A code generator for fast optimization

User guide for solving nonlinear problems with FORCES Pro

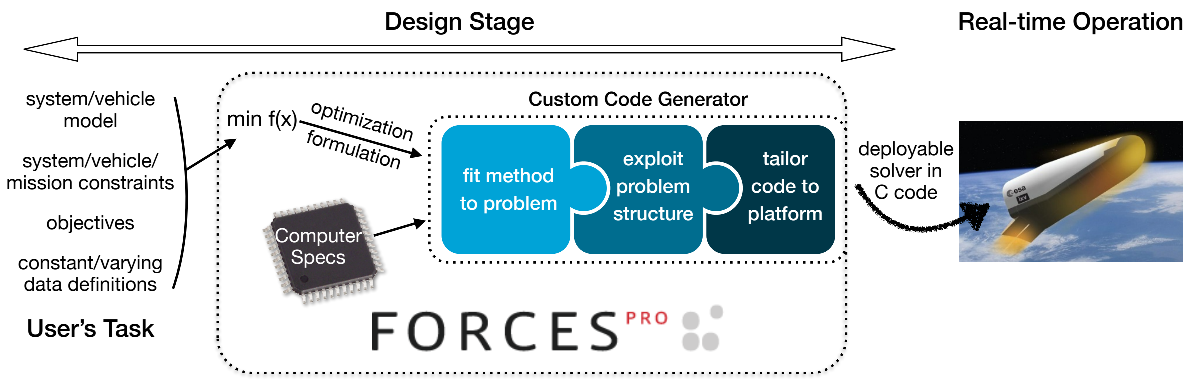
Contact us at
`support@embotech.com`

April 10, 2017

Contents

1 Introduction

This document gives an introduction to the nonlinear interface of 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 wants 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 FORCES nonlinear interface

FORCES Pro has several interfaces with different capabilities and flexibility. This document focuses on the MATLAB® text interface for describing nonlinear optimization problems. The interface can also be used for describing convex optimization problems.

1.2 Troubleshooting and support

FORCES Pro typically returns meaningful error messages when code generation errors occur due to invalid user inputs. When encountering other errors the first place to check is the 'Frequently Asked Questions (FAQ)' section of the website. This page has

Table 1: Available FORCES Pro licenses

Simulation	For generating compiled code that can run on your desktop platform
Prototyping	For generating compiled code that can run on an embedded target (x86, ARM, PowerPC, Tricore, or SPARC)
Deployment	For generating source code that can run on any embedded target
Enterprise	For running the code generation system inside an institution's network

solutions to the most commonly occuring problems. 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.

1.3 Licensing

FORCES Pro licenses are available as a yearly subscription or as a one-off payment with optional maintenance. There are four types of licenses with the following functionality:

1.4 Academic licensing

Users at degree granting institutions can have access to the SIMULATION version of FORCES Pro free of charge provided they are not doing research for an industrial partner. PROTOTYPING licenses are also available at highly reduced rates.

1.5 Citing FORCES

If you use FORCES Pro in published scientific work, please cite the following:

```
@misc{FORCESPro,
Author = {Alexander Domahidi and Juan Jerez},
Howpublished = {{embotech GmbH
(\nobreak{\url{http://embotech.com/FORCES-Pro}})}},
Month = jul,
Title = {{FORCES Professional}},
Year = 2014}
}
```

2 Installing FORCES Pro

FORCES Pro is a client-server code generation system. The user describes the optimization problem using the client software, which communicates with the server for code generation (and compilation if applicable).

The client software is the same for all users, independent of their license type. It can be freely downloaded from

`https://www.embotech.com/FORCES-Pro/Download`

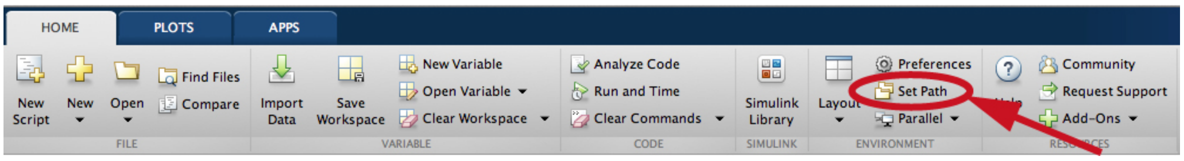
following a registration on the embotech website. You can register as a new user here:

`https://www.embotech.com/Register`

2.1 Installing FORCES for MATLAB®

Add the path of the downloaded folder `FORCES_PRO` to the MATLAB® path by either using the command `addpath DIRNAME`, e. g. `addpath /home/user/FORCES_PRO` in the command window.

Alternatively, set the path to the `FORCES_PRO` folder via the graphical user interface of MATLAB®.



2.2 Installing FORCES for Python

Add the path of the downloaded folder `FORCES_PRO` to the Python path. In UNIX systems this can be accomplished by adding the following commands to your `.bash_profile` or `.bashrc` configuration files:

```
PYTHONPATH="/home/user/FORCES_PRO:${PYTHONPATH}"
export PYTHONPATH
```

Alternatively, set the path in your Python application or script by adding:

```
import sys
sys.path.insert(0, "/home/user/FORCES_PRO")
```

2.3 System requirements

FORCES Pro is supported on Windows, Mac OSX 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

`http://ch.mathworks.com/support/compilers/R2014a/`

Run `mex -setup` to configure your C compiler in MATLAB®.

For the Python interface, we support Python 2.7 and Python 3.4. If you need a client for a different Python 3 version please contact us at support@embotech.com. Additionally, the following Python packages are required:

- `scipy`
  An open-source Python library for mathematics, science, and engineering. The package can be downloaded from `http://www.scipy.org`.
- `numpy`
  The fundamental package for scientific computing with Python. The package can be downloaded from `http://www.numpy.org`.
- `suds`
  A lightweight SOAP client. The package can be downloaded from `https://pypi.python.org/pypi/suds`. For Python 3, please use `suds-jurko` instead, which can be obtained from `https://pypi.python.org/pypi/suds-jurko`.

If you have pip installed on your system you can install these packages by typing the following on your command prompt:

```
pip install package_name
```

To install all three packages:

```
pip install numpy
pip install scipy
pip install suds-jurko
```

2.4 Keeping your software 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®/Python command prompt. `updateClient` without any arguments uses the default embotech server to grab the client, and updates all corresponding client files. The command

```
>> updateClient(URL)
```

overrides the default server selection and uses the server located at `URL` instead. Alternatively, you can always download the most recent version from

`http://www.embotech.com/FORCES-Pro/Download`

and install it over the existing client.

2.5 Terms of use

By downloading the software you accept the software license agreement set forth in:

- For industrial users:

  `https://www.embotech.com/FORCES-Pro/License-Terms/Commercial`

- For acadmic users:

  `https://www.embotech.com/FORCES-Pro/License-Terms/Academic`

3 Supported Problem Formulation

3.1 Canonical problem for discrete-time dynamics

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

$$
\begin{aligned}
\text{minimize} \quad & \sum_{k=1}^{N} f_k(z_k, p_k) && && \text{(1a)}\\
\text{subject to} \quad & z_1(\mathcal{I}) = z_{\text{init}}, && && \text{(1b)}\\
& E_k z_{k+1} = c_k(z_k, p_k), && k = 1, \dots, N-1 && \text{(1c)}\\
& z_N(\mathcal{N}) = z_{\text{final}}, && && \text{(1d)}\\
& \underline{z}_k \le z_k \le \bar{z}_k, && k = 1, \dots, N && \text{(1e)}\\
& \underline{h}_k \le h_k(z_k, p_k) \le \bar{h}_k, && k = 1, \dots, N && \text{(1f)}
\end{aligned}
$$

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 $c_k : \mathbb{R}^{n_k} \to \mathbb{R}^{w_k}$ represent (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_{init} and z_{final} can also be changed in real-time.

All real-time data is coloured in blue.

3.2 Continuous-time dynamics

In (1), instead of having discrete-time dynamics as in (1c), we also support using continuous-time dynamics of the form

$$\dot{x} = f(x, u, p) \tag{2}$$

and then discretizing this equation by one of the standard integration methods. The methods offered by FORCES Pro are outlined in Section 4.1.7.

3.3 Other variants

Not all elements in (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 $\underline{z}_{k,i} \equiv -\infty$ ($\bar{z}_{k,i} \equiv +\infty$);
- without nonlinear inequalities h_k;
- with $N = 1$ (single stage problem), then the equality constraints (1c) can be omitted;
- that optimize over the initial value z_{init} and do not include the initial constraint (1b);
- that optimize over the final value z_{final} and do not include the final constraint (1d).

3.4 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 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[1]. 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 functions (see below). This is the recommended way of getting started with FORCES NLP (if CasADi is installed on the user's machine). See Section 4 to learn how to use this approach.
2. C-functions (source files). These can be hand-coded, or generated by any automatic differentiation tool. We detail in Section 5 on how to provide own function evaluations and derivatives to FORCES Pro.

[1] `www.casadi.org`

4 The easiest way of using FORCES (requires CasADi)

When solving nonlinear programs of type (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 (`www.casadi.org`), 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 to learn how to provide your own derivatives to FORCES NLP solvers. This section will outline the MATLAB®-CasADi approach in detail.

4.1 Guiding example: Problem formulation

The simplicity of the user interface of the FORCES NLP solver is illustrated on a vehicle maneuver optimal control problem (OCP) example. In particular, we use a simple vehicle model described by a set of ordinary differential equations (ODEs):

$$\begin{aligned} \dot{x} &= v\cos(\theta) \\ \dot{y} &= v\sin(\theta) \\ \dot{v} &= F/m \\ \dot{\theta} &= s/I. \end{aligned} \tag{3}$$

The model consists of $n_x = 4$ differential states: x and y are the Cartesian coordinates of the car, v is the linear velocity and θ is the heading angle. Next, there are $n_u = 2$ control inputs to the model: the acceleration force F and the steering torque s. The car mass is $m = 1\,\mathrm{kg}$ and its moment of inertia is $I = 1\,\mathrm{kgm}^2$.

The maneuver is defined as an NLP and is defined and coded in MATLAB® code as follows. First, we define the stage variable z by stacking input and differential state variables

$$z := [F, s, x, y, v, \theta]^T. \tag{4}$$

4.1.1 Objective

In this simple example, all stage cost functions $f_k(.)$ are equal and defined as[2]

$$f_k(z) = -100y + 0.1F^2 + 0.01s^2. \tag{5}$$

This stage cost function can be coded in MATLAB® using the following function:

[2]For the sake of brevity we omit the sub-indices in the objective and constraint functions. Moreover, we omit dimensions of the weighting coefficients in the objective for notational convenience.

```
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
```

In this example, we want to maximize position in y direction, with quadratic penalties on the inputs F and s.

4.1.2 Vector-valued equality constraints

The vector-valued equality constraints (1c) in this example represent the discretized dynamic equations of the car using an explicit Runge-Kutta integrator of order 4. The car dynamics defined in (3) are represented by the function `continuous_dynamics` and the constraint function $c(.)$ as the function `dynamics`. Note that `RK4` is an auxiliary function included in the FORCES Pro client software. To learn more about integrators in FORCES Pro, cf. Section 4.1.7.

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

function xnext = dynamics( z )
  x = zeros(4,1);
  x = z(1: 4);
  u = zeros(2,1);
  u = z(5: 6);
  % implements a RK4 integrator for the dynamics.
  h = 0.1; % step size
  xnext = RK4(x, u, @continuous_dynamics, h);
end
```

In our example, the matrices $E_k = [0_{n_x \times n_u}\ I_{n_x \times n_x}]$ select the last four components of the stage variable z, see (1c). While general selection matrices E_k is supported, it is highly advised to use the aforementioned form for performance reasons.

4.1.3 Real-time data

If we want to vary certain parameters in our functions after code generation, for example the mass of the vehicle, we can declare these quantities as real-time data. In this case we add an extra argument – called p, the parameter vector in this document – to our functions:

```
function xdot = continuous_dynamics ( x, u, p )
  F = u( 1 );
  s = u( 2 );
  v = x( 3 );
  theta = x( 4 );
  m = p(1); % car mass
  I = 1; % car inertia
  xdot = [v * cos( theta );
          v * sin( theta );
          F / m; % mass is supplied in real-time
          s / I];
end
```

```
function xnext = dynamics( z, p )
  x = zeros(4,1);
  x = z(1:4);
  u = zeros(2,1);
  u = z(5:6);
  % implements a RK4 integrator for the dynamics.
  h = 0.1; % step size
  xnext = RK4(x, u, @continuous_dynamics, h, p);
end
```

Note that we also had to pass the parameter to the function `RK4`.

Please check the obstacle avoidance example in the 'examples' folder that comes with your client for an illustration of how to use real-time parameters in the high-level interface.

4.1.4 Inequality constraints

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

$$\begin{aligned} -5\,\mathrm{N} &\le F \le 5\,\mathrm{N}, \\ -1\,\mathrm{Nm} &\le s \le 1\,\mathrm{Nm}, \\ -3\,\mathrm{m} &\le x \le 0\,\mathrm{m}, \\ 0\,\mathrm{m} &\le y \le 3\,\mathrm{m}, \\ 0\,\mathrm{m/s} &\le v \le 2\,\mathrm{m/s}, \\ 0\,\mathrm{rad} &\le \theta \le \pi\,\mathrm{rad}, \end{aligned} \tag{6}$$

as well the nonlinear nonconvex constraints

$$\begin{aligned} 9\,\text{m}^2 &\leq x^2 + y^2, \\ 1\,\text{m}^2 &\leq (x+2)^2 + (y-2.5)^2 \leq 0.9025\,\text{m}^2. \end{aligned} \tag{7}$$

In this case, the nonlinear constraint functions $h_k(.)$ are equal for all stages and can be coded in MATLAB® as follows:

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

4.1.5 Initial and final conditions

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

$$x_{\text{init}} = -2\,\text{m}, \quad y_{\text{init}} = 0\,\text{m}, \quad v_{\text{init}} = 0\,\text{m/s}, \quad \theta_{\text{init}} = 2.0944\,\text{rad} \tag{8}$$

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

$$v_{\text{final}} = 0\,\text{m/s}, \quad \theta_{\text{final}} = 0\,\text{rad}. \tag{9}$$

4.1.6 Complete problem formulation in MATLAB®

With the above defined MATLAB®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.

```
%% NLP relevant dimensions
nlp.N = 50;      % horizon length
nlp.nvar = 6;    % number of stage variables
nlp.neq  = 4;    % number of equality constraints
nlp.nh = 2;      % number of inequality constraints
nlp.npar = 0;    % number of runtime parameters

%% Objective function
nlp.objective = @objective;

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

%% Inequality constraints
% upper/lower bounds lb <= x <= ub
%           F    s   x     y     v    theta
nlp.lb = [-5,  -1, -3,    0     0     0 ];
nlp.ub = [+5,  +1,  0,    3     2    +pi];

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

%% Initial and final conditions
nlp.xinitidx = 3:6;
nlp.xfinalidx = 5:6;
```

4.1.7 Integrators for continuous dynamics

In the example above we have provided the discretized equality constraints directly using the field `nlp.eq`. For problems including continuous dynamics it is possible to instead provide just the ordinary differential equations using

```
nlp.continuous_dynamics = @dynamics;
```

The function describing the continuous dynamics must have the following arguments in the right order:

- states
- inputs
- parameters (optional)

You also have to specify the left-hand side matrix `nlp.E`. For performance reasons it is best to give it as

```
nlp.E = [zeros(nx, nu), eye( nx )];
```

where `nx` is the number of states and `nu` the number of inputs. In other words, the stage variable `z` first contains the inputs `u` and then the states `x`. The same ordering has to be used in the other functions handles, e.g. `nlp.objective` or `nlp.ineq`, that use the entire stage-variables `z` instead of separate states and inputs.

The user can choose between several explicit and implicit integrators that provide a trade off between accuracy and speed. See Section 6.7 to learn how to select integrators and set the discretization interval and various other options.

4.1.8 Varying dimensions, parameters, constraints, or functions for different stages

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 variables F and s from the last stage one could write the following:

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

Shorthand notations It is 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 in the equalities (1c)
- `nhN`: number of inequalities in last stage
- `ineqN`: inequalities for last stage

Add any of these fields to the model struct (`nlp` in the example above) 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:

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

4.1.9 Providing analytic derivatives

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

Note that the user must be particularly careful to make sure that the provide functions and derivatives are consistent, e.g.

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

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

4.2 Solver generation

In addition to the definition of the NLP, solver generation requires an (optional) set of options for customization (for more information we refer to Section 6). At this point we instantiate the default solver options and generate a solver using

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

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

4.2.1 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 command:

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

4.3 Solver execution

After code generation has been successful, one can obtain information about the real-time data needed to call the generated solver by typing

```
help FORCESNLPsolver
```

Table 2: FORCES Pro exitflags for nonlinear solvers

Exitflag	Description
1	Local optimal solution found (i.e. the point satisfies the KKT optimality conditions to the requested accuracy)
0	Maximum number of iterations reached. You can examine the value of optimality conditions returned by FORCES to decide whether the point returned is acceptable.
-6	NaN or INF occured during evaluation of functions and derivatives. If this occurs at iteration zero, try changing the initial point. For example, for a cost function $1/\sqrt{x}$ with an initialization $x^0 = 0$, this error would occur.
-7	The solver could not proceed. Most likely cause is that the problem is infeasible. Try formulating a problem with slack variables (soft constraints) to avoid this error.
-100	License error. This typically happens if you are trying to execute code that has been generated with a Simulation license of FORCES Pro on another machine. Regenerate the solver using your machine.

The `PARAMS` section describes the number of fields that solver is expecting and their sizes. The following subsections give further details for commonly occurring runtime fields.

4.3.1 Initial guess

The FORCES NLP solver solves NLPs to local optimality, hence the resulting optimal solution depends on the initialization of the solver. The following code sets the initial point to be in the middle of all bounds. One can also choose another initialization point when a better guess is available.

```
x0i=nlp.lb+(nlp.ub-nlp.lb)/2;
x0=repmat(x0i', nlp.N,1);
params.x0=x0;
```

4.3.2 Initial and final conditions

If there are initial and/or final conditions on the optimization variables, the solver will expect the corresponding runtime fields:

```
params.xinit = nlp.xinit;
params.xfinal = nlp.xfinal;
```

4.3.3 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

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

4.3.4 Exitflags and quality of the result

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

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

The possible exitflags are documented in Table 2. 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:

```
assert(exitflag == 1,'Some problem in FORCES solver');
```

5 External function evaluations

This approach allows the user to integrate existing efficient C implementations of the required functions or to use other automatic differentiation tools. 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.

5.1 Interface

5.1.1 Array of parameters

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

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

The name of the function can be changed but it should have the same name as the C file containing it, i.e. in this case this function should be included in `myfunctions.c`.

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

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

This will change the interface of the expected external function to

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

i.e. you can pass arbitrary data structures to your own function by setting the pointer in the `params` struct:

```
myData p; /* define your own parameter structure */
...       /* fill it with data */

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

/* Call solver (assuming everything else is defined) */
mySolver_solve(&params, &output, &info, stdout, &external_func);
```

Note: Using the option `customParams = 1` will disable building high-level interfaces such as for Python, MATLAB® and Simulink®. Only C header- and source files will be generated.

5.2 Rules for function evaluation code

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

```
    /* cost */
    if (f)
    {   /* notice the increment of f */
        (*f) += -100*x[3] + 0.1*x[0]*x[0] + 0.01*x[1]*x[1];
    }

    /* gradient - only nonzero elements have to be filled in */
    if (nabla_f)
    {
        nabla_f[0] = 0.2*x[0];
        nabla_f[1] = 0.02*x[1];
        nabla_f[3] = -100;
    }

    /* eq constr */
    if (c)
    {
        car_dyanmics(x, c);
    }

    /* jacobian equalities (column major) */
    if (nabla_c)
    {
        car_dyanmics_jacobian(x, nabla_c);
    }

    /* ineq constr */
    if (h)
    {
        h[0] = x[2]*x[2] + x[3]*x[3];
        h[1] = (x[2]+2)*(x[2]+2) + (x[3]-2.5)*(x[3]-2.5);
    }

    /* jacobian inequalities (column major)
     - only non-zero elements to be filled in */
    if (nabla_h)
    {
        /* column 3 */
        nabla_h[4] = 2*x[2];
        nabla_h[5] = 2*x[2] + 4;

        /* column 4 */
        nabla_h[6] = 2*x[3];
        nabla_h[7] = 2*x[3] - 5;
    }
```

Notice that every function evaluation is only carried out if the corresponding pointer is not null. This is used by FORCES NLP to call the same interface with different pointers depending on the functions that it requires.

5.3 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.

5.4 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);
```

5.5 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 parameter `stage` to evaluate different functions depending on the stage number. The same applies to all other quantities.

5.6 Other arguments

Finally, notice that the prototype function has several unused arguments. These will be used in future extensions.

5.7 Supplying function evaluation information

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

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

If there are any additional C files that are used by the main C file defined above use a string with spaces separating the different files.

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

6 Solver Options

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

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

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

The FORCES NLP solver implements a nonlinear barrier interior-point method. We will first discuss how to change several parameters in the solver.

6.1 Accuracy requirements

One can modify the termination criteria by altering the KKT tolerance with respect to stationarity, equality constraints, inequality constraints and complementarity conditions, respectively, using the following fields:

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

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

6.2 Barrier strategy

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

```
codeoptions.nlp.BarrStrat = 'loqo';
```

It can be set to `'loqo'` (default) or to `'monotone'`. The default settings often leads to faster convergence, while `'monotone'` may help convergence for difficult problems.

6.3 Hessian approximation

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

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

The current version supports BFGS updates (`'bfgs'`) (default) and Gauss-Newton approximation (`'gauss-newton'`). Exact Hessians will be supported in a future version.

6.3.1 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.

```
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.

6.3.2 Gauss-Newton options

For problems that have a least squares objective, i.e. the cost function can be expressed by a vector-valued function $r_k : \mathbb{R}^n \to \mathbb{R}$ by

$$f_k(z_k, p_k) := \frac{1}{2}\|r_k(z_k, p_k)\|_2^2 \ ,$$

a Gauss-Newton approximation of the Hessian, which is defined by The Gauss-Newton approximation is given by

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

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,

```
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)];
```

! Note that 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.

6.4 Linear system solver

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

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

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

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

It is recommended to try different linear solvers when experiencing convergence problems. The most stable method is `'symm_indefinite'`, while the fastest solver (and also very reliable!) is `'symm_indefinite_fast'`.

Independent of the linear system solver choice, the generated code is always library-free and statically allocated, i.e. it can be embedded anywhere.

6.5 Regularization

To avoid ill-conditioned saddle point systems, FORCES employs two different types of regularization, static and dynamic regularization.

6.5.1 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 δ_w and δ_c vary at each iteration according to the following heuristic rule:

$$\delta_w = \eta_w \cdot \min(\mu, \|c(x)\|)^{\beta_w} \cdot (i+1)^{-\gamma_w} + \delta_{w,\min} \tag{10a}$$

$$\delta_c = \eta_c \cdot \min(\mu, \|c(x)\|)^{\beta_c} \cdot (i+1)^{-\gamma_c} + \delta_{c,\min} \tag{10b}$$

where μ is the barrier parameter and i is the number of iterations. This encourages low regularizations close to the solution, whereas higher ones are applied if further away. You can change these parameters by using the following settings:

```
% default static regularization parameters
codeoptions.nlp.reg_eta_dw = 1E-4;
codeoptions.nlp.reg_beta_dw = 0.8;
codeoptions.nlp.reg_min_dw = 1E-9;
codeoptions.nlp.reg_gamma_dw = 1.0/3.0;

codeoptions.nlp.reg_eta_dc = 1E-4;
codeoptions.nlp.reg_beta_dc = 0.8;
codeoptions.nlp.reg_min_dc = 1E-9;
codeoptions.nlp.reg_gamma_dc = 1.0/3.0;
```

Note that by choosing $\delta_w = 0$ and $\delta_c = 0$, you can turn off the progress and iteration dependent regularization, and rely on a completely static regularization by $\delta_{w,\text{min}}$ and $\delta_{c,\text{min}}$, respectively.

6.5.2 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 ϵ, it is replaced by δ. There are two parameter pairs: (ϵ, δ) affects the augmented Hessian and (ϵ_2, δ_2) affects the search direction computation. You can set these parameters by

```
% default dynamic regularization parameters
codeoptions.regularize.epsilon = 1E-12;  % (for Hessian approx.)
codeoptions.regularize.delta = 4E-6;     % (for Hessian approx.)
codeoptions.regularize.epsilon2 = 1E-14; % (for Normal eqs.)
codeoptions.regularize.delta2 = 1E-14;   % (for Normal eqs.)
```

6.6 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;
```

6.7 Integrators

When providing the continuous dynamics the user must select a particular integrator using the field `nlp.integrator.type`, which can be set to:

- `'ForwardEuler'` — forward / explicit Euler method
- `'ERK2'` — explicit second-order Runge-Kutta method
- `'ERK3'` — explicit third-order Runge-Kutta method

- ‘ERK4’ — explicit fourth-order Runge-Kutta method
- ‘BackwardEuler’ — backward / implicit Euler method
- ‘IRK2’ — implicit second-order Runge-Kutta method
- ‘IRK4’ — implicit fourth-order Runge-Kutta method

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

```
codeoptions.nlp.integrator.Ts = 0.01;
codeoptions.nlp.integrator.nodes = 10;
```

6.8 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

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

6.9 Maximum number of iterations

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

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

6.10 Print level

To control the amount of information the generated solver prints to the console, set the settings field `printlevel` to:

- 0 - no output will be printed
- 1 - a summary line will be printed after each full solve
- 2 - a summary line will be printed after each iteration of solver

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

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

6.11 Compiler optimization level

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

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

! It is recommended to set optlevel to zero 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.

6.12 Computation time

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

```
codeoptions.timing = 1;
```

By default the wall clock 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 zero.

! Note that 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.

6.13 Datatypes

The datatype can be changed from double precision ('double') to single precision ('float') by setting the field floattype.

! 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).

6.14 Overwriting old solvers

When a new solver is generated with the same name as an exisitng solver one can set the overwriting behaviour by setting the field overwrite to

- 0 - never overwrite
- 1 - always overwrite
- 2 - ask to overwrite

Table 3: Target platforms supported by FORCES Pro

`'Generic'`	for the architecture of the host platform (default)
`'x86_64'`	for x86 based 64-bit platforms, the same operating system as the host will be assumed when requesting compiled code
`'x86'`	for x86 based 32-bit platforms, the same operating system as the host will be assumed when requesting compiled code
`'ARM Cortex-M3'`	for ARM Cortex M3 32-bit processors
`'ARM Cortex-M4 (NO FPU)'`	for ARM Cortex M4 32-bit processors without a floating-point unit
`'ARM Cortex-M4 (FPU)'`	for ARM Cortex M4 32-bit processors with a floating-point unit
`'ARM Cortex-A'`	for ARM Cortex A processors
`'Tricore'`	for Tricore 32-bit processors
`'PowerPC'`	for 32-bit PowerPC processors
`'PowerPC64'`	for 64-bit PowerPC processors
`'MIPS'`	for 32-bit MIPS processors
`'MIPS64'`	for 64-bit MIPS processors

6.15 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` as follows:

```
codeoptions.platform = 'x86';
```

The platforms currently supported by FORCES Pro are given in Table 3.

> Note that 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.

6.15.1 Cross compilation

For x86-based platforms, one can set the target operating system by setting the appropriate flag to 1 and 0

```
codeoptions.win = 0;
codeoptions.mac = 0;
codeoptions.gnu = 1;
codeoptions.bit64 = 1;
```

Note that the automatic building of interfaces will likely fail, but this allows for obtaining a solver compiled for Linux on a Windows machine, for example.

6.15.2 SIMD instructions

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

```
codeoptions.sse = 1
```

6.16 Code generation server

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

```
codeoptions.server = 'http://ownforces.company.com:8114/v1.5';
```

6.17 Cleanup of files

6.17.1 Prevent automatic cleanup after codegen

FORCES Pro automatically cleans up some of the files that it generates during the code generation, but which are usually not needed any more after building the MEX file. In particular, some intermediate CasADi generated files are deleted. If you would like to prevent any cleanup by FORCES, set the option

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

The default value is 1 (true).

! 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.

6.17.2 Manual cleanup

If you want to get rid of the files related to a solver, you can use the `FORCEScleanup` script:

```
FORCEScleanup(solvername)
```

This cleans up all files related to a specific solver. If you want to delete *all* files related to a specific solver and any auxiliary files created by FORCES Pro, use

```
FORCEScleanup(solvername, 'all')
```

7 Some Performance Results

The example provided with the software consists of a vehicle with nonlinear dynamics trying to maximize progress on a non-convex space while minimizing the input energy for the maneouver. Figure 1 describes the computed trajectory for the vehicle.

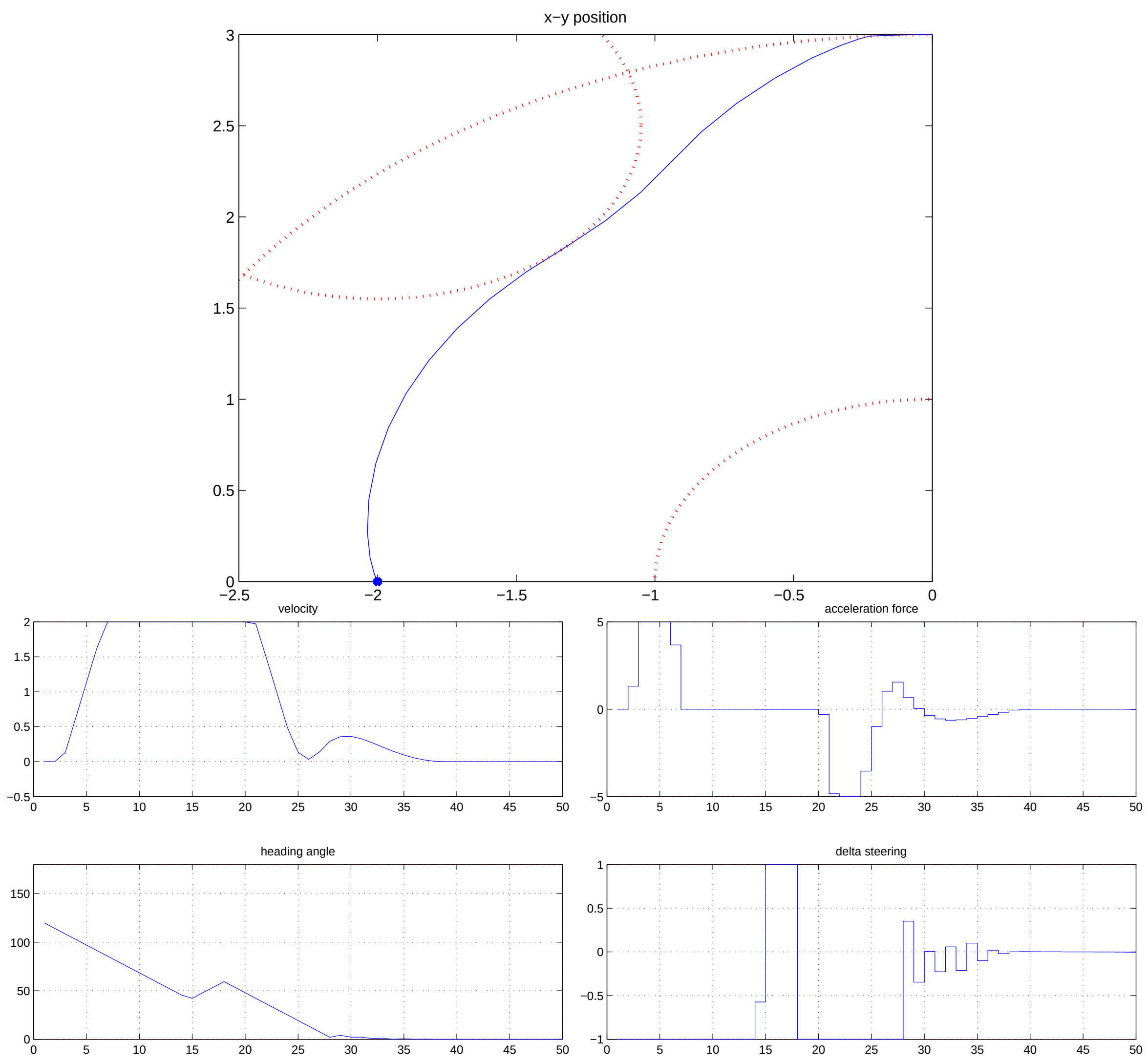


Figure 1: Trajectory of the vehicle on a 2D space (top), vehicle's velocity and heading angle (left), and actuators (right). Constraints are represented with dotted red lines. The vehicle starts at position (-2,0) and should maximize progress in y direction, while being at standstill and at heading angle of zero degress at its final position after 50 integration steps.

The following table compares the computation time with respect to IPOPT. The experiments were carried out on an Intel Core i7 at 3.4 GHz with 16GB of memory running Windows 7 64-bit. Printing was turned off both for IPOPT and for the FORCES NLP solver. The IPOPT timings do not include the time spent on external function evaluations, which in this case were done in MATLAB®.

	Total solution time	Iterations
FORCES NLP - BFGS	**6.7 ms**	76
IPOPT* - BFGS	171 ms	84
IPOPT* - exact Hessian	115 ms	74

* Reported IPOPT times do not include external function evaluations, hence it is an underestimate of the total solution time.

Notice that the current version of the FORCES NLP solver is more than one order of magnitude faster than the fastest version of IPOPT, even though IPOPT timings do not include the time spent on external function evaluations. We expect further efficiency gains in future releases.