# ACADO for Matlab User's Manual

**Version 1.0beta - v2022 (June 2010)**

David Ariens et al.[1]

Optimization in Engineering Center (OPTEC) and

Department of Electrical Engineering, K. U. Leuven

matlab-support@acadotoolkit.org

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

# Contents

**1 Introduction** **5**
- 1.1 What is ACADO Toolkit . . . 5
- 1.2 What is ACADO for Matlab . . . 5
- 1.3 Installation . . . 6
  - 1.3.1 I'm a Linux or Mac user . . . 7
  - 1.3.2 I'm a Windows user . . . 8
- 1.4 Compatibility . . . 9
- 1.5 About compiling and MEX functions . . . 10
- 1.6 Feedback and Questions . . . 10
- 1.7 Outlook . . . 11
- 1.8 Citing ACADO . . . 11

**2 Problem Classes** **13**
- 2.1 Integration of Differential Equations . . . 13
- 2.2 Optimal Control Problems . . . 15
- 2.3 Parameter and State Estimation . . . 15
- 2.4 Model Based Feedback Control . . . 16

**3 Tutorials: Stand-alone integrator** **19**
- 3.1 Highlights . . . 19
- 3.2 Quick Start Tutorial . . . 19
- 3.3 Four Ways to define a Model . . . 21
- 3.4 Sensitivity Generation . . . 25

**4 Tutorials: Optimization Interface** **29**
- 4.1 Introducing the Generic Optimization Interface . . . 29
- 4.2 ACADO Optimal Control . . . 34
  - 4.2.1 Step 1: The Optimal Control Formulation . . . 34
  - 4.2.2 Step 2: Linking an Optimization Algorithm . . . 36
  - 4.2.3 A first example: a simple free space rocket . . . 38
  - 4.2.4 A second example: movement of a buoy . . . 40
- 4.3 ACADO Parameter Estimation . . . 42
  - 4.3.1 Step 1: Getting your measurements . . . 43
  - 4.3.2 Step 2: The Optimal Control Formulation . . . 44
  - 4.3.3 Step 3: Linking a Parameter Estimation Algorithm . . . 45

4.3.4 An example: estimating parameters of a simple pendulum model . . 45
4.4 ACADO Simulation Environment . . . . . . . . . . . . . . . . . . . . . . . 47
4.4.1 Step 1: The Optimal Control formulation . . . . . . . . . . . . . . 48
4.4.2 Step 2: Setting up the simulated process . . . . . . . . . . . . . . . 48
4.4.3 Step 3: Setting up the MPC controller . . . . . . . . . . . . . . . . 49
4.4.4 Step 4: Linking it to a Simulation Environment . . . . . . . . . . . 50
4.4.5 A first example: Controlling a simple quarter car . . . . . . . . . . 50
4.4.6 A second example: Periodic tracking . . . . . . . . . . . . . . . . . 54

**5 Tutorials: Advanced Features 57**
5.1 `MEX` Input and Output . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
5.2 Using a controller stand-alone . . . . . . . . . . . . . . . . . . . . . . . . 58
5.3 Writing and compiling your own C++ file . . . . . . . . . . . . . . . . . . 59
5.4 Multi Stage OCP . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60

**6 Developer reference 63**
6.1 ACADO for Matlab class structure and code generating methods . . . . . . 63
6.2 Documentation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66
6.3 C++ Code Generation . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67
6.4 Overview of important files and folders . . . . . . . . . . . . . . . . . . . 69

**Index 73**

**Bibliography 73**

# Chapter 1

# Introduction

## 1.1 What is 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 [9]:

- *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 should be as intuitive as possible, 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 [9], [2] and [8].

## 1.2 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, nor any functionality to use the capability of MATLAB to work with matrices, sparse data...

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 1.4 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 1.4).

## 1.3 Installation

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 1.4).

- 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.

### 1.3.1 I'm a Linux or Mac user

**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']));
```

### 1.3.2 I'm a Windows user

**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 1.3.1.

## 1.4 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 1.1 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 1.1: Tested platforms ACADO for Matlab

## 1.5 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.

## 1.6 Feedback and Questions

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

`http://forum.acadotoolkit.org/list.php?9`

To be able to understand your problem include the following:

- The version number of ACADO Toolkit, the version of MATLAB, the platform you are using and your compiler version. To get all this information, simply run

  `<ACADOtoolkit-inst-dir>/interfaces/matlab/getversioninformation.m`.

- The exact error message.
- Your ACADO .m file, black box models, data...

If you have a question regarding ACADO for Matlab, try to solve it yourself first, using one of these tools:

- If you have problems formulating an optimization problem or you don't understand concepts like Mayer terms, bounds, convexity etc. consult for example [7], [10], [6], [5].
- If you don't understand how to use the ACADO for Matlab syntax read this manual carefully, take a look at all examples and comments in

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

  or go to the website.

- Take a look at the FAQ's where common problems are posted:

  `http://forum.acadotoolkit.org/list.php?12`.

- If you still have questions about the ACADO syntax in general, consult [8] or take a look at the C++ examples in

  `<ACADOtoolkit-inst-dir>/examples`,

  but bare in mind that the syntax is not exactly the same in MATLAB.

- Ask your questions on the forum

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

## 1.7 Outlook

The current version is the first release of hopefully many. The author would like to encourage everyone who would like to develop this interface further to contact him. All source code is open-source and thus everyone is free to adapt and improve. Changes can always be submitted to `matlab-support@acadotoolkit.org` for approval.

Currently these topics are still on our TODO list and they might be implemented by us (or you) in the future:

- Simulink support,
- Octave and SciLab support (only when Octave and/or SciLab fully support Matlab's object oriented syntax),
- Multi objective problems,
- Make better usage of matric-vector features of MATLAB (eg being able to directly write `A*x+b` where `A` is a matrix and `b` a vector).

## 1.8 Citing ACADO

If you are using ACADO for Matlab in your research work, please consider citing us [4, 3]:

```
@Manual{acadoMatlabManual,
  title = {ACADO for Matlab User's Manual},
  author = {Ariens, D. and Houska, B. and Ferreau, H. and Logist, F.},
  organization = {Optimization in Engineering Center (OPTEC)},
  edition = {1.0beta},
  month = {May},
  year = {2010},
  note = {\url{http://www.acadotoolkit.org/}}
}


@misc{acado,
  title = {ACADO: Toolkit for Automatic Control and Dynamic Optimization},
  author = {Ariens, D. and Houska, B. and Ferreau, H. and Logist, F.},
  organization = {Optimization in Engineering Center (OPTEC)},
  edition = {1.0beta},
  note = {\url{http://www.acadotoolkit.org/}}
}
```

# Chapter 2

# Problem Classes

This chapter describes the three important problem classes highlighted in ACADO Toolkit, together with a general section about differential equations. This chapter is mainly based on [9] and [8].

The first problem class - Optimal Control Problems (OCP's) - consists of off-line dynamic optimization problems. An open-loop control needs to be found which minimizes a given objective functional. The second class consists of parameter and state estimation problems. Here parameters should be identified by measuring an output of a given (non-linear) dynamic system. The third class are on-line estimation problems combined with model predictive control problems.

## 2.1 Integration of Differential Equations

This section highlights 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.

Further details can be found in [8].

**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.

**BDF Integrator:**
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

$$\forall t \in [t_{\mathsf{start}}, t_{\mathsf{end}}] : \quad F(t, \dot{x}(t), x(t), z(t)) = 0 \quad \text{with} \quad x(t_{\mathsf{start}}) = x_0 \ . \tag{2.1.1}$$

where $F : \mathbb{R} \times \mathbb{R}^{n_x} \times \mathbb{R}^{n_x} \times \mathbb{R}^{n_z} \to \mathbb{R}^{n_x+n_z}$ 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 $\dot{x}(t_{\mathsf{start}}), x(t_{\mathsf{start}}), z(t_{\mathsf{start}})$ is consistent if it satisfies $F(\dot{x}(t_{\mathsf{start}}), x(t_{\mathsf{start}}), z(t_{\mathsf{start}})) = 0$. 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

$$\begin{aligned} &\forall t \in [t_{\mathsf{start}}, t_{\mathsf{end}}] : \\ &F(t, \dot{x}(t), x(t), z(t)) - F(t_{\mathsf{start}}, \dot{x}(t_{\mathsf{start}}), x(t_{\mathsf{start}}), z(t_{\mathsf{start}}))e^{-\Theta \frac{t - t_{\mathsf{start}}}{t_{\mathsf{end}} - t_{\mathsf{start}}}} = 0 \\ &\text{with} \quad x(t_{\mathsf{start}}) = x_0 \ . \end{aligned} \tag{2.1.2}$$

Here, the constant $\Theta$ 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

$$\frac{\partial}{\partial \left(\dot{x}^T, z^T\right)^T} F(t, \dot{x}(t), x(t), z(t)) \tag{2.1.3}$$

is regular for all $t \in [t_{\mathsf{start}}, t_{\mathsf{end}}]$. 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 $z(t_{\mathsf{start}})$ should be provided. If $F$ is affine in $x$ and $z$ we do not have to meet any consistency requirements.

## 2.2 Optimal Control Problems

Regard a dynamic system

$$\dot{x}(t) \;=\; f(t, \dot{x}(t), x(t), z(t), u(t), p, T) \tag{2.2.1}$$

with differential states $x : \mathbb{R} \to \mathbb{R}^{n_x}$, optional time varying control inputs $u : \mathbb{R} \to \mathbb{R}^{n_u}$ and parameters $p : \mathbb{R} \in \mathbb{R}^{n_p}$. Optionally algebraic states are used which are written as $z : \mathbb{R} \to \mathbb{R}^{n_z}$.

In the previous section both the initial states $x(0), z(0)$ were known as well as the control sequence $u(t)$. In optimization, however, we want to determine the inputs in an optimal way subject to a given objective functional. Additionally, we might want to fix the terminal states $x(T), z(T)$, introduce bounds on the inputs, introduce periodic sequences etc.

The standard formulation of an optimal control problem is as follows:

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

with $\Phi$ typically a Bolza functional of the form:

$$\Phi[x(\cdot), z(\cdot), u(\cdot), p, T] = \int_{t_0}^{T} L(\tau, x(\tau), z(\tau), u(\tau), p), T)\, \mathrm{d}\tau \; + \; M(x(T), p, T) \; . \tag{2.2.2}$$

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, collocation is currently under development.

An illustration of the controls needed to bring a system from $x(0)$ to $x(T)$ is given in Figure 2.1. The controls are always returned on a discrete grid.

## 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 $\Phi$ now equal

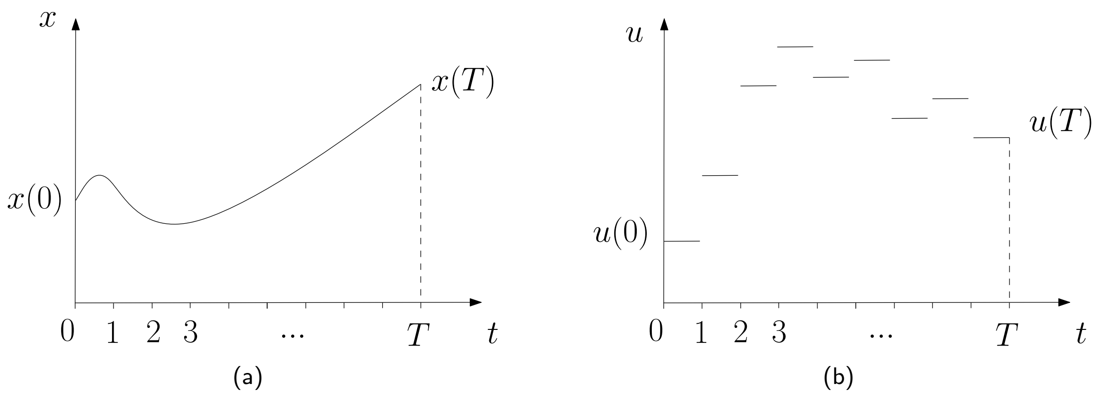

Figure 2.1: An example of an optimal control problem with states (a) and controls (b).

to

$$\Phi[x(\cdot), z(\cdot), u(\cdot), p, T] \;=\; \sum_{i=0}^{N} \| \, h_i(t_i, x(t_i), z(t_i), u(t_i), p) - \eta_i \|_{S_i}^2 \, . \qquad (2.3.1)$$

Estimation problems are thus optimization problems with a least squares objective. Here, $h$ is called a measurement function while $\eta_1, \dots, \eta_N$ are the measurements taken at the time points $t_1, \dots, t_N \in [0, T]$. Note that the least-squares term is in this formulation weighted with positive semi-definite weighting matrices $S_1, \dots, S_N$, 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.

A more theoretical approach of estimation problems can be found in [7] and [10].

## 2.4 Model Based Feedback Control

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

$$\Phi[x(\cdot), z(\cdot), u(\cdot), p, T] = \int_{t_0}^{T} \|y(t, x(t), z(t), u(t), p) - y_{\text{ref}}\|_S^2 + \|y^{\text{end}}(x(T), p) - y_{\text{ref}}^{\text{end}}\|_R^2 \, . \qquad (2.4.1)$$

Therein, $y_{\text{ref}}$ is a tracking reference for the output function $y$ and $y_{\text{ref}}^{\text{end}}$ a reference for a terminal-weight. The matrices $S$ and $R$ are 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.

The methodology of an MPC controller is represented in Figure 2.2. The steps are:

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.

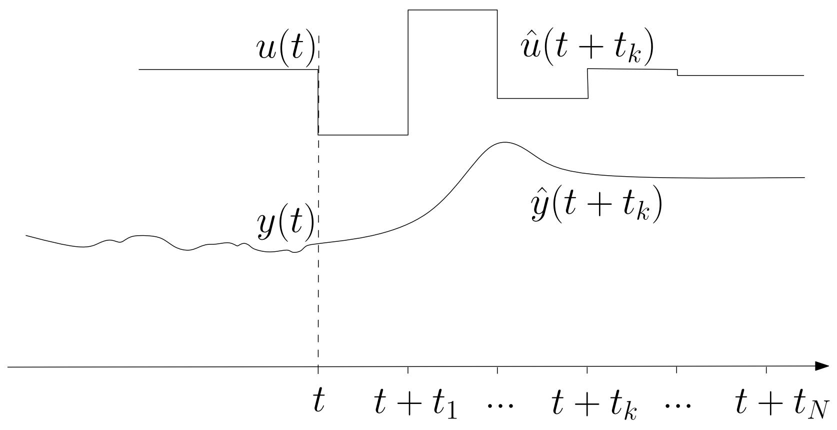

Figure 2.2: An example of an MPC strategy.

# Chapter 3

# Tutorials: Stand-alone integrator

After having presented the outline of ACADO for Matlab and having introduced the problem classes we will now continue with tutorial examples for all classes. It's important to note that two different interfaces are available. This chapter deals with a specific interface for using the ACADO integrators stand-alone. The next chapter provides more insight in the generic interface for optimization problems (optimal control, parameter estimation, model predictive control).

Even if you don't plan to use the integrators stand-alone, we advise you to read this chapter anyway because this is where the syntax used in black-box models will be introduced. This chapter starts with an overview of the highlights and a quick start tutorial. After that, the details of the interface are discussed.

## 3.1 Highlights

The following main features of the ACADO Integrators are available in the MATLAB interface:

- Simulation of ODE and DAE models that are written in MATLAB.
- Easy linking of external C or C++ models.
- Numeric first and second order sensitivity generation for all kinds of ODE and DAE models via internal numerical differentiation.
- First and second order automatic differentiation for all models that are written in the ACADO Toolkit syntax.

## 3.2 Quick Start Tutorial

This tutorial assumes that we use the ACADO Integrators from MATLAB to integrate a simple ODE model function which is given in the ACADO Toolkit syntax. The model file can be found in

`/interfaces/matlab/examples/integrator/getting_started.cpp` and defines a simple ODE of the form $\dot{x} = -x$:

```
void getting_started( DifferentialEquation *f ){

    // Define a Right-Hand-Side:
    // -------------------------
    DifferentialState    x;

    *f << dot(x) == -x;
}
```

This C++ file should be compiled and needs to be added to the file

`/interfaces/matlab/MODEL_INCLUDE.m`.

All the C++ files referenced here will automatically be compiled by typing `make` in MATLAB. Moreover, the function should have this header:

`void FUNCTION_NAME( DifferentialEquation *f ) ...` .

As soon as this C++ function is provided the model is available within MATLAB. To run the integration we have a look into the file

`/interfaces/matlab/examples/integrator/getting_started.m`:

```
%  DEFINE THE INTEGRATOR SETTINGS:
%  -------------------------------
   settings = ACADOintegratorsSettings;

%  DEFINE THE NAME OF THE MODEL:
%  -----------------------------
   settings.Model = 'getting_started';

%  CHOOSE ONE OF THE FOLLOWING RUNGE-KUTTA INTEGRATORS:
%  ----------------------------------------------------
  %settings.Integrator = 'RK12'    ;
  %settings.Integrator = 'RK23'    ;
  %settings.Integrator = 'RK45'    ;
  %settings.Integrator = 'RK78'    ;
   settings.Integrator = 'BDF'     ;

%  DEFINE AN INITIAL POINT:
%  ------------------------
   xStart = [ 1 ]';

%  DEFINE THE START AND THE END OF THE TIME HORIZON:
%  -------------------------------------------------
   tStart = 0;
   tEnd   = 1;

%  DEFINE PLOT OPTIONS:
%  --------------------
   settings.PlotXTrajectory = 1:length(xStart);
```

```
    settings.UseSubplots = 0;

%   RUN THE INTEGRATION ROUTINE:
%   ----------------------------
    tic
        [ xEnd ] = ACADOintegrators( settings,xStart,tStart,tEnd )
    toc

%   END OF THE FILE.
%   ----------------
```

Running this simple m-file will start the integration of the model $\dot{x} = -x$ on the time interval $[0, 1]$ starting at $x(0) = 1$ as defined in the file above. The result at the time `tEnd = 1` will be written into the vector `xEnd`. Additionally, the result for the trajectory will be plotted.

Note that the main communication between the user and the integrator is organized via the `ACADOintegratorsSettings`. Just type

```
help ACADOintegratorsSettings
```

to display the variable `settings` in MATLAB. To get a quick overview it is recommended to use the command

```
help ACADOintegrators
```

to display a short manual.

## 3.3 Four Ways to define a Model

So far, we have only discussed how to link a model that is written in form of a C++ code using the special ACADO Toolkit syntax. This has the advantage that automatic differentiation is possible and that the code is quite efficient. However, there are four ways to define a model in the interface:

**Simple Matlab ODE (black-box) Models:**
The easiest but also least efficient way of defining a ODE model is to write it directly in MATLAB. In this case a reference to one function containing the ODE must be provided. In the tutorial files

`/interfaces/matlab/examples/integrator/beachball.m` and
`/interfaces/matlab/examples/integrator/beachball_ode.m`

a simple ODE is defined:

$$\ddot{y} = -g - \frac{1}{m} k \dot{y}, \tag{3.3.1}$$

which can be written as:

$$\begin{aligned} \dot{x}_1 &= x_2 & (3.3.2)\\ \dot{x}_2 &= -g - \frac{1}{m} k x_2. & (3.3.3) \end{aligned}$$

These equations should be written down in a file with header

```
[ dx ] = FUNCTION_NAME( t,x,u,p,w )
```

which results in:

```
function [ dx ] = beachball_ode( t,x,u,p,w )

    dx(1)  = x(2);
    dx(2)  = -9.81 - .02*x(2);

end
```

Optionally, if you have a Jacobian, you can also link it. In this case the Jacobian is:

$$J = \begin{pmatrix} \frac{\partial \dot{x}_1}{\partial x_1} & \frac{\partial \dot{x}_1}{\partial x_2} \\ \frac{\partial \dot{x}_2}{\partial x_1} & \frac{\partial \dot{x}_2}{\partial x_2} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 0 & -\frac{k}{m} \end{pmatrix}, \tag{3.3.4}$$

which can be added in a file with header

```
[ J ] = FUNCTION_NAME( t,x,u,p,w )
```

resulting in:

```
function [ J ] = beachball_jacobian( t,x,u,p,w )

    J = [0 1;
         0 -.02];

end
```

J is thus a matrix of size $n_x \times (n_x + n_u + n_p + n_w)$.

To integrate this ODE for $t \in [0, 3]$ with initial conditions $x_1(0) = 0$ and $x_2(0) = 10$, consider the file `beachball.m`:

```
 %  DEFINE THE INTEGRATOR SETTINGS:
 %  -------------------------------
    settings = ACADOintegratorsSettings;


 %  DEFINE THE NAME OF THE MODEL:
 %  -----------------------------
    settings.Model = { @beachball_ode };
    settings.Jacobian = { @beachball_jacobian };   % optional


 %  CHOOSE ONE OF THE FOLLOWING RUNGE-KUTTA INTEGRATORS:
 %  ----------------------------------------------------
     settings.Integrator = 'BDF';
    %settings.Integrator = 'RK45';

 %  DEFINE AN INITIAL POINT:
 %  ------------------------
```

```
    xStart = [0, 10]';


%   DEFINE THE START AND THE END OF THE TIME HORIZON:
%   ------------------------------------------------
    tStart = 0  ;
    tEnd   = 3;


%   DEFINE PLOT OPTIONS:
%   -----------------------
    settings.PlotXTrajectory = 1:length(xStart);
    settings.UseSubplots = 0;


%   RUN THE INTEGRATION ROUTINE:
%   ----------------------------
    tic
        [ xEnd, output ] = ACADOintegrators( settings,xStart,tStart,tEnd )
                        % 4 arguments -> represents ODE
    toc
```

Note that the ODE is linked by the line `settings.Model = @beachball_ode` whereas the Jacobian is linked by setting `settings.Jacobian = @beachball_jacobian`. Automatic plotting is enabled by setting `settings.PlotXTrajectory`.

**Simple Matlab DAE (black-box) Models:**

Analogous to defining ODE models also DAE's can be defined in MATLAB. The most important difference in this case is another header (extended by algebraic states). In the tutorial files

`/interfaces/matlab/examples/integrator/simple_dae.m` and
`/interfaces/matlab/examples/integrator/simple_dae_matlab.m`

a simple DAE is defined:

$$\begin{array}{rclcl} \dot{x} & = & f(x,p,z) & := & -p_1 x^2 z \\ 0 & = & g(x,p,z) & := & p_2^2 - z^2 . \end{array} \tag{3.3.5}$$

All equations $f(x,p,z)$ and $g(x,p,z)$ should be defined in a file with fixed header of form

`[ f ] = FUNCTION_NAME( t,x,xa,u,p,w )`.

First, all differential equations in $f(x,p,z)$ should be added to `f`, secondly all algebraic equations $g(x,p,z)$.

```
function [ f ] = simple_dae_matlab( t,x,xa,u,p,w )

    f(1)  = -p(1)*x(1)*x(1)*xa(1);     % Differential equation
    % Add more differential equations if needed...
    f(2)  = p(2)*p(2) - xa(1)*xa(1)    % Algebraic equation
    % Add more algebraic equations if needed...
end
```

Now, we like to integrate this simple DAE. For this aim we consider the file `simple_dae.m`:

```
%  DEFINE THE INTEGRATOR SETTINGS:
%  -------------------------------
   settings = ACADOintegratorsSettings;

%  DEFINE THE NAME OF THE MODEL:
%  -----------------------------
   settings.Model = { @simple_dae_matlab };

%  USE THE BDF INTEGRATOR:
%  -----------------------------
   settings.Integrator = 'BDF';

%  ADJUST THE INTEGRATOR:
%  -------------------------------------------
   settings.Tolerance = 1e-5;     % local error tolerance.

%  DEFINE AN INITIAL POINT:
%  ------------------------
   xStart  = [ 1.0 ]';
   xaStart = [ 1.0 ]';

%  DEFINE THE START AND THE END OF THE TIME HORIZON:
%  -------------------------------------------------
   tStart = 0;
   tEnd   = 0.2;

%  DEFINE THE PARAMETERS:
%  ----------------------
   settings.p = [1.0 1.0]';

%  DEFINE PLOT OPTIONS:
%  -----------------------
   settings.PlotXTrajectory  = 1:length(xStart);
   settings.PlotXaTrajectory = 1:length(xaStart);
   settings.UseSubplots = 0;

%  RUN THE INTEGRATION ROUTINE:
%  ----------------------------
   tic
       [ xEnd, xaEnd, outputs ] = ACADOintegrators(
           settings,xStart,xaStart,tStart,tEnd )
               % 5 arguments -> represents DAE
   toc
```

Note that this file defines the model function by the code line

`settings.Model = { @simple_dae_matlab } .`

In this example we have to use the BDF integrator as the explicit Runge Kutta routines can not deal with DAE's. Please run this file and also display the variable `outputs` which contains all results of the integration.

**C++ Model with ACADO Syntax:**
In the quick start tutorial we have already discussed how to integrate a simple ODE that is written with the ACADO Toolkit syntax. This way of linking C++ code has the advantage that the model is evaluated in C++ which is much more efficient than MATLAB. In addition automatic differentiation can be used. To run the simple DAE example from above, just adapt the file `simple_dae.m` by defining the model as

```
    settings.Model = 'simple_dae';
```

instead of passing the MATLAB function handles. Now, the routine will call the $C++$ file `simple_dae.cpp`:

```
void simple_dae( DifferentialEquation *f ){

    DifferentialState            x;
    AlgebraicState               z;
    Parameter                    p;
    Parameter                    q;

    *f << dot(x) == -p*x*x*z  ;
    *f <<      0  ==  q*q - z*z;
}
```

which defines exactly the same DAE as the MATLAB function above. Thus, running the file leads to the same results and usually to faster computation times.
Don't forget to add `simple_dae.cpp` to the list of to-be-compiled files in:

`/interfaces/matlab/MODEL_INCLUDE.m`,

resulting in this line of code:

```
addModelFile('../examples/integrator/simple_dae.cpp', 'simple_dae');
```

**Plain C or C++ Model:**
Finally, we might sometimes be in the situation that we have a large C model which we do not want to translate to the ACADO Toolkit syntax. In this case the C model function can easily be linked using `CFunction`. This is however not yet fully automatically supported by the interface.

## 3.4 Sensitivity Generation

As mentioned before, automatic differentiation is only supported if the model function is written in the ACADO Toolkit syntax as explained above. For the case that backward derivatives are requested while the function is not in the right format an error message will appear. We explain how the sensitivity generation can be used by looking again into the simple DAE example which has already been explained above. The corresponding tutorial code for the sensitivity generation can be found in interfaces/matlab/examples/integrator/simple_dae_sens.m

and reads as follows:

```
%  DEFINE THE INTEGRATOR SETTINGS:
%  -------------------------------
   settings = ACADOintegratorsSettings;

%  DEFINE THE NAME OF THE MODEL:
%  -----------------------------
    settings.Model = 'simple_dae';
   %settings.Model = { @simple_dae_matlab_f,@simple_dae_matlab_g,1 };

%  USE THE BDF INTEGRATOR:
%  -----------------------------
   settings.Integrator = 'BDF';

%  DEFINE AN INITIAL POINT:
%  ------------------------
   xStart  = [ 1.0 ]';  xaStart = [ 1.0 ]';

%  DEFINE THE START AND THE END OF THE TIME HORIZON:
%  -------------------------------------------------
   tStart = 0;  tEnd   = 0.2;

%  DEFINE THE PARAMETERS:
%  ----------------------
   settings.p = [1.0 1.0]';

%  DEFINE FORWARD SEED (FORWARD SENSITIVITY GENERATION):
%  -----------------------------------------------------
   settings.SensitivityMode = 'AD_FORWARD';             % sensitivity mode
   settings.lambdaX         =  eye( length(xStart) );  % forward seed

%  DEFINE BACKWARD SEED (ADJOINT SENSITIVITY GENERATION):
%  ------------------------------------------------------
%  settings.SensitivityMode = 'AD_BACKWARD';            % sensitivity mode
%  settings.mu              =  eye( length(xStart) );  % backward seed

%  DEFINE FORWARD SEED (2nd ORDER FORWARD SENSITIVITY GENERATION):
%  ---------------------------------------------------------------
%  settings.SensitivityMode = 'AD_FORWARD2'  ;  % sensitivity mode
%  settings.lambdaX         =  [1.0]         ;  % forward seed
%  settings.lambda2X =  eye( length(xStart) );  % second order forw. seed

%  DEFINE FORWARD SEED AND SECOND ORDER BACKWARD SEED (AUTO FORWARD SENS):
%  -----------------------------------------------------------------------
%  settings.SensitivityMode = 'AD_FORWARD_BACKWARD';% sensitivity mode
%  settings.lambdaX         =  [1.0];               % 1st order forw. seed
%  settings.mu2             =  eye(length(xStart)); % 2nd order backw. seed

%  RUN THE INTEGRATION ROUTINE:
%  ----------------------------
   [ xEnd, xaEnd, outputsB ] = ACADOintegrators(
       settings,xStart,xaStart,tStart,tEnd )
```

In this example all typical ways on how to define seeds are explained. In the default version of this tutorial a forward seed in $x$ direction is set. Just display the integrator settings to learn how other seeds are set. By uncommenting one of the examples for the sensitivity generation it can be tested how the other modes work. Note that it is possible to define e.g. mixed forward and backward seeds for the second order differentiation. The results for all sensitivities can be found in the struct `outputsB` which should be self-explaining.

# Chapter 4

# Tutorials: Optimization Interface

## 4.1 Introducing the Generic Optimization Interface

For optimization problems an interface is introduced allowing you to write down your problems in a style analogous to the ACADO Toolkit C++ style. To achieve this, the same classes and methods available in ACADO Toolkit are also made available in ACADO for Matlab. Moreover, the syntax is very similar. Before starting with the problem-class-specific tutorials, this section will briefly introduce the used syntax.

When writing down your optimization problem, bare in mind these steps:

1. **BEGIN_ACADO:**
   Every problem should start with `BEGIN_ACADO`. This command will set all correct paths and global variables.

2. **Options:**
   Next, you might want to set some MATLAB specific options. These options include your problem name, whether or not you want ACADO to write your results to files (instead of just to the workspace)... Do so by using `acadoSet`, eg:

   ```
       acadoSet('problemname', 'active_damping');
   ```

   All options are explained in the help file: `help acadoSet`. Setting a good problem name is important since it will become the prefix for all files generated by ACADO. These files include the C++ file (`problemname.cpp`) compiled `MEX` file (`problemname_RUN.mexXX`), the MATLAB file to run the `MEX` file (`problemname_RUN.m`), files containing matrices... If you do not set a problem name, your problem will be called `myAcadoProblem`.

3. **Expressions:**
   You can now start writing down your problem. You will start by defining your differential states, algebraic states, controls, parameters and disturbances. You can define a differential state by writing

   ```
   DifferentialState state_name;
   ```

or if you have more than one state:

```
DifferentialState state_name1 state_name2 state_name3.
```

or

```
DifferentialState state_name1
DifferentialState state_name2
DifferentialState state_name3.
```

The syntax is analoguous for other expressions:

```
    DifferentialState x1 x2;
    AlgebraicState z;
    Control F;
    Disturbance R;
    Parameter p q;
```

From now on, you can just use these expressions as variables in your syntax. You can for example write `x1*F+R` and use this as an equation in a differential equation (up next).

4. **Differential equation (continuous time):**
   After having defined all expressions, you will have to link or write down a differential equation. There are several possibilities: you can write down an ODE or DAE yourself using our symbolic syntax. However, if you have a black-box model in C or MATLAB syntax it's also possible to link them. No matter what you will be linking, always start with setting up an ACADO `DifferentialEquation` object:

   ```
   f = acado.DifferentialEquation().
   ```

   Note the prefix "acado.". Since all classes are embedded in the package acado, you will always need to write "`acado.`" before you are able to call one of them.

   More information is always available in the help files which you can open by simply calling the class help file :

   ```
   help acado.DifferentialEquation.
   ```

   After having set up a differential equation object, you will now have to tell ACADO what your differential (algebraic) equation actually is:

   (a) **Using the ACADO symbolic syntax for time-continuous models:**
   Use the add function to add a new equation to the list of differential (algebraic) equations. Regard this set of differential equations, describing a simple freespace

rocket model with 3 states: the distance $s$, the velocity $v$, and the mass $m$ of the rocket:

$$\begin{aligned}\dot{s}(t) &= v(t)\\ \dot{v}(t) &= \frac{u(t) - 0.2\,v(t)^2}{m(t)}\\ \dot{m}(t) &= -0.01\,u(t)^2.\end{aligned}$$

Each differential equation needs to be added individually to the differential equation object:

```
    f.add(dot(s) == v);
    f.add(dot(v) == (u-0.02*v*v)/(m));
    f.add(dot(m) == -0.01*u*u);
```

Note the `dot()` operator to indicate a derivative and the double equals sign "==" instead of just one "=". As always, don't forget the help files:

`help acado.DifferentialEquation.add`.

(b) **Linking a Matlab black-box:**
Using exactly the same file and heading structure as in Section 3.3 you can now link a MATLAB ODE:

`f.linkMatlabODE('rocketode')`.

Or, if you also have a Jacobian:

`f.linkMatlabODE('rocketode', 'rocketjacobian')`.

If you want to link a MATLAB DAE:

`f.linkMatlabDAE('exampledae')`.

See also: `help acado.DifferentialEquation.linkMatlabODE` and `help acado.DifferentialEquation.linkMatlabDAE`.

(c) **Linking a C++ file:**
If you have your model in the form of a C++ file, there is no need converting it to MATLAB code. Simply link it by calling:

```
f.linkCFunction('cfunction.cpp',
 'myAcadoDifferentialEquation')
```

where the first argument is your file name which needs to be included and the second one is the name of the function that needs to be called.
The file `cfunction.cpp` will need to have a specific header to allow ACADO to make the necessary calls:

```
void myAcadoDifferentialEquation( double *x, double *f, void
    *user_data ){
```

```
    double t = x[0];
    double v = x[1];
    double s = x[2];
    double m = x[3];
    double u = x[4];

    f[0] =  (u-0.02*v*v)/(m);          //dot(v)
    f[1] =  v;                         //dot(s)
    f[2] =  -0.01*u*u;                 //dot(m)

}
```

This example implements the same rocket model as before. Now `double *x` is an array containing all available expressions. The first element is always the time, the second element is the first state and so on... This sequence is the same sequence as you used to define your expressions.

**Differential equation (discrete time):**
Analogous to a `DifferentialEquation` object which contains a continuous time model, you can also set up a discrete time model with ACADO using:

`f = acado.DiscretizedDifferentialEquation(h)`,

where `h` is the step length. You will now have to define or link your model, which is very similar to a continuous time model:

(a) **Using the ACADO symbolic syntax for discrete time models:**
Use the `next` function to add a new equation to the list of differential (algebraic) equations. Regard the same simple freespace rocket model as before with 3 states: the distance $s$, the velocity $v$, and the mass $m$ of the rocket:

$$\begin{aligned} s_{k+1} &= s_k + h \cdot v_k \\ v_{k+1} &= v_k + h \cdot \frac{u_k - 0.2\, v_k^2}{m_k} \\ m_{k+1} &= m_k - h \cdot 0.01 \cdot u_k^2. \end{aligned}$$

Each differential equation needs to be added individually to the discrete differential equation object:

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

(b) **Linking a Matlab black-box:**
*This functionality is still under development*

(c) **Linking a C++ file:**
*This functionality is still under development*

5. **Optimization problem:**
   You will now formulate your optimization problem (see the sections below).

6. **END_ACADO:**
   At the end, you will finish your problem by writing `END_ACADO`. By calling this function your optimization problem will be compiled into a C++ file. This file is stored in the directory you are currently working in, together with other helper files. All files will start with your problem name and your executable will be called `problemname_RUN`. This means you can now execute your problem by simply typing:

   ```
   out = active_damping_RUN(); .
   ```

   This will start your optimization problem and ACADO will try to find a solution. Once a solution is found, the result will be stored in the struct "out". You can rerun this problem as many times as you want, but if you change something in your problem file, you will need to compile again by simply running the file containing `BEGIN_ACADO` and `END_ACADO`.

   The struct `out` contains different fields with the results. Depending on the expressions you have defined these fields will be:

   - STATES,
   - CONTROLS,
   - PARAMETERS,
   - DISTURBANCES,
   - ALGEBRAICSTATES,
   - INTERMEDIATESTATES (not everywhere available).

   The rows contain different time points. The first element of each row is the time, the next elements are the values corresponding to the ones you have defined as expressions. If you have for example 3 states, the field `out.STATES` will contain $1+3$ columns.

Note that all classes and methods belong to the MATLAB package "acado". This means that when you would like to read the documentation, you should always call:

```
help acado.FUNCTION
```

or for the help file of a method:

```
help acado.FUNCTION.METHOD.
```

To summarize this section we end with a code example:

```
clear;

BEGIN_ACADO;                            % Always start with "BEGIN_ACADO".

    acadoSet('problemname', 'active_damping');
                                        % Call the problem active_damping


    DifferentialState xB xW vB vW;      % Differential States:
                                        % xBody, xWheel, vBody, vWheel

    Control F;                          % Control:
                                        % dampingForce

    Disturbance R;                      % Disturbance:
                                        % roadExcitation


    %% Differential Equation
    f = acado.DifferentialEquation();   % Set the differential equation object
    f.linkMatlabODE('ode');             % Link to a Matlab ODE


    %%%% HERE COMES THE OPTIMIZATION PROBLEM %%%%


END_ACADO;                              % End with "END_ACADO" to compile.


out = active_damping_RUN();             % Run the test.
```

## 4.2 ACADO Optimal Control

This section will describe how to formulate an Optimal Control Problem or OCP. All examples used (and more) can be found in the examples directory:

`/interfaces/matlab/examples/ocp/`.

We advise you to take a close look at these examples since they cover the most commonly used problems. Moreover, they are very well documented.

Figure 4.1 summarizes the main components of an Optimal Control Problem. These components will now be explained in more detail.

### 4.2.1 Step 1: The Optimal Control Formulation

Setting up an optimal control problem starts by creating an `acado.OCP` object. Upon creation you need to provide a few arguments: the starting time of your simulation, the end time (can be a fixed value or a parameter to be optimized) and optionally the number of control intervals:

`ocp = acado.OCP(start, end [, intervals])`.

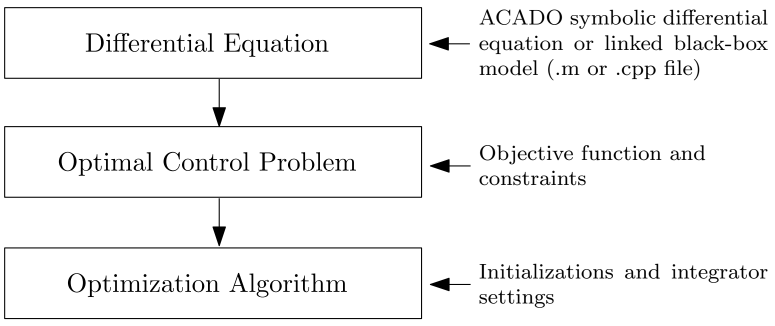
Differential Equation
Optimal Control Problem
Optimization Algorithm
ACADO symbolic differential equation or linked black-box model (.m or .cpp file)
Objective function and constraints
Initializations and integrator settings

Figure 4.1: Optimal Control Problem

After having defined the OCP object you can add several types of objectives (Mayer terms, Lagrange terms, least squares terms) and your constraints (as described in Section 2.2). All different options are explained in

```
help acado.OCP
```

but for now, we only introduce the most important ones.

**Objective function:**

1. **Minimize a Mayer or Lagrange term:**
   Simply call

   `ocp.minimizeMayerTerm(expression)` or
   `ocp.minimizeLagrangeTerm(expression)`

   where an expression can be any kind of expression containing differential states, controls... eg `x + p*p`.

   *Please note that in the current version of ACADO for Matlab you cannot use a Lagrange term when linking to a black box model. This is however no problem, since you could just as well introduce another differential state integrating a certain state you would like to minimize. Combine this extra differential state with a Mayer term and you'll get the same result.*

2. **Minimize a Least Squares Term (LSQ):**
   You can minimize a least squares term of the form:

   $$\frac{1}{2}\sum_{i} \|(h(t_i, x(t_i), u(t_i), p(t_i), ...) - r)\|_S^2 \tag{4.2.1}$$

   resulting in this call:

   `ocp.minimizeLSQ(h[, r])` or `ocp.minimizeLSQ(S, h, r)`.

If you want to penalise $N$ expressions, then $S \in \mathbb{R}^{N \times N}$ is a weighting matrix, $h$ is a $1 \times N$ cell array containing the expressions and $r$ is the reference. If your reference is just one static value for each expression then $r \in \mathbb{R}^{1 \times N}$ is just a vector. However, if you would like to use a time dependant trajectory with $q$ different time points, $r \in \mathbb{R}^{q \times (N+1)}$ becomes a matrix with $q$ rows and $N+1$ columns. The first column contains the time points, the other columns the reference.

It is also possible to add different types of objectives to the same OCP.

**Constraints:**
A constraint can be added by calling

```
ocp.subjectTo( ... ).
```

The first thing that you always want to do is to link your differential equation to this optimal control problem since you always want to minimize/maximize an objective with respect to a dynamic model. Do so by simply writing:

```
ocp.subjectTo( f ).
```

Next-up it's time to add other constraints, e.g.:

1. An initial constraint: `ocp.subjectTo( 'AT_START', x == 1.0 ).`
2. A terminal constraint: `ocp.subjectTo( 'AT_END', x == 1.0 ).`
3. A path constraint: `ocp.subjectTo( 0.1 <= p <= 2.0 )`
   or just an equality: `ocp.subjectTo( 0.1 == p ).`
4. A time depending constraint: `ocp.subjectTo( u1 == myMatrix )` where `myMatrix` contains in the first column the time points and in the second column the time depending values.

### 4.2.2 Step 2: Linking an Optimization Algorithm

After having described the optimal control problem, it needs to be linked to an optimization algorithm. This algorithm will solve the problem. You can set initializations, choose tolerances, select the integrator you would like to use etc using the `set`-method. However these settings are all optional and can be omitted if not necessary. ACADO will automatically select a good integrator, use the default tolerances and other settings and will take a guess for initializations.

To link your OCP to an optimization algorithm, simply write:

```
algo = acado.OptimizationAlgorithm(ocp).
```

And that's it! You can now call `END_ACADO` to compile your problem.

If you take a look into the help file, you will see some other options appearing which you can use:

`help acado.OptimizationAlgorithm.`

You can call these methods:

1. Initialisation calls:

   ```
   algo.initializeDifferentialStates(M),
   algo.initializeControls(M),
   algo.initializeParameters(M),
   algo.initializeAlgebraicStates(M),
   algo.initializeDisturbances(M),
   ```

   where `M` is a matrix with in the first column a time reference (e.g. "0" when initializing on $t = 0$) and in the other columns values for all corresponding states, controls, disturbances or parameters. The order in which to define the initialisation values is the same order in which you have defined the expressions.

   Define two differential states:

   ```
   DifferentialState x1 x2.
   ```

   These states need to be initialized on $t = 0$: $x_1(0) = 2$ and $x_2(0) = 5$. Now `M` will become:

   ```
   M = [0 2 5].
   ```

   Here "0" is the first element (the time), the value corresponding to $x_1$ is the second element and the last element is a value for $x_2$. If you also want to provide initializations on other time points (e.g. when setting time varying parameters) you can add other rows. For example an initialisation on $t = 2$: $x_1(2) = 4$ and $x_2(2) = 10$ will become:

   ```
   M = [0 2 5; 2 4 10].
   ```

   See also the corresponding help files for more examples:

   ```
   help acado.OptimizationAlgorithm.initializeDifferentialStates,
   help acado.OptimizationAlgorithm.initializeAlgebraicStates, etc...
   ```

2. Another important function is `algo.set(setting, value)`. This methods allows you to set tolerances, integrator types, step sizes... for example:

   ```
   algo.set('INTEGRATOR_TOLERANCE', 1e-5 ).
   ```

   A full list with options is available in the help file:

   ```
   help acado.OptimizationAlgorithm.set.
   ```

### 4.2.3 A first example: a simple free space rocket

This first example regards a simple rocket model with 3 states: distance, velocity and mass. The aim is to fly in $t \in [0, 10]$ seconds from $s(0) = 0$ to $s(10) = 10$ while using minimum energy (represented by the control input $u$):

$$\boxed{\begin{array}{ll} \underset{s(\cdot),v(\cdot),m(\cdot),u(\cdot)}{\text{minimize}} & \int_0^{10} u(\tau)^2 \mathrm{d}\tau \\ \text{subject to:} & \\ \forall t \in [0,10] : & \dot{s}(t) = v(t) \\ & \dot{v}(t) = \frac{u(t)-0.2\,v(t)^2}{m(t)} \\ & \dot{m}(t) = -0.01\,u(t)^2 \\ & s(0) = 0\ ,\ v(0) = 0\ ,\ m(0) = 1 \\ & s(10) = 10\ ,\ v(10) = 0 \\ & -0.01 \le v(t) \le 1.3 \end{array}} \, . \tag{4.2.2}$$

**Using the ACADO notation:**
This first code snippet shows how to implement this while writing down the differential equation yourself in symbolic ACADO syntax. The resulting code is listed below and can be found in

`/interfaces/matlab/examples/ocp/simplerocket`:

```
BEGIN_ACADO;

    acadoSet('problemname', 'simplerocket');

    DifferentialState v;                   % Velocity
    DifferentialState s;                   % Distance
    DifferentialState m;                   % Mass
    DifferentialState L;                   % Dummy state

    Control u;                             % Control input


    %% Diferential Equation
    f = acado.DifferentialEquation();      % Differential equation object

    f.add(dot(s) == v);                    % Write down your ODE.
    f.add(dot(v) == (u-0.02*v*v)/(m));     %
    f.add(dot(m) == -0.01*u*u);            %
    f.add(dot(L) == u*u);                  % Dummy equation: \int{power}


    %% Optimal Control Problem
    ocp = acado.OCP(0.0, 10.0, 20);        % Start OCP at 0s, control in
                                           % 20 intervals upto 10s
    ocp.minimizeMayerTerm(L);                 % Minimize the consumed energy

    ocp.subjectTo( f );     % Optimize w.r.t. the differential equation
    ocp.subjectTo( 'AT_START', s ==  0.0 ); % s(0) = 0
    ocp.subjectTo( 'AT_START', v ==  0.0 ); % v(0) = 0
    ocp.subjectTo( 'AT_START', L ==  0.0 ); % L(0) = 0
    ocp.subjectTo( 'AT_START', m ==  1.0 ); % m(0) = 1
    ocp.subjectTo( 'AT_END'  , s == 10.0 ); % s(10) = 10 fly in 10 sec to
                                            % s=10 with minimum energy
    ocp.subjectTo( 'AT_END'  , v ==  0.0 ); % v(10) = 0  speed at the end
                                            % should be zero
    ocp.subjectTo( -0.01 <= v <= 1.3 );     % path constraint on speed


    %% Optimization Algorithm
    algo =acado.OptimizationAlgorithm(ocp); % Set up the optimization
        algorithm, link it to OCP
    algo.set( 'KKT_TOLERANCE', 1e-10 );     % Set a custom KKT tolerance

END_ACADO;
```

*Note that instead of using `ocp.minimizeLagrangeTerm( u*u )` we choose to introduce a Mayer term together with an extra differential equation. We do this because in the next code snippet a black-box will be linked. Remember from Section 4.2.1 that Lagrange terms do not work together with black-box models.*

**Using a Matlab black-box model:**
Instead of using the ACADO notation for differential equations, we will now link a black-box MATLAB model. This example can be found in `examples/ocp/simplerocket_matlab`. The only difference is that `rocketode.m` is now linked:

```
    f = acado.DifferentialEquation();
    f.linkMatlabODE('rocketode');
```

And thus `rocketode.m` contains:

```
function [ dx ] = rocketode( t,x,u,p,w )
    % THIS IS THE DEFINED SEQUENCE:
    % DifferentialState v s m L;
    % Control u;
    %
    % Thus x(1) = v, x(2) = s, x(3) = m and x(4) = L
    % Since only one control is set is u(1) = u.

    v = x(1);
    m = x(3);

    dx(1) = (u-0.02*v*v)/m;
    dx(2) = v;
    dx(3) = -0.01*u*u;
    dx(4) = u*u;
end
```

**Using a C++ black-box model:**
This last variant (see `examples/ocp/simplerocket_c`) links a C++ model:

```
    f = acado.DifferentialEquation();
    f.linkCFunction('cfunction.cpp', 'myAcadoDifferentialEquation');
```

And the corresponding C++ file is:

```
void myAcadoDifferentialEquation( double *x, double *f, void *user_data ){

    // x[0] -> time t
    // x[1] -> v
    // x[2] -> s
    // x[3] -> m
    // x[4] -> L
    // x[5] -> u

    double t = x[0];
    double v = x[1];
    double s = x[2];
    double m = x[3];
    double L = x[4];
    double u = x[5];

    f[0] =  (u-0.02*v*v)/(m);        //dot(v)
    f[1] =  v;                       //dot(s)
    f[2] =  -0.01*u*u;               //dot(m)
    f[3] =  u*u;                     //dot(L)

}
```

### 4.2.4 A second example: movement of a buoy

This next examples considers a wave power plant which aims to harvest energy using a buoy. The buoy moves on the waves and by using a clever control to subtract the cable connected to the buoy we hope to produce energy. This system is represented in Figure 4.2.

The buoy moves on a wave represented by a sinus:

$$h_w(t) = h_{hw} + A_{hw} * \sin\left(\frac{2\pi t}{T_{hw}}\right). \qquad (4.2.3)$$

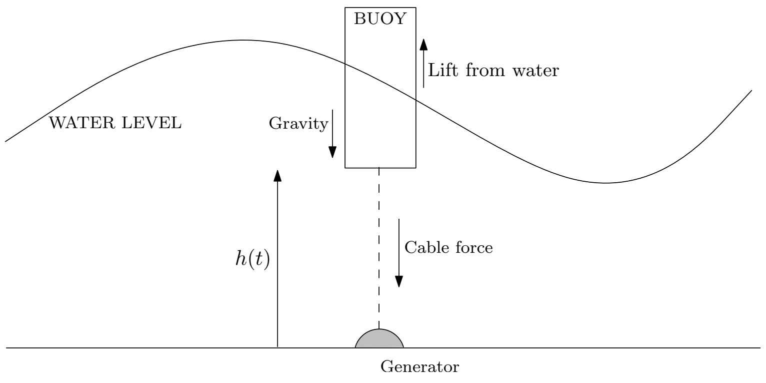

Figure 4.2: Schematic representation buoy.

The wave power plant optimal control formulation is now given by:

$$\boxed{\begin{array}{ll}\underset{s(\cdot),v(\cdot),m(\cdot),u(\cdot)}{\text{maximize}} & w \\ \text{subject to:} & \\ \forall t \in [0,15]: & \dot{h}(t) = v(t) \\ & \dot{v}(t) = \frac{\rho}{m}A(h_w(t)-h(t)) - g - u(t) \\ & \dot{w}(t) = u(t)v(t) \\ & h(0) = h_{hw} - A_{hw}\ ,\ v(0)=0\ ,\ w(0)=0 \\ & -h_b \leq h(t) - h_w(t) \leq 0.0 \\ & 0 \leq u \leq 100.0\end{array}} \ . \tag{4.2.4}$$

The problem results in the formulation found in `examples/ocp/wave_energy`:

```
BEGIN_ACADO;

    acadoSet('problemname', 'wave_energy');

    DifferentialState       h;              % Position of the buoy
    DifferentialState       v;              % Velocity of the buoy
    DifferentialState       w;              % Produced wave energy
    Control                 u;              % Control: adjust force
                                            % at the generator
    TIME                    t;              % the TIME

    h_hw  = 10;                             % water level
    A_hw  = 1.0;                            % amplitude of the waves
    T_hw  = 5.0;                            % duration of a wave
    h_b   = 3.0;                            % height of the buoy
    rho   = 1000;                           % density of water
    A     = 1.0;                            % bottom area of the buoy
    m     = 100;                            % mass of the buoy
    g     = 9.81;                           % gravitational constant
    %% Intermediate state
    hw = h_hw + A_hw*sin(2*pi*t/T_hw);       % Height of the wave


    %% Differential Equation
    f = acado.DifferentialEquation();        % Set the differential eq.
        object

    f.add(dot(h) ==  v);                     % ODE
    f.add(dot(v) ==  rho*A*(hw-h)/m-g-u);    %
    f.add(dot(w) ==  u*v);                   %


    %% Optimal Control Problem
    ocp = acado.OCP(0.0, 15.0, 100);         % Set up Optimal Control Problem
                                             % Start at 0s, control in 100
                                                 intervals to 15s

    ocp.maximizeMayerTerm( w );              % Maximize the energy

    ocp.subjectTo( f );                      % OCP w.r.t. differential eq.

    ocp.subjectTo( 'AT_START', h - (h_hw-A_hw) ==  0.0 );
    ocp.subjectTo( 'AT_START', v ==  0.0 );
    ocp.subjectTo( 'AT_START', w ==  0.0 );

    ocp.subjectTo( -h_b <= h-hw <= 0.0 );
    ocp.subjectTo( 0.0 <= u <= 100.0 );


    %% Optimization Algorithm
    algo = acado.OptimizationAlgorithm(ocp);

END_ACADO;

out = wave_energy_RUN();
```

Figure 4.3 summarizes the results.

## 4.3 ACADO Parameter Estimation

This section will describe how to formulate a parameter estimation problem using the Optimal Control framework. All examples used (and more) can be found in the examples directory:

`/interfaces/matlab/examples/parameterestimation/`.

We advise you to take a close look at these examples since they cover the most commonly encountered problems. Moreover, they are very well documented.

Figure 4.4 is very similar to the flow of an OCP and represents a parameter estimation problem. These components will now be explained in more detail.

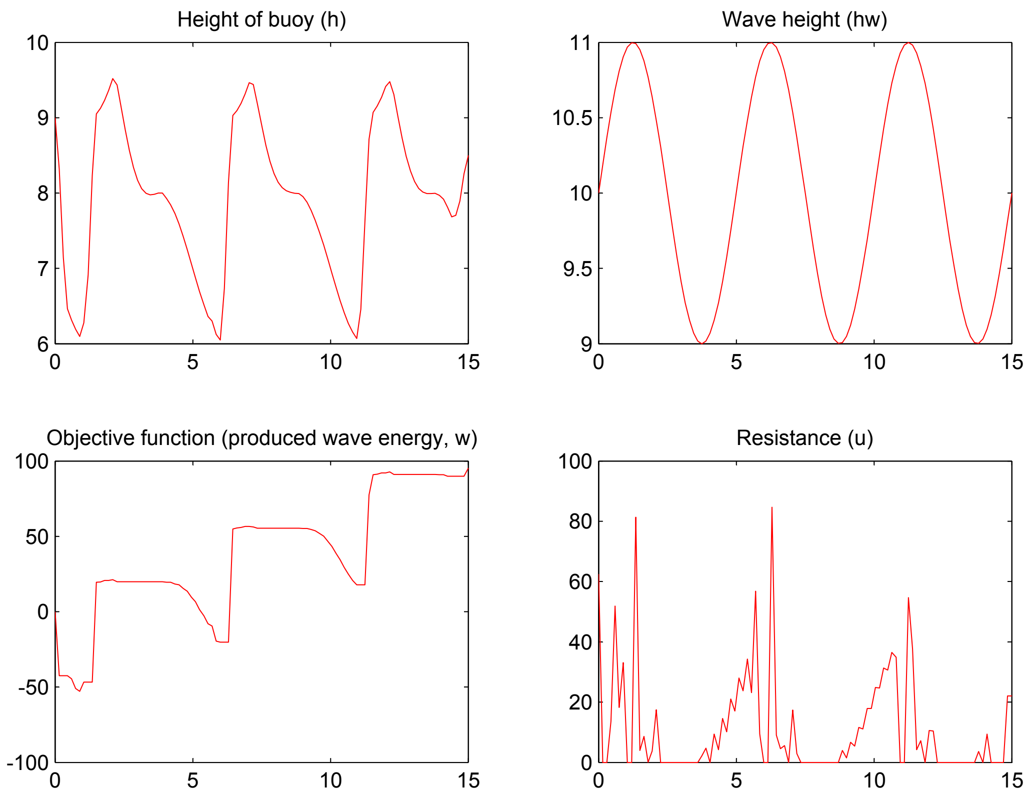

Figure 4.3: Results buoy.

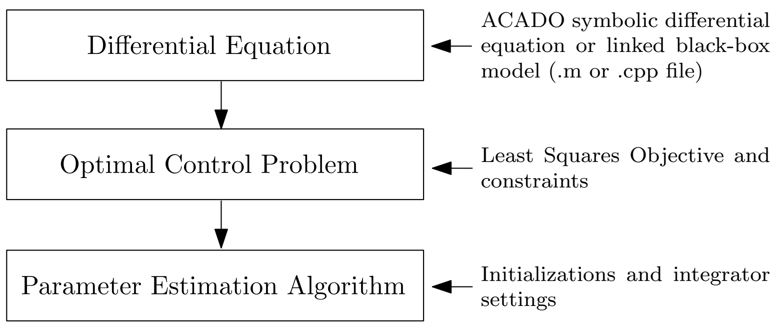
Differential Equation
ACADO symbolic differential equation or linked black-box model (.m or .cpp file)
Optimal Control Problem
Least Squares Objective and constraints
Parameter Estimation Algorithm
Initializations and integrator settings

Figure 4.4: Parameter Estimation Problem

### 4.3.1 Step 1: Getting your measurements

First of all, you will need to have some measurements available (in a matrix). The way you do this is completely the same as how you would initialize a state as descibed in Section 4.2.2. Suppose you have $q$ measurements of $N$ states, then $M \in \mathbb{R}^{q\times(N+1)}$ is the measurement matrix. The first column will contain time points, the other columns the states you are measuring. For example, if you have 10 measurements of 1 state, then $M \in \mathbb{R}^{10\times(1+1)}$:

```
    M = [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];
        % TIME        MEASUREMENT
```

*Note that ACADO can deal with non successful measurements leading to "NaN" entries. Moreover, the measurements do not have to be taken on an equidistant time grid.*

### 4.3.2 Step 2: The Optimal Control Formulation

Parameter estimation problems were introduced in Section 2.3. They can be handled as a special class of optimal control problems where a least squares objective needs to be minimized.

Set up the optimal control problem by creating an `acado.OCP` object. When dealing with a parameter estimation problem, you will have to define a vector containing the time points of your measurements instead of a start and end time. These time points will be the shooting nodes for the algorithm:

```
ocp = acado.OCP(timepoints).
```

If you store your data in a matrix `M`, the header will become:

```
ocp = acado.OCP(M(:,1)).
```

As you would with a "normal" OCP, you will now set the objective function (now a least squares term) and constraints. Setting up constraints is identical to Section 4.2.1, so we will only take a look at the LSQ-term:

```
ocp.minimizeLSQ([S,] h, M).
```

If $N$ states are being measured, then $S \in \mathbb{R}^{N\times N}$ is a weighting matrix, $h$ is a $1 \times N$ cell array containing the expressions for the measured states and $M$ is the reference. Because you have $q$ measurements for $N$ states, $M \in \mathbb{R}^{q\times(N+1)}$ is a matrix with $q$ rows and $N+1$ columns. The first column contains the time points, the other columns the reference.

Note: what if you would like to use a known time depending input? In this case introduce controls as you should in a normal optimal control problem. Next, you'll need to fix them to a certain trajectory. Do so by storing this trajectory in a matrix and by adding this time constraint: `ocp.subjectTo( u1 == myMatrix )` where `myMatrix` contains in the first column the time points and in the second column the time depending values.

### 4.3.3 Step 3: Linking a Parameter Estimation Algorithm

After having described the optimal control problem, it needs to be linked to a parameter estimation algorithm (completely analogous to Section 4.2.2). To link your OCP to a parameter estimation algorithm, simply write:

```
algo = acado.ParameterEstimationAlgorithm(ocp).
```

The same initialisation calls and settings are available as in Section 4.2.2. You could for example initialize the states you have measured or set a custom absolute tolerance.

### 4.3.4 An example: estimating parameters of a simple pendulum model

We consider a very simple pendulum model with differential states $\phi$ and $\omega$, representing the excitation angle and the corresponding angular velocity of the pendulum respectively. Moreover, the model of 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 $\alpha$. The parameter estimation problem of our interest has now the following form:

$$\boxed{\begin{array}{rl} \underset{\phi(\cdot),\alpha,l}{\text{minimize}} & \sum_{i=1}^{10} \left(\phi(t_i) - \eta_i\right)^2 \\ \text{subject to:} & \\ \forall t \in [0,T]: & \ddot{\phi}(t) = -\frac{g}{l}\phi(t) - \alpha\dot{\phi}(t) \\ & 0 \leq \alpha \leq 4 \\ & 0 \leq l \leq 2 \end{array}} \; . \tag{4.3.1}$$

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

The corresponding implementation is listed below as follows and can be found in

`examples/parameterestimation/getting_started`:

```
BEGIN_ACADO;

    acadoSet('problemname', 'getting_started');

    DifferentialState        phi;       % excitation angle
    DifferentialState        omega;     % angular velocity of the pendulum
    Parameter                l;         % the length of the pendulum
    Parameter                alpha;     % frictional constant
    Parameter                g;         % the gravitational constant
    %% Differential Equation
    f = acado.DifferentialEquation();  % Set differential equation object

    % possibility 1: link a Matlab ODE
    f.linkMatlabODE('myode');

    % possibility 2: write down the ODE directly in ACADO syntax
    % f.ODE(dot(phi ) == omega);
    % f.ODE(dot(omega) == -(g/l)*sin(phi) - alpha*omega);


    %% Optimal Control Problem
    % MEASUREMENT DATA. First column are time points, second column is phi.
    % At least two measurements are needed !!
    M = [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];

    % DEFINE A MEASUREMENT FUNCTION
    h={phi};   %The state phi is being measured.

    % DEFINE THE INVERSE OF THE VARIANCE-COVARIANCE MATRIX OF THE
      MEASUREMENTS:
    S = eye(1);
    S(1,1) = 1/(0.1)^2;     % (1 over the variance of the measurement)
                            % the standard deviation of the measurement
                            % is assumed to be 0.1, thus S = 1/(0.1)^2.

    ocp = acado.OCP(M(:,1));            % The OCP is evaluated on a grid
                                        % equal to the timepoints in
                                        % the measurement matrix M.

    ocp.minimizeLSQ( S, h, M );         % Minimize a least squares problem
                                        % with S, h and the measurements

    ocp.subjectTo( f );
    ocp.subjectTo( 0.0 <= alpha <= 4.0  );
    ocp.subjectTo( 0.0 <=   l   <= 2.0  );
    ocp.subjectTo( g == 9.81 );


    %% Optimization Algorithm
    % Setup the parameter estimation algorithm
    algo = acado.ParameterEstimationAlgorithm(ocp);

    % Initialize the differential states (optional)
    %algo.initializeDifferentialStates(M);
```

```
END_ACADO;

out = getting_started_RUN();
```

with as linked ODE file:

```
function [ dx ] = myode( t,x,u,p,w )

    % Just for ease of use we write down all variables. This is not needed.
    phi     = x(1);
    omega   = x(2);
    l       = p(1);
    alpha   = p(2);
    g       = p(3);

    % Differential equation.
    % The states are located in x. Dot(state) is located in dx. The index
    % refers to the corresponding state. Remark that this is the same
    % sequence how your states, parameters... are defined in your problem
    dx(1)  = omega;
    dx(2)  = -(g/l)*sin(phi) - alpha*omega;

end
```

The result returned on the console is:

```
Results for the parameters:
-----------------------------------------------
 l = 1.00103918e+00
 alpha = 1.84679609e+00
 g = 9.81000000e+00
```

These results are also stored in `out.PARAMETERS` in the sequence as they were defined in your ACADO problem.

## 4.4 ACADO Simulation Environment

This section describes the third problem class: using the ACADO Simulation Environment to formulate a Model Predictive Control (MPC) problem. All examples used (and more) can be found in the examples directory:

`/interfaces/matlab/examples/simulationenv/`.

We advise you to take a close look at these examples since they cover the most commonly used problems. Moreover, they are very well documented.

Figure 4.5 summarizes how a Model Predictive Control problem needs to be formulated using the ACADO simulation environment. Details follow further on in this section.

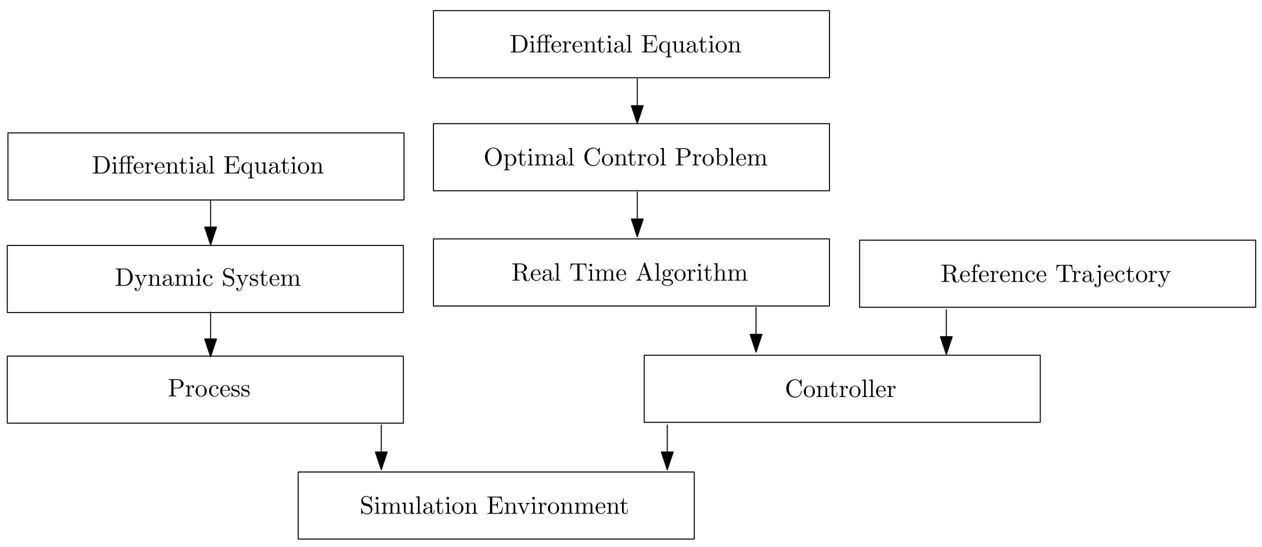

Figure 4.5: Simulation Environment

### 4.4.1 Step 1: The Optimal Control formulation

First of all you will need to define your Optimal Control Problem (objective(s) and constraint(s)). This step is 100% analogous to Section 4.2.1.

### 4.4.2 Step 2: Setting up the simulated process

The first building-block of an MPC simulation is a Process. The class `acado.Process` is one of the two main building-blocks within the Simulation Environment and complements the Controller. It simulates the to-be-controlled process based on a dynamic model and it also sets the integrator you would like to use:

```
process = acado.Process(dynamicSystem, 'INTEGRATOR_NAME').
```

Here `INTEGRATOR_NAME` will be one of the following options:

- `INT_RK12`: Explicit Runge-Kutta integrator of order 1/2
- `INT_RK23`: Explicit Runge-Kutta integrator of order 2/3
- `INT_RK45`: Explicit Runge-Kutta integrator of order 4/5
- `INT_RK78`: Explicit Runge-Kutta integrator of order 7/8
- `INT_BDF`: Implicit backward differentiation formula integrator.

This leaves only `dynamicSystem` to be set. The class Dynamic System is a data class for storing a differential equation together with an `acado.OutputFcn`. The differential equation can be another one than the one used in the OCP formulation. This means that is possible to have seperate differential equations for simulation and optimization. This is the call to set a dynamic system:

```
acado.DynamicSystem(f, outputFunction).
```

Here `f` is a differential equation object and `outputFunction` is simply an object of `acado.OutputFcn`:

```
outputFunction = acado.OutputFcn().
```

If the differential equation contains disturbances, you can link these disturbances to the process. This requires knowledge of the values of this disturbance over time:

```
process.setProcessDisturbance(W).
```

Here `W` is a matrix analogous to the measurement matrix of an optimal control problem or initialisation matrix in an optimization algorithm. The rows are different measurements (say $q$) and the columns contain the different measured values of the disturbances ($N$ different disturbances) over time. The first column contains the time points, thus $W \in \mathbb{R}^{q\times(N+1)}$.

In the event that you would like to initialize algebraic states (if you have any), call:

```
process.initializeAlgebraicStates(v).
```

Here `v` is a vector containing initial values for all algebraic states. Please note that setting `algo.initializeAlgebraicStates` in a `RealTimeAlgorithm` will not have the same result. Your differential states can be initialized in either the `Controller` if you are using it stand-alone or in the `SimulationEnvironment`.

### 4.4.3 Step 3: Setting up the MPC controller

Now comes the part where a controller will be set up. The controller complements a Process. It contains an on-line control law (e.g. a DynamicFeedbackLaw comprising a RealTimeAlgorithm) for obtaining the control inputs of the Process.

```
controller = acado.Controller( controllaw [, reference] ).
```

The `controllaw` is at the moment just an `acado.RealTimeAlgorithm`. This real time algorithm is comparable to an optimization algorithm or parameter estimation algorithm: you use it to link an OCP, set integrator options etc. In addition to these other algorithms, you can also set the sampling time:

```
controllaw = acado.RealTimeAlgorithm(ocp, SAMPLING_TIME).
```

The field `reference` needs an object of the type `acado.ReferenceTrajectory`. This can either be a static reference trajectory:

```
reference = acado.StaticReferenceTrajectory([r])
```

or a periodic reference trajectory:

```
reference = acado.PeriodicReferenceTrajectory([r]).
```

Here `r` is an optional argument containing a matrix with reference values. This matrix has time references in the first column and reference values for each of the elements of the Least Squares Term (OCP) in the other columns. Suppose your LSQ-Term consists of 4 entries (let's say 2 states and 2 controls). In this case the matrix `r` will contain 4+1 columns. The easiest way to use a reference trajectory is by just calling:

```
zeroReference = acado.StaticReferenceTrajectory()
```

to invoke a zero-reference trajectory. If you just make sure your LSQ-Term want's to track to zero (the reference in the LSQ-Term is zero), then your done.

### 4.4.4 Step 4: Linking it to a Simulation Environment

In this last step the process and controller must be linked by using an `acado.Simulation Environment` object:

```
sim = acado.SimulationEnvironment(startTime, endTime,
                    process, controller).
```

Here `startTime` and `endTime` are the start and end time of the closed-loop simulation, `process` and `controller` are the objects defined above. Optionally initialisation values for the states can be defined:

```
sim.init( x0 ),
```

where `x0` is a vector $x(0) \in \mathbb{R}^{1\times n_x}$ containing starting values for all states.

*A remark about the start and end time of the OCP and simulation environment: the OCP is the off-line problem to be solved repeatedly on-line; it is very natural that its horizon is (much) shorter than the simulation horizon. However, the OCP horizon has to be longer than the sampling time.*

### 4.4.5 A first example: Controlling a simple quarter car

In this first example, we consider a simple car model with active suspension, as shown in Figure 4.6 (only one of the four wheels is shown).

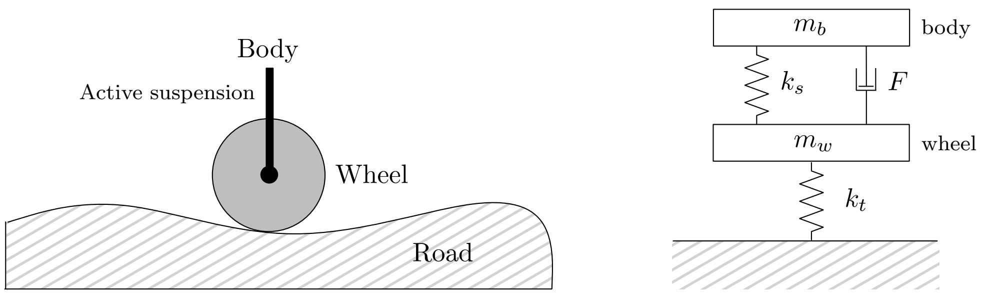

Figure 4.6: Wheel with active suspension. [1]

The states should be kept close to a given reference, making use of active damping. 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 $F$ is a limited damping force acting between the body and the wheel. The road, denoted by the variable $R$, is considered as an unknown disturbance. Now, the dynamic equations have the following form:

$$\boxed{\begin{array}{ll}
\underset{x_b,x_w,v_b,v_w,F}{\text{minimize}} & \int\limits_0^1 \|y(t)\|_Q^2 \, \mathrm{d}\tau \\
\text{subject to:} & \\
\forall t \in [0,1]: & \dot{x}_b(t) = v_b(t) \\
 & \dot{x}_w(t) = v_w(t) \\
 & \dot{w}(t) = u(t)v(t) \\
 & \dot{v}_b(t) = \frac{1}{m_b}\left[-k_s x_b(t) + k_s x_w(t) + F(t)\right] \\
 & \dot{v}_w(t) = \frac{1}{m_w}\left[-k_t x_b(t) - (k_t + k_s)x_w(t) + k_t R(t) - F(t)\right] \\
 & y(t) = (x_b(t), x_w(t), v_b(t), v_w(t))^T \\
 & -500 \ \leq \ F(t) \ \leq \ 500
\end{array}} \ . \qquad (4.4.1)$$

The objective of the MPC problems is now to bring the body and the wheel back to zero displacement, which can be achieved if there are no disturbances arising from the road. I.e. the function $h$ is in our case simply equal to the differential state vector. Moreover, we would like to regard control constraints of the form $-500 \leq F(t) \leq 500$.
The implementation can be found in

```
/interfaces/matlab/examples/simulationenv/active_damping
```

and is listed below:

```
BEGIN_ACADO;

    acadoSet('problemname', 'active_damping');
    acadoSet('results_to_file', false);     % Don't write results to file.

    DifferentialState xB xW vB vW;          % Differential States:
                                            % xBody, xWheel, vBody, vWheel

    Control F;                              % Control:
                                            % dampingForce

    Disturbance R;                          % Disturbance:
                                            % roadExcitation

    mB = 350.0;                             % Some static parameters
    mW = 50.0;
    kS = 20000.0;
    kT = 200000.0;
    %% Differential Equation
    f = acado.DifferentialEquation();

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


    %% Optimal Control Problem
    ocp = acado.OCP(0.0, 1.0, 20);

    h={xB, xW, vB, vW};                     % the LSQ-Function

    Q = eye(4);                             % The weighting matrix
    Q(1,1) = 10;
    Q(2,2) = 10;

    r = zeros(1,4);                         % The reference

    ocp.minimizeLSQ( Q, h, r );             % Minimize Least Squares Term
    %ocp.minimizeLSQ( h, r );               % (Other possibility)
    %ocp.minimizeLSQ( h );                  % (Other possibility)

    ocp.subjectTo( f );
    ocp.subjectTo(  -500.0 <= F <= 500.0 );
    ocp.subjectTo(  R == 0.0 );


    %% SETTING UP THE (SIMULATED) PROCESS
    identity = acado.OutputFcn();
    dynamicSystem = acado.DynamicSystem(f, identity);
        % A dynamic system with the differential equation and an output
           function

    process = acado.Process(dynamicSystem, 'INT_RK45');
        % Simulate proces based on a dynamic model.

    disturbance = [                         % process disturbance matrix
        0.1       0.01
        0.15      0.01
        0.2       0.00
        5.0       0.00];
    process.setProcessDisturbance(disturbance);


    %% SETTING UP THE MPC CONTROLLER:
    algo = acado.RealTimeAlgorithm(ocp, 0.05);
        % The class RealTimeAlgorithm serves as a user-interface to
           formulate and solve model predictive control problems.

    algo.set('MAX_NUM_ITERATIONS', 2 );
        % Set some algorithm parameters

    zeroReference = acado.StaticReferenceTrajectory();
```

```
        % Static reference trajectory that the ControlLaw aims to track.
           Use a zero reference since the LSQ−Term has a zero reference as
           well.

    controller = acado.Controller( algo,zeroReference );
        % Online control law for obtaining the control inputs of a process


    %% SETTING UP THE SIMULATION ENVIRONMENT,  RUN THE EXAMPLE..
    sim = acado.SimulationEnvironment( 0.0,3.0,process,controller );
        % Setup the closed−loop simulations, Simulate from 0 to 3 sec

    r = zeros(1,4);                        % Initialize the states
    sim.init( r );

END_ACADO;

out = active_damping_RUN();
```

Figure 4.7 summarizes the results using the `draw` command. Here, we have simulated the road disturbance, which is displayed in the lower right part of the figure. 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.

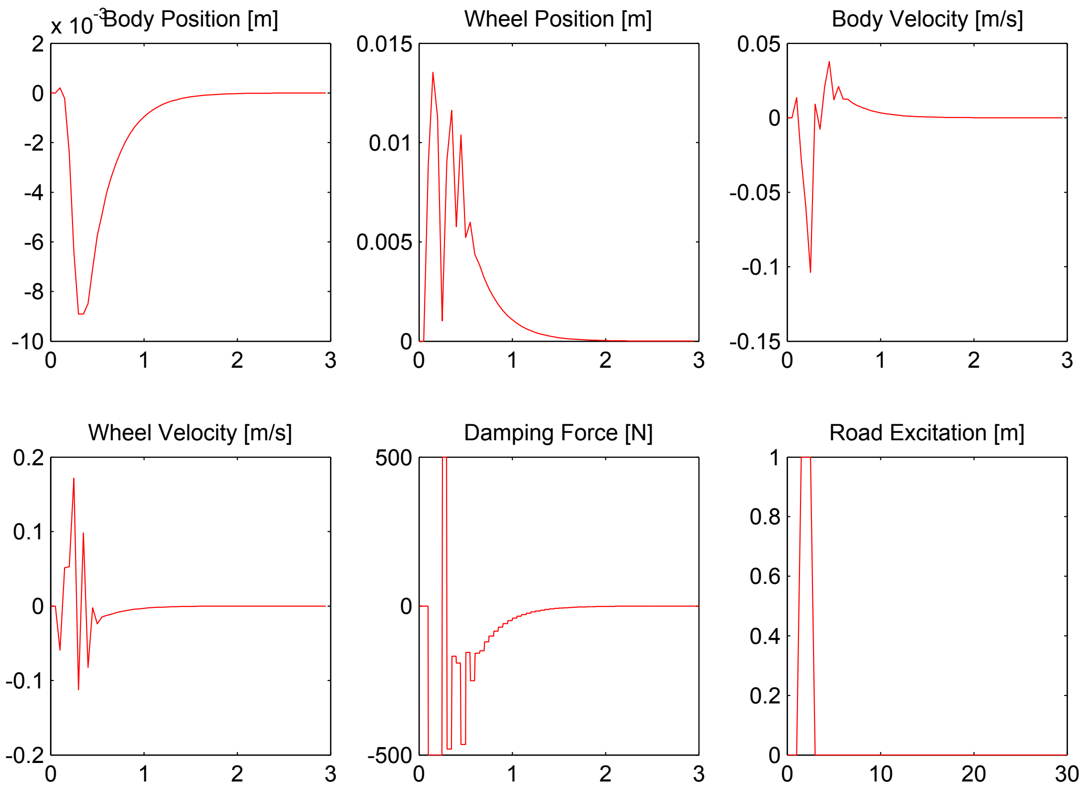

Figure 4.7: Results Active Damping.

### 4.4.6 A second example: Periodic tracking

This second examples introduces a periodic reference trajectory and uses two different differential equations written as C++ code. This example can be found in

/interfaces/matlab/examples/simulationenv/periodic_tracking_c

but also a MATLAB and ACADO version of the differential equation are available in

/interfaces/matlab/examples/simulationenv/periodic_tracking_matlab and
/interfaces/matlab/examples/simulationenv/periodic_tracking.

```
BEGIN_ACADO;

    acadoSet('problemname', 'periodic_tracking');

    DifferentialState x;
    Control           u;
    Disturbance       w;


    % Differential Equation
    f = acado.DifferentialEquation();
    f.linkCFunction('cfunction1.cpp', 'myAcadoDifferentialEquation1');

    f2 = acado.DifferentialEquation();
    f2.linkCFunction('cfunction2.cpp', 'myAcadoDifferentialEquation2');


    % OCP
    h={x u};
    Q = eye(2);
    r = zeros(1,2);

    ocp = acado.OCP(0.0, 7.0, 14);
    ocp.minimizeLSQ( Q, h, r );

    ocp.subjectTo( f );
    ocp.subjectTo( -1.0 <= u <= 2.0 );
    %ocp.subjectTo( w == 0.0 );


    % SETTING UP THE (SIMULATED) PROCESS:
    identity = acado.OutputFcn();
    dynamicSystem = acado.DynamicSystem(f2, identity);
    process = acado.Process(dynamicSystem, 'INT_RK45');
    disturbance = [
        0.0       0.00
        0.5       0.00
        1.0       0.00
        1.5       1.00
        2.0       1.00
        2.5       1.00
        3.0       0.00
        3.5       0.00
        4.0       0.00
        15.0      0.00
        30.0      0.00];
    process.setProcessDisturbance(disturbance);


    % SETUP OF THE ALGORITHM AND THE TUNING OPTIONS:
    algo = acado.RealTimeAlgorithm(ocp, 0.5);
    algo.set( 'HESSIAN_APPROXIMATION', 'GAUSS_NEWTON' );
    algo.set('MAX_NUM_ITERATIONS', 2 );


    % SETTING UP THE NMPC CONTROLLER:
    ref = [0.0       0.00       0.00    % Set up a given reference
        trajectory
        0.5       0.00       0.00       % This trajectory is PERIODIC!
        1.0       0.00       0.00       % This means that the trajectory
        1.25      0.00       0.00       % will be repeated until the end
        1.5       0.00       0.00       % of the simulation is reached.
        1.75      0.00       0.00
        2.0       0.00       0.00
        2.5       0.00       0.00
        3.0       0.00       -0.50
        3.5       0.00       -0.50
        4.0       0.00       0.00];
    %   TIME      X_REF      U_REF

    reference = acado.PeriodicReferenceTrajectory(ref);
    controller = acado.Controller( algo,reference );


    % SETTING UP THE SIMULATION ENVIRONMENT,  RUN THE EXAMPLE..
    sim = acado.SimulationEnvironment( 0.0,15.0,process,controller );

    r = zeros(1,1);
    r(1,1) = 1;
    sim.init( r );


END_ACADO;

out = periodic_tracking_RUN();
```

With as corresponding C++ file `cfunction1.cpp`:

```
void myAcadoDifferentialEquation1( double *x, double *f, void *user_data ){

    double t = x[0];
    double xx = x[1];
    double u = x[2];
    double w = x[3];

    f[0] = -2.0*xx + u;
}
```

And cfunction2.cpp:

```
void myAcadoDifferentialEquation2( double *x, double *f, void *user_data ){

    double t = x[0];
    double xx = x[1];
    double u = x[2];
    double w = x[3];

    f[0] = -2.0*xx + u + 0.1*w;

}
```

Figure 4.8 summarizes the results. The impact of the disturbance at $t = 1.5$ is clearly visible in the state.

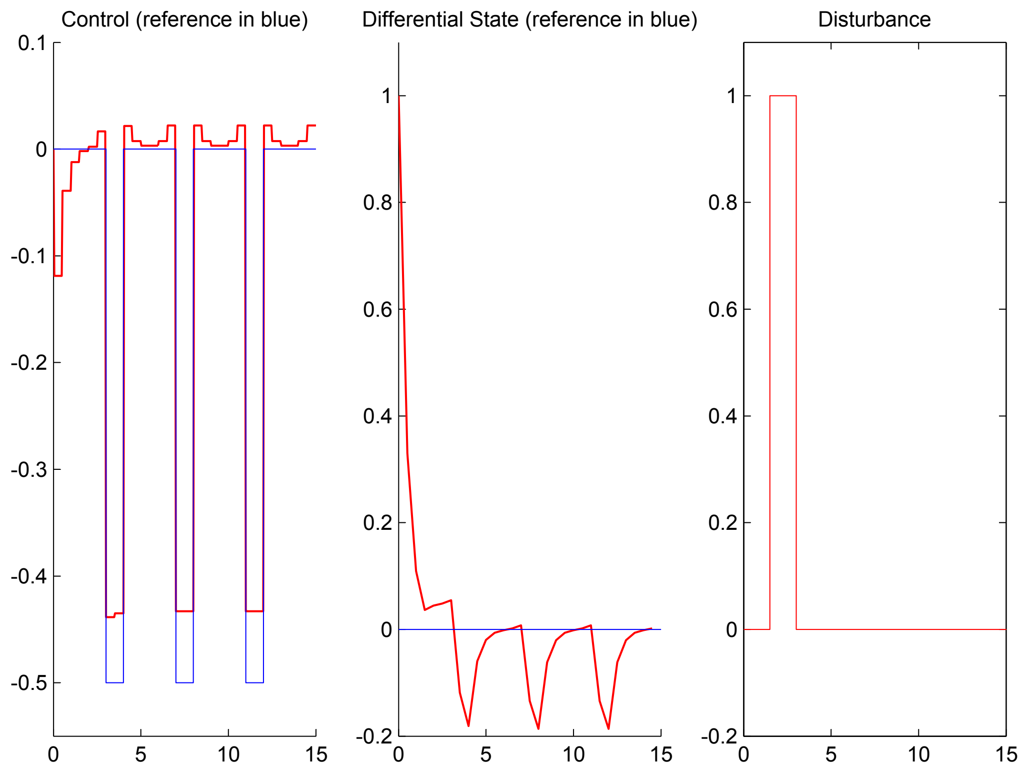

Figure 4.8: Results Periodic Tracking.

# Chapter 5

# Tutorials: Advanced Features

## 5.1 MEX Input and Output

Normally you should always recompile your problem after changing certain values, disturbances, initializations, bounds etc. It is also possible to use variable input arguments which are not compiled to a MEX file, but are handled as free arguments of the right hand side of your problem. This way you can introduces numeric values, vectors and matrices in a dynamic way. Different examples can be found in:

`/interfaces/matlab/examples/matlabarguments/`.

To use "dynamic" parameters (values that can change without recompiling your problem), you'll need to use "mexInput's". At the start of your ACADO problem description you can define 3 different "mexInput's":

- `input = acado.MexInput`: represents a free to choose scalar input. Use this as variable in a differential equation, a bound, the number of samples, an end time...
- `input = acado.MexInputVector`: represents a vector of undefined length. Use it for example to pass through a time grid in a parameter estimation problem or to initialize a controller with a vector containing initial values for states.
- `input = acado.MexInputMatrix`: represents a matrix of undefined size. Use this for a weighting matrix, a matrix containing a periodic reference trajectory, measurements in your parameter estimation problem...

Once you have defined `input` you can use it as a symbolic expression in nearly every spot you want[1].

After compiling you should now run your problem file with the argument you have defined, for example define:

```
acadoSet('problemname', 'myexample');
```

[1]Note: At the moment you cannot use a "mexInput" in `algorithm.set` yet

```
DifferentialState x1 x2 x3;

input1 = acado.MexInput;
input2 = acado.MexInputVector;
input3 = acado.MexInputMatrix;

[...]

ocp = acado.OCP(0.0, 2, input1);      % A scalar

[...]

ocp.minimizeLSQ( input3, h, r );      % A weighting matrix

[...]

controller.init(2, input2);           % A vector

[...]
```

Now run it with the three arguments as defined in the same sequence:

`myexample_RUN(10, [0 1 2 3], eye(3,3))`,

without needing to recompile your problem.

## 5.2 Using a controller stand-alone

You can use the ACADO controller without needing to use the simulation environment. This way it's possible to link a controller to a real-life process. This type of problems leads perfectly to the use of "mexInput's", resulting in this example:

`examples/matlabarguments/simulationenv_dev_dcmotor/dev_dcmotor2.m`.

What you would like to do is to call the next step of the ACADO controller and return the result to use it in your process. While calling you will always need to pass through a time and initialisation vector. This is why you should use "mexInput's": they allow you to pass new information to a compiled problem without having to recompile.

Setting up a stand-alone controller is fairly easy. Start by defining your differential equation and OCP as you always should. Next, set up a real-time algorithm, reference trajectory and controller. This is also how you would have done it before. The difference now is that we will not set up a process, since our process is a real-life one. In contrast to using the simulation environment, you will now have to call `controller.init` to initialize the controller and `controller.step` to get the next optimized controls.

Figure 5.1 gives a schematic overview of all needed components. To indicate the difference between Figure 4.5 (simulated process) and this one (real process) we repeated Figure 4.5, but now with a line separating the stand-alone controler (right) to the entire simulation environment.

This code sample implements the needed calls:

```
    input1_startime = acado.MexInput;         % Startime is free
    input2_x0 = acado.MexInputVector;         % x0 is free

    % [.....]


    %% SETTING UP THE MPC CONTROLLER:
    algo = acado.RealTimeAlgorithm(ocp, 0.05);

    %% CONTROLLER
    ref = [ 0.0    0.0
            1.0    0.2
            3.0    0.8
            5.0    0.5];
    reference = acado.PeriodicReferenceTrajectory(ref);

    controller = acado.Controller( algo,reference );

    controller.init(input1_startime, input2_x0);
    controller.step(input1_startime, input2_x0);
```

In the output struct the next optimal controls will be stored. It contains two fields: `out.U` and `out.P`, both vectors containing the optimized controls and parameters.

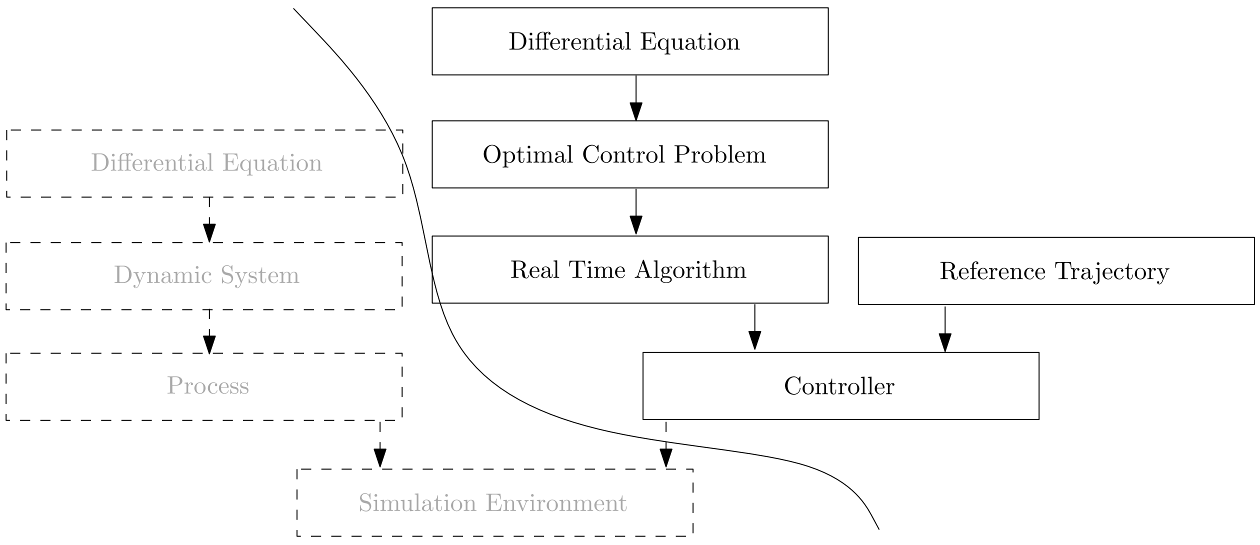

Figure 5.1: Stand-alone controller

## 5.3 Writing and compiling your own C++ file

Although we try to implement as much features as possible, some people may still decide to write their own C++ code and compile it themselves to MATLAB `MEX` files. This way all features of the C++ version of the ACADO Toolkit are available in a MATLAB environment.

To help you building and compiling your own C++ file, we introduced "`makemex`". `Makemex` automatically compiles each `MEX` function to an executable. Furthermore, we provide templates and methods to help you dealing with common actions such as plotting and linking MATLAB ODE files.

The folder `examples/mexfiles/` contains many examples which can be compiled in a very easy way. To compile, go to the top directory containing all MATLAB `make` methods and run:

```
makemex('examples/mexfiles/empty.cpp', 'empty', 'examples/mexfiles/').
```

The first element is the full path to the C++ file, the second on is the name of the executable and the last one is the folder where the executable should be stored. After compiling the executable can now be called:

```
cd ('examples/mexfiles/')
empty().
```

The example folder contains many easy examples getting you started writing most methods. Note that because you are now writing C++ code, it's now possible to use all C++ examples of the ACADO Toolkit. What you could do as well is generate a C++ file with the interface and copy-paste the content (or parts of the content) to a new file.

## 5.4 Multi Stage OCP

When two different models should be executed after each other multi stage optimal control problems can be used. Examples are available in `examples/multi_stage_ocp/`. Only a few things need to change:

- Define multiple differential equations and pass the start and end time as parameters with the differential equation instead of with the OCP:

```
    f = acado.DifferentialEquation(0.0, 5.0); % Differential equation 1
    f.add(dot(x) == -x + 0.01*x*x + u);       % from 0.0 to 5.0
    f.add(dot(L) == -x + 0.01*x*x + u);

    g = acado.DifferentialEquation(5.0,10.0); % Differential equation 2
    g.add(dot(x) ==  x - 2.00*x*x + u);
    g.add(dot(L) ==  x*x + u*u);
```

- Set-up an OCP without further options:

```
    ocp = acado.OCP();
```

- Link all differential equations to the OCP and indicate the number of control intervals:

```
    ocp.subjectTo( f, 30 );
    ocp.subjectTo( g, 30 );
```

# Chapter 6

# Developer reference

## 6.1 ACADO for Matlab class structure and code generating methods

The class structure of ACADO for Matlab is pictured in Figure 6.1. The class names and inheritance is nearly identical to that of the ACADO Toolkit. Some classes were added specifically for the interface:

- `AcadoMatlab`: very important class, responsible for the C++-file generation,
- `EmptyWrapper`: not important, represents an empty object,
- `MexInput`, `MexInputVector`, `MexInputMatrix`: See Section 5.1,
- `SimulationEnvironment` inherits from `OptimizationAlgorithmBase` and `ControlLaw` instead of just from `ControlLaw` because `SimulationEnvironment` should implement all methodes available in `OptimizationAlgorithmBase`.

Furthermore the methods `getInstructions` and `toString` were also added to most classes:

- `toString` is only used in expressions. When, for example, an expression is an addition (which has two objects), then `toString` will be called on this object to print it. On it's turn, `toString` will now be called on each of the two objects, thus creating a recursive path and returning the expression (eg `x+(y+sin(u))*(u)`).
- `getInstructions` is the most important call you will use when implementing new functionality. Every class the user can use in his problem should have a `getInstructions` method. Furthermore, the constructor of this class should have this line of code:

  ```
  ACADO_.helper.addInstruction(obj).
  ```

  By using this call, the class will be logged in the helper class `AcadoMatlab`. The helper class will also remember the sequence of these calls. When the problem needs to be compiled, the helper class will ask at each of the entries in it's instruction list to execute `getInstructions`, which on it's turn will write some lines of C++ code to the C++ file (see also Section 6.3).

*Remark:* `ACADO_` *is a global variable used throughout the software and contains important compilation and problem information. When using it, first call* `global ACADO_` *in order to get access to global variables in your method.*

SVN VERSION 2031
DATE 04/06/2010

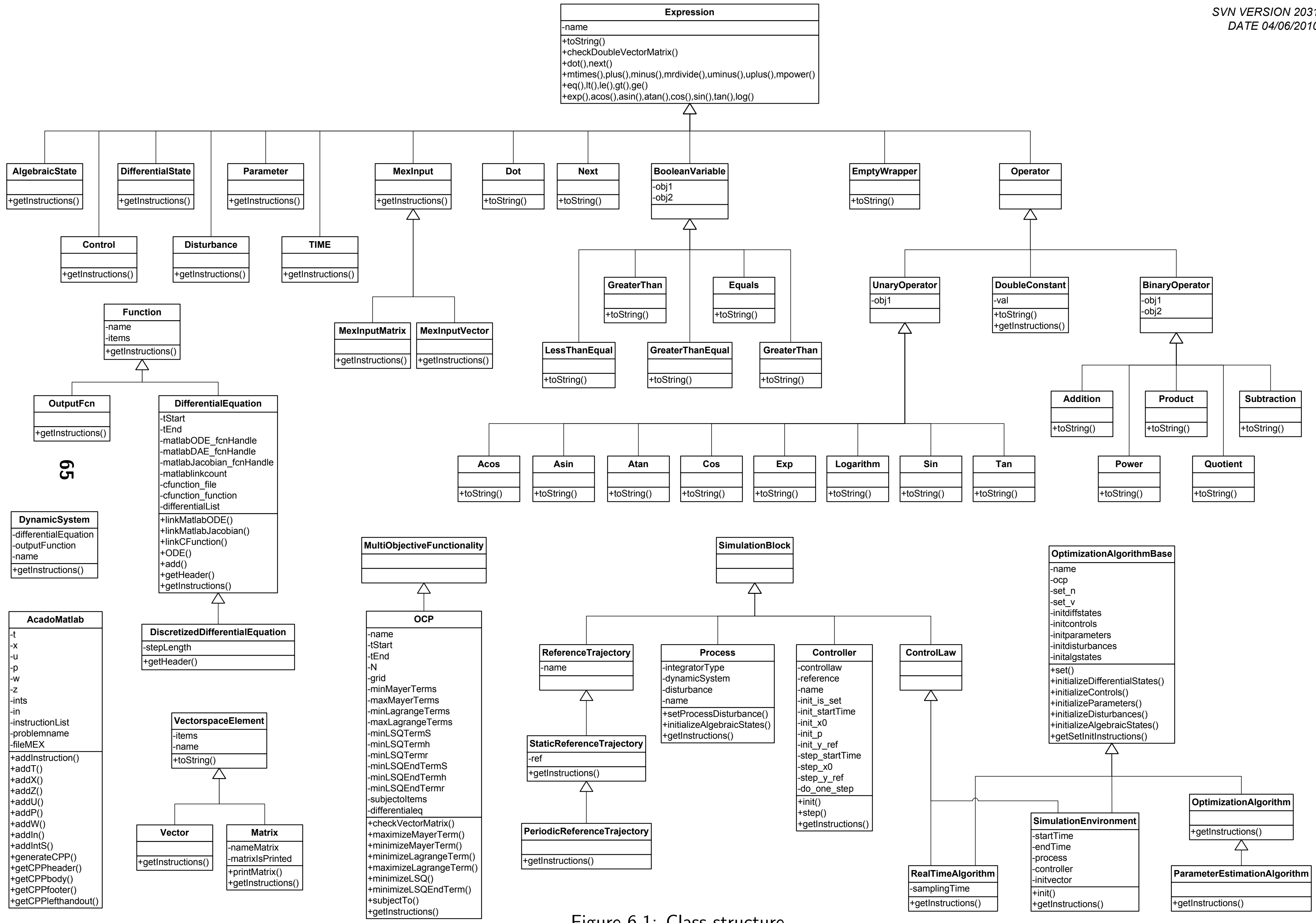

Figure 6.1: Class structure

## 6.2 Documentation

The complete function reference is available at `http://www.acadotoolkit.org/matlab/doc/`. The documentation is automatically generated by `m2html` using the interface's documentation.

When changing or adding classes, use the same documentation style to make sure all documentation can properly be extracted:

```
%Short one line introduction
% Longer multi-line
% text explaining all features
%
%  Usage:
%    >> class(obj1, obj2);  Explain...
%    >> class(obj1);        Explain...
%
%  Parameters:
%    obj1       explain....       [NUMERIC]
%    obj2       explain....       [NUMERIC/PARAMETER]
%
%  Example:
%    >> an example...
%
%  See also:
%    acado.class.method1
%    acado.class.method2
%
%  Licence:
%    This file is part of ACADO Toolkit  - (http://www.acadotoolkit.org/)
%
%    ACADO Toolkit, toolkit for Automatic Control and Dynamic Optimization.
%    Copyright (C) 2008-2009 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 free software; you can redistribute it and/or
%    modify it under the terms of the GNU Lesser General Public
%    License as published by the Free Software Foundation; either
%    version 3 of the License, or (at your option) any later version.
%
%    ACADO Toolkit is distributed 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.
%
%    You should have received a copy of the GNU Lesser General Public
%    License along with ACADO Toolkit; if not, write to the Free Software
%    Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston,
%    MA  02110-1301  USA
%
%    Author: ....
%    Date: ...
```

In a `classdef` file, put the documentation at the top (before writing `classdef`). In a `function` file, put the documentation below the function header.

## 6.3 C++ Code Generation

This section will offer a short tutorial-style manual about how to add new functionality to the interface. First of all it's important to see how the generated C++ file is structured. Just open any generated file in one of the examples to see the result yourself.

```
#include <acado_optimal_control.hpp>
#include <acado_toolkit.hpp>
#include <acado/utils/matlab_acado_utils.hpp>

USING_NAMESPACE_ACADO


// INCLUDE HEADER HERE


void mexFunction( int nlhs, mxArray *plhs[], int nrhs, const mxArray
    *prhs[] )
 {

    clearAllStaticCounters( );

    acadoPrintf("\nACADO Toolkit for Matlab - Developed by David Ariens,
        2009-2010 \n");
    acadoPrintf("Support available at http://www.acadotoolkit.org/matlab \n
        \n");

    if (nrhs != XX){
      mexErrMsgTxt("This problem expects XX right hand side argument(s)
          since you have defined XX MexInput(s)");
    }

    // INCLUDE BODY HERE


    // INCLUDE FOOTER HERE

    clearAllStaticCounters( );

}
```

The header can include helper functions (eg: used to link black-box models), the body will include the actual optimization problem and the footer can include things the software should do after the optimization problem is executed. Don't worry, normally you'll only need to use the "body" part.

First thing to do is to "register" your class when it's called. This way `AcadoMatlab` (the helper class) knows that this is class which should be written to a C++ file. Do so by adding this line to the class constructor:

```
global ACADO_;
ACADO_.helper.addInstruction(obj);
```

The helper class will call `getInstructions` on each object that is in it's list. The method will look like this:

```
function getInstructions(obj, cppobj, get)

    if (get == 'H')
        % Things that should be in header (helper functions)

    elseif (get == 'FB')
        % Things that should be first in body (eg constants)

    elseif (get == 'B')
        % Things that should be in body (the optimization problem)

    elseif (get == 'F')
        % Things that should be in the footer (clean-up, plotting)
    end

end
```

In practice your method will just look like this:

```
function getInstructions(obj, cppobj, get)

    if (get == 'B')
        % Things that should be in body

    end

end
```

The `getInstructions` method now only needs to write the needed information to a C++ file. You can do this by writing your commands directly to the correct file. Writing to a file can be done by calling `fprintf`, the correct pointer can be found in `cppobj.fileMEX`:

```
fprintf(cppobj.fileMEX,'print some text to the .cpp file \n');
```

An example of such a method is listed below (the instructions for the Controller). Note the usage of `sprintf`:

```
function getInstructions(obj, cppobj, get)

if (get == 'B')

    if (~isempty(obj.reference.name))
        fprintf(cppobj.fileMEX,sprintf('    Controller %s( %s,%s );\n',
           obj.name, obj.controllaw.name, obj.reference.name));
    else
        fprintf(cppobj.fileMEX,sprintf('    Controller %s( %s );\n',
            obj.name, obj.controllaw.name));
    end

    if (obj.init_is_set == 1)
        % [...]

        if (~isempty(obj.init_y_ref))   %
            fprintf(cppobj.fileMEX,sprintf('    %s.init(%s, %s, %s,
                %s);\n', obj.name, obj.init_startTime.name,
                obj.init_x0.name, obj.init_p.name, obj.init_y_ref.name));
        elseif (~isempty(obj.init_p))
            fprintf(cppobj.fileMEX,sprintf('    %s.init(%s, %s, %s);\n',
                obj.name, obj.init_startTime.name, obj.init_x0.name,
                obj.init_p.name));
        elseif (~isempty(obj.init_x0))
            fprintf(cppobj.fileMEX,sprintf('    %s.init(%s, %s);\n',
                obj.name, obj.init_startTime.name, obj.init_x0.name));
        else
            fprintf(cppobj.fileMEX,sprintf('    %s.init(%s);\n', obj.name,
                obj.init_startTime.name));
        end
    end

    % [...]

    fprintf(cppobj.fileMEX,'\n');
end

end
```

That's it! To summarize: log your class to the helper file and implement a `getInstructions` method.

## 6.4 Overview of important files and folders

The folder `<ACADOtoolkit-inst-dir>/interfaces/matlab/` includes many interesting files and folders:

- `acado`: all files of the optimization interface.
    - `functions`: some general functions that should be available in the entire interface. This folder is always added to the Matlab path when using the optimization interface (and is thus always available).
    - `keywords`: includes some special keywords used in ACADO. This folder is only available after calling `BEGIN_ACADO` and is removed from the path after calling `END_ACADO`.
    - `packages`: includes the package "+acado" which contains all class files.
- `bin`: all compiled object files in different sub folders. The files that should be compiled are listed in `shared/objects.m`.

- `examples`: all examples (integrator and optimization interface).
- `integrator`: `MEX` file and other needed `m`-files for the integrator interface.
    - `ACADOintegrators.cpp`: implementation of integrator interface.
    - `ACADOintegrators.m`: Matlab file used to call `ACADOintegrators.cpp`.
    - `ACADOintegratorsOutputs.m`: contains default output struct and help file.
    - `ACADOintegratorsSettings.m`: contains default settings struct and help file.
    - `model_include.hpp`: automatically generated header file (generated by `shared/automatic_model_detection_integrator.m`).
    - `plot_trajectory.m`: file used to automatically generate plots for the integrator.
- `shared`: important files used in the entire interface.
    - `acadoglobals.m`: automatically generated file containing important compiler information (generated by `makehelper.m`).
    - `automatic_model_detection_integrator.m`: generates `integrator/model_include.hpp` (called by `makehelper.m`).
    - `generic_dae.m`: function used by the integrator and optimization interface to call Matlab DAE functions.
    - `generic_jacobian.m`: function used by the integrator and optimization interface to call Matlab Jacobians.
    - `generic_ode.m`: function used by the integrator and optimization interface to call Matlab ODE functions.
    - `objects.m`: contains a list of all source files which should be compiled when running `make`.
- `MODEL_INCLUDE.m`: includes a list of all C++ files which needs to be compiled when the integrator is compiled. These files can afterwards be used as models in the integrator interface.
- `make.m`: Make the integrators and optimization interface. Calls `makehelper.m`.
- `makeintegrators.m`: Make only the integrators. Calls `makehelper.m`.
- `makeocp.m`: Make only the optimal control optimization interface. Calls `makehelper.m`.
- `makesimulation.m`: Make only the optimal control optimization interface and simulation environment. Calls `makehelper.m`.
- `makemex.m`: Make an individual MEX file. Calls `makehelper.m`.

# Index

ACADO for Matlab, 5
  Installation, 6
  Key Properties, 6
ACADO Toolkit, 5
ACADOintegrators, 21
ACADOintegratorsSettings, 21
acadoSet, 29
AlgebraicState, 29

BEGIN_ACADO, 29
Black-box model, 21, 31
Bolza Functional, 15
Bug reports, 10

C++ Model, 25, 31
Compatibility, 9
Compiler, 6, 8, 10
Constraint, 36
Contact, 10
Control, 29
Controller stand-alone, 58

DAE, 23, 31
Differential Equation, 13
  DifferentialEquation, 30
  Discretized Differential Equation, 32
DifferentialEquation, 30
DifferentialState, 29
Discrete time, 32
DiscretizedDifferentialEquation, 32
Disturbance, 29
dot, 31

END_ACADO, 33

FAQ, 10

Initialize, 37
Installation, 6
Integrator, 13, 19
  BDF, 14
  C++ Model, 25
  DAE, 23
  Jacobian, 22
  Model, 21
  ODE, 21
  Runge Kutta, 14
  Sensitivity Generation, 25

Jacobian, 22, 31

Lagrange Term, 35
linkMatlabDAE, 31
linkMatlabODE, 31
Linux, 7
LSQ Term, 35

Mac, 7
Make, 7
Makemex, 59
Matlab model, 21, 31
Matlab versions, 9
Mayer Term, 35
Measurements, 43
Mex, 7, 10
MexInput, 57
  MexInputMatrix, 57
  MexInputVector, 57
Model, 21
Model Based Feedback Control, 16
Model Predictive Control, 16
MODEL_INCLUDE, 25
MPC, 16, 47
Multi Stage OCP, 60

next, 32

Objective Function, 35
OCP, 34

ODE, 21, 31
Optimal Control Problem, 15, 34
    Constraint, 36
    Multi Stage, 60
    Objective Function, 35
Optimization Algorithm, 36
Outlook, 11

Parameter, 29
Parameter Estimation, 42
Parameter estimation, 15
Path constraint, 36
Paths, 8
Process disturbance, 49

Reference Trajectory, 49

Sensitivity Generation, 25
Simulation Environment, 47
State estimation, 15
Subject To, 36
Supported Matlab versions, 9

TODO, 11

Website, 10
Windows, 8

# Bibliography

[1] D. Ariens. Acado voor matlab, een softwareomgeving voor dynamische optimalisatie met toepassingen in de chemische technologie. Master's thesis, K.U.Leuven, 2010.

[2] D. Ariens, B. Houska, and H. Ferreau. Acado toolkit website. `http://www.acadotoolkit.org`.

[3] D. Ariens, B. Houska, H. Ferreau, and F. Logist. Acado: Toolkit for automatic control and dynamic optimization. `http://www.acadotoolkit.org/`.

[4] D. Ariens, B. Houska, H. Ferreau, and F. Logist. *ACADO for Matlab User's Manual*. Optimization in Engineering Center (OPTEC), 1.0beta edition, May 2010. `http://www.acadotoolkit.org/`.

[5] S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

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

[7] M. Diehl. *Script for Numerical Optimization Course B-KUL-H03E3A*. 2009. `http://homes.esat.kuleuven.be/~mdiehl/NUMOPT/numopt_hyper.pdf`.

[8] B. Houska and H. Ferreau. *ACADO Toolkit User's Manual*. Optimization in Engineering Center (OPTEC), 1.0beta edition, May 2009.

[9] B. Houska, H. Ferreau, and M. Diehl. Acado toolkit - an open-source framework for automatic control and dynamic optimization (accepted). *Optimal Control Applications & Methods*, 2010.

[10] J. Nocedal and S. Wright. *Numerical Optimization*. Springer-Verlag, 2000.