ACADO Toolkit **User's Manual**

Version 1.0.2613beta (June 2011)

Boris Houska, Hans Joachim Ferreau et al.[1]
Optimization in Engineering Center (OPTEC) and
Department of Electrical Engineering, K. U. Leuven
support@acadotoolkit.org

[1] ACADO Toolkit developers in alphabetical order: David Ariens, Moritz Diehl, Hans Joachim Ferreau, Boris Houska, Filip Logist, Milan Vukov

Contents

Part I

Getting Started

Chapter 1

Introduction

1.1 What is the ACADO Toolkit

ACADO Toolkit is a software environment and algorithm collection written in C++ for automatic control and dynamic optimization. It provides a general framework for using a great variety of algorithms for direct optimal control, including model predictive control as well as state and parameter estimation. It also provides (stand-alone) efficiently implemented Runge-Kutta and BDF integrators for the simulation of ODE's and DAE's.

ACADO Toolkit is designed to meet these four key properties [**?**]:

- *Open-source*: The toolkit is freely available and is distributed under the GNU Lesser General Public Licence (LGPL). The latest release together with documentation and examples can be downloaded at `http://www.acadotoolkit.org`.
- *User-friendliness*: The syntax of ACADO Toolkit has been designed to be as intuitive as possible close in order to allow the user to formulate control problems in a way that is very close to the usual mathematical syntax. Moreover, the syntax of ACADO for Matlab should feel familiar to both MATLAB users and ACADO Toolkit users.
- *Code extensibility*: It should be easy to link existing algorithms to the toolkit. This is realized by the object-oriented software design of the ACADO Toolkit.
- *Self-containedness*: The ACADO Toolkit is written in a completely self-contained manner. No external packages are required, but external solvers or packages for graphical output can be linked.

More information about the ACADO Toolkit is available in [12, 2].

1.2 Problem Classes

This chapter describes the four problem classes supported by the current version of the ACADO Toolkit:

1. *Optimal control problems* are off-line dynamic optimization problems. These problems aim at calculating open-loop control inputs that minimize a given objective functional while respecting given constraints.

2. *Multi-objective optimisation and optimal control* problems, which require the simultaneous minimisation of more than one objective. These multi-objective optimisation problems typically result in a set of Pareto optimal solutions instead of one single (local) optimum.

3. *Parameter and state estimation* problems, where parameters, unknown control inputs or initial states are to be identified by measuring an output of a given (nonlinear) dynamic system.

4. *Model predictive control* problems and online state estimation, where parameterised dynamic optimisation problems have to be solved repeatedly to obtain a dynamic feedback control law.

1.2.1 Optimal Control Problems

The ACADO Toolkit can deal with optimal control problems of the following form:

$$\boxed{\begin{array}{lrcl} \underset{x(\cdot),z(\cdot),u(\cdot),p,T}{\text{minimize}} & & & \Phi[x(\cdot),z(\cdot),u(\cdot),p,T] \\ \text{subject to:} & & & \\ \forall t \in [t_0,T]: & 0 & = & f(t,\dot{x}(t),x(t),z(t),u(t),p,T) \\ & 0 & = & r(x(0),z(0),x(T),z(T),p,T) \\ \forall t \in [t_0,T]: & 0 & \geq & s(t,x(t),z(t),u(t),p,T) \end{array}} \quad \text{(OCP)}$$

with Φ typically a Bolza functional of the form:

$$\Phi[x(\cdot),z(\cdot),u(\cdot),p,T] = \int_{t_0}^{T} L(\tau,x(\tau),z(\tau),u(\tau),p),T)\,\mathrm{d}\tau \;+\; M(x(T),p,T)\;. \quad (1.1)$$

The right-hand side function f should be smooth or at least sufficiently often differentiable. Moreover, we assume that the function $\frac{\partial f}{\partial(\dot{x},z)}$ is always regular, i.e. the index of the DAE should be one. The remaining functions, namely the Lagrange term L, the Mayer term M, the boundary constraint function r, as well the path constraint function s are assumed to be at least twice continuously differentiable in all their arguments. For discretization single and multiple shooting algorithms are implemented.

1.2.2 Multi-objective Optimisation and Optimal Control Problems

In contrast to the general optimal control problem formulation, in which only one objective has to be minimized, the general MOOCP formulation requires the simultaneous minimiza-

tion of m objectives:

$$
\begin{array}{lll}
\underset{x(\cdot),u(\cdot),p,T}{\text{minimize}} & \{\Phi_1(x(\cdot),u(\cdot),p,T),\ldots,\Phi_j(x(\cdot),u(\cdot),p,T),\ldots,\Phi_m(x(\cdot),u(\cdot),p,T)\} & \\
\text{subject to:} & & \\
\forall t\in[0,T]: & 0 \;=\; F(t,x(t),\dot{x}(t),u(t),p) & \\
\forall t\in[0,T]: & 0 \;\leq\; h(t,x(t),u(t),p) & \\
& 0 \;=\; r(x(0),x(T),p) &
\end{array}
\tag{1.2}
$$

where Φ_j denotes the j-th individual objective functional. Typically, these Multi-Objective Optimal Control Problems (MOOCPs) give rise to a set of Pareto optimal solutions instead of one single optimum.

1.2.3 Parameter and State Estimation

A special class of optimal control problems is state and parameter estimation. The formulation takes the same form of the optimal control formultation (OCP) with Φ now equal to

$$
\Phi[x(\cdot),z(\cdot),u(\cdot),p,T] \;=\; \sum_{i=0}^{N} \| \, h_i(t_i,x(t_i),z(t_i),u(t_i),p)-\eta_i\|^2_{S_i} \; . \tag{1.3}
$$

Estimation problems are thus optimization problems with a least squares objective. Here, h is called a measurement function while $\eta_1,\ldots,\eta_N$ are the measurements taken at the time points $t_1,\ldots,t_N\in[0,T]$. Note that the least-squares term is in this formulation weighted with positive semi-definite weighting matrices $S_1,\ldots,S_N$, which are typically the inverses of the variance covariance matrices associated with the measurement errors.
This type of optimization problem arises in applications like:

- on-line estimation for process control,
- function approximation,
- weather forecast (weather data reconciliation),
- orbit determination.

1.2.4 Model Based Feedback Control

The MPC problem is a special case of an (OCP) for which the objective takes typically the form:

$$
\begin{aligned}
&\Phi[x(\cdot),z(\cdot),u(\cdot),p,T] = \\
&\qquad \int_{t_0}^{T} \|y(t,x(t),z(t),u(t),p)-y_{\mathsf{ref}}\|^2_S + \|x(T)-x_{\mathsf{ref}}(T)\|^2_P \; .
\end{aligned}
\tag{1.4}
$$

Therein, x_{ref} and y_{ref} are tracking reference trajectories for the states and the output function y, respectively. The matrices S and P are positive semi-definite weighting matrices

with appropriate dimensions. In contrast to OCPs, MPC problems are usually assumed to be formulated on a fixed horizon T and employing the above tracking objective function. An MPC controller performs the following steps:

1. At each timestep t the future outputs on a determined horizon N are predicted. This prediction $y(t+t_k), k = 1 \dots N$ uses the process model f and depends on the past outputs and inputs and on the future control signals $u(t+t_k), k = 0 \dots N-1$.

2. These future control signals $u(t+t_k), k = 0 \dots N-1$ are calculated in an optimization algorithm which aims to track a certain reference trajectory.

3. The control signal $u(t)$ on instant t is sent to the process. At the next sampling instant step 1 is repeated (and thus the calculated controls $u(t+t_k), k = 1 \dots N-1$ are never sent to the process).

We refer to [6] for an in-depth study of MPC.

1.3 What is ACADO for Matlab

ACADO for Matlab is a MATLAB interface for ACADO Toolkit. It brings the ACADO Integrators and algorithms for direct optimal control, model predictive control and parameter estimation to MATLAB. ACADO for Matlab uses the ACADO Toolkit C++ code base and implements methods to communicate with this code base. It is thus important to note that in the interface no new algorithms are implemented.
The key properties of ACADO for Matlab are:

- *Same key properties as ACADO Toolkit* : The ACADO for Matlab is distributed under the same GNU Lesser General Public Licence and is available at `http://www.acadotoolkit.org/matlab`. The code is easily extendible to meet future demands and is also written in a self-contained manner. No external MATLAB packages (for example the Symbolic Toolbox) are required. See Section 2.2.3 for more information.

- *No knowledge of C++ required*: No C++ knowledge (both syntax and compiling) is required to use the interface. Therefore ACADO for Matlab is the perfect way to start using ACADO Toolkit when you are familiar with MATLAB but don't have any C++ experience yet.

- *Familiar* MATLAB *syntax and workspace*: The interface should not be an identical duplicate of the C++ version but should make use of MATLAB style notations. On the one hand, it should be possible to directly use variables and matrices stored in the workspace. On the other hand, results should be directly available in the workspace after having executed a problem.

- *Use* MATLAB *black box models*: Although the ACADO Toolkit supports a symbolic syntax to write down differential (algebraic) equations, the main property of the interface is to link (existing) MATLAB black box models to ACADO Toolkit. Moreover, in addition to MATLAB black box models also C++ black box models can be used in the interface.

- *Cross-platform*: The interface should work on the most popular platforms around: Linux, Windows and Mac (more about this in Section 2.2.3).

1.4 Feedback and Questions

If you think you have found a bug, please add a bug report on

`http://forum.acadotoolkit.org/`

To be able to understand your problem include the following:

- The version number of ACADO Toolkit (and possibly the version of your MATLAB installation), the platform you are using and your compiler version.
- The exact error message.
- Your ACADO source file to reproduce the bug.

If you have a question regarding the ACADO Toolkit or ACADO for Matlab, try to answer them as follows:

- For questions regarding the ACADO syntax, consult the manual, the DOXYGEN source code documentation as well as the examples and comments in

  `<ACADOtoolkit-inst-dir>/examples` or
  `<ACADOtoolkit-inst-dir>/interfaces/matlab/examples`,

  respectively.
- Take a look at the FAQs where common problems are posted:

  `http://forum.acadotoolkit.org/`.
- Ask your questions on the forum

  `http://forum.acadotoolkit.org/`

  or send a mail to

  `support@acadotoolkit.org`

1.5 Citing the ACADO Toolkit

ACADO Toolkit and ACADO for Matlab are open-source software, so you can use it free of charge under the terms of the GNU LGPL licence. If you are using the software in your research work, please consider citing one or more of the following references [12, 11, 3]:

```
@ARTICLE{Houska2011,
  author = {B. Houska and H.J. Ferreau and M. Diehl},
  title = {{ACADO} {T}oolkit -- {A}n {O}pen {S}ource {F}ramework for
           {A}utomatic {C}ontrol and {D}ynamic {O}ptimization},
  journal = {Optimal Control Applications and Methods},
  year = {2011},
  volume = {32},
  pages = {298--312},
  number = {3}
}

@MISC{acadoManual,
  author = {B. Houska and H.J. Ferreau},
  title = {{ACADO} {T}oolkit {U}ser's {M}anual},
  howpublished = {http://www.acadotoolkit.org},
  year = {2009--2011}
}

@MISC{acadoForMatlabManual,
  author = {D. Ariens and B. Houska and H.J. Ferreau},
  title = {ACADO for Matlab User's Manual},
  howpublished = {http://www.acadotoolkit.org},
  year = {2010--2011}
}
```

Chapter 2

Installation

2.1 Installing the ACADO Toolkit

The software package ACADO Toolkit is written in an object-oriented manner in C++ and comes along with fully commented source code files. Besides some standards libraries *no further software packages are required*.

2.1.1 Installation under Linux

For installing ACADO Toolkit under LINUX, perform the following steps:

1. *Download the current version of* ACADO Toolkit from

   http://www.acadotoolkit.org

   by saving the file ACADO Toolkit-1.0.2613beta.tar.gz on your local machine.

2. *Unpack the archive:*

   tar xvfz rtcTOOLKIT-1.0.2613beta.tar.gz

   A new directory will be created; from now on we refer to (the full path of) this directory by <install-dir>. It contains five subfolders, namely

   - src (ACADO Toolkit source files),
   - include (ACADO Toolkit header files),
   - examples (example files),
   - interfaces (interfaces to third-party software),
   - doc (this manual and a DOXYGEN configuration file).

3. ACADO Toolkit is distributed under the terms of the GNU Lesser General Public License 3. *Please read this licence file carefully before you proceed with the installation*, as you agree with this licence by using ACADO Toolkit!

4. Continue with the

   Compilation of the ACADO Toolkit *library* libacado_toolkit.a:

   ```
   cd <install-dir>/src
   make
   ```

   The library libacado_toolkit.a provides the whole functionality of the ACADO Toolkit software package. It can be used by, e.g., linking it against a main function from the examples folder.

5. *Compilation of simple test examples:*

   ```
   cd <install-dir>/examples/getting_started
   make
   ```

   Run the executable ./simple_ocp in order to test your installation.

6. *Optional, create source code documentation*[1]*:*

   ```
   cd <install-dir>/doc
   doxygen doxygen.config
   ```

   Afterwards, you can open the file <install-dir>/doc/html/index.html with your favorite browser in order to view ACADO's source code documentation.

2.1.2 Installation under Windows

In order to install ACADO Toolkit under WINDOWS, we recommend to install the LINUX environment CYGWIN:

1. Go to www.cygwin.com, download and execute the setup program (setup.exe). This may require administrator priveleges.
2. Follow the instructions of the installation program. Make sure that you chose an installation path without any spaces (e.g. c:/cygwin).
3. In "select packages" step, select at least the following packages for installation:
   - From the Devel category, select "gcc-g++" and "make". If you plan to use SVN, you might want to select "subversion" as well.
   - From the Graphics category, select "gnuplot"
   - You will need an X-server to display the gnuplot graphics. For this, select "xinit" from the X11 cathgory, or manually configure your existing X-server to work with Cygwin after installation.
   - Click next and wait for the automatic installation
4. Start cygwin (a linux terminal will open). To use plotting, you need to open a terminal with the X-server. You can do this directly by running

   ```
   c:\cygwin\bin\run.exe /usr/bin/bash.exe -l -c /usr/bin/startxwin.exe
   ```

   in "run".

[1] All source code files are commented in a way suitable for the documentation system DOXYGEN [14].

5. Create a directory where you want to install ACADOtoolkit. Make sure that there are no spaces in the path, e.g.

   ```
   cd c:
   mkdir ACADO
   cd ACADO
   ```

6. Copy ACADO Toolkit-1.0.2613beta.tar.gz to the installation directory (here c:/ACADO) and unpack it:

   ```
   tar xfvz ACADO Toolkit-1.0.2613beta.tar.gz
   ```

7. Open the file ACADO Toolkit-1.0.2613beta/include/acado/include.mk in the ACADO directory with your favorite source-code editor (you can always use Wordpad) and make sure that the system is set to WIN32 and that the GNU compiler is used:

   ```
   # Compiler:  (GNU or VC)
   COMPILER = GNU

   # System:  (WIN32 or LINUX)
   SYSTEM = WIN32
   ```

8. Finally, make the ACADO Toolkit and run an example:

   ```
   cd ACADO Toolkit-1.0.2613beta
   make

   cd examples/getting_started
   ./simple_ocp
   ```

2.2 Installating ACADO for Matlab

To use ACADO for Matlab you'll need:

- The latest release of the toolkit available at

  http://www.acadotoolkit.org/download.php.

- A recent version of Matlab (see Section 2.2.3).
- A recent C++ compiler.

First of all, you will need to install a compiler (if you don't have a compiler yet), next the installed compiler will have to be linked to MATLAB. As a last step ACADO Toolkit needs to be compiled. These steps are now explained in more detail.

2.2.1 Installation under Linux or Mac

Step 1: Installing a compiler
Make sure you have installed a recent version of the GCC compiler (at least version 4.1 but 4.2 or later is advised). To check the current version of GCC run gcc -v in your terminal.

Step 2: Configuring Matlab

To link the compiler to MATLAB run:

```
mex -setup;
```

Matlab will return an output similar to this one:

```
The options files available for mex are:

  1: /software/matlab/2009b/bin/gccopts.sh :
      Template Options file for building gcc MEX-files

  2: /software/matlab/2009b/bin/mexopts.sh :
      Template Options file for building MEX-files via the system ANSI
          compiler


  0: Exit with no changes

Enter the number of the compiler (0-2):
```

In this case you should write 1 and hit enter. A confirmation message will be shown.

Step 3: Building ACADO for Matlab

Unzip all files to a location of your choice. We will refer to this location as

<ACADOtoolkit-inst-dir>.

Open MATLAB in this directory. Navigate to the Matlab installation directory by running:

```
cd interfaces/matlab/;
```

You are now ready to compile ACADO for Matlab. This compilation will take several minutes, but needs to be ran only once. Run `make clean all` in your command window. By doing a "clean" first, you are sure old ACADO object files are erased:

```
make clean all;
```

You will see:

```
Making ACADO...

and after a while when the compilation is finished:

ACADO successfully compiled.
Needed to compile xxx file(s).

If you need to restart Matlab, run this make file again
to set all paths or run savepath in your console to
save the current search path for future sessions.
```

ACADO Toolkit has now been compiled. As the output indicates, every time you restart MATLAB, you need to run make again to set all needed paths, but no new files will need to be compiled. It is easier to save your paths for future MATLAB session. Do so by running savepath in your command window (this step is optional). If you would like to add the needed paths manually, run these commands in <ACADOtoolkit-inst-dir>/interfaces/matlab/:

```
addpath(genpath([pwd filesep 'bin']));
addpath(genpath([pwd filesep 'shared']));
addpath([pwd filesep 'integrator']);
addpath([pwd filesep 'acado']);
addpath([pwd filesep 'acado' filesep 'functions']);
addpath(genpath([pwd filesep 'acado' filesep 'packages']));
```

2.2.2 Installation under Windows

Step 1: Installing a compiler
Install the Microsoft Visual C++ 2008 Express Edition compiler available at

http://www.microsoft.com/express/Downloads/#2008-Visual-CPP.

Complete the installation and restart your PC.

Step 2: Configuring Matlab
To link the compiler to MATLAB, run:

```
mex -setup;
```

Matlab will return an output similar to this one:

```
Select a compiler:
[1] Lcc-win32 C 2.4.1 in C:\PROGRA~1\MATLAB\R2009a\sys\lcc
[2] Microsoft Visual C++ 2008 Express in C:\Program Files\Microsoft Visual
    Studio 9.0

[0] None

Compiler:
```

In this case you should write 2 and hit enter. A confirmation message will be shown:

```
Please verify your choices:

Compiler: Microsoft Visual C++ 2008 Express
Location: C:\Program Files\Microsoft Visual Studio 9.0

Are these correct [y]/n?
```

Write down y and hit enter to confirm.

Step 3: Building ACADO for Matlab
Identical to step 3 of Section 2.2.1.

2.2.3 Compatibility

ACADO for Matlab is developed and tested on recent versions of Windows, Linux and Mac. At least Matlab 7.6 (R2008a) is required. This requirement is due to the fact that the interface uses the object oriented programming style of MATLAB and this is not (fully) available in older versions.

Table 2.1 summarizes the currently tested combinations of platforms, compiler versions and MATLAB versions. Post a message on http://forum.acadotoolkit.org/list.php?14 if you can confirm that ACADO for Matlab is running on another combination.

Platform	Compiler	Matlab Version
Windows XP	Visual C++ Compiler 2008 Express	Matlab 7.8.0.347 (R2009a)
Windows Vista	Visual C++ Compiler 2008 Express	Matlab 7.9.0.529 (R2009b)
Windows 7	Visual C++ Compiler 2008 Express	Matlab 7.10.0.499 (R2010a)
Mac OS X	GCC 4.2.1	Matlab 7.8.0.347 (R2009a)
Linux 64bit	GCC 4.4.3	Matlab 7.7.0.471 (R2008b)
Linux 64bit	GCC 4.4.3	Matlab 7.8.0.347 (R2009a)
Linux 64bit	GCC 4.4.3	Matlab 7.9.0.529 (R2009b)
Linux 64bit	GCC 4.4.3	Matlab 7.10.0.499 (R2010a)
Linux x86	GCC 4.4.3	Matlab 7.7.0.471 (R2008b)
Linux x86	GCC 4.4.3	Matlab 7.8.0.347 (R2009a)
Linux x86	GCC 4.4.3	Matlab 7.9.0.529 (R2009b)
Linux x86	GCC 4.4.3	Matlab 7.10.0.499 (R2010a)
Linux x86	GCC 4.3.3-5ubuntu4	Matlab 7.8.0.347 (R2009a)

Table 2.1: Tested platforms ACADO for Matlab

2.2.4 About Compiling and MEX Functions

The interface will generate a C++ file of your problem formulation and compile it to a MEX-file. MEX stands for MATLAB Executable and provides an interface between Matlab and C++. When running the initial make call upon installation all ACADO source files are compiled to individual object files. Upon completing your problem formulation, the object files will be used to build one MEX-file.

Part II

Dynamic Optimization

Chapter 3

Optimal Control Problem

3.1 A Guiding Example: Time Optimal Control of a Rocket Flight

This section explains how to setup a simple optimal control problem using the ACADO Toolkit. As an example a simple model of a rocket is considered, which should fly as fast as possible from one to another point in space while satisfying state and control constraints during the flight.

3.1.1 Mathematical Formulation

We consider a simple rocket model with three differential states s, v, and m representing the traveling distance, the velocity, and the mass of the rocket, respectively. Moreover, we assume that the rocket can be accelerated by a control input u . The fuel optimal control problem of our interest has the following form:

$$
\begin{aligned}
&\underset{s(\cdot),v(\cdot),m(\cdot),u(\cdot),T}{\text{minimize}} && T \\
&\text{subject to:} && \\
&\forall t \in [0,T]: && \dot{s}(t) = v(t) \\
&\forall t \in [0,T]: && \dot{v}(t) = \tfrac{u(t) - 0.2 * v(t)^2}{m(t)} \\
&\forall t \in [0,T]: && \dot{m}(t) = -0.01 * u(t)^2 \\
& && s(0) = 0 \; v(0) = 0 \; m(0) = 1 \\
& && s(10) = 10 \; v(10) = 0 \\
&\forall t \in [0,T]: && -0.1 \leq v(t) \leq 1.7 \\
&\forall t \in [0,T]: && -1.1 \leq u(t) \leq 1.1 \\
& && 5.0 \leq T \leq 15.0
\end{aligned}
\tag{3.1}
$$

Here, the aim is to fly in minimum time T from $s(0) = 0$ to $s(T) = 10$, while constraints on the velocity v and the control input u should be satisfied. Note that the rocket is assumed

to start with velocity $v(0) = 0$ and required to stop at the end time T, which can be formulated in form of the constraint $v(T) = 0$.

3.1.2 Implementation in ACADO Syntax

The following piece of code shows how to implement the above optimal control problem. In addition, a GNUPLOT window is constructed, such that the results can automatically be visualized:

```
#include <acado_optimal_control.hpp>
#include <include/acado_gnuplot/gnuplot_window.hpp>

int main( ){

    USING_NAMESPACE_ACADO

    DifferentialState        s,v,m      ;   // the differential states
    Control                  u          ;   // the control input u
    Parameter                T          ;   // the time horizon T
    DifferentialEquation     f( 0.0, T );   // the differential equation

  //------------------------------------
    OCP ocp( 0.0, T )                   ;   // time horizon of the OCP: [0,T]
    ocp.minimizeMayerTerm( T )          ;   // the time T should be optimized

    f << dot(s) == v                    ;   // an implementation
    f << dot(v) == (u-0.2*v*v)/m        ;   // of the model equations
    f << dot(m) == -0.01*u*u            ;   // for the rocket.

    ocp.subjectTo( f                    );   // minimize T s.t. the model,
    ocp.subjectTo( AT_START, s ==  0.0 );   // the initial values for s,
    ocp.subjectTo( AT_START, v ==  0.0 );   // v,
    ocp.subjectTo( AT_START, m ==  1.0 );   // and m,

    ocp.subjectTo( AT_END  , s == 10.0 );   // the terminal constraints for s
    ocp.subjectTo( AT_END  , v ==  0.0 );   // and v,

    ocp.subjectTo( -0.1 <= v <=  1.7   );   // as well as the bounds on v
    ocp.subjectTo( -1.1 <= u <=  1.1   );   // the control input u,
    ocp.subjectTo(  5.0 <= T <= 15.0   );   // and the time horizon T.
  //------------------------------------

    GnuplotWindow window                ;   // visualize the results in a
    window.addSubplot( s, "DISTANCE s" );   // Gnuplot window.
    window.addSubplot( v, "VELOCITY v" );
    window.addSubplot( m, "MASS     m" );
    window.addSubplot( u, "CONTROL  u" );

    OptimizationAlgorithm algorithm(ocp);   // construct optimization algorithm,
    algorithm << window                 ;   // flush the plot window,
    algorithm.solve()                   ;   // and solve the problem.

    return 0                            ;
}
```

This code example is also coming with the ACADO Toolkit and can in this version directly be compiled. The translation of the mathematical formulation into the C++ code should be intuitive. Although the problem is nonlinear, we do not necessarily need to provide an initialization. Note that the ACADO Toolkit tries to guess an initialization based on the

constraints which occur in the problem formulation. Moreover, we did not specify any options regarding the optimization algorithm; the ACADO Toolkit chooses default options. In this example, a multiple shooting discretization with 20 nodes is chosen, while the integration is performed by a Runge-Kutta method (order 4/5). Finally, the optimization of the discretized mathematical program is by default based on a sequential quadratic programming (SQP) method.

3.1.3 Numerical Results

Compiling and running the code should lead to both: An output of the SQP iterations on the terminal as well as a GNUPLOT window, which is shown as soon as convergence is achieved. The result should look as follows:

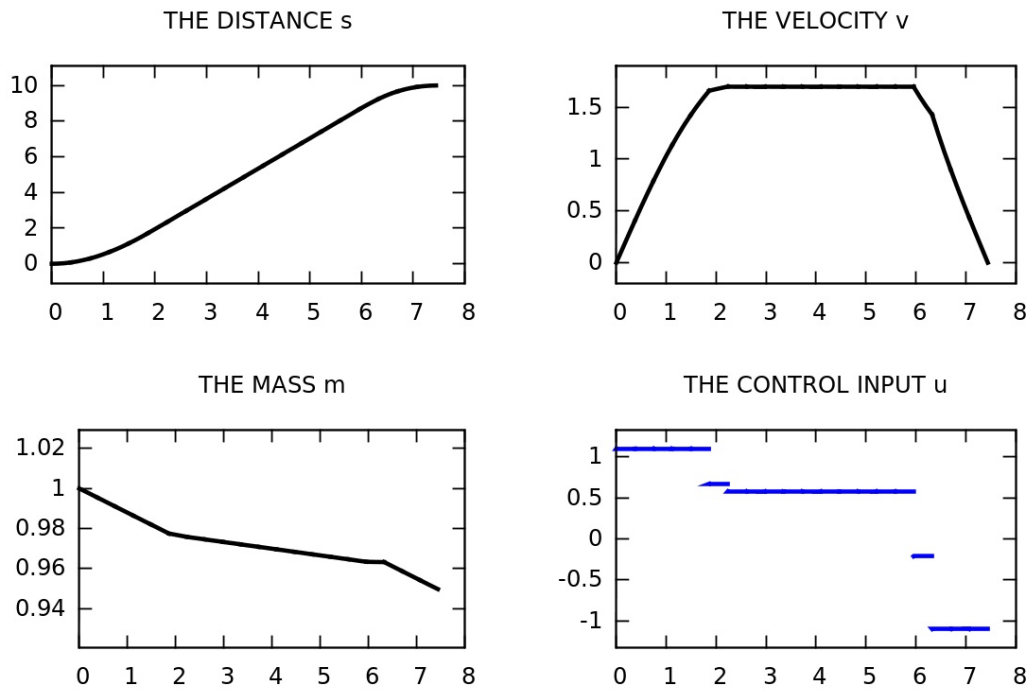


Figure 3.1: GNUPLOT window illustrating the time optimal rocket flight.

The output on the terminal looks as follows:

```
ACADO Toolkit::SCPmethod -- A Sequential Quadratic Programming Algorithm.
Copyright (C) 2008-2011 by Boris Houska and Hans Joachim Ferreau, K.U.Leuven.
Developed within the Optimization in Engineering Center (OPTEC) under
supervision of Moritz Diehl. All rights reserved.

ACADO Toolkit is distributed under the terms of the GNU Lesser
General Public License 3 in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU Lesser General Public License for more details.

  1:  KKT tolerance = 4.016e+01     objective value = 9.9500e+00
```

```
  2:  KKT tolerance = 1.306e-01    objective value = 9.9316e+00
  3:  KKT tolerance = 2.549e-02    objective value = 9.9061e+00
  4:  KKT tolerance = 7.485e-02    objective value = 9.8314e+00
  5:  KKT tolerance = 3.458e-01    objective value = 9.4875e+00
  6:  KKT tolerance = 3.045e-01    objective value = 9.1909e+00
  7:  KKT tolerance = 5.194e-01    objective value = 8.6915e+00
  8:  KKT tolerance = 4.739e-01    objective value = 8.2481e+00
  9:  KKT tolerance = 3.335e-01    objective value = 7.9276e+00
 10:  KKT tolerance = 4.999e-01    objective value = 7.4579e+00
 11:  KKT tolerance = 1.653e-02    objective value = 7.4419e+00
 12:  KKT tolerance = 1.461e-04    objective value = 7.4417e+00
 13:  KKT tolerance = 1.190e-07    objective value = 7.4417e+00

convergence achieved.
```

Here, the optimal results for the three states as well as for the control input are plotted. Note that the time optimal result can quite intuitively be understood: In the first phase, it is optimal to accelerate as fast as possible, i.e. the upper bound constraint for the control input is active. In the second phase, the path constraint for the maximum velocity is active and thus the control input is chosen in such a way that the friction is copensated. Finally, In the third phase, the rocket must brake as fast as possible, i.e. the lower bound constraint is active. Note that in this example only 20 piecewise constant control intervals have been chosen, i.e. the discretiziation of the controls is quite poor in this example.

3.2 Initialization of Nonlinear Optimization Algorithms

When nonlinear optimization algorithms are used to solve mathematical programs often initializations are required. In some special cases, e.g. if an optimization problem is convex, no such initialization is needed as there are guarantees that the algorithm converges. However, even for such convex problems, initial guesses that are close to the optimal solution might considerably speed up the iteration progress. This section describes three possible ways to initialize nonlinear optimization algorithms within the ACADO Toolkit.

3.2.1 Using the Built-In Auto-Initialization

The most convenient way of initializing an algorithm is by relying on the auto-initialization. This auto-initialization routine does often work for not too difficult problems, which are either convex or not too nonlinear. In the previous example of Section 3.1 we have already used the auto-initialization without understanding the details. The key strategy of ACADO is to use the constraints of the problem to generate an initial guess. For example the code lines

```
ocp.subjectTo( -1.1 <= u <=  1.1   );
ocp.subjectTo(  5.0 <= T <= 15.0   );
```

define bounds on a control input u and the horizon length T. If nothing else is specified, ACADO will detetect these bounds and initialize with $u(t) = 0$ for all $t \in [0, T]$ as this is the arithmetic mean between the upper and the lower bound. Similarly, the parameter T, representing in our example the duration of the rocket flight, will be initialized with $T = 10$. If only one bound is specified, the corresponding variable will be initialized at this bound.

If no constraint has been detected the auto-initialization routine will start with 0 as an initial guess. Similarly, the differential states are initialized by the first simulation with the specified initial values.

Let us sketch the algotithmic strategy of the auto-initialization routine as follows:

- The auto-intialization routine uses the bounds on the variables to generate initial guesses. If an upper and a lower bound is given, the initial guess will be the arithmetic mean (this contains the case of an equality bound, where upper and lower bound are equal). If only one of these bounds is specified (while the other one is $\pm\infty$) the initial guess will be equal to this bound. If there is a variable for which no bounds are specified, the initial guess will simply be 0.

- The initial values for the differential equations are also generated from their bounds. However, the intermediate values are obtained by a simulation of the differential system with the initial guess for the controls, parameters, and initial states.

- Bounds on the differential states are also taken into account in order to improve the heuristic. If multiple shooting is used, the multiple shooting nodes will during the simulation be projected into the feasible box, if a state bound is violated.

- In contrast to bounds, general nonlinear path constraints are not regarded by the auto-initialization.

Summarizing the strategy, all bounds on the variables are used to improve the initial guess. Thus, it is recommended to provide reasonable bounds for the case that auto-initialization should be used.

Advantages of the auto-initialization:

- The main advantage of the auto-initialization is that it is very convenient to use as we do not need to provide any information about the problem—beside the problem itself.

- The bounds on the variables in an optimal control problem do often specify the domain in which the model has a physical meaning or interpretation. In this case, the auto-initialization leads to a kind of natural initialization.

Disadvantages of the auto-initialization:

- The auto-initialization is only a heuristic which does not work in general. For nonlinear problems there is no guarantee that the heuristic leads to a convergence of the optimization routine.

- If one of the bounds is changed, the initialization also changes. Thus, the algorithm might work for a given bound while it fails if this bound is changed—even if the bound is never active and would not affect the optimal solution.

3.2.2 Loading the Initialization from a Text File

As an alternative to the auto-initialization it is possible to specify initial values in a simple text file. In ACADO Toolkit convenient reading routines are implemented. In order to demonstrate an example we assume that we have defined an optimal control problem "ocp" as in section 3.1. Now, we try to solve this optimal control problem via the following lines of code:

```
OptimizationAlgorithm algorithm(ocp);

algorithm.initializeDifferentialStates( "x.txt" );
algorithm.initializeControls          ( "u.txt" );
algorithm.initializeParameters        ( "p.txt" );

algorithm.solve();
```

Here, the initialization for the differential states, controls, and parameters are assumed to be stored in separate files, which contain the corresponding time-series. For example, these file `x.txt` could read as follows:

```
time            s            v            m

0.00e+00   0.00e+00   0.00e+00   1.00e+00
1.00e-01   2.99e-01   7.90e-01   9.90e-01
2.00e-01   1.13e+00   1.42e+00   9.81e-01
3.00e-01   2.33e+00   1.69e+00   9.75e-01
4.00e-01   3.60e+00   1.70e+00   9.73e-01
5.00e-01   4.86e+00   1.70e+00   9.70e-01
6.00e-01   6.13e+00   1.70e+00   9.68e-01
7.00e-01   7.39e+00   1.70e+00   9.65e-01
8.00e-01   8.66e+00   1.70e+00   9.63e-01
9.00e-01   9.67e+00   8.98e-01   9.58e-01
1.00e+00   1.00e+01   0.00e+00   9.49e-01
```

Actually, this tutorial already describes the most difficult case: first, the time T is optimized in our example, such that the time series for the states and controls have to be rescaled to $[0, 1]$. And second, the number of controls in the file `u.txt` is 11—but in our example uses the default settings, i.e. 20 control intervals. Note that the ACADO Toolkit does not require the files to be consistent, i.e. in the above case the missing control and state inititalizations are automatically generated by linear interpolation. Fortunately, having understood this difficult example, we have already understood everything that needs to known about initialization via text files.
Let us summarize the six important key concepts regarding the initialization via text files:

- The text file for the initialization should contain a time series with the values of the time in the first column and the values of the states, controls, or parameters respectively in the remaining columns.
- The number of rows, i.e. the number of time points at which an initial guess is specified, is not required to be equal to the number of control or discretization intervals of the algorithm. If there are some time points missing the corresponding values will automatically be generated by linear interpolation. In particular, the file `u.txt` could in this example contain a different number of rows than the file `x.txt`, for example.

- The files may contain characters like the word "time" in our examples. `ACADO Toolkit` will simply ignore every character in the text which can not possibly be interpreted as a number. On the one hand, this allows to add comments to a text file; but on the other hand, we should be careful, as there might be a character in our comment which can be interpreted as a number—possibly leading to unwanted behaviour.

- It is possible to combine different initialization methods. In the above situation we could for example only provide the file `u.txt`. In this case, the control input u would be initialized from the file, while the initial guesses for the state vector x and the horizon length T are generated by the automatic initialization strategy.

- The time points in the first column of the file do not need to be equidistant, but they are required to be strictly monotonically increasing.

- For the case that the duration is a parameter to be optimized, the time series for the states and controls have to be rescaled to $[0, 1]$. This convention is on the first view a little confusing. However, just assume that the parameters are not initialized by the user, while a time series for the control is specified. In this case, ACADO would automatically choose a horizon length T which might not be consistent with the control initialization... Thus, it has turned out that it is in fact better to introduce the convention that the time series are rescaled in order to scope with this case.

Finally, we discuss the general advantages and disadvantages of the initialization method via text files:
Advantages of the initialization via text files:

- The initialization via text files allows to exchange the initial guess without re-compiling the code as the file is read at run time.

- The initialization via text files decouples the initialization of the algorithm with the formulation of the mathematical problem. For example if a bound on a variable changes within the problem, the auto-initialization would be affected, while the text file remains of course the same.

Disadvantages of the initialization via text files:

- We need a way to generate the text file with some method—e.g. with another program like MATLAB. Writing a text file by hand might be quite some work.

- If an optimization problem should be initialized for many times with many different initialization (e.g. in an online context), it might not be a good idea to use text files, as reading the txt-files might be too slow. Moreover, if we like to use the ACADO Toolkit from or within another program, it is usually—depending on the situation—a rather bad design of an interface to communicate the initialization via files.

3.2.3 Using ACADO Data Structures for the Initialization

The third way of initializing a nonlinear optimization algorithm is based on the data structures which are available in the ACADO Toolkit. The class which is needed for this purpose is called VariablesGrid. This data class is suitable to store time series of vector valued functions. Let us explain this concept by considering the following piece of code:

```
OptimizationAlgorithm algorithm(ocp);

Grid timeGrid( 0.0, 1.0, 11 );

VariablesGrid   x_init( 3, timeGrid );
VariablesGrid   u_init( 1, timeGrid );
VariablesGrid   p_init( 1, timeGrid );

x_init(0,0 ) = 0.00e+00; x_init(1,0 ) = 0.00e+00; x_init(2,0 ) = 1.00e+00;
x_init(0,1 ) = 2.99e-01; x_init(1,1 ) = 7.90e-01; x_init(2,1 ) = 9.90e-01;
x_init(0,2 ) = 1.13e+00; x_init(1,2 ) = 1.42e+00; x_init(2,2 ) = 9.81e-01;
x_init(0,3 ) = 2.33e+00; x_init(1,3 ) = 1.69e+00; x_init(2,3 ) = 9.75e-01;
x_init(0,4 ) = 3.60e+00; x_init(1,4 ) = 1.70e+00; x_init(2,4 ) = 9.73e-01;
x_init(0,5 ) = 4.86e+00; x_init(1,5 ) = 1.70e+00; x_init(2,5 ) = 9.70e-01;
x_init(0,6 ) = 6.13e+00; x_init(1,6 ) = 1.70e+00; x_init(2,6 ) = 9.68e-01;
x_init(0,7 ) = 7.39e+00; x_init(1,7 ) = 1.70e+00; x_init(2,7 ) = 9.65e-01;
x_init(0,8 ) = 8.66e+00; x_init(1,8 ) = 1.70e+00; x_init(2,8 ) = 9.63e-01;
x_init(0,9 ) = 9.67e+00; x_init(1,9 ) = 8.98e-01; x_init(2,9 ) = 9.58e-01;
x_init(0,10) = 1.00e+01; x_init(1,10) = 0.00e+00; x_init(2,10) = 9.49e-01;

u_init(0,0 ) =  1.10e+00;
u_init(0,1 ) =  1.10e+00;
u_init(0,2 ) =  1.10e+00;
u_init(0,3 ) =  5.78e-01;
u_init(0,4 ) =  5.78e-01;
u_init(0,5 ) =  5.78e-01;
u_init(0,6 ) =  5.78e-01;
u_init(0,7 ) =  5.78e-01;
u_init(0,8 ) = -2.12e-01;
u_init(0,9 ) = -1.10e+00;
u_init(0,10) = -1.10e+00;

p_init(0,0 ) =  7.44e+00;

algorithm.initializeDifferentialStates( x_init );
algorithm.initializeControls          ( u_init );
algorithm.initializeParameters        ( p_init );

algorithm.solve();
```

Note that the above example is equivalent to the previous example with the text files. The only difference is that the initialization is not read-in but directly hard-coded in the C++ file. The class Grid is constructed with three arguments: the line `Grid timeGrid( 0.0, 1.0, 11 );` constructs a grid with 11 time points that are equally distributed over the interval $[0.0, 1.0]$. Moreover, the constructor of the VariablesGrid gets the dimensions of the function and the sampling time grid (in our example the differential states have the dimension 3, while the controls and parameters have both the dimension 1). The rest is the same as for the initialization with text files.

Main advantage of the initialization via ACADO data structures:

- The main advantage of the initialization with the ACADO data structure VariablesGrid

is that no files are needed. This method is especially useful if the code should be used from another program or in an online context where a communication via files might be too slow.

Main disadvantages of the initialization via ACADO data structures:

- The initialization is not read at run-time. I.e., if we like to change the initialization, the code must be re-compiled.

3.3 Algorithmic Options

In the guiding example of section 3.1, we have only used the optimization algorithms with its default settings. For optimal control problems these default settings are usually a multiple-shooting SQP type method combined with a standard Runge-Kutta integrator for the state integration. This section describes how to overwrite the default settings.

3.3.1 A Tutorial Code using Algorithmic Options

Let us re-view the listing of section 3.1 but now specifying several algorithmic options.

```
#include <acado_optimal_control.hpp>
#include <include/acado_gnuplot/gnuplot_window.hpp>

int main( ){

    USING_NAMESPACE_ACADO

    DifferentialState       s,v,m     ;   // the differential states
    Control                 u         ;   // the control input u
    Parameter               T         ;   // the time horizon T
    DifferentialEquation    f( 0.0, T );   // the differential equation

  // ------------------------------------
    OCP ocp( 0.0, T, 50 )               ;   // time horizon of the OCP: [0,T]
                                            // use 50 control intervals
    ocp.minimizeMayerTerm( T )          ;   // the time T should be optimized

    f << dot(s) == v                    ;   // an implementation
    f << dot(v) == (u-0.2*v*v)/m        ;   // of the model equations
    f << dot(m) == -0.01*u*u            ;   // for the rocket.

    ocp.subjectTo( f                    );   // minimize T s.t. the model,
    ocp.subjectTo( AT_START, s ==  0.0 );   // the initial values for s,
    ocp.subjectTo( AT_START, v ==  0.0 );   // v,
    ocp.subjectTo( AT_START, m ==  1.0 );   // and m,

    ocp.subjectTo( AT_END  , s == 10.0 );   // the terminal constraints for s
    ocp.subjectTo( AT_END  , v ==  0.0 );   // and v,

    ocp.subjectTo( -0.1 <= v <=  1.7   );   // as well as the bounds on v
    ocp.subjectTo( -1.1 <= u <=  1.1   );   // the control input u,
    ocp.subjectTo(  5.0 <= T <= 15.0   );   // and the time horizon T.
  // ------------------------------------

    OptimizationAlgorithm algorithm(ocp);   // construct optimization
        algorithm,
```

```
    algorithm.set( INTEGRATOR_TYPE       , INT_RK78         );
    algorithm.set( INTEGRATOR_TOLERANCE  , 1e-8             );
    algorithm.set( DISCRETIZATION_TYPE   , SINGLE_SHOOTING  );
    algorithm.set( KKT_TOLERANCE         , 1e-4             );

    algorithm.solve()                    ;   // and solve the problem.

    return 0                             ;
}
```

The options which have been set in this example are first the integrator type: now, the Runge-Kutta integrator with order (7/8) will be used (instead of a Runge Kutta integrator with order 4/5, which is the default choice). In addition, the integrator tolerance has been set, while single shooting is used instead of the multiple shooting method, which would be the default choice. Finally, the KKT tolerance, which is used for the convergence criterion of the SQP algorithm, has been set to $1e-4$. Here, $1e-6$ would have been the default choice.

Note that all options can be set on the optimization algorithm by using the syntax

```
set( <Option Name>, <Option Value> )
```

An important exception are the number of control intervals which are specified in the constructor of the OCP following the definition of the time interval.

3.3.2 Most Common Algorithmic Options

The following table summarizes the most commonly used algorithmic options for solving optimal control with the ACADO Toolkit:

Option Name:	**Possible Values:**	**Default Value:**
IntegratorType	INT_RK12 (Runge-Kutta $1,2$) INT_RK23 (Runge-Kutta $2,3$) INT_RK45 (Runge-Kutta $4,5$) INT_RK78 (Runge-Kutta $7,8$) INT_BDF (BDF integrator)	INT_RK45 (for ODE's) or INT_BDF (for DAE's)
maxNumIterations	int	1000
KKTtolerance	double	10^{-6}
LevenbergMarquardt	double	0
printLevel	PL_NONE PL_LOW PL_MEDIUM PL_HIGH	PL_LOW

HessianApproximation	CONSTANT_HESSIAN FULL_BFGS_UPDATE BLOCK_BFGS_UPDATE GAUSS_NEWTON GAUSS_NEWTON_WITH_BLOCK_BFGS	BLOCK_BFGS_UPDATE
DynamicSensitivity	FORWARD_SENSITIVITY BACKWARD_SENSITIVITY	BACKWARD_SENSITIVITY
ObjectiveSensitivity	FORWARD_SENSITIVITY BACKWARD_SENSITIVITY	BACKWARD_SENSITIVITY
ConstraintSensitivity	FORWARD_SENSITIVITY BACKWARD_SENSITIVITY	BACKWARD_SENSITIVITY
DiscretizationType	MULTIPLE_SHOOTING SINGLE_SHOOTING	MULTIPLE_SHOOTING
LineSearchTolerance	double	SQRT_EPS
MinimumLineSearchParameter	double	0.25
MaximumNumberOfQPiterations	int	10000
InitialStepSize	double	10^{-3}
MinimumStepSize	double	10^{-8}
MaximumStepSize	double	10^{8}
StepSizeTuning	double	0.5
IntegratorPrintLevel	PL_NONE PL_LOW PL_MEDIUM PL_HIGH	PL_LOW

3.4 Storing the Results of Optimization Algorithms

This section explains how to obtain and store the results of an optimization algorithm. In the guiding example of section 3.1, it has already been explained how to plot the results with GNUPLOT. However, once an optimization problem has been solved with ACADO, one of the first question that arises is how to obtain the numerical results.

3.4.1 Storing the Results in a Text File

The easiest way to store results with ACADO is via text files. Analogous to the initialization of optimal control algorithms, the results can e.g. be obtained by the following lines of code:

```
#include <acado_optimal_control.hpp>
#include <include/acado_gnuplot/gnuplot_window.hpp>
```

```
int main( ){

    USING_NAMESPACE_ACADO

    // ... (IMPLEMENTATION OF THE OPTIMIZATION PROBLEM) ...

    OptimizationAlgorithm algorithm(ocp);
    algorithm.solve()                   ;

    algorithm.getDifferentialStates("states.txt"     );
    algorithm.getParameters         ("parameters.txt");
    algorithm.getControls           ("controls.txt"  );

    return 0;
}
```

The above example will store the results for differential states, parameters, and controls in the text files `states.txt`, `parameters.txt`, and `controls.txt`, respectively. As an easy exercise, it is recommended to test the following:

- Solve an optimal control (e.g. the time optimal rocket problem).
- Store the results in text files as explained above.
- Initiliaze the optimization algorithm with the soultion and run it again.

The result of this exercise should be that the optimization algorithm detects directly that the problem is initialized in the solution and performs only one SQP iteration.

3.4.2 Obtaining the Results in Form of ACADO Data Structures

Similar to the storage of results in form of text files, the result can also be obtained in form of a `VariablesGrid`. The syntax is analogous:

```
#include <acado_optimal_control.hpp>
#include <include/acado_gnuplot/gnuplot_window.hpp>

int main( ){

    USING_NAMESPACE_ACADO

    // ... (IMPLEMENTATION OF THE OPTIMIZATION PROBLEM) ...

    OptimizationAlgorithm algorithm(ocp);
    algorithm.solve()                   ;

    VariablesGrid states, parameters, controls;

    algorithm.getDifferentialStates(states    );
    algorithm.getParameters         (parameters);
    algorithm.getControls           (controls  );

    states.print();
    parameters.print();
    controls.print();

    return 0;
}
```

The advantage of getting the results in form of a VariablesGrid is that they can for example processed by a user-written C++ routine or modified and then written to a text file. In addition, in a real-time context, communication via files is not recommended and thus a VariablesGrid is right medium for communication in this case.

3.4.3 The ACADO Logging Functionality

Another way to retrieve results is provided by the logging functionality of ACADO Toolkit. It allows you to setup so-called LogRecords to be passed to the optimization algorithm. Therein, you can specify which information you would like to log and the algorithm will take care of that. After running the optimization algorithm, the desired information is logged within your LogRecord and can be printed onto the screen or to a file. We give a simple example:

```
#include <acado_optimal_control.hpp>
#include <include/acado_gnuplot/gnuplot_window.hpp>

int main( ){

    USING_NAMESPACE_ACADO

    // ... (IMPLEMENTATION OF THE OPTIMIZATION PROBLEM) ...

    OptimizationAlgorithm algorithm(ocp);

    // setup a logging object and flush it into the algorithm
    LogRecord logRecord( LOG_AT_EACH_ITERATION,"kkt.txt" );
    logRecord << LOG_KKT_TOLERANCE;

    algorithm << logRecord;

    // solve the optimization problem
    algorithm.solve( );

    // get the logging object back and print it
    algorithm.getLogRecord( logRecord );
    logRecord.print( );

    return 0;
}
```

In this example a LogRecord is defined that logs the KKT tolerance at each iteration that shall be written into the file kkt.txt. Note that you can add more than one entry to each LogRecord and that you can flush several LogRecords containing different entries with different log schemes into the same algorithm. Also the format of the output on printing can be adjusted in detail. You might either log at each iteration as above, or only at start/end of the optimization using LOG_AT_START/LOG_AT_END, respectively. For example, the following information can be logged:

Logging name:	Description:
LOG_NUM_NLP_ITERATIONS	Number of iterations of the NLP solver
LOG_KKT_TOLERANCE	KKT tolerance
LOG_OBJECTIVE_FUNCTION	Objective function value
LOG_MERIT_FUNCTION_VALUE	Value of merit function
LOG_LINESEARCH_STEPLENGTH	Steplength of the line search routine (if used)
LOG_ALGREBRAIC_STATES	All algebraic states in the order of occurence
LOG_PARAMETERS	All parameters in the order of occurence
LOG_CONTROLS	All controls in the order of occurence
LOG_DISTURBANCES	All disturbances in the order of occurence
LOG_INTERMEDIATE_STATES	All intermediate states in the order of occurence
LOG_DIFFERENTIAL_STATES	All differential states in the order of occurence

3.5 Optimization of Differential Algebraic Systems

This section explains how to solve optimal control problems for which the model equation contains not only differential, but also algebraic states.

3.5.1 Mathematical Formulation

For the general DAE formulation we summarize the differential and algebraic states of the DAE in one vector x . Moreover, we denote by u the control input, by p a constant parameter, and by T the time horizon length of an DAE optimization problem. The general problem formulation reads now as follows:

$$\begin{aligned} &\underset{x(\cdot),u(\cdot),p,T}{\text{minimize}} && \Phi(x(\cdot),u(\cdot),p,T) \\ &\text{subject to:} && \\ &\forall t \in [0,T]: && 0 = F(t,x(t),\dot{x}(t),u(t),p) \\ &\forall t \in [0,T]: && 0 \leq h(t,x(t),u(t),p) \\ & && 0 = r(x(0),x(T),p) \end{aligned} \tag{3.2}$$

Here, the function F denotes the model equation, Φ the objective functional, h the path constraints, and r the boundary constraints of the optimization problem.

Remarks:

- The model function F can in practice often be written as

$$0 = F(t,x(t),\dot{x}(t),u(t),p) = \begin{pmatrix} \dot{x}(t) - f_1(t,x(t),u(t),p) \\ f_2(t,x(t),u(t),p) \end{pmatrix}$$

  In this case, we say that the DAE is semi-implicit.

- Another special case, which often occurs in practice, is that the function F is linear in dx/dt such that we have

$$0 = F(t,x(t),\dot{x}(t),u(t),p) = M(t,x(t),u(t),p)\,\dot{x}(t) - f(t,x(t),u(t),p)$$

for a matrix valued function M. However, in ACADO linear dependencies are automatically detected such that from the user point of view, we do not have to make a difference between linear and fully-implicit DAEs.

3.5.2 An ACADO Tutorial Code for Semi-Implicit DAEs

The following piece of code illustrates how to setup a simple DAE optimization problem for the case that the DAE is semi-implicit:

```
#include <acado_optimal_control.hpp>
#include <include/acado_gnuplot/gnuplot_window.hpp>

int main( ){

    USING_NAMESPACE_ACADO

    // INTRODUCE THE VARIABLES:
    // -------------------------
    DifferentialState     x;
    DifferentialState     l;
    AlgebraicState        z;
    Control               u;
    DifferentialEquation  f;

    const double t_start =  0.0;
    const double t_end   = 10.0;

    // DEFINE A DIFFERENTIAL EQUATION:
    // -------------------------------
    f << dot(x) == -x + 0.5*x*x + u + 0.5*z;
    f << dot(l) ==  x*x + 3.0*u*u          ;
    f <<      0 ==  z + exp(z) - 1.0 + x   ;


    // DEFINE AN OPTIMAL CONTROL PROBLEM:
    // ----------------------------------
    OCP ocp( t_start, t_end, 10 );
    ocp.minimizeMayerTerm( l );

    ocp.subjectTo( f );
    ocp.subjectTo( AT_START, x == 1.0 );
    ocp.subjectTo( AT_START, l == 0.0 );

    GnuplotWindow window;
      window.addSubplot(x,"DIFFERENTIAL STATE  x");
      window.addSubplot(z,"ALGEBRAIC STATE  z"   );
      window.addSubplot(u,"CONTROL  u"   );


    // DEFINE AN OPTIMIZATION ALGORITHM AND SOLVE THE OCP:
    // ---------------------------------------------------
    OptimizationAlgorithm algorithm(ocp);

    algorithm.set( ABSOLUTE_TOLERANCE     , 1e-7          );
    algorithm.set( INTEGRATOR_TOLERANCE   , 1e-7          );
    algorithm.set( HESSIAN_APPROXIMATION , EXACT_HESSIAN );

    algorithm << window;
    algorithm.solve();

    return 0;
```

```
}
```

Running this example, the corresponding GNUPLOT output should look as follows:

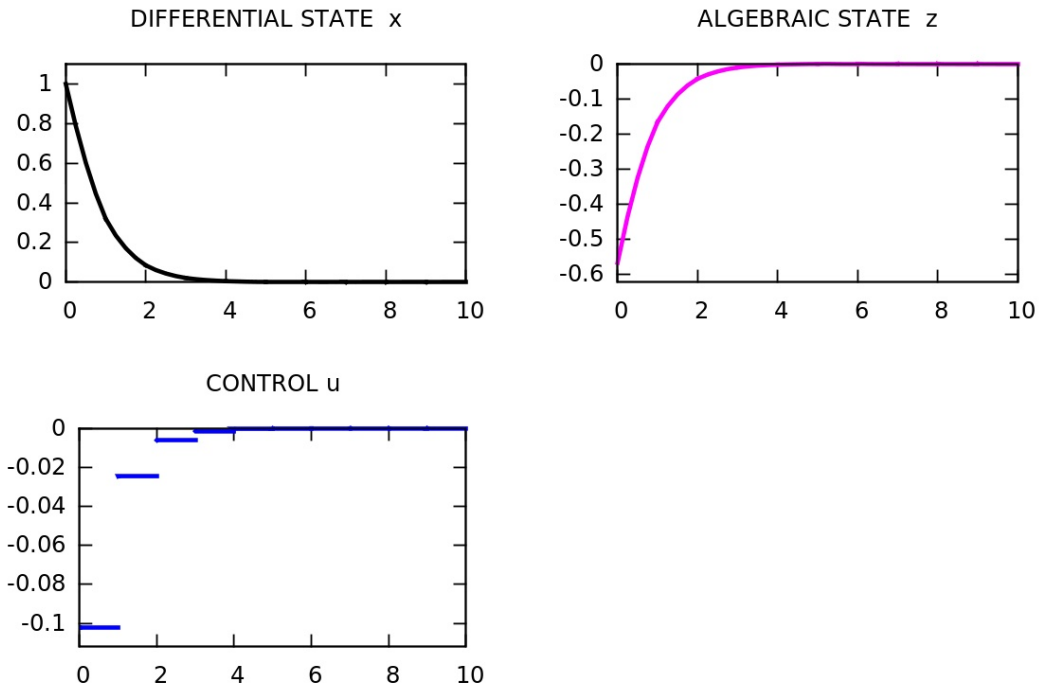


Figure 3.2: GNUPLOT window illustrating the solution to the DAE optimization problem.

3.6 Optimal Control of Discrete-Time Systems

This section explains how to setup a optimal control problems for discrete time systems.

3.6.1 Mathematical Formulation

A discrete time system consists typically of a state sequence (x_k) and an associated time sequence (t_k) satisfying an iteration of the form

$$\begin{aligned} x_{k+1} &= f(t_k, x_k) \\ t_{k+1} &= t_k + h_k \end{aligned}$$

for $k = 1, 2, ..., N$. Here, h_k are given time steps. In the optimal control context, the right-hand side function f might of course additionally depenend on controls u_k, parameters p etc. The rest of the formulation is analoguous to the description given in section 3.1 with the only difference that the continuous dynamics are exchanged with the discrete-time system.

3.6.2 Implementation in ACADO Syntax

In the following code example, the problem given in section 3.1 is implemented based on a discrete-time system, which can e.g. be obtained by applying an Euler method with constant step size h. (Note that this example is just for demonstration. In practice, it is usually not recommended to discretize continuous systems with Euler methods.)

```
#include <acado_optimal_control.hpp>
#include <include/acado_gnuplot/gnuplot_window.hpp>

int main( ){

    USING_NAMESPACE_ACADO

    // INTRODUCE THE VARIABLES:
    // -------------------------
    DifferentialState                 v,s,m;
    Control                           u    ;

    const double t_start =    0.0;
    const double t_end   =   10.0;
    const double h       =   0.01;

    DiscretizedDifferentialEquation  f(h) ;


    // DEFINE A DISCRETE-TIME SYTSEM:
    // ------------------------------
    f << next(s) == s + h*v;
    f << next(v) == v + h*(u-0.02*v*v)/m;
    f << next(m) == m - h*0.01*u*u;


    // DEFINE AN OPTIMAL CONTROL PROBLEM:
    // ----------------------------------
    OCP ocp( t_start, t_end, 50 );

    ocp.minimizeLagrangeTerm( u*u );
    ocp.subjectTo( f );

    ocp.subjectTo( AT_START, s ==  0.0 );
    ocp.subjectTo( AT_START, v ==  0.0 );
    ocp.subjectTo( AT_START, m ==  1.0 );

    ocp.subjectTo( AT_END  , s == 10.0 );
    ocp.subjectTo( AT_END  , v ==  0.0 );

    ocp.subjectTo( -0.01 <= v <= 1.3 );


    // DEFINE A PLOT WINDOW:
    // ---------------------
    GnuplotWindow window;
      window.addSubplot( s,"DifferentialState s" );
      window.addSubplot( v,"DifferentialState v" );
      window.addSubplot( m,"DifferentialState m" );
      window.addSubplot( u,"Control u" );
      window.addSubplot( PLOT_KKT_TOLERANCE,"KKT Tolerance" );
      window.addSubplot( 0.5 * m * v*v,"Kinetic Energy" );


    // DEFINE AN OPTIMIZATION ALGORITHM AND SOLVE THE OCP:
```

```
    // ------------------------------------------------------------
    OptimizationAlgorithm algorithm(ocp);

    algorithm.set( HESSIAN_APPROXIMATION, EXACT_HESSIAN );
    algorithm.set( KKT_TOLERANCE        , 1e-10         );

    algorithm << window;
    algorithm.solve();

    return 0;
}
```

In this example, the basic syntax for discrete-time dynamic systems is introduced. The notation of the form

```
    DiscretizedDifferentialEquation  f(h);
    f << next(s) == s + h*v;
    f << next(v) == v + h*(u-0.02*v*v)/m;
    f << next(m) == m - h*0.01*u*u;
```

defines a right hand side f of the form

$$\begin{aligned} s_{k+1} &= s_k + h v_k \\ v_{k+1} &= v_k + h \frac{u_k - 0.2\, v_k^2}{m_k} \\ m_{k+1} &= m_k - \frac{h}{100} u_k^2 \,. \end{aligned}$$

In the current version of the ACADO Toolkit only constant step sizes h are implemented but more advanced options will be made available in future releases. Note that the start time, end time, step size, and the number m of control intervals should be chosen in such a way that the relation

$$\frac{t_{\text{end}} - t_{\text{start}}}{h} = mn$$

holds for some integer n.

Chapter 4

Multi-Objective Optimization

The ACADO Toolkit offers advanced and systematic features for efficiently solving optimal control problems with multiple and conflicting objectives. Typically, these Multi-Objective Optimal Control Problems (MOOCPs) give rise to a set of Pareto optimal solutions instead of one single optimum. This chapter explains how to generate this Pareto set (or trade-off curve) efficiently.

4.1 Introduction to Multi-Objective Optimal Control Problems

4.1.1 Mathematical Formulation

In contrast to the general optimal control problem formulation, in which only one objective has to be minimized, the general MOOCP formulation requires the simultaneous minimization of m objectives:

$$\begin{array}{ll} \underset{x(\cdot),u(\cdot),p,T}{\text{minimize}} & \{\Phi_1(x(\cdot),u(\cdot),p,T),\ldots,\Phi_j(x(\cdot),u(\cdot),p,T),\ldots,\Phi_m(x(\cdot),u(\cdot),p,T)\} \\ \text{subject to:} & \\ \forall t\in[0,T]: & 0 \;=\; F(t,x(t),\dot{x}(t),u(t),p) \\ \forall t\in[0,T]: & 0 \;\leq\; h(t,x(t),u(t),p) \\ & 0 \;=\; r(x(0),x(T),p) \end{array} \tag{4.1}$$

Here, the function F still represents the model equation, with x the model states, u the control inputs, p the constant parameters, and T the final time. Now Φ_j denotes the j-th individual objective functional, while h and r are still the path constraints and boundary conditions of the optimal control problem.

4.1.2 Multi-Objective Optimization: Concepts and Philosophy

In contrast to the general optimal control problem formulation, in which only one objective has to be minimized, the general MOOCP formulation requires the simultaneous minimiza-

tion of m objectives. Before continuing, some concepts of Multi-Objective Optimization and scalarization methods are briefly introduced:

- *Pareto optimality concept*: A feasible point is considered to be a solution to a multi-objective optimization problem, and is called Pareto optimal, when there exist no other feasible point that improves one of the objectives without worsening at least one of the other objectives. The set of these mathematically equivalent point is often referred to as the Pareto set or Pareto front.

  Figure 4.1 illustrates the Pareto concept for a bi-objective optimization problem. The feasible objective space is depicted in blue and the Pareto set is diplayed in green. Hence, all points a to e are Pareto optimal, while f and g are not.

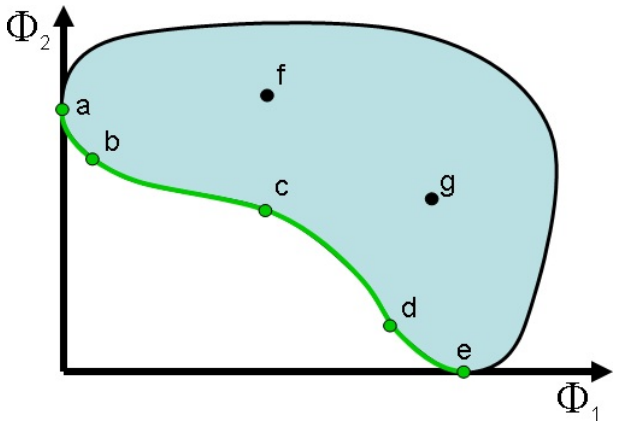


Figure 4.1: Pareto concept for a bi-objective optimization problem.

- *Scalarization methods for multi-objective optimization problems*: The rationale behind this class of solution methods is to convert the original multi-objective optimization problem into a series of parametric single objective optimization problems. By consistently varying the method's parameters an approximation of the Pareto front is obtained. Despite several intrinsic drawbacks, the convex Weighted Sum (WS) is still the most popular scalarization method. Alternatively, novel approaches that mitigate the drawbacks of the WS have been reported: Normal Boundary Intersection (NBI) and Normalized Normal Constraint (NNC).

4.1.3 Implementation in the `ACADO Toolkit`

The current structure and features of `ACADO` Multi-Objective are schematically depicted in Figure 4.2. As multi-objective scalarization techniques WS, NNC and NBI are available. To provide an approximation of the Pareto, single objective optimization problems have to be solved for different sets of the scalarization method's reformulation parameters. These sets of parameters are automatically generated by the weights generation scheme. Hot-start re-initialization options allow to seed up the solution of this series of single objective optimization problems. Afterwards, whenever necessary, non-Pareto optimal solutions can be removed by the Pareto filter. Finally, the resulting Pareto set can be exported and visualized. However, visualization is limited to cases with up to three objectives.

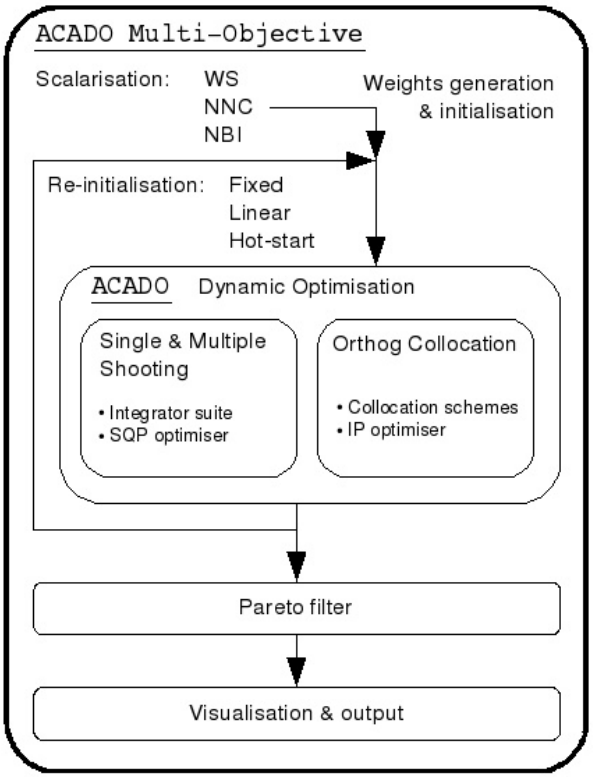


Figure 4.2: ACADO Multi-Objective functionality.

4.2 Static Optimization Problem with Two Objectives

4.2.1 Mathematical Formulation

For the static bi-objective problem only two scalar variables are involved: y_1 and y_2. The aim is to simultaneously minimize these two variables. However, both are bounded and have to satisfy a nonlinear constraint:

$$\begin{aligned} &\underset{y_1,y_2}{\text{minimize}} \quad \{y_1, y_2\} \\ &\text{subject to:} \\ &\qquad 0 \leq y_1 \leq 5.0 \\ &\qquad 0 \leq y_2 \leq 5.2 \\ &\qquad y_2 \geq 5\exp(-y_1) + 2\exp(-0.5(y_1-3)^2) \end{aligned} \tag{4.2}$$

4.2.2 Implementation in ACADO Syntax

The following piece of code illustrates how to set up the bi-objective optimization problem mentioned above. The Pareto set is first generated with 41 points based on NBI, and filtered afterwards using the Pareto filter algorithm. Both original and the filtered Pareto set are plotted and exported. This code is available in the directory

<install-dir>/examples/multi_objective as scalar2_nbi.cpp. The WS and NNC version are called scalar2_ws.cpp and scalar2_nnc.cpp, respectively.

```
#include <acado_optimal_control.hpp>
#include <include/acado_gnuplot/gnuplot_window.hpp>

int main( ){

    USING_NAMESPACE_ACADO


    // INTRODUCE THE VARIABLES:
    // -------------------------
    Parameter y1,y2;


    // DEFINE AN OPTIMIZATION PROBLEM:
    // -------------------------------
    NLP nlp;
    nlp.minimize( 0, y1 );
    nlp.minimize( 1, y2 );

    nlp.subjectTo( 0.0 <= y1 <= 5.0 );
    nlp.subjectTo( 0.0 <= y2 <= 5.2 );
    nlp.subjectTo( 0.0 <= y2 - 5.0*exp(-y1)
                           - 2.0*exp(-0.5*(y1-3.0)*(y1-3.0)) );


    // DEFINE A MULTI-OBJECTIVE ALGORITHM AND SOLVE THE NLP:
    // -----------------------------------------------------
    MultiObjectiveAlgorithm algorithm(nlp);

    algorithm.set(PARETO_FRONT_GENERATION,PFG_NORMAL_BOUNDARY_INTERSECTION);
    algorithm.set(PARETO_FRONT_DISCRETIZATION,41);
    algorithm.set(KKT_TOLERANCE,1e-12);

    // Minimize individual objective function
    algorithm.initializeParameters("scalar2_initial2.txt");
    algorithm.solveSingleObjective(1);

    // Minimize individual objective function
    algorithm.initializeParameters("scalar2_initial1.txt");
    algorithm.solveSingleObjective(0);

    // Generate Pareto set
    algorithm.solve();


    // GET THE RESULT FOR THE PARETO FRONT AND PLOT IT:
    // ------------------------------------------------
    VariablesGrid paretoFront;
    algorithm.getParetoFront( paretoFront );

    GnuplotWindow window1;
      window1.addSubplot( paretoFront,"Pareto Front y1 vs y2",
                                      "y1","y2", PM_POINTS );
    window1.plot( );

    FILE *file = fopen("scalar2_nbi_pareto.txt","w");
    paretoFront.print();
    file << paretoFront;
    fclose(file);
```

```
    // FILTER THE PARETO FRONT AND PLOT IT:
    // ------------------------------------
    algorithm.getParetoFrontWithFilter( paretoFront );

    GnuplotWindow window2;
      window2.addSubplot( paretoFront,"Pareto Front (with filter) y1 vs y2",
                                      "y1","y2", PM_POINTS );
    window2.plot( );

    FILE *file2 = fopen("scalar2_nbi_pareto_filtered.txt","w");
    paretoFront.print();
    file2 << paretoFront;
    fclose(file2);


    // PRINT INFORMATION ABOUT THE ALGORITHM:
    // --------------------------------------
    algorithm.printInfo();

    return 0;
}
```

Typical settings for multi-objective optimization:

- The choice of scalarization method: Currently, three approaches are available, namely Normal Boundary Intersection, Weighted Sum and Normalized Normal Constraint. The desired method can be selected in the option `PARETO_FRONT_GENERATION`:

  ```
  algorithm.set(PARETO_FRONT_GENERATION,PFG_NORMAL_BOUNDARY_INTERSECTION);
  //algorithm.set(PARETO_FRONT_GENERATION,PFG_WEIGHTED_SUM);
  //algorithm.set(PARETO_FRONT_GENERATION,PFG_NORMALIZED_NORMAL_CONSTRAINT);
  ```

  As both NBI and NNC require the individual minima, these points are first calculated, before the Pareto set is computed. In the current case, initial guesses are provided for both minimizations. However, for WS precomputing the individual minima is not required.

  ```
  // Minimize individual objective function
  algorithm.initializeParameters("scalar2_initial2.txt");
  algorithm.solveSingleObjective(1);

  // Minimize individual objective function
  algorithm.initializeParameters("scalar2_initial1.txt");
  algorithm.solveSingleObjective(0);

  // Generate Pareto set
  algorithm.solve();
  ```

- The number of Pareto points n_p: The number of Pareto points n_p relates to the number of points between two individual minima. Hence, the number of single objective optimizations is n_p for a bi-objective case and $\frac{1}{2}n_p(n_p+1)$ for a tri-objective case. Or for a general multi-objective case with m objectives this number is $\frac{1}{2m!}n_p \cdot (n_p+1)\cdots(n_p+m-2)$.

  ```
  algorithm.set( PARETO_FRONT_DISCRETIZATION, 41 );
  ```

- Hot-start re-initialization of the different single objective problems: To speed-up the solution of the different single objective problems, the hot-start strategy is used by default. Here, the solution of a previous single objective optimization is used to initialize the next one. This options can be switched of as follows.

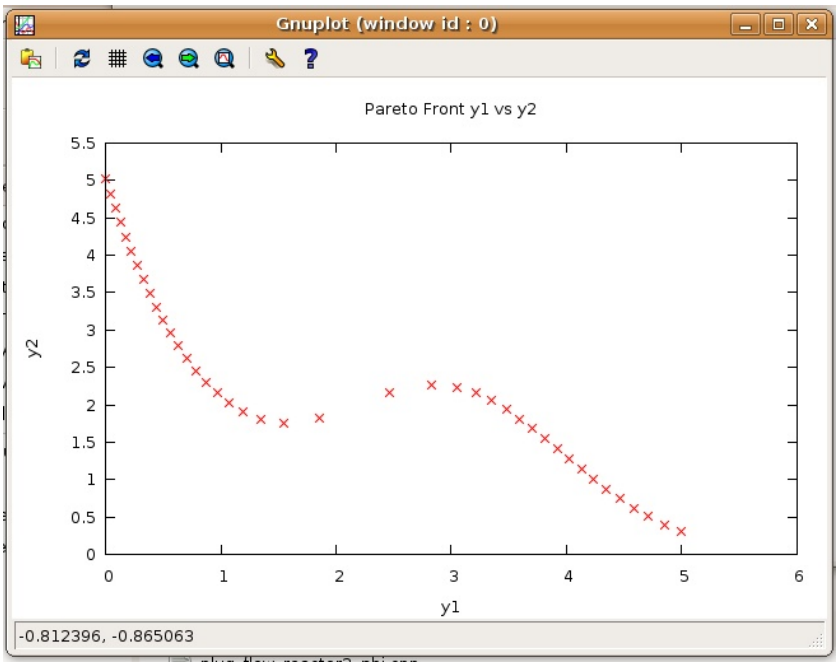
```
algorithm.set( PARETO_FRONT_HOTSTART, BT_FALSE );
```

- Pareto filter: As both NBI and NNC can produce non-Pareto optimal points, a Pareto filter can be employed to remove these points. The rationale behind this Pareto filter is a pairwise comparison of the Pareto candidates.

```
VariablesGrid paretoFront;
algorithm.getParetoFront( paretoFront );
algorithm.getParetoFrontWithFilter( paretoFront );
```

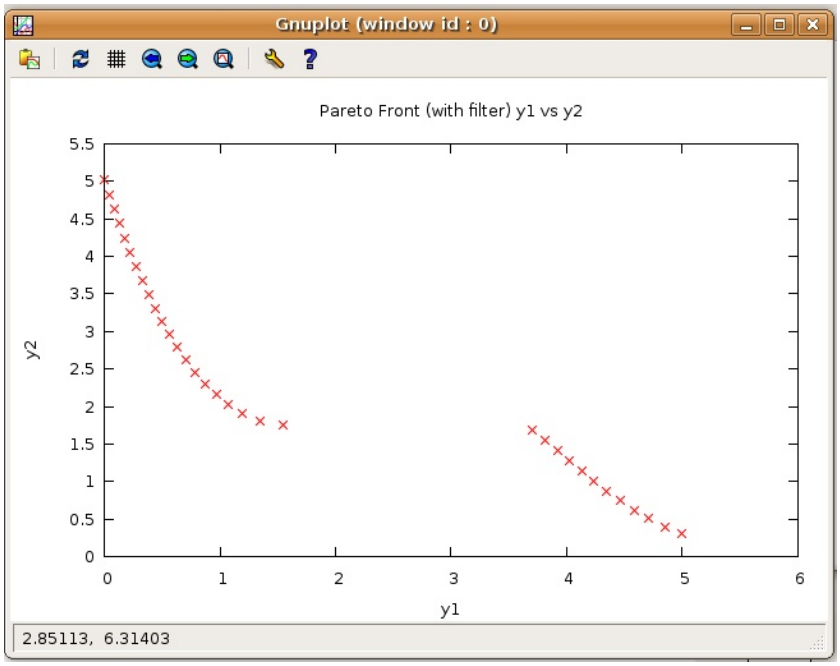
4.2.3 Numerical Results

The corresponding Pareto plot as returned by NBI looks as follows in GNUPLOT.



After filtering, part of the candidate solutions are removed and the following Pareto set is obtained.

The resulting Pareto sets (without and with filtering) are stored in separate files.



4.3 Static Optimization Problem with Three Objectives

4.3.1 Mathematical Formulation

For the static tri-objective problem only three scalar variables are involved: y_1, y_2 and y_3. The aim is to simultaneously minimize these three variables. However, all are bounded and have to satisfy a nonlinear constraint:

$$\begin{aligned} &\underset{y_1,y_2,y_3}{\text{minimize}} && \{y_1, y_2, y_3\} \\ &\text{subject to:} && \\ & && -5.0 \leq y_1 \leq 5.0 \\ & && -5.0 \leq y_2 \leq 5.0 \\ & && -5.0 \leq y_3 \leq 5.0 \\ & && y_1^2 + y_2^2 + y_3^2 - 4 \leq 0 \end{aligned} \tag{4.3}$$

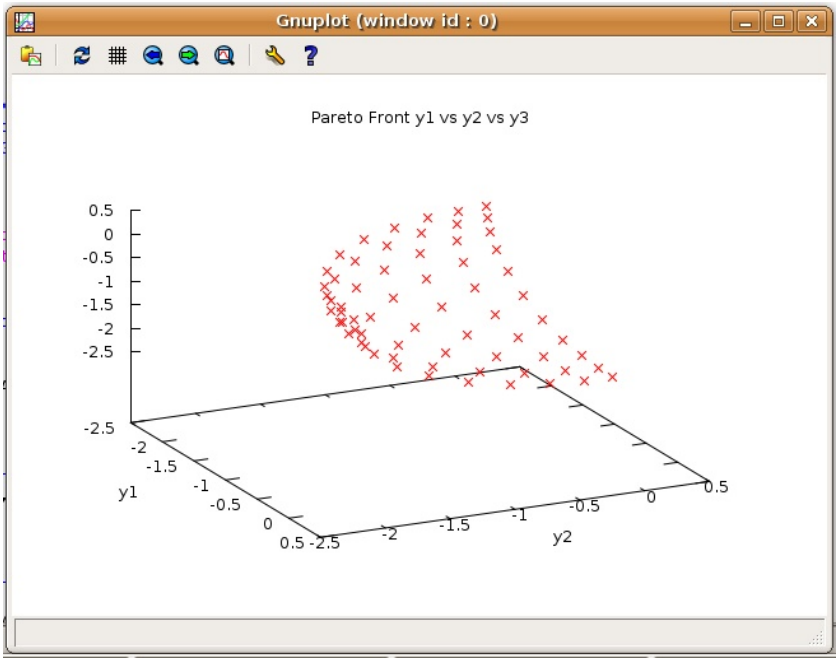
4.3.2 Implementation in ACADO Syntax

The following piece of code illustrates how to set up the tri-objective optimization problem mentioned above. The Pareto set is generated based on WS. The number of Pareto points n_p between two indidivual objectives is set to 11. Hence, this results in $\frac{1}{2}n_p(n_p+1) = 66$ single objective optimization problems to be solved and also in 66 points on the global Pareto front. This code is available in the directory `<install-dir>/examples/multi_objective` as `scalar3_ws.cpp`. The NBI and NNC version are called `scalar3_nbi.cpp` and `scalar3_nnc.cpp`, respectively.

```
#include <acado_optimal_control.hpp>
#include <include/acado_gnuplot/gnuplot_window.hpp>

int main( ){

    USING_NAMESPACE_ACADO


    // INTRODUCE THE VARIABLES:
    // -----------------------
    Parameter y1,y2,y3;


    // DEFINE AN OPTIMIZATION PROBLEM:
    // -------------------------------
    NLP nlp;
    nlp.minimize( 0, y1 );
    nlp.minimize( 1, y2 );
    nlp.minimize( 2, y3 );

    nlp.subjectTo( -5.0 <= y1 <= 5.0 );
    nlp.subjectTo( -5.0 <= y2 <= 5.0 );
    nlp.subjectTo( -5.0 <= y3 <= 5.0 );

    nlp.subjectTo( y1*y1+y2*y2+y3*y3 <= 4.0 );


    // DEFINE A MULTI-OBJECTIVE ALGORITHM AND SOLVE THE NLP:
    // -----------------------------------------------------
    MultiObjectiveAlgorithm algorithm(nlp);

    algorithm.set( PARETO_FRONT_GENERATION, PFG_WEIGHTED_SUM );
    algorithm.set( PARETO_FRONT_DISCRETIZATION, 11 );

    // Generate Pareto set
    algorithm.solve();

    algorithm.getWeights("scalar3_ws_weights.txt");


    // GET THE RESULT FOR THE PARETO FRONT AND PLOT IT:
    // ------------------------------------------------
    VariablesGrid paretoFront;
    algorithm.getParetoFront( paretoFront );
    paretoFront.print();

    GnuplotWindow window;
      window.addSubplot3D( paretoFront, "Pareto Front y1 vs y2 vs y3",
                                        "y1","y2", PM_POINTS );
    window.plot( );

    FILE *file = fopen("scalar3_ws_pareto.txt","w");
    paretoFront.print();
    file << paretoFront;
    fclose(file);


    // PRINT INFORMATION ABOUT THE ALGORITHM:
    // --------------------------------------
    algorithm.printInfo();

    return 0;
}
```

4.3.3 Numerical Results

The corresponding Pareto surface as returned by WS looks as follows in GNUPLOT. Note however that it not possible to visualize Pareto fronts for more than three objectives in ACADO.



The resulting Pareto set is stored in a separate file `scalar3_ws_pareto.txt`. These output files can be generated for optimization problems with any number of objectives.

4.4 Dynamic Optimization Problem with Two Objectives

This section explains how to set up multi-objective optimal control problems in ACADO. As an example the optimal and safe operation of a jacketed tubular reactor is considered. Inside the tubular reactor an exothermic irreversible first order reaction takes place. The heat produced by this reaction is removed through the surrounding jacket. In addition, it is assumed that the reactor operates in steady-state conditions and that the fluid flow as a plug through the tube. The aim is to find an optimal profile along the reactor for the temperature of the fluid in the jacket such that conversion and energy costs are minimized.

4.4.1 Mathematical Formulation

The optimal control problem involves two states: the dimensionless temperature x_1 and the dimensionless reactant concentration x_2 and one control: the dimensionless jacket fluid temperature u. The reactor length has been fixed to L. The conversion objective involves the minimization of the reactant concentration at the outlet: $C_F(1 - x_1(L))$ with C_F the reactant concentration in the feed stream. The energy objective relates to the minimization of the terminal heat loss by penalizing deviations between the reactor in- and

outlet temperature: $\frac{T_F^2}{K_1}x_2^2(L)$. The conditions at the reactor inlet are given and equal to the values of the feed stream. The dimensionless concentration is intrinsically bounded between 0 and 1, whereas upper and lower constraints are imposed on the jacket and reactor temperatures for safety and constructive reasons.

$$
\begin{aligned}
&\underset{x(\cdot),u(\cdot)}{\text{minimize}} && \left\{ C_F(1-x_1(L)),\ \frac{T_F^2}{K_1}x_2^2(L) \right\} \\
&\text{subject to:} \\
&\forall z \in [0,L]: && \frac{dx_1}{dz} = \frac{\alpha}{v}(1-x_1)e^{\frac{\gamma x_2}{1+x_2}} \\
& && \frac{dx_2}{dz} = \frac{\alpha\delta}{v}(1-x_1)e^{\frac{\gamma x_2}{1+x_2}} + \frac{\beta}{v}(u-x_2) \\
&\forall z \in [0,L]: && 0.0 \leq x_1 \leq 1.0 \\
& && x_{2,\min} \leq x_2 \leq x_{2,\max} \\
& && u_{\min} \leq u \leq u_{\max} \\
&\text{at } z = 0: && x_1(0) = 0.0 \\
& && x_2(0) = 0.0
\end{aligned}
\tag{4.4}
$$

Note that the time t as independent variable has been replaced by the spatial coordinate z, since optimal spatial profiles along the length of the reactor are required.

4.4.2 Implementation in ACADO Syntax

The following piece of code illustrates how to set up the multi-objective optimal control problem mentioned above. NBI is used to approximate the Pareto set with 11 points. The pareto front is plotted and exported. Also all corresponding optimal state and control profiles are exported. This code is available in the directory `<install-dir>/examples/multi_objective` as `plug_flow_reactor_nbi.cpp`. The WS and NNC version are called `plug_flow_reactor_ws.cpp` and `plug_flow_reactor_nnc.cpp`, respectively.

```
#include <acado_optimal_control.hpp>
#include <include/acado_gnuplot/gnuplot_window.hpp>

int main( ){

    USING_NAMESPACE_ACADO

    // INTRODUCE FIXED PARAMETERS:
    // ---------------------------
    #define  v      0.1
    #define  L      1.0
    #define  Beta   0.2
    #define  Delta  0.25
```

```
#define  E      11250.0
#define  k0     1E+06
#define  R      1.986
#define  K1     250000.0
#define  Cin    0.02
#define  Tin    340.0


// INTRODUCE THE VARIABLES:
// -----------------------
DifferentialState     x1,x2;
Control               u    ;
DifferentialEquation  f( 0.0, L );


// DEFINE A DIFFERENTIAL EQUATION:
// -------------------------------
double Alpha, Gamma;
Alpha = k0*exp(-E/(R*Tin));
Gamma = E/(R*Tin);

f << dot(x1) ==  Alpha       /v * (1.0-x1) * exp((Gamma*x2)/(1.0+x2));
f << dot(x2) == (Alpha*Delta)/v * (1.0-x1) * exp((Gamma*x2)/(1.0+x2))
                                           + Beta/v * (u-x2);


// DEFINE AN OPTIMAL CONTROL PROBLEM:
// ----------------------------------
OCP ocp( 0.0, L, 50 );
// Solve conversion optimal problem
ocp.minimizeMayerTerm( 0, Cin*(1.0-x1) );
// Solve energy optimal problem (perturbed by small conversion cost;
// otherwise the problem is ill-defined.)
ocp.minimizeMayerTerm( 1, (pow((Tin*x2),2.0)/K1) + 0.005*Cin*(1.0-x1) );

ocp.subjectTo( f );

ocp.subjectTo( AT_START, x1 ==  0.0 );
ocp.subjectTo( AT_START, x2 ==  0.0 );

ocp.subjectTo(  0.0           <= x1 <=  1.0           );
ocp.subjectTo( (280.0-Tin)/Tin <= x2 <= (400.0-Tin)/Tin );
ocp.subjectTo( (280.0-Tin)/Tin <= u  <= (400.0-Tin)/Tin );


// DEFINE A MULTI-OBJECTIVE ALGORITHM AND SOLVE THE OCP:
// -----------------------------------------------------
MultiObjectiveAlgorithm algorithm(ocp);

algorithm.set(INTEGRATOR_TYPE,INT_BDF);
algorithm.set(KKT_TOLERANCE,1e-9);

algorithm.set(PARETO_FRONT_GENERATION,PFG_NORMAL_BOUNDARY_INTERSECTION);
algorithm.set(PARETO_FRONT_DISCRETIZATION,11);

// Minimize individual objective function
algorithm.solveSingleObjective(0);

// Minimize individual objective function
algorithm.solveSingleObjective(1);

// Generate Pareto set
algorithm.solve();
```

```
    algorithm.getWeights("plug_flow_reactor_nbi_weights.txt");
    algorithm.getAllDifferentialStates("plug_flow_reactor_nbi_states.txt");
    algorithm.getAllControls("plug_flow_reactor_nbi_controls.txt");


    // VISUALIZE THE RESULTS IN A GNUPLOT WINDOW:
    // ------------------------------------------
    VariablesGrid paretoFront;
    algorithm.getParetoFront( paretoFront );

    GnuplotWindow window1;
      window1.addSubplot( paretoFront,"Pareto Front (conversion vs. energy)",
                                      "OUTLET CONCENTRATION","ENERGY",
                                      PM_POINTS );
    window1.plot( );


    // PRINT INFORMATION ABOUT THE ALGORITHM:
    // --------------------------------------
    algorithm.printInfo();


    // SAVE INFORMATION:
    // -----------------
    FILE *file = fopen("plug_flow_reactor_nbi_pareto.txt","w");
    paretoFront.print();
    file << paretoFront;
    fclose(file);

    return 0;
}
```

Remarks:

- Exporting the scalarization parameters: The sequence of the different values for the scalarization parameters ("weights") can be exported to a txt file.

```
  algorithm.getWeights("plug_flow_reactor_nbi_weights.txt");
```

- Exporting the optimal control and state profiles: Also the optimal control and state profiles along the Pareto set can be exported as txt files.

```
  algorithm.getWeights("plug_flow_reactor_nbi_weights.txt");
  algorithm.getAllDifferentialStates("plug_flow_reactor_nbi_states.txt");
  algorithm.getAllControls("plug_flow_reactor_nbi_controls.txt");
```

  To indicate the order of the different solutions, each time `MOx` is added to the name, with x the position in the series of parametric single objective optimization problems. As in the current case 11 Pareto points are required, the state profiles are named from `MOOplug_flow_reactor_nbi_states.txt` to `MO10plug_flow_reactor_nbi_states.txt` and the control profiles are given names from `MOOplug_flow_reactor_nbi_controls.txt` to `MO10plug_flow_reactor_nbi_controls.txt`. Note that corresponding values for the scalarization parameters can be found in the weights file `plug_flow_reactor_nbi_weights.txt`.

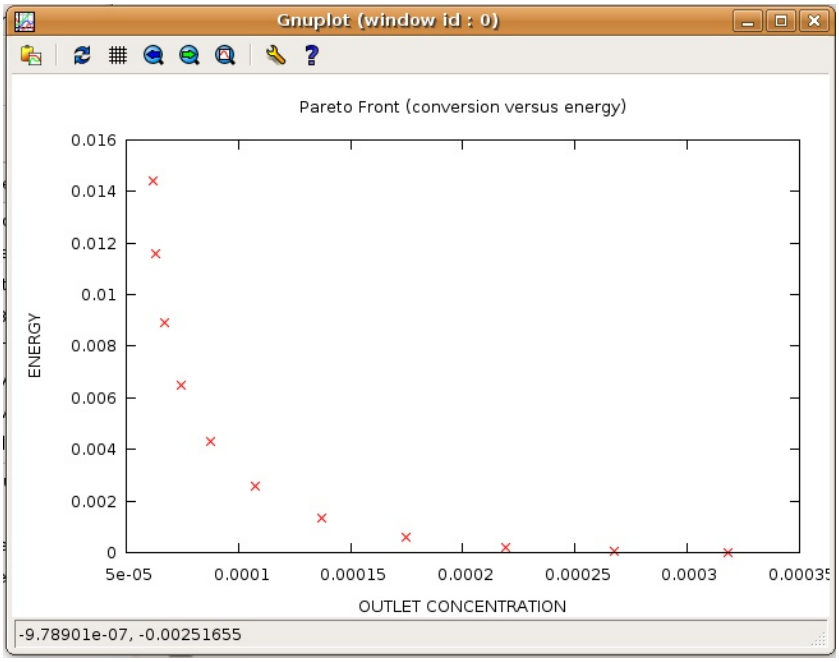
- Perturbation of energy cost: In the current case, a fraction of the conversion cost is added to the energy cost as the pure energy optimal case is not uniquely defined.

(There are infinitely many profiles with an outlet temperature equal to the inlet temperature.) However, adding this small focus on conversion leads to chemically consistent and gradual results. Moreover, when comparing the current results to results reported in literature, no significant differences are observed.

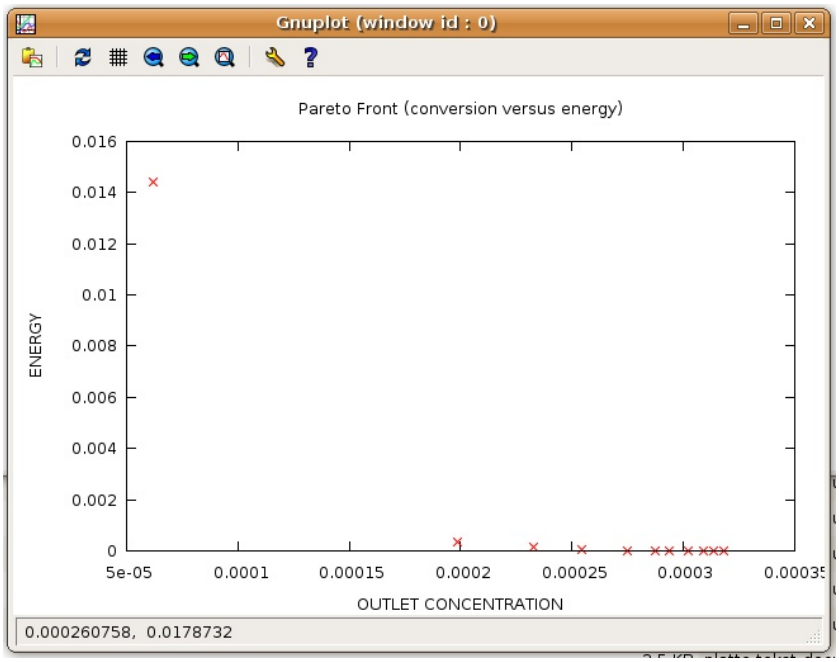
```
// Define energy cost (perturbed by small conversion cost;
// otherwise the problem is ill-defined.)
ocp.minimizeMayerTerm( 1,(pow((Tin*x2),2.0)/K1) + 0.005*Cin*(1.0-x1) );
```

4.4.3 Numerical Results

The corresponding Pareto plot as returned by NBI looks as follows in GNUPLOT:



When comparing with the result provided by WS, NBI clearly yields a much nicer spread of the Pareto points along the Pareto front:


Gnuplot (window id : 0)
Pareto Front (conversion versus energy)
ENERGY
0.016
0.014
0.012
0.01
0.008
0.006
0.004
0.002
0
5e-05
0.0001
0.00015
0.0002
0.00025
0.0003
OUTLET CONCENTRATION
0.000260758, 0.0178732

Chapter 5

State and Parameter Estimation

5.1 A State and Parameter Estimation Tutorial

This section explains how to setup a simple parameter estimation problem with ACADO. As an example a very simple pendulum model is considered. The aim is to estimate the length of the cable as well as a friction coefficient from a measurent of the excitation of the pendulum at several time points.

5.1.1 Mathematical Formulation

We consider a very simple pendulum model with two differential states φ and ω representing the excitation angle and the corresponding angular velocity of the pendulum, respectively. Moreover, the model for the pendulum depends on two parameters: the length of the cable is denoted by l while the friction coefficient of the pendulum is denoted by α. The parameter estimation problem of our interest has now the following form:

$$
\begin{aligned}
&\underset{\varphi(\cdot),\omega(\cdot),l,\alpha}{\text{minimize}} \quad \sum_{i=1}^{10} (\varphi(t_i) - \eta_i)^2 \\
&\text{subject to:} \\
\forall t \in [0,T]: \quad & \dot{\varphi}(t) = \omega(t) \\
\forall t \in [0,T]: \quad & \dot{\omega}(t) = -\tfrac{g}{l}\sin\varphi(t) - \alpha\omega(t) \\
& 0.0 \leq \alpha \leq 4.0 \\
& 0.0 \leq l \leq 2.0
\end{aligned}
\tag{5.1}
$$

Here, we assume that the state φ has been measured at 10 points in time which are denoted by $t_1, \ldots, t_{10}$ while the corresponding measurement values are $\eta_1, ldots, \eta_{10}$. Note that the above formulation does not only regard the parameters l and α as free variables. The initial values of two states φ and ω are also assumed to be unknown and must be estimated from the measurements, too.

5.1.2 Implementation in ACADO Syntax

The implementation of the above optimization problem is similar to the standard optimal control problem implementation which has been discussed in section 3.1. However, the main difference is now that the measurements have to be provided and that objective has a special least-squares form:

```
#include <acado_toolkit.hpp>
#include <include/acado_gnuplot/gnuplot_window.hpp>

int main( ){

    USING_NAMESPACE_ACADO

    DifferentialState           phi, omega;     // the states of the pendulum
    Parameter                   l, alpha  ;     // its length and the friction
    const double                g = 9.81  ;     // the gravitational constant
    DifferentialEquation        f         ;     // the model equations
    Function                    h         ;     // the measurement function

    VariablesGrid measurements;                 // read the measurements
    measurements = readFromFile( "data.txt" );  // from a file.

    //  ---------------------------------------
    OCP ocp(measurements.getTimePoints());     // construct an OCP
    h << phi                                ;     // the state phi is measured
    ocp.minimizeLSQ( h, measurements )   ;     // fit h to the data

    f << dot(phi  ) == omega                ;     // a symbolic implementation
    f << dot(omega) == -(g/l) *sin(phi )          // of the model
                       - alpha*omega        ;     // equations

    ocp.subjectTo( f                      );     // solve OCP s.t. the model,
    ocp.subjectTo( 0.0 <= alpha <= 4.0    );     // the bounds on alpha
    ocp.subjectTo( 0.0 <=   l   <= 2.0    );     // and the bounds on l.
    //  ---------------------------------------

    GnuplotWindow window;
      window.addSubplot( phi  , "The angle phi", "time [s]", "angle [rad]" );
      window.addSubplot( omega, "The angular velocity dphi"                );
      window.addData( 0, measurements(0) );

    //  ---------------------------------------
    ParameterEstimationAlgorithm algorithm(ocp); // the parameter estimation
    algorithm << window;
    algorithm.solve();                           // solves the problem

    return 0;
}
```

Note that the measurement are in this example provided in form of the text file `data.txt` which has the following contents:

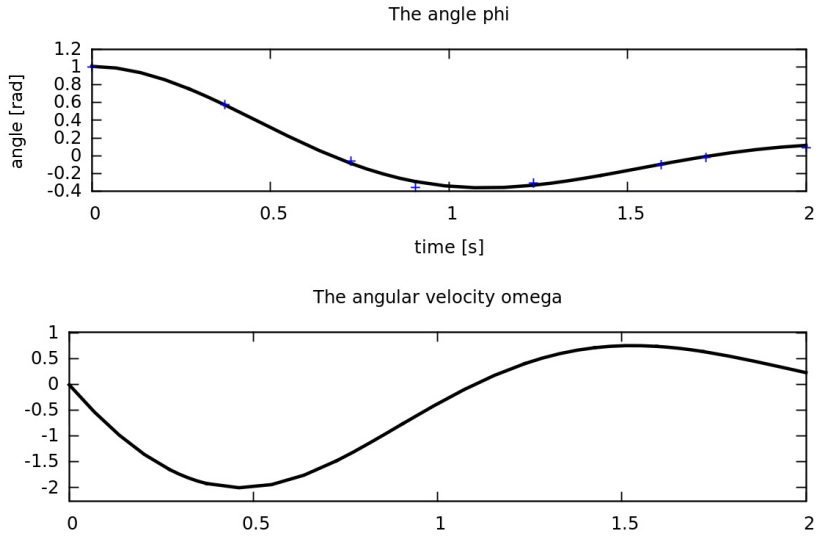
```
    TIME POINTS     MEASUREMENT OF PHI
   ------------     ------------------
    0.00000e+00      1.00000e+00
    2.72321e-01      nan
    3.72821e-01      5.75146e-01
    7.25752e-01     -5.91794e-02
    9.06107e-01     -3.54347e-01
```

```
1.23651e+00   -3.03056e-01
1.42619e+00    nan
1.59469e+00   -9.64208e-02
1.72029e+00   -1.97671e-02
2.00000e+00    9.35138e-02
```

At two time points the measurement was not successful leading to `nan` entries in the data file. In addition, the time points at which the measurements have been taken are not equidistant. Note that `ACADO` detects automatically the number of valid measurements in the file. Moreover, it is not necessary to specify any dimensions while the initialization is auto-generated, too.

5.1.3 Numerical Results

The parameter estimation algorithm chooses by default a Gauss Newton method. Running the above piece of code leads to the following output:



The output on the terminal is:

```
ACADO Toolkit::SCPmethod -- A Sequential Quadratic Programming Algorithm.
Copyright (C) 2008-2011 by Boris Houska and Hans Joachim Ferreau, K.U.Leuven.
Developed within the Optimization in Engineering Center (OPTEC) under
supervision of Moritz Diehl. All rights reserved.

ACADO Toolkit is distributed under the terms of the GNU Lesser
General Public License 3 in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU Lesser General Public License for more details.

  1:  KKT tolerance = 1.933e+00    objective value = 8.8999e-04
  2:  KKT tolerance = 1.692e-04    objective value = 8.6602e-04
  3:  KKT tolerance = 1.929e-05    objective value = 8.7832e-04
  4:  KKT tolerance = 3.827e-08    objective value = 8.7797e-04
```

```
convergence achieved.
```

Note that the algorithm converges rapidly within 4 iterations as expected for a Gauss-Newton method. Recall that the Gauss-Newton method works very well for least-squares problem, where either the problem is almost linear or the least-squares residuum is small.

5.1.4 A Posteriori Analysis

Once we are able to solve the parameter estimation we are usually interested in the results for the parameters. In addition, variance-covariance information about the quality of the fit is avaliable. A typical piece of code to get the output is as follows:

```
#include <acado_toolkit.hpp>
#include <include/acado_gnuplot/gnuplot_window.hpp>

int main( ){

    USING_NAMESPACE_ACADO


    // ... (IMPLEMENTATION OF THE OCP AS ABOVE) ...


    ParameterEstimationAlgorithm algorithm(ocp);
    algorithm.solve();


    // GET THE OPTIMAL PARAMETERS:
    // ---------------------------
    VariablesGrid params;
    algorithm.getParameters( params );

    // GET THE VARIANCE COVARIANCE IN THE SOLUTION:
    // --------------------------------------------
    Matrix var;
    algorithm.getParameterVarianceCovariance( var );


    // PRINT THE RESULT ON THE TERMINAL:
    // ---------------------------------
    printf("\n\nResults for the parameters: \n");
    printf("-------------------------------------------------------\n");
    printf("  l      =  %.3e +/- %.3e \n", params(0,0), sqrt( var(0,0) ) );
    printf("  alpha  =  %.3e +/- %.3e \n", params(0,1), sqrt( var(1,1) ) );
    printf("-------------------------------------------------------\n\n\n");

    return 0;
}
```

Running the above piece of code leads to the common output for the results of a parameter estimation problem:

```
Results for the parameters:
-----------------------------------------------
  l      =  1.001e+00  +/-  1.734e+00
  alpha  =  1.847e+00  +/-  4.060e+00
-----------------------------------------------
```

Note that the computation of the variance covariance matrix is based on a linear approximation in the optimal solution. The details of this strategy have originally been published by Bock [4].

Part III

Model Predictive Control and Closed-Loop Simulations

Chapter 6

Process for Closed-Loop Simulations

The ACADO Toolkit also provides a built-in simulation environment for perfoming realistic closed-loop simulations. Its main components are the Process for setting up a simulation of the process to be controlled, as described in this chapter, and the Controller for implementing the closed-loop controller.

The Process class has as members a dynamic system, comprising a differential equation and an optional output function, modelling the process as well as an integrator capable of simulating these model equations. The simulation uses (optimised) control inputs from the controller, which might be subject to noise or delays that can be introduced via an optional Actuator. In addition, so-called process disturbances can be specified by the user for setting up arbitrary disturbance scenarios for the simulation. Finally, the outputs obtained by integrating the model equations can again be subject to noise or delays introduced via an optional Sensor. It is important to note that the model used for simulating the process does not need to be the same as specified within the optimal control formulations within the controller.

6.1 Setting-Up a Simple Process

This section explains how to setup a simple Process for MPC simulations. As a guiding example, we consider a simple actively damped quarter car model.

6.1.1 Mathematical Formulation

We consider a first principle quarter car model with active suspension. The four differential states of this model x_B, v_B, x_W, and v_W are the position/velocity of the body/wheel, respectively. Our control input is a limited damping force F acting between the body and the wheel. The road, denoted by the variable R, is considered as an (unknown) external

disturbance. The dynamic equations have the following form:

$$f: \quad \begin{pmatrix} \dot{x}_{\mathrm{B}}(t) \\ \dot{x}_{\mathrm{W}}(t) \\ \dot{v}_{\mathrm{B}}(t) \\ \dot{v}_{\mathrm{W}}(t) \end{pmatrix} = \begin{pmatrix} v_{\mathrm{B}}(t) \\ v_{\mathrm{W}}(t) \\ \frac{1}{m_{\mathrm{B}}}\left[-k_{\mathrm{S}}x_{\mathrm{B}}(t) + k_{\mathrm{S}}x_{\mathrm{W}}(t) + F(t)\right] \\ \frac{1}{m_{\mathrm{W}}}\left[-k_{\mathrm{T}}x_{\mathrm{B}}(t) - (k_{\mathrm{T}} + k_{\mathrm{S}})x_{\mathrm{W}}(t) + k_{\mathrm{T}}R(t) - F(t)\right] \end{pmatrix} \tag{6.1}$$

Within our simulation, we start at $x_B = 0.01$ and all other states at zero. Moreover, we treat the mass of the body m_B as manipulatable (time-constant) parameter, whereas fixed values are assigned to all other quantities.
In order to illustrate the concept, let us assume that not all states can be measured directly but only the first one together with a combination of the third one and the control input. For realizing this, we introduce the following output function:

$$g: \quad \begin{pmatrix} g_1(t) \\ g_2(t) \end{pmatrix} = \begin{pmatrix} x_{\mathrm{B}}(t) \\ 500v_{\mathrm{B}}(t) + F(t) \end{pmatrix} \tag{6.2}$$

6.1.2 Implementation in ACADO Syntax

The following piece of code shows how to implement a `Process` simulation based on this quarter car model. It comprises four main steps:

1. Introducing all variables and constants.
2. Setting up the quarter car ODE model together with the output function; these two functions form the `DynamicSystem` used for the `Process` simulation. (In case you do not define an output function, the `Process` output will be all differential states.)
3. Setting up the `Process`, which comprises at least to define a dynamic system to be used for simulation together with the information which integrator is to be used. In our example, also integrator options are set, initial values for the differential states are defined and a plot window is specified. As the dynamic system of the quarter car comprises an external disturbance, we also have to specify values for it. This is done by reading the disturbance data from the file `road.txt`.
4. Simulating the `Process` by first initializing it, passing the start time of the simulation (otherwise simulation starts at 0.0), and second run it with given values for the control input and the parameter input. Afterwards, results can be obtained and are plotted according to the previously flushed plot window.

```
#include <acado_toolkit.hpp>
#include <include/acado_gnuplot/gnuplot_window.hpp>

int main( ) {

    USING_NAMESPACE_ACADO


    // INTRODUCE THE VARIABLES:
    // ------------------------
```

```
DifferentialState xB;
DifferentialState xW;
DifferentialState vB;
DifferentialState vW;

Disturbance R;
Control F;

Parameter mB;
double mW = 50.0;
double kS = 20000.0;
double kT = 200000.0;


// DEFINE THE DYNAMIC SYSTEM:
// --------------------------
DifferentialEquation f;

f << dot(xB) == vB;
f << dot(xW) == vW;
f << dot(vB) == ( -kS*xB + kS*xW + F ) / mB;
f << dot(vW) == (  kS*xB - (kT+kS)*xW + kT*R - F ) / mW;

OutputFcn g;
g << xB;
g << 500.0*vB + F;

DynamicSystem dynSys( f,g );


// SETUP THE PROCESS:
// ------------------
Process myProcess;

myProcess.setDynamicSystem( dynSys,INT_RK45 );
myProcess.set( ABSOLUTE_TOLERANCE,1.0e-8 );

Vector x0( 4 );
x0.setZero( );
x0( 0 ) = 0.01;

myProcess.initializeStartValues( x0 );
myProcess.setProcessDisturbance( "road.txt" );

myProcess.set( PLOT_RESOLUTION,HIGH );

GnuplotWindow window;
  window.addSubplot( xB, "Body Position [m]" );
  window.addSubplot( xW, "Wheel Position [m]" );
  window.addSubplot( vB, "Body Velocity [m/s]" );
  window.addSubplot( vW, "Wheel Velocity [m/s]" );

  window.addSubplot( F,"Damping Force [N]" );
  window.addSubplot( mB,"Body Mass [kg]" );
  window.addSubplot( R, "Road Disturbance" );
  window.addSubplot( g(0),"Output 1" );
  window.addSubplot( g(1),"Output 2" );

myProcess << window;


// SIMULATE AND GET THE RESULTS:
// -----------------------------
VariablesGrid u( 1,0.0,1.0,6 );
```

```
    u( 0,0 ) = 10.0;
    u( 1,0 ) = -200.0;
    u( 2,0 ) = 200.0;
    u( 3,0 ) = 0.0;
    u( 4,0 ) = 0.0;
    u( 5,0 ) = 0.0;

    Vector p( 1 );
    p(0) = 350.0;

    myProcess.init( 0.0 );
    myProcess.run( u,p );


    VariablesGrid xSim, ySim;

    myProcess.getLast( LOG_SIMULATED_DIFFERENTIAL_STATES,xSim );
    xSim.print( "Simulated Differential States" );

    myProcess.getLast( LOG_PROCESS_OUTPUT,ySim );
    ySim.print( "Process Output" );

    return 0;
}
```

The file `road.txt` contains the following disturbance data:

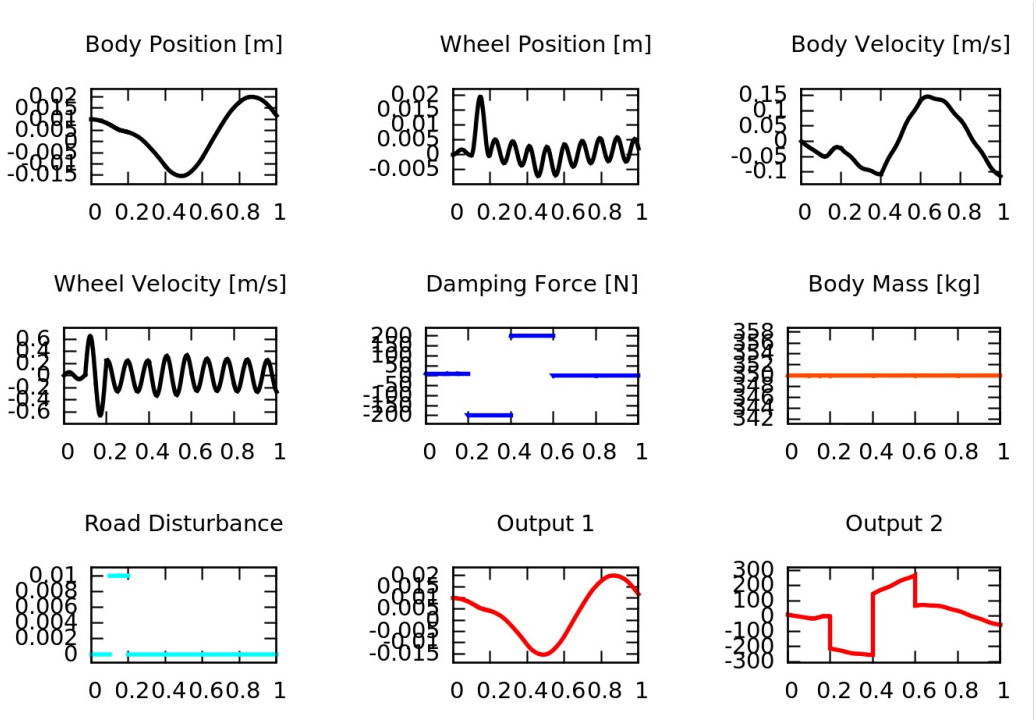
```
DATA FILE: road.txt

_____________________________

TIME        W

0.0         0.00
0.1         0.01
0.15        0.01
0.2         0.00
5.0         0.00

_____________________________
```

6.1.3 Simulation Results

If we run the above piece of code in ACADO, the corresponding GNUPLOT output should be as follows:

Note that this is only a simulation with user-specified control inputs; no feedback control is applied.

We end this section with proving a list of all results that can be obtained from the `Process` after simulation:



Logging name:	Short Description:
LOG_PROCESS_OUTPUT	All process outputs as specified via the output function
LOG_SIMULATED_DIFFERENTIAL_STATES	All differential states as simulated within the Process
LOG_SIMULATED_ALGEBRAIC_STATES	All algebraic states as simulated within the Process
LOG_SIMULATED_CONTROLS	All control inputs as simulated within the Process
LOG_SIMULATED_PARAMETERS	All parameter inputs as simulated within the Process
LOG_SIMULATED_DISTURBANCES	All external disturbances as simulated within the Process
LOG_NOMINAL_CONTROLS	All nominal control inputs as given to the Process
LOG_NOMINAL_PARAMETERS	All nominal parameter inputs as given to the Process

6.2 Advanced Features

This section introduces more advanced features of the ACADO Process for MPC simulations. In particular, actuator and sensor behaviour can be simulated to yield more realistic results.

6.2.1 Adding a Actuator to the Process

Actuator effects can be simulated by adding an Actuator block to the Process as demonstrated in the following code fragment:

```
// to be added to code fragment from previous section ...

// SETUP NOISE:
// ------------
Vector mean( 1 ), amplitude( 1 );
```

```
mean.setZero( );
amplitude.setAll( 50.0 );

GaussianNoise myNoise( mean,amplitude );

// SETUP ACTUATOR:
// ---------------
Actuator myActuator( 1,1 );

myActuator.setControlNoise( myNoise,0.1 );
myActuator.setControlDeadTimes( 0.1 );

myActuator.setParameterDeadTimes( 0.2 );

// ...

myProcess.setActuator( myActuator );
```

The code fragment shows how to setup a class generating one-dimensional, Gaussian noise with given amplitude (standard deviation) and mean. Afterwards, an `Actuator` accepting one control and one parameter input is defined. The previously defined noise will generated with a sampling time of 0.1 second and added to the control input. Moreover, both control and parameter inputs are delayed by the actuator by 0.1 and 0.2 seconds, respectively.

6.2.2 Adding a Sensor to the Process

Sensor effects can be simulated analogously by adding a `Sensor` block to the `Process` as demonstrated in the following code fragment:

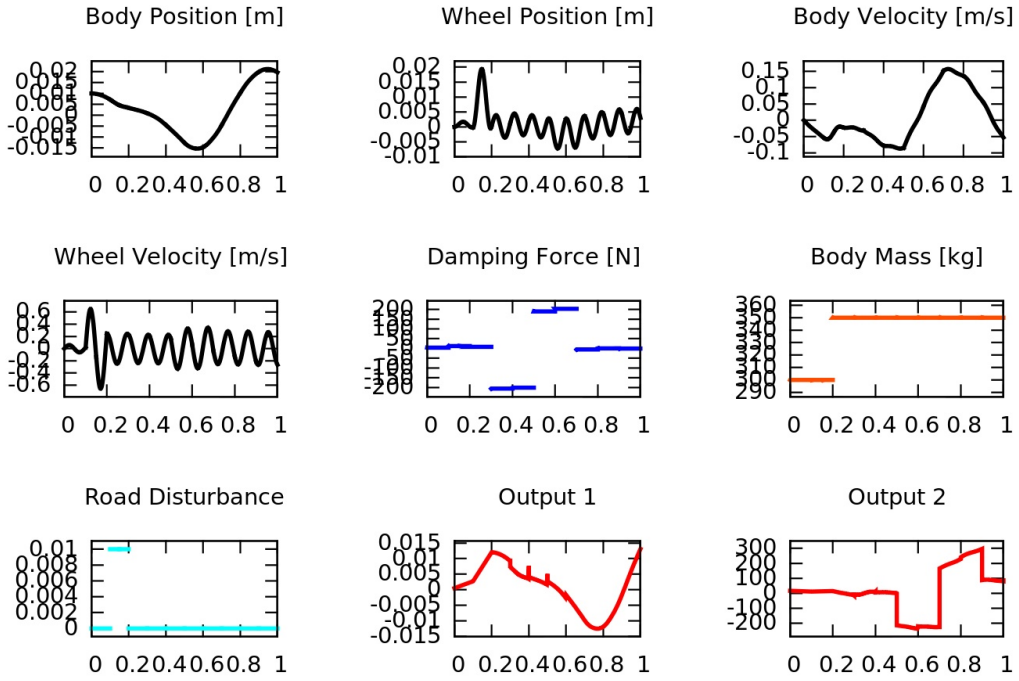
```
// to be added to code fragment from previous section ...

// SETUP NOISE:
// ------------
Vector mean( 1 ), amplitude( 1 );

mean.setZero( );
amplitude.setAll( 0.005 );
UniformNoise myOutputNoise1( mean,amplitude );

mean.setAll( 10.0 );
amplitude.setAll( 50.0 );
GaussianNoise myOutputNoise2( mean,amplitude );

// SETUP SENSOR:
// -------------
Sensor mySensor( 2 );

mySensor.setOutputNoise( 0,myOutputNoise1,0.1 );
mySensor.setOutputNoise( 1,myOutputNoise2,0.1 );

mySensor.setOutputDeadTimes( 0.2 );

// ...

myProcess.setSensor( mySensor );
```

In this code fragment, noise is setup that is to added to the process ouput. Two different instances of the noise class with different means and amplitudes are instantiated and

assigned to the two components of the process output. Note that to the first component uniformly-distributed noise is added, while Gaussian noise is used for the second component in order to illustrate the flexibility of the concept. Finally, all output components are delayed by a dead time of 0.2 seconds.

6.2.3 Simulation Results

For completeness, we show the GNUPLOT window of the quarter car process simulation from section 6.1 with the additions discussed before (and the parameter initialized at 300 to make the dead time visible):



Note that this is only a simulation with user-specified control inputs; no feedback control is applied.

6.2.4 List of Algorithmic Options

We end this section with proving a list of the most common options that can be set when performing `Process` simulations:

Option Name:	Possible Values:	Short Description:
INTEGRATOR_TOLERANCE	double	relative tolerance of the integrator
ABSOLUTE_TOLERANCE	double	absolute tolerance of the integrator
MAX_NUM_INTEGRATOR_STEPS	int	maximum number of integrator steps
CONTROL_PLOTTING	PLOT_NOMINAL PLOT_REAL	specifying whether nominal or actual controls shall be plotted
PARAMETER_PLOTTING	PLOT_NOMINAL PLOT_REAL	specifying whether nominal or actual parameters shall be plotted
OUTPUT_PLOTTING	PLOT_NOMINAL PLOT_REAL	specifying whether nominal or actual outputs shall be plotted
PLOT_RESOLUTION	LOW MEDIUM HIGH	specifying screen resolution when plotting

Chapter 7

Controller for Closed-Loop Simulations

The `Controller` class consists of three major blocks: first, an online state/parameter estimator uses the outputs of the process to obtain estimates for the differential states or other parameters. Second, a reference trajectory can be provided to the control law. These references can either be statically given by the user according to a desired simulation scenario or can be calculated dynamically based on information from the estimator. Finally, both the state/parameter estimates as well as the reference trajectory are used by the `ControlLaw` class to compute optimised control inputs. The control law will usually be a `RealTimeAlgorithm` based on the real-time iteration algorithms (see Section 7.1) but can also be something as simple as a linear state feedback (see Section 7.2).

7.1 Setting-Up an MPC Controller

This section explains how to setup a basic MPC controller. Again, we consider a simple actively damped quarter car model.

7.1.1 Mathematical Formulation

Let x denote the states, u the control input, p a time-constant parameter, and T the time horizon of an MPC optimization problem. We are interested in tracking MPC problems, which are of the general form:

$$
\begin{aligned}
&\underset{x(\cdot),u(\cdot),p}{\text{minimize}} \quad \int_{t_0}^{t_0+T} \|h(t,x(t),u(t),p)-\eta(t)\|_Q^2 \,\mathrm{d}t \\
&\qquad\qquad + \|m(x(t_0+T),p,t_0+T)-\mu\|_P^2 \\
&\text{subject to:} \\
&\qquad\qquad x(t_0) = x_0 \\
&\forall t\in[t_0,t_0+T]: \quad 0 = f(t,x(t),\dot{x}(t),u(t),p) \\
&\forall t\in[t_0,t_0+T]: \quad 0 \geq s(t,x(t),u(t),p) \\
&\qquad\qquad 0 = r(x(t_0+T),p,t_0+T)
\end{aligned}
\tag{7.1}
$$

Here, the function f represents the model equations, s the path constraints and r the terminal constraints. Note that in the online context, the above problem must be solved iteratively for changing x_0 and t_0. Moreover, we assume here that the objective is given in least square form. Most of the tracking problems that arise in practice can be formulated in this form with η and μ denoting the tracking and terminal reference.

7.1.2 Implementation in `ACADO` Syntax

The following piece of code shows how to implement an MPC controller based on this quarter car model. It comprises six main steps:

1. Introducing all variables and constants.
2. Setting up the quarter car ODE model.
3. Setting up a least-squares objective function by defining the five components of the measurement function h and an appropriate weighting matrix.
4. Defining a complete optimal control problem (OCP) comprising the dynamic model, the objective function as well as constraints on the input.
5. Setting up a `RealTimeAlgorithm` defined by the OCP to be solved at each sampling instant together with a sampling time specifying the time lag between two sampling instants. Moreover, several options can be set and plot windows flushed.
6. Setting up a `Controller` by specifying a control law, i.e. the real-time algorithm solving our OCP in this case, and a reference trajectory to be tracked. In this example, the reference trajectory is read from a file where the value of all components are defined over time. (Note that the reference trajectory can be left away when calling the `Controller` constructor which is equivalent to all entries zero over the whole simulation horizon.)

```
#include <acado_toolkit.hpp>
#include <include/acado_gnuplot/gnuplot_window.hpp>

int main( )
{
    USING_NAMESPACE_ACADO


    // INTRODUCE THE VARIABLES:
    // -----------------------
    DifferentialState xB;
    DifferentialState xW;
    DifferentialState vB;
    DifferentialState vW;

    Control F;
    Disturbance R;

    double mB = 350.0;
    double mW = 50.0;
    double kS = 20000.0;
```

```
double kT = 200000.0;


// DEFINE A DIFFERENTIAL EQUATION:
// -------------------------------
DifferentialEquation f;

f << dot(xB) == vB;
f << dot(xW) == vW;
f << dot(vB) == ( -kS*xB + kS*xW + F ) / mB;
f << dot(vW) == (  kS*xB - (kT+kS)*xW + kT*R - F ) / mW;


// DEFINE LEAST SQUARE FUNCTION:
// -----------------------------
Function h;

h << xB;
h << xW;
h << vB;
h << vW;
h << F;

// LSQ coefficient matrix
Matrix Q(5,5);
Q(0,0) = 10.0;
Q(1,1) = 10.0;
Q(2,2) = 1.0;
Q(3,3) = 1.0;
Q(4,4) = 1.0e-8;

// Reference
Vector r(5);
r.setAll( 0.0 );


// DEFINE AN OPTIMAL CONTROL PROBLEM:
// ----------------------------------
const double tStart = 0.0;
const double tEnd   = 1.0;

OCP ocp( tStart, tEnd, 20 );

ocp.minimizeLSQ( Q, h, r );

ocp.subjectTo( f );

ocp.subjectTo( -200.0 <= F <= 200.0 );
ocp.subjectTo( R == 0.0 );


// SETTING UP THE REAL-TIME ALGORITHM:
// -----------------------------------
RealTimeAlgorithm alg( ocp,0.025 );
alg.set( MAX_NUM_ITERATIONS, 1 );
alg.set( PLOT_RESOLUTION, MEDIUM );

GnuplotWindow window;
  window.addSubplot( xB, "Body Position [m]" );
  window.addSubplot( xW, "Wheel Position [m]" );
  window.addSubplot( vB, "Body Velocity [m/s]" );
  window.addSubplot( vW, "Wheel Velocity [m/s]" );
  window.addSubplot( F,  "Damping Force [N]" );
  window.addSubplot( R,  "Road Excitation [m]" );
```

```
    alg << window;


    // SETUP CONTROLLER AND PERFORM A STEP:
    // ------------------------------------
    StaticReferenceTrajectory zeroReference( "ref.txt" );

    Controller controller( alg,zeroReference );

    Vector y( 4 );
    y.setZero( );
    y(0) = 0.01;

    controller.init( 0.0,y );
    controller.step( 0.0,y );


    return 0;
}
```

The file `ref.txt` contains the data of the (trivial) reference trajectory:

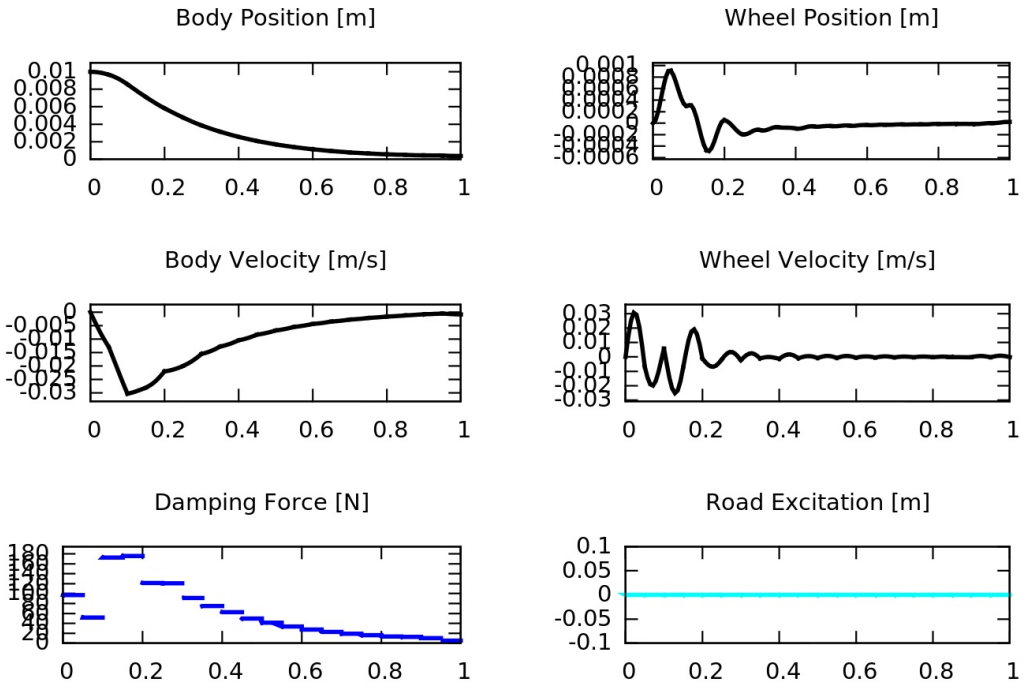
```
    DATA FILE: ref.txt

    ____________________________________________________

    TIME     xB       xW       vB       vW       F

    0.0      0.00     0.00     0.00     0.00     0.00
    1.0      0.00     0.00     0.00     0.00     0.00
    1.5      0.00     0.00     0.00     0.00     0.00
    2.0      0.00     0.00     0.00     0.00     0.00
    3.0      0.00     0.00     0.00     0.00     0.00

    ____________________________________________________
```

7.1.3 Simulation Results

If we run the above piece of code in `ACADO`, the corresponding GNUPLOT output should be as follows:

7.1.4 List of Algorithmic Options

We end this section with providing lists comprising the most common options that can be set when defining a `RealTimeAlgorithm`:



Option name:	Possible values:	Short Description:
MAX_NUM_ITERATIONS	int	maximum number of SQP iterations (default: only one SQP iteration)
USE_REALTIME_ITERATIONS	YES NO	specifying whether real-time iterations shall be used or not
USE_IMMEDIATE_FEEDBACK	YES NO	specifying whether immediate feedback shall be given or not
KKT_TOLERANCE	double	termination tolerance for the optimal control algorithm
HESSIAN_APPROXIMATION	CONSTANT_HESSIAN BLOCK_BFGS_UPDATE FULL_BFGS_UPDATE GAUSS_NEWTON EXACT_HESSIAN	constant hessian BFGS update of the whole hessian structure-exploiting BFGS update (default) Gauss-Newton Hessian approximation exact Hessian computation
DISCRETIZATION_TYPE	SINGLE_SHOOTING MULTIPLE_SHOOTING	single or multiple (default) shooting discretization
INTEGRATOR_TYPE	INT_RK12 INT_RK23 INT_RK45 INT_RK78 INT_BDF	Runge Kutta integrator (order 1/2) Runge Kutta integrator (order 2/3) Runge Kutta integrator (order 4/5) Runge Kutta integrator (order 7/8) BDF integrator
LEVENBERG_MARQUARDT	double	value for Levenberg-Marquardt regularization (default: 0.0)
INTEGRATOR_TOLERANCE	double	relative tolerance of the integrator
ABSOLUTE_TOLERANCE	double	absolute tolerance of the integrator
MAX_NUM_INTEGRATOR_STEPS	int	maximum number of integrator steps
PLOT_RESOLUTION	LOW MEDIUM HIGH	specifying screen resolution when plotting

7.2 Setting-Up More Classical Feedback Controllers

This section explains explains how to setup a basic MPC controller. Again, we consider a simple actively damped quarter car model.

7.2.1 Implementation of a PID Controller

The following piece of code sets-up a PID controller that could be used to control a quarter car:

```
#include <acado_toolkit.hpp>
#include <include/acado_gnuplot/gnuplot_window.hpp>

int main( )
{
    USING_NAMESPACE_ACADO


    // SETTING UP THE FEEDBACK CONTROLLER:
    // ------------------------------------
    PIDcontroller pid( 4,1,0.01 );

    Vector pWeights( 4 );
    pWeights(0) = 1000.0;
    pWeights(1) = -1000.0;
    pWeights(2) = 1000.0;
    pWeights(3) = -1000.0;

    Vector dWeights( 4 );
    dWeights(0) = 0.0;
    dWeights(1) = 0.0;
    dWeights(2) = 20.0;
    dWeights(3) = -20.0;

    pid.setProportionalWeights( pWeights );
    pid.setDerivativeWeights( dWeights );

    pid.setControlLowerLimit( 0,-200.0 );
    pid.setControlUpperLimit( 0, 200.0 );

    StaticReferenceTrajectory zeroReference;

    Controller controller( pid,zeroReference );


    // INITIALIZE CONTROLLER AND PERFORM A STEP:
    // -----------------------------------------
    Vector y( 4 );
    y.setZero( );
    y(0) = 0.01;

    controller.init( 0.0,y );
    controller.step( 0.0,y );

    Vector u;
    controller.getU( u );
    u.print( "Feedback control" );

    return 0;
}
```

First, a `PIDcontroller` comprising four inputs and one output with a sampling time of 10 ms is defined. In case the number of outputs equls the number of inputs, all outputs are calculated component-wise; otherwise, as in our example, the PID terms of all inputs are summed to yield the single output. Second, proportional and derivative weights are set. Third, lower and upper limits are specified for the control output, i.e. if the control signal exceed these limits, it is clipped. Finally, the controller is initialized, one step is performed and the control signal is printed.

7.2.2 Implementation of a LQR Controller

The following piece of code sets-up a LQR controller that could be used to control a quarter car:

```
#include <acado_toolkit.hpp>
#include <include/acado_gnuplot/gnuplot_window.hpp>

int main( )
{
    USING_NAMESPACE_ACADO


    // SETTING UP THE FEEDBACK CONTROLLER:
    // -----------------------------------
    Matrix K( 1,4 );
    K(0,0) = -3.349222044080232e+04;
    K(0,1) = -3.806600292165519e+03;
    K(0,2) =  9.999999999999985e+02;
    K(0,3) = -1.040810121403324e+03;

    LinearStateFeedback lqr( K,0.025 );

    lqr.setControlLowerLimit( 0,-200.0 );
    lqr.setControlUpperLimit( 0, 200.0 );

    StaticReferenceTrajectory zeroReference;

    Controller controller( pid,zeroReference );


    // INITIALIZE CONTROLLER AND PERFORM A STEP:
    // -----------------------------------------
    Vector y( 4 );
    y.setZero( );
    y(0) = 0.01;

    controller.init( 0.0,y );
    controller.step( 0.0,y );

    Vector u;
    controller.getU( u );
    u.print( "Feedback control" );

    return 0;
}
```

First, the gain matrix of the LQR controller is defined (that has been calculated beforehand). Afterwards, the `LinearStateFeedback` controller is defined by specifying the LQR gain matrix as well as a sampling time of 25 ms. Third, lower and upper limits are specified

for the control output, i.e. if the control signal exceed these limits, it is clipped. Finally, the controller is initialized, one step is performed and the control signal is printed.

Chapter 8

Simulation Environment

Communication between `Process` and `Controller` is orchestrated by an instance of the `SimulationEnviroment` class. It also features the simulation of computational delays, i.e.it can delay the control input to the `Process` by the amount of time the `Controller` took to determine the control inputs. This feature seems to be crucial for realistic closed-loop simulations of fast processes where the sampling time is not negliglible compared to the settling time of the controlled process.

8.1 Performing a Basic Closed-Loop MPC Simulation

This section explainshow to setup a basic closed-loop simulation using a model predictive controller. Again, we consider the simple quarter car model as a guiding example (see section 6.1).

8.1.1 Implementation in `ACADO` Syntax

The following piece of code shows how to implement a closed-loop simulation based on our quarter car model. It comprises three main steps:

1. Setting up the ODE model of the quarter car and defining a `Process` as explained in detail in chapter 6.
2. Setting up an MPC controller as explained in detail in chapter 7.
3. Setting up the `SimulationEnvironment` by defining the start and end time of the closed-loop simulation as well as the process and controller used for simulation. Afterwards, it is initialized with the initial value of the differential states to be used in the process and the whole simulation is ran. Finally, results are obtained and plotted.

```
#include <acado_optimal_control.hpp>
#include <include/acado_gnuplot/gnuplot_window.hpp>

int main( )
{
    USING_NAMESPACE_ACADO
```

```
// INTRODUCE THE VARIABLES:
// -------------------------
DifferentialState xB;
DifferentialState xW;
DifferentialState vB;
DifferentialState vW;

Disturbance R;
Control F;

double mB = 350.0;
double mW = 50.0;
double kS = 20000.0;
double kT = 200000.0;


// DEFINE A DIFFERENTIAL EQUATION:
// -------------------------------
DifferentialEquation f;

f << dot(xB) == vB;
f << dot(xW) == vW;
f << dot(vB) == ( -kS*xB + kS*xW + F ) / mB;
f << dot(vW) == (  kS*xB - (kT+kS)*xW + kT*R - F ) / mW;


// SETTING UP THE (SIMULATED) PROCESS:
// -----------------------------------
OutputFcn identity;
DynamicSystem dynamicSystem( f,identity );

Process process( dynamicSystem,INT_RK45 );

VariablesGrid disturbance = readFromFile( "road.txt" );
process.setProcessDisturbance( disturbance );


// DEFINE LEAST SQUARE FUNCTION:
// -----------------------------
Function h;

h << xB;
h << xW;
h << vB;
h << vW;
h << F;

// LSQ coefficient matrix
Matrix Q(5,5);
Q(0,0) = 10.0;
Q(1,1) = 10.0;
Q(2,2) = 1.0;
Q(3,3) = 1.0;
Q(4,4) = 1.0e-8;

// Reference
Vector r(5);
r.setAll( 0.0 );


// DEFINE AN OPTIMAL CONTROL PROBLEM:
// ----------------------------------
const double t_start = 0.0;
```

```
    const double t_end   = 1.0;

    OCP ocp( t_start, t_end, 20 );

    ocp.minimizeLSQ( Q, h, r );

    ocp.subjectTo( f );
    ocp.subjectTo( -200.0 <= F <= 200.0 );
    ocp.subjectTo( R == 0.0 );


    // SETTING UP THE MPC CONTROLLER:
    // ------------------------------
    RealTimeAlgorithm alg( ocp,0.025 );
    alg.set( INTEGRATOR_TYPE, INT_RK78 );
    //alg.set( "MAX_NUM_ITERATIONS",2 );

    StaticReferenceTrajectory zeroReference;

    Controller controller( alg,zeroReference );


    // SETTING UP THE SIMULATION ENVIRONMENT,  RUN THE EXAMPLE...
    // ----------------------------------------------------------
    SimulationEnvironment sim( 0.0,2.5,process,controller );

    Vector x0(4);
    x0.setZero();

    sim.init( x0 );
    sim.run( );


    // ... AND PLOT THE RESULTS
    // ------------------------
    VariablesGrid diffStates;
    sim.getProcessDifferentialStates( diffStates );

    VariablesGrid feedbackControl;
    sim.getFeedbackControl( feedbackControl );

    GnuplotWindow window;
      window.addSubplot( diffStates(0),   "Body Position [m]" );
      window.addSubplot( diffStates(1),   "Wheel Position [m]" );
      window.addSubplot( diffStates(2),   "Body Velocity [m/s]" );
      window.addSubplot( diffStates(3),   "Wheel Velocity [m/s]" );
      window.addSubplot( feedbackControl, "Damping Force [N]" );
      window.addSubplot( disturbance,     "Road Excitation [m]" );
    window.plot( );

    return 0;
}
```

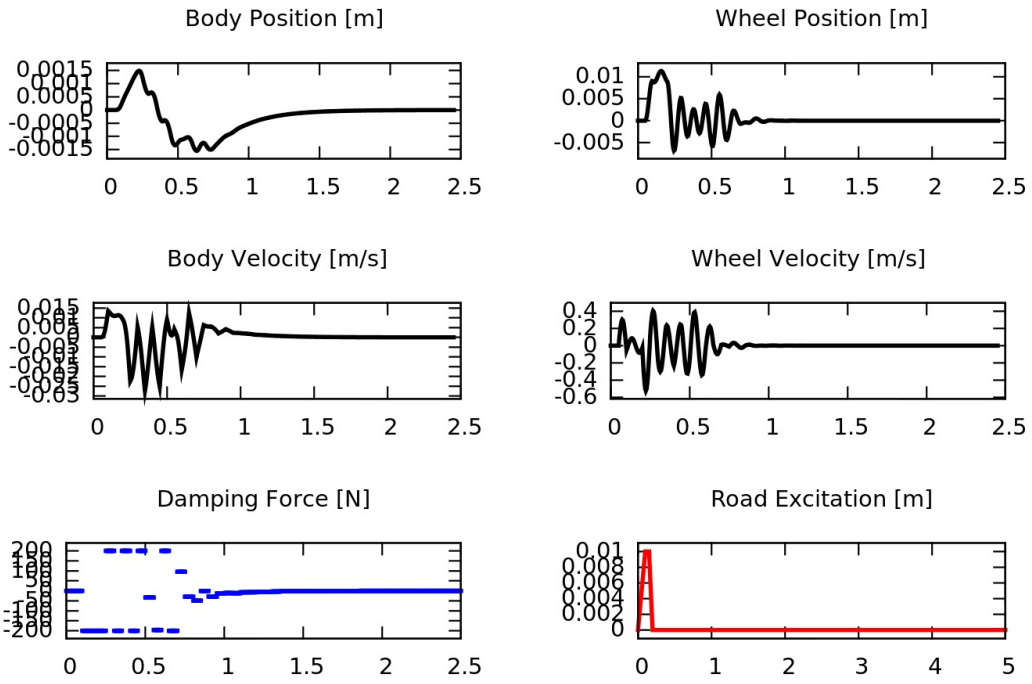
The file `road.txt` contains the following disturbance data:

```
DATA FILE: road.txt

------------------------------

TIME       W

0.0         0.00
0.1         0.01
```

```
0.15        0.01
0.2         0.00
5.0         0.00

_____________________________________
```

8.1.2 Simulation Results

If we run the above piece of code in ACADO, the corresponding GNUPLOT output should be as follows:



Here, we have simulated the road disturbance, which is displayed in the lower right part of the GNUPLOT window. Due to the "bump" in the road we observe an excitation of the body and the wheel, which is however quickly regulated back to zero, by the MPC controller. In addition, the control constraints on the damping force have been satisfied.

Chapter 9

Code Generation

9.1 Introduction

This chapter explains how to use the ACADO Code Generation tool. This first section describes which problems can be tackled using the ACADO Code Generation tool, which numerical algorithms are implemented and how to install the software.

9.1.1 Scope

ACADO Code Generation allows to export optimized, highly efficient C code to solve nonlinear model predictive control (MPC) problems of the following form:

$$\begin{aligned}
\min_{x(\cdot),u(\cdot)} \quad & \int_{t_0}^{t_0+T} \left(\|x(\tau) - x_{\mathrm{ref}}(\tau)\|_Q^2 + \|u(\tau) - u_{\mathrm{ref}}(\tau)\|_R^2 \right) \mathrm{d}\tau \; + \; \|x(t_0+T) - x_{\mathrm{ref}}(t_0+T)\|_P^2 \\
\text{s.t.} \quad & \dot{x}(t) \;=\; f(\, x(t),\, u(t)\,) \\
& x(t_0) \;=\; x_0 \\
& \underline{u}(\tau) \;\leq\; u(\tau) \;\leq\; \overline{u}(\tau) \quad \text{for all } \tau \in [t_0, t_0+T] \\
& \underline{x}(\tau) \;\leq\; x(\tau) \;\leq\; \overline{x}(\tau) \quad \text{for all } \tau \in [t_0, t_0+T]\,.
\end{aligned} \tag{9.1}$$

Here, $x : \mathbb{R} \to \mathbb{R}^n$ denotes the differential state, $u : \mathbb{R} \to \mathbb{R}^m$ the control input. $x_0 \in \mathbb{R}^n$ denotes the current state measurement, $x_{\mathrm{ref}}(\tau) \in \mathbb{R}^n$, $u_{\mathrm{ref}}(\tau) \in \mathbb{R}^m$ suitable, possibly time-varying reference values. $\underline{u}(\tau) \leq \overline{u}(\tau) \in \mathbb{R}^m$ and $\underline{x}(\tau) \leq \overline{x}(\tau) \in \mathbb{R}^n$ are bounds on control inputs and states control bounds, respectively, that also might change along the horizon.
The right-hand side function f defining an ordinary differential equation (ODE) can be nonlinear in both states and controls, while the objective needs to be a least-squares tracking term with $\|\cdot\|_M$ denoting the Euclidean norm weighted by a symmetric, positive semi-definite matrix M.

Note: The current version of ACADO Code Generation does only support continuous-time formulations in the form given above. More general constraint formulations or differential

equations comprising also time-constant parameters, disturbances or algebraic states might become available in a future release.

9.1.2 Implemented Algorithms

ACADO Code Generation exports highly efficient C code solving nonlinear MPC problems by means of the real-time iteration scheme with Gauss-Newton Hessian approximation. Discretization of the time-continuous ODE is done via shooting techniques (only single shooting at the moment) using an equidistant grid. The resulting large but sparse quadratic program (QP) is not solved directly, but first reduced to an equivalent smaller-scale but dense QP. Afterwards, an embedded variant of the online QP solver qpOASES (using an active-set method) or a tailored QP solver exported by the tool CVXGEN (employing an interior point method) can be used to solve this smaller-scale QP. Other optimization algorithms, discretization schemes or QP solvers are not yet supported.

More details on the implemented algorithms and how ACADO Code Generation exports them can be found in Section 9.3 or in [**?**].

9.1.3 Installation

Currently, ACADO Code Generation is only distributed as part of the usual ACADO Toolkit release. Please consult the ACADO Toolkit installation instructions to install the full package; afterwards ACADO Code Generation is at your service.

Note that ACADO Code Generation includes an embedded variant of the open-source QP solver qpOASES only. For employing CVXGEN, you currently need to contact its author and manually generate the desired QP solver via an web interface.

9.2 Getting Started

This section describes in detail how to setup a simple example and to generate optimized code using ACADO Code Generation. We also sketch how to run the generated code.

9.2.1 A Tutorial Example

We consider the following tutorial example: a crane with mass m, line length L, excitation angle ϕ, and horizontal trolley position p. Our control input is the acceleration a of the trolley. With v being the trolley velocity and ω being the angular velocity of the mass point, the system can be described by a simple but non-linear differential equation system:

$$\begin{aligned}
\dot{p}(t) &= v(t) \\
\dot{v}(t) &= a(t) \\
\dot{\phi}(t) &= \omega(t) \\
\dot{\omega}(t) &= -g\sin(\phi(t)) - a(t)\cos(\phi(t)) - b\omega(t)\ ,
\end{aligned}$$

where $b = 0.2\,\mathrm{J\,s}$ is a positive damping constant and we use the parameters $m = 1\,\mathrm{kg}$, $L = 1\,\mathrm{m}$, and $g = 9.81\,\frac{\mathrm{m}}{\mathrm{s}^2}$.

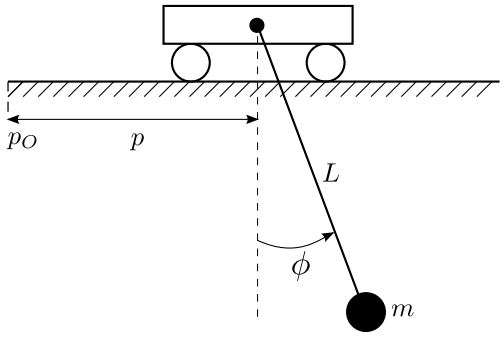


Figure 9.1: Illustration of a overhead crane.

Our aim is to minimize the objective function

$$\min_{x(\cdot),a(\cdot)} \int_{t_0}^{t_0+3\,\mathrm{s}} \left(\|x(\tau) - x_{\mathrm{ref}}(\tau)\|_2^2 + \|a(\tau) - a_{\mathrm{ref}}(\tau)\|_2^2 \right) \mathrm{d}\tau \; + \; \|x(t_0+3\,\mathrm{s}) - x_{\mathrm{ref}}(t_0+3\,\mathrm{s})\|_P^2$$

where we define $x(t) := (p(t),\, v(t),\, \phi(t),\, \omega(t))^T$ and $P := 5 \cdot \mathrm{Id}$. This minimization is to be performed subject to the following control and state constraints:

$$\begin{aligned} -1.0\,\frac{\mathrm{m}}{\mathrm{s}^2} \;&\leq\; a(\tau) \;&\leq\; 1.0\,\frac{\mathrm{m}}{\mathrm{s}^2} \\ -0.5\,\frac{\mathrm{m}}{\mathrm{s}} \;&\leq\; v(\tau) \;&\leq\; 1.5\,\frac{\mathrm{m}}{\mathrm{s}} \end{aligned}$$

for all $\tau \in [t_0, t_0 + 3\,\mathrm{s}]$.

9.2.2 Generating Code

Outline

In order to generate optimized code for this tutorial example using `ACADO Code Generation`, the following steps are required:

1. Setup a source file describing your nonlinear MPC problem in `ACADO` syntax, defining desired options and specifying a target directory for the exported code. Compile this source file.

2. Create a target directory for the code to be exported. Afterwards, copy the source code of the specified QP solver into this target directory.

3. Run your compiled source file to export your code (i.e. the complete nonlinear MPC algorithm) into the target directory.

We will now describe these steps in detail.

Formulating Your Nonlinear MPC Problem

We will now discuss step-by-step a possible nonlinear MPC formulation of the tutorial example in Subsection 9.2.1. The full source code can also be found at <install-dir>/examples/code_generation/getting_started.cpp. Please consult the ACADO User's Manual for more details on the ACADO syntax.

1. First of all, we need to formulate the ODE describing the crane in ACADO syntax. Together with the required header and namespace specification, this can read as follows:

```
#include <acado_code_generation.hpp>

int main( )
{
    USING_NAMESPACE_ACADO

    // DEFINE THE VARIABLES:
    // -----------------------------------------------------------
    DifferentialState  p    ;  // the trolley position
    DifferentialState  v    ;  // the trolley velocity
    DifferentialState  phi  ;  // the excitation angle
    DifferentialState  omega;  // the angular velocity
    Control            a    ;  // the acc. of the trolley

    const double  g = 9.81;    // the gravitational constant
    const double  b = 0.20;    // the friction coefficient
    // -----------------------------------------------------------



    // DEFINE THE MODEL EQUATIONS:
    // -----------------------------------------------------------
    DifferentialEquation f;

    f << dot( p     )  ==  v                                ;
    f << dot( v     )  ==  a                                ;
    f << dot( phi   )  ==  omega                            ;
    f << dot( omega )  == -g*sin(phi) - a*cos(phi) - b*omega;
    // -----------------------------------------------------------
```

   At the top four differential states and one control input are declared. Afterwards, an ODE comprising the four differential equations describing the crane dynamics are defined.

2. We use this ODE formulation to setup an optimal control problem (OCP) to be solved at each MPC sampling instant:

```
    // DEFINE THE WEIGHTING MATRICES:
    // -----------------------------------------------------------
```

```
    Matrix Q  = eye(4);
    Matrix R  = eye(1);

    Matrix P  = eye(4);
    P *= 5.0;
    // ------------------------------------------------------------


    // SET UP THE MPC - OPTIMAL CONTROL PROBLEM:
    // ------------------------------------------------------------
    OCP ocp( 0.0,3.0, 10 );

    ocp.minimizeLSQ       ( Q,R );
    ocp.minimizeLSQEndTerm( P   );

    ocp.subjectTo( f );
    ocp.subjectTo( -1.0 <= a <= 1.0 );
    ocp.subjectTo( -0.5 <= v <= 1.5 );
    // ------------------------------------------------------------
```

The above code snippet starts with defining the weighting matrices of the objective function. Afterwards, an OCP is defined over the time horizon $[0, 3\,\mathrm{s}]$ to be devided into 10 control intervals of equal length. Finally, the weighting matrices for the objective function are set (note that defining an end term is optional) and the OCP constraints are defined. Including the system dynamics into the constraints is mandatory, specifying limits on the control input or differential states is optional.

3. The OCP formulation is passed to the `MPCexport` class, several export options are set and code export to a given target directory is requested:

```
    // DEFINE AN MPC EXPORT MODULE AND GENERATE THE CODE:
    // ------------------------------------------------------------
    MPCexport mpc( ocp );

    mpc.set( HESSIAN_APPROXIMATION, GAUSS_NEWTON    );
    mpc.set( DISCRETIZATION_TYPE,   SINGLE_SHOOTING );
    mpc.set( INTEGRATOR_TYPE,       INT_RK4         );
    mpc.set( NUM_INTEGRATOR_STEPS,  30              );
    mpc.set( QP_SOLVER,             QP_QPOASES      );
    mpc.set( HOTSTART_QP,           NO              );
    mpc.set( GENERATE_TEST_FILE,    YES             );
    mpc.set( GENERATE_MAKE_FILE,    YES             );

    mpc.exportCode( "getting_started_export" );
    // ------------------------------------------------------------

    return 0;
}
```

All options and possible values are explained in detail in Subsection 9.4.1. We will

shortly continue to prepare the specified target directory ./getting_started_export for the code export.

Finally, you need to compile this source file. In order to facilitate this step, you can make use of the Makefile in <install-dir>/examples/code_generation.

Preparing the Target Directory

The target directory has to be given relative to the location of the compiled source file that exports the code. If it does not exist yet, you need to manually create it.

Moreover, ACADO Code Generation expects the target directory to contain the source code of a suitable QP solver. Currently, you can choose between two possible codes:

- The open-source online QP solver qpOASES: An embedded variant of this code comes along with your ACADO Toolkit installation; you can find it at

  <install-dir>/external_packages/qpoases.

  If you want to use it, make a complete copy of this directory into your target directory. Do not rename it as ACADO Code Generation expects the qpOASES code to be located within the sub-directory qpoases of your target directory.

- Tailored QP solver exported by the tool CVXGEN: If you want to use this code, obtain an automatically generated QP solver with appropriate dimensions from

  http://cvxgen.com.

  When generating the code, you need to obey the following naming conventions: H, g, lb, ub for the Hessian matrix, the gradient and the lower and upper bounds, respectively. If your QP also comprises constraints, use A, lbA, ubA for the constraint matrix and the corresponding lower and upper limits, respectively.

  Extract the solver into a sub-directory called cvxgen of your target directory. The required QP dimensions can be printed to screen by calling

  ```
  MPCexport mpc( ocp );
  // ...
  mpc.printDimensionsQP( );
  ```

  Finally, you need to delete the forward declaration of the main function at the end of the file cvxgen/solver.h.

  Note: When using CVXGEN, you need to obtain a tailored QP solver each time the QP dimensions of your problem change. Otherwise the exported code will not work properly.

Actually Exporting Your Nonlinear MPC Code

After compiling your nonlinear MPC problem formulation and preparing the desired target directory, you can actually export optimized C code implementing a complete nonlinear MPC algorithm. For doing so, it is sufficient to simply run the compiled source file containing you problem formulation. Afterwards, you will find all necessary files within the target directory.

9.2.3 Running the Generated Code

Unless deactivated via the export options, the target directory also contains the generated file `test.c` with a main function template to run the generated nonlinear MPC algorithm. Let us browse through this file explaining the main steps:

1. Defining global constants, including headers and declare global variables:

```
#include "acado.h"
#include "auxiliary_functions.c"

// SOME CONVENIENT DEFINTIONS:
// ---------------------------------------------------------------
   #define NX          4      /* number of differential states  */
   #define NU          1      /* number of control inputs       */
   #define N          10      /* number of control intervals    */
   #define NUM_STEPS   5      /* number of real time iterations */
   #define VERBOSE     1      /* show iterations: 1, silent: 0  */
// ---------------------------------------------------------------

// GLOBAL VARIABLES FOR THE ACADO REAL-TIME ALGORITHM:
// --------------------------------------------------
   ACADOvariables acadoVariables;
   ACADOworkspace acadoWorkspace;

// GLOBAL VARIABLES FOR THE QP SOLVER:
// -----------------------------------
   Vars         vars;
   Params       params;
```

   At the beginning, the automatically generated header `acado.h` containing all forward declarations required to run the generated nonlinear MPC algorithm and the (optional) source file `auxiliary_functions.c` are included. Next, a couple of global constants are defined for convenience that contain the number of differential states, control inputs, control intervals, respectively. Moreover, global variables are declared: `acadoVariables` will be used later to call the algorithm, while `acadoWorkspace` is only used internally to store intermediate results. `vars` and `params` contain data to call the QP solver.

2. Initialising the nonlinear MPC algorithm:

```
// A TEMPLATE FOR TESTING THE REAL-TIME IMPLEMENTATION:
// ---------------------------------------------------
int main(){

   // INTRODUCE AUXILIARY VAIRABLES:
   // ------------------------------
      int    i, iter        ;
      double measurement[NX];
```

```
// INITIALIZE THE STATES AND CONTROLS:
// -----------------------------------
   for( i = 0; i < NX*N; i++ )  acadoVariables.x[i] = 0.0;
   for( i = 0; i < NU*N; i++ )  acadoVariables.u[i] = 0.0;

// INITIALIZE THE STATES AND CONTROL REFERENCE:
// --------------------------------------------
   for( i = 0; i < NX*N; i++ )  acadoVariables.xRef[i] =  0.0;
   for( i = 0; i < NU*N; i++ )  acadoVariables.uRef[i] =  0.0;

// SETUP THE FIRST STATE MEASUREMENT:
// ----------------------------------------------
   for( i = 0; i < NX; i++ )  measurement[i] = 0.0;

 // PREPARE FIRST STEP:
 // -------------------
    preparationStep();
```

Before actually calling the nonlinear MPC algorithm, the struct `acadoVariables` needs to be initialised. `acadoVariables.x` and `acadoVariables.u` contain the initialisations of the differential states and control inputs at all interval points[1], respectively. `acadoVariables.xRef` and `acadoVariables.uRef` contain the possibly time-varying reference values for differential states and control inputs at all interval points[1], respectively. Moreover, the array `measurement` is initialised, whose dimension matches that of the differential states. At each call of the nonlinear MPC algorithm, it contains the current state measurement (or estimate), i.e. x_0 within the MPC formulation (9.1). Finally, the first step of the real-time iteration scheme is prepared.

3. Performing real-time iterations:

```
// THE REAL-TIME ITERATION LOOP:
// -----------------------------------------------
   for( iter = 0; iter < NUM_STEPS; iter++ ){

     // OBTAIN A MEASUREMENT:
     // -----------------------------
        /// meausrement = ...

     // PERFORM THE FEEDBACK STEP:
     // ------------------------------
        feedbackStep( measurement );

     // APPLY THE NEW CONTROL IMMEDIATELY TO THE PROCESS:
     // -------------------------------------------------
        /// send first piece of acadoVariables.u to process;
        if( VERBOSE ) printf("============================== ...
        if( VERBOSE ) printf("      Real-Time Iteration %d:  ...
```

[1] stored point-wise in an one-dimensional array

```
    if( VERBOSE ) printf("\n============================ ...

  // OPTIONAL: SHIFT THE INITIALIZATION:
  // -----------------------------------
    /// shiftControls( acadoVariables.uRef );
    /// shiftStates  ( acadoVariables.xRef );

  // PERFORM THE PREPARATION STEP:
  // -----------------------------
    preparationStep();
}
```

This code snippet illustrates how to call the real-time iteration algorithm. At each sampling instant, the user obtains the current state measurement. This measurement is then passed to the feedback step to obtain the optimised control inputs. Afterwards, initialisation of control inputs and states might be shifted and the next iteration is prepared by calling `preparationStep()`.

This sample main function illustrates the main steps to use the auto-generated nonlinear MPC algorithm. More algorithmic details can be found in Section 9.3.
Unless deactivated via the export options, the target directory also contains a basic `Makefile` to facilitate compilation of the exported code and this sample main function. This `Makefile` can also create the library `libacado_exported_rti.a` for performing closed-loop simulations based on the exported code as described in Subsection 9.4.2.

9.3 A Closer Look at the Generated Code

This section provides more details on the algorithms that are implemented by the generated code. It also lists all exported files and illustrates how they iteract to solve nonlinear MPC problems.

9.3.1 Outline of Algorithmic Components

Once a specific MPC problem of the form (9.1) has been set up in `ACADO` syntax, the `MPCexport` class can auto-generate a complete real-time iteration algorithm. It will generate optimized C-code based on hard-coded dimensions which uses static memory only. There are four major algorithmic components:

1. The right-hand side of the ODE as well as its derivatives with respect to the differential states and control inputs are exported as C-code. Derivatives are calculated by means of the associated variational differential equations. They are symbolically simplified employing automatic differentiation tools and exploiting zero-entries in the Jacobian.

   For integrating the ODE system and its derivatives, a tailored Runge-Kutta method is generated [10]. Only the choice of a constant step-sizes is supported, which guarantees a deterministic runtime of the integration.

2. A discretization algorithm is exported which organizes the single- or multiple-shooting evaluation [5] together with the required linear algebra routines to condense the large-scale, sparse QP to a dense but smaller-scale one.

3. A real-time iteration Gauss-Newton method is auto-generated [4, 7, 8]. It performs initial value embedding and employs a tailored algorithm for solving the underlying dense QPs.

4. Finally, an interface to a dedicated QP solver is exported: either a tailored interior-point QP solver generated by the tool CVXGEN [13] (implemented in plain C) or an embedded variant of the active-set online QP solver qpOASES [1, 9] (implemented in basic C++) using fixed dimensions and static memory can be used.

9.3.2 Overview of Generated Files

ACADO Code Generation exports the following files, which correspond to the algorithmic components described in Subsection 9.3.1:

Filename:	*Description:*
acado.h	Contains global variable declarations and forward declarations of all exported algorithmic functions.
integrator.c	Implements ODE and derivative evaluation in the function acado_rhs and the tailored integration routine in the integrate function.
condensing.c	Sets up condensed QP within the function setupQP that makes use of the condense* functions. Linear algebra operations are implemented in the multiply* routines.
gauss_newton_method.c	Implements an Gauss-Newton real-time algorithm based on the functions feedbackStep, preparationStep and initialValueEmbedding. It also provides the functions shiftControls, shiftStates and getKKT.
qpoases/solver.hpp	Declares an interface to call an embedded variant of qpOASES (optional).
qpoases/solver.cpp	Provides an interface to qpOASES that exploits if QP comprises only box constraints (optional).
auxiliary_functions.c	Implements a couple of auxiliary functions for time measurements or for printing results (optional).
test.c	Provides a main function template to run the generated MPC algorithm (see Subsection 9.2.2) (optional).
Makefile	Provides a basic makefile to facilitate compilation of the exported code (optional).

Figure 9.2 on page 97 illustrates how these auto-generated functions implemented within these files interact to perform an nonlinear real-time iteration.

9.4 Advanced Functionality

This section describes all available user-options to adjust the exported code and also explains how to use it for performing closed-loop simulations with the ACADO simulation environment.

9.4.1 Options

The way ACADO Code Generation exports the source code can be adjusted by changing the default values of a number of options. The following list comprises all options that can be set by the user:

Name:	*Possible values:*	*Description:*
HESSIAN_APPROXIMATION	GAUSS_NEWTON	Specifies how to compute or approximate Hessian matrix
DISCRETIZATION_TYPE	SINGLE_SHOOTING (MULTIPLE_SHOOTING)	Shooting technique to discretize time-continuous formulation
INTEGRATOR_TYPE	INT_RK4	Integrator for integrating ODE and generating sensitivities
NUM_INTEGRATOR_STEPS	int (> 0)	Number of itegrator steps along the prediction horizon
QP_SOLVER	QP_QPOASES QP_CVXGEN	Solver for solving small-scale, dense QP: qpOASES or CVXGEN
HOTSTART_QP	YES $\hat{=}$ BT_TRUE NO $\hat{=}$ BT_FALSE	Specifies whether to hotstart QP from previous solution
GENERATE_TEST_FILE	YES $\hat{=}$ BT_TRUE NO $\hat{=}$ BT_FALSE	Specifies whether to generate a test file with sample main function
GENERATE_MAKE_FILE	YES $\hat{=}$ BT_TRUE NO $\hat{=}$ BT_FALSE	Specifies whether to generate a basic Makefile

Note: Modifying the value of any of the above mentioned option will only take effect at the next call to exportCode( ).

9.4.2 Performing Closed-Loop Simulations

The ACADO Toolkit offers a full simulation environment for performing closed-loop simulations including model-plant mismatch, disturbances, noise or dead-times. It is also possible to link the exported code against this environment and use it for closed-loop nonlinear MPC simulations.

Main Steps

The main steps for performing closed-loop simulations based on code produced by the ACADO Code Generation tool are:

1. Setup a source file describing your nonlinear MPC problem in ACADO syntax and compile it.

2. Prepare a target directory for the code to be exported.

3. Run your compiled source file to export your nonlinear MPC algorithm into the target directory. Compile it creating the library `libacado_exported_rti.a`.

4. Setup a source file describing your closed-loop simulation in ACADO syntax. Compile it linking the library of your exported algorithm.

5. Run your closed-loop simulation.

Steps 1 to 3 are explained in detail in Section 9.2.2. We will now illustrate the remaining steps.

Setting-Up and Running a Closed-Loop Simulation

An example for nominally simulating our tutorial example from Section 9.2.1 can be found at `<install-dir>/examples/ code_generation/getting_started_closed_loop.cpp`. In the following, we will go through this example explaining the most important steps:

1. Include headers and declaring global variables:

```
#include <acado_toolkit.hpp>
#include <include/acado_gnuplot/gnuplot_window.hpp>

extern "C"
{
#include "./getting_started_export/acado.h"
#include "./getting_started_export/auxiliary_functions.c"
} // extern "C"

ACADOvariables acadoVariables;
ACADOworkspace acadoWorkspace;

Vars         vars;
Params       params;
#ifdef USE_CVXGEN
Workspace    work;
Settings     settings;
#endif
```

   First, the usual ACADO Toolkit headers as well as the main header `acado.h` of the generated code needs to be included. Also inclusion of the auto-generated file `auxiliary_functions.c` is mandatory in this case. Afterwards, the usual set of global variables for the exported code as described in Section 9.2.2 are declared.

2. Setting-up the simulated process and a controller based on the auto-generated code:

```
int main( )
{
```

```
USING_NAMESPACE_ACADO

// ...

// DEFINE THE MODEL EQUATIONS:
// ------------------------------------------------------------
DifferentialEquation f;
// ...

// SETTING UP THE (SIMULATED) PROCESS:
// -----------------------------------
OutputFcn identity;
DynamicSystem dynamicSystem( f, identity );
Process process( dynamicSystem, INT_RK45 );

// SETTING UP THE MPC CONTROLLER:
// ------------------------------
ExportedRTIscheme rtiScheme(
    4,  // Number of differential states
    1,  // Number of control inputs
    10, // Number of control intervals
    0.3 // Sampling time of the controller
);
#ifdef USE_CVXGEN
set_defaults( );
#endif

Vector xuRef(5);
xuRef.setZero( );

VariablesGrid reference;
reference.addVector( xuRef,  0.0 );
reference.addVector( xuRef, 10.0 );

StaticReferenceTrajectory referenceTrajectory( reference );

Controller controller( rtiScheme,referenceTrajectory );
```

As for usual ACADO closed-loop simulations (see the ACADO Toolkit User's Manual for further details), a Process needs to be setup. In this example we simply use our four-states ODE describing the crane from Section 9.2.1 and integrate it using an adaptive Runge-Kutta integrator. Next, we setup an ExportedRTIscheme based on our auto-generated real-time iteration algorithm. As arguments it takes the number of differential states and control inputs, the number of control intervals as well as the sampling time of the controller. If you use CVXGEN, you also need to call its internal function set_defaults in order to make the controller run correctly. Finally, our ExportedRTIscheme is used together with a reference trajectory to setup an ACADO Controller.

3. Setting-up the simulation environment and run the simulation:

```
    // SETTING UP THE SIMULATION ENVIRONMENT,  RUN THE EXAMPLE...
    // ------------------------------------------------------------
    SimulationEnvironment sim( 0.0,10.0, process,controller );

    Vector x0(4);
    x0(0) = 1.0;
    x0(1) = 0.0;
    x0(2) = 0.0;
    x0(3) = 0.0;

    sim.init( x0 );
    sim.run( );

    // possibly plotting the results

    return 0;
}
```

Having defining `Process` and a `Controller` based on the auto-generated code, a usual `SimulationEnvironment` can be setup. Afterwards, we define an initial value for the simulation and ask run it.

We can now compile this file linking the library `libacado_exported_rti.a` containing the exported algorithm. Afterwards, we simply run it to perform the specified closed-loop simulation.

9.5 Frequently Asked Questions and Troubleshooting

(work in progress)

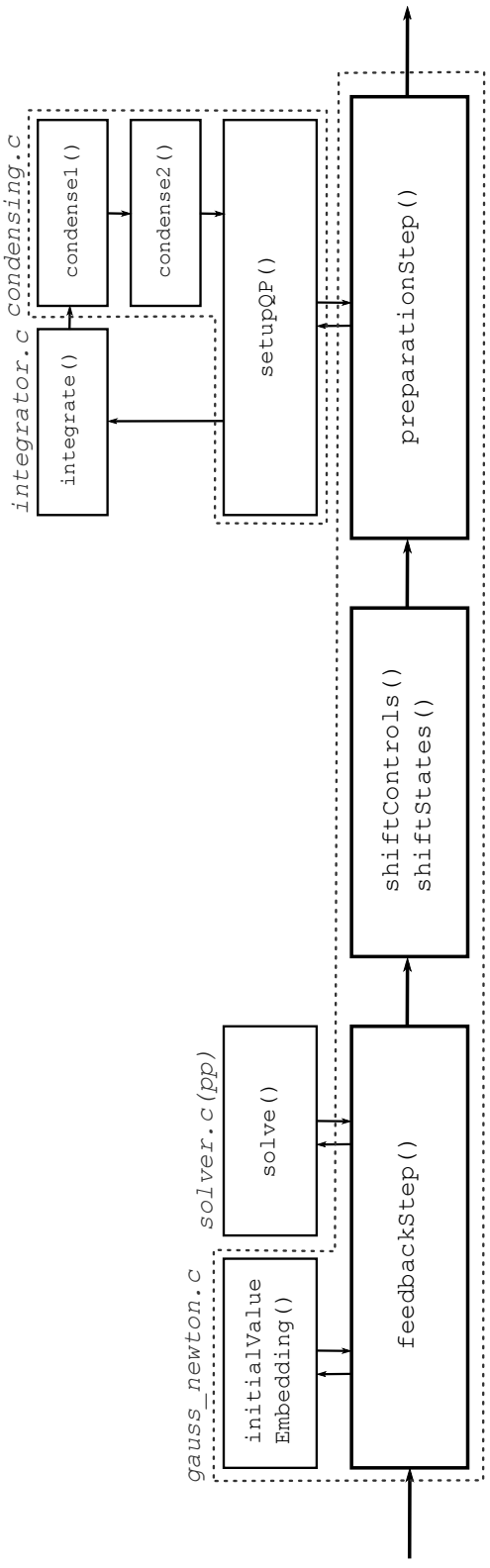


Figure 9.2: Illustration of an nonlinear real-time iteration at the level of the auto-generated functions.

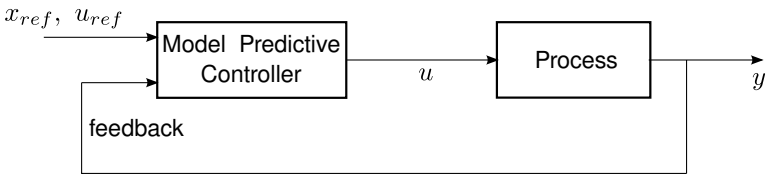


Figure 9.3: Illustration of a simulation environment.

Part IV

Numerical Algorithms

Chapter 10

Integrators

10.1 Introduction

As dynamic optimisation often requires to integrate differential equations numerically, this chapter briefly highlight the most important features of the ACADO Integrators:

- The package ACADO Integrators is a sub-package of ACADO Toolkit providing efficiently implemented Runge-Kutta and BDF integrators for the simulation of ODE's and DAE's.
- For all integrators it is possible to provide ODE or DAE models in form of plain C or C++ code or by using the ACADO Toolkit modeling environment which comes with this package. On top of this, ACADO for Matlab makes it possible to link black-box ODE's, DAE's and Jacobians to the ACADO Toolkit.
- All integrators in ACADO provide first and second order sensitivity generation via internal numerical differentiation. For the case that the model is written within the ACADO Toolkit modeling environment first and second order automatic differentiation is supported in forward and backward mode. Mixed second order directions like e.g. the forward-forward or forward-backward automatic differentiation mode are also possible.

10.2 Runge Kutta Integrators

In ACADO Toolkit several integrators are implemented but at least for ODE's (ordinary differential equations) a Dormand Prince integrator with order 4 is in many routines used by default. The corresponding step size control is of order 5. The following (explicit) Runge-Kutta integrators are available in ACADO Toolkit:

- IntegratorRK12 : A Euler method with second order step-size control.
- IntegratorRK23 : A Runge Kutta method of order 2.
- IntegratorRK45 : The Dormand-Prince 4/5 integrator.
- IntegratorRK78 : The Dormand-Prince 7/8 integrator.

10.3 BDF Integrato

The BDF-method that comes with ACADO Toolkit is designed to integrate stiff systems or implicit DAE's. The mathematical form of DAE's that can be treated by IntegratorBDF is given by

$$\forall t \in [t_{\text{start}}, t_{\text{end}}]: \quad F(t, \dot{x}(t), x(t), z(t)) = 0 \quad \text{with} \quad x(t_{\text{start}}) = x_0 \,. \tag{10.1}$$

where $F : \mathbb{R} \times \mathbb{R}^{n_x} \times \mathbb{R}^{n_x} \times \mathbb{R}^{n_z} \rightarrow \mathbb{R}^{n_x+n_z}$ is the DAE function with index 1 and the initial value $x_0 \in \mathbb{R}^{n_x}$ is given. We say that an initialization $\dot{x}(t_{\text{start}}), x(t_{\text{start}}), z(t_{\text{start}})$ is consistent if it satisfies $F(\dot{x}(t_{\text{start}}), x(t_{\text{start}}), z(t_{\text{start}})) = 0$. If we have a consistent initialization for a simulation we can simply run the integrator to simulate the solution. However if an initialization is provided which is not consistent, the integrator will by default use a relaxation. This means that the integrator solves the system

$$\begin{aligned} &\forall t \in [t_{\text{start}}, t_{\text{end}}]: \\ &F(t, \dot{x}(t), x(t), z(t)) - F(t_{\text{start}}, \dot{x}(t_{\text{start}}), x(t_{\text{start}}), z(t_{\text{start}}))e^{-\Theta \frac{t-t_{\text{start}}}{t_{\text{end}}-t_{\text{start}}}} = 0 \\ &\text{with} \quad x(t_{\text{start}}) = x_0 \,. \end{aligned} \tag{10.2}$$

Here, the constant Θ is equal to 5 by default but it can be specified by the user.
Furthermore, we always assume that the user knows which of the components of F are algebraic - in ACADO Toolkit the last n_z components of F are always assumed to be independent on $\dot{x}$.
Note that the index 1 assumption is equivalent to the assumption that

$$\frac{\partial}{\partial \left(\dot{x}^T, z^T\right)^T} F(t, \dot{x}(t), x(t), z(t)) \tag{10.3}$$

is regular for all $t \in [t_{\text{start}}, t_{\text{end}}]$. For the special case that F is affine in $\dot{x}$ it is not necessary to provide a consistent initial value for $\dot{x}$. In this case only the last n_z components of F (that are not depending on $\dot{x}$) should be 0 at the start, i.e. only a consistent value for $z(t_{\text{start}})$ should be provided. If F is affine in x and z we do not have to meet any consistency requirements.

Chapter 11

Discretization Methods for Dynamic Systems

(work in progress)

11.1 Introduction

(work in progress)

11.2 Shooting Methods

(work in progress)

Chapter 12

NLP Solvers

(work in progress)

12.1 Introduction

(work in progress)

12.2 SQP-Type Methods

(work in progress)

Part V

Low-Level Data Structures

Chapter 13

Matrices and Vectors

ACADO Toolkit comes along with its own stand-alone matrix vector class that does not require any additional packages.

13.1 Getting Started

The classes Vector and Matrix are usually constructed by specifying the dimension in the constructor call. Afterwards, these objects can for example be used to add and multiply them with each other via the standard operators + and * as expected. Note, that these matrix vector operations will be valid whenever this operation is possible, i.e. if the dimensions are correct.
More precisely, the two default constructors of the class Vector are

Vector( ) or Vector(uint dim)

where dim is the dimension of the vector that should be constructed. Correspondingly, a Matrix is constructed by one of the following calls:

Matrix( ) or Matrix(uint nRows, uint nCols)

Here, nRows defines the number of rows and nCols the number of columns of the matrix. There are also several other constructors that allow to directly specify the entries of the constructed matrix or vector which will be discussed in the following sections.
The main reason why the matrix and the vector class are useful is that they provide a convenient syntax for matrix-matrix or matrix-vector multiplications, adding or subtracting matrices or vectors etc.. In addition the components of matrices and vectors can be accessed via the operator (). For example the code

```
Matrix A(3,2); Vector x(2), b(3), c;

A(0,0) = 1.0; A(0,1) = 2.0;
A(1,0) = 3.0; A(1,1) = 4.0;
A(2,0) = 5.0; A(2,1) = 6.0;

x(0)   = 1.0; x(1)   = 2.0;

b(0)   = 1.0;
```

```
b(1)   = 2.0;
b(1)   = 3.0;

c = A*x+b;
```

would actually define a 3×2-matrix `A` as well as vectors `x` and `b` and compute the vector $A * x + b$. The only thing that is important here, is that the dimensions of all operation should fit together - otherwise an error message will be thrown.

13.1.1 Running a Tutorial Example

To understand how the classes `Vector` and `Matrix` are used, we consider the tutorial example

`examples/matrix_vector/getting_started.cpp`

coming with `ACADO Toolkit`:

The corresponding output is as expected:

```
The result for a+b is:
[ 5.0000000000000000e+00  5.0000000000000000e+00  5.0000000000000000e+00 ]

The scalar product of a and b is:
1.6000000000000000e+01

The matrix A*B+A is:
[ 2.0000000000000000e+00  4.0000000000000000e+00 ]
[ 0.0000000000000000e+00  8.0000000000000000e+00 ]

The dyadic product of a and b is:
[ 4.0000000000000000e+00  2.0000000000000000e+00  3.0000000000000000e+00 ]
[ 1.2000000000000000e+01  6.0000000000000000e+00  9.0000000000000000e+00 ]
[ 8.0000000000000000e+00  4.0000000000000000e+00  6.0000000000000000e+00 ]
```

13.1.2 Reading Vectors or Matrices from an `ASCII`-File

It is of course a rather trivial task to read a `ASCII`-File in C++ and store it in a second step into a `Vector` or `Matrix` by using the notation that has been introduced in the previous sections. However, `ACADO Toolkit` provides a convenient notation that allows to read data in several formats directly into matrices or vectors. Moreover, both the `Vector` and the `Matrix` class auto-detect the dimension of vector or matrix data which is given in form of a `ASCII`-file.
The follwing example demonstrates how a vector can be read from a given file with the name `vector.dat`. The tutorial can be found in

`examples/matrix_vector/vector_from_file.cpp`
and `examples/matrix_vector/vector.dat`

coming with `ACADO Toolkit`:

The corresponding file vector.dat that is read here looks as follows:

Note that this file contains the data in different formats. Indeed, the matrix and vector class of ACADO Toolkit provide a quite robust reading routine. Basically, everything that looks like a number will be read. The dimension of the vector is automatically determined - so it will be equal to the number of values that are detected in the file. If nothing else is specified keywords are ignored, i.e. numbers that e.g. appear in comments are also read.
The output of the above example is:

```
[ -5.0000000000000000e-01
   2.0000000000000000e+02
  -3.0000000000000000e+00
   3.0000000000000000e+00
   3.3300000000000000e+02 ]
```

Thus, the dimension of the detected vector is 5 in this example. Of course, for the case that the dimension of the vector which should be read is known, it is recommended to check the dimension of the vector with the function getDim() to provide at least an error message if e.g. numbers in comments are read by accident.
For matrices an analogous constructor exists. If a matrix is read, lines in which no number is found are ignored. The first line in the file which contains numbers defines the number of columns nCols. All following lines, which contain at least on number, are expected to contain exactly nCols numbers. Otherwise, an error message will be thrown. The number of rows of the matrix will coincide will the number of lines in which a valid number of entries has been detected.
A corresponding tutorial example can be found in

examples/matrix_vector/vector_from_file.cpp
and examples/matrix_vector/vector.dat

coming with ACADO Toolkit:

The corresponding file matrix.dat that is read here looks as follows:

The associated output looks as follows

```
[ 1.0000000000000000e+00 2.0000000000000000e+00 -3.0000000000000000e+03 ]
[ 1.0000000000000001e-01 2.0000000000000001e-01  3.0000000000000004e-01 ]
[ 1.2345678901234499e+05 1.2345678901234589e-53 -4.0000000000000018e-04 ]
```

It is important to note that this output is only coinciding with the data in the file up to an numerical accuracy in the order of the machine precision.
ACADO Toolkit provides convenient, robust and generic reading routines that are more than sufficient for most purposes where a small amount of data has to be read. (This is usually the case in the context of dynamic optimization where the algorithms are the expensive part while file reading should not be time critical as a large amount of data can not be

processed through an expensive optimization algorithm anyhow.) However, these reading routines are not guaranteed to be the most efficient solution. These routines are optional and it is of course possible to link self-written reading routines (cf. in work) whenever this is necessary.

13.1.3 Storing Vectors or Matrices into an ASCII-File

Similar to the reading routines it is possible to store a vector or matrix into a file by using a convenient notation. The tutorial example

`examples/matrix_vector/matrix_to_file.cpp`

coming with ACADO Toolkit explains how to do this:

This simple piece of code stores a 3×3 unit matrix into a file with the name `matrix_output.dat`. This file should contain the following three lines:

```
1.0000000000000000e+00 0.0000000000000000e+00 0.0000000000000000e+00
0.0000000000000000e+00 1.0000000000000000e+00 0.0000000000000000e+00
0.0000000000000000e+00 0.0000000000000000e+00 1.0000000000000000e+00
```

Note that the file should be closed again (with `fclose(file)`) in contrast to the reading routine where the file is automatically closed by the constructor.

Chapter 14

Time and Variables Grids

(work in progress)

Chapter 15

Differentiable Functions and Expressions

(work in progress)

Bibliography

[1] qpOASES Homepage. http://www.qpOASES.org, 2007–2011-hp.

[2] ACADO Toolkit Homepage. http://www.acadotoolkit.org, 2009–2011.

[3] D. Ariens, B. Houska, and H.J. Ferreau. ACADO Toolkit User's Manual. http://www.acadotoolkit.org, 2010–2011.

[4] H.G. Bock. Recent advances in parameter identification techniques for ODE. In P. Deuflhard and E. Hairer, editors, *Numerical Treatment of Inverse Problems in Differential and Integral Equations*. Birkhäuser, Boston, 1983.

[5] H.G. Bock and K.J. Plitt. A multiple shooting algorithm for direct solution of optimal control problems. In *Proceedings 9th IFAC World Congress Budapest*, pages 243–247. Pergamon Press, 1984.

[6] E.F. Camacho and C. Bordons. *Model Predictive Control*. Springer, 2nd edition, 2007.

[7] M. Diehl. *Real-Time Optimization for Large Scale Nonlinear Processes*. PhD thesis, Universität Heidelberg, 2001. http://www.ub.uni-heidelberg.de/archiv/1659/.

[8] M. Diehl, H.G. Bock, J.P. Schlöder, R. Findeisen, Z. Nagy, and F. Allgöwer. Real-time optimization and Nonlinear Model Predictive Control of Processes governed by differential-algebraic equations. *J. Proc. Contr.*, 12(4):577–585, 2002.

[9] H. J. Ferreau, H. G. Bock, and M. Diehl. An online active set strategy to overcome the limitations of explicit MPC. *International Journal of Robust and Nonlinear Control*, 18(8):816–830, 2008.

[10] E. Hairer, S.P. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations*, volume II of *Springer Series in Computational Mathematics*. Springer, Berlin, 2nd edition, 1996.

[11] B. Houska and H.J. Ferreau. ACADO Toolkit User's Manual. http://www.acadotoolkit.org, 2009–2011.

[12] B. Houska, H.J. Ferreau, and M. Diehl. ACADO Toolkit – An Open Source Framework for Automatic Control and Dynamic Optimization. *Optimal Control Applications and Methods*, 32(3):298–312, 2011.

[13] J. Mattingley and S. Boyd. *Convex Optimization in Signal Processing and Communications*, chapter Automatic Code Generation for Real-Time Convex Optimization. Cambridge University Press, 2009.

[14] D. van Heesch. Doxygen homepage.