User's Guide for FORTRAN Dynamic Optimisation Code DYNO

M. Fikar and M. A. Latifi

Rapport final
Programme "Accueil de Chercheurs Étrangers"
Conseil Régional de Loraine
Fevrier 2002

Actual compilation:August 2, 2005

Abstract

DYNO is a set of FORTRAN 77 subroutines for determination of optimal control trajectory with unknown parameters given the description of the process, the cost to be minimised, subject to equality and inequality constraints.

The actual optimal control problem is solved via control vector parameterisation. That is, the original continuous control trajectory is 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 the 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 is described by a set of ordinary differential equations.

Contact

M. A. Latifi Laboratoire des Sciences du Génie Chimique, CNRS-ENSIC, B.P. 451, 1 rue Grandville, 54001 Nancy Cedex, France.

e-mail:latifi@ensic.inpl-nancy.fr, tel. +33 (0)3 83 17 52 34, fax +33 (0)3 83 17 53 26

M. Fikar 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.

email:miroslav.fikar@stuba.sk, tel. +421 (0)2 593 25 354, fax +421 (0)2 39 64 69

Internet

http://www.kirp.chtf.stuba.sk/~fikar/dyno.html

Actual Version

1.2, 21.08.2004

From the licence

The DYNO 1.x Software is **not** in the public domain. However, it is available for license without fee, for education and non-profit research purposes. Any entity desiring permission to incorporate this software or a work based on the software into commercial products or otherwise use it for commercial purposes should contact the authors.

1. The "Software", below, refers to the DYNO 1.x (in either source-code, object-code or executable-code form), and a "work based on the Software" means a work based on either the Software, on part of the Software, or on any derivative work of the Software under copyright law: that is, a work containing all or a portion of the DYNO, either verbatim or with modifications. Each licensee is addressed as "you" or "Licensee".

2. STU Bratislava and ENSIC-LSGC Nancy are copyright holders in the Software. The copyright holders reserve all rights except those expressly granted to the Licensee herein.

3. Licensee shall use the Software solely for education and non-profit research purposes within Licensee's organization. Permission to copy the Software for use within Licensee's organization is hereby granted to Licensee, provided that the copyright notice and this license accompany all such copies. Licensee shall not permit the Software, or source code generated using the Software, to be used by persons outside Licensee's organization or for the benefit of third parties. Licensee shall not have the right to relicense or sell the Software or to transfer or assign the Software or source code generated using the Software.

4. Due acknowledgment must be made by the Licensee of the use of the Software in research reports or publications. Whenever such reports are released for public access, a copy should be forwarded to the authors.

5. If you modify a copy or copies of the Software or any portion of it, thus forming a work based on the Software, and make and/or distribute copies of such work within your organization, you must meet the following conditions:

   (a) You must cause the modified Software to carry prominent notices stating that you changed specified portions of the Software.

   (b) If you make a copy of the Software (modified or verbatim) for use within your organization it must include the copyright notice and this license.

6. NEITHER STU, ENSIC-LSGC NOR ANY OF THEIR EMPLOYEES MAKE ANY WARRANTY, EXPRESS OR IMPLIED, OR ASSUMES ANY LEGAL LIABILITY OR RESPONSIBILITY FOR THE ACCURACY, COMPLETENESS, OR USEFULNESS OF ANY INFORMATION, APPARATUS, PRODUCT, OR PROCESS DISCLOSED AND COVERED BY A LICENSE GRANTED UNDER THIS LICENSE AGREEMENT, OR REPRESENTS THAT ITS USE WOULD NOT INFRINGE PRIVATELY OWNED RIGHTS.

7. IN NO EVENT WILL STU, ENSIC-LSGC BE LIABLE FOR ANY DAMAGES, INCLUDING DIRECT, INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM EXERCISE OF THIS LICENSE AGREEMENT OR THE USE OF THE LICENSED SOFTWARE.

Changes to previous versions

Version 1.2 The main change was support for automatic differentiation and implementation of ADIFOR.

- `info(16)`: The input array `info` has one more element specifying the AD usage.
- A new file has to be linked with `DYNO`: Either `adifno.f` if `info(16)=0` or `adifyes.f` otherwise.
- Documentation includes new section about usage of ADIFOR.
- Examples are implemented both with and without AD tools.

Contents

Chapter 1

Introduction

The speed of computers has enabled to solve many engineering problems that were not possible to deal with in the past. One of them is optimal control for dynamic systems. Many engineering problems fall into the scope of this formulation. Although the problem has received much attention in the literature, not many computer implementations currently exists. For this reason, we have developed the dynamic optimisation software package DYNO (DYNnamic Optimisation).

The main aim of this report is to be a user manual. However, it also gives the theoretical foundations of the algorithms used so that the code and its usage is more comprehensible for the user.

1.1 Problem Formulation

1.1.1 System and Cost Description

Consider an ordinary differential system (ODE) system described by the following equations

$$\dot{\boldsymbol{x}} = \boldsymbol{f}(t, \boldsymbol{x}, \boldsymbol{u}, \boldsymbol{p}), \qquad \boldsymbol{x}(t_0) = \boldsymbol{x}_0(\boldsymbol{p}) \tag{1.1}$$

where t denotes time from the interval $[t_0, t_P]$, $\boldsymbol{x} \in \mathcal{R}_{n_x}$ is the vector of differential variables, $\boldsymbol{u} \in \mathcal{R}_{n_u}$ is the vector of controls, and $\boldsymbol{p} \in \mathcal{R}_{n_p}$ is the vector of parameters. The vector valued function $\boldsymbol{f} \in \mathcal{R}_{n_x}$ describes the right hand sides of differential equations. We suppose that the initial conditions of the process can be a function of the parameters.

We assume that the (originally continuous) control can be approximated as piece-wise constant on P time intervals

$$\boldsymbol{u}(t) = \boldsymbol{u}_j, \quad t_{j-1} \leq t < t_j \tag{1.2}$$

and we denote the interval lengths by $\Delta t_j = t_j - t_{j-1}$.

Consider now the criterion to be minimised J_0 and constraints J_i of the form

$$\begin{aligned} J_0 &= G_0(t_j, \boldsymbol{x}(t_1), \ldots, \boldsymbol{x}(t_P), \boldsymbol{u}(t_1), \ldots, \boldsymbol{u}(t_P), \boldsymbol{p}) + \int_{t_0}^{t_P} F_0(t, \boldsymbol{x}, \boldsymbol{u}, \boldsymbol{p}) \mathrm{d}t \qquad (1.3) \\ J_i &= G_i(t_j, \boldsymbol{x}(t_1), \ldots, \boldsymbol{x}(t_P), \boldsymbol{u}(t_1), \ldots, \boldsymbol{u}(t_P), \boldsymbol{p}) + \int_{t_0}^{t_P} F_i(t, \boldsymbol{x}, \boldsymbol{u}, \boldsymbol{p}) \mathrm{d}t \qquad (1.4) \end{aligned}$$

where the constraints are for $i = 1, \ldots, m$, (m is the number of constraints). In the sequel we will assume that the constraints are ordered such that the first m_e are equality constraints of the form $J_i = 0$ and the last $m_i = m - m_e$ constraints are inequalities of the form $J_i \geq 0$.

The further constraints that are considered include simple bounds of the form

$$\begin{aligned} \boldsymbol{u}_j &\in [\boldsymbol{u}_j^{\min}, \boldsymbol{u}_j^{\max}] \\ \Delta t_j &\in [\Delta t_j^{\min}, \Delta t_j^{\max}] \\ \boldsymbol{p} &\in [\boldsymbol{p}^{\min}, \boldsymbol{p}^{\max}] \end{aligned} \qquad (1.5)$$

1.1.2 Optimised Variables

The optimised variables $\boldsymbol{y}$ are parameters $\boldsymbol{p}$, piece-wise constant parametrisation of control $\boldsymbol{u}_j$, and time interval lengths Δt_j. The reason for utilising Δt_j instead of t_j is simpler description of the bounds (lower bound is usually a small positive number for Δt_j). Also, simple bounds are more easily handled by NLP solvers.

Hence the vector $\boldsymbol{y} \in \mathcal{R}_q$ of optimised variables is given as

$$\boldsymbol{y}^T = (\Delta t_1, \ldots, \Delta t_P, \boldsymbol{u}_1^T, \ldots, \boldsymbol{u}_P^T, \boldsymbol{p}^T). \qquad (1.6)$$

Example 1 (Minimum time problem for a linear system)**.** Consider a system described by a second order linear time invariant equation

$$y''(t) + 2y'(t) + y(t) = u(t), \quad y'(0) = 0, y(0) = 0 \qquad (1.7)$$

and the task is to minimise the final time t_f to obtain a new stationary state characterised by $y'(t_f) = 0$, $y(t_f) = 1$, and subject to the constraints on control $u \in [-0.5, 1.5]$. Let us divide the total optimisation time t_f for example into 3 intervals where the control can be assumed to be constant. We can define the optimisation problem with the unknown variables u_1, u_2, u_3, $\Delta t_1, \Delta t_2, \Delta t_3$.

Rewriting the system equation into two first order ODE's and specifying the constraints gives

$$\begin{aligned} x_1' &= x_2, & x_1(0) &= 0 \\ x_2' &= u - x_1 - 2x_2, & x_2(0) &= 0 \\ J_0 &= \Delta t_1 + \Delta t_2 + \Delta t_3 \\ J_1 &= x_1(t_3) - 1 \\ J_2 &= x_2(t_3) \\ u_i &\in [-0.5, 1.5], \quad i = 1, 2, 3 \\ \Delta t_i &\geq 0, \quad i = 1, 2, 3 \end{aligned} \qquad (1.8)$$

Chapter 2

Optimal Control Problems

We discuss here some types of problems that can be solved by the package.

2.1 Control Parameterisation

It might seem that the piece-wise constant control parameterisation can be too imprecise in certain situation and that it cannot approximate closely enough the original continuous type trajectory.

Of course, one can take the number of time intervals P sufficiently large to obtain finer resolution. However, this will add to complexity of the master NLP problem, possibly find many local minima or it will results in convergence problems.

However, also any other parameterisation can be reformulated for the problem with piece-wise constant controls. Consider for example piece-wise linear control of the form

$$u(t) = a + bt \tag{2.1}$$

It is clear, that the coefficients a, b can serve as new commands. Hence u_1, u_2 in

$$\boldsymbol{u}(t) = u_1 + u_2 t \tag{2.2}$$

are new control variables.

Another useful parameterisation is by the Lagrange polynomials. These are defined as

$$P_j(t) = \prod_{i=0,j}^{N} \frac{t - t_i}{t_j - t_i} \tag{2.3}$$

where N is the degree of the polynomial $P_j(t)$ and the notation $i = 0, j$ denotes i starting from zero and $i \neq j$.

The times t_i are usually specified as the roots of the Legendre polynomials (Villadsen and Michelsen, 1978; Cuthrell and Biegler, 1989).

The control parameterisation can then be expressed as

$$u(t) = \sum_{j=1}^{N} \bar{u}_j \psi_j(t), \quad \psi_j(t) = \prod_{i=1,j}^{N} \frac{t - t_i}{t_j - t_i} \tag{2.4}$$

and the elements $\bar{u}_i$ serve as piece-wise constant control variables in the optimisation. The advantage of the Lagrange polynomials follows from the fact, that

$$u(t_i) = \bar{u}_i \tag{2.5}$$

and thus the coefficients $\bar{u}_j$ are physically meaningful quantities. This is useful as their bounds are the same as the bounds on the original control.

2.1.1 Piece-wise Continuous Control

By default, control variables across the time intervals are considered as independent. There are however, situations, when it is desired that the overall control trajectory remains continuous. There are two possible solutions:

1. Add constraints on control across time boundaries of the form

   $$\boldsymbol{u}(t_i^-) = \boldsymbol{u}(t_i^+) \tag{2.6}$$

   This adds $P - 1$ equality constraints (each of dimension of the control vector). As these constraints do not contain states, no adjoint system of equations has to be generated for them. However, NLP solver can have convergence problems due to a large number of equality constraints.

2. Add a new state variable for each element of the control vector. It will represent control. Thus, its initial value has to be optimised (control at time t_0) and its derivative is equal to the control approximation derivative. Assuming for example linear control parameterisation of the form $u_j = a_j + b_j t$, the differential equation of the new state is given as

   $$\dot{x}^u = b_j, \quad x(t_0) = a_1 \tag{2.7}$$

   and x^u replaces all occurrences of control u in the process and cost equations. The optimised parameters are then $b_1, \ldots, b_P, a_1$ – the slopes and the initial value.

2.2 State Path Constraints

State path constraints are usually of the form

$$g(\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{p}, t) \geq 0, \quad t \in [t_0, t_P] \tag{2.8}$$

$$g(\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{p}, t) = 0, \quad t \in [t_0, t_P] \tag{2.9}$$

and are in general very difficult to be satisfied along the desired time range. There are several methods that can deal with these constraints by either removing them or transforming them to the form (1.3). To do so, it is important to use only such kinds of transformations, that do not introduce non-smooth (non-differentiable) behaviour.

2.2.1 Equality Constraints

These constraints impose relations between state and control variables. The consequence is that some of the control variables, or rather their linear combinations cannot be regarded as optimised variables, and the number of degrees of freedom in optimisation decreases.

The first possible method when dealing with equality constraints that depend directly on control variables is to try to find explicitly the dependence of control on states and replace the corresponding control in the state and cost equations.

If the constraints do not depend directly on control variables or this dependence is difficult to specify explicitly, one can use a general method of converting the path constraint to integral constraint. To do so, a new state variable is defined as

$$\dot{x}^g = g(\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{p}, t), \quad x^g(0) = 0 \tag{2.10}$$

and this differential equation is appended to other state equations. If the equality constraint holds, the derivative is zero all the time and one can define an integral constraint of the form

$$J^g = \int_{t_0}^{t_P} (\dot{x}^g)^2 \, \mathrm{d}t = \int_{t_0}^{t_P} (g)^2 \, \mathrm{d}t = 0 \tag{2.11}$$

If the integral constraint is zero, then the equality constraint holds.

Due to the convergence reasons of the NLP, the equality is often relaxed to

$$J^g = \int_{t_0}^{t_P} g^2 \mathrm{d}t < \varepsilon, \quad \varepsilon > 0 \tag{2.12}$$

Note that this means that a small violation of the constraint is allowed.

2.2.2 Inequality Constraints

Integral Approach

An inequality constraint can be transformed into an end-point constraint by the use of the integral transformation technique described above:

$$J^g = \int_{t_0}^{t_P} h(g) \mathrm{d}t = 0 \tag{2.13}$$

where h measures the degree of violation of the inequality constraint during the entire trajectory.

Several formulations for selection of a suitable h have been proposed:

Max The most simple approach is to use $h = \min(g, 0)$. As the gradient of this operator is discontinuous, it poses problems in the integration and is in general not recommended.

Max2 An improvement over the previous solution is to avoid the discontinuity, for example as with $h = [\min(g, 0)]^2$.

Smoothing The proposition given in Teo et al. (1991) lies in the discontinuity replacement by a smooth approximation. Therefore, in the region of $|g| < \delta$, a quadratic smoothing is employed:

$$h = \begin{cases} g & \text{if } g < -\delta \\ -\dfrac{(g-\delta)^2}{4\delta} & \text{if } |g| \leq \delta \\ 0 & \text{if } g > \delta \end{cases} \tag{2.14}$$

The advantage over the Max2 method is elimination of squaring, therefore small deviations are penalised more heavily. A drawback (in practice not very important) is slightly smaller feasibility region.

The disadvantage of these methods is that no distinction is made whether the path constraint is not active on the overall trajectory or it is exactly at the limit. In both cases the integral is zero. Moreover, if the path constraint is not violated, its gradient with respect to the optimised parameters is zero. This poses problems to NLP and reduces its convergence speed considerably as it oscillates between zero and nonzero gradients. Further, the gradients are zero at the optimum.

Interior point constraints

As the previous methods transformed the inequality constraints into a single integral measure, only a very little information is given to NLP problem. A possible solution is to discretise the constraint and require it to be satisfied only at some points – usually at the end of the segments at times t_j. Thus,

$$g(t_j) \geq 0 \tag{2.15}$$

Although such a formulation is more easily solved by the NLP solver, it can result in pathological behaviour when the constraint is respected only at the prescribed points but violated in between. The situation gets worse if also the times t_j are optimised. The usual solution found contains one very large time t_k within which the constraint exceeds largely the possible tolerated violation.

The situation is better if the times are not optimised and tolerable constraint violations can be obtained. This approach is used in Model Predictive Control.

Combination of End and Interior Point Methods

A straightforward improvement consists in the combination of the integral end-point function with the set of interior point constraints at the end time segments. The advantage lies in the fact that the NLP solver gets more information even when the integral is zero. There is a possible drawback: when gradients are calculated by the method of adjoint variables, as it is done in `DYNO`, because each interior point constraint is state dependent. Thus, for each point constraint, another system of adjoint equations has to be formed and integrated backwards in time. This slows down computational time significantly.

Slack Variables

The slack variable method (Jacobson and Lele, 1969; Bryson and Ho, 1975; Feehery and Barton, 1998) is based on the techniques of optimal control and among other methods mentioned above it is one of the most rigorous. However, it has many drawbacks so it cannot be used as a general purpose method.

The principle of the method is given by changing the original inequality to equality by means of a slack variable $a(t)$:

$$g(x) - \frac{1}{2}a^2 = 0 \tag{2.16}$$

The slack variable is squared so that any value of a is admissible.

The equation is then differentiated so many times until a explicit solution for u can be found. The control variable is then eliminated from the system equations. Therefore, the principle of the method is to make the control (one scalar element from the control vector) a new state variable for the corresponding constraint. Thus, the state constraint is appended as equality to the system, the control variable is solved as a state variable during the integration and the slack variable $a(t)$ becomes the new optimised variable (and it has to be parametrised suitably).

The original proposition in Jacobson and Lele (1969) dealt only with ODE (ordinary differential equations). Improvement and generalisation of this procedure for general DAE (differential algebraic systems) has been proposed by Feehery and Barton (1998).

Although the method is appealing and gives very good results, its drawbacks are as follows:

- Number of the state constraints cannot be larger than the dimension of the control vector. The other approaches can give a feasible solution when the number of *active* constraints is equal or smaller than the control dimension. A possible workaround has been proposed by Feehery and Barton (1998) where state event based approach is used.

- As the method changes the category of $\boldsymbol{u}$ from the optimised variable to a state variable, this means that any possible bounds on $\boldsymbol{u}$ cannot longer be satisfied. Therefore,

a control constraint is traded versus a state constraint. As in the real world any control signal has bounds well defined, it remains only to hope that these will not be violated.

- If more control variables are candidates for state variables, no suitable selection strategy exists. In practice, usually a combination of control variables (not only one of them) influences the state constraint; the method is not capable to find it.
- It is implicitly given that the control variable can vary continuously in time. There are some optimisation problems where control has to be piece-wise constant – for example bang-bang type of control strategy where only time lengths can be varied.

On the other hand, the primary advantages are as follows:

- Only feasible solutions are generated by the integration (IVP – initial value problem) solver. This may particularly be important when the solutions that violate the path constraint would generate feasibility problems to IVP solver.
- The constraints are respected within the integration precision of the IVP solver and need not be relaxed for improved convergence of the optimisation.

2.3 Minimum Time Problems

Many dynamic optimisation tasks lead to the formulations that minimise the time. To express it in the form of general description (1.3), two possible approaches can be used.

The most straightforward is to define the cost function as

$$J_0 = G_0 = \Delta t_1 + \Delta t_2 + \cdots + \Delta t_P = \sum_{j=1}^{P} \Delta t_j \tag{2.17}$$

the corresponding gradients are therefore

$$\frac{\partial J_0}{\partial \Delta t_j} = 1 \tag{2.18}$$

Other possibility is to define a parameter $p_t = t_P$ and to describe the system differential equations with normalised time $\tau = t/p_t$, $\tau \in [0, 1]$ (assuming $t_0 = 0$ for simplicity). This yields

$$\frac{\mathrm{d}\boldsymbol{x}}{\mathrm{d}\tau} = p_t \boldsymbol{f}(t, \boldsymbol{x}, \boldsymbol{u}, \boldsymbol{p}) \tag{2.19}$$

This normalisation also influences the general cost description that is now written as

$$J_i = G_i(\tau_j, \boldsymbol{x}(1/P), \ldots, \boldsymbol{x}(1), \boldsymbol{u}(1/P), \ldots, \boldsymbol{u}(1), \boldsymbol{p}, p_t) + \int_0^1 F_i(t, \boldsymbol{x}, \boldsymbol{u}, \boldsymbol{p}, p_t)\mathrm{d}t \tag{2.20}$$

and the minimum time cost formulation is then simply

$$J_0 = G_0 = p_t \tag{2.21}$$

2.4 Periodic Problems

Sometimes, the dynamic optimisation problem is specified to find a periodic operational policy – that the process operates in quasi stationary state when the time-scale of the problem is enlarged. Consider for example bang-bang type of control, where the control variable can attain only low and high value. Of course, if stationary operation of the process is desired, such a control policy is undesirable. However, if stationarity is defined by constraining the process to be in some interval of states, bang-bang type of control is perfectly admissible.

Mathematically speaking, the periodic operation problem can be stated as: Find such an operating policy and such an initial states vector $\boldsymbol{x}_0 = \boldsymbol{p}$ so that the constraint

$$\boldsymbol{x}(t_P) = \boldsymbol{p} \tag{2.22}$$

is respected.

Although this can be rewritten to n_x equality constraints, it is not desirable, if the gradients are calculated via the solution of adjoint equations (as implemented in `DYNO`). Therefore, a suitable reformulation is given as

$$||\boldsymbol{x}(t_P) - \boldsymbol{p}||_2^2 = \sum_i \left(x_i(t_P) - p_i\right)^2 < \varepsilon, \quad \varepsilon > 0 \tag{2.23}$$

Chapter 3

Description of the Optimisation Method

3.1 Static Optimisation Problem

As the control trajectory is considered to be piece-wise constant, the original problem of dynamic optimisation has been converted into static optimisation – non-linear programming.

We then utilise static non-linear optimisation solver (NLP) of the form:

$$\begin{aligned} \min_{\boldsymbol{y}} J_0(\boldsymbol{y}) \quad & \text{subject to:} \\ & J_i(\boldsymbol{y}) = 0 \quad i = 1 \dots m_e \\ & J_i(\boldsymbol{y}) \geq 0 \quad i = m_e + 1 \dots m_i \end{aligned} \tag{3.1}$$

To obtain the values J_i, the system state equations and the integral functions have to be integrated from t_0 to t_P. In addition to the cost function and constraints, their gradients with respect to optimised variables $\boldsymbol{y}$ must be given. If they do not depend on the state variables $\boldsymbol{x}$, the gradients are obtained in the straightforward manner. In the opposite case, several methods can be utilised. We have implemented two methods: adjoint approach based on optimality conditions and finite differences.

3.2 Gradient Derivation

When the dynamic optimisation problem is to be solved, the nonlinear programming (NLP) solver needs to know gradients of the cost (and the constraints) with respect to the vector $\boldsymbol{y}$ of the optimised variables. One possibility to derive them consists in application of the theory of optimal control that specifies first order optimality conditions (Bryson and Ho, 1975). In the language of optimal control we search optimality conditions for a continuous systems with functions of state variables specified at unknown terminal and intermediate times.

Let us treat the problem of the functional J_i. The equation (1.1) is a constraint to the cost function J_i and is adjoined to the functional by a vector of non-determined adjoint variables $\boldsymbol{\lambda}_i(t) \in \mathcal{R}_{n_x}$, thus

$$J_i = G_i + \int_{t_0}^{t_P} (F_i + \boldsymbol{\lambda}_i^T(\boldsymbol{f} - \dot{\boldsymbol{x}}))\mathrm{d}t \tag{3.2}$$

For any J_i we can form a Hamiltonian H_i defined as

$$H_i(t, \boldsymbol{x}, \boldsymbol{u}, \boldsymbol{p}, \boldsymbol{\lambda}_i) = F_i + \boldsymbol{\lambda}_i^T \boldsymbol{f} \tag{3.3}$$

Substituting for F_i in (1.3) and dividing the interval to parts corresponding to optimised time intervals yields for J_i

$$J_i = G_i + \sum_{j=1}^{P} \int_{t_{j-1}^+}^{t_j^-} (H_i - \boldsymbol{\lambda}_i^T \dot{\boldsymbol{x}})\mathrm{d}t \tag{3.4}$$

where t_j^- signifies the time just before $t = t_j$ and t_j^+ is the time just after $t = t_j$. The last term in the integral can be integrated by parts, hence

$$J_i = G_i + \sum_{j=1}^{P} \int_{t_{j-1}^+}^{t_j^-} (H_i + \dot{\boldsymbol{\lambda}}_i^T \boldsymbol{x})\mathrm{d}t + \sum_{j=1}^{P} \boldsymbol{\lambda}_i^T(t_{j-1}^+)\boldsymbol{x}(t_{j-1}^+) - \boldsymbol{\lambda}_i^T(t_j^-)\boldsymbol{x}(t_j^-) \tag{3.5}$$

In order to derive the necessary optimality conditions, variation of the cost is to be found. The variation of the cost (see Bryson and Ho (1975)) is caused by the variation of the optimised variables $\delta t_j, \delta \boldsymbol{u}, \delta \boldsymbol{p}$ and by the variation of the variables that are functions of the optimised variables, i.e. $\delta \boldsymbol{x}, \delta \boldsymbol{\lambda}_i$. This gives

$$\begin{aligned}
\delta J_i = & \sum_{j=1}^{P} \frac{\partial G_i}{\partial \boldsymbol{x}^T(t_j)} \delta \boldsymbol{x}(t_j) + \sum_{j=1}^{P} \frac{\partial G_i}{\partial \boldsymbol{u}^T} \delta \boldsymbol{u}_j + \sum_{j=1}^{P} \frac{\partial G_i}{\partial t_j} \delta t_j + \frac{\partial G_i}{\partial \boldsymbol{p}^T} \delta \boldsymbol{p} \\
& - H_i(t_0^+)\delta t_0 + H_i(t_P^-)\delta t_P + \sum_{j=1}^{P-1} \left[H_i(t_j^-) - H_i(t_j^+)\right] \delta t_j \\
& + \boldsymbol{\lambda}_i^T(t_0^+)\delta \boldsymbol{x}(t_0) - \boldsymbol{\lambda}_i^T(t_P^-)\delta \boldsymbol{x}(t_P) + \sum_{j=1}^{P-1} \left[\boldsymbol{\lambda}_i^T(t_j^+) - \boldsymbol{\lambda}_i^T(t_j^-)\right] \delta \boldsymbol{x}(t_j) \\
& + \int_{t_0}^{t_P} \left[\left(\dot{\boldsymbol{\lambda}}_i^T + \frac{\partial H_i}{\partial \boldsymbol{x}^T} \right) \delta \boldsymbol{x} + \frac{\partial H_i}{\partial \boldsymbol{u}^T} \delta \boldsymbol{u} + \frac{\partial H_i}{\partial \boldsymbol{p}^T} \delta \boldsymbol{p} \right] \mathrm{d}t
\end{aligned} \tag{3.6}$$

where we have used the facts that $\delta \boldsymbol{\lambda}_i = \delta \dot{\boldsymbol{\lambda}}_i \delta t$ and that $\delta \boldsymbol{x}(t_j^-) = \delta \boldsymbol{x}(t_j^+)$ (continuity of states over the interval boundaries).

Regrouping the corresponding terms together, noting that $\delta t_0 = 0$ (t_0 is fixed), and $\delta \boldsymbol{x}(t_0) = (\partial \boldsymbol{x}_0 / \partial \boldsymbol{p}^T)\delta \boldsymbol{p}$ we get

$$\begin{aligned}
\delta J_i &= \left[\frac{\partial G_i}{\partial \boldsymbol{x}^T(t_P)} - \boldsymbol{\lambda}_i^T(t_P^-)\right]\delta \boldsymbol{x}(t_P) + \sum_{j=1}^{P-1}\left[\frac{\partial G_i}{\partial \boldsymbol{x}^T(t_j)} + \boldsymbol{\lambda}_i^T(t_j^+) - \boldsymbol{\lambda}_i^T(t_j^-)\right]\delta \boldsymbol{x}(t_j) \\
&\quad + \int_{t_0}^{t_P}\left(\dot{\boldsymbol{\lambda}}_i^T + \frac{\partial H_i}{\partial \boldsymbol{x}^T}\right)\delta \boldsymbol{x}\,\mathrm{d}t \\
&\quad + \left[H_i(t_P^-) + \frac{\partial G_i}{\partial t_P}\right]\delta t_P + \sum_{j=1}^{P-1}\left[H_i(t_j^-) - H_i(t_j^+) + \frac{\partial G_i}{\partial t_j}\right]\delta t_j \\
&\quad + \sum_{j=1}^{P}\left[\frac{\partial G_i}{\partial \boldsymbol{u}^T} + \int_{t_{j-1}^+}^{t_j^-}\frac{\partial H_i}{\partial \boldsymbol{u}^T}\mathrm{d}t\right]\delta \boldsymbol{u}_j \\
&\quad + \left[\frac{\partial G_i}{\partial \boldsymbol{p}^T} + \boldsymbol{\lambda}_i^T(t_0^+)\frac{\partial \boldsymbol{x}_0}{\partial \boldsymbol{p}^T} + \int_{t_0}^{t_P}\frac{\partial H_i}{\partial \boldsymbol{p}^T}\mathrm{d}t\right]\delta \boldsymbol{p} \qquad (3.7)
\end{aligned}$$

The vector $\boldsymbol{\lambda}_i$ is now chosen so as to cancel all terms containing the variation of the state vector $\delta \boldsymbol{x}$

$$\dot{\boldsymbol{\lambda}}_i^T = -\frac{\partial H_i}{\partial \boldsymbol{x}^T} \qquad (3.8)$$

$$\boldsymbol{\lambda}_i^T(t_P) = \frac{\partial G_i}{\partial \boldsymbol{x}^T(t_P)} \qquad (3.9)$$

$$\boldsymbol{\lambda}_i^T(t_j^-) = \boldsymbol{\lambda}_i^T(t_j^+) + \frac{\partial G_i}{\partial \boldsymbol{x}^T(t_j)}, \quad j = 1, \ldots, P-1 \qquad (3.10)$$

The variation of J_i can finally be expressed as

$$\begin{aligned}
\delta J_i &= \left[H_i(t_P^-) + \frac{\partial G_i}{\partial t_P}\right]\delta t_P + \sum_{j=1}^{P-1}\left[H_i(t_j^-) - H_i(t_j^+) + \frac{\partial G_i}{\partial t_j}\right]\delta t_j \\
&\quad + \sum_{j=1}^{P}\left[\frac{\partial G_i}{\partial \boldsymbol{u}^T} + \int_{t_{j-1}^+}^{t_j^-}\frac{\partial H_i}{\partial \boldsymbol{u}^T}\mathrm{d}t\right]\delta \boldsymbol{u}_j \\
&\quad + \left[\frac{\partial G_i}{\partial \boldsymbol{p}^T} + \boldsymbol{\lambda}_i^T(t_0^+)\frac{\partial \boldsymbol{x}_0}{\partial \boldsymbol{p}^T} + \int_{t_0}^{t_P}\frac{\partial H_i}{\partial \boldsymbol{p}^T}\mathrm{d}t\right]\delta \boldsymbol{p} \qquad (3.11)
\end{aligned}$$

The conditions of optimality follow directly from the last equation. As it is required, that the variation of the cost J_i should be zero at the optimum, all terms in brackets have to be zero.

3.2.1 Procedure

Assume that functions G_i, F_i and their partial derivatives with respect to $t_j, \boldsymbol{x}, \boldsymbol{u}, \boldsymbol{p}$ are specified. Also needed is the function $\boldsymbol{f}$ and its derivatives with respect to $\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{p}$.

The actual algorithm can briefly be given as follows :

1. Integrate the system (1.1) and integral terms F_i together from $t = t_0$ to $t = t_P$. Restart integration at discontinuities (beginning of the time intervals),

2. For $i = 0, \ldots, m$ repeat

   (a) Initialise adjoint variables $\boldsymbol{\lambda}_i(t_P)$ as

   $$\boldsymbol{\lambda}_i(t_P) = \frac{\partial G_i}{\partial \boldsymbol{x}^T(t_P)} \tag{3.12}$$

   (b) Initialise the intermediate variables $\boldsymbol{J}_u$, $\boldsymbol{J}_p$ as zero

   (c) Integrate backwards from $t = t_P$ to $t = t_0$ the adjoint system and intermediate variables. Allow for discontinuities of the adjoint equations as given in (3.10) and restart integration at these points

   $$\dot{\boldsymbol{\lambda}}_i^T = -\frac{\partial H_i}{\partial \boldsymbol{x}^T} \tag{3.13}$$

   $$\dot{\boldsymbol{J}}_{u,i}^T = \frac{\partial H_i}{\partial \boldsymbol{u}^T} \tag{3.14}$$

   $$\dot{\boldsymbol{J}}_{p,i}^T = \frac{\partial H_i}{\partial \boldsymbol{p}^T} \tag{3.15}$$

   (d) Calculate the gradients of J_i with respect to times t_j, control $\boldsymbol{u}$ and parameters $\boldsymbol{p}$

   $$\begin{aligned}
   \frac{\partial J_i}{\partial t_P} &= H_i(t_P^-) + \frac{\partial G_i}{\partial t_P} \\
   \frac{\partial J_i}{\partial t_j} &= H_i(t_j^-) - H_i(t_j^+) + \frac{\partial G_i}{\partial t_j}, \quad j = 1, \ldots, P-1 && (3.16) \\
   \frac{\partial J_i}{\partial \boldsymbol{p}} &= \frac{\partial G_i}{\partial \boldsymbol{p}^T} - \boldsymbol{J}_{p,i}(0) + \boldsymbol{\lambda}_i^T(t_0^+)\frac{\partial \boldsymbol{x}_0}{\partial \boldsymbol{p}^T} && (3.17) \\
   \frac{\partial J_i}{\partial \boldsymbol{u}_j} &= \boldsymbol{J}_{u,i}(t_{j-1}) - \boldsymbol{J}_{u,i}(t_j) && (3.18)
   \end{aligned}$$

In this manner, the values of J_i are obtained in the step 1 and the values of gradients in the step 2d. This is all what is needed as input to non-linear programming routines.

3.2.2 Notes

Gradients with respect to times

The expressions (3.16) for the calculation of the gradient of the cost with respect to time did not take into account that the time increments rather than times are optimised. The

relations between times and their increments are given as

$$\begin{aligned} t_1 &= \Delta t_1 \\ t_2 &= \Delta t_1 + \Delta t_2 \\ &\vdots \\ t_P &= \textstyle\sum_{j=1}^{P} \Delta t_j \end{aligned} \tag{3.19}$$

As the following holds for the derivatives

$$\frac{\partial J_i}{\partial \Delta t_j} = \sum_{k=1}^{P} \frac{\partial J_i}{\partial t_k} \frac{\partial t_k}{\partial \Delta t_j} \tag{3.20}$$

we finally get the desired expressions

$$\frac{\partial J_i}{\partial \Delta t_j} = \sum_{k=j}^{P} \frac{\partial J_i}{\partial t_k} \tag{3.21}$$

Integration of adjoint equations

When the adjoint equations are integrated backwards in time, the knowledge of states $\boldsymbol{x}(t)$ is needed. There are several ways to supply this information. For example, the state equations can be integrated together with adjoint equations backwards. Although this is certainly a correct approach, there may be numerical problems as the backward integration of states can be unstable. In Rosen and Luus (1991), the states are stored in equidistant intervals and integration of both states and adjoint equations is corrected at the begin of each interval. We have adopted another approaches : The first is to store the state vector every Δ time units in the forward pass and to interpolate states in backward pass. The drawback of this approach is large memory requirement. The second approach is to store at the forward pass only the states at the interval boundaries. In the backward pass to integrate the states at the respective time interval once again, again to store the states every Δ time units. The memory requirements are much smaller, but computational times longer. Several types of interpolation have been tested, the best results have been obtained with the approximations having continuous states and continuous first order derivatives across boundaries. Although the time needed for calculation of such approximations is longer, the adjoint equations are easier to integrate and the overall time of gradient calculations is greatly reduced.

It is always recommended to implement at least two methods of gradients calculation. In this manner, a user can cross-check if the gradients are correct. Also, if there is a problem in NLP algorithm, the gradient method can be changed.

Therefore, we have also implemented the method of finite differences: The system (1.1) is integrated q times and at each time one y_i is slightly perturbed. After the integrations, the gradients are given as

$$\nabla_{y_j} g_i = \frac{g_i(y_1, \ldots, \Delta y_j, \ldots, y_q) - g_i(\boldsymbol{y})}{\Delta y_j}, \quad i = 0, \ldots, m \tag{3.22}$$

Chapter 4

Specification of the subroutines

The package DYNO has to communicate with several subroutines. In addition to the principal routine dyno, the subroutines containing information about the process optimised (process), cost and constraints (costi, costni) have to be specified. Finally, user has to choose among the NLP and IVP solvers supported and decide whether partial derivatives of functions will be given by automatic differentiation software or manually.

4.1 Subroutine dyno

The package DYNO exists in double precision version. The calling syntax is as follows:

```
      subroutine DYNO(nsta, ncont, nmcont, npar, nmpar, ntime, ncst,
     &      ncste, ul, u, uu,pl, p, pu, tl, t, tu, ista, rw, nrw, iw,
     &      niw, lw, nlw,rpar, ipar, ifail, infou)
      implicit none
      integer nsta, ncont, nmcont, npar, nmpar, ntime, ncst, ncste, ista
     &      , nrw, iw, niw,nlw, ipar, ifail, infou
      logical lw
      double precision ul, u, uu, pl, p, pu, tl, t, tu, rw, rpar
      dimension ul(nmcont, ntime), u(nmcont, ntime), uu(nmcont, ntime)
      dimension pl(nmpar), p(nmpar), pu(nmpar)
      dimension tl(ntime), t(ntime), tu(ntime)
      dimension ista(ncst+1)
      dimension rw(nrw), iw(niw), lw(nlw), rpar(*), ipar(*), infou(16)
```

The subroutine call specifies the problem dimensions, bounds on the optimised variables, and various parameters influencing the behaviour of the routine.

nsta – **Input** Number of state variables (state dimension).

ncont – **Input** Number of control variables.

`nmcont` – **Input** Dimension of control variables. Must be $\max(1, \texttt{ncont})$

`npar` – **Input** Number of time independent parameters.

`nmpar` – **Input** Dimension of time independent parameters. Must be $\max(1, \texttt{npar})$

`ntime` – **Input** Number of time intervals. Denoted by P in the text.

`ncst` – **Input** Total number of constraints (equality + inequality). Simple (lower, upper) bounds on optimised variables do not count here. Denoted by $m = m_i + m_e$ in the text.

`ncste` – **Input** Total number of equality constraints. Denoted by m_e in the text.

`ul(nmcont, ntime)` – **Input** Minimum control bounds in each time interval.

`u(nmcont, ntime)`

Input Initial estimate of the control trajectory.

Output Final estimate of the optimal control trajectory.

`uu(nmcont, ntime)` – **Input** Maximum control bounds in each time interval.

`pl(nmpar)` – **Input** Minimum parameter bounds.

`p(nmpar)`

Input Initial estimate of the parameters.

Output Final estimate of the optimal parameters.

`pu(nmpar)` – **Input** Maximum parameter bounds.

`tl(ntime)` – **Input** Minimum time interval bounds.

`t(ntime)`

Input Initial estimate of time intervals.

Output Final estimate of the optimal time intervals.

`tu(ntime)` – **Input** Maximum time interval bounds.

`ista(ncst)` – **Input** Characterisation of each constraints/cost J_i. As only constraints containing state variables have to be integrated backwards each constraint is specified as follows: 0 - does not depend on states, only directly on optimised variables (for example $u_1 u_3 + u_2 = 1$), 1 - depends on states, but it is not integral path state constraint, 3 - integral path constraint that needs to be integrated very carefully with small steps.

`rw(nrw)` Double precision workspace.

Input The first 5 components should be either zero or positive. The number in parentheses indicates default value used if the corresponding entry is zero.

`rw(1)` Relative tolerance of the integrator (= 1d-7).

`rw(2)` Absolute tolerance of the integrator (= 1d-7).

`rw(3)` Tolerance of the NLP solver (= 1d-5).

`rw(4)` Minimum relative tolerance of the IVP/NLP solvers if `info(9)` > 0 (= 1d-3).

`rw(5)` starting time $t_0 \geq 0$ for the simulation (= 0.0d0).

Output See file `work.txt` for full description of the workspace organisation.

`nrw` – Dimension of the double precision workspace `rw`.

Input Sufficiently large value and at least 10. If not enough, the program writes error message with the minimum necessary value.

Output If the package is called with `ifail=-3` flag, it returns minimum needed value.

`iw(niw)` Integer workspace.

Input Nothing required.

Output See file `work.txt` for full description of the workspace organisation.

`niw` – Dimension of the integer workspace `iw`.

Input Sufficiently large value and at least 120. If not enough, the program writes error message with the minimum necessary value.

Output If the package is called with `ifail=-3` flag, it returns minimum needed value.

`lw(nlw)` Logical workspace.

Input Nothing required.

Output See file `work.txt` for full description of the workspace organisation.

`nlw` – Dimension of the logical workspace `lw`.

Input Sufficiently large value and at least `2*max(ncst,1)+15`. If not enough, the program writes error message with the minimum necessary value.

Output If the package is called with `ifail=-3` flag, it returns minimum needed value.

`ipar(*)` – **Input/Output** User defined integer parameter vector that is not changed by the package. Can be used in communication among the user subroutines.

`rpar(*)` – **Input/Output** User defined double precision parameter vector that is not changed by the package. Can be used in communication among the user subroutines. Should not be a function of optimised variables $\Delta t_i, \boldsymbol{u}_i, \boldsymbol{p}$ or $\boldsymbol{x}$.

`ifail` – Communication with `DYNO`.

Input

-3 Find minimum needed workspace allocation and return it in `nrw` , `nil`, `nlw`.

-2 Check gradients with finite difference technique.

-1 Simulate the process based on the actual values of optimised variables.

0 Perform optimisation.

Output Zero if optimum has been found. Other values indicate failure specified more precisely in the used NLP solver.

`info(16)` – **Input** The first 16 components should be either zero or positive. The number in parentheses indicates default value used if the corresponding entry is zero.

`info(1)` Maximum number of function call evaluations in NLP/line search (=40).

`info(2)` Maximum number of iterations in NLP (=999).

`info(3)` Level of information printed by the subroutine (=2). Possible values are 0 - prints nothing, 1 - single line per NLP iteration, 2 - as 1 with additional information, 3 - very detailed informations, also with initial and final trajectory simulation, 4 - added gradient checks.

`info(4)` Number of the output routine (= 6).

`info(5)` Maximum number of time instants on one time interval when state is to be saved (=200).

`info(6)` Method of state interpolation when integrating adjoint equations (=3). 0 - none(left one), 1 - linear, 2 - polynomial of the second order with continuous derivative at the beginning and continuous state on both boundaries, 3 - polynomial of the third order with continuous derivative and state on both boundaries.

`info(7)` Choice of the optimised variables (= 2): 1 - times, 2 - control, 4 - parameters. Their combination is given by their summations, e.g. 3 - optimise times and control, 7 - optimise all.

`info(8)` Gradients via 0 - adjoint equations, 1 - finite differences (= 0).

`info(9)` Choice of the optimising strategy (= 0).

0 Standard one pass optimisation.

1 Mesh refining with multiple pass optimisation. Start with only a minimum number of `info(10)` time intervals and minimum precision `rw(4)`. The optimum found improve by adding some new time intervals where the initial values at the new intervals are optimal from the preceding iteration. The precision is tightened. Thus the first solution is obtained very quickly as both IVP and NLP solvers work with reduced precisions. Repeat for `info(11)` times until the number of intervals is `ntime`.

2 Multirate multipass optimisation. Optimise with only a minimum number of `info(10)` time intervals. The optimal control trajectory found is fixed in the second half and the first half is again optimised with `info(10)` intervals. Repeat `info(11)` times. The precision is variable as in the previous case.

3 Control horizon N_u implementation as in predictive control. Standard one pass optimisation with only `info(10)` time intervals. Control and times in the last `ntime-info(10)` intervals are the same as the in the last optimised segment. (Alternatively, the same effect can be obtained with `info(9)=0`, `info(10)=Nu`.)

`info(10)` Starting number of time intervals for `info(9).ge 0` (= `ntime`).

`info(11)` Number of master NLP problems for `info(9)=1,2` (= 1).

`info(12)` Periodicity (= 1). Whether some parts of the optimal solution should be repeatedly used. Choose `info(9)=0, info(10)=1`. Then only `info(12)` elements will be optimised and repeated over `ntime/per` intervals. Examples:

`ntime`=8, `per`=2, `ntmin`=1: t1,t2,t1,t2,t1,t2,t1,t2 (and correspondingly u1, u2, ...)

`ntime`=8, `per`=4, `ntmin`=1: t1,t2,t3,t4,t1,t2,t3,t4.

`ntime`=8, `per`=2, `ntmin`=4: t1,t2,t1,t2,t3,t4,t3,t4

`info(13)` Save state trajectory (= 0): 0 - in the forward pass, 1 - re-integrate each segment in the backward pass. Forward pass is faster but has to save the whole state simulation trajectory - needs very much memory. Backward pass saves the states only on the actual time interval as thus needs less memory. On the other hand, it integrates the states twice.

`info(14)` Choice of the integration (IVP) solver(= 0). Currently implemented are: 0 - VODE, 1 - DDASSL. Usually VODE is about 30% faster.

`info(15)` Choice of the NLP solver (= 0). Currently implemented are: 0 - SLSQP, 1 - NLPQL. SLSQP is public domain code taken from www.netlib.org, NLPQL is commercial (Schittkowski, 1985)[1].

`info(16)` Generation of Jacobians manually or with automatic differentiation tools (= 0). Currently implemented are: 0 - manually, 1 - ADIFOR. More information to ADIFOR at http://www-unix.mcs.anl.gov/autodiff/ADIFOR.

Negative values indicate that some of the partial derivatives are coded manually:

[1] NLPQL cannot be included with the package

-1 df/dx
-2 df/du
-4 df/dp
-8 dx0/dp
-16 dG/dxupt
-32 dF/dxup

Thus, if for example value -6 has been specified then df/du, df/dp are coded manually, others are given by automatic differentiation.

Example 2 (Example 1 continued)**.** For the problem defined by the equation (1.8), the dimensions are as follows: dimension of states (2), dimension of control (1), dimension of parameters (0), number of time intervals (3), number of constraints (2), and from it number of equality constraints (2). Sufficient dimensions of the workspace have been found as `niw=400`, `nrw=60000`, `nlw=50`.

The cost does not depend on states and the constraints depend on states. Thus the `ista` vector is given as (0,1,1).

The corresponding file is shown below.

```
 PROGRAM EXM1
 implicit none
 integer nmcont, nmpar
 integer nsta, ncont, npar, ntime, ncst, ncste
 parameter (nsta = 2, ncont = 1, npar = 0, ntime = 3,
&      ncst = 2, ncste = 2)
 parameter (nmcont = ncont, nmpar = 1)
 integer ista, nrwork, iwork, niwork, nlwork, ipar, ifail, info
 double precision ul, u, uu, pl, p, pu, tl, t, tu, rwork, rpar
 logical lwork

 dimension ul(nmcont, ntime), u(nmcont, ntime), uu(nmcont, ntime)
 dimension pl(nmpar), p(nmpar), pu(nmpar)
 dimension tl(ntime), t(ntime), tu(ntime)
 dimension ista(ncst+1)

 parameter (niwork=400, nrwork=60000, nlwork = 50)
 dimension iwork(niwork), rwork(nrwork), lwork(nlwork)
 dimension ipar(10), rpar(50), info(16)

 integer i
 ista(1) = 0
 ista(2) = 1
 ista(3) = 1
```

```
      do i = 1, 5
         rwork(i) = 0.0d0
      end do

      do i = 1, 16
         info(i) = 0
      end do

c     optimise what: 0/1-ti, 0/2-ui, 0/4-p
      info(7) = 3

c     initial values of the optimised parameters
c     upper, lower bounds
c     control and time
      do i=1,  ntime
         u(1,i) = 1.0d0
         ul(1,i) = -0.50d0
         uu(1,i) = 1.50d0
         t(i)  = 1.0d0
         tl(i) = 0.01d0
         tu(i) = 10.0d0
      end do

      ifail = 0
      call DYNO(nsta, ncont, nmcont, npar, nmpar, ntime, ncst, ncste, ul
     &      , u, uu, pl, p,pu, tl, t, tu, ista, rwork, nrwork, iwork,
     &      niwork, lwork, nlwork, rpar, ipar, ifail, info)
      call trawri(rwork, iwork, rpar, ipar)
      END
```

4.2 Subroutine process

This subroutine specifies differential equations of the process as well as its initial conditions. In addition, various partial derivatives of the both are required. The name of the subroutine `process` is currently fixed and cannot be specified via keyword `external`.

```
      subroutine process(t, x, nsta, u, ncont, nmcont, p, npar, nmpar,
     &      sys, nsys, dsys, ndsys1, ndsys2, ipar, rpar, flag, iout)
      implicit none
      integer nsta, ncont, nmcont, npar, nmpar, nsys, ndsys1, ndsys2,
```

```
&      ipar, flag, iout
 double precision t, x, u, p, sys, dsys, rpar
 dimension x(nsta), u(nmcont), p(nmpar),
&      sys(nsys), dsys(ndsys1, ndsys2), rpar(*), ipar(*)
```

The subroutine accomplishes various tasks based on the integer `flag`. It takes the values of the input arguments (state, control, parameters at actual time) and returns their evaluation in either vector `sys` or matrix `dsys`. When `dsys` is to be returned, the real dimensions `ndsys1, ndsys2` provided by the calling routine are usually larger than desired. Especially the leading dimension `ndsys1` is important when passing `dsys` as a parameter to other subroutines.

`t` – **Input** Actual time.

`x(nsta)` – **Input** State values at actual time.

`nsta` – **Input** Number of state variables (state dimension).

`u(nmcont)` – **Input** Control values at actual time.

`ncont` – **Input** Number of control variables.

`nmcont` – **Input** Dimension of control variables. Must be $\max(1, \texttt{ncont})$

`p(nmpar)` – **Input** Parameter values.

`npar` – **Input** Number of time independent parameters.

`nmpar` – **Input** Dimension of time independent parameters. Must be $\max(1, \texttt{npar})$

`sys(nsys)` – **Output** One dimensional output – vector (see `flag` for explanation).

`nsys` – **Input** Dimension of `sys`. Depends on `flag`.

`dsys(ndsys1,ndsys2)` – **Output** Two dimensional output – matrix (see `flag` for explanation).

`ndsys1,ndsys2` – **Input** Dimensions of `dsys`. Depends on `flag`.

`ipar(*)` – **Input/Output** User defined integer parameter vector that is not changed by the package. Can be used in communication among the user subroutines.

`rpar(*)` – **Input/Output** User defined double precision parameter vector that is not changed by the package. Can be used in communication among the user subroutines. Should not be a function of optimised variables $\Delta t_i, \boldsymbol{u}_i, \boldsymbol{p}$ or $\boldsymbol{x}$.

`iout` – **Input** Number of the output routine. To be used with `flag=-3`.

`flag` – **Input** Decision which information the subroutine has to provide.

0 Right hand sides of the differential equations.
Output/Vector: `sys(nsta)` $= \boldsymbol{f}(t, \boldsymbol{x}, \boldsymbol{u}, \boldsymbol{p})$.

1 Partial derivatives of the state equations with respect to states.
Output/Matrix: `dsys(nsta,nsta)` $= \partial \boldsymbol{f} / \partial \boldsymbol{x}^T$
Note, that the true dimensions `dsys(ndsys1, ndsys2)` usually differ from `nsta`. Therefore, if this matrix is to be returned by another subroutine, always pass the true dimensions.

2 Partial derivatives of the state equations with respect to control.
Output/Matrix: `dsys(nsta,nmcont)` $= \partial \boldsymbol{f} / \partial \boldsymbol{u}^T$
See also the note for `flag=1`

3 Partial derivatives of the state equations with respect to parameters.
Output/Matrix: `dsys(nsta,nmpar)` $= \partial \boldsymbol{f} / \partial \boldsymbol{p}^T$
See also the note for `flag=1`

-1 Initial conditions.
Output/Vector: `sys(nsta)` $= \boldsymbol{x}(t_0, \boldsymbol{p})$

-2 Partial derivatives of the initial conditions with respect to parameters.
Output/Matrix: `dsys(nsta,nmpar)` $= \partial \boldsymbol{x}_0 / \partial \boldsymbol{p}^T$
See also the note for `flag=1`

-3 Define output to be printed (on one line) at time t. Besides of states, also the values of integral cost functions are available as `x(nsta+1,..,nsta+ncst+1)`.

Example 3 (Example 1 continued)**.** For the problem defined by the equation (1.8), the resulting vectors and matrices are given as:

$$\boldsymbol{f} = \begin{pmatrix} x_2 \\ u - x_1 - 2x_2 \end{pmatrix}, \quad \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}^T} = \begin{pmatrix} 0 & 1 \\ -1 & -2 \end{pmatrix}, \quad \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{u}^T} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \quad \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{p}^T} = \boldsymbol{0} \tag{4.1}$$

$$\boldsymbol{x}_0 = \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \quad \frac{\partial \boldsymbol{x}_0}{\partial \boldsymbol{p}^T} = \boldsymbol{0} \tag{4.2}$$

The corresponding file is shown below. Note, that for all matrices of partial derivatives only non-zero elements are to be specified.

```
 subroutine process(t, x, nsta, u, ncont, nmcont, p, npar, nmpar,
&      sys, nsys, dsys, ndsys1, ndsys2, ipar, rpar, flag, iout)
 implicit none
 integer nsta, ncont, nmcont, npar, nmpar, nsys, ndsys1, ndsys2,
&      ipar, flag, iout
 double precision t, x, u, p, sys, dsys, rpar
```

```
      dimension x(nsta), u(nmcont), p(nmpar),
     &      sys(nsys), dsys(ndsys1, ndsys2), rpar(*), ipar(*)

      if (flag .eq. 0) then
         sys(1) = x(2)
         sys(2) = u(1)-2*x(2)-x(1)
         return
      end if

      if (flag .eq. 1) then
         dsys(1,2) = 1.0d0
         dsys(2,1) = -1.0d0
         dsys(2,2) = -2.0d0
         return
      end if
      if (flag .eq. 2) then
         dsys(2,1) = 1.0d0
         return
      end if
      if (flag .eq. -1) then
         sys(1) = 0.0d0
         sys(2) = 0.0d0
         return
      end if
      if (flag .eq. -2) then
         return
      end if
      if (flag .eq. -3) then
         write(iout,100) t, u(1), x(1), x(2)
  100    format(f8.5,3X,3f20.15)
         return
      end if
      end
```

4.3 Subroutine `costi` of Integral Constraints

The first part of the constraints is specified by the terms involved in the integrals. These can be functions of states, control, and parameters.

The cost/constraints have to be ordered with the equality constraints first and positive inequality constraints next. In addition to cost/constraints, partial derivatives with respect

to states, control, and parameters are required. The name of the subroutine `costi` is currently fixed and cannot be specified via keyword `external`.

```
      subroutine costi(t, x, u, p, ntime, nsta, ncont, nmcont, npar,
     &      nmpar, ncst, ncste, ipar,rpar, flag, xupt, nti, sys, nsys)
      implicit none
      integer ntime, nsta, ncont, nmcont, npar, nmpar, ncst, ncste, ipar
     &      , flag, xupt,nti, nsys
      double precision t, x, u, p, rpar, sys
      dimension x(nsta), u(nmcont), p(nmpar), ipar(*), rpar(*), sys(nsys
     &      )
```

The subroutine accomplishes various tasks based on the integers `flag, nti, xupt`. It takes the values of the input arguments (time, states, control, parameters) and returns their evaluations in vector `sys(nsys)`.

`t` – **Input** Actal time.

`x(nsta)` – **Input** State values at actual time.

`u(nmcont)` – **Input** Control values at actual time.

`p(nmpar)` – **Input** Parameter values.

`ntime` – **Input** Number of time intervals.

`nsta` – **Input** Number of state variables (state dimension).

`ncont` – **Input** Number of control variables.

`nmcont` – **Input** Dimension of control variables. Must be $\max(1, \texttt{ncont})$

`npar` – **Input** Number of time independent parameters.

`nmpar` – **Input** Dimension of time independent parameters. Must be $\max(1, \texttt{npar})$

`ncst` – **Input** Total number of constraints (equality + inequality). Simple (lower, upper) bounds on optimised variables do not count here.

`ncste` – **Input** Total number of equality constraints.

`ipar(*)` – **Input/Output** User defined integer parameter vector that is not changed by the package. Can be used in communication among the user subroutines.

`rpar(*)` – **Input/Output** User defined double precision parameter vector that is not changed by the package. Can be used in communication among the user subroutines. Should not be a function of optimised variables $\Delta t_i, \boldsymbol{u}_i, \boldsymbol{p}$ or $\boldsymbol{x}$.

`sys(nsys)` – **Output** One dimensional output – vector (see `flag` for explanation).

`nsys` – **Input** Dimension of `sys`. Depends on `flag`.

`nti` – **Input** Which time interval is actually treated.

`xupt` – **Input** Decision which information has to be provided (see `flag` for explanation).

`flag` – **Input** Decision which information the subroutine has to provide:

-1 Integral term of the cost and constraints.
Output/Vector `sys(ncst + 1)` $= (F_0\ F_1\ \ldots\ F_m)^T$.

0 Partial derivatives of the cost with respect to states/control/parameters:
Output/Vector: `xupt=1`, `sys(nsta)` $= \partial \boldsymbol{F_0}/\partial \boldsymbol{x}^T$.
Output/Vector: `xupt=2`, `sys(nmcont)` $= \partial \boldsymbol{F_0}/\partial \boldsymbol{u}^T$.
Output/Vector: `xupt=3`, `sys(nmpar)` $= \partial \boldsymbol{F_0}/\partial \boldsymbol{p}^T$.

$i = 1, \texttt{ncst}$ Partial derivatives of constraint i with respect to states/control/parameters:
Output/Vector: `xupt=1`, `sys(nsta)` $= \partial \boldsymbol{F_i}/\partial \boldsymbol{x}^T$.
Output/Vector: `xupt=2`, `sys(nmcont)` $= \partial \boldsymbol{F_i}/\partial \boldsymbol{u}^T$.
Output/Vector: `xupt=3`, `sys(nmpar)` $= \partial \boldsymbol{F_i}/\partial \boldsymbol{p}^T$.

Example 4 (Example 1 continued)**.** For the problem defined by the equation (1.8), all integral functions are zero.

The corresponding file is shown below. Note, that for all matrices of partial derivatives only non-zero elements are to be specified.

```
 subroutine costi(t, x, u, p, ntime, nsta, ncont, nmcont, npar,
&      nmpar, ncst, ncste, ipar,rpar, flag, xupt, nti, sys, nsys)
 implicit none
 integer ntime, nsta, ncont, nmcont, npar, nmpar, ncst, ncste, ipar
&      , flag, xupt,nti, nsys
 double precision t, x, u, p, rpar, sys
 dimension x(nsta), u(nmcont), p(nmpar), ipar(*), rpar(*), sys(nsys
&      )

 if (flag .eq. -1) then
    sys(1) = 0.0d0
    sys(2) = 0.0d0
    sys(3) = 0.0d0
    return
 end if
 end
```

4.4 Subroutine `costni` of Non-integral Constraints

The second part of the constraints is specified by the terms not involved in the integrals. These can be functions of time intervals, any element of the control matrix trajectory, parameters, as well as the states defined at the end of any time interval. This can give quite a large number of terms and the correct indices have to be carefully verified.

Recall that these cost/constraints have to be ordered as the integrals are. In addition to cost/constraints, partial derivatives with respect to states, control, and parameters are required. The name of the subroutine `costni` is currently fixed and cannot be specified via keyword `external`.

```
      subroutine costni(ti, xi, ui, p, ntime, nsta, ncont, nmcont, npar,
     &      nmpar, ncst,ncste,ipar, rpar, flag, sys, n1, n2, n3)
      implicit none
      integer ntime, nsta, ncont, nmcont, npar, nmpar, ncst, ncste, ipar
     &      , flag, n1, n2,n3
      double precision ti, xi, ui, p, rpar, sys
      dimension ti(ntime), xi(nsta, ntime), ui(nmcont, ntime), p(nmpar),
     &      ipar(*), rpar(*), sys(n1,n2,n3)
```

The subroutine accomplishes various tasks based on the integer `flag`. It takes the values of the input arguments (time interval vector, state trajectory matrix at any t_i, control trajectory matrix, parameter vector) and returns their evaluations in tensor `sys(n1,n2,n3)`. When `sys` is to be passed to other subroutines, always pass the real dimensions along as these may differ (be larger) than the actual ones.

`ti(ntime)` – **Input** Vector of time intervals.

`xi(nsta,ntime)` – **Input** Matrix of state trajectories at end of each time interval. For example `xi(i,ntime)` represents state x_i at the final time, `xi(2,3)` represents second element of state at the end of the third time interval.

`ui(nmcont,ntime)` – **Input** Matrix of control trajectory.

`p(nmpar)` – **Input** Parameter values.

`ntime` – **Input** Number of time intervals.

`nsta` – **Input** Number of state variables (state dimension).

`ncont` – **Input** Number of control variables.

`nmcont` – **Input** Dimension of control variables. Must be $\max(1, \texttt{ncont})$

`npar` – **Input** Number of time independent parameters.

`nmpar` – **Input** Dimension of time independent parameters. Must be $\max(1, \texttt{npar})$

`ncst` – **Input** Total number of constraints (equality + inequality). Simple (lower, upper) bounds on optimised variables do not count here.

`ncste` – **Input** Total number of equality constraints.

`ipar(*)` – **Input/Output** User defined integer parameter vector that is not changed by the package. Can be used in communication among the user subroutines.

`rpar(*)` – **Input/Output** User defined double precision parameter vector that is not changed by the package. Can be used in communication among the user subroutines. Should not be a function of optimised variables $\Delta t_i, \boldsymbol{u}_i, \boldsymbol{p}$ or $\boldsymbol{x}$.

`sys(nsys)` – **Output** Tensor of output of the routine (see `flag` for explanation).

`n1, n2, n3` – **Input** Dimensions of `sys`. Depend on `flag`.

`flag` – **Input** Decision which information the subroutine has to provide.

0 Non-integral term of the cost and constraints.
Output: $i = 1 \ldots$`ncst` $+ 1$, $j = 1$, $k = 1$.
`sys(i,j,k)` $= (G_0\ G_1\ \ldots\ G_m)^T$.

1 Partial derivatives of all costs with respect to states:
Output: $i = 1 \ldots$`nsta`, $j = 1 \ldots$`ntime`, $k = 1 \ldots$`ncst` $+ 1$.
`sys(i,j,k)` $= \partial G(k)/\partial x(i,j)$.

2 Partial derivatives of all costs with respect to control:
Output: $i = 1 \ldots$`nmcont`, $j = 1 \ldots$`ntime`, $k = 1 \ldots$`ncst` $+ 1$.
`sys(i,j,k)` $= \partial G(k)/\partial u(i,j)$.

3 Partial derivatives of all costs with respect to parameters:
Output: $i = 1 \ldots$`nmpar`, $j = 1 \ldots$`ncst` $+ 1$, $k = 1$.
`sys(i,j,k)` $= \partial G(j)/\partial p(i)$.

4 Partial derivatives of all costs with respect to time intervals:
Output: $i = 1 \ldots$`ntime`, $j = 1 \ldots$`ncst` $+ 1$, $k = 1$.
`sys(i,j,k)` $= \partial G(j)/\partial \Delta t_i$.

Example 5 (Example 1 continued)**.** For the problem defined by the equation (1.8), the non-integral functions are defined as

$$\boldsymbol{G} = \begin{pmatrix} \Delta t_1 + \Delta t_2 + \Delta t_3 \\ x_1(t_3) - 1 \\ x_2(t_3) \end{pmatrix} \tag{4.3}$$

$$\frac{\partial \boldsymbol{G}}{\partial \boldsymbol{x}^T(t_1)} = \frac{\partial \boldsymbol{G}}{\partial \boldsymbol{x}^T(t_2)} = \boldsymbol{0}, \quad \frac{\partial \boldsymbol{G}}{\partial \boldsymbol{x}^T(t_3)} = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \tag{4.4}$$

$$\frac{\partial \boldsymbol{G}}{\partial \Delta \boldsymbol{t}^T} = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \tag{4.5}$$

The corresponding file is shown below. Note, that for all matrices of partial derivatives only non-zero elements are to be specified.

```
 subroutine costni(ti, xi, ui, p, ntime, nsta, ncont, nmcont, npar,
&      nmpar, ncst,ncste,ipar, rpar, flag, sys, n1, n2, n3)
 implicit none
 integer ntime, nsta, ncont, nmcont, npar, nmpar, ncst, ncste, ipar
&      , flag, n1, n2,n3
 double precision ti, xi, ui, p, rpar, sys
 dimension ti(ntime), xi(nsta, ntime), ui(nmcont, ntime), p(nmpar),
&      ipar(*), rpar(*), sys(n1,n2,n3)

 if (flag .eq. 0 ) then
    sys(1,1,1) = ti(1)+ti(2)+ti(3)
    sys(2,1,1)= xi(1,ntime)-1.0d0
    sys(3,1,1)= xi(2,ntime)
    return
 end if
 if (flag .eq. 1 ) then
    sys(1,ntime,2) = 1.0d0
    sys(2,ntime,3) = 1.0d0
    return
 end if
 if (flag .eq. 2 ) then
    return
 end if
 if (flag .eq. 3 ) then
    return
 end if
 if (flag .eq. 4 ) then
       sys(1,1,1) = 1.0d0
       sys(2,1,1) = 1.0d0
       sys(3,1,1) = 1.0d0
    return
 end if
 end
```

4.5 Organisation of Files and Subroutines

4.5.1 Files

The user has to provide the principal routine that calls `dyno` as well as routines for process and cost descriptions. In the examples, it is usually assumed that the calling routine resides in `amain.f`, process is specified by `process.f` and the cost description is given in `cost.f` (both subroutines).

With the above files serving as the user information, the `DYNO` package consists of several modules that can be combined together according to the needs of the particular problem:

`dyno.f` – main part of the code, needs always to be included.

IVP solvers – currently the following integration routines are supported. One of them should be included in the project and the entry in `info(14)` should be set correctly.

`vodedo.f` – VODE solver (Brown et al., 1989) modified to pass internal `DYNO` workspace, replaced LINPACK calls by LAPACK calls, removed BLAS routines.

`ddassldo.f` – DDASSL solver (Brenan et al., 1989) modified to pass internal `DYNO` workspace, removed BLAS routines.

The modified files also include interface routines from `dyno`.

NLP solvers – currently the following NLP routines are supported. One of them should be included in the project and the entry in `info(15)` should be set correctly.

`slsqp.f` – Public domain solver (Kraft, 1988). The code remains unchanged, only BLAS routines have been removed.

`nlpql.f` – Software NLPQL (Schittkowski, 1985) that has to be bought directly from the author. However, the file `nlpql.f` used here has been modified with interface to `DYNO`.

Automatic differentiation Currently supported is ADIFOR (Bischof et al., 1998). The user should either to include to the project the file `adifno.f` if all partial derivatives are coded manually, or file `adifyes.f` that contains interface to automatically generated files by ADIFOR. The actual organisation with AD tools is explained later in section 4.5.4.

In future, other AD tools may be implemented in the similar way.

LAPACK The code makes heavy use of BLAS and LAPACK routines (available from NETLIB). As the routines are de facto standard, they are usually installed locally in optimised version for the given processor and can be linked directly when creating executables. If it is not the case, the file `blalap.f` contains all (unoptimised) needed routines and has to be added to the project.

With the default DYNO settings for IVP, NLP solvers and AD tools, a possible project includes files amain.f, process.f, cost.f, dyno.f, vodedo.f, slsqp.f, adifno.f, and blalap.f.

4.5.2 Subroutines in dyno.f

The principal flow of information when the routine dyno is called is as follows:

dyno – Initialises workspace, allocates space for the workspace vectors (initdo2), and copies information from the calling routine into the internal structures (initdo3). Depending on the value of the flag, gradients are checked (chkgrd), initial trajectory is simulated (trawri) or the main routine (nlp) is invoked.

nlp – Initialises some pointers to the workspace and calls workhouse routine (nlpw).

nlpw – Performs main loop of optimisation. Calls the corresponding NLP solver via nlpslv. According to its output, it can request evaluation of the cost/constraints (trasta), evaluation of gradients (either findif or tralam), or returns with status ifail. This routine also performs various specialised tasks described by the value of info(9). The NLP solver then obtains transformed problem containing only real optimised variables. The conversions between are realised with subroutines vartox and xtovar. The choice of times and control segments that are optimised is determined by the integer vector indopt.

Integration of system equations

trasta – Principal routine for system simulation and evaluation of the cost and constraints. Initialises pointers and calls workhouse routine trastaw.

trastaw – Integrates the optimised system state trajectory together with integral costs F_i. Calls stepsta for each time interval.

stepsta – Integrates at one time interval, saves intermediate states and their derivatives, and eventually makes calls to print the trajectory. Calls the appropriate IVP solver interface stepstaw. The states are saved in the instants when IVP solver returns, not at regular intervals. The state matrix trajectory thus contains also the corresponding times so that it is possible to interpolate.

fsta – Calculates right hand sides of the system differential equations (together with integral terms F_i at given time t.

jacsta – Calculates the corresponding Jacobian matrix.

Gradient calculation

tralam – Principal routine for adjoint system simulation and evaluation of the gradients. Initialises pointers and calls workhouse routine tralamw.

`trastaw` – Integrates the adjoint trajectory from t_P to t_0 together with the Hamiltonian terms. Calculates the gradients and calls `steplam` for each time interval.

`steplam` – Integrates at one time interval. Calls the appropriate IVP solver interface `steplamw`.

`flam` – Calculates right hand sides of the adjoint system of differential equations (together with Hamiltonian terms) at given time t. States are approximated via call to `actstates`.

`jaclam` – Calculates the corresponding Jacobian matrix.

`actstates` – The routine `flam` needs to know actual states. They are approximated here from the saved state trajectory matrix and eventually also from the saved state derivative trajectory matrix.

`findif` – Calculates gradient information by the method of forward finite differences. The value of the perturbation depends on the chosen integration tolerances.

`chkgrd` – Calculates gradient information by the method of adjoint variables and by finite differences. Print results for elements that are nor similar.

`trawri` – Simulates and prints the actual trajectories.

`dynodump` – Dumps out content of workspace for debugging purposes.

4.5.3 Reserved common blocks

The package does not introduce any common blocks. All information transfer is done by the workspace vectors with their organisation described by the file `work.txt`. The (NLP, IVP) solvers introduce their own common blocks, see their documentation for further details.

Example 6 (Example 1 – results)**.** For the problem defined by the equation (1.8), the files have been compiled on a PC with GNU/Linux operating system. The optimum values have been obtained as follows:

$$\Delta t_j^T = (1.0965, 1.0965, 0.200) \tag{4.6}$$

$$u_j^T = (1.500, 1.500, -0.500) \tag{4.7}$$

$$J_i^T = (2.393, -8.055\,10^{-11}, -3.014\,10^{-15}) \tag{4.8}$$

and the optimal state trajectory is shown in Fig. 4.1.

4.5.4 Automatic Differentiation

Partial derivatives of the process and cost equations have to be provided. Although it is necessary with manual coding to specify the non-zero elements only, it is still quite a lot of manual work. On the other hand, the hand-coded derivatives are faster and without

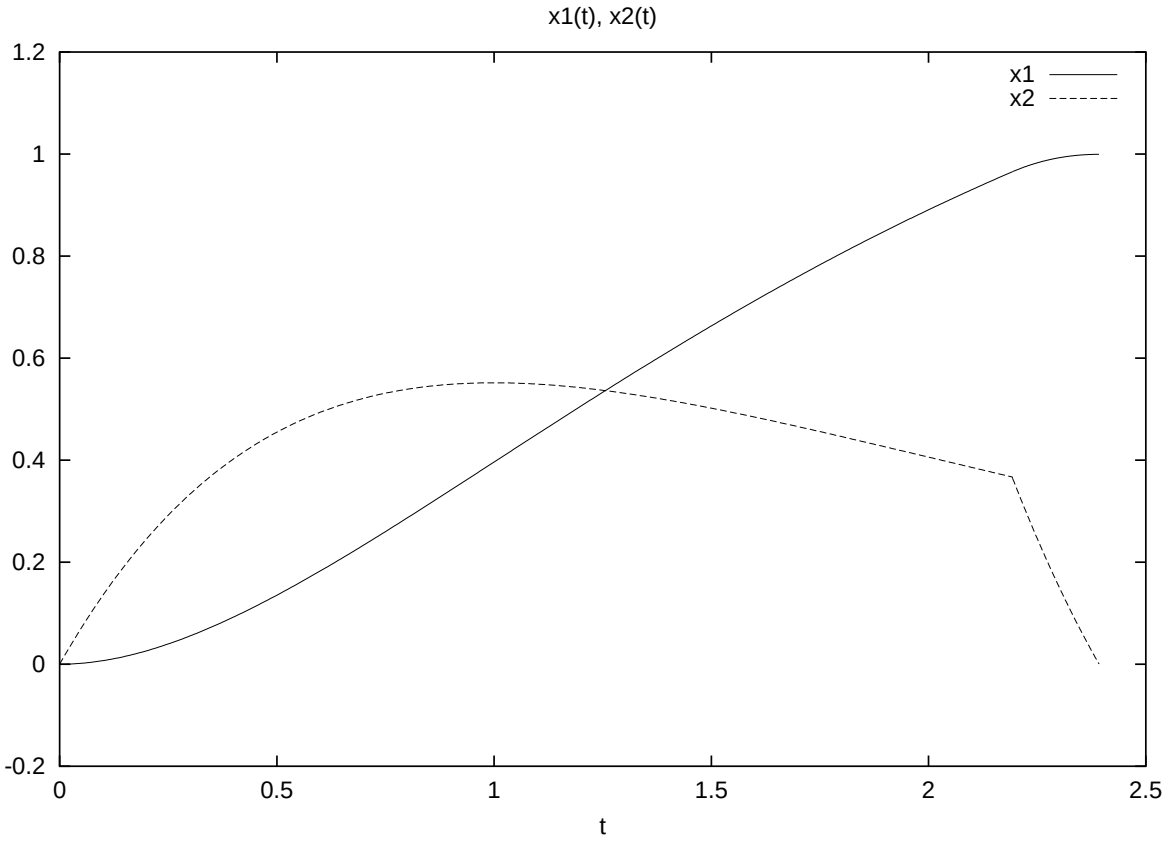

Figure 4.1: Optimal state trajectory for example 1

unnecessary overhead when compared to AD derivatives. For example, every call to a function gradient routine has to evaluate the function as well.

There are several packages available in the Internet that can help to generate the derivatives automatically (for example JAKEF, ADIFOR, TAPENADE, ...). The current version supports ADIFOR. We will assume that ADIFOR has correctly been installed and that it is understood how it works. Then, the following steps are necessary to plug it into `DYNO`:

1. We assume that subroutine `process` resides in file `process.f`, subroutines `costni`, `costi` are in file `cost.f`. If not, the whole procedure given below has to be changed according to ADIFOR instructions. Not all flags have to be filled in the subroutines. If for example `info(16)=1` then `process` only needs `flag=0,-1,-3`, and similarly `costni`, `costi`.

2. Next, we provide a file `adf.f` that contains calls to all subroutines to be differentiated. It can be found in all examples with AD (directory `adexamples`). This file should not be included into the project files to be compiled, it only serves for ADIFOR directly.

3. It is necessary to create an ADIFOR file with dependencies of differentiated routines. In our case, this is the file `adf.cmp` containing:

   ```
   adf.f
   cost.f
   process.f
   ```

4. Now come ADIFOR files specifying independent and dependent variables, names of routines generated, etc. We need to generate derivatives with respect to $\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{p}, t_i, \boldsymbol{x}_i$ from subroutines `process`, `costni`, `costi`. These are ADIFOR `adf` files, in our case `adfx.adf`, `adfu.adf`, `adft.adf`. For example, the file `adfx.adf` can be as follows:

   ```
   AD_PMAX  = 100    #  .ge. dim(nsta*ntime)
   AD_TOP   = adf
   AD_PROG  = adf.cmp
   AD_IVARS = x
   AD_OVARS = sys
   AD_OUTPUT_DIR = .
   AD_PREFIX = x
   ```

   As this file is used for both derivatives with respect to $\boldsymbol{x}, \boldsymbol{x}_i$, it is necessary to specify `AD_PMAX` sufficiently large. The other commands specify the top routine to be differentiated (`adf`), the file where all dependent routines can be found (`adf.cmp`), independent variable (`x`), dependent variable (`sys`), directory where the output should be generated (actual directory), and what is the name of generated routine (in our case there will be `x_process`, `x_costni`, `x_costi`, in files `x_process.f`, `x_cost.f`, respectively).

5. The way how ADIFOR is invoked depends on the operating system. In our case the Solaris version has been tested and the calls are as follows:

```
Adifor2.0 AD_SCRIPT=adfx.adf
Adifor2.0 AD_SCRIPT=adfu.adf
Adifor2.0 AD_SCRIPT=adfp.adf
Adifor2.0 AD_SCRIPT=adft.adf
```

   This generates the needed files `x_process.f`, `u_process.f`, `p_process.f`, `x_cost.f`, `u_cost.f`, `p_cost.f`, `t_cost.f` that can be included into the project.

6. The file `adifyes.f` contains all calls to these routines and has to be included into the project files.

Chapter 5

Examples

Some examples will be presented here that can be used to check the `DYNO` distribution as well as to get acquainted with the desired file/subroutine specifications. Each example is associated with the code accompanying this manual. The code sources can be found in the `examples` and `adexamples` directories for manual and AD generated gradients, respectively.

All examples have been tested in GNU/Linux operating system with Absoft Linux Compiler, GNU g77, and in Microsoft Windows with Digital Fortran compiler.

The AD examples use Sun f77 compiler due to some problems with g77 on Sparc Solaris platform.

5.1 Terminal Constraint Problem

Optimised system:

$$\dot{x}(t) = u(t), \quad x(0) = 1 \tag{5.1}$$

is to be optimised for $u(t) \in [-1, 1]$ with the cost function

$$\min_u J_0 = \int_0^1 (x^2 + u^2)\mathrm{d}t \tag{5.2}$$

subject to constraints:

$$J_1 = x(1) = 0.5 \tag{5.3}$$
$$J_2 = x(0.6) = 0.8 \tag{5.4}$$

We will fix the number of time intervals to 10 and optimise only the control u parameterised as piece-wise constant. The `DYNO` problem formulation is $(t_0 = 0, t_1 = 1, m = 2, m_e = 2)$

Process:

$$\dot{x}(t) = u(t), \qquad x(0) = 1 \tag{5.5}$$

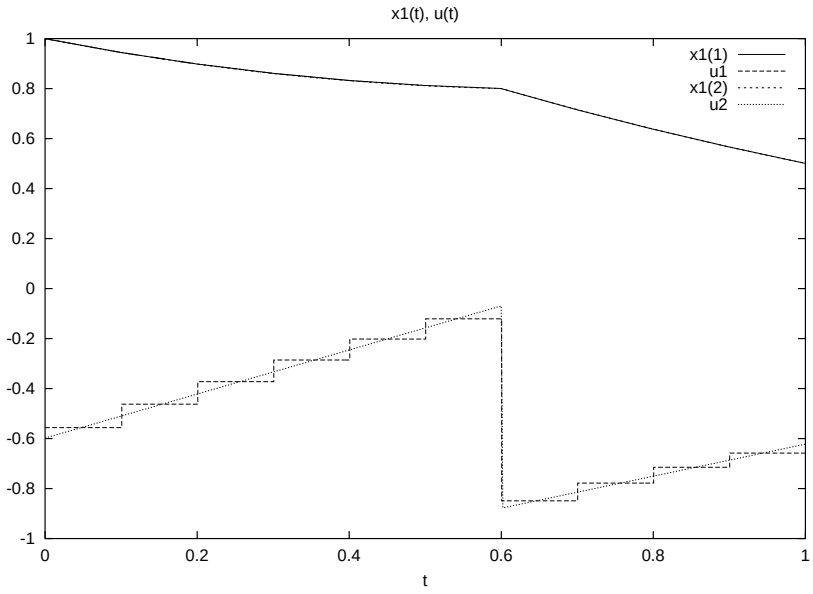

Figure 5.1: Comparison of optimal trajectories for Problems 5.1 and 5.2

Cost:

$$G_0 = 0, \qquad F_0 = x^2 + u^2 \tag{5.6}$$

Constraints:

$$G_1 = x(t_{10}) - 0.5, \quad F_1 = 0 \tag{5.7}$$
$$G_2 = x(t_6) - 0.8, \quad F_2 = 0 \tag{5.8}$$

Bounds:

$$u_i \in [-1, 1] \qquad i = 1 \ \ldots \ 10 \tag{5.9}$$

Optimum has been found in 3 iterations. The example files and results as well are given in directory `problem1`. The optimal control and state trajectory are also shown in Fig. 5.1.

5.2 Terminal Constraint Problem 2

Much better approximation can be obtained with piece-wise linear control with only 2 time intervals.

The new problem formulation is hence:

Process:

$$\dot{x}(t) = u_1(t) + tu_2(t), \quad x(0) = 1 \tag{5.10}$$

Cost:

$$G_0 = 0, \qquad F_0 = x^2 + (u_1 + tu_2)^2 \tag{5.11}$$

Constraints:

$$G_1 = x(t_2) - 0.5, \qquad F_1 = 0 \tag{5.12}$$
$$G_2 = x(t_1) - 0.8, \qquad F_2 = 0 \tag{5.13}$$

We have not included control bounds as they were not active in the previous problem.

Optimum has been found in 10 iterations (Fig. 5.1). The example files and results as well are given in directory `problem2`.

5.3 Inequality State Path Constraint Problem

Consider a process described by the following system of 2 ODE's (Jacobson and Lele, 1969; Logsdon and Biegler, 1989; Feehery, 1998) :

$$\dot{x}_1(t) = x_2(t), \qquad x_1(0) = 0 \tag{5.14}$$
$$\dot{x}_2(t) = -x_2(t) + u(t), \quad x_2(0) = -1 \tag{5.15}$$

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

$$\min_u J_0 = \int_0^1 (x_1^2 + x_2^2 + 0.005u^2)\mathrm{d}t \tag{5.16}$$

subject to state path constraint:

$$J_1 \quad = \quad x_2 - 8(t - 0.5)^2 + 0.5 \leq 0, \quad t \in [0, 1] \tag{5.17}$$

We will fix the number of time intervals to 10 and optimise both times and control u parameterised as piece-wise linear. One possible `DYNO` problem formulation is ($t_0 = 0, t_1 = 1, m = 2, m_e = 1$):

Process:

$$\dot{x}_1(t) = x_2(t), \qquad x_1(0) = 0 \tag{5.18}$$
$$\dot{x}_2(t) = -x_2(t) + (u_1 + tu_2), \quad x_2(0) = -1 \tag{5.19}$$

Cost:

$$G_0 = 0, \qquad F_0 = x_1^2 + x_2^2 + 0.005(u_1 + tu_2)^2 \tag{5.20}$$

Constraints:

$$G_1 = -1 + \sum_{j=1}^{10} \Delta t_j, \qquad F_1 = 0 \tag{5.21}$$

$$G_2 = 0, \qquad F_2 = \varepsilon - (\max(x_2 - 8(t-0.5)^2 + 0.5, 0))^2 \tag{5.22}$$

Note, that the state path inequality constraint has been rewritten as integral equality constraint and then relaxed to inequality with $\varepsilon = 10^{-5}$. The example files and results as well are given in directory `problem31`.

Another possibility how to define a suitable `DYNO` problem formulation is to apply the technique of slack variables.

As the constraint is not a function of u, it is differentiated with respect to time, thus

$$x_2 - 8(t-0.5)^2 + 0.5 + \frac{1}{2}a^2 = 0 \tag{5.23}$$

$$\dot{x}_2 - 16(t-0.5) + a\dot{a} = 0, \quad a(0) = \sqrt{5} \tag{5.24}$$

The initial condition $a(0)$ follows from the conditions at time $t = 0$ when both states are known.

Comparing (5.15) and (5.24) follows for u

$$u = x_2 + 16(t-0.5) - a\dot{a} \tag{5.25}$$

Now setting $a_1 = \dot{a}$ as an optimised variable leads to the optimisation problem:

$$\min_{a_1(t)} J = \int_0^1 (x_1^2 + x_2^2 + 0.005(x_2 + 16(t-0.5) - aa_1)^2)\mathrm{d}t \tag{5.26}$$

subject to:

$$\dot{x}_1 = x_2, \qquad x_1(0) = 0 \tag{5.27}$$

$$\dot{x}_2 = 16(t-0.5) - aa_1, \quad x_2(0) = -1 \tag{5.28}$$

$$\dot{a} = a_1, \qquad a(0) = \sqrt{5} \tag{5.29}$$

and a_1 is parametrised as an optimised variable.

This `DYNO` problem formulation is ($t_0 = 0, t_1 = 1, m = 1, m_e = 1$, optimised control variable denoted by v):

Process:

$$\dot{x}_1(t) = x_2(t), \qquad x_1(0) = 0 \tag{5.30}$$

$$\dot{x}_2(t) = -x_3(v_1 + tv_2) + 16(t-0.5), \qquad x_2(0) = -1 \tag{5.31}$$

$$\dot{x}_3(t) = (v_1 + tv_2) \qquad x_3(0) = \sqrt{5} \tag{5.32}$$

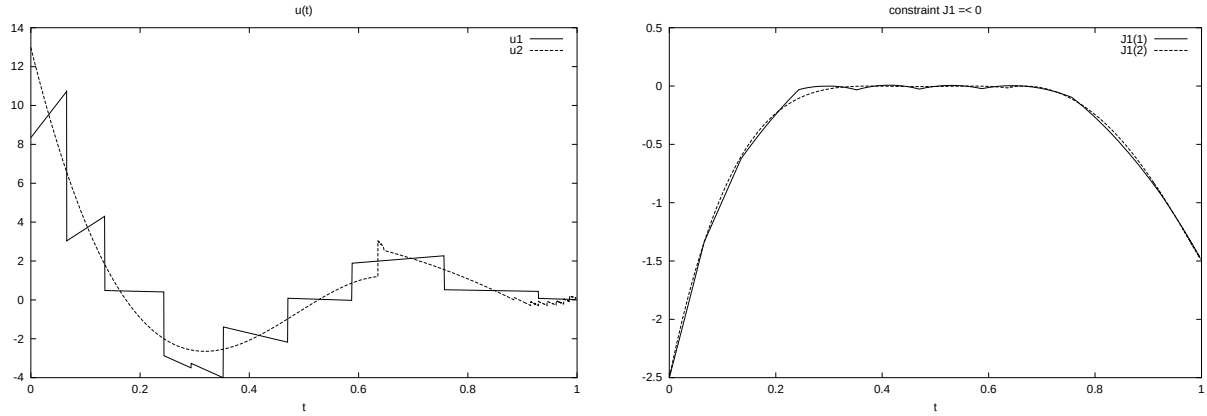

Figure 5.2: Optimal trajectories found for Problem 5.3. Left: control, right: constraint

Cost:

$$G_0 = 0, \quad F_0 = x_1^2 + x_2^2 + 0.005(x_2 + 16(t - 0.5) - x_3(v_1 + tv_2))^2 \tag{5.33}$$

Constraints:

$$G_1 = -1 + \sum_{j=1}^{10} \Delta t_j, \quad F_1 = 0 \tag{5.34}$$

Original control variable:

$$u(t) = x_2(t) + 16(t - 0.5) - x_3(t)(v_1 + tv_2) \tag{5.35}$$

The example files and results as well are given in directory `problem32`.

Different optima have been found. In the first approach, minimal value was $J_0 = 0.1771$, whereas in the second $J_0 = 0.1729$. Comparison of both for optimal control and constraint trajectories is shown in Fig. 5.2.

As it can be seen, slack variable approach approximates optimal control trajectory much better. To obtain comparable results with the first approach, IVP and NLP precisions had to be increased (here with default values) and solver NLPQL has been used. (See Fikar (2001) for more detailed analysis of the path constrained problems).

5.4 Batch Reactor Optimisation

Consider a simple batch reactor with reactions $A \rightarrow B \rightarrow C$ and problem of its dynamic optimisation as described in Crescitelli and Nicoletti (1973). The parameters of the reactor are $k_{10} = 0.535e11$, $k_{20} = 0.461e18$, $e_1 = 18000$, $e_2 = 30000$, $r = 2.0$, final time $t_f = 8.0$, $\beta_1 = 0.53$, $\beta_2 = 0.43$, $\alpha = e_2/e_1$, $c = k_{20}/k_{10}^{\alpha}$. For more detailed description of the parameters see Crescitelli and Nicoletti (1973).

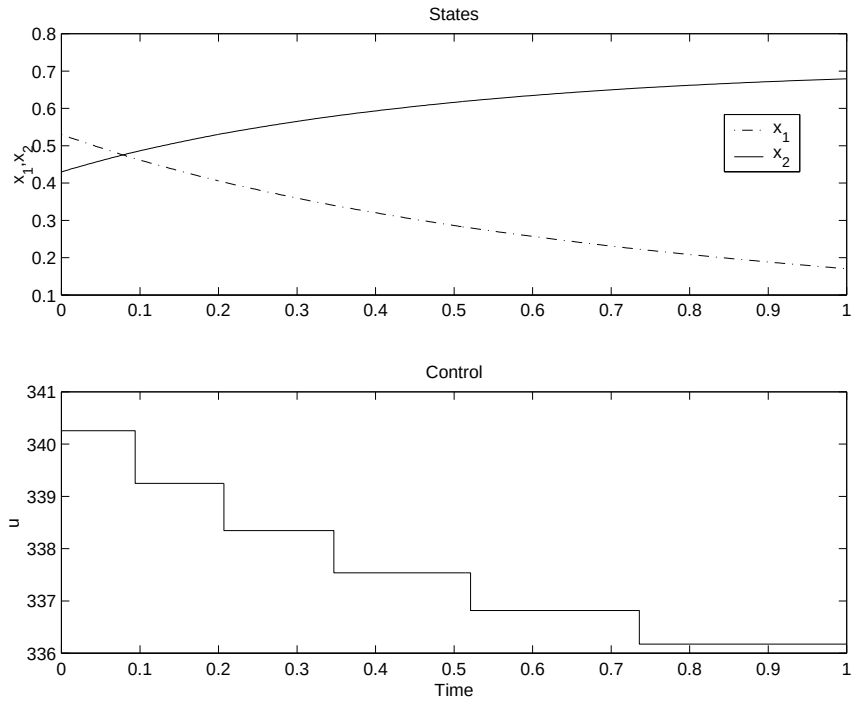

Figure 5.3: Optimal trajectories found for Problem 5.4

The differential equations describing the process are as follows

$$\dot{x}_1 = -ux_1 \tag{5.36}$$
$$\dot{x}_2 = ux_1 - cu^{\alpha}x_2 \tag{5.37}$$

with the initial state

$$x_1(0) = \beta_1, \quad x_2(0) = \beta_2 \tag{5.38}$$

The control variable u is related to the reactor temperature T via the relation

$$T = -\frac{e_1}{r \ln \frac{u}{k_{10}}} \tag{5.39}$$

The objective of the optimisation is to maximise the yield of product B at time t_f: $x_2(t_f)$ subject to piece-wise constant control. In the original article, 3 piece-wise constant control segments are considered, with segment lengths being also optimised variables. Here, we assume 6 intervals

This `DYNO` problem formulation is ($t_0 = 0, t_1 = t_f, m = 1, m_e = 1$):

Process:

$$\dot{x}_1 = -ux_1, \qquad x_1(0) = \beta_1 \tag{5.40}$$
$$\dot{x}_2 = ux_1 - cu^{\alpha}x_2, \qquad x_2(0) = \beta_2 \tag{5.41}$$

Cost:

$$G_0 = -x_2(t_3), \qquad F_0 = 0 \tag{5.42}$$

Constraints:

$$G_1 = -t_f + \sum_{j=1}^{3} \Delta t_j, \quad F_1 = 0 \tag{5.43}$$

The example files and results as well are given in directory `problem4` and in Fig. 5.3.

5.5 Non-linear CSTR

Consider the problem given in (Luus, 1990; Balsa-Canto et al., 2001). The problem consists of determining four optimal controls of a chemical reactor in order to obtain maximum economic benefit. The system dynamics describe four simultaneous chemical reactions taking place in an isothermal continuous stirred tank reactor. The controls are the flowrates of three feed streams and an electrical energy input used to promote a photochemical reaction.

Problem formulation: Find $\boldsymbol{u}(t) = [u_1, u_2, u_3, u_4]$ over $t \in [t_0, t_f]$ to maximise

$$J_0 = x_8(t_f) \tag{5.44}$$

Subject to:

$$\begin{aligned}
\dot{x}_1 &= u_4 - qx_1 - 17.6x_1x_2 - 23x_1x_6u_3 && (5.45)\\
\dot{x}_2 &= u_1 - qx_2 - 17.6x_1x_2 - 146x_2x_3 && (5.46)\\
\dot{x}_3 &= u_2 - qx_3 - 73x_2x_3 && (5.47)\\
\dot{x}_4 &= -qx_4 + 35.20x_1x_2 - 51.30x_4x_5 && (5.48)\\
\dot{x}_5 &= -qx_5 + 219x_2x_3 - 51.30x_4x_5 && (5.49)\\
\dot{x}_6 &= -qx_6 + 102.60x_4x_5 - 23x_1x_6u_3 && (5.50)\\
\dot{x}_7 &= -qx_7 + 46x_1x_6u_3 && (5.51)\\
\dot{x}_8 &= 5.80((qx_1) - u_4) - 3.70u_1 - 4.10u_2 + q(23x_4 + 11x_5 + 28x_6 + 35x_7) \\
&\quad - 5.0u_3^2 - 0.099 && (5.52)
\end{aligned}$$

where $q = u_1 + u_2 + u_4$. The process initial conditions are

$$\boldsymbol{x}(0)^T = [0.1883\ 0.2507\ 0.0467\ 0.0899\ 0.1804\ 0.1394\ 0.1046\ 0.000] \tag{5.53}$$

and the bounds on control variables are $u_1 \in [0, 20]$, $u_2 \in [0, 6]$, $u_3 \in [0, 4]$, $u_4 \in [0, 20]$. The final time is considered fixed as $t_f = 0.2$.

Optimal solution obtained ($J_0 = 21.757$) for $P = 11$ control segments and for equidistant time intervals is the same as given in the literature.

The example files and results as well are given in the directory `problem5`.

5.6 Parameter Estimation Problem

Optimised system:

$$\dot{x}_1(t) = x_2(t), \quad x_1(0) = p_1 \tag{5.54}$$
$$\dot{x}_2(t) = 1 - 2x_2(t) - x_1(t) \quad x_2(0) = p_2 \tag{5.55}$$

represents a second order system with gain and time constants equal to one. The input to the system is 1 and its initial conditions are to be found that correspond to the points

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

The cost function is defined as sum of squares of deviations

$$\min_p J_0 = \sum_{i=1,2,3,5} (x_1(i) - x_1^m(i))^2 \tag{5.56}$$

We will fix the number of time intervals to 6 with stepsize equal to one and optimise only the parameters p_1, p_2. The `DYNO` problem formulation is $(t_0 = 0, t_1 = 6, m = 0, m_e = 0)$

Process:

$$\dot{x}_1(t) = x_2(t), \qquad x_1(0) = p_1 \tag{5.57}$$
$$\dot{x}_2(t) = 1 - 2x_2(t) - x_1(t), \quad x_2(0) = p_2 \tag{5.58}$$

Cost:

$$F_0 = 0, \qquad G_0 = (x(t_1) - .264)^2 + (x(t_2) - .594)^2 \tag{5.59}$$
$$+ (x(t_3) - .801)^2 + (x(t_5) - .959)^2 \tag{5.60}$$

Bounds:

$$p_i \in [-1.5, 1.5] \qquad i = 1 \ \ldots \ 2 \tag{5.61}$$

Optimum has been found in 7 iterations. The example files and results as well are given in directory `problem6`. The optimal parameter values are -0.00112, 0.00163 and both trajectories are shown in Fig. 5.4.

Acknowledgments

We wish to thank Dr. B. Chachuat of ENSIC, Nancy for the numerous discussions during the package implementation that have lead to various enhancements of the package.

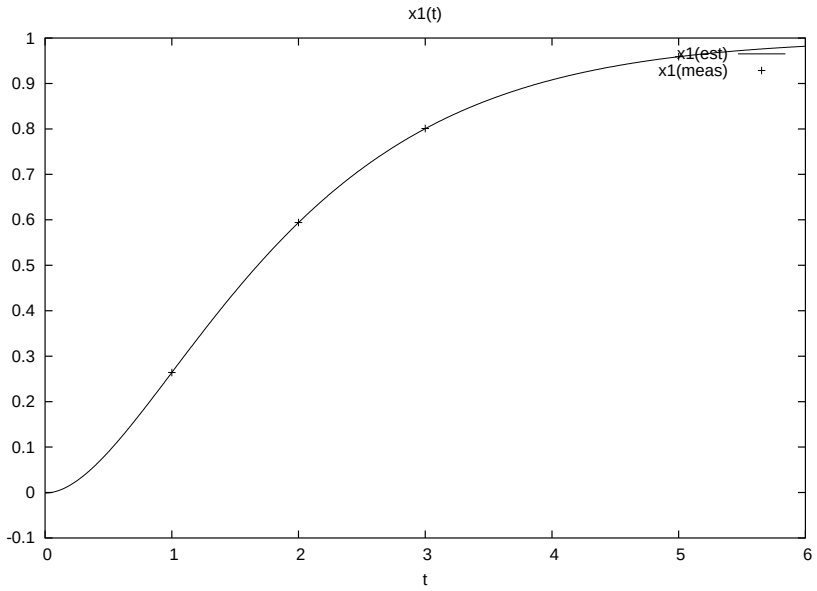

Figure 5.4: Comparison of estimated and measured state trajectory for x_1 trajectory in Problem 5.6

Bibliography

E. Balsa-Canto, J. R. Banga, A. A. Alonso, and V. S. Vassiliadis. Dynamic optimization of chemical and biochemical processes using restricted second-order information. *Computers chem. Engng.*, 25(4–6):539–546, 2001. 48

C. Bischof, A. Carle, P. Hovland, P. Khademi, and A. Mauer. *ADIFOR 2.0 User's Guide (Revision D)*. Mathematics and Computer Science Division (MCS-TM-192), Center for Research on Parallel Computation (CRPC-TR95516-S), 1998. http://www.cs.rice.edu/~adifor. 36

K. E. Brenan, S. E. Campbell, and L. R. Petzold. *Numerical Solution of Initial Value Problems in Differential-Algebraic Equations*. North-Holland, New York, 1989. 36

P. N. Brown, G. D. Byrne, and A. C. Hindmarsh. VODE: A variable coefficient ODE solver. *SIAM J. Sci. Stat. Comput.*, 10:1038 – 1051, 1989. 36

A. E. Bryson, Jr. and Y. C. Ho. *Applied Optimal Control*. Hemisphere Publishing Corporation, 1975. 13, 16, 17

S. Crescitelli and S. Nicoletti. Near optimal control of batch reactors. *Chem. Eng. Sci.*, 28:463–471, 1973. 46

J. E. Cuthrell and L. T. Biegler. Simultaneous optimization and solution methods for batch reactor control profiles. *Computers chem. Engng.*, 13(1/2):49–62, 1989. 9

W. F. Feehery. *Dynamic Optimization with Path Constraints*. PhD thesis, MIT, 1998. 44

W. F. Feehery and P. I. Barton. Dynamic optimization with state variable path constraints. *Computers chem. Engng.*, 22(9):1241–1256, 1998. 13

M. Fikar. On inequality path constraints in dynamic optimisation. Technical Report mf0102, Laboratoire des Sciences du Génie Chimique, CNRS, Nancy, France, 2001. 46

D. Jacobson and M. Lele. A transformation technique for optimal control problems with a state variable inequality constraint. *IEEE Trans. Automatic Control*, 5:457–464, 1969. 13, 44

D. Kraft. A software package for sequential quadratic programming. Technical report, DFVLR Oberpfaffenhofen, 1988. available from `www.netlib.org`. 36

J. S. Logsdon and L. T. Biegler. Accurate solution of differential-algebraic optimization problem. *Ind. Eng. Chem. Res.*, 28:1628–1639, 1989. 44

R. Luus. Application of dynamic programming to high-dimensional non-linear optimal control problems. *Int. J. Control*, 52(1):239–250, 1990. 48

O. Rosen and R. Luus. Evaluation of gradients for piecewise constant optimal control. *Computers chem. Engng.*, 15(4):273–281, 1991. 20

K. Schittkowski. NLPQL : A FORTRAN subroutine solving constrained nonlinear programming problems. *Annals of Operations Research*, 5:485–500, 1985. 25, 36

K. L. Teo, C. J. Goh, and K. H. Wong. *A Unified Computational Approach to Optimal Control Problems*. John Wiley and Sons, Inc., New York, 1991. 12

J. V. Villadsen and M. L. Michelsen. *Solution of Differential Equation Models by Polynomial Approximation*. Prentice Hall, Englewood Cliffs, NJ, 1978. 9