STU Institute of Information Engineering, Automation, and Mathematics,
Department of Information Engineering and Process Control,
Faculty of Chemical and Food Technology,
Slovak University of Technology in Bratislava,
Radlinského 9, 812 37 Bratislava, Slovak Republic.

MATLAB *DYN*amic *OPT*imisation Code

DYNOPT

M. Čižniar
M. Fikar
M. A. Latifi

USER'S GUIDE Version 5.0, July 16, 2020

Contact

M. Fikar Institute of Information Engineering, Automation, and Mathematics, Department of Information Engineering and Process Control, Faculty of Chemical and Food Technology, Slovak University of Technology in Bratislava, Radlinského 9, 812 37 Bratislava, Slovak Republic.
https://www.uiam.sk/~fikar, e-mail:miroslav.fikar@stuba.sk,
tel: +421 2 593 25 367, fax: +421 2 593 25 340.

Internet and Download

https://bitbucket.org/dynopt/dynopt_code
https://www.uiam.sk/assets/publication_info.php?id_pub=271&lang=en

Reference

M. Čižniar, M. Fikar, M. A. Latifi, MATLAB Dynamic Optimisation Code DYNOPT. User's Guide, Technical Report, KIRP FCHPT STU Bratislava, Slovak Republic, 2006.

Acknowledgments

The authors gratefully acknowledge the contribution of the Slovak Research and Development Agency under the project APVV-15-0007.

Contents

List of Figures

List of Tables

CHAPTER 1

Before You Begin

1.1 What is *dynopt*

dynopt is a set of MATLAB functions for determination of optimal control trajectory by given description of the process, the cost to be minimised, subject to equality and inequality constraints, using orthogonal collocation on finite elements method.

The actual optimal control problem is solved by complete parametrisation both the control and the state profile vector. That is, the original continuous control and state profiles are approximated by a sequence of linear combinations of some basis functions. It is assumed that the basis functions are known and optimised are the coefficients of their linear combinations. In addition, each segment of the control sequence is defined on a time interval whose length itself may also be subject to optimisation. Finally, a set of time independent parameters may influence the process model and can also be optimised.

It is assumed, that the optimised dynamic model may be described by a set of ordinary differential equations (ODE's) or differential-algebraic equations (DAE's).

1.2 What is New

Version 5

Version 5 does not bring any improvements in the algorithm but focuses on ease of implementation. In the default configuration, it uses automatic differentiation package Adigator [**?**] to construct Jacobians of process differential equations, cost function, and constraints. This has a consequence that the problem source files from version 4 or earlier are not compatible and have to be modified. On the other hand, it eases the implementation considerably and reduces errors in problem definition. However, manual specification of Jacobians is still possible if needed.

Incompatibility with Version 4

When files from older releases are to be reused, please keep in mind following items:

- Main file does not need to turn on gradients in NLP options.
- Files *process.m*, *objfun.m*, *confun.m* must not contain *case* statement – use *if/else* instead.
- File *process.m* contains only information for *flag* equal to 0 (differential euqations) and 5 (initial conditions) – no gradients. Mass matrix information was moved to main file.
- File *objfun.m* does not contain information about gradients.
- File *confun.m* does not contain information about gradients.
- Gradient information can be moved to newly defined files.

Version 4

Version 4 introduces several new properties of the package:

- three type of constraints can be defined in the same time: constraints in t_0, constraints over full time interval $[t_0, t_f]$, and constraints in t_f. Previously only one of them was possible.
- time independent parameters are introduced into the *process* function, *objfun* function and *confun*.

Version 4.2 enables definition of min/max constraints on state and control independently on each interval (see Section **??**).

Version 4.3 makes possible to use different NLP solvers (see Chapter **??** for example of use Ipopt and Chapter **??** for reference).

1.3 How to Use this Manual

This manual has four main parts:

Chapter 2 introduces implementation of orthogonal collocation on finite elements method into general optimisation problems.

Chapter 3 provides a tutorial for solving different optimisation problems.

Chapter 4 provides a detailed reference description of *dynopt* function. Reference descriptions include the function syntax, detailed information about arguments to the function, including relevant optimisation options parameters.

Chapter 5 provides some more examples solved by *dynopt*, their definitions and solutions.

1.4 Code, Documentation, and License

Both source code of the toolbox and of this documentation are available online.

Code:
https://github.com/miroslavfikar/dynopt_code

Documentation:
https://github.com/miroslavfikar/dynopt_doc

dynopt is free of charge to use and is openly distributed, but note that (YALMIP license)

- Copyright is owned by Miroslav Fikar.
- dynopt must be referenced when used in a published work (give me some credit for saving your valuable time!)
- dynopt, or forks or versions of dynopt, may not be re-distributed as a part of a commercial product unless agreed upon with the copyright owner (if you make money from dynopt, let me in first!)
- dynopt 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 (if your satellite crash or you fail your PhD due to a bug in dynopt, your loss!).
- Forks or versions of dynopt must include, and follow, this license in any distribution.

1.5 Installation

1.5.1 Manual Installation

Download *dynopt* from:
https://github.com/miroslavfikar/dynopt_code/archive/master.zip.

dynopt needs *fminsdp* toolbox [**?**] and *adigator* toolbox [**?**] to be installed. For convenience, both are packed within the zip file, but feel free to get them from their original locations. To install them, follow directions to add respective directories to MATLAB path.

Next, install *dynopt*: add the directory *dynoptim* to MATLAB path.

All packages can be installed by getting into *dynopt* directory in MATLAB and running the following commands:

```
currpath = pwd;
addpath([currpath,filesep,'dynoptim']);
addpath([currpath,filesep,'adigator']);
addpath([currpath,filesep,'adigator',filesep,'lib']);
addpath([currpath,filesep,'adigator',filesep,'lib',filesep,'cadaUtils']);
addpath([currpath,filesep,'adigator',filesep,'util']);
addpath([currpath,filesep,'fminsdp']);
addpath([currpath,filesep,'fminsdp',filesep,'interfaces']);
addpath([currpath,filesep,'fminsdp',filesep,'utilities']);
```

and then saving the path for future.

1.5.2 Installation using tbxmanager

tbxmanager (https://www.tbxmanager.com) is a tool to install and manage several third party optimisation toolboxes for MATLAB. The complete list of available packages is disponible at its home page.

Install *tbxmanager* according to directions at its home page. Then, *dynopt* is installed (or upgraded) as:

```
tbxmanager install dynopt
tbxmanager update dynopt
```

1.5.3 Optimisation Toolbox

dynopt is a set of functions that extend the capability of the MATLAB Optimization Toolbox. That means, that for using *dynopt* this toolbox has to be provided. To determine if the Optimization Toolbox is installed on your system, type this command at the MATLAB prompt:

```
ver
```

After entering this command, MATLAB displays information about the version of MATLAB you are running, including a list of all toolboxes installed on your system and their version numbers. If the Optimization Toolbox is not installed, check the Installation Guide for instructions on how to install it.

dynopt has been developed and tested since MATLAB 6.5 (R13). The results in this guide are obtained with MATLAB 2020a (Linux 64b) using solvers *fmincon* and *ipopt*. Please, note that it is quite usual that results obtained and convergence criteria achieved with different versions of MATLAB or its toolboxes can/will produce slightly different (better, worse) results.

CHAPTER 2

Dynamic Optimisation

This chapter deals with dynamic optimisation in general. The chapter starts with several dynamic optimisation problem definitions. Finally, this chapter ends with the NLP problem formulation.

2.1 Optimisation Problem Statement

The objective of dynamic optimisation is to determine, in open loop control, a set of time dependent decision variables (pressure, temperature, flow rate, current, heat duty, ...) that optimise a given performance index (or cost functional or optimisation criterion)(cost, time, energy, selectivity, ...) subject to specified constraints (safety, environmental and operating constraints). Optimal control refers to the determination of the best time-varying profiles in closed loop control.

2.1.1 Cost Functional

The performance index (cost functional or optimisation criterion) can in general be written in one of three forms as follows:

Bolza form

$$\mathcal{J}(\boldsymbol{u}(t), \boldsymbol{p}, t_f) = \mathcal{G}(\boldsymbol{x}(t_f), \boldsymbol{p}, t_f) + \int_{t_0}^{t_f} \mathcal{F}(\boldsymbol{x}(t), \boldsymbol{u}(t), \boldsymbol{p}, t)\mathrm{d}t \tag{2.1}$$

Lagrange form

$$\mathcal{J}(\boldsymbol{u}(t), \boldsymbol{p}, t_f) = \int_{t_0}^{t_f} \mathcal{F}(\boldsymbol{x}(t), \boldsymbol{u}(t), \boldsymbol{p}, t)\mathrm{d}t \tag{2.2}$$

Mayer form

$$\mathcal{J}(\boldsymbol{u}(t), \boldsymbol{p}, t_f) = \mathcal{G}(\boldsymbol{x}(t_f), \boldsymbol{p}, t_f) \tag{2.3}$$

where

$\mathcal{J}(\cdot)$ – optimisation criterion,

$\mathcal{G}(\cdot)$ – component of objective function evaluated at final conditions,

$\int_{t_0}^{t_f} \mathcal{F}(\cdot)\mathrm{d}t$ – component of the objective function evaluated over a period of time,

$\boldsymbol{x}(t)$ – vector of state variables,

$\boldsymbol{u}(t)$ – vector of control variables,

$\boldsymbol{p}$ – vector of time independent parameters,

t_0, t_f – initial and final times.

Note that all three forms are interchangeable and can be derived one from another. In the sequel, Mayer form will be used.

2.1.2 Process Model Equations

The behaviour of many of processes can in general be described either by a set of ordinary differential equations (ODE's) or by a set of differential-algebraic equations (DAE's) as follows:

$$\boldsymbol{M}\dot{\boldsymbol{x}}(t) = \boldsymbol{f}(\boldsymbol{x}(t), \boldsymbol{u}(t), \boldsymbol{p}, t), \quad \boldsymbol{x}(t_0) = \boldsymbol{x}_0 \quad \text{over} \quad t_0 \leq t \leq t_f \tag{2.4}$$

with initial condition for states $\boldsymbol{x}_0$ which may also be a function of some time independent parameters. Here $\boldsymbol{M}$ is a constant mass matrix. This ODE or DAE system forms equality constraint in optimal control problem.

2.1.3 Constraints

Constraints to be accounted for typically include equality and inequality infinite dimensional, interior-point, and terminal-point constraints [**?**]. Moreover, they may be written in the following canonical form similar to the cost form (**??**):

$$\mathcal{J}_i(\boldsymbol{u}(t), \boldsymbol{p}, t_i) = \mathcal{G}_i(\boldsymbol{x}(t_i), \boldsymbol{p}, t_i) \tag{2.5}$$

where $t_i \leq t_f$, $i = 1, \ldots, nc$, and nc is the number of constraints.

2.2 Optimal Control Problem Solutions

There are several approaches that can solve optimal control problems. These can be divided into analytical methods that have been used originally and numerical methods preferred nowadays. In this work only numerical methods are considered.

The numerical methods used for the solution of dynamic optimisation problems can then be grouped into two categories: indirect and direct methods. In this work only direct methods are considered. In this category, there are two strategies: sequential method and simultaneous method. The sequential strategy, often called control vector parameterisation (CVP), consists in an approximation of the control trajectory by a function of only few parameters and leaving the state equations in the form of the original ODE/DAE system [**?**]. In the simultaneous strategy often called total discretisation method, both the control and

state variables are discretised using polynomials (e.g., Lagrange polynomials) of which the coefficients become the decision variables in a much larger NLP problem [**?**]. Implementation of this method is subject of this work.

Next section reviews the general NLP formulation for optimal control problems using orthogonal collocation on finite elements method.

2.3 NLP Formulation Problem

As mentioned before, the optimal control problem will be solved by complete parametrisation of both the control and the state profile vector [**? ?**]. That means, that the control and state profiles are approximated by a linear combination of some basis functions. It is expected here, that the basis functions are known so only the coefficients of linear combination of these fundamentals have to be optimised. In addition, each control sequence segment is defined on time interval, which length itself can be the optimised variable. Finally, a set of time independent parameters may influence the process model and can also be optimised. As mentioned in section sec:pme, it is supposed that the optimised dynamic model can be described either by an ODE system or by an DAE system.

Consider the following general control problem for $t \in [t_0, t_f]$:

$$\min_{\boldsymbol{u}(t),\boldsymbol{p},t_f} \{\mathcal{G}(\boldsymbol{x}(t_f), \boldsymbol{p}, t_f)\} \tag{2.6}$$

such that

$$\begin{gathered}
\boldsymbol{M}\dot{\boldsymbol{x}}(t) = \boldsymbol{f}(\boldsymbol{x}(t), \boldsymbol{u}(t), \boldsymbol{p}, t), \quad \boldsymbol{x}(t_0) = \boldsymbol{x}_0(\boldsymbol{p}) \\
\boldsymbol{h}(\boldsymbol{x}(t), \boldsymbol{u}(t), \boldsymbol{p}, t) = \boldsymbol{0} \\
\boldsymbol{g}(\boldsymbol{x}(t), \boldsymbol{u}(t), \boldsymbol{p}, t) \leq \boldsymbol{0} \\
\boldsymbol{x}(t)^L \leq \boldsymbol{x}(t) \leq \boldsymbol{x}(t)^U \\
\boldsymbol{u}(t)^L \leq \boldsymbol{u}(t) \leq \boldsymbol{u}(t)^U \\
\boldsymbol{p}^L \leq \boldsymbol{p} \leq \boldsymbol{p}^U
\end{gathered}$$

with following nomenclature:

$\boldsymbol{h}(\cdot)$ – equality design constraint vector,

$\boldsymbol{g}(\cdot)$ – inequality design constraint vector,

$\boldsymbol{x}(t)^L, \boldsymbol{x}(t)^U$ – state profile bounds,

$\boldsymbol{u}(t)^L, \boldsymbol{u}(t)^U$ – control profile bounds,

$\boldsymbol{p}^L, \boldsymbol{p}^U$ – parameter bounds.

In order to derive the NLP problem the differential equations are converted into algebraic equations using collocation on finite elements. Residual equations are then formed and solved as a set of algebraic equations. These residuals are evaluated at the shifted roots of Legendre polynomials. The procedure is then following: Consider the initial-value problem over a finite element i with time $t \in [\zeta_i, \zeta_{i+1}]$:

$$\boldsymbol{M}\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}(t), \boldsymbol{u}(t), \boldsymbol{p}, t) \quad t \in [t_0, t_f] \tag{2.7}$$

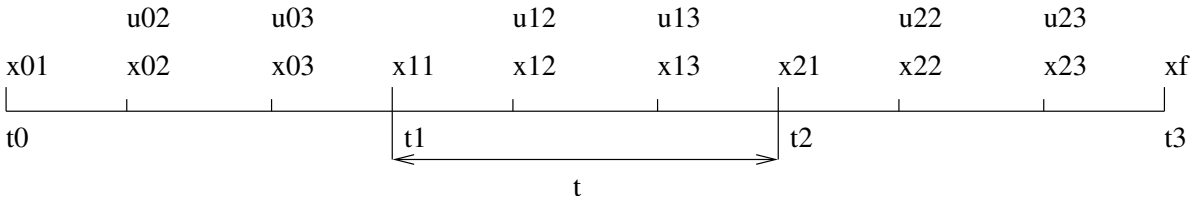

Figure 2.1: Collocation method on finite elements for state profiles, control profiles and element lengths ($K_x = K_u = 2$)

The solution is approximated by Lagrange polynomials over element i, $\zeta_i \leq t \leq \zeta_{i+1}$ as follows:

$$\boldsymbol{x}_{K_x}(t) = \sum_{j=0}^{K_x} \boldsymbol{x}_{ij}\phi_j(t); \quad \phi_j(t) = \prod_{k=0,j}^{K_x} \frac{(t - t_{ik})}{(t_{ij} - t_{ik})} \tag{2.8}$$
$$\text{in element } i \qquad i = 1, \ldots, \text{NE}$$

$$\boldsymbol{u}_{K_u}(t) = \sum_{j=1}^{K_u} \boldsymbol{u}_{ij}\theta_j(t); \quad \theta_j(t) = \prod_{k=1,j}^{K_u} \frac{(t - t_{ik})}{(t_{ij} - t_{ik})} \tag{2.9}$$
$$\text{in element } i \qquad i = 1, \ldots, \text{NE}$$

Here $k = 0, j$ means k starting form 0 and $k \neq j$, NE is the number of elements. Also $\boldsymbol{x}_{K_x}(t)$ is a $(K_x + 1)$th degree piecewise polynomial and $\boldsymbol{u}_{K_u}(t)$ is piecewise polynomial of order K_u. The polynomial approximating the state $\boldsymbol{x}$ takes into account the initial conditions of $\boldsymbol{x}(t)$ for each element i. Also, the Lagrange polynomial has the desirable property that (for $\boldsymbol{x}_{K_x}(t)$, for example):

$$\boldsymbol{x}_{K_x}(t_{ij}) = \boldsymbol{x}_{ij} \tag{2.10}$$

which is due to the Lagrange condition $\phi_k(t_j) = \delta_{kj}$, where δ_{kj} is the Kronecker delta. This polynomial form allows the direct bounding of the states and controls, e.g., path constraints can be imposed on the problem formulation.

Using $K = K_x = K_u$ point orthogonal collocation on finite elements as shown in Fig. **??**, and by defining the basis functions, so that they are normalised over the each element $\Delta\zeta_i(\tau \in [0, 1])$, one can write the residual equation as follows:

$$\Delta\zeta_i \boldsymbol{r}(t_{ik}) = \boldsymbol{M} \sum_{j=0}^{K_x} \boldsymbol{x}_{ij}\dot{\phi}_j(\tau_k) - \Delta\zeta_i \boldsymbol{f}(t_{ik}, \boldsymbol{x}_{ik}, \boldsymbol{u}_{ik}, \boldsymbol{p}) \tag{2.11}$$
$$i = 1, \ldots, \text{NE}, \quad j = 0, \ldots, K_x, \quad k = 1, \ldots, K_x$$

where $\dot{\phi}_j(\tau_k) = \mathrm{d}t\phi_j/\mathrm{d}t\tau$, and together with $\phi_j(\tau)$, $\theta_j(\tau)$ terms (basis functions), they are calculated beforehand, since they depend only on the Legendre root locations. Note that $t_{ik} = \zeta_i + \Delta\zeta_i\tau_k$. This form is convenient to work with when the element lengths are included as decision variables. The element lengths are also used to find possible points of discontinuity for the control profiles and to insure that the integration accuracy is within a numerical tolerance. Additionally, the continuity of the states is enforced at element endpoints (interior knots $\zeta_i, i = 2, ..., \text{NE}$), but it is allowed that the control profiles to have discontinuities at these endpoints. Here

$$\boldsymbol{x}^i_{K_x}(\zeta_i) = \boldsymbol{x}^{i-1}_{K_x}(\zeta_i) \tag{2.12}$$
$$i = 2, \ldots, \text{NE}$$

or

$$\boldsymbol{x}_{i0} = \sum_{j=0}^{K_x} \boldsymbol{x}_{i-1,j}\phi_j(\tau = 1) \tag{2.13}$$
$$i = 2, \ldots, \mathrm{NE}, \quad j = 0, \ldots, K_x$$

These equations extrapolate the polynomial $\boldsymbol{x}_{K_x}^{i-1}(t)$ to the endpoints of its element and provide an accurate initial conditions for the next element and polynomial $\boldsymbol{x}_{K_x}^{i}(t)$.

At this point a few additional comments concerning construction of the control profile polynomials must be made. Note that these polynomials use only K_u coefficients per element and are of lower order than the state polynomials. As a result these profiles are constrained or bounded only at collocation points. The constraints of the control profile are carried out by bounding the values of each control polynomial at both ends of the element. This can be done by writing the equations:

$$\boldsymbol{u}_i^L \leq \boldsymbol{u}_{K_u}^i(\zeta_i) \leq \boldsymbol{u}_i^U \quad i = 1, \ldots, \mathrm{NE} \tag{2.14}$$
$$\boldsymbol{u}_i^L \leq \boldsymbol{u}_{K_u}^i(\zeta_{i+1}) \leq \boldsymbol{u}_i^U \quad i = 1, \ldots, \mathrm{NE} \tag{2.15}$$

Note that since the polynomial coefficients of the control exist only at collocation points, enforcement of these bounds can be done by extrapolating the polynomial to the endpoints of the element. This is easily done by using:

$$\boldsymbol{u}_{K_u}^i(\zeta_i) = \sum_{j=1}^{K_u} \boldsymbol{u}_{ij}\theta_j(\tau = 0), \quad i = 1, \ldots, \mathrm{NE} \tag{2.16}$$

and

$$\boldsymbol{u}_{K_u}^i(\zeta_{i+1}) = \sum_{j=1}^{K_u} \boldsymbol{u}_{ij}\theta_j(\tau = 1), \quad i = 1, \ldots, \mathrm{NE} \tag{2.17}$$

Adding these constraints affects the shape of the final control profile and the net effect of these constraints is to keep the endpoint values of the control profile from varying widely outside their ranges $[\boldsymbol{u}_i^L, \boldsymbol{u}_i^U]$.

The NLP formulation consists of the ODE model (**??**) discretised on finite elements, continuity equation for state variables, and any other equality and inequality constraints that may be required. It is given by

$$\min_{\boldsymbol{x}_{ij}, \boldsymbol{u}_{ij}, \boldsymbol{p}, \Delta\zeta_i} \Big[\mathcal{G}(\boldsymbol{x}_f, \boldsymbol{p}, t_f)\Big] \tag{2.18}$$

such that

$$\begin{gathered}
\boldsymbol{x}_{10} - \boldsymbol{x}_0(\boldsymbol{p}) = \mathbf{0} \\
\boldsymbol{r}(t_{ik}) = \mathbf{0} \quad i = 1, \ldots, \mathrm{NE} \quad k = 1, \ldots, K_x \\
\boldsymbol{x}_{i0} - \boldsymbol{x}_{K_x}^{i-1}(\zeta_i) = \mathbf{0} \quad i = 2, \ldots, \mathrm{NE} \\
\boldsymbol{x}_f - \boldsymbol{x}_{K_x}^{NE}(\zeta_{NE+1}) = \mathbf{0} \\
\boldsymbol{u}_i^L \leq \boldsymbol{u}_{K_u}^i(\zeta_i) \leq \boldsymbol{u}_i^U \quad i = 1, \ldots, \mathrm{NE} \\
\boldsymbol{u}_i^L \leq \boldsymbol{u}_{K_u}^i(\zeta_{i+1}) \leq \boldsymbol{u}_i^U \quad i = 1, \ldots, \mathrm{NE} \\
\boldsymbol{u}_{ij}^L \leq \boldsymbol{u}_{K_u}(\tau_j) \leq \boldsymbol{u}_{ij}^U \quad i = 1, \ldots, \mathrm{NE} \quad j = 1, \ldots, K_u \\
\boldsymbol{x}_{ij}^L \leq \boldsymbol{x}_{K_x}(\tau_j) \leq \boldsymbol{x}_{ij}^U \quad i = 1, \ldots, \mathrm{NE} \quad j = 0, \ldots, K_x \\
\Delta\zeta_i^L \leq \Delta\zeta_i \leq \Delta\zeta_i^U \quad i = 1, \ldots, \mathrm{NE} \\
\boldsymbol{p}^L \leq \boldsymbol{p} \leq \boldsymbol{p}^U \\
\sum_{i=1}^{NE} \Delta\zeta_i = t_f \\
\boldsymbol{h}(t_{ij}, \boldsymbol{x}_{ij}, \boldsymbol{u}_{ij}, \boldsymbol{p}) = \mathbf{0} \\
\boldsymbol{g}(t_{ij}\boldsymbol{x}_{ij}, \boldsymbol{u}_{ij}, \boldsymbol{p}) \leq \mathbf{0}
\end{gathered}$$

where i refers to the time-interval, j, k refers to the collocation point, $\Delta\zeta_i$ represents finite-element length of each time-interval $i = 1, \ldots, \mathrm{NE}$, $\boldsymbol{x}_f = \boldsymbol{x}(t_f)$, and $\boldsymbol{x}_{ij}, \boldsymbol{u}_{ij}$ are the collocation coefficients for the state and control profiles. Problem (**??**) can be now solved by any large-scale nonlinear programming solver.

To solve this problem within MATLAB, default settings of *dynopt* use the Optimization Toolbox and its function *fmincon*. This can minimise/maximise given objective function with respect to nonlinear equality and inequality constraints. In order to use this function it was necessary to create and program series of additional functions. These additional functions together with *fmincon* are formed within *dynopt* which is simple for user to employ. This function is presented in next chapter.

Alternate nonlinear programming solvers can be used thanks to interface *fminsdp*. This version of *dynopt* was also tested with Ipopt [**?**].

CHAPTER 3

Tutorial

This chapter discusses the *dynopt* application. It shows, that *dynopt* is suitable for a quite large variety of problems ranging from simple unconstrained problem to inequality state path constraint problems described either by ODE's or DAE's. As mentioned in the title of this chapter, it is a step-by-step tutorial. It shows the user how to define his problem into *dynopt* by filling the input argument functions *process*, *objfun*, *confun*.

3.1 ODE systems

3.1.1 Unconstrained Problem

Consider a simple integrator with LQ cost function to be optimised [**?** **?**] (MATLAB code is located at *examples/problem1a*):

$$\dot{x}_1 = u, \quad x_1(0) = 1 \tag{3.1}$$

$$\dot{x}_2 = x_1^2 + u^2, \quad x_2(0) = 0 \tag{3.2}$$

The cost function is given in the Mayer form:

$$\min_{\boldsymbol{u}(t)} \mathcal{J} = x_2(t_f) \tag{3.3}$$

with $x_1(t),\ x_2(t)$ as states and $u(t)$ as control, such that $t_f = 1$.

Function *process*, *objfun* definitions

Problem (**??**) is described by two differential equations which together with initial values of state variables should be defined in *process*.

Step1: Write an M-file *process.m*

```
function sys = process(t,x,flag,u,p)

  if flag == 0 % ODE system
    sys = [u(1);
   x(1)^2 + u(1)^2];
  elseif flag == 5  % initial conditions
    sys = [1; 0];
  else
    sys = [];
  end
end
```

Thus, *process.m* contains definition of right sides of differential equations (`flag` = 0) and initial conditions (`flag` = 5). These are evaluated at time `t`, with inputs `x` - scalar/vector of state variable(s), `u` - scalar/vector of control variable(s), `p` - scalar/vector of time independent parameters.

As the performance index is given in Mayer form, *dynopt* optimises it at final conditions, thus the input arguments of *objfun* are evaluated at corresponding final time : `t` - scalar value t_f. *objfun* should be defined as follows:

Step2: Write an M-file *objfun.m*

```
function fun = objfun(t,x,u,p)
  fun = x(2);
end
```

After the problem has been defined in the functions, user has to invoke the *dynopt* function by writing an M-file *problem1a.m* as follows:

Step3: Invoke *dynopt*

```
clear; close all; clc;
options = sdpoptionset('LargeScale','on','Display','iter','TolFun',1e-7,...
                       'TolCon',1e-7,'TolX',1e-7,...
                       'MaxFunEvals',1e6,'MaxIter',4000,...
                       'Algorithm','sqp','NLPsolver','fmincon');

optimparam.optvar = 3;
optimparam.objtype = [];
optimparam.ncolx = 3;
optimparam.ncolu = 2;
optimparam.li = ones(3,1)*(1/3);
optimparam.tf = 1;
optimparam.ui = zeros(1,3);
optimparam.par = [];
optimparam.bdu = [];
optimparam.bdx = [];
optimparam.bdp = [];
optimparam.objfun  = @objfun;
optimparam.confun  = [];
```

```
optimparam.process = @process;
optimparam.options = options;

[optimout,optimparam] = dynopt(optimparam);
```

In this case the variables: `t`, `u` were chosen as decision variables, so the parameter `optimparam.optvar` was set to 3. As the objective is to minimise the functional in Mayer form the parameter `optimparam.objtype` was left an empty matrix. Moreover 3 collocation points for state variables (`optimparam.ncolx`), 2 collocation points for control variables (`optimparam.ncolu`), 3 time intervals with the same initial lengths (`optimparam.li`) equal to 1/3 were chosen. Final time $t_f = 1$ was given by the problem definition (`optimparam.tf`), the control variable initial values (`optimparam.ui`) were set to 0 for each time interval. As can be seen from the problem definition (**??**) no parameters (`optimparam.par`), no bounds for the control variables (`optimparam.bdu`), the state variables (`optimparam.bdx`), and the parameters (`optimparam.bdp`) are needed, so the values of this parameters have been left an empty matrix. As mentioned before, this problem is unconstrained, so parameter `optimparam.confun` was set to [ ].

Optimisation parameters for NLP solver are defined by `optimparam.options`.

The results returned by *dynopt* in `optimout` structure contain the vector of times `t`, the vector of optimal control profile `u`. They are ready to be plotted.
The objective function at the optimal solution [`t`,`u`] is returned after 1554 iterations in above mentioned output structure `optimout` as parameter `fval`:

```
optimout.fval = 0.7616
```

The parameter `exitflag` tells if the algorithm converged. An `exitflag` > 0 means a local minimum was found:

```
optimout.exitflag = 1
```

More details about the optimisation are given by the `optimout.output` structure. In this example, the default selection of the large-scale algorithm has been turned off, so the medium-scale algorithm is used. Also all termination tolerances have been changed. For more information about `options` and *dynopt* input and output arguments, see Chapter **??**.

The user may want to plot also the state profiles but without integrating the *process* with respect to the optimal control profile in `optimout.u`. It is possible to use an additional function *profiles* for this reason as follows:

```
[tplot,uplot,xplot] = profiles(optimout,optimparam,ntimes);
```

where `ntimes` represents the density of the points plotted per interval. Note that profiles are represented by collocation polynomials and not as a solution of the original ODE/DAE system. Therefore, there might be slight differences with regard to the original problem. For more precise plots, integrate the original problem with optimal control trajectories.

Graphical representation of the problem (**??**) solution is shown in Figs. **??** and **??**.

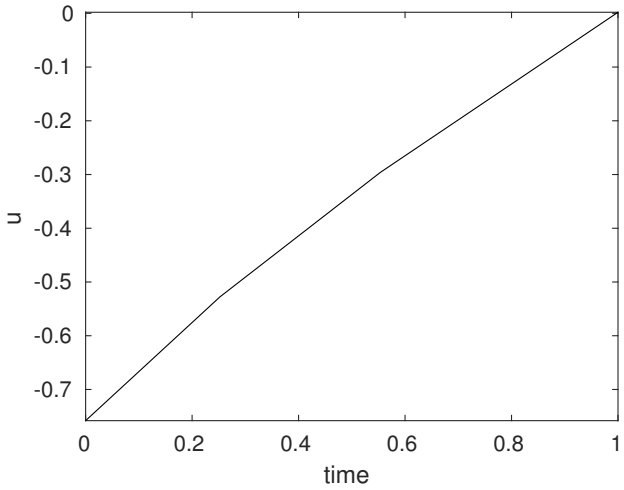

Figure 3.1: Control profile for unconstrained problem

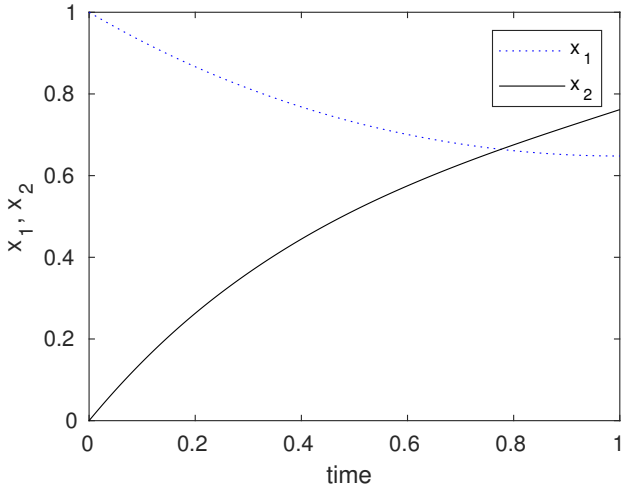

Figure 3.2: State profiles for unconstrained problem

3.1.2 Constrained Problem with Gradients

A process described by the following system of 2 ODE's [**?** **?**] (MATLAB code for this example is located at *examples/problem1b*):

$$\dot{x}_1 = u, \quad x_1(0) = 1 \tag{3.4}$$

$$\dot{x}_2 = x_1^2 + u^2, \quad x_2(0) = 0 \tag{3.5}$$

is to be optimised for $u(t)$ with the cost function:

$$\min_{\boldsymbol{u}(t)} \mathcal{J} = x_2(t_f) \tag{3.6}$$

subject to the constraint:

$$x_1(1) = 1 \tag{3.7}$$

with $x_1(t)$, $x_2(t)$ as states, $u(t)$ as control, such that $t_f = 1$.

Problem (**??**) is similar to problem (**??**), it differs in constraint of state variable x_1 at final time $t_f = 1$. This example will be solved by supplying analytical gradients manually. Compared to automatic differentiation using Adigator, user has complete control over the code and can implement it efficiently and faster. On the other side, this may be error-prone.

Function *process*, *objfun*, *confun* definitions

As mentioned before the problem (**??**) is described by the same differential equations as problem (**??**). As we decided to supply analytical gradients, they should be defined for all the user supplied functions: *process*, *objfun*, *confun*. The form of the gradients will be explained on the function *process* and is valid for all above mentioned user functions.

Step1a: Write an M-file *process.m*

```
function sys = process(t,x,flag,u,p)

  if flag == 0 % ODE system
    sys = [u;x(1)^2+u^2];
```

```
  elseif flag == 5  % initial conditions
    sys = [1;0];
  else
    sys = [];
  end
end
```

Step1b: Provide gradients in file *processd.m*

```
function sys = processd(t,x,flag,u,p)

  sys = [];
  if flag == 1 % df/dx
    sys = [0 2*x(1);0 0];
  elseif flag == 2 % df/du
    sys = [1 2*u];
  end
end
```

Definition of gradients results from problem definition (**??**). As the problem consists of one control variable u and two states variables x_1, x_2 just the gradients with respect to this variables have to be supplied by filling the appropriate flag.
`sys` in case 1 contains the partial derivatives of the *process* function, defined as `sys` in case 0, with respect to each of the elements in `x`:

$$\texttt{sys} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_2}{\partial x_1} \\ \frac{\partial f_1}{\partial x_2} & \frac{\partial f_2}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 0 & 2x_1 \\ 0 & 0 \end{bmatrix}$$

`sys` in case 2 contains the partial derivatives of the *process* function, defined as `sys` in case 0, with respect to each of the elements in `u`:

$$\texttt{sys} = \begin{bmatrix} \frac{\partial f_1}{\partial u} & \frac{\partial f_2}{\partial u} \end{bmatrix} = \begin{bmatrix} 1 & 2u \end{bmatrix}$$

If needed, the gradients with respect to other defined variables (`t`, `p`) are filled similarly. For more information about *process*, *processd* definition, and their input and output arguments see chapter **??**.

As mentioned before user has also to supply the gradients to the objective function *objfun* as follows:

Step2a: Write an M-file *objfun.m*

```
function fun = objfun(t,x,u,p)
    fun = x(2);
end
```

Step2b: Provide gradients in file *objfund.m*

```
function Df = objfund(t,x,u,p)
% gradients of the objective function
  Df.t = [];    % dJ/dt
  Df.x = [0;1]; % dJ/dx
  Df.u = [];    % dJ/du
  Df.p = [];    % dJ/dp
```

Here they are written in the structure `Df` containing variables `t`, `u`, `x`, and `p` and representing the gradients with respect to the appropriate variable. Just the variables used in problem are filled by user. Unused variables are set to be an empty matrix. For more information about *objfun*, *objfund* definition, and their input and output arguments see chapter **??**.

dynopt optimises a given performance index, subject to the constraints defined at the beginning $t = t_0$ (`flag` $= 0$), over the full time interval $t \in [t_0, t_f]$ (`flag` $= 1$), and at the end $t = t_f$ (`flag` $= 2$). Thus the input arguments of *confun* are the same as of *process* but it is necessary to tell *dynopt* by defining the constraints and their gradients with respect to the appropriate variables in the corresponding flag in which time should they be evaluated. How the gradients have to seem like, was explained before. *confun* should be defined as follows:

Step3a: Write an M-file *confun.m*

```
function [c, ceq] = confun(t,x,flag,u,p)

  if flag == 0                % constraints in t0
    c   = [];
    ceq = [];
  elseif flag == 1            % constraints over interval [t0,tf]
    c   = [];
    ceq = [];
  elseif flag == 2            % constraints in tf
    c = [];
    ceq = [x(1) - 1];
  end
end
```

Step3b: Provide gradients in file *confund.m*

```
function [Dc,Dceq] = confund(t,x,flag,u,p)

  Dc.t = [];
  Dc.x = [];
  Dc.u = [];
  Dc.p = [];
  Dceq.t = [];
  Dceq.x = [];
  Dceq.u = [];
  Dceq.p = [];
if flag ==2 % constraints in tf
  Dceq.x = [1;0];
end
```

Here the gradients are written into the structures `Dc`, `Dceq` similar to those, described in *objfun*. For more information about *confun*, *confund* definition, and their input and output arguments see chapter **??**.

Since you are providing the gradients of the process in *processd.m*, of the objective function in *objfund.m* and the gradients of the constraints in *confund.m*, you must tell *dynopt* that these M-files contain this additional information. Use *adoptionset* to turn the gradients on

```
optimparam.adoptions = adoptionset('jacuser',true);
```

If this option is deactivated, files *processd.m*, *objfund.m*, *confund.m* are not used and automatic gradients are invoked instead. See chapter **??** for more information on *adoptionset* and its parameters.

After the problem has been defined in the functions, user has to invoke the *dynopt* function by writing an M-file *problem1b.m* as follows :

Step4: Invoke *dynopt*

```
clear; close all; clc;
options = sdpoptionset('LargeScale','off','Display','iter','TolFun',1e-7,...
                       'TolCon',1e-6,'TolX',1e-7,...
                       'MaxFunEvals',1e6,'MaxIter',40000,...
                       'Algorithm','interior-point','NLPsolver','fmincon',...
                       'DerivativeCheck','on');

optimparam.optvar = 3;
optimparam.objtype = [];
optimparam.ncolx = 6;
optimparam.ncolu = 2;
optimparam.li = ones(2,1)*(1/2);
optimparam.tf = 1;
optimparam.ui = zeros(1,2);
optimparam.par = [];
optimparam.bdu = [];
optimparam.bdx = [0 1;0 1];
optimparam.bdp = [];
optimparam.objfun  = @objfun;
optimparam.confun  = @confun;
optimparam.process = @process;
optimparam.options = options;
optimparam.adoptions = adoptionset('jacuser',true);

[optimout,optimparam] = dynopt(optimparam);
[tplot,uplot,xplot]   = profiles(optimout,optimparam,50);
```

Note the parameter *DerivativeCheck='on'*. As gradients are implemented manually, it is wise to check them for correctness.

As this problem differs from the problem (**??**) in the constraint applied in final time $t_f = 1$, the input parameter `optimparam.confun` is set to the constraint function name

@*confun*. Next, 3 collocation points for state variables, 2 intervals with the same initial lengths of intervals equal to 1/2 have been chosen. Other parameters are as same as in problem (**??**).

The optimal solution is shown in Figs. **??** and **??**. The value of the objective function at this solution is 0.9242 after 37 iterations and with `exitflag` equal to 2.

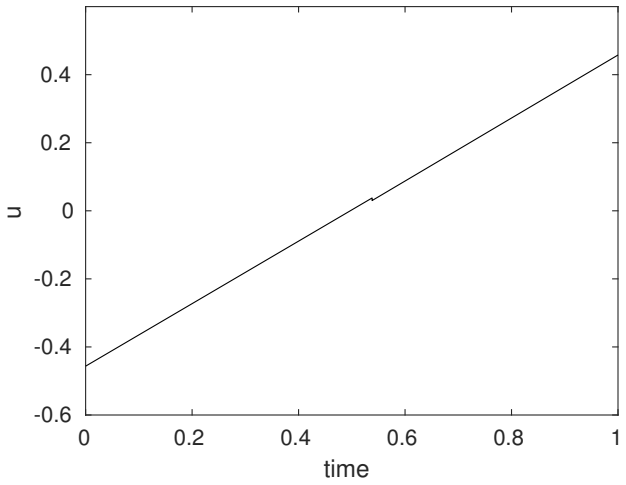

Figure 3.3: Control profile for constrained problem with gradients

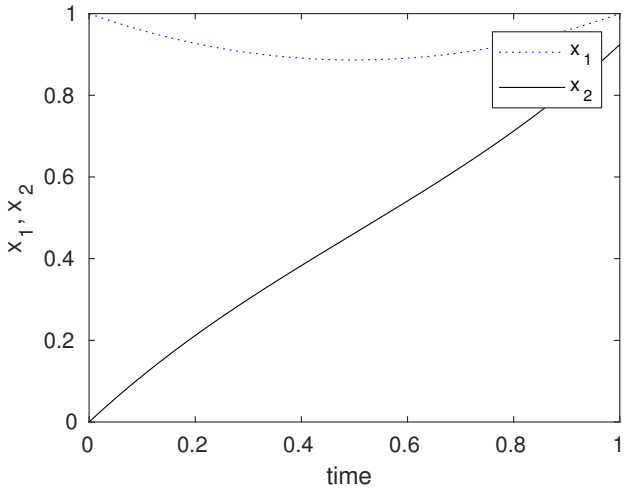

Figure 3.4: State profiles for constrained problem with gradients

Alternative NLP solver Ipopt can be invoked by setting the parameter

```
NLPsolver = 'ipopt'
```

in the main file. The optimum is found in 1369 iterations. More about alternate NLP solvers can be found in Chapter **??**.

3.1.3 Unconstrained Problem with Bounds

Following mathematical problem [**?** **?**] with system of four ODE's (MATLAB code for this example is located at *examples/problem2*):

$$\dot{x}_1 = x_2, \quad x_1(0) = 0 \tag{3.8}$$

$$\dot{x}_2 = -x_3 u + 16t - 8, \quad x_2(0) = -1 \tag{3.9}$$

$$\dot{x}_3 = u, \quad x_3(0) = -\sqrt{5} \tag{3.10}$$

$$\dot{x}_4 = x_1^2 + x_2^2 + 0.0005(x_2 + 16t - 8 - 0.1 x_3 u^2)^2, \quad x_4(0) = 0 \tag{3.11}$$

is to be optimised for $-4 \leq u(t) \leq 10$ with the cost function:

$$\min_{\boldsymbol{u}(t)} \mathcal{J} = x_4(t_f) \tag{3.12}$$

with $x_1(t) - x_4(t)$ as states, $u(t)$ as control, such that $t_f = 1$.

Function *process*, *objfun*, *confun* definitions

Step1: Write an M-file *process.m*

```
function sys = process(t,x,flag,u,p)

  if flag == 0 % ODE system
    sys = [x(2);
   -x(3)*u(1)+16*t-8;
    u;
    x(1)^2+x(2)^2+0.0005*(x(2)+16*t-8-0.1*x(3)*u(1)^2)^2];
  elseif flag == 5  % initial conditions
    sys = [0;-1;-sqrt(5);0];
  else
    sys = [];
  end
end
```

Step2: Write an M-file *objfun*

```
function fun = objfun(t,x,u,p)
    fun = x(4);
end
```

Step3: Invoke *dynopt* writing an M-file *problem2.m* as follows:

```
clear; close all; clc;
options = optimset('LargeScale','off','Display','iter');
options = optimset(options,'MaxFunEvals',1e5);
options = optimset(options,'MaxIter',1e4);
%options = optimset(options,'MaxIter',22);
options = optimset (options,'TolFun',1e-7);
options = optimset (options,'TolCon',1e-7);
options = optimset (options,'TolX',1e-7);
options = optimset(options,'Algorithm','sqp'); %2010a
options = optimset(options,'Algorithm','interior-point'); %2010a

%options.NLPsolver='ipopt';

optimparam.optvar = 3;
optimparam.objtype = [];
optimparam.ncolx = 6;
optimparam.ncolu = 2;
optimparam.li = ones(4,1)*(1/4);
optimparam.tf = 1;
optimparam.ui = ones(1,4)*7;
optimparam.par = [];
optimparam.bdu = [-4 10];
optimparam.bdx = [];
optimparam.bdp =[];
optimparam.objfun = @objfun;
```

```
optimparam.confun = [];
optimparam.process = @process;
optimparam.options = options;
%optimparam.adoptions = adoptionset('jacuser', true);

[optimout,optimparam]=dynopt(optimparam)
save optimresults optimout optimparam
[tplot,uplot,xplot] = profiles(optimout,optimparam,50);
save optimprofiles tplot uplot xplot
```

The value of the objective function evaluated for optimal control profile is of value of 1.204784e-01 after 359 iterations with exitflag equal to 1 (using default NLP solver *fmincon*). Graphical representation of the solution of the problem (**??**) is shown in Figs. **??** and **??**. Ipopt solver converged to a sightly different local optimum after 5883 iterations with the cost function value of 1.2026883e-01.

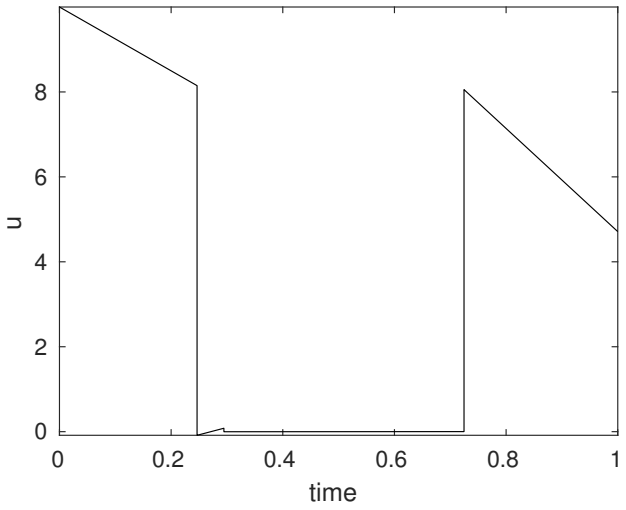

Figure 3.5: Control profile for unconstrained problem with bounds

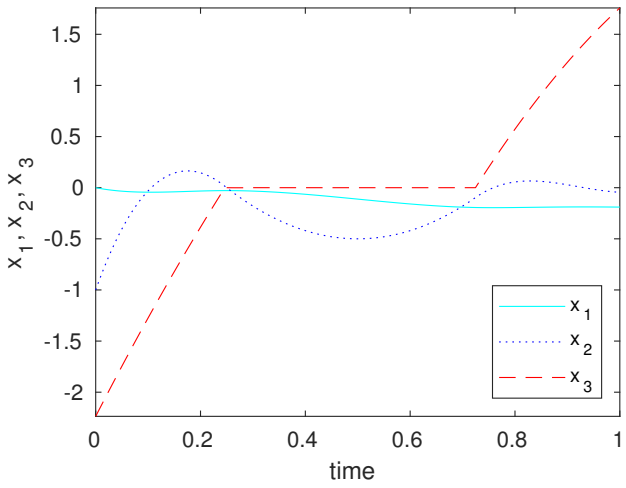

Figure 3.6: State profiles for unconstrained problem with bounds

Investigation of the issue with different local optima showed extreme sensitivity of gradient calculations and were affected by their construction using automatic differentiation. When process gradients were specified manually, the number of iterations in Ipopt differed (421 vs 659) based on expresion for the gradient $\partial f_4/\partial u$ specified with different multiplication order.

```
2*0.0005*(x(2)+16*t-8-0.1*x(3)*u(1)^2)*(-2*0.1*x(3)*u(1))
0.0005*(2*(x(2)+16*t-8-0.1*x(3)*u(1)^2)*(-0.1*x(3)*2*u(1)))
```

These options can be tested by activating either one of the lines:

```
optimparam.adoptions = adoptionset('processjac', "processd");
optimparam.adoptions = adoptionset('processjac', "processdalt");
```

The multiplication order specified in the file *processd.m* gave the same optimum as *fmincon*.

Control Constrained on Intervals

If we would like to constrain control on the first interval to be at most 9, we will redefine variable `bdu` (*examples/problem2_bdu/problem2bdu.m*). Now it will be a matrix composed of lower and upper bounds on every interval as follows

```
optimparam.bdu = [-4 9 -4 10 -4 10 -4 10];
```

The value of the objective function evaluated for optimal control profile has increased to 0.1249686 after 154 iterations with exitflag equal to 1. Graphical representation of the modified solution is shown in Figs. **??** and **??**.

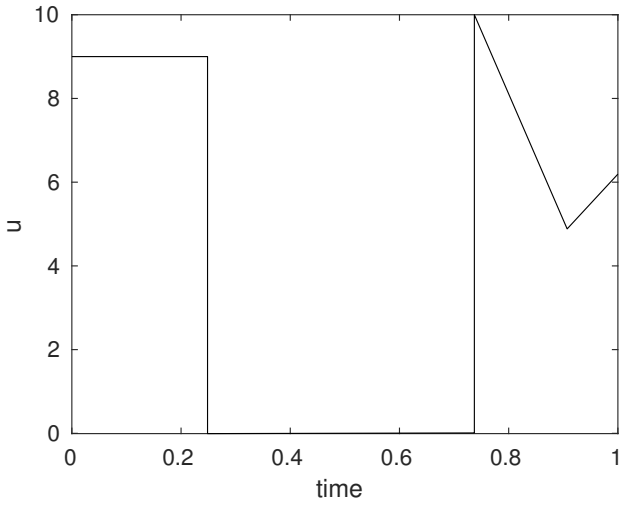

Figure 3.7: Control profile for unconstrained problem with gradients and bounds on intervals

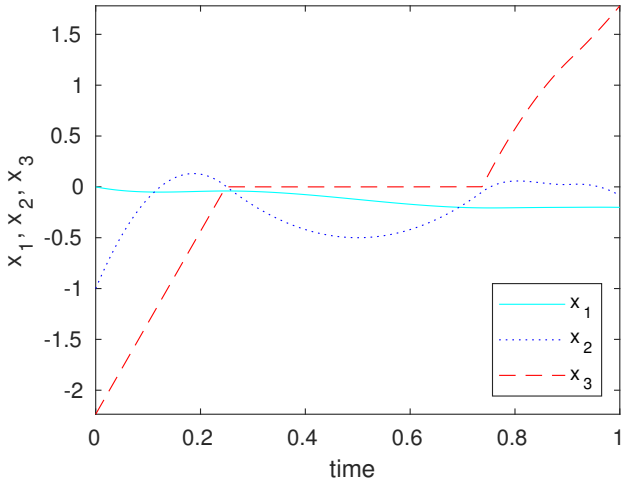

Figure 3.8: State profiles for unconstrained problem with gradients and bounds on intervals

3.1.4 Inequality State Path Constraint Problem

A process described by the following system of 2 ODE's [**?** **?**] (MATLAB code for this example is located at *examples/problem3*):

$$\dot{x}_1 = x_2, \qquad x_1(0) = 0 \tag{3.13}$$

$$\dot{x}_2 = -x_2 + u, \qquad x_2(0) = -1 \tag{3.14}$$

is to be optimised for $u(t)$ with the cost function:

$$\min_{\boldsymbol{u}(t)} \mathcal{J} = \int_0^1 (x_1^2 + x_2^2 + 0.005u^2)\mathrm{d}t \tag{3.15}$$

subject to state path constraint:

$$x_2 - 8(t - 0.5)^2 + 0.5 \leq 0, \quad t \in [0, 1] \tag{3.16}$$

with $x_1(t)$, $x_2(t)$ as states, $u(t)$ as control, such that $t_f = 1$.

As the objective function is not in the Mayer form as required by *dynopt*, we define an additional differential equation

$$\dot{x}_3 = x_1^2 + x_2^2 + 0.005u^2, \quad x_3(0) = 0 \tag{3.17}$$

and rewrite the cost as

$$\min_{\boldsymbol{u}(t)} \mathcal{J} = x_3(t_f) \tag{3.18}$$

Function *process*, *objfun*, *confun* definitions

Step1: Write an M-file *process.m*

```
function sys = process(t,x,flag,u,p)

  if flag == 0 % ODE system
    sys = [x(2);
   -x(2)+u;
   x(1)^2 + x(2)^2 + 0.005*u^2];
  elseif flag == 5  % initial conditions
    sys = [0;-1;0];
  else
    sys = [];
  end
end
```

Step2: Write an M-file *objfun*

```
function fun = objfun(t,x,u,p)
    fun = x(3);
end
```

Step3: Write an M-file *confun*

```
function [c, ceq] = confun(t,x,flag,u,p)

    if flag == 0                % constraints in t0
        c   = [];
        ceq = [];
    elseif flag == 1            % constraints over interval [t0,tf]
        x1 = x(1); x2 = x(2);
        c   = [x2 - 8*(t-0.5)^2 + 0.5];
        ceq = [];
    elseif flag == 2            % constraints in tf
          c = [];
        ceq = [];
    end
end
```

Step4: Invoke *dynopt* writing an M-file *problem3.m* as follows:

```
clear; close all; clc;
options = sdpoptionset('LargeScale','on','Display','iter','TolFun',1e-7,...
                       'TolCon',1e-7,'TolX',1e-7,...
                       'MaxFunEvals',1e5,'MaxIter',1e5,'Algorithm','sqp',...
                       'NLPsolver','fmincon');

optimparam.optvar = 3;
optimparam.objtype = [];
optimparam.ncolx = 6;
```

```
optimparam.ncolu = 2;
optimparam.li = ones(7,1)*(1/7);
optimparam.ui = zeros(1,7);
optimparam.tf = 1;
optimparam.par = [];
optimparam.bdu = [];
optimparam.bdx = [];
optimparam.bdp = [];
optimparam.objfun  = @objfun;
optimparam.confun  = @confun;
optimparam.process = @process;
optimparam.options = options;
%optimparam.adoptions = adoptionset('jacuser',true);

[optimout,optimparam] = dynopt(optimparam);
[tplot,uplot,xplot]   = profiles(optimout,optimparam,50);
[tp,cp,ceqp] = constraints(optimout,optimparam,50);
```

An optimal value of $x_3(t_f) = 0.1701564$ was computed after 3012 iterations with exitflag equal to 1 (using *fmincon*). Graphical representation of the solution of the problem (**??**) is shown in Figs. **??**, **??**, and **??**.

3.1.5 Parameter Estimation Problem

Consider a state estimation problem [**?**] where the cost functional is defined as the sum of squares of deviations between the model and measured outputs as follows (MATLAB code for this example is located at *examples/problem8*):

$$\min_{\boldsymbol{p}} \mathcal{J} = \sum_{i=1,2,3,5} (x_1(t_i) - x_1^{\mathrm{m}}(t_i))^2 \tag{3.19}$$

subject to the following ODE's:

$$\dot{x}_1 = x_2, \qquad x_1(0) = p_1 \tag{3.20}$$

$$\dot{x}_2 = 1 - 2x_2 - x_1, \qquad x_2(0) = p_2 \tag{3.21}$$

with x_1, x_2 as states and $t_f = 6$. The task is to find initial conditions denoted by the parameters $p_1, p_2 \in [-1.5, 1.5]$, if the input to the system is equal to 1. Measured outputs x_1^{m} and times of measurements are specified in Tab. **??**.

t	1	2	3	5
x_1^{m}	0.264	0.594	0.801	0.959

Table 3.1: Measured data for parameter estimation problem

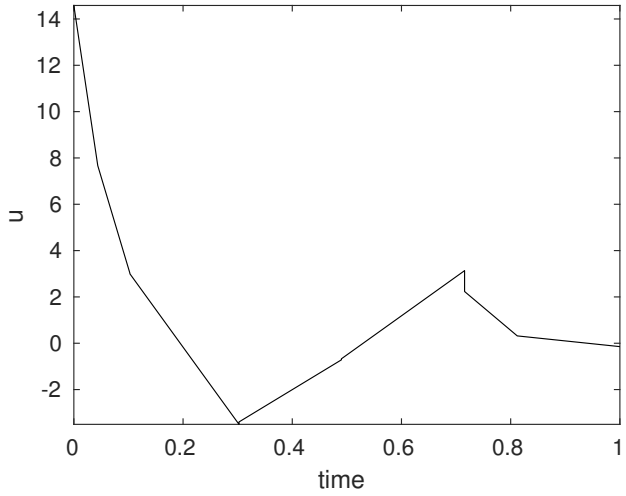

Figure 3.9: Control profile for inequality state path constraint problem

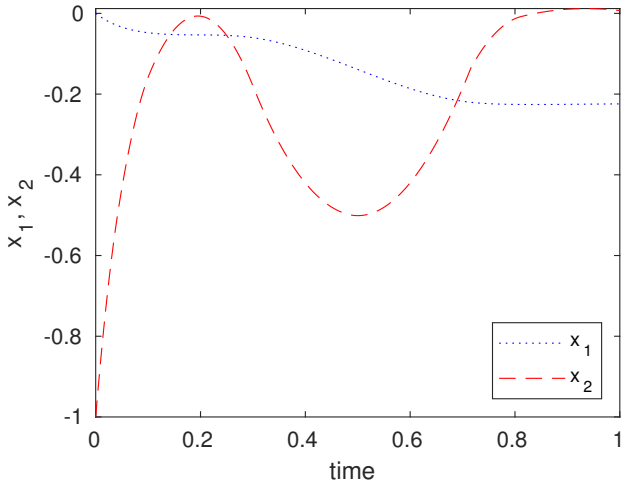

Figure 3.10: State profiles for inequality state path constraint problem

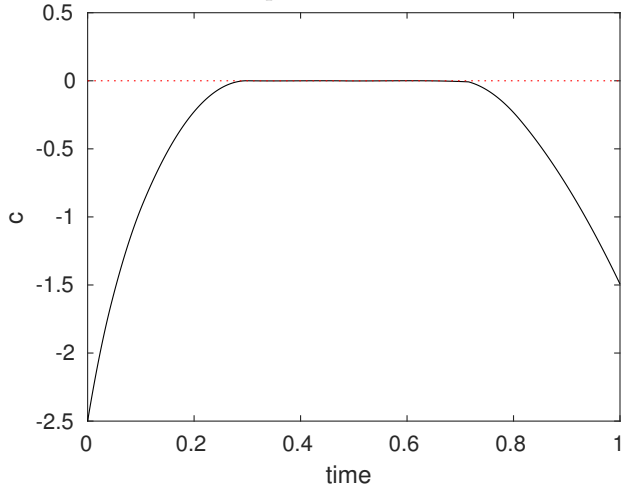

Figure 3.11: Constraint profile for inequality state path constraint problem

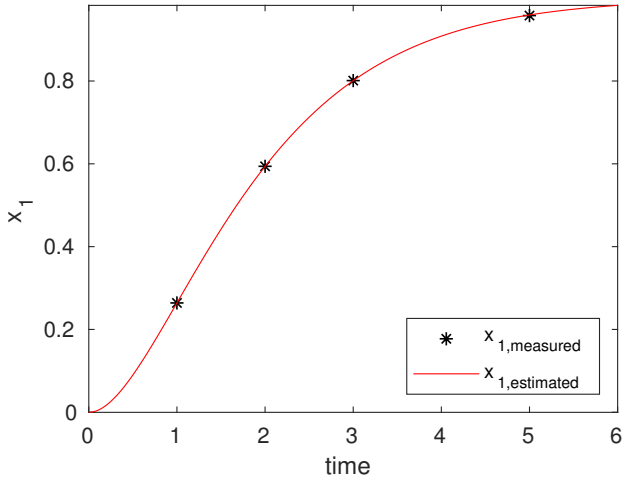

Figure 3.12: Comparison of estimated and measured state trajectory for state x_1 in parameter estimation problem

Function *process*, *objfun*, *confun* definitions

Step1: Write an M-file *process.m*

```
function sys = process(t,x,flag,u,p)

  if flag == 0 % ODE system
    sys = [x(2);
   1-2*x(2)-x(1)];
  elseif flag == 5  % initial conditions
    sys = [p(1);p(2)];
  else
    sys = [];
  end
end
```

Step2: Write an M-file *objfun*

```
function f = objfun(t,x,u,p,xm)
  f = (x(1)-xm(1))^2;
```

Step3: Write an M-file *confun*

Step4: Invoke *dynopt* writing an M-file *problem8.m* as follows:

```
clear; close all; clc;
options = optimset('LargeScale','off','Display','iter');
options = optimset(options,'TolFun',1e-7);
options = optimset(options,'TolCon',1e-7);
options = optimset(options,'TolX',1e-7);
options = optimset(options,'Algorithm','sqp');

%options.NLPsolver='ipopt';

objtype.tm = [1;2;3;5];
objtype.xm = [0.264 0.594 0.801 0.958;
              NaN NaN NaN NaN];

optimparam.optvar = 4;
optimparam.objtype = objtype;
optimparam.ncolx = 4;
optimparam.ncolu = [];
optimparam.li = ones(6,1);
optimparam.tf = [];
optimparam.ui = [];
optimparam.par = [1;1];
optimparam.bdu = [];
optimparam.bdx = [];
optimparam.bdp = [-1.5 1.5;-1.5 1.5];
optimparam.objfun = @objfun;
optimparam.confun = [];
optimparam.process = @process;
optimparam.options = options;

[optimout,optimparam]=dynopt(optimparam)
save optimresults optimout optimparam
[tplot,uplot,xplot] = profiles(optimout,optimparam,50);
save optimprofiles tplot uplot xplot
```

The results obtained by *dynopt* are the same as those published in [**?**] ($p_1 = 0, p_2 = 0$). Fig. **??** shows the comparison of estimated and measured state trajectory.

3.1.6 Minimum Time Problem

Consider a following problem optimising a car that has to be moved from origin and stopped at a distance of 300 m in a minimum time (MATLAB code for this example is located at

examples/problem-car)

$$\dot{x}_1 = u, \qquad x_1(0) = 0, \qquad x_1(t_f) = 0 \tag{3.22}$$

$$\dot{x}_2 = x_1, \qquad x_2(0) = 0, \qquad x_2(t_f) = 300 \tag{3.23}$$

optimised for $-2 \leq u(t) \leq 1$ with the cost function:

$$\min_{u(t)} \mathcal{J} = t_f \tag{3.24}$$

with $x_1(t)$ – velocity, $x_2(t)$ – distance, $u(t)$ control – acceleration.

Function *process*, *objfun*, *confun* definitions

Step1: Write an M-file *process.m*

```
function sys = process(t,x,flag,u,p)
  global x10 x20

  if flag == 0 % ODE system
    sys = [u(1);
   x(1)];
  elseif flag == 5  % initial conditions
    sys = [x10;x20];
  else
    sys = [];
  end
end
```

Step2: Write an M-file *objfun*

```
function fun = objfun(t,x,u,p)
    fun = t;
end
```

Step3: Write an M-file *confun*

```
function [c, ceq] = confun(t,x,flag,u,p)
global x1f x2f

    if flag == 0               % constraints in t0
        c   = [];
        ceq = [];
    elseif flag == 1           % constraints over interval [t0,tf]
        c   = [];
        ceq = [];
    elseif flag == 2           % constraints in tf
          c = [];
        ceq = [x(1) - x1f; x(2) - x2f];
    end
end
```

Step4: Invoke *dynopt* writing an M-file *car.m* as follows. Note that optimising the final time is indicated by setting it to an empty matrix: `optimparam.tf=[]` and by optimising the time interval lengths.

```
clear; close all; clc;
global x10 x20 x1f x2f
% initial conditions :
x10 = 0; x20 = 0;
% final conditions :
x1f = 0; x2f = 300;

options = sdpoptionset('LargeScale','on','Display','iter','TolFun',1e-7,...
                       'TolCon',1e-7,'TolX',1e-7,...
                       'MaxFunEvals',1e4,'MaxIter',1e3,'Algorithm','sqp',...
                       'NLPsolver','fmincon');

optimparam.optvar = 3;
optimparam.objtype = [];
optimparam.ncolx = 3;
optimparam.ncolu = 1;
optimparam.li = 100*ones(2,1)*(1/2);
optimparam.tf = [];
optimparam.ui = zeros(1,2);
optimparam.par = [];
optimparam.bdu = [-2 1];
optimparam.bdx = [0 300;0 400];
optimparam.bdp =[];
optimparam.objfun  = @objfun;
optimparam.confun  = @confun;
optimparam.process = @process;
optimparam.options = options;
%optimparam.adoptions = adoptionset('jacuser', true);

[optimout,optimparam]=dynopt(optimparam)
save optimresults optimout optimparam
[tplot,uplot,xplot] = profiles(optimout,optimparam,50);
save optimprofiles tplot uplot xplot
```

The value of the objective function evaluated for optimal control profile is of value of 30.00 after 85 iterations with exitflag equal to 1 (*fmincon*) or after 28 iterations (*ipopt*). Graphical representation of the solution of the problem (**??**) is shown in Fig. **??**.

The same problem with constrained velocity during the whole trajectory $x_1 < 10$ (MATLAB code for this example is located at *examples/problem-car2*) converged to the minimum value of 37.50 after 139/31 iterations (fmincon/ipopt). Graphical representation of the solution is shown in Fig. **??**.

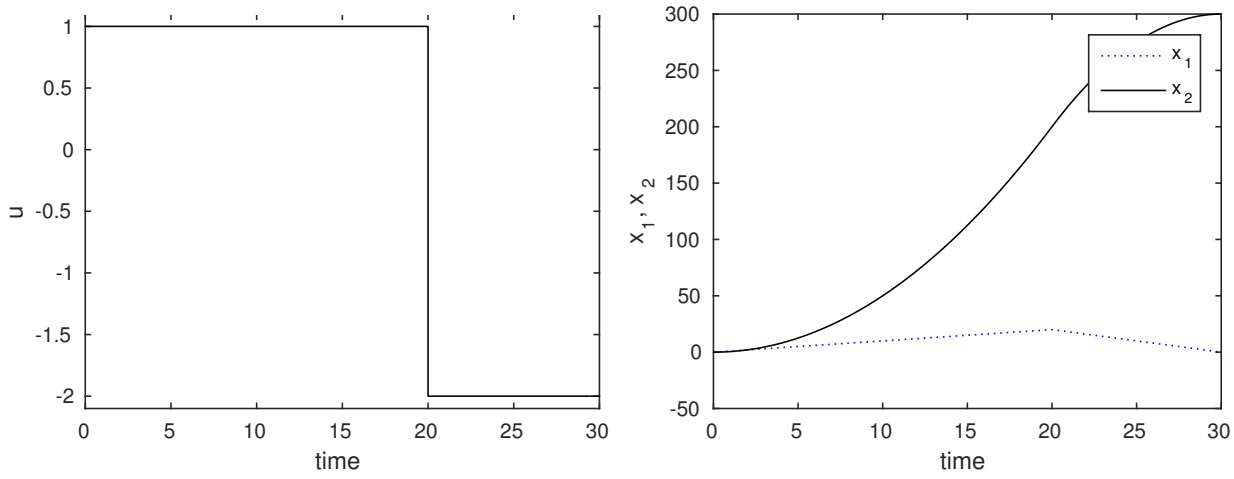

Figure 3.13: Control and state profiles for minimum time car problem

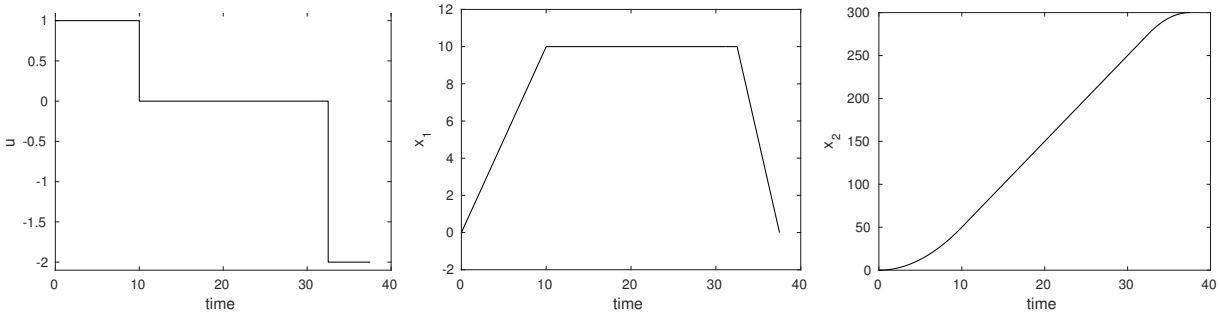

Figure 3.14: Control and state profiles for constrained minimum time car problem

3.2 DAE systems

3.2.1 Batch Reactor Problem

Consider a batch reactor [**?** **?**] with the consecutive reactions $A \rightarrow B \rightarrow C$ (MATLAB code for this example is located at *examples/problem5dae*):

$$\max_{\boldsymbol{u}(t)} \mathcal{J} = x_2(t_f) \tag{3.25}$$

such that

$$\dot{x}_1 = -k_1 x_1^2, \qquad x_1(0) = 1 \tag{3.26}$$

$$\dot{x}_2 = k_1 x_1^2 - k_2 x_2, \qquad x_2(0) = 0 \tag{3.27}$$

$$0 = k_1 - 4000 e^{(-\frac{2500}{T})} \tag{3.28}$$

$$0 = k_2 - 620000 e^{(-\frac{5000}{T})} \tag{3.29}$$

with x_1, x_2 as states representing concentrations of A, and B, temperature $T \in [298, 398]$ as control variable, such that $t_f = 1$. The system is defined by two ODEs and two DAEs.

Function *process*, *objfun*, *confun* definitions

Step1: Write an M-file *process.m*

```
function sys = process(t,x,flag,u,p)

  if flag == 0 % differential equations
      sys = [-x(3)*(x(1)^2);
     x(3)*(x(1)^2)-x(4)*x(2);
     x(3)-4000*exp(-u);
     x(4)-620000*exp(-2*u)];
  elseif flag == 5  % initial conditions
    sys = [1;0;5.0736;0.9975];
  else
    sys = [];
  end
end
```

Step2: Write an M-file *objfun*

```
function fun = objfun(t,x,u,p)
    fun = -x(2);
end
```

Step3: Invoke *dynopt* by writing an M-file *problem5dae.m* as follows:

```
clear; close all; clc;
options = optimset('LargeScale','off','Display','iter');
options = optimset(options,'MaxFunEvals',1e5);
options = optimset(options,'MaxIter',1e5);
options = optimset(options,'TolFun',1e-7);
options = optimset(options,'TolCon',1e-7);
options = optimset(options,'TolX',1e-7);
options = optimset(options,'Algorithm','sqp'); %2010a

%options.NLPsolver='ipopt';

optimparam.optvar = 3;
optimparam.objtype = [];
optimparam.ncolx = 5;
optimparam.ncolu = 2;
optimparam.li = ones(3,1)*(1/3);
optimparam.tf = 1;
optimparam.ui = ones(1,3)*7.35;
optimparam.par = [];
optimparam.bdu = [6.2813 8.3894];
optimparam.bdx = [0 1;0 1;0.9085 7.4936;0.0320 2.1760];
optimparam.bdp =[];
optimparam.objfun = @objfun;
optimparam.confun = [];
optimparam.process = @process;
optimparam.options = options;
M = zeros(4,4); M(1,1)=1; M(2,2)=1; optimparam.M = M;
```

```
[optimout,optimparam]=dynopt(optimparam)
save optimresults optimout optimparam
[tplot,uplot,xplot] = profiles(optimout,optimparam,50);
save optimprofiles tplot uplot xplot

%graph
```

Note the definition of the mass matrix M. For ODE systems, it is implicitly defined as identity matrix, for DAE systems it needs to be defined explicitly.

After 134 iterations, optimal value of $x_2(t_f) = 0.6106136$ was found using fmincon. The same solution was found using ipopt but after 13086 iterations. Graphical representation of the problem (**??**) solution is shown in Figs. **??** and **??**.

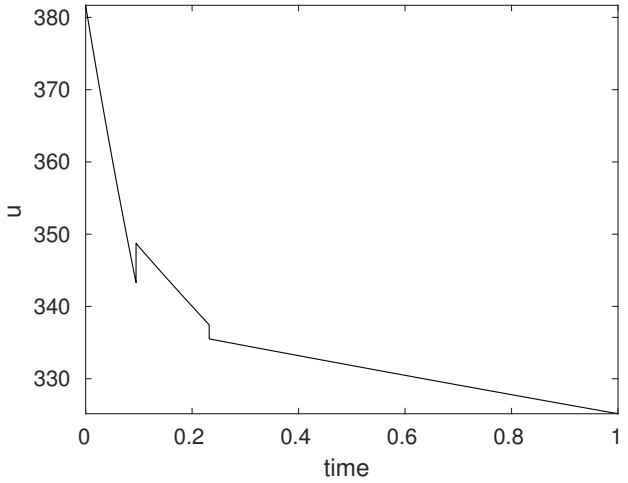

Figure 3.15: Control profile for batch reactor problem as DAE problem

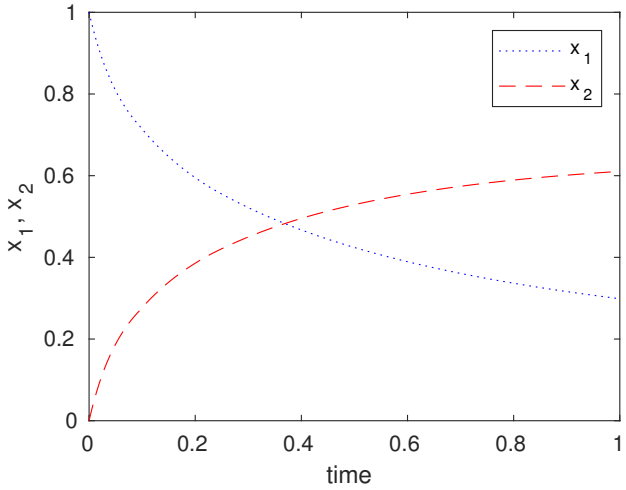

Figure 3.16: State profiles for batch reactor problem as DAE problem

3.3 Maximisation

dynopt performs minimisation of the objective function $f(t, x, u)$. Maximisation is achieved by supplying the routine with $-f(t, x, u)$.

3.4 Greater than Zero Constraints

The Optimisation Toolbox assumes nonlinear inequality constraints are of the form $C_i(x) \leq 0$. Greater than zero constraints are expressed as less than zero constraints by multiplying them by -1. For example, a constraint of the form $C_i(x) \geq 0$ is equivalent to the constraint $-C_i(x) \leq 0$.

CHAPTER 4

Reference

This chapter contains description of the function *dynopt*, the main function of the collection of functions which extend the capability of MATLAB Optimisation Toolbox, specifically of the constrained nonlinear minimisation routine *fmincon*. The chapter starts with section listing general descriptions of all the input and output arguments and the parameters in the optimisation options structure, continues with the function description, and ends with some tutorial.

4.1 Function Arguments

All input and output arguments to the *dynopt* function are described in this section. Section **??** describes all input arguments built in input structure `optimparam`. Then output arguments built in output structure `optimout` are treated in section **??** and as last the optimisation options parameters structure `options` which is given by MATLAB is described in Tab. **??**. It is important to mention here, that the names of input and output structures can be changed by user, but their fields described later have to be used as described.

ni	– number of intervals
nx	– number of state variables
nu	– number of control variables
np	– number of parameters
nm	– number of measurements

Table 4.1: Some predefined variables which are used for function description

Table **??** describes some predefined variables which are used to simplify *dynopt's* description in sections **??** and **??**.

4.1.1 Input Arguments

As mentioned before, input arguments described bellow do entry *dynopt* in a structure called `optimparam`. This contains them as fields, e.g., `optimparam.optvar`. `optimparam` has following fields to be set:

`optvar` – The choice of optimisation variables: 1 - times, 2 - control, 2 - parameters. Their combination is given by their summations, e.g., 3 - optimise times and control. All the possibilities are listed below

1 - optimise times,

2 - optimise control,

3 - optimise times and control,

4 - optimise parameters,

5 - optimise times and parameters,

6 - optimise control and parameters,

7 - optimise all: times, control, and parameters.

`objtype` – Parameter which defines the type of objective function to be minimised/maximised in optimisation. Two possible types of objective function may have been used:

Mayer type - if Mayer type objective function is used set the parameter `objtype` to an empty matrix.

Sum type - if Sum type objective function is used, parameter `objtype` is a structure containing two variables `tm`, and `xm`. `tm` is a nm-by-1 vector of times, in which the measurements are taken. `xm` is a nx-by-nm matrix of taken measurements in times `tm`. For more information about the types of objective functions see *objfun* description in section **??**.

`ncolx` – Parameter which represents the number of collocation points for state variables. This has always to be a number greater than zero.

`ncolu` – Parameter which represents the number of collocation points for control variables. It may have been defined as [ ] if control variable doesn't belong to optimisation variables and also doesn't occur in *process*, *objfun*, *confun*. Otherwise it has to be a number greater than zero.

`li` – Parameter representing lengths of intervals. It has always to be filled with ni–by–1 vector of initial lengths of intervals.

`tf` – Parameter representing the final time, if the value of t_f is not specified use empty brackets [ ].

`ui` – Parameter representing control variables applied on each time interval in `li`. As mentioned for `ncolu` parameter, if control variable is needed it has to be defined as nu–by–ni matrix of control variables for each interval. Otherwise it has to be an empty matrix [ ].

`par` – Parameter representing time independent parameters. As in `ui` also here it may have been defined either np–by–1 vector of time independent parameters or an empty matrix [ ].

`bdu` – Parameter representing bounds to the control variables. If not defined it has to be an empty matrix [ ]. If control constraints are the same in each time interval then it has to be an nu–by–2 matrix: [$\boldsymbol{lbu}$ $\boldsymbol{ubu}$]. If control constraints are not the same in each time interval then it has to be an nu–by–ni*2 matrix: [$\boldsymbol{lbu}_1$ $\boldsymbol{ubu}_1, \ldots, \boldsymbol{lbu}_{n_i}$ $\boldsymbol{ubu}_{n_i}$] where n_i is the number of intervals.

`bdx` – Parameter representing bounds to the states. If not defined it has to be an empty matrix [ ]. If state constraints are the same in each time interval then it has to be an nx–by–2 matrix: [$\boldsymbol{lbx}$ $\boldsymbol{ubx}$]. If state constraints are not the same in each time interval then it has to be an nx–by–ni*2 matrix: [$\boldsymbol{lbx}_1$ $\boldsymbol{ubx}_1, \ldots, \boldsymbol{lbx}_{n_i}$ $\boldsymbol{ubx}_{n_i}$] where n_i is the number of intervals.

`bdp` – Parameter representing bounds to the parameters. If defined it has to be an np–by–2 matrix: [$\boldsymbol{lbp}$ $\boldsymbol{ubp}$], otherwise an empty matrix [ ].

`objfun` – The function to be optimised. *objfun* is the name of an M-file. For more information about this input argument, see section **??**.

`confun` – The function that computes the nonlinear equality and inequality constraints. *confun* is the name of an M-file. For more information about this input argument, see section **??**.

`process` – The function that describes given process. *process* is the name of an M-file. For more information about this input argument, see section **??**.

`M` – If the process model is described by system of ODE's the mass matrix $\boldsymbol{M}$ does not need to be defined and *dynopt* sets it to identity matrix. If the system is described by DAE's the mass matrix $\boldsymbol{M}$ is specified here.

`options` – An optimisation options parameter structure that defines parameters used by the optimisation functions. This parameter is defined by MATLAB for all optimisation routines of MATLAB Optimization Toolbox. For information about the parameters which are important for *dynopt*, see Tab. **??** or the individual function reference pages. These parameters can be specified using the function *optimset*.

In addition to these, parameter `NLPsolver` determines NLP solver using the *fminsdp* toolbox. The default one is *fmincon* but others can be specified as well if installed separately. This parameter can be set directly as

```
options.NLPsolver='ipopt';
```

Parameter `options.adoptions` contains automatic differentiation arguments. It is supposed to be set using function `adoptionset` as name/value pairs (see Tab. **??**).

Table 4.2: Optimisation options parameters

Parameter Name	Description
DerivativeCheck	Compare user-supplied analytic derivatives (gradients) to finite differencing derivatives (medium-scale algorithm only), default value: 'off'.
Diagnostics	Print diagnostic information about the function to be minimised or solved, default value: 'off'.
Display	Level of display. 'off' displays no output, 'iter' displays output at each iteration, 'final' displays just the final output, default value: 'final'.
LargeScale	User large-scale algorithm if possible, default value: 'on'.
MaxFunEvals	Maximum number of function evaluations allowed, default value: '100*numberofvariables'.
MaxIter	Maximum number of iterations allowed, default value: 400.
TolCon	Termination Tolerance on the constraint violation, default value: 1.0000e-006.
TolFun	Termination Tolerance on the function value, default value: 1.0000e-006.
TolX	Termination Tolerance on x, default value: 1.0000e-006.
TypicalX	Typical x values (large-scale algorithm only), default value: 'ones(numberofvariables,1)'.
NLPsolver	Choice of NLP solver, possible values: 'fmincon', 'snopt', 'ipopt', 'knitro', 'mma', 'gcmma', default: 'fmincon'.

Commonly used options are

- Save automatically differentiated code for the next run

```
options.adoptions=adoptionset('keep', true);
```

- Automatically differentiated code reused from the previous run

```
options.adoptions=adoptionset('generate', false, 'keep', true);
```

- User defined jacobians, no automatic differentiation takes place

```
options.adoptions=adoptionset('userjac', true);
```

This is a shortcut for

```
options.adoptions=adoptionset('processjac', "processd",
      'objfunjac', "objfund", 'confunjac', "confund");
```

Table 4.3: Automatic differentiation parameters

Parameter Name	Description
processjac	function for user supplied gradients for the process. If not set, gradients are generated using Adigator. Default value: "processd".
objfunjac	function for user supplied gradients for the cost function. If not set, gradients are generated using Adigator. Default value: "objfund".
confunjac	function for user supplied gradients for the constraints. If not set, gradients are generated using Adigator. Default value: "objfund".
userjac	all gradients are provided manually. If set, it implies default function names "processd", "objfund", "confund". Default value: 'false'
generate	automatic gradients are generated at the initialisation phase. Default value: 'true'.
keep	automatic gradients are not deleted after the solution is found. Default value: 'false'.

4.1.2 Output Arguments

As for input arguments, the same holds for output arguments. That means that the output arguments described bellow do leave *dynopt* in a structure called `optimout`. This contains them as fields, e.g., `optimout.nlpx`. `optimout` has following fields:

`nlpx` – holds the solution found by the *dynopt*. If `exitflag` > 0, then `nlpx` is a solution otherwise, `nlpx` is the value the optimisation routine was at when it terminated prematurely. Vector `nlpx` contains all the parameters $\Delta\zeta_i, \boldsymbol{u_{ij}}, \boldsymbol{x_{ij}}, \boldsymbol{p}$ defined in the NLP formulation in section **??**.

`fval` – holds the value of the objective function in `objfun` at the solution `nlpx`.

`exitflag` – represents the exit condition of optimisation. `exitflag` may be:

> 0 indicates that the function converged to a solution `nlpx`,

0 indicates that the maximum number of function evaluations or iterations was reached,

< 0 indicates that the function did not converge to a solution.

`output` – represents an output structure that contains information about the results of the optimisation. `output.iterations` gives the information about the number of iteration, `output.funcCount` gives the information about the number of function evaluations, `output.algorithm` returns the used algorithm, `output.stepsize` returns the taken final stepsize (medium-scale algorithm only), `output.firstorderopt` gives the information about a measure of first-order optimality (large-scale algorithm only).

`lambda` – The Lagrange multipliers at the solution `nlpx`. `lambda` is a structure where each field is for a different constraint type. `lamdba.lower` for the lower bounds lb, `lambda.upper` for the upper bounds ub, `lambda.ineqlin` for the linear inequalities, `lambda.eqlin` for the linear equalities, `lambda.ineqnonlin` for the nonlinear inequalities, `lambda.eqnonlin` for the nonlinear equalities.

`grad` – holds the value of the gradient of objfun at the solution `nlpx`.

`t` – is a vector of times for optimal control profile returned by *dynopt*.

`u` – is a vector/matrix of optimal control profiles returned by *dynopt*.

`p` – is a vector/empty matrix of the optimal values of the parameters.

Function parameters described in section **??**, and Tab. **??** are implicitly given by MATLAB Optimization Toolbox for all it's subroutines. They also present parameters useful for *dynopt* through function *fmincon*.

4.2 Function Description

4.2.1 Purpose

The actual version of *dynopt* is able to solve dynamic optimisation problems which cost functions can be expressed either in the Mayer form or in the Sum form. The problem formulation can be described by following set of DAEs:

$$\min_{\boldsymbol{u}(t),\boldsymbol{p}} \mathcal{G}(\boldsymbol{x}(t_f), t_f, \boldsymbol{p}) \tag{4.1}$$

or

$$\min_{\boldsymbol{u}(t),\boldsymbol{p}} \sum_{i=1}^{nm} \mathcal{S}(t_i, \boldsymbol{x}(t_i), \boldsymbol{u}(t_i), \boldsymbol{p}, \boldsymbol{x}^{\text{mes}}(t_i)) \tag{4.2}$$

such that

$$\begin{aligned}
\boldsymbol{M}\dot{\boldsymbol{x}}(t) &= \boldsymbol{f}(t, \boldsymbol{x}(t), \boldsymbol{u}(t), \boldsymbol{p}) \\
\boldsymbol{x}(t_0) &= \boldsymbol{x}_0(\boldsymbol{p}) \\
\boldsymbol{h}(t, \boldsymbol{x}, \boldsymbol{u}, \boldsymbol{p}) &= \boldsymbol{0} \\
\boldsymbol{g}(t, \boldsymbol{x}, \boldsymbol{u}, \boldsymbol{p}) &\leq \boldsymbol{0} \\
\boldsymbol{x}_i^L \leq &\boldsymbol{x}(t) \leq \boldsymbol{x}_i^U, \quad i = 1, \ldots, n_i \\
\boldsymbol{u}_i^L \leq &\boldsymbol{u}(t) \leq \boldsymbol{u}_i^U, \quad i = 1, \ldots, n_i \\
\boldsymbol{p}^L \leq &\boldsymbol{p} \leq \boldsymbol{p}^U
\end{aligned}$$

with the following nomenclature:

$\mathcal{G}(\cdot)$ – objective function in Mayer form evaluated at final conditions,

$\sum_{i=1}^{nm} \mathcal{S}(\cdot)$ – objective function of Sum form evaluated in times of taking the measurements t_i,

$\boldsymbol{M}$ – a constant mass matrix,

$\boldsymbol{h}$ – equality design constraint vector,

$\boldsymbol{g}$ – inequality design constraint vector,

$\boldsymbol{x}(t)$ – state profile vector,

$\boldsymbol{u}(t)$ – control profile vector,

$\boldsymbol{p}$ – vector of time independent parameters,

$\boldsymbol{x}_0$ – initial conditions for state vector,

$\boldsymbol{x}_i^L, \boldsymbol{x}_i^U$ – state profile bounds on each interval $i = 1, \ldots, n_i$,

$\boldsymbol{u}_i^L, \boldsymbol{u}_i^U$ – control profile bounds on each interval $i = 1, \ldots, n_i$,

$\boldsymbol{p}^L, \boldsymbol{p}^U$ – bounds to the parameters.

4.2.2 Syntax and Description

```
[optimout,optimparam]=dynopt(optimparam)
```

starts with the initial lengths of intervals `li`, initial control values for each interval `ui` for defined number of collocation points for state variables `ncolx`, and for control variables `ncolu` to the final time `tf`, and minimises either a Mayer type `objfun` evaluated in the final time or Sum type `objfun` subject to the nonlinear inequalities or equalities defined in `confun` for time t_0, t_f or over full time interval characterised by flag in *confun* subject to a given system in `process` with the optimisation parameters specified in the structure `options`, with the defined set of lower and upper bounds on the control variables `bdu`, state variables `bdx`, and time independent parameters `bdp` so that solution is always in the range of these bounds. All before mentioned variables do entry *dynopt* in `optimparam` structure. The solution is returned in the `otpimout` structure described in section **??**.

4.2.3 Arguments

The arguments passed into the function are described in section **??**. The arguments returned by the function are described in section **??**. Details relevant to *dynopt* are included below for `objfun`, `confun`, `process`.

`objfun` The function to be minimised. `objfun` is a string containing the name of an M-file function, e.g., *objfun.m*. Whereas *dynopt* optimises a given performance index

Mayer form (**??**) objective function is evaluated at the final time t_f, thus *objfun* takes a scalar `t` - final time t_f, scalar/vector `x` - the state variable(s), scalar/vector `u` - the control variable(s), both evaluated at corresponding final time t_f, scalar/vector `p` - time independent parameters, and returns a scalar value `f` of the objective function evaluated at these value. The M-file function has to have the following form:

```
function f = objfun(t,x,u,p)

f = []; % J
```

Sum form (**??**) objective function is evaluated in the times of taking measurements t_i, thus *objfun* takes a scalar `t` - time of taking measurements t_i, scalar/vector `x` - state variable(s), `u` - the control variable(s), both evaluated at corresponding time t_i, scalar/vector `p` - time independent parameters, scalar/vector `xm` - measured

variable(s) in the above mentioned time t_i, and returns a scalar value `f` of the objective function evaluated at these values. The M-file function has to have the following form:

```
function f = objfun(t,x,u,p,xm)

f = []; % J
```

If the gradients of the objective function are not generated by Adigator but manually, option `'objfunjac'` defines the name of the function file (default function *objfund*) which returns the structure `Df` holding the gradient values with respect to time `t`, states `x`, controls `u` and parameters `p` as follows:

```
function Df = objfund(t,x,u,p)

% gradients of the objective function
Df.t = [];    % dJ/dt
Df.x = [];    % dJ/dx
Df.u = [];    % dJ/du
Df.p = [];    % dJ/dp
```

The gradients `Df.t`, `Df.x`, `Df.u`, `Df.p` are the partial derivatives of `f` at the points `t`, `x`, `u`, `p`. That means, `Df.t` is the partial derivative of `f` with respect to the `t`, the ith component of `Df.x` is the partial derivative of `f` with respect to the ith component of `x`, the ith component of `Df.u` is the partial derivative of `f` with respect to the ith component of `u`, the ith component of `Df.p` is the partial derivative of `f` with respect to the ith component of `p`.

`confun` The function that computes the nonlinear inequality constraints $\boldsymbol{g}(t,x,u,p) <= \mathbf{0}$ marked as output argument `c` and nonlinear equality constraints $\boldsymbol{h}(t,x,u,p) = 0$, marked as output argument `ceq`. As mentioned before, *dynopt* optimises a given performance index subject to the constraints defined in corresponding flag:

`flag` $= \mathbf{0}$ the constraints are implied at the beginning $t = t_0$,

`flag` $= \mathbf{1}$ the constraints are implied over the whole time interval $t \in [t_0, t_f]$,

`flag` $= \mathbf{2}$ the constraints are implied at the end $t = t_f$.

`confun` is a string containing the name of an M-file function, e.g., *confun.m*. *confun* takes a scalar `t` - time value corresponding to the time t, scalar/vector `x` - state variable value(s), and scalar/vector `u` - control variable value(s) both corresponding to the value of `t`, scalar/vector `p` - time independent parameters, and returns two arguments, a vector `c` of the nonlinear inequalities and a vector `ceq` of the nonlinear equalities, both evaluated at `t`, `x`, `u`, `p` for given flag. For example, if confun=@confun, then the M-file *confun.m* would have the form:

```
function [c,ceq] = confun(t,x,flag,u,p)
  if flag == 0          % constraints in t0
    c   = [];
    ceq = [];
  elseif flag == 1      % constraints over interval [t0,tf]
```

```
    c   = [];
    ceq = [];
  elseif flag == 2      % constraints in tf
    c   = [];
    ceq = [];
  end
end
```

If the gradients of the constraints are not generated by Adigator but manually, option `'confunjac'` defines the name of the function file (default function *confund*) with the same input arguments and with two output arguments, structures `Dc`, and `Dceq` holding the gradient values of constraints with respect to `t`, `x`, `u`, `p`.

```
function [Dc,Dceq] = confund(t,x,flag,u,p)

  if flag == 0 % constraints in t0
    Dc.t = [];
    Dc.x = [];
    Dc.u = [];
    Dc.p = [];
    Dceq.t = [];
    Dceq.x = [];
    Dceq.u = [];
    Dceq.p = [];
  elseif flag == 1 % constraints over interval [t0,tf]
    Dc.t = [];
    Dc.x = [];
    Dc.u = [];
    Dc.p = [];
    Dceq.t = [];
    Dceq.x = [];
    Dceq.u = [];
    Dceq.p = [];
  elseif flag == 2 % constraints in tf
    Dc.t = [];
    Dc.x = [];
    Dc.u = [];
    Dc.p = [];
    Dceq.t = [];
    Dceq.x = [];
    Dceq.u = [];
    Dceq.p = [];
  end
```

The gradients `Dc.t`, `Dc.x`, `Dc.u`, `Dc.p` are the partial derivatives of `c` at the points `t`, `x`, `u`, `p`. That means, `Dc.t` is the partial derivative of `c` with respect to `t`, the ith component of `Dc.x` is the partial derivative of `c` with respect to the ith component of `x`, the ith component of `Dc.u` is the partial derivative of `c` with respect to the ith component of `u`, the ith component of `Dc.p` is the partial derivative of `c` with respect

to the ith component of `p`, and the gradients `Dceq.t`, `Dceq.x`, `Dceq.u`, `Dceq.p` are the partial derivatives of `ceq` at the points `t`, `x`, `u`, `p`.

`process` The function which describes process model, that means the right hand sizes of ODE or DAE equations and process initial conditions:

`flag` = **0** right hand sides of ODE/DAE's,

`flag` = **5** initial conditions.

`process` is a string containing the name of an M-file function, e.g., *process.m*. *process* takes a time `t`, scalar/vector of state variable `x`, scalar `flag`, scalar/vector of control variable `u`, both corresponding to time `t`, and scalar/vector of time independent parameters `p`, and returns `sys` values with respect to `flag` value evaluated at time `t`. The M-file function has to be written in the following form:

```
function sys = process(t,x,flag,u,p)

  if flag == 0 % right sides of differential equations
    sys = [];
  elseif flag == 5 % initial conditions
    sys = [];
  else
    error(['unhandled flag = ',num2str(flag)]);
  end
end
```

If the gradients of the differential equations and initial conditions are not generated by Adigator but manually, option `'processjac'` defines the name of the function file (default function *processd*) with the same input arguments. The parameter `flag` defines the respective gradient:

`flag` = **1**

$$\partial \boldsymbol{f}/\partial \boldsymbol{x} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_{nx}}{\partial x_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_1}{\partial x_{nx}} & \cdots & \frac{\partial f_{nx}}{\partial x_{nx}} \end{bmatrix}$$

`flag` = **2** $\partial \boldsymbol{f}/\partial \boldsymbol{u}$,

`flag` = **3** $\partial \boldsymbol{f}/\partial \boldsymbol{p}$,

`flag` = **4** $\partial \boldsymbol{f}/\partial \boldsymbol{t}$,

`flag` = **6** $\partial \boldsymbol{x}_0/\partial \boldsymbol{p}$.

```
function sys = processd(t,x,flag,u,p)

  if flag == 1 % df/dx
    sys = [];
  elseif flag == 2 % df/du
    sys = [];
```

```
  elseif flag == 3 % df/dp
    sys = [];
  elseif flag == 4 % df/dt
    sys = [];
  elseif flag == 6 % dx0/dp
    sys = [];
  else
    sys = [];
  end
end
```

4.2.4 Algorithm

Large-scale optimisation By default *dynopt* will choose the large-scale algorithm of *fmincon* if only upper and lower bounds exists or only linear equality constraints exist. This algorithm is a subspace trust region method and is based on the interior-reflective Newton method described in [**?**]. Each iteration involves the approximate solution of a large linear system using the method of preconditioned conjugate gradients (PCG). See the trust-region and preconditioned conjugate gradient method descriptions in the Large-Scale Algorithms chapter in [**?**].

Medium-scale optimisation *dynopt* uses through the *fmincon* Sequential Programming (SQP) method. In this method, a Quadratic Programming (QP) subproblem is solved at each iteration. An estimate of the Hessian of the Lagrangian is updated at each iteration using the BFGS formula [**?**].

A line search is performed using a merit function similar to that proposed by [**?**]. The QP subproblem is solved using an active set strategy similar to that described in [**?**]. A full description of this algorithm is found in the Constrained optimisation section of the Introduction to algorithms chapter of the Optimization Toolbox manual. See also the SQP implementation section in the Introduction to Algorithms chapter for more details on the algorithm used.

The software layer of *fminsdp* toolbox adds different NLP solvers that have to be installed separately. This version of *dynopt* was tested with open source solver Ipopt 3.12.4. For concrete options of this solver and other available other ones see their documentations.

4.3 Additional Functions

In this section, two functions are presented: *profiles*, which prepares plot-able state and control profiles and *constraints*, which prepares a user given equality and inequality plot-able constraints from the optimisation results returned in `optimout`.

4.3.1 Function *profiles*

```
[tplot,uplot,xplot] = profiles(optimout,optimparam,ntimes)
```

takes an optimal output `optimout` and other input arguments `optimparam` described in section **??**, and returns vector `tplot`, vector/matrix `uplot`, vector/matrix `xplot` with respect to `ntimes` which defines the number of points plotted per interval.

4.3.2 Function *constraints*

```
[tp,cp,ceqp] = constraints(optimout,optimparam,ntimes)
```

takes an optimal output `optimout` returned by *dynopt*, and other input arguments `optimparam` described in section **??**, and returns vector `tp`,nonlinear inequality constraint vector/matrix `cp`, nonlinear equality constraint vector/matrix `ceqp` defined in `confun` with respect to `ntimes` which defines the number of points plotted per interval.

It is simple to make a graphical representation of obtained results by using MATLAB's *plot* function.

CHAPTER 5

Examples

This chapter contains a few another examples from the literature dealing with chemical reactors. The examples were chosen to illustrate the ability of the *dynopt* package to treat the problems of varying levels of difficulty. The example files can be found in the directory *examples/problemX*, where X means the number of the problem presented in this chapter.

5.1 Tubular Reactor

Consider a tubular reactor with parallel reactions $A \rightarrow B$, $A \rightarrow C$ taking place [**?** **?** **?**] (MATLAB code for this example is located at *examples/problem4*):

$$\max_{\boldsymbol{u}(t)} \mathcal{J} = x_2(t_f) \tag{5.1}$$

such that

$$\begin{aligned} \dot{x}_1 &= -(u + 0.5u^2)x_1 & x_1(0) &= 1 \\ \dot{x}_2 &= ux_1 & x_2(0) &= 0 \\ u &\in [0, 5] & t_f &= 1 \end{aligned}$$

where

$x_1(t)$ – dimensionless concentration of A,

$x_2(t)$ – dimensionless concentration of B,

$u(t)$ – control variable.

This problem was treated by [**?** **?** **?**] and the value of performance index of value of 0.57353 was reported as global optimum by [**?**]. Moreover the value of 0.57284 was reported by [**?**]. By using 6 collocation points for state variables, 2 collocation points for control variables on the same number of intervals as in the literature to this problem, we obtained a slightly closer value of performance index of 0.5734171 to the reported global maximum (109 iterations, exitflag equal to 1). Ipopt needed more than 11200 iterations and converged to 0.57305461. The optimal control and state profiles are given in Figs. **??** and **??**.

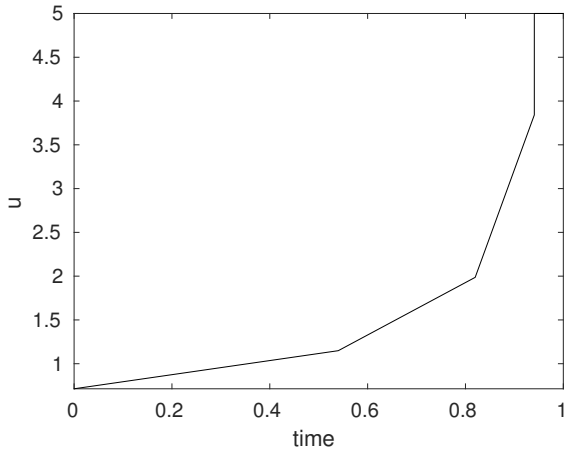

Figure 5.1: Control profile for problem 4

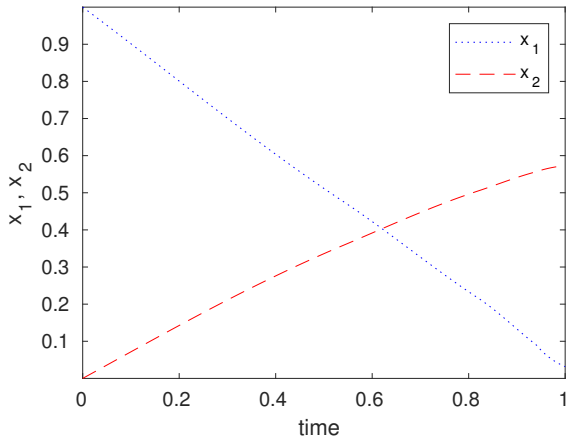

Figure 5.2: State profiles for problem 4

5.2 Batch Reactor

Consider a batch reactor [**?** **?**] where a series of reactions $A \to B \to C$ is involved. This example is similar to that in section **??**. The difference is just in the reactor model description. Here the process is described as an ODE system (MATLAB code for this example is located at *examples/problem5*):

$$\max_{\boldsymbol{u}(t)} \mathcal{J} = x_2(t_f) \tag{5.2}$$

such that

$$\begin{aligned} \dot{x}_1 &= -k_1 x_1^2 & x_1(0) &= 1 \\ \dot{x}_2 &= k_1 x_1^2 - k_2 x_2 & x_2(0) &= 0 \\ k_1 &= 4000 e^{(-\frac{2500}{T})} & k_2 &= 620000 e^{(-\frac{5000}{T})} \\ T &\in [298, 398] & t_f &= 1 \\ u &= \frac{2500}{T} \end{aligned}$$

where

$x_1(t)$ – concentration of A,

$x_2(t)$ – concentration of B,

T – temperature (control variable).

The objective of problem (**??**) is to obtain the optimal temperature profile that maximises the yield of the intermediate product B at the end of a specified time of operation in a batch reactor where the reaction $A \to B \to C$ take place. The problem was solved using a relaxed reduced space SQP strategy by [**?**] and the value of 0.610775 was reported as global maximum. **?** reached the value of 0.61045. We obtained optimal value of 0.6107682 (1000 iterations, 11468 function evaluations, exitflag equal to 0 (maxiter exceeded), by using 5 collocation points for state variables and keeping control variable profile as piecewise linear on 4 time intervals. This is quite closer to the global one. The same settings for ipopt resulted in a non-optimal solution. The optimal control and state profiles are given in Figs. **??** and **??**.

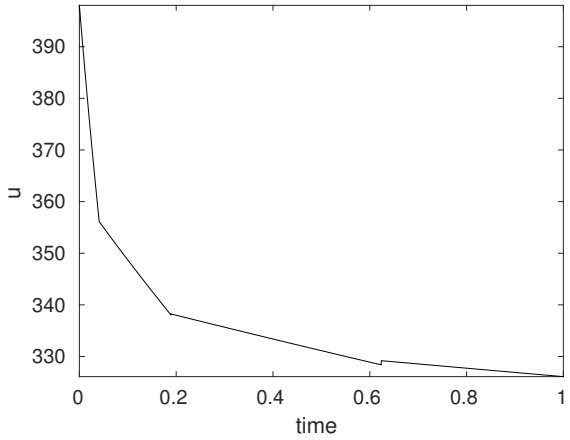

Figure 5.3: Control profile for problem 5

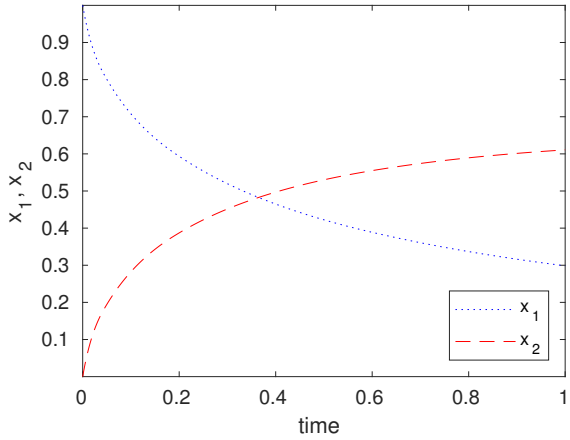

Figure 5.4: State profiles for problem 5

5.3 Catalytic Plug Flow Reactor

Consider a catalytic plug flow reactor [**?** **?**] involving the following reactions (MATLAB code for this example is located at *examples/problem6*):
$A \leftrightarrow B \rightarrow C$

$$\max_{\boldsymbol{u}(t)} \mathcal{J} = 1 - x_1(t_f) - x_2(t_f) \tag{5.3}$$

such that

$$\begin{aligned} \dot{x}_1 &= u(10x_2 - x_1) & x_1(0) &= 1 \\ \dot{x}_2 &= -u(10x_2 - x_1) - (1-u)x_2 & x_2(0) &= 0 \\ u &\in [0,1] & t_f &= 12 \end{aligned}$$

where

$x_1(t)$ – mole fraction of A,

$x_2(t)$ – mole fraction of B,

$u(t)$ – fraction of type 1 catalyst.

Optimisation of this problem has also been analysed. This problem was solved by [**?** **?**] and the optima 0.476946, 0.47615 were reported. Value of the performance index obtained for this problem using *dynopt* was 0.477712 (243 iterations, exitflag equal to 1, fmincon) or 0.47745804 (165 iterations, ipopt). In this case 5 collocation points for state variables and 2 collocation points for control variables were chosen. The number of time-intervals have been set to 12. The optimal control and state profiles are given in Figs. **??** and **??**.

5.4 Problem 7

Consider the following problem [**?** **?** **?**]

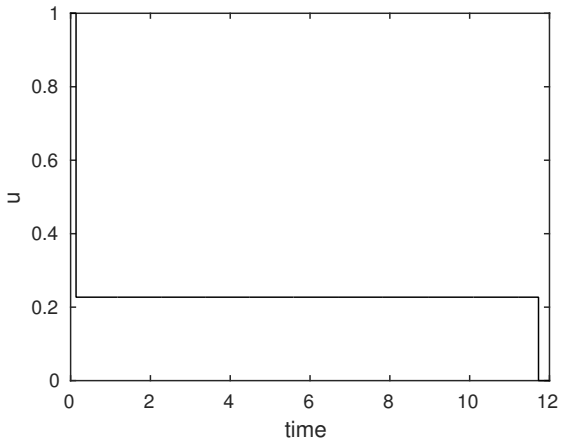

Figure 5.5: Control profile for problem 6

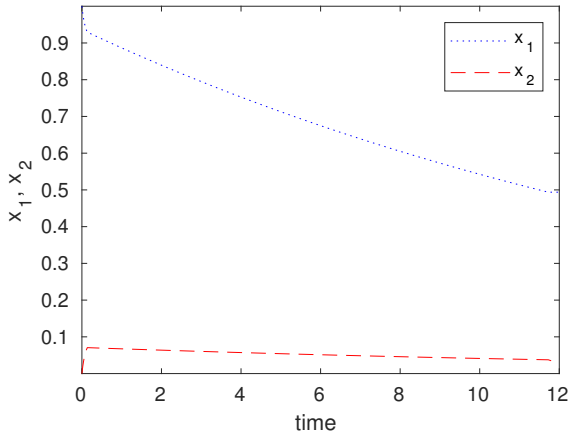

Figure 5.6: State profiles for problem 6

$$\begin{aligned}\max_{\boldsymbol{u}(t)} \mathcal{J} = \int_0^{0.2} \big(&5.8(qx_1 - u_4) - 3.7u_1 - 4.1u_2 \\ &+ q(23x_4 + 11x_5 + 28x_6 + 35x_7) - 5.0u_3^2 \\ &- 0.099\big)\mathrm{d}t \end{aligned} \tag{5.4}$$

such that

$$\begin{aligned}\dot{x}_1 &= u_4 - qx_1 - 17.6x_1x_2 - 23x_1x_6u_3 \\ \dot{x}_2 &= u_1 - qx_2 - 17.6x_1x_2 - 146x_2x_3 \\ \dot{x}_3 &= u_2 - qx_3 - 73x_2x_3 \\ \dot{x}_4 &= -qx_4 + 35.2x_1x_2 - 51.3x_4x_5 \\ \dot{x}_5 &= -qx_5 + 219x_2x_3 - 51.3x_4x_5 \\ \dot{x}_6 &= -qx_6 + 102.6x_4x_5 - 23x_1x_6u_3 \\ \dot{x}_7 &= -qx_7 + 46x_1x_6u_3 \\ \boldsymbol{x}(0) &= [0.1883\ 0.2507\ 0.0467\ 0.0899\ 0.1804\ 0.1394\ 0.1046]^{\mathrm{T}} \\ q &= u_1 + u_2 + u_4 \\ 0 &\leq u_1 \leq 20 \\ 0 &\leq u_2 \leq 6 \\ 0 &\leq u_3 \leq 4 \\ 0 &\leq u_4 \leq 20 \\ t_f &= 0.2\end{aligned}$$

where

$x_1(t) - x_7(t)$ – states,

$u_1(t) - u_4(t)$ – controls.

Analogous to the section **??**, the cost function can be rewritten to the Mayer form by introducing a new state defined by the integral function with its initial value equal to zero.

This problem was solved by [**?** **?**]. Reported optimal value of 21.757 was obtained using CVP method implemented in DYNO. For this problem, 4 collocation points for state variables, 2 collocation points for control variables for 10 intervals were defined and an optimum was found at value of 21.82346 (840 iterations, exitflag equal to 1, fmincon). The solver Ipopt did not converge after 10000 iterations. The optimal control and state profiles are given in Fig. **??**.

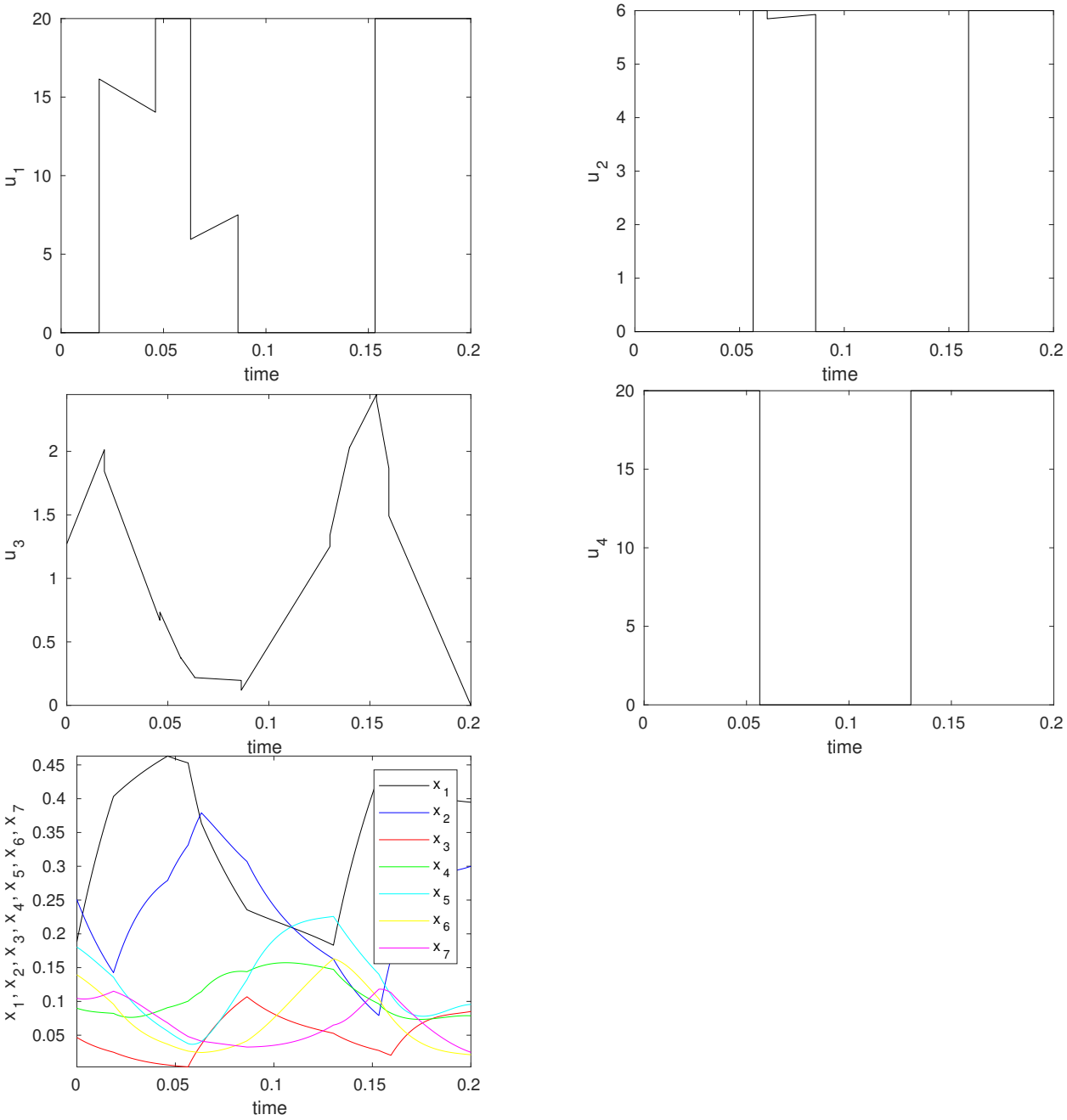

Figure 5.7: Control and state profiles for problem 7

5.5 Membrane System

Consider the following problem [**?**] which describes diafiltration optimal design problem (MATLAB code for this example is located at *examples/problem9*):

$$\min_{\alpha(t)} \mathcal{J} = x_2(t_f) \tag{5.5}$$

subject to differential equations:

$$\dot{x}_1 = \frac{x_1}{x_3} q(x_1, x_2) \left[\mathcal{R}_1(x_1, x_2) - \alpha\right], \qquad x_1(0) = 150 \tag{5.6}$$

$$\dot{x}_2 = \frac{x_2}{x_3} q(x_1, x_2) \left[\mathcal{R}_2(x_1, x_2) - \alpha\right], \qquad x_2(0) = 300 \tag{5.7}$$

$$\dot{x}_3 = q(x_1, x_2)(\alpha - 1), \qquad x_3(0) = 0.03 \tag{5.8}$$

state path constraints:

$$x_3(t) \geq 0.01 \tag{5.9}$$

$$x_3(t) \leq 0.035 \tag{5.10}$$

final time constraints:

$$x_3(t_f) = 0.01 \tag{5.11}$$

and simple bound constraints on optimized variable

$$\alpha \in [0, 1] \tag{5.12}$$

where $\mathcal{R}_1, \mathcal{R}_2, q$ are function of states determined experimentally as

$$q = S_1(x_2) e^{S_2(x_2) x_1} \tag{5.13}$$

$$\mathcal{R}_1 = (z_1 x_2 + z_2) x_1 + (z_3 x_2 + z_4) \tag{5.14}$$

$$\mathcal{R}_2 = W_1(x_2) e^{W_2(x_2) x_1} \tag{5.15}$$

where S_1, S_2, W_1, W_2 are second order polynomials in x_2

$$S_1(x_2) = s_1 x_2^2 + s_2 x_2 + s_3 \tag{5.16}$$

$$S_2(x_2) = s_4 x_2^2 + s_5 x_2 + s_6 \tag{5.17}$$

$$W_1(x_2) = w_1 x_2^2 + w_2 x_2 + w_3 \tag{5.18}$$

$$W_2(x_2) = w_4 x_2^2 + w_5 x_2 + w_6 \tag{5.19}$$

and $s_{1-6}, z_{1-4}, w_{1-6}$ are coefficients that were determined from laboratory experiments with the process solution (see Table **??**).

The optimal control profile $\alpha(t)$ is at zero for the first part of trajectory and one after the switch. The volume (x_3) shows that the first part of the trajectory basically decreases the volume until it is on the lower constraint and keeps it approximately constant until end of the batch. Thus, the optimal control strategy for this problem represents a traditional diafiltration process with two parts: pre-concentration followed by approximately constant-volume step until end of the batch. The minimum of $x_2(6) = 23.13$ was obtained with 2 piece-wise constant profiles of α. Simulation results are shown in Fig. **??**.

Table 5.1: Experimentally obtained coefficient values for $\mathcal{R}_j$ and q.

	s	w	z
1	$68.1250\,10^{-9}$	$7.8407\,10^{-6}$	$-0.0769\,10^{-6}$
2	$-56.4512\,10^{-6}$	$-4.0507\,10^{-3}$	$-0.0035\,10^{-3}$
3	$32.5553\,10^{-3}$	1.0585	$0.0349\,10^{-3}$
4	$-4.3529\,10^{-9}$	$1.2318\,10^{-9}$	0.9961
5	$3.3216\,10^{-6}$	$-9.7660\,10^{-6}$	
6	$-2.7141\,10^{-3}$	$-1.1677\,10^{-3}$	

Figure 5.8: Diafiltration problem: optimal α (left), concentrations (middle), and volume (right) as functions of time

5.6 Fed-bacth Reactor

Consider a fed-batch reactor for the production of ethanol. The optimal control problem is to maximize the yield of ethanol using the feed rate as the control variable. The aim is to find the feed flow rate $u(t)$ and the final time t_f in order to maximize

$$\max_{\boldsymbol{u}(t),t_f} \mathcal{J} = x_3(t_f)x_4(t_f) \tag{5.20}$$

such that

$$\begin{aligned}
\dot{x}_1 &= g_1 x_1 - u\frac{x_1}{x_4} \\
\dot{x}_2 &= -10 g_1 x_1 + u\frac{150 - x_2}{x_4} \\
\dot{x}_3 &= g_2 x_1 - u\frac{x_3}{x_4} \\
\dot{x}_4 &= u \\
g_1 &= \frac{0.408}{1 + x_3/16} \quad \frac{x_2}{0.22 + x_2} \\
g_2 &= \frac{1}{1 + x_3/71.5} \quad \frac{x_2}{0.44 + x_2} \\
x(0) &= [1\ 150\ 0\ 10]^{\mathrm{T}} \\
u &\in [0, 12] \\
x_4(t_f) &\leq 200
\end{aligned}$$

where

$$\begin{aligned}
x_1(t) - x_4(t) - &states \\
u(t) - &control.
\end{aligned}$$

The optimal control trajectory and the state profiles are depicted in Fig.**??** and Fig.**??**, respectively. First, until approximately 12 seconds, the control input is equal to zero and the concentration of the substrate is decreasing. After this concentration reaches the value equal to 1, the control input starts to increase progressively, what causes the increase of the product concentration, as well as the volume. At the end of the control, the volume is equal to 200, which satisfies the defined constrains.

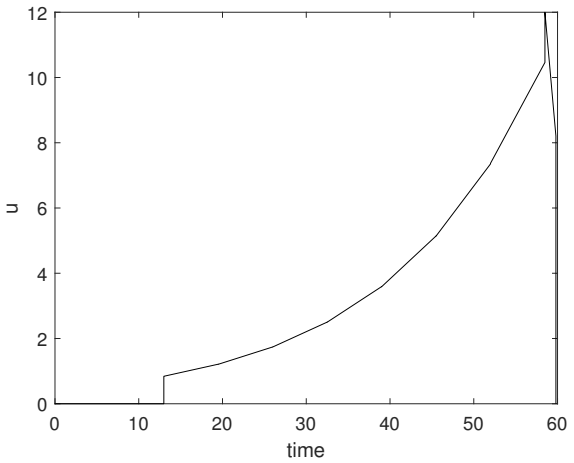

Figure 5.9: Control profile

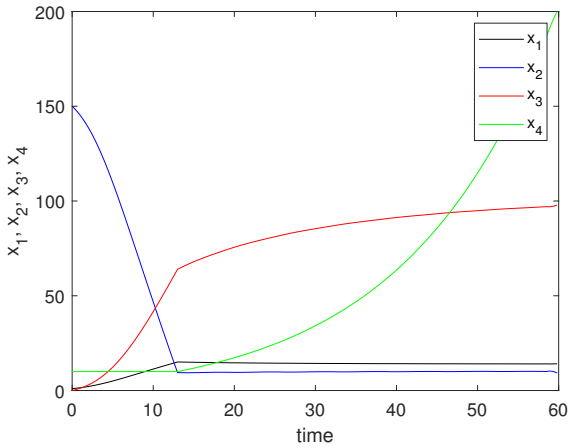

Figure 5.10: State profile