions.showinfo = 1;
```

By default these ports are not present in the Simulink block.

12.1.10 Code generation server

By default, code generation requests are routed to embotech's server. To send a code generation request to a local server, for example when FORCES PRO is used in an enterprise setting, set the following field to an appropriate value:

```
codeoptions.server = 'http://embotech-server2.com:8114/v1.5.beta';
```

12.1.11 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:

```
codeoptions.BuildSimulinkBlock = 0;
```

12.1.12 Skipping automatic cleanup

FORCES PRO 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 FORCES, set the option:

```
codeoptions.cleanup = 0;
```

The default value is `1` (true).

Important: The library or object files generated by FORCES PRO 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.

12.1.13 Target platform

As a default option, FORCES PRO 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 FORCES PRO are given in Table 12.4. 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 12.4: Target platforms supported by FORCES PRO

`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'`
`'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'`

Continued on next page

Table 12.4 – continued from previous page

platform	Description	Portal Selection
'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-ARM-x86'	For ARM Cortex 32-bit processors using the Integrity toolchain.	'Integrity-ARM-x86'
'Integrity-ARM-x64'	For ARM Cortex 64-bit processors using the Integrity toolchain.	'Integrity-ARM-x64'
'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'
'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'

Continued on next page

Table 12.4 – continued from previous page

platform	Description	Portal Selection
'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-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-MABII'	For the dSPACE MicroAutoBox II real-time system (supports Microtec compiler).	'dSPACE-MABII-Microtec'
'dSPACE-MABIII'	For the dSPACE MicroAutoBox III real-time system (supports Gcc compiler).	'dSPACE-MABIII-Gcc'
'dSPACE-MABXII'	For the dSPACE MicroAutoBox II real-time system (supports Microtec compiler).	'dSPACE-MABII-Microtec'
'dSPACE-MABXIII'	For the dSPACE MicroAutoBox III real-time system (supports Gcc compiler).	'dSPACE-MABIII-Gcc'
'Speedgoat-x86'	For Speedgoat 32-bit real-time platforms (supports Microsoft compiler).	'Speedgoat-x86'
'Speedgoat-x64'	For Speedgoat 64-bit real-time platforms (supports Microsoft compiler).	'Speedgoat-x64'
'IAtomE680_Bachmann'	For Bachmann PLC platforms (supports VxWorks compiler).	'IAtomE680-VxWorks'

Note: If a solver for another platform is requested, FORCES PRO will still provide the simulation interfaces for the 'Generic' host platform to enable users to run simulations.

Cross compilation

To generate code for other operating systems different from the host platform, set the appropriate flag from the following list to `1`:

```
codeoptions.win
codeoptions.mac
codeoptions.gnu
```

Note that this will only affect the target platform. Interfaces for the host platform will be automatically built.

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:

```
codeoptions.maccompiler
```

SIMD instructions

On x86-based host platforms, one can enable the `sse` field to accelerate the execution of the solver

```
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`,

```
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:

```
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:

```
codeoptions.avx = 0;
```

On 'NVIDIA-Cortex-A57', '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 behaviour 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

```
codeoptions.floattype = 'float'
codeoptions.neon = 1;
```

For double precision, the options are

```
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

```
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:

```
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.

12.1.14 MISRA 2012 compliance

If your license allows it, add the following field to generate C code that is compliant with the MISRA 2012 rules:

```
codeoptions.misra2012_check = 1;
```

This option makes the generated solver code MISRA compliant. After compilation, the client also downloads a folder whose name terminates with `_misra2012_analysis`. The folder contains one summary of all MISRA violations for the solver source and header files. Note that the option only produces MISRA compliant code when used with algorithms `PDIP` and `PDIP_NLP`.

12.1.15 Optimizing code size

The size of the solver libraries generated with code option `PDIP_NLP` can be reduced by means of the option `nlp.compact_code`. By setting

```
codeoptions.nlp.compact_code = 1;
```

the user enables the FORCES PRO 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.

The size of sparse linear algebra routines in the generated code can be reduced by changing the option `compactSparse` from `0` to `1`:

```
codeoptions.compactSparse = 1;
```

12.1.16 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.

```
codeoptions.optimize_choleskydivision = 1;
codeoptions.optimize_registers = 1;
codeoptions.optimize_uselocalsall = 1;
codeoptions.optimize_uselocalsheavy = 1; % overriden if uselocalsall is enabled
codeoptions.optimize_uselocalssimple = 1; % overriden if uselocalsheavy is enabled
codeoptions.optimize_operationsrearrange = 1;
codeoptions.optimize_loopunrolling = 1;
codeoptions.optimize_enableoffset = 1;
```

12.1.17 Dump problem formulation

The MATLAB client of FORCES PRO 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 13 and can be turned on by using the setting:

```
codeoptions.dump_formulation = 1;
```

12.2 High-level interface options

The FORCES PRO 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.

12.2.1 Integrators

When providing the continuous dynamics the user must select a particular integrator by setting `nlp.integrator.type` as outlined in Table 12.5.

Table 12.5: 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 Euler Method	2
'IRK4'	Implicit Euler Method	4

The user must also provide the discretization interval (in seconds) and the number of intermediate shooting nodes per interval. For instance:

```
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 to work.

12.2.2 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:

```
% 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 infinitiy norm $\|\cdot\|_\infty$.

12.2.3 Barrier strategy

The strategy for updating the barrier parameter is set using the field:

```
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.

12.2.4 Hessian approximation

The way the Hessian of the Lagrangian function is computed can be set using the field:

```
codeoptions.nlp.hessian_approximation = 'bfgs';
```

FORCES PRO 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.:

```
codeoptions.nlp.bfgs_init = diag([0.1, 10, 4]);
```

Note that BFGS updates are carried out individually per stage in the FORCES 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.

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:

```
codeoptions.nlp.max_update_skip = 2;
```

The default value for `max_update_skip` is 2.

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. $f(z) = 50 z_1^2 + 3 z_2^2$, then the following code defines the least-squares objective (note that r is a vector-valued function):

```
nlp.objective = @(z) 0.1* z(1)^2 + 0.01*z(2)^2;
nlp.LSobjective = @(z) [sqrt(0.2)*z(1); sqrt (0.02)*z(2)];
```

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.

12.2.5 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:

```
 % default fraction-to-boundary scaling
codeoptions.nlp.ftbr_scaling = 0.9900;
```

12.2.6 Regularization

To avoid ill-conditioned saddle point systems, FORCES 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. FORCES 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:

```
% 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;
```

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:

```
% 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.)
```

12.2.7 Linear system solver

The interior-point method solves a linear system to find a search direction at every iteration. FORCES NLP offers the following three linear solvers:

- `'normal_eqs'` (default): Solving the KKT system in normal equations form.
- `'symm_indefinite_fast'`: Solving the KKT system in augmented / symmetric indefinite form, using regularization and positive definite Cholesky factorizations only.
- `'symm_indefinite'`: Solving the KKT system in augmented / symmetric indefinite form, using block-indefinite factorizations.

The linear system solver can be selected by setting the following field:

```
codeoptions.nlp.linear_solver = 'symm_indefinite';
```

It is recommended to try different linear solvers when experiencing convergence problems. The most stable method is `'symm_indefinite'`, while the fastest solver is `'symm_indefinite_fast'`.

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.

The `'normal_eqs'` solver is the standard FORCES 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 the most robust solver, but still high-speed. It is based on block-wise factorization of the symmetric indefinite form of the KKT system (the so-called augmented form). Each block is handled by symmetric indefinite LDL factorization, with (modified) on-the-fly Bunch-Kaufmann permutations leading to boundedness of lower triangular factors for highest numerical stability. This is our most robust linear system solver, with only a modest performance penalty (about 30% compared to `'symm_indefinite_fast'`).

The `'symm_indefinite_fast'` solver is robust, but even faster. It is based on block-wise factorization of the symmetric indefinite KKT matrix, where each block is handled by a Cholesky factorization. It uses regularization to increase numerical stability. Currently only used for receding-horizon/MPC-like problems where dimensions of all stages are equal (minus the first and last stage, those are handled separately). It is more robust and faster than the normal equations form. This solver is likely to become the default option in the future.

12.2.8 Automatic differentiation tool

If external functions and derivatives are not provided directly as C code by the user, FORCES PRO 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, three different AD tools are supported that can be chosen by means of the setting `nlp.ad_tool` as summarized in Table 12.6.

Table 12.6: Automatic differentiation tool options

nlp.ad_tool	Tool	URL
'casadi'	CasADi v2.4.2	CasADi
'casadi-351'	CasADi v3.5.1	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

```
codeoptions.nlp.ad_tool = 'casadi';
```

Also note that the use of implicit integrators `BackwardEuler`, `IRK2` and `IRK4` (see Section 12.2.1) currently still rely on using the CasADi AD tool.

12.2.9 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:

```
codeoptions.nlp.checkFunctions = 0;
```

12.3 Convex branch-and-bound options

The settings of the FORCES PRO 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 12.7, 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 FORCES PRO nonlinear solver. For mixed-integer nonlinear programs and the available options, please have a look at paragraph *Mixed-integer nonlinear solver*.

Table 12.7: 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.

12.4 Solve methods

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 12.8.

Table 12.8: Solve methods

solvemethod	Method	Description
'PDIP' (default)	Primal-Dual Interior-Point Method	The Primal-Dual Interior-Point Method is a stable and robust method for most problems.
'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.

12.4.1 Primal-Dual Interior-Point Method

The Primal-Dual Interior-Point Method is the default optimization method. It is a stable and robust method for most of the problems.

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 FORCES PRO set the field `init` to one of the values in Table 12.9.

Table 12.9: 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 $\mu_0 : z_i = 0, s_i = b_{\text{ineq}}$ 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:

```
codeoptions.mu0 = 10;
```

Accuracy Requirements

The accuracy for which FORCES PRO returns the OPTIMAL flag can be set as follows:

```
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 FORCES PRO 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.

```
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.

```
codeoptions.linesearch.minstep = 1e-8;
codeoptions.linesearch.maxstep = 0.995;
```

Regularization

During factorization of supposedly positive definite matrices, FORCES PRO 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 FORCES PRO allows to occur.

```
codeoptions.regularize.epsilon = 1e-13; % if pivot element < epsilon ...
codeoptions.regularize.delta = 1e-8;    % then set it to delta
```

Multicore parallelization

FORCES PRO 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

```
codeoptions.parallel = 1;
```

By default multicore computation is switched off.

12.4.2 Alternating Directions Method of Multipliers

FORCES PRO implements several optimization methods based on the ADMM framework. Different variants can handle different types of constraints and FORCES PRO will automatically choose an ADMM variant that can handle the constraints in a given problem. To manually choose a specific method in FORCES PRO, use the `ADMMvariant` field of `codeoptions`:

```
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 for which FORCES PRO returns the OPTIMAL flag can be set as follows:

```
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 achive 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:

```
codeoptions.ADMMrho = 1;
```

In some cases it may be possible to let FORCES PRO choose the value ρ automatically. To enable this feature set:

```
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:

```
codeoptions.ADMMalpha = 1;
```

Precomputations

For problems with time-invariant data, FORCES PRO can compute full matrix inverses at code generation time and then implement matrix solves online by dense matrix-vector multiplication. In some cases, especially when the prediction horizon is long, it may be better to factorize the matrix and implement matrix solves using forward and backward solves with the pre-computed factors. To manually switch on this option, use the `ADMMfactorize` field of `codeoptions`.

When the data is time-varying, or when the prediction horizon is larger than 15 steps, FORCES PRO automatically switches to a factorization-based method.

```
codeoptions.ADMMfactorize = 0;
```

12.4.3 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.

12.4.4 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 for which FORCES PRO returns the OPTIMAL flag can be set as follows:

```
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:

```
codeoptions.FGstep = 1/1000;
```

In some cases it may be possible to let FORCES PRO choose the step size automatically. To enable this feature set:

```
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:

```
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.

Chapter 13

Dumping Problem Formulation and Data

13.1 Why to use the dump tool?

Along with its MATLAB client, FORCES PRO 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). Problem formulation and data stored in "stand-alone" `mat` files, i.e. there is no need to keep copies of other files that may be used to specify the formulation (such as the dynamic equations).

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 the dump tool does not merely save your MATLAB structs into a file. Those structs may contain MATLAB function handles referencing external functions. Instead, the dumped formulation already contains *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.

13.2 How to use the dump tool?

Dumping a problem consist 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.

13.2.1 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 = ForcesDumpFormulation( formulation,codeoptions,outputs );
```

The third argument `outputs` is optional. The function `ForcesDumpFormulation` will create a `mat` file in the directory from where it is called containing the passed information. The filename is automatically chosen and will contain the name of your solver, a unique tag, a timestamp as well as the suffix `_F`, e.g. `myFORCESsolver_ABC3DEFGHIJ_20200101120000000_F.mat`.

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 also dumping corresponding problem data.

13.2.2 Dumping problem data

Assuming your generated FORCES PRO 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

```
ForcesDumpProblem( problem,tag );
```

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 function `ForcesDumpProblem` will create a `mat` file in the directory from where it is called containing the passed information. The filename is automatically chosen and will contain the name of your solver, a unique tag, a timestamp as well as the suffix `_P`, e.g. `myFORCESsolver_ABC3DEFGHIJ_20200101120001000_P.mat`.

You may dump as many problem instances as you have disk space available.

13.2.3 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 FORCES PRO 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.

13.3 Limitations

Currently, the dump tool has the following limitations:

- It is only provided for the MATLAB client of FORCES PRO.
- It cannot be used if you pass external functions in form of C code.

We aim at overcoming these limitations in a future release.

Chapter 14

Frequently asked questions

14.1 Quick links

Features of FORCES PRO
Issues during code generation
Issues when running the solver
Simulink interface
Code deployment
Other topics

14.2 Features of FORCES PRO

- **I have been using FORCES in the past. Why should I use FORCES PRO?**

The development of the free version of FORCES by ETH (forces.ethz.ch) has been discontinued, and the code generation service is no longer available.

The professional version of FORCES PRO comes with professional support, additional interfaces, and a large performance increase.

- **Can FORCES PRO target dSpace hardware?**

Yes, FORCES PRO can be seamlessly integrated in the dSpace design flow with the new Simulink interface. For more details see *dSPACE MicroAutoBox II* and *dSPACE MicroAutoBox III*.

- **Can I use FORCES for non-multistage programs?**

Yes, FORCES PRO 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 *stages=MultistageProblem(1)* and fill in the matrices as described in *Low-level interface*.

- **I need to re-linearize the model of my plant each sampling time. Does FORCES PRO support this?**

When re-linearizing non-linear dynamics, you obtain in each sampling time a different matrix A, B and also a new affine part g:

$$x_{k+1} = Ax_k + Bu_k + g$$

FORCES PRO supports changing these variables at run-time by defining them as parameters.

- **I don't have a state-space model of my system. Can I still use FORCES PRO to design an optimal controller?**

Yes, the graphical interface allows one to design optimal controllers for models described by a Simulink diagram - there is no need for equations. If you have a model in another form, please send us a feature request and we will try to support your model type as soon as possible.

14.3 Issues during code generation

- **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 FORCES PRO (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 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 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.

- **I get the following error message when using the Python interface: 'csmatio.io.MatlabIOException: Incorrect Matlab array class: int32'**

Make sure that the parametric data is passed to the solver as numpy arrays of floating point numbers, i.e. instead of

```
problem['Q'] = np.array([1 1])
```

use

```
problem['Q'] = np.array([1.0 1.0])
```

- **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 will all optimizations turned on (*-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 changing the compiler optimization flags to *-O0*, *-O1*, or *-O2*. You can change this setting using the following flag set to the appropriate value.

```
codeoptions.optlevel = 2;
```

14.4 Issues when running the solver

- **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 some 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

```
codeoptions.printlevel = 0;
```

- **My solver is producing a segmentation fault.**

When the solver has a large amount of parameters or the problem is relatively large, compiling with *codeoptions.optlevel = 0;* can produce a segmentation fault. Please try to increase the value of *codeoptions.optlevel* or submit a bug report to support@embotech.com.

- **ADMM does not converge for my problem.**

Unlike interior-point methods, the convergence of ADMM depends 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.

- **The solver outputs exitcode -7.**

Exitcode -7 means that the solver could not proceed. A common cause is the problem being infeasible. FORCES PRO does not have infeasibility detection to speed up the solution time. However, one can use the function *stages2qcqp* to convert the FORCES problem into a standard (QC)QP that can be given to standard QP solvers like quadprog, MOSEK or CPLEX to check for infeasibility.

- **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 will 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 this setting using the following flag set to the appropriate value.

```
codeoptions.optlevel = 2;
```

14.5 Simulink interface

- **When I have a long prediction horizon I have too 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.**

The new version of FORCES PRO 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.

14.6 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 MicroAutoBox II*.

14.7 Other topics

- **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 FORCES PRO?**

  When contacting the solver generation server, the FORCES PRO 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:
    - Cputype and Cpusubtype
    - 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.

- **Why am I being asked to update the FORCES PRO 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 substantial change in the code. To make this process easier and faster, FORCES PRO comes with a functionality that allows users to update their clients by simply typing the following in the MATLAB command prompt:

```
>> updateClient
```

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

[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.