Contents

1 Introduction .......................... 1
  1.1 Troubleshooting and support .................................. 1
  1.2 Licensing .................................................. 2
    1.2.1 Industrial licensing .................................... 2
    1.2.2 Academic licensing ..................................... 2
  1.3 Citing FORCES Pro ........................................... 2
  1.4 Release Notes ............................................... 2
    1.4.1 New features in FORCES Pro 1.8.0 ......................... 3
    1.4.2 Improvements in FORCES Pro 1.8.0 ....................... 3
    1.4.3 Bug Fixes in FORCES Pro 1.8.0 ......................... 3
    1.4.4 New features in FORCES Pro 1.7.0 ......................... 3
    1.4.5 Improvements in FORCES Pro 1.7.0 ....................... 3
    1.4.6 Bug Fixes in FORCES Pro 1.7.0 ......................... 4
  1.5 Version history of manual ................................... 4

2 Installation .......................... 5
  2.1 Obtaining FORCES Pro ................................. 5
  2.2 Installing FORCES Pro .................................. 5
  2.3 System requirements ................................. 5
  2.4 Keeping FORCES Pro up to date ......................... 6

3 Y2F Interface .......................... 7
  3.1 Installing Y2F ........................................... 7
  3.2 Generating a solver ..................................... 7
  3.3 Calling the solver ...................................... 8
  3.4 Solver info ............................................. 8
    3.4.1 Exitflags .......................................... 8
    3.4.2 Additional diagnostics ................................ 9
  3.5 Performance ............................................. 9
  3.6 Examples ............................................... 11

4 Low-level Interface .................. 13
  4.1 Supported problem class ............................ 13
  4.2 Expressing the optimization problem in MATLAB or Python .................................. 13
  4.3 Multistage struct ...................................... 14
  4.4 Dimensions ............................................. 14
  4.5 Cost function ........................................... 14
  4.6 Equality constraints .................................. 15
  4.7 Lower and upper bounds ............................ 15
  4.8 Polytopic constraints .................................. 15
  4.9 Quadratic constraints ............................... 15
    4.9.1 Example ........................................ 16
  4.10 Binary constraints ................................... 16

5 High-level Interface ................. 17
  5.1 Supported problems ............................... 17
    5.1.1 Canonical problem for discrete-time dynamics .......... 17
### 5.4.1 Initial guess
- Page 23
- Details on specifying initial conditions.

### 5.4.2 Initial and final conditions
- Page 24
- Instructions for defining initial and final conditions.

### 5.4.3 Real-time parameters
- Page 24
- Guidance on incorporating real-time parameters.

### 5.4.4 Exitflags and quality of the result
- Page 24
- Explanation of exit flags and result quality assessment.

### 5.5 External function evaluations in C
- Page 25
- Overview of interfacing with C functions.

#### 5.5.1 Interface
- Page 25
- Interface details for C function evaluations.

#### 5.5.2 Suppling function evaluation information
- Page 26
- Methods for providing function evaluation data.

#### 5.5.3 Rules for function evaluation code
- Page 26
- Guidelines for coding function evaluations.

#### 5.5.4 Matrix format
- Page 27
- Format specifications for matrix inputs.

#### 5.5.5 Multiple source files
- Page 27
- Information on using multiple C source files.

#### 5.5.6 Stage-dependent functions
- Page 27
- Description of stage-dependent function evaluations.

### 5.6 Mixed-integer nonlinear solver
- Page 28
- Introduction to the mixed-integer solver.

#### 5.6.1 Writing a mixed-integer model
- Page 28
- Guide for constructing mixed-integer models.

#### 5.6.2 Mixed-integer solver customization via user callbacks
- Page 29
- Techniques for customizing the mixed-integer solver.

### 5.7 Examples
- Page 30
- Case studies and examples.

### 6 Examples

#### 6.1 Y2F interface: Basic example
- Page 31
- Basic example for Y2F interface.

##### 6.1.1 Defining the problem data
- Page 31
- Steps for defining problem data.

##### 6.1.2 Defining the MPC problem
- Page 32
- Procedures for defining the MPC problem.

##### 6.1.3 Defining the problem data
- Page 32
- Details on defining problem data.

##### 6.1.4 Calling the generated solver
- Page 33
- Procedure for calling the solver.

##### 6.1.5 Simulation
- Page 33
- Simulation methods.

##### 6.1.6 Results
- Page 34
- Results analysis.

##### 6.1.7 Variation 1: Parametric cost
- Page 34
- Variation 1 details.

##### 6.1.8 Variation 2: Time-varying dynamics
- Page 35
- Variations details.

##### 6.1.9 Variation 3: Time-varying constraints
- Page 35
- Variation 3 specifics.

#### 6.2 High-level interface: Basic example
- Page 37
- High-level interface basics.

##### 6.2.1 Defining the problem data
- Page 37
- Defining problem data in high-level interface.

##### 6.2.2 Defining the MPC problem
- Page 37
- Defining the MPC problem in high-level interface.

##### 6.2.3 Generating a solver
- Page 38
- Generating a solver in high-level interface.

##### 6.2.4 Calling the generated solver
- Page 38
- Calling the generated solver.

##### 6.2.5 Simulation
- Page 38
- Simulation protocols.

##### 6.2.6 Results
- Page 39
- Result evaluation.

#### 6.3 High-level interface: Obstacle avoidance
- Page 40
- Obstacle avoidance example.

##### 6.3.1 Defining the problem data
- Page 40
- Defining data for obstacle avoidance.

##### 6.3.2 Matrix equality constraints
- Page 40
- Matrix equality constraint details.

##### 6.3.3 Inequality constraints
- Page 41
- Inequality constraint specifics.
6.3.4 Calling the generated solver .................................................. 42
6.3.5 Results ................................................................................. 43
6.3.6 Variation 1: Parameters ....................................................... 45
6.3.7 Variation 2: Different integrator .......................................... 45
6.3.8 Variation 3: Terminal cost ................................................... 46
6.3.9 Variation 4: External functions ........................................... 46
6.4 High-level interface: Indoor localization ................................... 47
6.4.1 Time of flight measurements .............................................. 47
6.4.2 Estimation error ................................................................. 48
6.4.3 Minimize the error .............................................................. 48
6.4.4 Implementation ................................................................. 48
6.5 Mixed-integer nonlinear solver: F8 Crusader aircraft .................. 50
6.5.1 Defining the problem data .................................................. 50
6.5.2 Defining the MPC problem ................................................ 51
6.5.3 Generating an MINLP solver ............................................. 52
6.5.4 Calling the generated MINLP solver ................................... 53
6.5.5 Results ................................................................................. 53
7 Parametric problems .................................................................. 57
7.1 Defining parameters .............................................................. 57
7.2 Example ............................................................................... 58
7.3 Parametric Quadratic Constraints .......................................... 58
7.4 Diagonal Hessians .................................................................. 59
7.5 Sparse Parameters .................................................................. 59
7.6 Special Parameters .................................................................. 59
7.7 Python: Column vs Row Major Storage Format .......................... 60
8 Code Deployment ...................................................................... 61
8.1 Main Targets .......................................................................... 61
8.1.1 High-level interface .......................................................... 61
8.1.2 Y2F interface ..................................................................... 63
8.2 dSPACE MicroAutoBox II ....................................................... 65
8.2.1 High-level interface .......................................................... 65
8.2.2 Y2F interface ..................................................................... 70
8.3 Speedgoat .............................................................................. 75
8.3.1 High-level interface .......................................................... 75
8.3.2 Y2F interface ..................................................................... 81
9 Solver Options ......................................................................... 87
9.1 General options ....................................................................... 87
9.1.1 Solver name ....................................................................... 87
9.1.2 Print level .......................................................................... 87
9.1.3 Maximum number of iterations ........................................ 88
9.1.4 Compiler optimization level ............................................. 88
9.1.5 Running solvers in parallel .............................................. 88
9.1.6 Measure Computation time .............................................. 88
9.1.7 Datatypes .......................................................................... 89
9.1.8 Overwriting existing solvers ............................................. 89
9.1.9 Solver info in Simulink block .......................................... 89
9.1.10 Code generation server .................................................... 89
9.1.11 Skipping the Build of Simulink S-function ....................... 90
9.1.12 Skipping automatic cleanup .......................................... 90
# FORCES Pro User Manual

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.1.13</td>
<td>90</td>
</tr>
<tr>
<td>9.1.13.1</td>
<td>92</td>
</tr>
<tr>
<td>9.1.13.2</td>
<td>92</td>
</tr>
<tr>
<td>9.1.13.3</td>
<td>92</td>
</tr>
<tr>
<td>9.1.14</td>
<td>92</td>
</tr>
<tr>
<td>9.1.15</td>
<td>92</td>
</tr>
<tr>
<td>9.1.16</td>
<td>93</td>
</tr>
<tr>
<td>9.2</td>
<td>93</td>
</tr>
<tr>
<td>9.2.1</td>
<td>93</td>
</tr>
<tr>
<td>9.2.2</td>
<td>94</td>
</tr>
<tr>
<td>9.2.3</td>
<td>94</td>
</tr>
<tr>
<td>9.2.4</td>
<td>94</td>
</tr>
<tr>
<td>9.2.4.1</td>
<td>94</td>
</tr>
<tr>
<td>9.2.4.2</td>
<td>95</td>
</tr>
<tr>
<td>9.2.5</td>
<td>95</td>
</tr>
<tr>
<td>9.2.6</td>
<td>95</td>
</tr>
<tr>
<td>9.2.6.1</td>
<td>96</td>
</tr>
<tr>
<td>9.2.6.2</td>
<td>96</td>
</tr>
<tr>
<td>9.2.7</td>
<td>96</td>
</tr>
<tr>
<td>9.2.8</td>
<td>97</td>
</tr>
<tr>
<td>9.3</td>
<td>97</td>
</tr>
<tr>
<td>9.4</td>
<td>98</td>
</tr>
<tr>
<td>9.4.1</td>
<td>99</td>
</tr>
<tr>
<td>9.4.1.1</td>
<td>99</td>
</tr>
<tr>
<td>9.4.1.2</td>
<td>100</td>
</tr>
<tr>
<td>9.4.1.3</td>
<td>100</td>
</tr>
<tr>
<td>9.4.1.4</td>
<td>100</td>
</tr>
<tr>
<td>9.4.1.5</td>
<td>101</td>
</tr>
<tr>
<td>9.4.1.6</td>
<td>101</td>
</tr>
<tr>
<td>9.4.2</td>
<td>101</td>
</tr>
<tr>
<td>9.4.2.1</td>
<td>101</td>
</tr>
<tr>
<td>9.4.2.2</td>
<td>102</td>
</tr>
<tr>
<td>9.4.2.3</td>
<td>102</td>
</tr>
<tr>
<td>9.4.3</td>
<td>102</td>
</tr>
<tr>
<td>9.4.4</td>
<td>102</td>
</tr>
<tr>
<td>9.4.4.1</td>
<td>102</td>
</tr>
<tr>
<td>9.4.4.2</td>
<td>103</td>
</tr>
<tr>
<td>9.4.4.3</td>
<td>103</td>
</tr>
<tr>
<td>10</td>
<td>105</td>
</tr>
<tr>
<td>10.1</td>
<td>105</td>
</tr>
<tr>
<td>10.2</td>
<td>105</td>
</tr>
<tr>
<td>10.3</td>
<td>106</td>
</tr>
<tr>
<td>10.4</td>
<td>107</td>
</tr>
<tr>
<td>10.5</td>
<td>108</td>
</tr>
<tr>
<td>10.6</td>
<td>108</td>
</tr>
<tr>
<td>10.7</td>
<td>109</td>
</tr>
</tbody>
</table>

**Bibliography**

v
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.

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 three 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.
- **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:

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

```latex
@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 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

1.4.2 Improvements in FORCES Pro 1.8.0

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

1.4.3 Bug Fixes in FORCES Pro 1.8.0

- Fixed numerical bug in v1.7.0

1.4.4 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)

1.4.5 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
1.4.6 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>
<thead>
<tr>
<th>Version</th>
<th>Revision</th>
<th>Date</th>
<th>Reason for change</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>04/10/2017</td>
<td>Initial version</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>09/27/2018</td>
<td>Overhaul of outdated manual</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>11/19/2018</td>
<td>Add dSPACE code deployment</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>02/20/2019</td>
<td>Updated manual for v1.7.0</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>06/04/2019</td>
<td>Updated manual for v1.8.0</td>
</tr>
</tbody>
</table>
Chapter 2

Installation

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.

2.1 Obtaining FORCES Pro

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, the FORCES Pro client will be sent to you via email. If a FORCES Pro client already exists on your machine, the `updateClient` script will update it to the new version.

2.2 Installing FORCES Pro

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.](image)

2.3 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.

## 2.4 Keeping FORCES Pro up to date

FORCES Pro is actively developed and client modification 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.
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:

```matlab
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:

   ```matlab
   [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):

   ```matlab
   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

<table>
<thead>
<tr>
<th>Exitflag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Optimal solution found to the requested accuracy.</td>
</tr>
<tr>
<td>2</td>
<td>(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.</td>
</tr>
<tr>
<td>0</td>
<td>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.</td>
</tr>
<tr>
<td>-1</td>
<td>(only branch-and-bound) Infeasible problem (problems solving the root relaxation to desired accuracy).</td>
</tr>
<tr>
<td>-2</td>
<td>(only branch-and-bound) Out of memory – cannot fit branch and bound nodes into pre-allocated memory.</td>
</tr>
<tr>
<td>-7</td>
<td>The convex solver could not proceed due to stalled line search. The problem might be infeasible. Otherwise, please submit a bug report to <a href="mailto:support@embotech.com">support@embotech.com</a> 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.</td>
</tr>
<tr>
<td>-10</td>
<td>The convex solver could not proceed due to an internal error. The problem might be infeasible. Otherwise, please submit a bug report to <a href="mailto:support@embotech.com">support@embotech.com</a> 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.</td>
</tr>
<tr>
<td>-100</td>
<td>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.</td>
</tr>
</tbody>
</table>

### 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

<table>
<thead>
<tr>
<th>Info</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>info.it</td>
<td>Number of iterations. In branch-and-bound mode this is the number of convex problems solved in total.</td>
</tr>
<tr>
<td>info.solvetime</td>
<td>Total computation time in seconds.</td>
</tr>
<tr>
<td>info.pobj</td>
<td>Value of the objective function.</td>
</tr>
<tr>
<td>info.it2opt</td>
<td>(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.</td>
</tr>
</tbody>
</table>

### 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

Low-level interface

FORCES Pro supports designing solvers and controllers via MATLAB and Python scripts. Regardless of the language used, 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.

4.1 Supported problem class

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

\[
\begin{align*}
\text{minimize} & \quad \sum_{i=1}^{N} \frac{1}{2}z_i^T H_i z_i + f_i^T z_i \\
\text{subject to} & \quad D_1 z_1 = c_1 \\
& \quad C_i z_{i-1} + D_i z_i = c_i \\
& \quad \bar{z}_i \leq z_i \\
& \quad z_i \leq \bar{z}_i \\
& \quad A_i z_i \leq b_i \\
& \quad z_i^T Q_{i,k} z_i + L_{i,k}^T z_i \leq r_i
\end{align*}
\]

for \(i = 1, \ldots, N\) and \(k = 1, \ldots, M\). 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 \(\bar{z}_i, z_i, b_i, L_{i,k}, r_{i,k}, c_i\), in a MATLAB struct or Python dictionary, along with the corresponding dimensions. The following steps will take you through this process.

Note: FORCES Pro supports all problem data to be parametric, i.e. to be unknown at code generation time. Read Section 7 to learn how to use parameters correctly.

4.2 Expressing the optimization problem in MATLAB or Python

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.
Python users first have to import the FORCES Pro module.

```python
from forcespro import *
```

### 4.3 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

```python
stages = MultistageProblem(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.

### 4.4 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:

```python
stages(i).dims.n = ...; % length of stage variable \( z_i \)
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
```

```python
stages.dims[i]['n'] = ... # length of stage variable \( z_i \)
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
```

### 4.5 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

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

```python
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).
4.6 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:

\[
\begin{align*}
stages(i).eq.C &= \ldots; \\
stages(i).eq.c &= \ldots; \\
stages(i).eq.D &= \ldots; \\
\end{align*}
\]

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

4.7 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 $lb/ub$:

\[
\begin{align*}
stages(i).ineq.b.lbidx &= \ldots; & \% index vector for lower bounds \\
stages(i).ineq.b.lb &= \ldots; & \% lower bounds \\
stages(i).ineq.b.ubidx &= \ldots; & \% index vector for upper bounds \\
stages(i).ineq.b.ub &= \ldots; & \% upper bounds \\
\end{align*}
\]

\[
\begin{align*}
stages.ineq[i][b][lbidx] &= \ldots & \# index vector for lower bounds \\
stages.ineq[i][b][lb] &= \ldots & \# lower bounds \\
stages.ineq[i][b][ubidx] &= \ldots & \# index vector for upper bounds \\
stages.ineq[i][b][ub] &= \ldots & \# upper bounds \\
\end{align*}
\]

4.8 Polytopic constraints

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

\[
\begin{align*}
stages(i).ineq.p.A &= \ldots; & \% Jacobian of linear inequality \\
stages(i).ineq.p.b &= \ldots; & \% RHS of linear inequality \\
\end{align*}
\]

\[
\begin{align*}
stages.ineq[i][a] &= \ldots & \# Jacobian of linear inequality \\
stages.ineq[i][b] &= \ldots & \# RHS of linear inequality \\
\end{align*}
\]

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.

4.9 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.
If the index vector \( \text{idx1} \) has length \( m_1 \), then the matrix \( Q \) must be square and of size \( m_1 \times m_1 \), the column vector \( l_1 \) 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.

### 4.9.1 Example

To express the two quadratic constraints

\[
\begin{align*}
    z_3^2 + 2z_3^2 - z_{4,5} & \leq 3 \\
    5z_{3,1}^2 & \leq 1
\end{align*}
\]

on the third stage variable, use the code

```plaintext
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
```

### 4.10 Binary constraints

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

```plaintext
stages(1).bidx = [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.
Chapter 5

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.

5.1 Supported problems

5.1.1 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:

\[
\begin{align*}
\text{minimize} & \quad \sum_{k=1}^{N-1} f_k(z_k, p_k) \\
\text{subject to} & \quad z_1(\mathcal{I}) = z_{\text{init}}, \quad z_N(\mathcal{N}) = z_{\text{final}} \\
& \quad E_k z_{k+1} = c_k(z_k, p_k) \\
& \quad z_k \leq \bar{z}_k \leq \underline{z}_k \\
& \quad F_k z_k \in [\underline{z}_k, \bar{z}_k] \cap \mathbb{Z} \\
& \quad \ell_k \leq h_k(z_k, p_k) \leq \bar{h}_k 
\end{align*}
\]

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 $f_k : \mathbb{R}^{n_k} \to \mathbb{R}$ are stage cost functions; the functions $c_k : \mathbb{R}^{n_k} \to \mathbb{R}$ 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 $h_k : \mathbb{R}^{n_k} \to \mathbb{R}^{m_k}$ 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 $z_{\text{init}}$ and $z_{\text{final}}$ 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.

### 5.1.2 Continuous-time dynamics

Instead of having discrete-time dynamics as can be seen in Section 5.1.1, 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 5.2.3.2 for more details.

### 5.1.3 Other variants

Not all elements in Section 5.1.1 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 \(z_{\text{init}}\) and do not include the initial equality;
- that optimize over the final value \(z_{\text{final}}\) 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.

### 5.1.4 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 constraints \(h_k(z_k)\) 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 5.2 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 5.5 for details on how to provide own function evaluations and derivatives to FORCES Pro.
5.2 Expressing the optimization problem in MATLAB

When solving nonlinear programs of the type in Section 5.1.1, FORCES requires the functions \( f, c, h \) and their derivatives (Jacobians) to be evaluated in each iteration. There are in principle 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 5.5 to learn how to provide your own derivatives to FORCES NLP solvers. This section will outline the MATLAB-CasADi approach in detail.

5.2.1 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:

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

If the dimensions vary for different stages use arrays of length \( N \). See Section 5.2.6.1 for an example.

5.2.2 Objective

The high-level interface allows you to define the objective function using a handle to a MATLAB function that evaluates the objective. FORCES Pro will process this information and generate the necessary C code to be included in the solver.

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

For instance, the MATLAB function could be:

```matlab
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
```

If the cost function varies for different stages use a cell array of function handles of length \( N \). See Section 5.2.6.1 for an example.

In order to be able to change the terms in the cost function during runtime one can add a second argument to the MATLAB function, i.e. define the objective function in MATLAB as:

```matlab
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
```
5.2.3 Equalities

5.2.3.1 Discrete-time

For discrete-time dynamics, one can define a handle to a MATLAB 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
model.eq = @eval_dynamics; % handle to inter-stage function
model.E = [zeros(4,2), eye(2)]; % selection matrix
```

If the equality constraint function varies for different stages use a cell array of function handles of length \( N - 1 \), and similarly for \( E_k \). See Section 5.2.6.1 for an example.

5.2.3.2 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 inertia. For example, let’s assume that the system to be controlled has the dynamics:

\[
\dot{x} = p_1x_1x_2 + p_2u
\]

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

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

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

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

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
model.continuous_dynamics = @continuous_dynamics;
```

or, if you are using anonymous functions:

```matlab
model.continuous_dynamics = continuous_dynamics_anonymous;
```

3. Choose one of the integrator functions from the `integrators` section (the default is ERK4):

```matlab
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.
5.2.4 Initial and final conditions

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

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

5.2.5 Inequalities

The MATLAB function evaluating the nonlinear inequalities can be provided in a similar way, for example:

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

The simple bounds and the nonlinear inequality bounds can have \( +\infty \) and \( -\infty \) elements, but must be the same length as the fields \( nvar \) and \( nh \), respectively:

```matlab
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
```

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
% 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 = [];
```

and then specify the exact values at runtime through the fields \( lb01-lbN \) and \( ub01-ubN \):

```matlab
% 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 parameter bounds more easily.

If the `model.lbidx` and `model.ubidx` fields vary for different stages use cell arrays of length `N`.

### 5.2.6 Variations

#### 5.2.6.1 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. For instance, to remove the first and second variables from the last stage one could write the following:

```matlab
for i = 1:nlp.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 < nlp.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];
```

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

#### 5.2.6.2 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 differ-
entiation (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 provide functions and derivatives are consistent, for example:

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

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

### 5.3 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
% Get the default solver options
codeoptions = getOptions('FORCESNLPsolver');

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

#### 5.3.1 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
output1 = newOutput('u0', 1, 1:2);
FORCES_NLP(model, codeoptions, output1);
```

### 5.4 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
help FORCESNLPsolver
```

#### 5.4.1 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
x0i = nlp.lb +(nlp.ub -nlp.lb)/2;
x0 = repmat(x0i', nlp.N, 1);
problem.x0 = x0;
```
5.4.2 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
problem.xinit = model.xinit;
problem.xfinal = model.xfinal;
```

5.4.3 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
problem.all_parameters = repmat(1.0, model.N, 1);
```

5.4.4 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
[output, exitflag, info] = FORCESNLPsolver(problem);
```

The possible exitflags are documented in Table 5.1. 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
assert(exitflag == 1, 'Some problem in FORCES solver');
```

<table>
<thead>
<tr>
<th>Exitflag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Local optimal solution found (i.e. the point satisfies the KKT optimality conditions to the requested accuracy).</td>
</tr>
<tr>
<td>0</td>
<td>Maximum number of iterations reached. You can examine the value of optimality conditions returned by FORCES to decide whether the point returned is acceptable.</td>
</tr>
<tr>
<td>-4</td>
<td>Wrong number of inequalities input to solver.</td>
</tr>
<tr>
<td>-5</td>
<td>Error occurred during matrix factorization.</td>
</tr>
<tr>
<td>-6</td>
<td>NaN or INF occurred during functions evaluations.</td>
</tr>
<tr>
<td>-7</td>
<td>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.</td>
</tr>
<tr>
<td>-10</td>
<td>NaN or INF occurred 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.</td>
</tr>
<tr>
<td>-11</td>
<td>Invalid values in problem parameters.</td>
</tr>
<tr>
<td>-100</td>
<td>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.</td>
</tr>
</tbody>
</table>
5.5 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.

5.5.1 Interface

5.5.1.1 Array of parameters

The FORCES NLP solver will automatically call the following function, which is implemented by the user, to obtain the necessary information:

```c
void myfunctions (  
  double *x, /* primal vars */  
  double *y, /* eq. constraint multipliers */  
  double *l, /* ineq . constraint multipliers */  
  double *p, /* runtime parameters */  
  double *f, /* objective function ( incremented in this function ) */  
  double *nabla_f, /* gradient of objective function */  
  double *c, /* dynamics */  
  double *nabla_c, /* Jacobian of the dynamics ( column major ) */  
  double *h, /* inequality constraints */  
  double *nabla_h, /* Jacobian of inequality constraints ( column major ) */  
  double *H, /* Hessian ( column major ) */  
  int stage, /* stage number (0 indexed ) */  
  int iteration /* Solver iteration count */  )
```

5.5.1.2 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:

```c
codeoptions.customParams = 1;
```

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

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

(continues on next page)
i.e. you can pass arbitrary data structures to your own function by setting the pointer in the
params struct:

```c
myData p; /* define your own parameter structure */
/* ... */ /* fill it with data */
/* Set parameter pointer to your data structure */
mySolver_params params; /* Define solver parameters */
params.customParams = &p;
/* 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.

### 5.5.2 Suppling function evaluation information

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

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

### 5.5.3 Rules for function evaluation code

The contents of the function have to follow certain rules. We will use the following example
to illustrate them:

```c
/* cost */
if (f)
  (f) += -100*x[3] + 0.1*x[0]*x[0] + 0.01*x[1]*x[1];
/* gradient - only nonzero elements have to be filled in */
if (nabla_f)
  nabla_f [0] = 0.2*x[0];
  nabla_f [1] = 0.02*x[1];
  nabla_f [3] = -100;
/* eq constr */
if (c)
  vehicle_dyanmics (x, c);
/* jacobian equalities ( column major ) */
if (nabla_c)
  vehicle_dymams_jacobian (x, nabla_c);
```
/* ineq constr */
if (h)
{
    h[0] = x[2]*x[2] + x[3]*x[3];
}
/* jacobian inequalities ( column major )
only non-zero elements to be filled in */
if (nabla_h)
{
    /* column 3 */
    nabla_h[4] = 2*x[2];
    /* column 4 */
    nabla_h[6] = 2*x[3];
}

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.

5.5.4 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.

5.5.5 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);

To let the code generator know about the location of these other files use a string with spaces separating the different files.

nlp.other_srcs = 'C/dynamics.c';

5.5.6 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.

5.6 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.
5.6.1 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
model.lb = [];  
model.ub = [];  
```

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. A MATLAB example is shown below.

```matlab
%% Add integer variables to existing nonlinear model
for s = 1:5
    model.intidx{s} = [4, 5, 6];
end
```

In the above code snippet, the user declares variables 4, 5 and 6 as integers from stage 1 to 5. 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 at stage 1 have lower and upper bounds, which are expected to be provided at run-time.

```matlab
model.lbidx{1} = 1 : 6;  
model.ubidx{1} = 1 : 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 5.2, 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 5.2.

<table>
<thead>
<tr>
<th>Code generation setting</th>
<th>Values</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>minlp.int_gap_tol</code></td>
<td>Any value ≥ 0</td>
<td>0.001</td>
</tr>
<tr>
<td><code>minlp.max_num_nodes</code></td>
<td>Any value ≥ 0</td>
<td>10000</td>
</tr>
<tr>
<td><code>minlp.seq_search_strat</code></td>
<td>'BEST_FIRST', 'BREADTH_FIRST', 'DEPTH_FIRST'</td>
<td>'BEST_FIRST'</td>
</tr>
<tr>
<td><code>minlp.par_search_strat</code></td>
<td>'BEST_FIRST', 'BREADTH_FIRST', 'DEPTH_FIRST'</td>
<td>'BEST_FIRST'</td>
</tr>
<tr>
<td><code>minlp.max_num_threads</code></td>
<td>Any nonnegative value preferably smaller than 8</td>
<td>4</td>
</tr>
</tbody>
</table>

- 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`. 

Chapter 5. High-level Interface
The `minlp.max_num_nodes` setting is the maximum number of nodes which can be explored during the search.

The `minlp.seq_search_strat` setting is the search strategy which is used to select candidate nodes during the sequential search phase.

The `minlp.par_search_strat` setting is the search strategy which is used to select candidate nodes during the parallelizable search phase.

The `minlp.max_num_threads` setting is the maximum number of threads allowed for a parallel search. The actual number of threads on which the branch-and-bound algorithm can be run can be set as a run-time parameter, as described below.

**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:

- `numThreadsBnB`, the number of threads used to parallelize the search. Its default value is 1, if not provided by the user.
- `timeOutBnB`, the maximum amount of time allowed for completing the search. Its default value is 1.0 seconds, if not set by the user.

### 5.6.2 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.

```plaintext
%% Add post-rounding callback to existing model
postRndCall = fileread('postRoundCallback_template.c'); % The file name can be changed by the user
model.minlpPostRounding = postRndCall;
```

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.

```plaintext
%% 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;
```
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.

**Note:** The MINLP feature is currently available through the MATLAB interface only.

### 5.7 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:** 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:** 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.
Chapter 6

Examples

6.1  Y2F interface: Basic example

Consider the following linear MPC problem with lower and upper bounds on state and inputs, and a terminal cost term:

\[
\begin{align*}
\text{minimize} & \quad x_N^T P x_N + \sum_{i=0}^{N-1} x_i^T Q x_i + u_i^T R u_i \\
\text{subject to} & \quad x_0 = x \\
& \quad x_{i+1} = A x_i + B u_i \\
& \quad \underline{x} \leq x_i \leq \bar{x} \\
& \quad \underline{u} \leq u_i \leq \bar{u}
\end{align*}
\]

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).

6.1.1 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:

```matlab
%% MPC problem data
% system matrices
A = [1.1 1; 0 1];
B = [1; 0.5];
[nx,nu] = size(B);
% horizon
N = 10;

(continues on next page)```
6.1.2 Defining the MPC problem

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

```matlab
%% 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.

6.1.3 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:

```matlab
%% Create controller object (generates code)
%c for a complete list of codeoptions, see
```
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.

### 6.1.4 Calling the generated solver

We can now use the `controller` object to call the solver:

```matlab
% Evaluate controller function for parameters
[output,exitflag,info] = controller{ xinit };
```

or call the generated MEX code directly:

```matlab
% 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.

### 6.1.5 Simulation

Let’s now simulate the closed loop over the prediction horizon $N$:

```matlab
%% 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.
');
    else
        info
        error('Some problem in solver');
    end
    % State update
    X(:,k+1) = A*X(:,k) + B*U(:,k);
end
```
6.1.6 Results

The results of the simulation are presented in Figure 6.1. 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.

![Simulation results of the states and input over time.](image)

Figure 6.1: 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.

6.1.7 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:

$$\begin{align*}
\text{minimize} & \quad x_N^T P x_N + \sum_{i=0}^{N-1} x_i^T Q x_i + u_i^T R u_i \\
\text{subject to} & \quad x_0 = x \\
& \quad x_{i+1} = A x_i + B u_i \\
& \quad x \leq x_i \leq \bar{x} \\
& \quad u \leq u_i \leq \bar{u}
\end{align*}$$

As usual, this problem is also parametric in the initial state $x$ and the first input $u_0$ is applied to the system after a solution has been obtained. To be able to define the weighting matrices $Q$, $R$ and $P$ as parameters, first we define them as `sdpvars` and then tell `optimizerFORCES` that they are parameters:
% Cost matrices - these will be parameters later
Q = sdpvar(nx);
R = sdpvar(nu);
P = sdpvar(nx);

% [... formulate MPC problem in YALMIP ...]

% Define parameters and outputs
codeoptions = getOptions('parametricCost_solver'); % give solver a name
parameters = { X(:,1), Q, R, P };  
parameterNames = { 'xinit', 'Q', 'R', 'P' };  
outputs = U(:,1);  
outputNames = {'controlInput'};
controller = optimizerFORCES(const, cost, codeoptions, parameters, outputs,  
parameterNames, outputNames);

You can download the Matlab code of this variation to try it out for yourself from https://raw.githubusercontent.com/embotech/Y2F/master/examples/mpc_parametric_cost.m.

### 6.1.8 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:

\[
\begin{align*}
\text{minimize} & \quad x^T N P x_N + \sum_{i=0}^{N-1} x_i^T Q x_i + u_i^T R u_i \\
\text{subject to} & \quad x_0 = x \\
& \quad x_{i+1} = A x_i + B u_i \\
& \quad \bar{x} \leq x_i \leq \bar{x} \\
& \quad \bar{u} \leq u_i \leq \bar{u}
\end{align*}
\]

As usual, this problem is also parametric in the initial state $x$ and the first input $u_0$ is applied to the system after a solution has been obtained. To be able to define the state-space dynamics matrices $A$ and $B$ as parameters, first we define them as sdpvars and then tell optimizerFORCES that they are parameters:

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

% [... 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.

### 6.1.9 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 $2 \times 2$ matrix.

As an example, consider the following constraints:

\[
\begin{align*}
\text{subject to} & \quad A x_i + B u_i \\ & \quad x_i \leq \bar{x} \\
& \quad \bar{u} \leq u_i \leq \bar{u}
\end{align*}
\]

As usual, this problem is also parametric in the initial state $x$ and the first input $u_0$ is applied to the system after a solution has been obtained. To be able to define the state-space dynamics matrices $A$ and $B$ as parameters, first we define them as sdpvars and then tell optimizerFORCES that they are parameters:

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

% [... 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.
matrix that is defined by 2 parameters:
\[ R_k x \leq R_k \dot{x} \]
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:

\[
\text{minimize } x_N^T P x_N + \sum_{i=0}^{N-1} x_i^T Q x_i + u_i^T R u_i
\]

subject to
\[
\begin{align*}
x_0 &= x \\
x_{i+1} &= A x_i + B u_i \\
\underline{x} &\leq x_i \leq \bar{x} \\
\underline{u} &\leq u_i \leq \bar{u} \\
R_k x_i &\leq R_k \dot{x}
\end{align*}
\]

As usual, this problem is also parametric in the initial state \( x \) and the first input \( u_0 \) is applied to the system after a solution has been obtained. To be able to define the rotation matrix \( R_k \) as a parameter, first we define \( r_1 \) and \( r_2 \) as sdpvars and then tell optimizerFORCES that they are parameters:

```matlab
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 }; % xinit', sprintf('cos(k*%.2f)',w), sprintf('sin(k*%.2f)',w) 
parameters = { 'xinit', sprintf('cos(k*%.2f)',w), sprintf('sin(k*%.2f)',w) 
parameters = { 'xinit', sprintf('cos(k*%.2f)',w), sprintf('sin(k*%.2f)',w) 
parameters = { X(:,1), r1, r2 }; % xinit', sprintf('cos(k*%.2f)',w), sprintf('sin(k*%.2f)',w) 
parameters = { 'xinit', sprintf('cos(k*%.2f)',w), sprintf('sin(k*%.2f)',w) 
parameters = { 'xinit', sprintf('cos(k*%.2f)',w), sprintf('sin(k*%.2f)',w) 
parameters = { X(:,1), r1, r2 }; % xinit', sprintf('cos(k*%.2f)',w), sprintf('sin(k*%.2f)',w) 
parameters = { 'xinit', sprintf('cos(k*%.2f)',w), sprintf('sin(k*%.2f)',w) 
parameters = { 'xinit', sprintf('cos(k*%.2f)',w), sprintf('sin(k*%.2f)',w) 
parameters = { X(:,1), r1, r2 }; % xinit', sprintf('cos(k*%.2f)',w), sprintf('sin(k*%.2f)',w) 
parameters = { 'xinit', sprintf('cos(k*%.2f)',w), sprintf('sin(k*%.2f)',w) 
parameters = { 'xinit', sprintf('cos(k*%.2f)',w), sprintf('sin(k*%.2f)',w) 
parameters = { X(:,1), r1, r2 }; % xinit', sprintf('cos(k*%.2f)',w), sprintf('sin(k*%.2f)',w) 
parameters = { 'xinit', sprintf('cos(k*%.2f)',w), sprintf('sin(k*%.2f)',w) 
parameters = { 'xinit', sprintf('cos(k*%.2f)',w), sprintf('sin(k*%.2f)',w) 
parameters = { X(:,1), r1, r2 }; % xinit', sprintf('cos(k*%.2f)',w), sprintf('sin(k*%.2f)',w) 
parameters = { 'xinit', sprintf('cos(k*%.2f)',w), sprintf('sin(k*%.2f)',w) 
parameters = { 'xinit', sprintf('cos(k*%.2f)',w), sprintf('sin(k*%.2f)',w) 
parameters = { X(:,1), r1, r2 }; % xinit', sprintf('cos(k*%.2f)',w), sprintf('sin(k*%.2f)',w) 
parameters = { 'xinit', sprintf('cos(k*%.2f)',w), sprintf('sin(k*%.2f)',w) 
parameters = { 'xinit', sprintf('cos(k*%.2f)',w), sprintf('sin(k*%.2f)',w) 
parameters = { X(:,1), r1, r2 }; % xinit', sprintf('cos(k*%.2f)',w), sprintf('sin(k*%.2f)',w) 
parameters = { 'xinit', sprintf('cos(k*%.2f)',w), sprintf('sin(k*%.2f)',w) 
parameters = { 'xinit', sprintf('cos(k*%.2f)',w), sprintf('sin(k*%.2f)',w) 
parameters = { X(:,1), r1, r2 }; % xinit', sprintf('cos(k*%.2f)',w), sprintf('sin(k*%.2f)',w) 
parameters = { 'xinit', sprintf('cos(k*%.2f)',w), sprintf('sin(k*%.2f)',w) 
parameters = { 'xinit', sprintf('cos(k*%.2f)',w), sprintf('sin(k*%.2f)',w) 
```

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.
6.2 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:

\[
\begin{align*}
\text{minimize} & \quad x_N^T P x_N + \sum_{i=0}^{N-1} x_i^T Q x_i + u_i^T R u_i \\
\text{subject to} & \quad x_0 = x \\
& \quad x_{i+1} = Ax_i + Bu_i \\
& \quad x_l \leq x_i \leq x_u \\
& \quad u_l \leq u_i \leq u_u
\end{align*}
\]

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.

6.2.1 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:

```matlab
%% 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];
```

6.2.2 Defining the MPC problem

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

```matlab
%% 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)];
```

(continues on next page)
6.2.3 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:

```matlab
%% Generate FORCES solver
% get options
codeoptions = getOptions('FORCESNLPsolver');
codeoptions.printlevel = 2;
% generate code
FORCES_NLP(model, codeoptions);
```

6.2.4 Calling the generated solver

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

```matlab
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.

6.2.5 Simulation

Let's now simulate the closed loop over the prediction horizon `N`:

```matlab
%% 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);
    FORCES_NLP(model, codeoptions);
```

(continues on next page)
[solverout,exitflag,info] = FORCESNLPsolver(problem);

if (exitflag == 1)
    U(:,k) = solverout.x01(i);
    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

6.2.6 Results

The results of the simulation are presented in Figure 6.1. 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 6.2: 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.](image)

Figure 6.2: 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.
6.3 High-level interface: Obstacle avoidance

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):

\[
\begin{align*}
\dot{x} &= v \cos(\theta) \\
\dot{y} &= v \sin(\theta) \\
\dot{v} &= \frac{F}{m} \\
\dot{\theta} &= \frac{s}{L}
\end{align*}
\]

The model consists of four differential states: \(x\) and \(y\) are the Cartesian coordinates of the car, \(v\) is the linear velocity and \(\theta\) 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 \(m = 1\) kg and the wheel base which we assume to be \(L = 1\) 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 Matlab code of this example to try it out for yourself in the examples folder that comes with your client.

6.3.1 Defining the problem data

6.3.1.1 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.:

\[f(z) = -100z_1 + 0.1z_1^2 + 0.01z_2^2\]

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

```matlab
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
```

6.3.1.2 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 defined 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
function xdot = continuous_dynamics(x, u)
    F = u(1);
    s = u(2);
    v = x(3);
    theta = x(4);
```

(continues on next page)
\[ m = 1; \]
\[ L = 1; \]
\[
\dot{x} = \begin{bmatrix} v \cdot \cos(\theta); \\ v \cdot \sin(\theta); \\ F / m; \\ s / L \end{bmatrix};
\]

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

### 6.3.1.3 Inequality constraints

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

\[
\begin{align*}
-5 \text{N} & \leq F \leq 5 \text{N} \\
-1 \text{Nm} & \leq s \leq 1 \text{Nm} \\
-3 \text{m} & \leq x \leq 0 \text{m} \\
0 \text{m} & \leq y \leq 3 \text{m} \\
0 \text{m/s} & \leq v \leq 2 \text{m/s} \\
0 \text{rad} & \leq \theta \leq \pi \text{rad}
\end{align*}
\]

as well the nonlinear nonconvex constraints:

\[
\begin{align*}
1 \text{m}^2 \leq x^2 + y^2 & \leq 9 \text{m}^2 \\
0.9025 \text{m}^2 & \leq (x + 2)^2 + (y - 2.5)^2
\end{align*}
\]

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

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

### 6.3.1.4 Initial and final conditions

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

\[
\begin{align*}
x_{\text{init}} & = -2 \text{m}, \\
y_{\text{init}} & = 0 \text{m}, \\
v_{\text{init}} & = 0 \text{m/s}, \\
\theta_{\text{init}} & = 2.0944 \text{rad}
\end{align*}
\]

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

\[
\begin{align*}
v_{\text{final}} & = 0 \text{m/s}, \\
\theta_{\text{final}} & = 0 \text{rad}
\end{align*}
\]
6.3.2 Defining the MPC problem

With the above defined 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
%% 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];
% 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;
```

6.3.3 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
%% 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;

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

6.3.4 Calling the generated solver

Once all parameters have been populated, the MEX interface of the solver can be used to invoke it.
%% 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('FORCES took %d iterations and %f seconds to solve the problem.
',info.it,info.solvetime);

6.3.5 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 6.3.

The vehicle's velocity and steering angle over time is presented in Figure 6.4, and the actuator commands in Figure 6.5. One can see that all constraints are respected.
Figure 6.4: Vehicle's velocity and steering angle over time.

Figure 6.5: Vehicle's actuator commands over time.
6.3.6 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
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
```

and then include them into our dynamics function handles:

```matlab
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
```

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

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

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

6.3.7 Variation 2: Different integrator

Another possible variation is if we want to change the integrator that is used to discretize the continuous-time dynamics. To do that we set, for example, the `codeoptions.nlp` fields:

```matlab
% define integrator options
codeoptions.nlp.integrator.type = 'IRK4'; % can also be 'ForwardEuler', 'ERK2', ...
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 Matlab code of this example to try it out for yourself in the `examples` folder that comes with your client.
6.3.8 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
%% 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;
```

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

6.3.9 Variation 4: External functions

One final variation is if we supply the required functions through external functions in C. To do that we have to provide the directory that contains said source files:

```matlab
%% Define source file containing function evaluation code
model.extfuncs = 'C/myfevals.c';
```

We also need to include the two external functions car_dyanmics and car_dyanmics_jacobian, both contained in the car_dynamics.c file, through the other_srcs options field:

```matlab
% add additional source files required - separate by spaces if more than 1
codeoptions.nlp.other_srcs = 'C/car_dynamics.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.
6.4 High-level interface: Indoor localization

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 6.6: Indoor localization example GUI.

You can find the Matlab 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 6.6.

6.4.1 Time of flight measurements

Given \( N \) anchors with known positions \((x_{a_i}, y_{a_i})\), \(i = 1, \ldots, N\), the distance to the target with unknown position \((x, y)\) is given by:

\[
d_i = c t_i = \sqrt{(x - x_{a_i})^2 + (y - y_{a_i})^2}
\]
where $t_i$ is the time the signal from anchor $i$ travels at the speed $c = 299 792 458 \text{ m/s}$

### 6.4.2 Estimation error

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

$$e_i = (x - x_i^0)^2 + (y - y_i^0)^2 - d_i^2$$

### 6.4.3 Minimize the error

The objective is a least-squares error function:

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

### 6.4.4 Implementation

The following Matlab 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
%% 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
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;
```

(continues on next page)
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
6.5 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):

\[
\dot{x}_0 = -0.877 x_0 + x_2 - 0.088 x_0 x_2 + 0.47 x_0^2 - 0.019 x_1^2 - x_0^2 x_2 + 3.846 x_0^3 \\
- 0.215 w + 0.28 x_0^2 w + 0.47 x_0 w^2 + 0.63 w^3 \\
\dot{x}_1 = x_2 \\
\dot{x}_2 = -0.4208 x_0 - 0.396 x_2 - 0.47 x_0^2 - 3.564 x_0^3 - 20.967 w \\
+ 6.265 x_0^2 w + 46 x_0 w^2 + 61.4 w^3
\]

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.

6.5.1 Defining the problem data

6.5.1.1 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:

\[f(z) = 150 x_0^2 + 5 x_1^2 + 5 x_2^2\]

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

```matlab
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:

```matlab
model.objective = @(z) 0.1 * z(1)^2;
```

6.5.1.2 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

\[w = 0.05236 \cdot (2u - 1)\]

If \(w\) has values within \([-0.05236, 0.05236]\), then \(u\) lies within the binary set \(\{0, 1\}\).
\( wa = 0.05236; \)
\( wa2 = wa^2; \)
\( wa3 = wa^3; \)
\[
\text{continuous\_dynamics} = @(x, u) \begin{align*}
&\begin{bmatrix}
-0.877 \times x(1) + x(3) - 0.088 \times x(1) \times x(3) \\
+ 0.47 \times x(1) + x(1) - 0.019 \times x(2) \times x(2) \\
- x(1) \times x(1) + x(1) \\
+ 3.846 \times x(1) \times x(1) \times x(1) \\
- 0.215 \times wa \times (2 \times u(1) - 1) \\
+ 0.28 \times x(1) \times x(1) \times wa \times (2 \times u(1) - 1) \\
+ 0.47 \times x(1) \times wa2 \times (2 \times u(1) - 1) \times (2 \times u(1) - 1) \\
+ 0.63 \times wa3 \times (2 \times u(1) - 1) \times (2 \times u(1) - 1) \times (2 \times u(1) - 1) \\
\end{bmatrix}; \\
&x(3); \\
-4.208 \times x(1) - 0.396 \times x(3) - 0.47 \times x(1) \times x(1) \\
- 3.564 \times x(1) \times x(1) \times x(1) \\
- 20.967 \times wa \times (2 \times u(1) - 1) \\
+ 6.265 \times x(1) \times x(1) \times wa \times (2 \times u(1) - 1) \\
+ 46.0 \times x(1) \times wa2 \times (2 \times u(1) - 1) \times (2 \times u(1) - 1) \\
+ 61.4 \times wa3 \times (2 \times u(1) - 1) \times (2 \times u(1) - 1) \times (2 \times u(1) - 1); \\
\end{align*}
\]
\]
\]
\]
\]
\]
\]
\]
\subsubsection{6.5.1.3 Inequality constraints}

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

\[
\begin{align*}
0 \text{ rad} &\leq u \leq 1 \text{ rad} \\
-10 \text{ rad} &\leq x_0 \leq 10 \text{ rad} \\
-10 \text{ rad} &\leq x_1 \leq 10 \text{ rad} \\
-10 \text{ rad/sec} &\leq x_2 \leq 10 \text{ rad/sec}
\end{align*}
\]

\subsubsection{6.5.1.4 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.

\subsubsection{6.5.2 Defining the MPC problem}

With the above defined MATLAB functions for objective and equality constraints, we can completely define the MINLP formulation in the next code snippet. For this example, the number of stages has been set to \( N = 100 \).

In the code snippet below, it is important to notice that the lower and upper bounds are declared as parametric before generating the solver. This needs to be done for generating mixed-integer NLP solvers. Lower and upper bounds are meant to be provided at run-time.

```matlab
%% Problem dimension
nx = 3;
nu = 1;
nz = nx + nu;
model.N = 100;
```
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(3) + 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) * (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 * x(1) * 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
model.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{1} = [1]; 
end

6.5.3 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.

6.5.4 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);

6.5.5 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)\(^7\), 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 6.7 and the input sequence is in Figure Figure 6.8. 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 6.9 and Figure 6.10.
Figure 6.7: Aircraft’s angle of attack over time computed with one thread.

Figure 6.8: Aircraft’s tail deflection angle over time with one thread.
Figure 6.9: Aircraft’s angle of attack over time computed with three threads.

Figure 6.10: Aircraft’s tail deflection angle over time with three threads.
Chapter 7

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.

7.1 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

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

and in Python, use

```python
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>
<thead>
<tr>
<th>Cost function</th>
<th>Equality constraints</th>
<th>Inequality constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>'cost.H'</td>
<td>'eq.c'</td>
<td>'ineq.b.lb'</td>
</tr>
<tr>
<td>'cost.f'</td>
<td>'eq.C'</td>
<td>'ineq.b.ub'</td>
</tr>
<tr>
<td>'eq.D'</td>
<td>'ineq.p.A'</td>
<td></td>
</tr>
<tr>
<td></td>
<td>'ineq.p.b'</td>
<td></td>
</tr>
<tr>
<td></td>
<td>'ineq.q.Q'</td>
<td></td>
</tr>
<tr>
<td></td>
<td>'ineq.q.l'</td>
<td></td>
</tr>
<tr>
<td></td>
<td>'ineq.q.r'</td>
<td></td>
</tr>
</tbody>
</table>

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

```python
maps2stage = [];
```
For instance, in order to provide a parametric linear cost across all stages, one should use the following code at codegen.

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

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

```matlab
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’.

### 7.2 Example

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

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

and in Python, use

```python
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

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

In Python

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

### 7.3 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

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

In Python

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

where `idxWithinStage` denotes the index of the quadratic constraints to which this parameters applies.
7.4 Diagonal Hessians

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

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

In Python

```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.

7.5 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

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

In Python

```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

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

In Python

```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.

7.6 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

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

In Python

```python
```
stages.newParam('C1', 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.

### 7.7 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:

```python
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 8

Code Deployment

8.1 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.

8.1.1 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:

```matlab
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

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)
8.1.2 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:

```matlab
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):

   ```bash
   <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)
8.2 dSPACE MicroAutoBox II

8.2.1 High-level interface

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

1. (Figure 8.1) Set the code generation options:

```matlab
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 8.2) Create a new Simulink model using the RTI1401 template provided by dSPACE.

3. (Figure 8.3) Populate the Simulink model with the system you want to control.

4. (Figure 8.4) Make sure the FORCESNLPsolver_simulinkBlock.mexw64 file (created during code generation) is on the Matlab path.

5. (Figure 8.5) Open the FORCESNLPsolver_lib.mdl Simulink model file, contained in the interface folder of the FORCESNLPsolver folder created during code generation.

6. (Figure 8.6) Copy-paste the FORCES Pro Simulink block into your simulation model and connect its inputs and outputs appropriately.

7. (Figure 8.7) Access the Simulink model’s options.

8. (Figure 8.8) 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 8.9) 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 8.10) In the “Code Generation/Custom Code” tab, include the directories:
   - BasicExample
   - BasicExample\FORCESNLPsolver\interface
   - BasicExample\FORCESNLPsolver\lib_target

11. (Figure 8.11) 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 8.12) In the “Code Generation/Custom Code” tab, add the library files:
   • FORCESNLPsolver.lib
13. (Figure 8.13) Access the FORCES Pro block’s parameters.
14. (Figure 8.14) Remove the “FORCESNLPsolver” prefix from the S-function module.
15. (Figure 8.15) Compile the code of the Simulink model. This will also automatically load
    the model to the connected MicroAutoBox.

Figure 8.1: Set the appropriate code generation options.

Figure 8.2: Create a Simulink model.

Figure 8.3: Populate the Simulink model.
Figure 8.4: Add the folder containing the .mexw64 solver file to the Matlab path.

Figure 8.5: Open the generated Simulink solver model.

Figure 8.6: Copy-paste and connect the FORCES Pro block.

Figure 8.7: Open the Simulink model options.
Figure 8.8: Set the Simulink solver options.

Figure 8.9: Set the Simulink code generation options.

Figure 8.10: Add the directories included for the code generation.

Figure 8.11: Add the source files used for the code generation.
Figure 8.12: Add the libraries used for the code generation.

Figure 8.13: Open the FORCES Pro block's parameters.

Figure 8.14: Remove the leading solver name from the S-function module.

Figure 8.15: Compile the code of the Simulink model.
8.2.2 Y2F interface

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

1. (Figure 8.16) Set the code generation options:

   ```matlab
   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 8.17) Create a new Simulink model using the RTI401 template provided by dSPACE.

3. (Figure 8.18) Populate the Simulink model with the system you want to control.

4. (Figure 8.19) Make sure the `simplempc_solver_simulinkBlock.mexw64` file (created during code generation) is on the Matlab path.

5. (Figure 8.20) 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 8.21) Access the Simulink model’s options.

7. (Figure 8.22) 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 8.23) In the “Code Generation/RTI general build options” tab, set the options:
   - System target file: `rti401.tlc`
   - Language: C
   - Generate makefile: On
   - Template makefile: `rti401.tmf`
   - Make command: `make_rti`

9. (Figure 8.24) In the “Code Generation/Custom Code” tab, include the directories:
   - Y2F
   - Y2F\simplempc_solver\interface
   - Y2F\simplempc_solver\lib_target

10. (Figure 8.25) In the “Code Generation/Custom Code” tab, add the source files:
    - `simplempc_solver_simulinkBlock.c`
    - `simplempc_solver.c`

11. (Figure 8.26) In the “Code Generation/Custom Code” tab, add the library files:
    - `internal_simplempc_solver_1.lib`

12. (Figure 8.27) Compile the code of the Simulink model. This will also automatically load the model to the connected MicroAutoBox.
Figure 8.16: Set the appropriate code generation options.

Figure 8.17: Create a Simulink model.

Figure 8.18: Populate the Simulink model.

Figure 8.19: Add the folder containing the .mexw64 solver file to the Matlab path.
Figure 8.20: Copy-paste and connect the FORCES Pro block.

Figure 8.21: Open the Simulink model options.

Figure 8.22: Set the Simulink solver options.

Figure 8.23: Set the Simulink code generation options.
Figure 8.24: Add the directories included for the code generation.

Figure 8.25: Add the source files used for the code generation.

Figure 8.26: Add the libraries used for the code generation.
Figure 8.27: Compile the code of the Simulink model.
8.3 Speedgoat

8.3.1 High-level interface

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

1. (Figure 8.28) Set the code generation options:

   ```matlab
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 8.29) Create a new Simulink model using the blank model template.

3. (Figure 8.30) Populate the Simulink model with the system you want to control.

4. (Figure 8.31) Make sure the `FORCESNLPsolver_simulinkBlock.mexw64` file (created during code generation) is on the Matlab path.

5. (Figure 8.32) Open the `FORCESNLPsolver_lib.mdl` Simulink model file, contained in the interface folder of the `FORCESNLPsolver` folder created during code generation.

6. (Figure 8.33) Copy-paste the FORCES Pro Simulink block into your simulation model and connect its inputs and outputs appropriately.

7. (Figure 8.34) Access the Simulink model's options.

8. (Figure 8.35) 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 8.36) 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 8.37) In the “Code Generation/Custom Code” tab, include the directories:

    - `BasicExample`
    - `BasicExample\FORCESNLPsolver\interface`
    - `BasicExample\FORCESNLPsolver\lib_target`

11. (Figure 8.38) 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 8.39) In the “Code Generation/Custom Code” tab, add the library files:

    - `FORCESNLPsolver.lib`
13. **(Figure 8.40)** Access the FORCES Pro block’s parameters.

14. **(Figure 8.41)** Remove “FORCESNLPsolver” and “FORCESNLPsolver_simulinkBlock” from the S-function module.

15. **(Figure 8.42)** 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 8.28: Set the appropriate code generation options.

Figure 8.29: Create a Simulink model.

Figure 8.30: Populate the Simulink model.
Figure 8.31: Add the folder containing the .mexw64 solver file to the Matlab path.

Figure 8.32: Open the generated Simulink solver model.

Figure 8.33: Copy-paste and connect the FORCES Pro block.

Figure 8.34: Open the Simulink model options.
Figure 8.35: Set the Simulink solver options.

Figure 8.36: Set the Simulink code generation options.

Figure 8.37: Add the directories included for the code generation.
Figure 8.38: Add the source files used for the code generation.

Figure 8.39: Add the libraries used for the code generation.

Figure 8.40: Open the FORCES Pro block's parameters.
Figure 8.41: Remove the default data from the S-function module.

Figure 8.42: Compile the code of the Simulink model.
The steps to deploy and simulate a FORCES Pro controller on a Speedgoat platform are detailed below.

1. (Figure 8.43) Set the code generation options:

```matlab
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 8.44) Create a new Simulink model using the blank model template.

3. (Figure 8.45) Populate the Simulink model with the system you want to control.

4. (Figure 8.46) Make sure the `simplempc_solver_simulinkBlock.mexw64` file (created during code generation) is on the Matlab path.

5. (Figure 8.47) 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 8.48) Access the Simulink model’s options.

7. (Figure 8.49) 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 8.50) 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 8.51) In the “Code Generation/Custom Code” tab, include the directories:
   - `Y2F\simplempc_solver\interface`
   - `Y2F\simplempc_solver\lib_target`

10. (Figure 8.52) In the “Code Generation/Custom Code” tab, add the source files:
    - `simplempc_solver_simulinkBlock.c`
    - `simplempc_solver.c`

11. (Figure 8.53) In the “Code Generation/Custom Code” tab, add the library files:
    - `internal_simplempc_solver_1.lib`

12. (Figure 8.54) 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 8.43: Set the appropriate code generation options.

Figure 8.44: Create a Simulink model.

Figure 8.45: Populate the Simulink model.

Figure 8.46: Add the folder containing the .mexw64 solver file to the Matlab path.

Figure 8.47: Copy-paste and connect the FORCES Pro block.
Chapter 8. Code Deployment

Chapter 8. Code Deployment

Figure 8.48: Open the Simulink model options.

Figure 8.49: Set the Simulink solver options.

Figure 8.50: Set the Simulink code generation options.
Figure 8.51: Add the directories included for the code generation.

Figure 8.52: Add the source files used for the code generation.

Figure 8.53: Add the libraries used for the code generation.
Figure 8.54: Compile the code of the Simulink model.
Chapter 9

Solver Options

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

\[
\text{codeoptions} = \text{getOptions('solvername');}
\]

In the documentation below, we assume that you have created this struct and named it \text{codeoptions}.

9.1 General options

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

9.1.1 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:

\[
\text{codeoptions.name} = \text{'solvername'};
\]

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

\[
\text{codeoptions} = \text{getOptions('solvername');}
\]

9.1.2 Print level

To control the amount of information the generated solver prints to the console, set the field \text{printlevel} as outlined in Table 9.1.

<table>
<thead>
<tr>
<th>printlevel</th>
<th>Result</th>
<th>Dependency</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No output will be written.</td>
<td>(None)</td>
</tr>
<tr>
<td>1</td>
<td>Summary line after each solve.</td>
<td>&lt;stdio.h&gt;</td>
</tr>
<tr>
<td>2 (default)</td>
<td>Summary after each iteration of solver.</td>
<td>&lt;stdio.h&gt;</td>
</tr>
</tbody>
</table>
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.

9.1.3 Maximum number of iterations

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

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

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

9.1.4 Compiler optimization level

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

```cpp
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.

9.1.5 Running solvers in parallel

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

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

9.1.6 Measure Computation time

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

```cpp
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:

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

### 9.1.7 Datatypes

The type of variables can be changed by setting the field `floattype` as outlined in Table 9.2.

<table>
<thead>
<tr>
<th>floattype</th>
<th>Decimation</th>
<th>Width (bits)</th>
<th>Supported algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>'double' (default)</td>
<td>64 bit</td>
<td>Floating point</td>
<td>PDIP, PDIP_NLP, ADMM, DFG, FG</td>
</tr>
<tr>
<td>'float'</td>
<td>32 bit</td>
<td>Floating point</td>
<td>PDIP, PDIP_NLP, ADMM, DFG, FG</td>
</tr>
<tr>
<td>'int'</td>
<td>32 bit</td>
<td>Fixed point</td>
<td>PDIP, PDIP_NLP, ADMM, DFG, FG</td>
</tr>
<tr>
<td>'short'</td>
<td>16 bit</td>
<td>Fixed point</td>
<td>PDIP, PDIP_NLP, ADMM, DFG, FG</td>
</tr>
</tbody>
</table>

**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).

### 9.1.8 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 9.3.

<table>
<thead>
<tr>
<th>overwrite</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Never overwrite.</td>
</tr>
<tr>
<td>1</td>
<td>Always overwrite.</td>
</tr>
<tr>
<td>2 (default)</td>
<td>Ask to overwrite.</td>
</tr>
</tbody>
</table>

### 9.1.9 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:

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

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

### 9.1.10 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:

```plaintext
codeoptions.server = 'http://embotech-server2.com:8114/v1.5.beta';
```
9.1.11 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:

```plaintext
codeoptions.BuilSimulinkBlock = 0;
```

9.1.12 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:

```plaintext
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.

9.1.13 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 9.4.

<table>
<thead>
<tr>
<th>platform</th>
<th>Platform</th>
</tr>
</thead>
<tbody>
<tr>
<td>'Generic' (default)</td>
<td>For the architecture of the host platform.</td>
</tr>
<tr>
<td>'x86_64'</td>
<td>For x86_64 based 64-bit platforms (detected OS).</td>
</tr>
<tr>
<td>'x86'</td>
<td>For x86 based 32-bit platforms (detected OS).</td>
</tr>
<tr>
<td>'Win-x86_64'</td>
<td>For Windows x86_64 based 64-bit platforms (supports Microsoft/Intel compiler).</td>
</tr>
<tr>
<td>'Win-x86'</td>
<td>For Windows x86 based 32-bit platforms (supports Microsoft/Intel compiler).</td>
</tr>
<tr>
<td>'Win-MinGW-x86_64'</td>
<td>For Windows x86 based 32-bit platforms (supports MinGW compiler).</td>
</tr>
<tr>
<td>'Win-MinGW-x86'</td>
<td>For Windows x86 based 32-bit platforms (supports MinGW compiler).</td>
</tr>
<tr>
<td>'Mac-x86_64'</td>
<td>For Mac x86_64 based 64-bit platforms (supports GCC/Clang compiler).</td>
</tr>
<tr>
<td>'Gnu-x86_64'</td>
<td>For Linux x86_64 based 64-bit platforms (supports GCC compiler).</td>
</tr>
<tr>
<td>'Gnu-x86'</td>
<td>For Linux x86 based 32-bit platforms (supports GCC compiler).</td>
</tr>
<tr>
<td>'Docker-Gnu-x86_64'</td>
<td>For Linux x86_64 based 64-bit platforms on Docker (supports GCC compiler).</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Platform</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'Docker-Gnu-x86'</td>
<td>For Linux x86 based 32-bit platforms on Docker (supports GCC compiler).</td>
</tr>
<tr>
<td>'ARM-Generic'</td>
<td>For ARM Cortex 32-bit processors (Gnueabih machine type).</td>
</tr>
<tr>
<td>'ARM-Generic64'</td>
<td>For ARM Cortex 64-bit processors (Aarch machine type).</td>
</tr>
<tr>
<td>'Integrity-ARM-x86'</td>
<td>For ARM Cortex 32-bit processors using the Integrity toolchain.</td>
</tr>
<tr>
<td>'Integrity-ARM-x64'</td>
<td>For ARM Cortex 64-bit processors using the Integrity toolchain.</td>
</tr>
<tr>
<td>'ARM Cortex-M3'</td>
<td>For ARM Cortex M3 32-bit processors.</td>
</tr>
<tr>
<td>'ARM Cortex-M4-NOPPU'</td>
<td>For the ARM Cortex M4 32-bit processors without a floating-point unit.</td>
</tr>
<tr>
<td>'ARM Cortex-M4'</td>
<td>For the ARM Cortex M4 32-bit processors with a floating-point unit.</td>
</tr>
<tr>
<td>'ARM Cortex-A7'</td>
<td>For the ARM Cortex A7 32-bit processors (Gnueabih machine type).</td>
</tr>
<tr>
<td>'ARM Cortex-A8'</td>
<td>For the ARM Cortex A8 32-bit processors (Gnueabih machine type).</td>
</tr>
<tr>
<td>'ARM Cortex-A9'</td>
<td>For the ARM Cortex A9 32-bit processors (Gnueabih machine type).</td>
</tr>
<tr>
<td>'ARM Cortex-A53'</td>
<td>For the ARM Cortex A53 64-bit processors (Gnueabih machine type).</td>
</tr>
<tr>
<td>'ARM Cortex-A72'</td>
<td>For the ARM Cortex A72 64-bit processors (Gnueabih machine type).</td>
</tr>
<tr>
<td>'TI-Cortex-A15'</td>
<td>For the ARM Cortex A15 32-bit processors (Gnueabih machine type).</td>
</tr>
<tr>
<td>'NVIDIA-Cortex-A57'</td>
<td>For the NVIDIA Cortex A57 64-bit processors (Aarch machine type).</td>
</tr>
<tr>
<td>'AARCH-Cortex-A57'</td>
<td>For the ARM Cortex A57 64-bit processors (Aarch machine type).</td>
</tr>
<tr>
<td>'AARCH-Cortex-A72'</td>
<td>For the ARM Cortex A72 64-bit processors (Aarch machine type).</td>
</tr>
<tr>
<td>'PowerPC'</td>
<td>For 32-bit PowerPC based platforms (supports GCC compiler).</td>
</tr>
<tr>
<td>'PowerPC64'</td>
<td>For 64-bit PowerPC based platforms (supports GCC compiler).</td>
</tr>
<tr>
<td>'MinGW32'</td>
<td>For Windows x86 based 32-bit platforms (supports MinGW compiler).</td>
</tr>
<tr>
<td>'MinGW64'</td>
<td>For Windows x86_64 based 64-bit platforms (supports MinGW compiler).</td>
</tr>
<tr>
<td>'dSPACE-MABII'</td>
<td>For the dSPACE MicroAutoBox II real-time system (supports Microtec compiler).</td>
</tr>
<tr>
<td>'Speedgoat-x86'</td>
<td>For Speedgoat 32-bit real-time platforms (supports Microsoft compiler).</td>
</tr>
<tr>
<td>'Speedgoat-x64'</td>
<td>For Speedgoat 64-bit real-time platforms (supports Microsoft compiler).</td>
</tr>
<tr>
<td>'IAtomE680_Bachmann'</td>
<td>For Bachmann PLC platforms (supports VxWorks compiler).</td>
</tr>
</tbody>
</table>

**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.
9.1.13.1 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.

9.1.13.2 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
```

9.1.13.3 SIMD instructions

On x86-based platforms one can also add the following field to accelerate the execution of the solver:

```
codeoptions.sse = 1;
% or
codeoptions.avx = 1;
```

Depending on the platform AVX may be automatically enabled. If the machine running the solver does not support AVX and returns the message "Illegal Instruction" one can explicitly disable avx by setting:

```
codeoptions.avx = -1;
```

9.1.14 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.

9.1.15 Optimizing code size

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;
```
9.1.16 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.

```plaintext
codeoptions.optimize_choleskydivision = 1;
codeoptions.optimize_registers = 1;
codeoptions.optimize_uselocalsall = 1;
codeoptions.optimize_uselocalsheavy = 1; % overridden if uselocalsall is enabled
codeoptions.optimize_uselocalsimple = 1; % overridden if uselocalsheavy is enabled
codeoptions.optimize_operationsrearrange = 1;
codeoptions.optimize_looopunrolling = 1;
codeoptions.optimize_enableoffset = 1;
```

9.2 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.

9.2.1 Integrators

When providing the continuous dynamics the user must select a particular integrator by setting `nlp.integrator.type` as outlined in Table 9.5.

<table>
<thead>
<tr>
<th><code>nlp.integrator.type</code></th>
<th>Type</th>
<th>Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>'ForwardEuler'</td>
<td>Explicit Euler Method</td>
<td>1</td>
</tr>
<tr>
<td>'ERK2'</td>
<td>Explicit Runge-Kutta</td>
<td>2</td>
</tr>
<tr>
<td>'ERK3'</td>
<td>Explicit Runge-Kutta</td>
<td>3</td>
</tr>
<tr>
<td>'ERK4' (default)</td>
<td>Explicit Runge-Kutta</td>
<td>4</td>
</tr>
<tr>
<td>'BackwardEuler'</td>
<td>Implicit Euler Method</td>
<td>1</td>
</tr>
<tr>
<td>'IRK2'</td>
<td>Implicit Euler Method</td>
<td>2</td>
</tr>
<tr>
<td>'IRK4'</td>
<td>Implicit Euler Method</td>
<td>4</td>
</tr>
</tbody>
</table>
The user must also provide the discretization interval (in seconds) and the number of intermediate shooting nodes per interval. For instance:

```matlab
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.

### 9.2.2 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:

```matlab
% default tolerances
codeoptions.nlp.TolStat = 1E-5; % inf norm tol. on stationarity
codeoptions.nlp.TolEq = 1E-6; % tol. on equality constraints
codeoptions.nlp.TolIneq = 1E-6; % tol. on inequality constraints
codeoptions.nlp.TolComp = 1E-6; % tol. on complementarity
```

All tolerances are computed using the infinity norm $\|\cdot\|_\infty$.

### 9.2.3 Barrier strategy

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

```matlab
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.

### 9.2.4 Hessian approximation

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

```matlab
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.

#### 9.2.4.1 BFGS options

When the Hessian is approximated using BFGS updates, the initialization of the estimates can play a critical role in the convergence of the method. The default value is the identity matrix, but the user can modify it using e.g.:

```matlab
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.

### 9.2.4.2 Gauss-Newton options

For problems that have a least squares objective, i.e. the cost function can be expressed by a vector-valued function \( r_k : \mathbb{R}^n \rightarrow \mathbb{R}^m \) which implicitly defines the objective function as:

\[
f_k(z_k, p_k) = \frac{1}{2} \left\| r_k(z_k, p_k) \right\|^2_2,
\]

the Gauss-Newton approximation of the Hessian is given by:

\[
\nabla^2_{xx} L_k \approx \nabla r_k(z_k, p_k) \nabla r_k(z_k, p_k)^T
\]

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 \( r(z) = \sqrt{100z_1^2 + 6z_2^2} \), i.e. \( f(z) = 50z_1^2 + 3z_2^2 \), then the following code defines the least-squares objective (note that \( r \) is a vector-valued function):

```matlab
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.

### 9.2.5 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;
```

### 9.2.6 Regularization

To avoid ill-conditioned saddle point systems, FORCES employs two different types of regularization, static and dynamic regularization.
9.2.6.1 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 $\delta_w$ and $\delta_c$ vary at each iteration according to the following heuristic rule:

$$
\delta_w = \eta_w \min(\mu, \|c(x)\|)\beta_w (i + 1)^{-\gamma_w} + \delta_{w, \text{min}} \\
\delta_c = \eta_c \min(\mu, \|c(x)\|)\beta_c (i + 1)^{-\gamma_c} + \delta_{c, \text{min}}
$$

where $\mu$ 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:

```matlab
% 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 $\delta_{w, \text{min}}$ and $\delta_{c, \text{min}}$, respectively.

9.2.6.2 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 $\epsilon$, it is replaced by $\delta$. There are two parameter pairs: $(\epsilon, \delta)$ affects the augmented Hessian and $(\epsilon^2, \delta^2)$ affects the search direction computation. You can set these parameters by:

```matlab
% 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.)
```

9.2.7 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:

```matlab
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 (and also very reliable) 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_fast' 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' 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.

### 9.2.8 Safety checks

By default, the output of the function evaluations is checked for the presence of NaNs or INFs in order to diagnose potential initialization problems. In order to speed up the solver one can remove these checks by setting:

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

### 9.3 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 9.6, 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 9.6: Branch-and-bound options

<table>
<thead>
<tr>
<th>Setting</th>
<th>Values</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>mip.timeout</td>
<td>Any value ≥ 0</td>
<td>31536000 (1 year)</td>
</tr>
<tr>
<td>mip.mipgap</td>
<td>Any value ≥ 0</td>
<td>0</td>
</tr>
<tr>
<td>mip.branchon</td>
<td>'mostAmbiguous', 'leastAmbiguous'</td>
<td>'mostAmbiguous'</td>
</tr>
<tr>
<td>mip.stageinorder</td>
<td>0 (OFF), 1 (ON)</td>
<td>1 (ON)</td>
</tr>
<tr>
<td>mip.explore</td>
<td>'bestFirst', 'depthFirst'</td>
<td>'bestFirst'</td>
</tr>
<tr>
<td>mip.inttol</td>
<td>Any value &gt; 0</td>
<td>1E-5</td>
</tr>
<tr>
<td>mip.queuesize</td>
<td>Any integer value ≥ 0</td>
<td>1000</td>
</tr>
</tbody>
</table>

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.

### 9.4 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 9.7.
### Table 9.7: Solve methods

<table>
<thead>
<tr>
<th>solvemethod</th>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'PDIP' (default)</td>
<td>Primal-Dual Interior-Point Method</td>
<td>The Primal-Dual Interior-Point Method is a stable and robust method for most problems.</td>
</tr>
<tr>
<td>'ADMM'</td>
<td>Alternating Direction Methods of Multipliers</td>
<td>For some problems, ADMM may be faster. The method variant and several algorithm parameters can be tuned in order to improve performance.</td>
</tr>
<tr>
<td>'DFG'</td>
<td>Dual Fast Gradient Method</td>
<td>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.</td>
</tr>
<tr>
<td>'FG'</td>
<td>Fast Gradient Method</td>
<td>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.</td>
</tr>
</tbody>
</table>

### 9.4.1 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.

#### 9.4.1.1 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 9.8.
Table 9.8: PDIP solver initialization

<table>
<thead>
<tr>
<th>init</th>
<th>Method</th>
<th>Initialization method</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Cold start</td>
<td>Set all primal variables to 0, and all dual variables to the square root of the initial complementarity gap ( \mu_0 : z_i = 0, s_i = \sqrt{\mu_0}, \lambda_i = \sqrt{\mu_0} ). The default value is ( \mu_0 = 10^6 ).</td>
</tr>
<tr>
<td>1</td>
<td>Centered start</td>
<td>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_{ineq} \text{ and } s_i \lambda_i = \mu_0 ).</td>
</tr>
<tr>
<td>2</td>
<td>Primal warm start</td>
<td>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 ).</td>
</tr>
</tbody>
</table>

9.4.1.2 Initial Complementary Slackness

The default value for \( \mu_0 \) is \( 10^6 \). To use a different value, use:

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

9.4.1.3 Accuracy Requirements

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

```matlab
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
```

9.4.1.4 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.

```matlab
codeoptions.linesearch.factor_aff = 0.9;
codeoptions.linesearch.factor_cc = 0.95;
```

The remaining two parameters of the field `linesearch` determine the minimum \( \text{minstep} \) and maximum step size \( \text{maxstep} \). Choosing `minstep` too high will cause the generated solver to quit with an exitcode saying that the line search has failed, i.e. no progress could be made along the computed search direction. Choosing `maxstep` too close to 1 is likely to cause numerical issues, but choosing it too conservatively (too low) is likely to increase the number of iterations needed to solve a problem.

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

During factorization of supposedly positive definite matrices, FORCES Pro applies a regularization to the $i$-th pivot element if it is smaller than $\epsilon$. In this case, it is set to $\delta$, which is the lower bound on the pivot element that FORCES Pro allows to occur.

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

9.4.1.6 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

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

By default multicore computation is switched off.

9.4.2 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`:

```matlab
codeoptions.ADMMvariant = 1; % can be 1 or 2
```

where variant 1 is as follows:

```
minimize $\frac{1}{2} y^T H y + f^T y$
subject to
$D y = c$
$z \leq z \leq \bar{z}$
$y = z$
```

and variant 2 is as follows:

```
minimize $\frac{1}{2} y^T H y + f^T y$
subject to
$D y = c$
$A y = z$
$z \leq b$
```

9.4.2.1 Accuracy requirements

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

```matlab
codeoptions.accuracy.consensus = 1e-3; % infinity norm of the consensus equality
codeoptions.accuracy.dres = 1e-3; % infinity norm of the dual residual
```

Note that, in contrast to primal-dual interior-point methods, the required number of ADMM iterations varies very significantly depending on the requested accuracy. ADMM typically requires few iterations to compute medium accuracy solutions, but many more iterations to achieve the same accuracy as interior-point methods. For feedback applications, medium accuracy solutions are typically sufficient. Also note that the ADMM accuracy requirements have to be changed depending on the problem scaling.
9.4.2.2 Method parameters

ADMM uses a regularization parameter $\rho$, which also acts as the step size in the gradient step. The convergence speed of ADMM is highly variable in the parameter $\rho$. Its value should satisfy $\rho > 0$. This parameter can be tuned using the following command:

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

In some cases it may be possible to let FORCES Pro choose the value $\rho$ automatically. To enable this feature set:

```plaintext
codeoptions.ADMMautorho = 1;
```

Please note that this does not guarantee that the choice of $\rho$ 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 $\alpha$. Its value should satisfy $1 \leq \alpha \leq 2$. This step using the following command:

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

9.4.2.3 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.

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

9.4.3 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.

9.4.4 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.

9.4.4.1 Accuracy requirements

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

```plaintext
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.

### 9.4.4.2 Method parameters

The user has to determine the step size in the fast gradient method. The convergence speed of FG is highly variable in this parameter, which should typically be set to be one over the maximum eigenvalue of the quadratic cost function. This parameter can be tuned using the following command:

```matlab
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:

```matlab
codeoptions.FGautostep = 1;
```

### 9.4.4.3 Warm starting

The performance of the fast gradient method can be greatly influenced by the way the variables are initialized. Unlike with interior-point methods, fast gradient methods can be very efficiently warm started with a good guess for the optimal solution. To enable this feature set:

```matlab
codeoptions.warmstart = 1;
```

When the user turns warm start on, a new parameter $z_{\text{init},0}$ 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 10

Frequently asked questions

10.1 Quick links

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

10.2 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.

- Can I use FORCES for non-multistage programs?
  Yes, FORCES Pro supports the case $N = 1$, i.e. a general QCQP of the form

  \[
  \begin{align*}
  \text{minimize} & \quad \frac{1}{2} z^T H z + f^T z \\
  \text{subject to} & \quad Dz = c \\
  & \quad z \leq z \leq \bar{z} \\
  & \quad Az \leq b \\
  & \quad z^T Q z + q^T z \leq r
  \end{align*}
  \]

  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.

### 10.3 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 this link.

- **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

```matlab
url = URL(endpoint);
```

and replace it with

```matlab
```

- **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

```matlab
url = URL(endpoint);
```

and replace it with

```matlab
```

- **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

and replace it by the default

url = URL(endpoint);

- **I get the following error when generating code:**
  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

  \[ C_{i-1}z_{i-1} + D_1z_i = c_{i-1} \]

  instead of

  \[ C_{i-1}z_{i-1} + D_1z_i = c_i \]

- **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: ‘csma-tio.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;
  ```

### 10.4 Issues when running the solver

- **When I run the solver in MATLAB I get the following error: ‘??? Error using ==> Test-Solver 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
· **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 $\rho$ and $\alpha$ 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 segmentation fault. What is the problem?**

By default, the code is compiled with all optimizations turned on (-O3). We have observed that sometimes there are problems when linking on 32-bit versions of MATLAB. This problem does not occur when the compiler optimization flags are set to -O0, -O1, or -O2. You can change this setting using the following flag set to the appropriate value.

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

### 10.5 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.

### 10.6 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*.

### 10.7 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 $G z \leq g$ and $s$ are slack variables for the same constraints, `res_ineq` is equal to $\|G z - g + s\|_\infty$.
  
  - If $\lambda$ are the Lagrange multipliers for the inequality constraints, $\mu$ is equal to $\lambda^T s$ divided by the number of constraints, i.e. the average complementary slackness.

- **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