# Imperial College London Optimal Control Software User Guide (ICLOCS)

Paola Falugi Eric Kerrigan
Eugene van Wyk
[*]Department of Electrical and Electronic Engineering,
Imperial College London London England, UK
*iclocs@imperial.ac.uk*

6 May 2010

## 1 Introduction

This document presents a brief user's guide to the optimal control software supplied. The code allows users to define optimal control problems with general path and boundary constraints, free or fixed final times and the ability to include constant design parameters as unknowns. The following optimal control problems fall within the scope of the code:

$$\begin{array}{ll}
\min_{u(t),t_f,p,x_0} J(x(\cdot),u(\cdot),p,t_f) & \\
\text{subject to} & \\
\dot{x} = f(x(t),u(t),p,t),\ x(t_0) = x_0 & \forall\, t \in [t_0, t_F] \\
g_L \le g(x(t),u(t),p,t) \le g_U & \forall\, t \in [t_0, t_F] \\
\phi_L \le \phi(x_0,x_f,u_0,u_f,p,t_f) \le \phi_U & \forall\, t \in [t_0, t_F] \\
x_L \le x(t) \le x_U & \forall\, t \in [t_0, t_F] \\
u_L \le u(t) \le u_U & \forall\, t \in [t_0, t_F] \\
p_L \le p \le p_U &
\end{array}$$

where $u_0 \triangleq u(t_0)$, $x_f \triangleq x(t_f)$ and $u_f \triangleq u(t_f)$. Here the cost function is defined as

$$J(x(\cdot),u(\cdot),p,t_f) \triangleq \int_{t_0}^{t_f} L(x(t),u(t),p,t)dt + E(x_0,x_f,u_{0,}u_f,p,t_f)$$

where $E(\cdot)$ is the cost associated with the boundary conditions and $L(\cdot)$ the stage cost function. The arguments over which the cost function can be minimised are the time-varying control input signals $u(\cdot)$, the initial state $x_0$, the final time $t_f$ and a set of parameters $p$ that are constant for the duration of the phase. The function $g(\cdot)$ describes general path constraints and $\phi(\cdot)$ imposes the boundary conditions at the beginning and end of the phase.

[*]This research was supported by EPSRC under the grant EESC/PO6675.

As a first step, the user-defined optimal control problem is transcribed to a static optimisation problem by either direct multiple shooting or direct collocation methods. The direct multiple shooting formulation requires the solution of initial value problems that can be determined using the open-source sensitivity solver package CVODES. The entire Sundials suite includes the CVODES solver and the SundialsTB Matlab Toolbox. The multiple shooting method does not yet support problems with the final time $t_f$ and the set of parameters $p$ as variables. The direct collocation formulations discretize the system dynamics using implicit Runge-Kutta formulae and can also be used to incorporate discrete-time problems. Once the optimal control problem has been transcribed it can be solved with a selection of nonlinear constrained optimisation algorithms given by the open-source code IPOPT or Matlab's own NLP solver fmincon. The derivatives of the ODE right-hand side, cost and constraint functions are also required for the optimisation and are either estimated numerically or supplied analytically.

## 2 Installation

The code is entirely Matlab-based and can be used without installing any additional software by making use of Matlab's own built-in functions. The code can be used in conjunction with the following packages:

- Ipopt with mex interface (highly recommended):

  https://projects.coin-or.org/Ipopt

- SUNDIALS v.2.4.0 (optional):

  `https://computation.llnl.gov/casc/sundials/main.html`

It is highly recommended that the free NLP solver Ipopt be installed since this will dramatically improve functionality and performance of the code. If you would like to use a multiple shooting method to solve your problem, the sensitivity solver CVODES (it comes with the SUNDIALS suite) also has to be installed.

Before running the code you will need to include the Ipopt libraries in the path by using `setenv` (or possibly `export`) commands and include the file *Ipopt/Contrib/MatlabInterface/ipopt.mexglx* in your Matlab path. To compile the CVODES mex file, simply add the sundialsTB directory to your Matlab path and run `startup_STB.m`. Since the optimal control code consists only of m-files no installation is necessary but don't forget to add *../ICLOCS/src/* to your Matlab path. The current version of ICLOCS has been tested under Linux (Red Hat Enterprise Linux Version 5.2 and Ubuntu 9.0.4) with Matlab 7.6.0. The compilation of Ipopt has been performed using the compilers gcc-4.1 and gfortran.

## 3 Solving Optimal Control Problems

This section details the procedure for defining and solving optimal control problems using the Matlab code provided. In general, the following steps have to be performed:

1. Copy `main.m`, `settings.m`, `myProblem.m` from *.../ICLOCS/usr/* to your working directory. The other files in this directory `gradCost.m`, `jacConst.m` and `hessianLagrangian.m`) should only be copied if required (see Section 3.2.1).

2. Define the optimal control problem by editing `myProblem.m` (see Section 3.1). Note that the file can be renamed as long as this change is reflected in the `main.m`.
3. Edit `settings.m` to choose the solution method and solver settings (see Section 3.2).
4. Run `main.m`.

Steps 2 and 3 are discussed in greater detail below.

## 3.1 Defining the Optimal Control Problem

The following section describes how the optimal control problem is to be defined in the problem definition file (originally called `myProblem.m`)

### 3.1.1 General Problem Definition

1. **Initial time**. The initial time $t_0$ has to be defined here

   problem.time.t0=$t_0$;

   For discrete-time systems $t_0$ is the initial index.

2. **Final time**. The final time $t_f$ can be a variable of the optimisation problem and the bounds for the final time have to be assigned.

   ```
   problem.time.tf_min=final_time_min;
   problem.time.tf_max=final_time_max;
   guess.tf=final_time_guess;
   ```

   If the final time is fixed set the minimum final time `final_time_min` equal to the maximum final time `final_time_max`.
   Note that `final_time_min`$>t_0$ and `final_time_max`$\geq$`final_time_min`.
   For discrete-time systems set the `final_time_min`, `final_time_max` and `final_time_guess` as empty matrices.
   In all other cases a `final_time_guess` has to be supplied.

3. **Parameters.** The bounds of any unknown (constant) parameters that are included in the optimisation should be defined here.

   ```
   problem.parameters.pl=[p1_lowerbound, ...];
   problem.parameters.pu=[p1_upperbound, ...];
   guess.parameters=[p1_guess p2_guess, ...];
   ```

   Define all lower and upper bounds on the parameters as entries in a row vector. If parameters are unbounded their bounds can be set to `-inf` or `inf` . As before, an initial guess for the unknown parameters should be provided. If there are no unknown parameters that can be optimised over, set the bound and initial guess vectors to [ ].

4. **Initial conditions.** The bounds for the initial condition $x_0$ of the system have to be defined in row vectors as shown in the following lines

```
problem.states.x0l=[x1(t0)_lowerbound ...  xn(t0)_lowerbound];
problem.states.x0u=[x1(t0)_upperbound ...  xn(t0)_upperbound];
problem.states.x0=[x1(t0), ...  xn(t0)];
```

If the initial conditions are fixed let `problem.states.x0l=problem.states.x0u`. Note that there will be $n$ bounds for a system with $n$ states. When the initial condition belongs to a box a value for $x_0$ can be assigned in `problem.states.x0` (it can be a guess or a desired value). Otherwise `problem.states.x0` can be the empty matrix.

5. **State variables.** Box constraints for the state variables at $t_0 \leq t \leq t_f$ are defined here.

```
problem.states.xl=[x1_lowerbound ...  xn_lowerbound];
problem.states.xu=[x1_upperbound ...  xn_upperbound];
```

If the states are unbounded their bounds have to be set to `-inf` or `inf`

6. **Final state**. Bounds on the final state at $t = t_f$ are specified here.

```
problem.states.xfl=[x1(tf)_lowerbound ...  xn(tf)_lowerbound];
problem.states.xfu=[x1(tf)_upperbound ...  xn(tf)_upperbound];
```

If the final states are unbounded their bounds have to be set to `-inf` or `inf`

7. **Guess state trajectories.** By default, the initial guess for the state trajectories is automatically generated by linearly interpolating between the expected initial and final value, for each state, which are provided as shown below.

```
guess.states(:,1)=[x1(t0) x1(tf)];
```

$$\vdots$$

```
guess.states(:,n)=[xn(t0) xn(tf)];
```

If the variable `guess.states` is the empty matrix, the initial trajectories will be generated by linearly interpolating between random (but feasible) initial and final values for each state. Note that the initial guess generated here can be overwritten and a user-supplied guess can be assigned in `main.m` by defining the variable `infoNLP.z0` (see Remark 2).

8. **Number of control actions.** The number of piecewise constant control actions can be defined here. For direct collocation methods $N = 0$ sets the number of control actions equal to the number of integration steps. Note that the number of integration steps $M - 1$ (defined in `settings.m`) have to be divisible by the number of control actions. For multiple shooting the number of control actions is equal to the number of integration steps ($N = M - 1$).

```
problem.inputs.N=number_of_control_actions;
```

9. **Control inputs.** Upper and lower bounds for the control inputs are defined as follows.

```
problem.inputs.ul=[u1_lowerbound ...  um_lowerbound];
problem.inputs.uu=[u1_upperbound ...  um_upperbound];
```

   Note that there are $m$ entries in the row vector if the problem is specified with $m$ control inputs.

10. **Guess input sequence.** Provide an initial guess for the optimal control sequence. The initial guess is generated in a similar way to that of the initial state trajectory.

```
guess.inputs(:,1)=[u1(t0) u1(tf)];

                        ⋮

guess.inputs(:,m)=[um(t0) um(tf)];
```

   If the variable `guess.inputs` is the empty matrix, the initial trajectories will be generated by linearly interpolating between random (but feasible) initial and final values for each state. Note that the initial guess generated here can be overwritten and a user-supplied guess can be supplied in `main.m` by defining the variable `infoNLP.z0` (see Remark 2).

11. **Choose set-points.** Constant state and input setpoints can be defined here if required. These will be formatted and passed to the stage cost function as `xr` and `ur` respectively to be used as reference trajectories along the optimisation horizon.

```
problem.setpoints.states=[x1_setpoint ...  xn_setpoint];
problem.setpoints.inputs=[u1_setpoint ...  um_setpoint];
```

   Alternatively time-varying setpoints can also be passed to the stage cost through the structured variable `data` (see Section 3.3)

12. **Bounds for path constraint function.** Set the upper and lower bounds for the path constraint function as entries in a row vector, if required. Set the variables to [ ] if there are no path constraints.

```
problem.constraints.gl=[g1_lowerbound g2_lowerbound ...];
problem.constraints.gu=[g1_upperbound g2_upperbound ...];
```

13. **Bounds for boundary constraints.** Set the upper and lower bounds for the boundary constraint function as entries in a row vector, if required. Set the variables to [ ] if there are no boundary constraints.

```
problem.constraints.bl=[b1_lowerbound b2_lowerbound ...];
problem.constraints.bu=[b1_upperbound b2_upperbound ...];
```

14. **Function definition**

    The stage cost function $L(\cdot)$, the boundary cost function $E(\cdot)$, the path constraints $g(\cdot)$, system equations $f(\cdot)$ and the boundary constraint function $b(\cdot)$ have to be defined in functions with the name `L`, `E`, `g`, `f` and `b` respectively. Their name has to be stored as follows and illustrated in the provided examples in the toolbox.

    ```
    problem.functions={@L,@E,@f,@g,@b};
    ```

    A detailed discussion of each function is carried out in the Section 3.1.2

15. **User defined data problem**

    It is possible to store constant parameters of the problem that are not optimisation variables (for instance transition matrices, reference trajectories, cost weights, etc) in a structured variable as follows (see also illustrative examples).

    ```
    problem.data.a=2;
    problem data.b=1;
    ```

    Set `problem.data=[]` if there are no data.

### 3.1.2 Function Definitions

1. **Stage cost function.** The stage cost function

    ```
    stageCost = L(x,xr,u,ur,p,t,data)
    ```

    computes the stage cost for a given state `x`, steady state reference `xr`, input `u`, steady input reference `ur`, parameters `p` and the time instant `t`. The variables `x`, `xr`, `u`, `ur` and `p` for a time instant $t$ are passed to the function as row vectors. In general, the arguments `x`, `xr`, `u`, `ur` and `p` will be matrices whose rows correspond to the states, inputs and parameters at different time instants. For instance the $i^{\text{th}}$ state variable `xi` can be obtained as follows

    ```
    xi = x(:,i)
    ```

    Importantly, the function should return a column vector if called with arguments that have more than one row. Each entry of the output corresponds to the evaluation of the stage cost for a point in time. If there is no stage cost, let `stageCost=0*t` so that the output will have the right dimension;

2. **Boundary cost function** The boundary cost function

    ```
    boundaryCost=E(x0,xf,u0,uf,p,tf,data)
    ```

    returns a scalar cost as a function of its arguments. If there is no boundary cost let `boundaryCost=0`.

3. **System equations**

The ODE right-hand side of the dynamical system is defined in the function

```
dx = f(x,u,p,t,data).
```

The input arguments follow the same rule as in the stage cost function. The function returns a row containing the evaluation of each state equation for a given $x$, $u$, $p$ for a point in time $t$. The function should return a matrix if $x$, $u$ and $p$ are matrices whose rows correspond to the states, inputs and parameters at different points in time. The adopted structure is shown here:

```
x1 = x(:,1); ...  xn=x(:,n),
u1 = u(:,1); ...  um = u(:,m);


dx(:,1) = f1(x1,..xn,u1,..um,p,t);
                                   ⋮
dx(:,n) = fn(x1,..xn,u1,..um,p,t);
```

If the $i^{\text{th}}$ ODE right-hand side does not depend on variables it is necessary to multiply the assigned value by a vector of ones with the same length of `t`, in order to have a vector with the right dimension when called for the optimization.
`Example:  dx(:,i)= 0*ones(size(t,1));`

4. **Path constraint function.** The path constraint function

```
c=g(x,u,p,t,data)
```

is defined in a manner similar to the system equations. It returns a row vector at each point in time. Each entry correspond to the evaluation of a constraint for a given $x$, $u$, $p$ for a point in time $t$. Again this function should be vectorised as follows:

```
x1 = x(:,1); ...  xn=x(:,n);
u1 = u(:,1); ...  um = u(:,m);


c(:,1)=g1(x1,...,u1,...p,t);
                                   ⋮
c(:,ng) = gng(x1,..xn,u1,..um,p,t);
```

where `ng` is the number of constraints. If the problem does not have any path constraints let `c=[]`.

5. **Boundary constraint function.** The boundary constraint function

```
bc=b(x0,xf,u0,uf,p,tf,data)
```

returns a *column* vector corresponding to the evaluation of each boundary constraint. If the problem does not have any path constraints let `bc=[]`.

## 3.2 Choosing Solver Settings

Once the optimal control problem has been defined in `myProblem.m`, the solver methods and settings have to edited in the file `settings.m`.

1. **Transcription Method.** As a first step, the transcription method has to be chosen by setting the variable **options.transcription** to either 'discrete', 'multiple_shooting', 'euler', 'trapezoidal' or 'hermite'. For instance:

   ```
   options.transcription='trapezoidal';
   ```

   Here 'discrete' has to be chosen for discrete-time systems. The multiple-shooting method can be used for continuous-time systems whenever the final time and the constant parameters are not decision variables of the optimisation problem. In this case, if IPOPT is used, the 'quasi-newton' option (`options.ipopt.hessian_approximation='limited-memory'`) for the Hessian computation has to be selected.

2. **Derivative generation.** This option selects how the derivatives are calculated. The string derivative_method can be set to the following values: 'analytic' or 'numeric'

   ```
   options.derivatives=derivative_method;
   ```

   For the 'analytic' option the files `gradCost.m`, `jacConst.m` (and possibly `hessianLagrangian.m`) have to be defined (see Section 3.2.1). For the 'numeric' option the derivatives are computed using finite-differences and do not require definitions of any additional function. If IPOPT is used, the 'quasi-newton' option for the computation of the Hessian can be used.

3. Whenever the numeric differentiation is enabled it is necessary to specify which kind of finite difference approximation to use between the following ones:
   Central difference ('central')
   forward difference ('forward')
   For instance:

   ```
   options.hessianFD='central';
   ```

4. The perturbation size for numerical differentiation can be chosen by setting the variables `options.perturbation.H` and `options.perturbation.J`. The perturbation size for second derivatives can be set in `options.perturbation.H`. The perturbation size for first derivatives can be set in `options.perturbation.J`. It is possible to select default values for the perturbations by setting `options.perturbation.H` and `options.perturbation.J` to the empty matrix:

   ```
   options.perturbation.H=[];
   options.perturbation.J=[];
   ```

   The default values for the Jacobian approximation is `(eps/2)^(1/3)` while for the Hessian is `(8*eps)^(1/3)`.

5. **NLP solver.** To choose the NLP solver the variable `options.NLPsolver` can be set to either 'ipopt' or 'fmincon'. For instance

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

   If Ipopt has been chosen as the solver the following basic settings can be defined:

   - Desired convergence tolerance (relative). The default value is 1e-8.

     ```
     options.ipopt.tol=1e-9;
     ```

   - Hessian computation can either be 'numeric' to use the method selected in `options.derivatives` or 'quasi-newton' for a limited-memory quasi-newton approximation.

     ```
     options.ipopt.hessian_approximation='exact';
     ```

   - Print level. Check out the Ipopt documentation for details.

     ```
     options.ipopt.print_level=5;
     ```

   - Maximum number of iterations. The default value is

     ```
     options.ipopt.max_iter=3000.
     ```

   - Select the barrier parameter update strategy. The default value for this string option is 'monotone'.
     Possible values:

     | | |
     |---|---|
     | 'monotone': | use the monotone (Fiacco-McCormick) strategy |
     | 'adaptive': | use the adaptive update strategy. |

     ```
     options.ipopt.mu_strategy = 'adaptive';
     ```

   - Indicate which information for the Hessian of the Lagrangian function is used by the algorithm. The default value is 'exact'.
     Possible values:

     | | |
     |---|---|
     | 'exact': | Use second derivatives provided by ICLOCS. |
     | 'limited-memory': | Perform a limited-memory quasi-Newton approximation implemented inside Ipopt. |

     ```
     options.ipopt.hessian_approximation='exact';
     ```

There are many other options which are described in the IPOPT documentation. These options can all be changed from the default settings in the file \ICLOCS\src\solveNLP.m if required.

If MATLAB's own NLP solver fmincon is used the options have to be set in

```
options.fmincon=optimset;
```

Consult the MATLAB documentation for detailed information on the various options for fmincon.

6. **Output settings.** It is possible to use the function `output.m` to display some information about the solved optimisation problem. This function uses the display options defined in `settings.m` The available options are described hereinafter. Set to zero to disable options.

   - Display computation time

     ```
     options.print.time=1;
     ```

   - Display relative local discretization error (recommended for direct transcription)

     ```
     options.print.relative_local_error=1;
     ```

   - Display optimal cost

     ```
     options.print.cost=1;
     ```

   - Plot states

     ```
     options.plot.states=1;
     ```

   - Plot inputs

     ```
     options.plot.inputs=1;
     ```

   - Plot Lagrange multipliers relative to the system equations.

     ```
     options.plot.multipliers=1;
     ```

7. **Direct transcription settings**

   - Number of integration nodes in the interval $[t_0,\ tf]$. The quantity steps/$N$ ($N$ number of control actions, steps=nodes-1) must be a positive integer

```
options.nodes=1001;
```

- Distribution of integration steps. Set to `tau=0` for equispaced steps. Otherwise `tau` is a vector of length $M-1$ with $0 <$`tau(i)`$< 1$ and $sum(tau) = 1$. For discrete-time system set `tau`$= 0$.

```
options.tau=0;
```

8. **Multiple shooting settings**

- The ODE solver has to be set to 'cvodes'

```
options.ODEsolver='cvodes';
```

- CVODES settings. Refer to CVODES documentation for more details. Method can be set to either 'Adams' or 'BDF' and solver to 'Newton' or 'Functional'.

```
options.cvodes = CVodeSetOptions('RelTol',1.e-4,'AbsTol',1.e-6,...
          'LinearSolver','Dense', 'MaxNumSteps',10000, 'LMM',Method,...
          'NonlinearSolver',Solver);
```

```
options.cvodesf = CVodeSensSetOptions('ErrControl',true,'method ',...
          'Staggered');
```

### 3.2.1 Derivative Definitions

If some of the analytical gradients are supplied copy `gradCost.m`, `jacConst.m` and `hessianLagrangian.m` to the working directory and edit them as follows.

1. **Gradient of the cost.** In the file `gradCost.m`, it is possible to define the gradient of the stage cost function and the boundary cost by defining the partial derivatives `dL.dp`, `dL.dx`, `dL.du` for the stage cost function and `dE.dtf`, `dE.dp`, `dE.dx0`, `dE.du0`, `dE.dxf`, `dE.duf` for the boundary cost function. Whenever the gradient of the stage cost is supplied set `dL.flag`$= 1$ otherwise set `dL.flag`$= 0$. Similarly if the gradient of the boundary cost is supplied set `dE.flag`$= 1$ otherwise set `dE.flag`$= 0$. The partial derivatives of the stage cost function must be expressed in vector form. For instance the derivative of $L(\cdot)$ with respect to the $i^{\text{th}}$ state variable, evaluated along all the horizon, corresponds to the $i^{\text{th}}$ column of `dL.dx`. The same rule holds for `dL.du`, `dL.dp` and `dL.dt` For details refer to the sample files supplied in some of the examples.

2. **Jacobian of the constraint function.**

   In the file `jacConst.m`, it is possible to define the Jacobian of the ODE right-hand side of the system equations, path constraint and boundary constraint functions. Whenever the gradient of the dynamics is supplied set `df.flag`$= 1$ otherwise set `df.flag`$= 0$. Similarly if the gradient of the path constraints is supplied set `dg.flag`$= 1$ otherwise set `dg.flag`$= 0$. The same rule holds for the boundary cost. Set `db.flag`$= 1$ if the analytic expression of the gradient is available, otherwise `db.flag`$= 0$. For details refer to the sample files supplied in some of the examples.

3. **Hessian of the Lagrangian.**

   The Hessian of the Lagrangian can be defined in the file `hessianLagrangian.m`. The toolbox allows to specify the Hessian of the following functions: stage cost, boundary cost, path constraints, boundary constraints are ODE right-hand side of the system equations. These can be supplied by defining the corresponding cell array of appropriate dimensions otherwise setting the related variable to [ ]. Refer to the directory \ICLOCS\examples\ for clear examples of how these analytic derivatives can be defined.

The derivatives that are not supplied (the corresponding flag is set to zero or the Hessian is set to [ ] ) are evaluated numerically. Note that the limited-memory quasi-newton approximation of the Hessian can still be used if Ipopt is used.

## 3.3 Solving the optimisation problem

Once all the data for the definition and solution of the optimisation problem have been specified in `myProblem.m`, `settings.m` and eventually in `gradCost.m`, `jacConst.m` and `hessianLagrangian.m`, as described in the previous section, the optimisation is performed running the following lines:

```
[problem,guess]=myProblem;
options= settings;
[infoNLP,data]=transcribeOCP(problem,guess,options);
[solution,status] = solveNLP(infoNLP,data);
```

The data inserted in `myProblem.m` and `settings.m` and contained in the variables `problem`, `guess` and `options` have to be properly transcribed for the nonlinear solver. The following subsections describes briefly the functions `transcribeOCP.m`, `solveNLP.m` and their output arguments.

**Transcription function:** `transcribeOCP.m`

The function `transcribeOCP.m` processes the information from `problem`, `guess` and `options` for the nonlinear solver. Mainly it defines bounds for the optimisation variable, dynamic equations, path and boundary constraints, it formats matrices for the direct transcription method (if required), it generates initial guesses for the optimisation variable and the structure of the Jacobian for the constraints and it constructs optimal finite-difference perturbation sets. The function has two output arguments that are structured variables. The first output `infoNLP` contains upper and lower bounds on the optimisation variable, additional constraints and the initial guess. Instead, the second output variable `data` contains the data used in the functions evaluated during optimisation. The optimisation variable $z$ depends on the transcription method, on the number of integration nodes (options.nodes) M and on the number of control actions $N$.

If the transcription method is multiple shooting, $N = M - 1$ and the optimisation variable is

$$z = [x(0),\ u(0),\ x(1),\ u(1),\ ..., x(M-1),\ u(M-1),\ x(M)]$$

The direct transcription methods present two different situations

1. If $N = M - 1$ the optimisation variable is in the most general case
$$z = [tf,\ p, x(0),\ u(0),\ x(1),\ u(1),\ ..., x(M-1),\ u(M-1),\ x(M)]$$
2. If $N < M - 1$ the optimisation variable is in the most general case
$$z = \left[tf,\ p, x(0),\ x(1), ...x\left(\frac{M-1}{N} - 1\right),\ u(0),\ x\left(\frac{M-1}{N}\right), ..., x(M-1),\ u(N),\ x(M)\right]$$

Notice that $M - 1$ have to be divisible by the number of control actions N. If $tf$ and/or $p$ are not variables of the problem, they are not introduced in $z$. For discrete-time systems $N = M - 1$ and $t_f$ cannot be a variable of the optimisation problem. For the Hermite-Simpson method it is imposed $M = 2*$`options.nodes`$-1$. For the trapezoidal method the required $u(M)$ is imposed equal to $u(M-1)$. A detailed description of the output variable follows:

1. Output variable `infoNLP`
   - `infoNLP.zl`: Lower bound of the optimisation variable $z$,
   - `infoNLP.zu`: Upper bound of the optimisation variable $z$;
   - `infoNLP.cl`: Lower bound of the all set of constraints for the optimisation problem
   - `infoNLP.cu`: Upper bound of the all set of constraints for the optimisation problem
   - `infoNLP.z0`: Initial guess for the optimisation problem.
2. Output variable `data`
   - `data.t0`: contains information to evaluate the time instants in the interval $[t_0,\ t_f]$ (see Remark 1).
   - `data.k0`: contains information to evaluate the time instants in the interval $[t_0,\ t_f]$ (see Remark 1).
   - `data.Nm`: contains information to evaluate the time instants in the interval $[t_0,\ t_f]$ (see Remark 1). $Nm = 1$ for continuous-time systems, $Nm = N$ for discrete-time systems.
   - `data.sizes`: contains information about dimensions involved in the problem

     ```
     [nt,np,n,m,ng,nb,M,N,ns]=deal(data.sizes{:});
     ```

     - `nt`= 1 if `tf` is a decision variable otherwise `nt`= 0;

     - `np` contains the number of free constant parameters;

     - `n` gives the number of states;

     - `m` gives the number of inputs;

     - `ng` gives the number of path constraints;

     - `nb` gives the number of boundary constraints;

     - `N` is the number of control actions;

     - `M` is the number of mesh nodes;

- `ns`= 2 for Hermite-Simpson transcription method and `ns`= 1. It is used to adjust sizes and values of some of the variables.

- `data.x0`: contains the initial condition for the state at the time $t_0$. If the variable problem.states.x0, defined in `myProblem.m`, is empty, `transcribeOCP.m` set problem.states.x0=problem.states.x0l.
- `data.x0t`: contains the measured initial condition $x(k_0)$. The function `transcribeOCP.m` sets `data.x0t=data.x0`. The variable `data.x0t` needs to be updated to change the initial condition for receding horizon optimisation problems. If the initial condition is fixed its bounds have to be updated in the following way

```
infoNLP.zl(nt+np+1:nt+np+n)=data.x0t';
infoNLP.zu(nt+np+1:nt+np+n)=data.x0t';
```

- `data.cx0`: `transcribeOCP.m` sets `data.cx0`= 0, if `problem.states.x0` defined in `myProblem.m` is empty, otherwise `data.cx0`= 1. When `data.cx0`= 1 the additional term $\lambda_0$(`data.x0t-data.x0`) is introduced in the Lagrangian, where $\lambda_0$ is the relative Lagrange multiplier.
- `data.options`: contains information about the transcription method, the derivative evaluation and the solver settings.
- `data.functions`: contains the function definition of the stage cost, boundary cost, system dynamic, path constraints and boundary constraints
```
[L,E,f,g,b]=deal(data.functions{:});
```
- `data.data`: contains the data to be used inside the functions and defined in the variable `problem.data` in the function `myProblem.m`.
- `data.references`: The variables `data.references.ur` and `data.references.xr` store respectively the state and input reference trajectories . The function `transcribeOCP.m` generates constant references in $[t_0, t_f]$ expanding the set-point values assigned in problem.setpoints. The user can overwrite the default references assigning other reference trajectories. `data.references.xr` and `data.references.ur` are matrices with dimension $M \times n$ and $M \times m$ respectively. If the transcription_method is Hermite-Simpson it is necessary to assign also the references at the interval midpoints ($M = 2*$`options.nodes`$-1$).
- `data.sparsity`: contains sparsity information about the functions $f(\cdot)$ , $L(\cdot)$ $E(\cdot)$ $g(\cdot)$ and $b(\cdot)$.
- `data.tau`: contains information to evaluate the time instants in the interval $[t_0,\ t_f]$ (see Remark 1).
- `data.map`: `data.map.A`, `data.map.B`, `data.map.w` and `data.map.W` contain the data used for the transcription with direct collocation methods while `data.map.Vx`, `data.map.xV`, `data.map.Vu` and `data.map.uV` are matrices for the mapping of the variables. For the details see Remark 2
- `data.FD`: contains information for the computation of the derivatives

- `data.jacStruct`: brings the structure of the Jacobian for the constraints
- `data.costStruct`: gives the structure of the Jacobian for the cost
- `data.hessianStruct`: brings the structure of the Hessian for the Lagrangian.
- `data.multipliers`: In this field the initial value for the Lagrange multipliers can specified. It is especially useful for "warm starting" the IPOPT solver. They can be specified defining the following fields:

  % Multipliers corresponding to the lower bounds on the variables
  `data.multipliers.zl`
  % Multipliers corresponding to the upper bounds on the variables
  `data.multipliers.zu`
  % Multipliers corresponding to the constraints
  `data.multipliers.lambda`

  The fields `data.multipliers.zl` and `data.multipliers.zu` have to be a column vector with the same length of $z$. The field `data.multipliers.lambda` have to be a column vector with the length given by the number of constraints (`size(data.jacStruct,1);`).

*Remark* 1. The general formula to compute the vector `t` of the time instants $t_k \in [t_0,\ t_f]$ for $k = 1, \ldots, M$ is

```
t=(data.tf-data.t0)*T+data.k0
```

where `T` is a vector taking values in the interval $[0,\ 1]$ such that `t(k)=(data.tf-data.t0)*T(k)+data.k0`. `T` depends on the distribution of integration steps stored in `data.tau/ns` and is given by

```
T=[0;cumsum(data.tau)]*data.Nm/ns;
```

where `ns`$=2$ if the transcription_method is 'hermite' and `ns`$=1$ otherwise so that `sum(data.tau)/ns`$=1$ for any transcription_method. The function `transcribeOCP.m` set `data.k0=data.t0`=initial_time for the continuous-time case while set `data.k0`=initial_time, `data.t0`$=0$ and `data.tf`$=1$ for discrete-time problems. Then, if the initial time has to be changed to solve a time varying optimisation problem in a receding horizon fashion, the variable `data.k0` has to be set properly before to call the function `solveNLP.m`.

*Remark* 2. The data mining of the state $x$ and input $u$ and the assignment of an initial guess for the variable $z$ from guess defined on $x$ and $u$ can be easily performed by mean of `data.map.Vx`, `data.map.xV`, `data.map.Vu` and `data.map.uV`.

`data.map.Vx`: maps $z \rightarrow x$
`data.map.xV`: maps $x \rightarrow z$
`data.map.Vu`: maps $z \rightarrow u$

`data.map.uV`: maps $u \rightarrow z$

- Consider `tf_guess`, `p_guess`, `x_guess`, and `u_guess` to be the guess for the final time (if any), the constant parameters (if any), the states and inputs respectively. `x_guess`, and `u_guess` are matrices with dimension $M \times n$ and $N \times m$ respectively where each row corresponds to the vector variables at some time instant. It is possible to update the guess for $z$ running the following lines

  ```
  u_guess=u_guess';u_guess=u_guess(:);
  x_guess=x_guess';x_guess=x_guess(:);
  infoNLP.z0=data.map.xV*x_guess+data.map.uV*u_guess;
  infoNLP.z0(1)=tf_guess; infoNLP.z0(1:np)=p_guess;
  ```

  If the initial condition is fixed, `data.x0t` and its bounds have to be updated as previously explained.

- Once a solution $z$ is available the variables `x` and `u` can be obtained in the following way

  ```
  x=reshape(data.map.Vx*sol.z,n,M)';
  u=reshape(data.map.Vu*sol.z,m,N)';
  U=kron(u,ones(((M-1)/N,1));
  ```

  `x`, `u` and `U` are matrices of dimension $M \times n$, $N \times m$, and $(M-1) \times m$ respectively. Indeed the output of `solveNLP.m` returns explicitly the variable `x` and `u` together with the optimisation variable $z$.

**Function:** `solveNLP.m`

The function `solveNLP.m` calls the selected solver to solve the optimisation problem on the basis of the information stored in `infoNLP` and `data` and return the solution. The function has two output arguments. The first output `solution` is a structured variable containing the optimal final time, parameters, states, controls and Lagrange multipliers. Instead the second one `status` returns the exit condition of the solver. Its values depend on the selected solver. For its description see the help for the outputs of ipopt and fmincon.

In particular, the output variable `solution`, whenever IPOPT is employed, returns

- `solution.multipliers`: contains the Lagrange multipliers at the solution $z$ (see the IPOPT and MATLAB user guide for its structure).
- `solution.computation_time`: Returns the CPU time in seconds that has been used by MATLAB to run the selected software.
- `solution.iterates`: gives the total number of iterations if `fmincon` is used.
- `solution.z`: the computed solution $z$.
- `solution.tf`: the value of the final time.

- `solution.p`: contains the computed values for the parameters $p$ (if any).
- `solution.X`: a matrix with dimension `options.nodes` $\times$ `n` containing the optimal solution for the states.
- `solution.x0`: the initial value of the determined solution.
- `solution.U`: a matrix with dimension `options.nodes` $\times$ `m` containing the optimal solution for the inputs.
- `solution.T`: brings information about the evaluation time instants for `solution.X` and `solution.U`.

*Remark* 3. It is important to observe that the auxiliary data passed to IPOPT may not change through the course of the IPOPT optimisation. In order to store information changing over time the global variable `sol` has been used

**Display output:** `output.m`

This function uses the display options in `settings.m` and the solution generated by the call to `solveNLP.m` in `main.m` and plots the state, input and adjoint variable trajectories. If direct collocation methods are used `output.m` can also estimate the relative local discretization error to estimate the accuracy with which the dynamics have been approximated [1].

**Extract multipliers:** `multipliers.m`

The function `multipliers.m` processes the information from `data` and the solution of the optimization problem (contained in the variable `solution`) to extract information relative to the multipliers. The function has two output arguments that are structured variables. The first output `lambdai` contains information on the multipliers. Instead, the second output variable `Time` contains the time instants relative to the multipliers

In particular, the output variable `lambdai` returns

- `lambdai.x0`: contains the Lagrange multipliers of the constraint on `(data.x0t-data.x0)`;
- `lambdai.adjoint`: Returns the adjoint variables; depending on the adopted sign convention their sign can be the opposite with respect to the expected one;
- `lambdai.g`: contains the Lagrange multipliers associated with the path constraints if the number `ng` of path constraints is not zero and if IPOPT is used;
- `lambdai.b`: contains the Lagrange multipliers associated with the boundary constraints if the number `nb` of boundary constraints is not zero and if IPOPT is used.

The output variable `Time` returns

- `Time.adjoint`: contains the time instants where the adjoint variables are evaluated;
- `Time.g`: contains the time instants where the path constraints are evaluated if the number `ng` of path constraints is not zero and if IPOPT is used.

# 4 Examples

The following sections contain all the optimal control examples included with the ICLOCS toolbox.

## 4.1 Example 1: A linear optimisation problem with bang-bang control

Find the final time $t_f$ and control input $u \in \mathbb{R}$ over $t$ in $[0,\ t_f]$ solving the following optimisation problem [9]

$$\min_{u(\cdot),\, t_f} t_f$$

subject to

$$\begin{cases} \dot{x}_1(t) = x_2(t) \\ \dot{x}_2(t) = u(t) \end{cases}$$

$$x_1(0) = 0,\ x_2(0) = 0$$
$$x_1(t_f) = 300,\ x_2(t_f) = 0$$

$$-2 \leq u(t) \leq 1$$
$$\begin{bmatrix} -10 \\ -100 \end{bmatrix} \leq \begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} \leq \begin{bmatrix} 300 \\ 100 \end{bmatrix} \quad \forall t \in [0,\ t_f]$$

**Problem setup**

- The Optimal Control Problem is defined in the file `BangBang.m` in the following way:

```
problem.time.t0=0; % Initial time t_0

problem.time.tf_min=0.1;
problem.time.tf_max=100;
guess.tf=1;

% Parameters bounds. pl=< p <=pu
problem.parameters.pl=[];
problem.parameters.pu=[];
guess.parameters=[];

% Initial condition x_0 and its bounds
problem.states.x0=[0, 0]; % x(t_0)
problem.states.x0l=[0, 0]; % Bounds for x(t_0)
problem.states.x0u=[0, 0];
```

% State bounds: $xl \leq x(t) \leq xu$
```
problem.states.xl=[-10 -100];
problem.states.xu=[300 100];
```

% Terminal state bounds: $xfl \leq x(t_f) \leq xfu$
```
problem.states.xfl=[300 0];
problem.states.xfu=[300 0];
```

% Guess the state trajectories with $[x(t_0),\ x(t_f)]$
`guess.states(:,1)=[0 300];` % $[x_1(t_0),\ x_1(t_f)]$
`guess.states(:,2)=[0 0];` % $[x_2(t_0),\ x_2(t_f)]$

Number of control actions $N$. If $N$ is equal to the number of integration steps, problem.inputs.N can be set to 0
```
problem.inputs.N=0;
```

```
problem.inputs.ul=[-2];
problem.inputs.uu=[1];
```

% Guess the input sequences with $[u(t_0),\ u(t_f)]$
```
guess.inputs(:,1)=[-2 1];
```

% Choose the set-points if required
```
problem.setpoints.states=[];
problem.setpoints.inputs=[];
```

% Bounds for path constraint function gl $\leq g(x,u,p,t) \leq$ gu
```
problem.constraints.gl=[];
problem.constraints.gu=[];
```

% Bounds for boundary constraints bl $\leq b(x(t_0),x(t_f),u(t_0),u(t_f),p,t_0,t_f) \leq$ bu
```
problem.constraints.bl=[];
problem.constraints.bu=[];
```

% Store the necessary problem data used in the functions.
```
problem.data=[];
```

```
problem.functions={@L,@E,@f,@g,@b;}
```

```
function stageCost=L(x,xr,u,ur,p,t,data)
```

```
    stageCost = 0*t;

function boundaryCost=E(x0,xf,u0,uf,p,tf,data)

    boundaryCost=tf;

function dx = f(x,u,p,t,data)

    x1 = x(:,1); x2 = x(:,2); u1 = u(:,1);
    dx(:,1) = x2;
    dx(:,2) = u1;

function c=g(x,u,p,t,data)

    c=[];

function bc=b(x0,xf,u0,uf,p,tf,data)

    bc=[];
```

- The solution method and solver settings are set in `settings.m` . See the file included in the directory *../ICLOCS-*.*/examples/BangBang*
  Notice that the multiple-shooting option can not be used since the final time $t_f$ is a decision variable.
- The files `gradCost.m`, `jacConst.m` and `hessianLagrangian.m` for this example are supplied. See inside the directory *../ICLOCS-*.*/examples/BangBang*.

  - `gradCost.m`:

```
    function [dL,dE]=gradCost(L,X,Xr,U,Ur,P,t,E,x0,xf,u0,uf,p,tf,data)

    % get the dimension of the state n and of the input m
    [n,m]=deal(data.sizes{[3:4]});

    % vector of ones with the same size of the number of point evaluation in time.
    Lt=ones(size(T));

    dL.flag=1;
    dL.dp=[]; % Derivatives of L(x,u,p,t) wrt. p
    dL.dt=0*t; % Derivatives of L(x,u,p,t) wrt. t
    % Derivatives of L(x,u,p,t) wrt. x=[x_1,...,x_n]
```

```
dL.dx=kron(zeros(1,n),Lt);
% Derivatives of L(x,u,p,t) wrt. u = [u_1,...,u_m]
dL.du=kron(zeros(1,m),Lt);



dE.flag=1;
dE.dtf=1; % Derivatives of E(x_0,x_f,u_0,u_f,p,t_f) wrt. t_f
dE.dp=[]; % Derivatives of E(x_0,x_f,u_0,u_f,p,t_f) wrt. p
dE.dx0=[]; % Derivatives of E(x_0,x_f,u_0,u_f,p,t_f) wrt. x_0
dE.du0=[]; % Derivatives of E(x_0,x_f,u_0,u_f,p,t_f) wrt. u_0
dE.dxf=[]; % Derivatives of E(x_0,x_f,u_0,u_f,p,t_f) wrt. x_f
dE.duf=[]; % Derivatives of E(x_0,x_f,u_0,u_f,p,t_f) wrt. u_f
```

The gradient of the stage cost and the boundary cost are supplied and so `dL.flag=1` and `dE.flag=1`. The final time $t_f$ is a decision variable and then the derivative of the stage cost with respect to the time $t$ is set to $0 * t$. It is zero at any time instant. The same rule applies for the derivatives of the stage cost with respect to the state and the input. The derivative of the stage cost with respect to the state $x$ is stored in a matrix with $n$ columns and a number of rows given by the different evaluation times. Each row corresponds to the evaluation of the derivative at a given time instant. The derivative of the boundary cost depends only on the final time and then the variables containing the derivatives with respect to the other decision variables are set to be empty matrices.

- `jacConst.m`:

```
function [df,dg,db]=jacConst(f,g,X,U,P,t,b,x0,xf,u0,uf,p,tf,t0,data)

% vector of ones with the same size of the number of point evaluation in time.
Lt=ones(size(t));


df.flag=1;
df.dp{1}=[]; % Derivatives of f(x,u,p,t) wrt. p
df.dt{1}=[0*Lt 0*Lt]; % Derivatives of f(x,u,p,t) wrt. t
df.dx{1}=[0*Lt, 0*Lt]; % Derivatives of f(x,u,p,t) wrt. x_1
df.dx{2}=[1*Lt, 0*Lt]; % Derivative of f(x,u,p,t) wrt. x_2
df.du{1}=[0*Lt, 1*Lt]; % Derivative of f(x,u,p,t) wrt. u


dg.flag=0;
db.flag=0;
```

The path constraints and boundary constraints are not present and then their derivatives have not be supplied (`dg.flag=0` and `db.flag=0`). Instead the derivatives of $f(x, u, p, t) = [f_1(x, u, p, t), \dots, f_n(x, u, p, t)]$ are supplied in the structured variable `df` as shown in the illustrated example. The derivative of $f(x, u, p, t)$ with respect to $x$ are stored in df.dx which is a cell array where each entry corresponds to the derivative with respect to a state variable. For instance the derivative of $f(x,u,p,t)$ with respect to $x_i$ is stored in `df.dx{i}`. `df.dx{i}` is a matrix with $n$ columns and a number of rows given by the different evaluation times. Each row of `df.dx{i}` corresponds to the evaluation of the derivative $[\partial f_1(x,u,p,t)/\partial x_i, \ldots, \partial f_n(x,u,p,t)/\partial x_i]$ at a given time instant. The same rule holds for the other derivatives.

- `hessianLagrangian.m`

```
function [HL,HE,Hf,Hg,Hb]=hessianLagrangian(X,U,P,t,E,x0,xf,u0,uf,p,tf,data)
```

The Hessian of the different parts of the Lagrangian must be supplied in cell arrays as follows:

```
[nt,np,n,m,ng,nb]=deal(data.sizes{1:4});
nfz=nt+np+n+m;
```

```
Hf=cell(nfz, nfz); % Hessian of $f(x,u,p,t)$
Hf{1,1}=[0.*t, 0.*t]; %$[\partial^2 f_1/\partial t^2,\ \partial^2 f_2/\partial t^2]$ (nt=1)
Hf{1,2}=[0.*t, 0.*t]; % $[\partial^2 f_1/(\partial t\ \partial x_1),\ \partial^2 f_2/(\partial t\ \partial x_1)]$
Hf{2,2}=[0.*t, 0.*t]; % $[\partial^2 f_1/(\partial x_1^2), \partial^2 f_2/(\partial x_1^2)]$
Hf{1,3}=[0.*t, 0.*t]; % $[\partial^2 f_1/(\partial t\ \partial x_2), \partial^2 f_2/(\partial t\ \partial x_2)]$
Hf{2,3}=[0.*t, 0.*t]; % $[\partial^2 f_1/(\partial x_1\ \partial x_2), \partial^2 f_2/(\partial x_1\ \partial x_2)]$
Hf{3,3}=[0.*t, 0.*t]; % $[\partial^2 f_1/(\partial x_2^2), \partial^2 f_2/(\partial x_2^2)]$
Hf{1,4}=[0.*t, 0.*t]; % $[\partial^2 f_1/(\partial t\ \partial u), \partial^2 f_2/(\partial t\ \partial u)]$
Hf{2,4}=[0.*t, 0.*t]; % $[\partial^2 f_1/(\partial x_1\ \partial u), \partial^2 f_2/(\partial x_1\ \partial u)]$
Hf{3,4}=[0.*t, 0.*t]; % $[\partial^2 f_1/(\partial x_2\ \partial u), \partial^2 f_2/(\partial x_2\ \partial u)]$
Hf{4,4}=[0.*t, 0.*t]; % $[\partial^2 f_1/(\partial u^2), \partial^2 f_2/(\partial u^2)]$
```

```
HL=cell(nfz, nfz); % Hessian of $L(x,u,p,t)$
HL{1,1}=[0.*t]; % $[\partial^2 L/\partial t^2]$ (nt=1)
HL{1,2}=[0.*t]; % $[\partial^2 L/(\partial t\ \partial x_1)]$
HL{2,2}=[0.*t]; % $[\partial^2 L/(\partial x_1^2)]$
HL{1,3}=[0.*t]; % $[\partial^2 L/(\partial t\ \partial x_2)]$
HL{2,3}=[0.*t]; % $[\partial^2 L/(\partial x_1\ \partial x_2)]$
HL{3,3}=[0.*t]; % $[\partial^2 L/(\partial x_2^2)]$
HL{1,4}=[0.*t]; % $[\partial^2 L/(\partial t\ \partial u)]$
HL{2,4}=[0.*t]; % $[\partial^2 L/(\partial x_1\ \partial u)]$
HL{3,4}=[0.*t]; % $[\partial^2 L/(\partial x_2\ \partial u)]$
HL{4,4}=[0.*t]; % $[\partial^2 L/(\partial u^2)]$
```

```
nE=nt;
Ez=zeros(nE,nE);
HE=num2cell(Ez); % Hessian of $E(t_f)$
```

```
Hg=[]; % Hessian of g(x,u,p,t)
Hb=[]; % Hessian of b(x0,xf,u0,uf,p,tf)
```

If the Hessian of some component of the Lagrangian is not available, set the corresponding output term equal to the empty matrix. For instance if the Hessian of the dynamical system is not available set `Hf=[]`.
Each Hessian has to be specified in a two dimensional cell array of dimension given by the size of $y$ defined as $[t,\ p,\ x,\ u]$ or $[t_f,\ p,\ x_0,\ u_0,\ u_f,\ x_f]$ or a subset of their variables. Each entry $\{i,j\}$ of the cell array corresponds to the derivative with respect to $y(i)$ and $y(j)$ The order to follow is given by the order of the vector variables inside $[t,\ p,\ x,\ u]$ or $[t_f,\ p,\ x_0,\ u_0,\ u_f,\ x_f]$. If $t_f$ (or/and $p$) is not a decision variable the derivative with respect to time (or/and $p$) must not be considered i.e the Hessian is computed with respect to $y=[p,\ x,\ u]$ or ($y=[t,\ x,\ u]$ or $y=[x,\ u]$. The same syntax has to be applied for $y=[t_f,\ p,\ x_0,\ u_0,\ u_f,\ x_f]$. The Hessian with respect to the variable $x$ and $u$ must be specified

If some variables do not contribute to the Hessian, the corresponding entries can be set to zero, otherwise the entries have to be vectors with the same length of `t`, containing the value of the Hessian at different time instants.

Notice that, in the example, the linear equations of dynamical system do not contribute to the Hessian; as a consequence all entries can be set to zero in the following way

```
fz=zeros(nfz,nfz);
Hf=num2cell(fz);
```

where nfz gives the dimension of the vector $y=[t,\ x,\ u]$.
The Hessian of $E(x_0,x_f,u_0,u_f,p,t_f)$ has to be considered on the basis of the specified analytic gradient when its evaluation is enabled (`dE.flag=1`). Since only `dE.dtf` has been specified (the other Jacobin are set to empty matrices) in the file `gradCost.m`, the Hessian only with respect to $t_f$ must be considered. If `dE.flag=0` the Hessian with respect to the variables `x0`, `u0`, `uf` and `xf` must be always specified and it would be the following:

```
nE=nt+np+2*n+2*m;
Ez=zeros(nE,nE);
HE=num2cell(Ez); % Hessian of E(x0,xf,u0,uf,tf)
```

### Solution of the problem and results

The solution of the optimisation problem is computed running the file `main.m`. The following lines are executed:

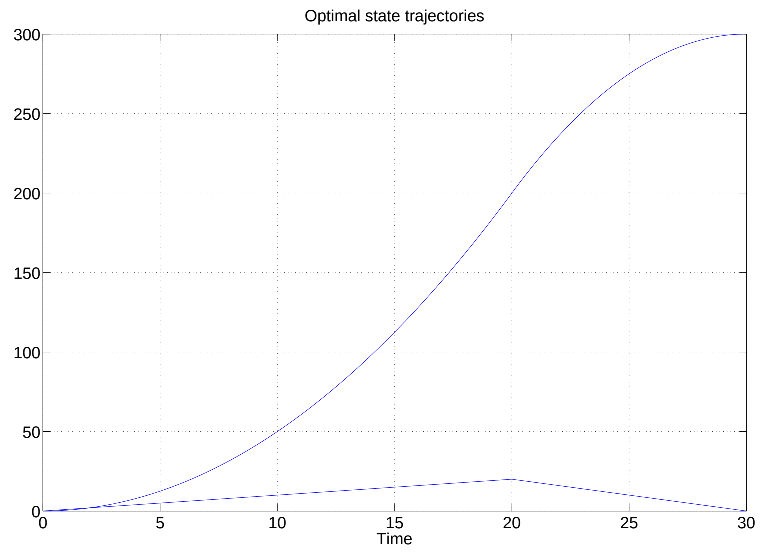

Figure 1: State trajectories for Bang-Bang problem

```
clear all;format compact;
[problem,guess]=BangBang; % Fetch the problem definition
options= settings; % Get options and solver settings
% Format for the solver
[infoNLP,data]=transcribeOCP(problem,guess,options);
[solution,status] = solveNLP(infoNLP,data); % Solve the problem
output(solution,options,data); % Output solutions
```

The state, control and adjoint variables solution to this problem using the Optimal Control Toolbox are shown in Figs. 1, 2 and 3.

## 4.2 Example 2: Fed-batch fermentor

Find the control input $u \in \mathbb{R}$ over $t$ in $[0,\ t_f]$ solving the following optimisation problem [4]

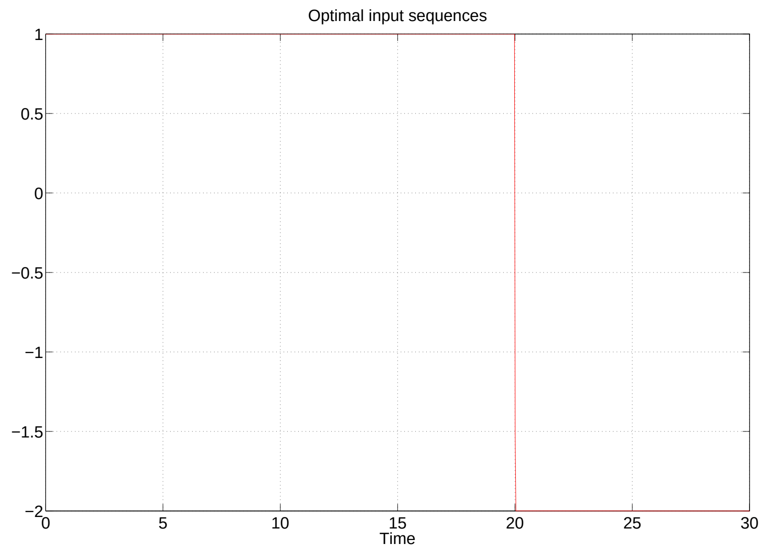

Figure 2: Input trajectory for Bang-Bang problem

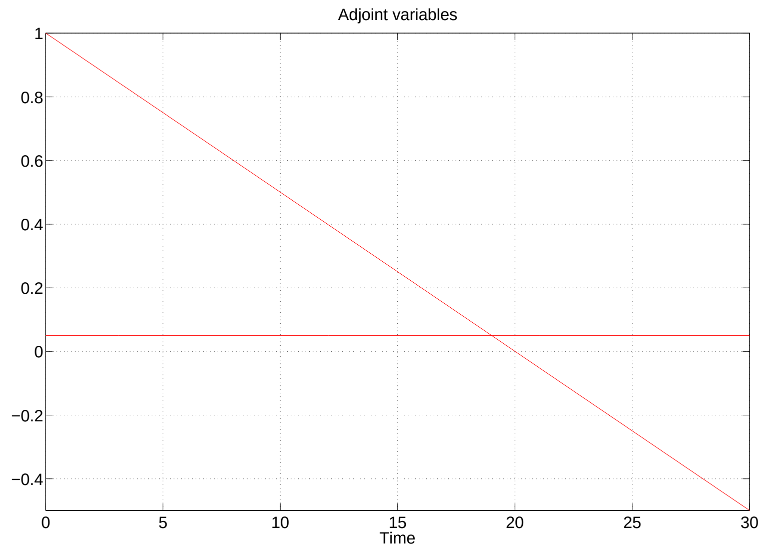

Figure 3: Adjoint variables for Bang-Bang problem

$$\min_{u(\cdot)} -x_2(t_f)x_4(t_f) + \int_0^{t_f} 0.00001u(t)^2 dt;$$

subject to

$$\begin{cases} \dot{x}_1 = h_1 x_1 - u\left(\dfrac{x_1}{500x_4}\right) \\ \dot{x}_2 = h_2 x_1 - 0.01x_2 - u\left(\dfrac{x_2}{500x_4}\right) \\ \dot{x}_3 = -h_1\dfrac{x_1}{0.47} - h_2\dfrac{x_1}{1.2} - x_1\left(\dfrac{0.029x_3}{0.0001+x_3}\right) + \dfrac{u}{x_4}\left(1-\dfrac{x_3}{500}\right) \\ \dot{x}_4 = \dfrac{u}{500} \end{cases}$$

$$h_1 = 0.11\left(\frac{x_3}{0.006x_1 + x_3}\right)$$
$$h_2 = 0.0055\left(\frac{x_3}{0.0001 + x_3(1+10x_3)}\right)$$
$$x(0) = [1.5,\ 0,\ 0,\ 7]$$

$$0 \le u(t) \le 50$$
$$\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \le \begin{bmatrix} x_1(t) \\ x_2(t) \\ x_3(t) \\ x_4(t) \end{bmatrix} \le \begin{bmatrix} 40 \\ 50 \\ 25 \\ 10 \end{bmatrix} \quad \forall t \in [0,\ t_f]$$
$$tf = 126$$

**Problem setup**

- The Optimal Control Problem is defined in the file `BatchFermentor.m` in the following way:

```
problem.time.t0=0; % Initial time t_0

% Final time t_f is fixed: tf_min=tf_max.
problem.time.tf_min=126;
problem.time.tf_max=126;
guess.tf=126;

% Parameters bounds. pl=< p <=pu
problem.parameters.pl=[];
problem.parameters.pu=[];
guess.parameters=[];

% Initial condition x_0 and its bounds
problem.states.x0=[1.5,  0.0,  0.0,  7];
problem.states.x0l=[1.5,  0.0,  0.0,  7]; % Bounds for x_0
problem.states.x0u=[1.5,   0.0,  0.0,  7];

% State bounds: $xl \leq x(t) \leq xu$
problem.states.xl=[0,  0,  0,  0];
problem.states.xu=[40,  50,  25,  10];

% Terminal state bounds: $xfl \leq x(t_f) \leq xfu$
problem.states.xfl=[0,  0,   0,  0];
problem.states.xfu=[40,   50,  25,  10];

% Guess the state trajectories: $[x(t_0),\ x(t_f)]$
guess.states(:,1)=[1.5,  30]; % $[x_1(t_0),\ x_1(t_f)]$
guess.states(:,2)=[0, 8.5]; % $[x_2(t_0),\ x_2(t_f)]$
guess.states(:,3)=[0,  0]; % $[x_3(t_0),\ x_3(t_f)]$
guess.states(:,4)=[7,  10]; % $[x_4(t_0),\ x_4(t_f)]$

% Number of control actions
problem.inputs.N=500;

% Input bounds: $ul \leq u(t) \leq uu$
problem.inputs.ul=[0];
problem.inputs.uu=[50];

% Guess the input sequences: $[u(t_0),\ u(t_f)]$
guess.inputs(:,1)=[2, 10];

% Choose the set-points if required
problem.setpoints.states=[];
problem.setpoints.inputs=[];

% Bounds for path constraint function gl $\leq g(x,u,p,t) \leq$ gu
problem.constraints.gl=[];
problem.constraints.gu=[];

% Bounds for boundary constraints bl $\leq b(x(t_0),x(t_f),u(t_0),u(t_f),p,t_0,t_f) \leq$ bu
problem.constraints.bl=[];
problem.constraints.bu=[];

% store the necessary problem parameters used in the functions
problem.data=[];

problem.functions={@L,@E,@f,@g,@b};
```

```
function stageCost=L(x,xr,u,ur,p,t,data)

    stageCost =0.00001*u(:,1).*u(:,1);

function boundaryCost=E(x0,xf,u0,uf,p,tf,data)

    boundaryCost=-xf(2)*xf(4);

function dx = f(x,u,p,t,data)

    x1=x(:,1); x2=x(:,2); x3=x(:,3); x4=x(:,4);
    u1=u(:,1);

    h1=0.11*(x3./(0.006*x1+x3));
    h2=0.0055*(x3./(0.0001+x3.*(1+10*x3)));

    dx(:,1)=(h1.*x1-u1.*(x1./500./x4));
    dx(:,2)=(h2.*x1-0.01*x2-u1.*(x2./500./x4));
    dx(:,3)=(-h1.*x1/0.47-h2.*x1/1.2-x1.*(0.029*x3./(0.0001+x3))...
            +u1./x4.*(1-x3/500));
    dx(:,4) = u1/500;

function c=g(x,u,p,t,data)

    c=[];
function bc=b(x0,xf,u0,uf,p,tf,data)

    bc=[];
```

- The solution method and solver settings are set in `settings.m` . See the file included in the directory *../ICLOCS-*.*/examples/BatchFermentor*

- The files `gradCost.m`, `jacConst.m` and `hessianLagrangian.m` for this example are supplied. See inside the directory *../ICLOCS-*.*/examples/BatchFermentor.*

  - `gradCost.m`:

```
    function [dL,dE]=gradCost(L,X,Xr,U,Ur,P,t,E,x0,xf,u0,uf,p,tf,data)

    n=deal(data.sizes{3});
    Lt=ones(size(t));
```

```
% dL - Gradient of the stage cost L(·) wrt. t, p, x, u
dL.flag=1;
dL.dp=[];
dL.dt=[];
dL.dx=kron(zeros(1,n),Lt);
dL.du=2*0.00001*U(:,1);
```

```
% dE - Gradient of E(·) with respect to tf, p, x0, u0, uf, xf
dE.flag=1;
dE.dtf=[];
dE.dp=[];
dE.dx0=[];
dE.du0=[];
dE.dxf=[0 -xf(4) 0 -xf(2)];
dE.duf=[];
```

The gradient of the stage cost and the boundary cost are supplied and so `dL.flag=1` and `dE.flag=1`. The final time $t_f$ and $p$ are not decision variables and then the derivatives `dL.dp`, `dL.dt`, `dE.dtf`, and `dE.dp` are set to be empty matrices. The derivative of the stage cost with respect to the state $u$ is stored in a matrix with $m$ columns and a number of rows given by the different evaluation times. It is expressed in term of the input evaluated at different time instant. The $i^{\text{th}}$ input $u_i$, evaluated at the time instants $t = [t_0, \ldots, t_k, \ldots, tf]$ is a column vectors taken as `U(:,i)`. The derivative of the stage cost with respect to the state $x$ must be specified even if it is identically zero. The derivative of the boundary cost depends only on the final state $x_f$ and then the variables containing the derivatives with respect to the other decision variables are set to be empty matrices.

- `jacConst.m`:

```
function [df,dg,db]=jacConst(f,g,X,U,P,t,b,x0,xf,u0,uf,p,tf,t0,data)

Lt=ones(size(t));

df.flag=1;

h1 = 0.11*(X(:,3)./(0.006*X(:,1)+X(:,3))); % h1(x1,x3)
h2 = 0.0055*(X(:,3)./(0.0001+X(:,3).*(1+10*X(:,3)))); % h2(x3)

% ∂h1(x1,x3)/∂x1
dh1x1=-0.11*0.006*X(:,3)./((0.006*X(:,1)+X(:,3)).^2);

% ∂h1(x1,x3)/∂x3
dh1x3=0.11*0.006*X(:,1)./((0.006*X(:,1)+X(:,3)).^2);
```

% $\partial h_2(x_3)/\partial x_3$
```
dh2= 0.0055*(0.0001-10*X(:,3).^2)./((0.0001+X(:,3).*(1+10*X(:,3))).^2);
```

```
df.dp{1}=[];  % ∂f(x,u,p,t)/∂p
df.dt{1}=[];  % ∂f(x,u,p,t)/∂t
```

% $\partial f(x,u,p,t)/\partial x_1 = [\partial f_1(\cdot)/\partial x_1, \dots, \partial f_n(\cdot)/\partial x_1]$
```
df.dx{1}=[h1+dh1x1.*X(:,1)-U(:,1)./500./X(:,4), h2,...
          -h1./0.47-dh1x1.*X(:,1)/0.47-h2/1.2...
          -(0.029*X(:,3)./(0.0001+X(:,3))), 0*Lt];
```

% $\partial f(x,u,p,t)/\partial x_2 = [\partial f_1(\cdot)/\partial x_2, \dots, \partial f_n(\cdot)/\partial x_2]$
```
df.dx{2}=[0*Lt,-0.01-U(:,1)./500./X(:,4), 0*Lt,0*Lt];
```

% $\partial f(x,u,p,t)/\partial x_3 = [\partial f_1(\cdot)/\partial x_3, \dots, \partial f_n(\cdot)/\partial x_3]$
```
df.dx{3}=[dh1x3.*X(:,1),dh2.*X(:,1), -dh1x3.*X(:,1)/0.47-...
         dh2.*X(:,1)/1.2-X(:,1).*(0.029*0.0001./((0.0001+X(:,3)).^2))...
          -U(:,1)./X(:,4)/500,0*Lt];
```

% $\partial f(x,u,p,t)/\partial x_4 = [\partial f_1(\cdot)/\partial x_4, \dots, \partial f_n(\cdot)/\partial x_4]$
```
df.dx{4}=[U(:,1).*X(:,1)./500./(X(:,4).^2),...
          U(:,1).*X(:,2)./500./(X(:,4).^2),...
          -U(:,1)./(X(:,4).^2).*(1-X(:,3)/500), 0*Lt];
```

% $\partial f(x,u,p,t)/\partial u = [\partial f_1(\cdot)/\partial u, \dots, \partial f_n(\cdot)/\partial u]$
```
df.du{1}=[-X(:,1)./500./X(:,4), -X(:,2)./500./X(:,4),...
          (1-X(:,3)/500)./X(:,4), Lt/500];
```

```
dg.flag=0;
db.flag=0;
```

The path constraints and boundary constraints are not present and then their derivatives have not be supplied (`dg.flag=0` and `db.flag=0`). Instead the derivatives of $f(x,u,p,t) = [f_1(x,u,p,t), \dots, f_n(x,u,p,t)]$ are supplied in the structured variable `df` as shown in the illustrated example. The derivative of $f(x,u,p,t)$ with respect to $x$ are stored in df.dx which is a cell array where each entry corresponds to the derivative with respect to a state variable. For instance the derivative of $f(x,u,p,t)$ with respect to $x_i$ is stored in `df.dx{i}`. `df.dx{i}` is a matrix with $n$ columns and a number of rows given by the different evaluation times. Each row of `df.dx{i}` corresponds to the evaluation of the derivative $[\partial f_1(x,u,p,t)/\partial x_i, \dots, \partial f_n(x,u,p,t)/\partial x_i]$ at a given time instant. They are expressed in term of the $i^{\text{th}}$ state and input $x_i$ and $u_i$, evaluated at the time instants $t = [t_0, \dots, t_k, \dots, tf]$ and taken as column vectors `X(:,i)` and `U(:,i)`.

- `hessianLagrangian.m`

```
function [HL,HE,Hf,Hg,Hb]=hessianLagrangian(X,U,P,t,E,x0,xf,u0,uf,p,tf,data)
```

The Hessian of the different parts of the Lagrangian (`HL, HE, Hf, Hg, Hb`) must be supplied in cell arrays as follows:

```
[nt,np,n,m,ng,nb]=deal(data.sizes{1:6});
nfz=nt+np+n+m;

dh1x1=-0.11*0.006*X(:,3)./((0.006*X(:,1)+X(:,3)).^2); % $\partial h_1(x_1,x_3)/\partial x_1$
dh1x3=0.11*0.006*X(:,1)./((0.006*X(:,1)+X(:,3)).^2); % $\partial h_1(x_1,x_3)/\partial x_3$

% $\partial h_2(x_3)/\partial x_3$
dh2= 0.0055*(0.0001-10*X(:,3).^2)./((0.0001+X(:,3).*(1+10*X(:,3))).^2);

% $\partial^2 h_2(x_3)/\partial^2 x_3$
d2h2=0.0055*2*(-30.*X(:,3)*0.0001+100*X(:,3).^3-0.0001)./...
        (0.0001+X(:,3)+10*X(:,3).^2).^3;
% $\partial^2 h_1(x_1,x_3)/\partial^2 x_1$
d2h1x1=2*0.11*0.006^2*X(:,3)./((0.006*X(:,1)+X(:,3)).^3);
% $\partial^2 h_1(x_1,x_3)/(\partial x_1\ \partial x_3)$
d2h1x1x3=-0.11*0.006*(0.006*X(:,1)-X(:,3))./((0.006*X(:,1)+X(:,3)).^3);
% $\partial^2 h_1(x_1,x_3)/\partial^2 x_3$
d2h1x3=-2*0.11*0.006*X(:,1)./((0.006*X(:,1)+X(:,3)).^3);

Hf=cell(nfz, nfz); % Hessian of $f(x,u,p,t)$

% $[\partial^2 f_1(\cdot)/\partial x_1^2,\ \partial^2 f_2(\cdot)/\partial x_1^2,\ \partial^2 f_3(\cdot)/\partial x_1^2,\ \partial^2 f_4(\cdot)/\partial x_1^2]$
Hf{1,1}=[2*dh1x1+d2h1x1, 0.*t, -(2*dh1x1+d2h1x1)/0.47, 0.*t];

% $[\partial^2 f_1(\cdot)/(\partial x_1\partial x_2),\ \partial^2 f_2(\cdot)/(\partial x_1\partial x_2),\ \partial^2 f_3(\cdot)/(\partial x_1\partial x_2),\ \partial^2 f_4(\cdot)/(\partial x_1\partial x_2)]$
Hf{1,2}=[0.*t, 0.*t, 0.*t, 0.*t];
% $[\partial^2 f_1(\cdot)/\partial x_2^2,\ \partial^2 f_2(\cdot)/\partial x_2^2,\ \partial^2 f_3(\cdot)/\partial x_2^2,\ \partial^2 f_4(\cdot)/\partial x_2^2]$
Hf{2,2}=[0.*t, 0.*t, 0.*t, 0.*t];

% $[\partial^2 f_1(\cdot)/(\partial x_1\partial x_3),\ \partial^2 f_2(\cdot)/(\partial x_1\partial x_3),\ \partial^2 f_3(\cdot)/(\partial x_1\partial x_3),\ \partial^2 f_4(\cdot)/(\partial x_1\partial x_3)]$
Hf{1,3}=[dh1x3+ d2h1x1x3.*X(:,1), dh2,...
        -dh1x3./0.47-d2h1x1x3.*X(:,1)/0.47-dh2/1.2...
        -(0.029*0.0001./((0.0001+X(:,3)).^2)), 0.*t];

% $[\partial^2 f_1(\cdot)/(\partial x_1\partial x_4),\ \partial^2 f_2(\cdot)/(\partial x_1\partial x_4),\ \partial^2 f_3(\cdot)/(\partial x_1\partial x_4),\ \partial^2 f_4(\cdot)/(\partial x_1\partial x_4)]$
Hf{1,4}=[U(:,1)./500./(X(:,4).^2), 0.*t, 0.*t, 0.*t];
```

```
% $[\partial^2 f_1(\cdot)/(\partial x_2 \partial x_3),\ \partial^2 f_2(\cdot)/(\partial x_2 \partial x_3),\ \partial^2 f_3(\cdot)/(\partial x_2 \partial x_3),\ \partial^2 f_4(\cdot)/(\partial x_2 \partial x_3)]$
Hf{2,3}=[0.*t, 0.*t, 0.*t, 0.*t];

% $[\partial^2 f_1(\cdot)/\partial x_3^2,\ \partial^2 f_2(\cdot)/\partial x_3^2,\ \partial^2 f_3(\cdot)/\partial x_3^2,\ \partial^2 f_4(\cdot)/\partial x_3^2]$
Hf{3,3}=[d2h1x3.*X(:,1), d2h2.*X(:,1), -d2h1x3.*X(:,1)/0.47...
        -d2h2.*X(:,1)/1.2+2*X(:,1).*(0.029*0.0001./...
        ((0.0001+X(:,3)).^3)), 0.*t];

% $[\partial^2 f_1(\cdot)/(\partial x_2 \partial x_4),\ \partial^2 f_2(\cdot)/(\partial x_2 \partial x_4),\ \partial^2 f_3(\cdot)/(\partial x_2 \partial x_4),\ \partial^2 f_4(\cdot)/(\partial x_2 \partial x_4)]$
Hf{2,4}=[0.*t, U(:,1)./500./(X(:,4).^2), 0.*t, 0.*t];

% $[\partial^2 f_1(\cdot)/(\partial x_3 \partial x_4),\ \partial^2 f_2(\cdot)/(\partial x_3 \partial x_4),\ \partial^2 f_3(\cdot)/(\partial x_3 \partial x_4),\ \partial^2 f_4(\cdot)/(\partial x_3 \partial x_4)]$
Hf{3,4}=[0.*t, 0.*t, U(:,1)./500./(X(:,4).^2), 0.*t];

% $[\partial^2 f_1(\cdot)/\partial x_4^2,\ \partial^2 f_2(\cdot)/\partial x_4^2,\ \partial^2 f_3(\cdot)/\partial x_4^2,\ \partial^2 f_4(\cdot)/\partial x_4^2]$
Hf{4,4}=[-2*U(:,1).*X(:,1)./500./(X(:,4).^3),...
        -2*U(:,1).*X(:,2)./500./(X(:,4).^3),...
        2*U(:,1)./(X(:,4).^3).*(1-X(:,3)/500), 0.*t];

% $[\partial^2 f_1(\cdot)/(\partial x_1 \partial u),\ \partial^2 f_2(\cdot)/(\partial x_1 \partial u),\ \partial^2 f_3(\cdot)/(\partial x_1 \partial u),\ \partial^2 f_4(\cdot)/(\partial x_1 \partial u)]$
Hf{1,5}=[ -1./500./X(:,4), 0.*t, 0.*t , 0.*t];

% $[\partial^2 f_1(\cdot)/(\partial x_2 \partial u),\ \partial^2 f_2(\cdot)/(\partial x_2 \partial u),\ \partial^2 f_3(\cdot)/(\partial x_2 \partial u),\ \partial^2 f_4(\cdot)/(\partial x_2 \partial u)]$
Hf{2,5}=[0.*t, -1./500./X(:,4), 0.*t , 0.*t];

% $[\partial^2 f_1(\cdot)/(\partial x_3 \partial u),\ \partial^2 f_2(\cdot)/(\partial x_3 \partial u),\ \partial^2 f_3(\cdot)/(\partial x_3 \partial u),\ \partial^2 f_4(\cdot)/(\partial x_3 \partial u)]$
Hf{3,5}=[0.*t, 0.*t, -1./500./X(:,4), 0.*t];

% $[\partial^2 f_1(\cdot)/(\partial x_4 \partial u),\ \partial^2 f_2(\cdot)/(\partial x_4 \partial u),\ \partial^2 f_3(\cdot)/(\partial x_4 \partial u),\ \partial^2 f_4(\cdot)/(\partial x_4 \partial u)]$
Hf{4,5}=[X(:,1)./500./(X(:,4).^2), X(:,2)./500./(X(:,4).^2),...
        -1./(X(:,4).^2).*(1-X(:,3)/500) , 0.*t];

% $[\partial^2 f_1(\cdot)/\partial u^2,\ \partial^2 f_2(\cdot)/\partial u^2,\ \partial^2 f_3(\cdot)/\partial u^2,\ \partial^2 f_4(\cdot)/\partial u^2]$
Hf{5,5}=[0.*t, 0.*t, 0.*t, 0.*t];

Lt=ones(size(t));

HL=cell(nfz, nfz);
HL{1,1}=[0.*t]; % $[\partial^2 L(\cdot)/\partial x_1^2]$
HL{1,2}=[0.*t]; % $[\partial^2 L(\cdot)/(\partial x_2 \partial x_1)]$
HL{2,2}=[0.*t]; % $[\partial^2 L(\cdot)/(\partial x_2^2)]$
HL{1,3}=[0.*t]; % $[\partial^2 L(\cdot)/(\partial x_1 \partial x_3)]$
```

```
HL{2,3}=[0.*t]; % $[\partial^2 L(\cdot)/(\partial x_2 \partial x_3)]$
HL{3,3}=[0.*t]; % $[\partial^2 L(\cdot)/(\partial x_3^2)]$
HL{1,4}=[0.*t]; % $[\partial^2 L(\cdot)/(\partial x_1 \partial x_4)]$
HL{2,4}=[0.*t]; % $[\partial^2 L(\cdot)/(\partial x_2 \partial x_4)]$
HL{3,4}=[0.*t]; % $[\partial^2 L(\cdot)/(\partial x_3 \partial x_4)]$
HL{4,4}=[0.*t]; % $[\partial^2 L(\cdot)/(\partial x_4^2)]$
HL{1,5}=[0.*t]; % $[\partial^2 L(\cdot)/(\partial x_1 \partial u)]$
HL{2,5}=[0.*t]; % $[\partial^2 L(\cdot)/(\partial x_2 \partial u)]$
HL{3,5}=[0.*t]; % $[\partial^2 L(\cdot)/(\partial x_3 \partial u)]$
HL{4,5}=[0.*t]; % $[\partial^2 L(\cdot)/(\partial x_4 \partial u)]$
HL{5,5}=[2*0.00001.*Lt]; % $[\partial^2 L(\cdot)/(\partial u^2)]$


nE=n;
Ez=zeros(nE,nE);
HE=num2cell(Ez);
HE{2,nE}=-1;


Hg=[];
Hb=[];
```

Each Hessian is specified in a two dimensional cell array of dimension given by the size of $y = [x,\ u]$ or $y = [x_f]$. Each entry $\{i, j\}$ of the cell array corresponds to the derivative with respect to $y(i)$ and $y(j)$ The order to follow is given by the order of the vector variable $y = [x,\ u]$ or $y = [x_f]$. If some variables do not contribute to the Hessian, the corresponding entries can be set to zero, otherwise the entries have to be vectors with the same length of t, containing the value of the Hessian at different time instants.
For instance, in the example, the Hessian of the stage cost can be specified in a compact way as follows

```
Lz=zeros(nfz,nfz);
Hf=num2cell(Lz);
HL{5,5}=[2*0.00001.*Lt];
```

where nfz gives the dimension of the vector $y = [x,\ u]$.

If `dE.flag=0` the Hessian with respect to the vector $y = [x0, u0, uf, xf]$ must be specified as follows:

```
nE=nt+np+2*n+2*m; Ez=zeros(nE,nE);
HE=num2cell(Ez); % Hessian of $E(x_0, x_f, u_{0,} u_f, p, t_f)$
HE{8,nE}=-1; % $\partial^2 E(x_0, x_f, u_{0,} u_f, p, t_f)/(\partial x_2(tf) \partial x_4(tf))$
```

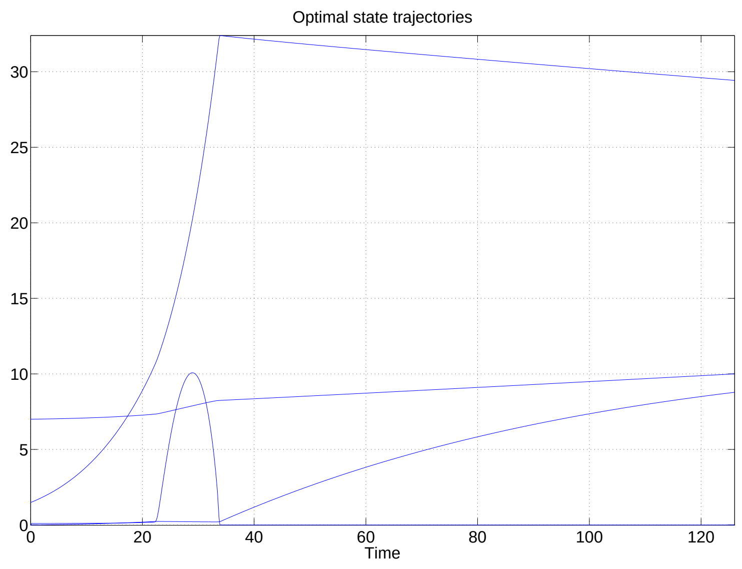

Figure 4: State trajectories for Fed-batch fermentor problem

### Solution of the problem and results

The solution of the optimisation problem is computed running the file `main.m`. The following lines are executed:

```
clear all;format compact;
[problem,guess]=BangBang; % Fetch the problem definition
options= settings; % Get options and solver settings
% Format for the solver
[infoNLP,data]=transcribeOCP(problem,guess,options);

% Initialize the dual point
nc=size(data.jacStruct,1);
[nt,np,n,m,M,N]=deal(data.sizes[1:4,7:8]);
nz=nt+np+n*M+m*N;
data.multipliers.zl=2*ones(1,nz);
data.multipliers.zu=2*ones(1,nz);
data.multipliers.lambda=2*ones(1,nc);

[solution,status] = solveNLP(infoNLP,data); % Solve the problem
output(solution,options,data); % Output solutions
```

The state, control and adjoint variables solution to this problem using the Optimal Control Toolbox are shown in Figs. 4, 5 and 6.

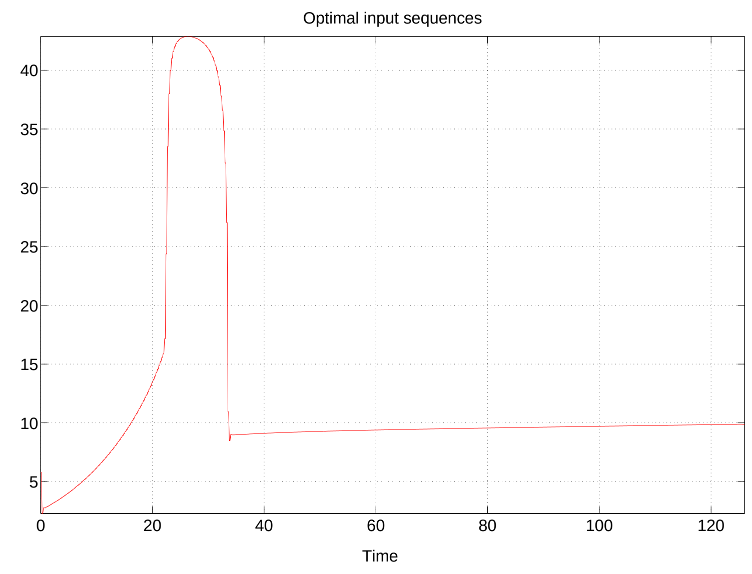

Figure 5: Input trajectory for Fed-batch fermentor problem

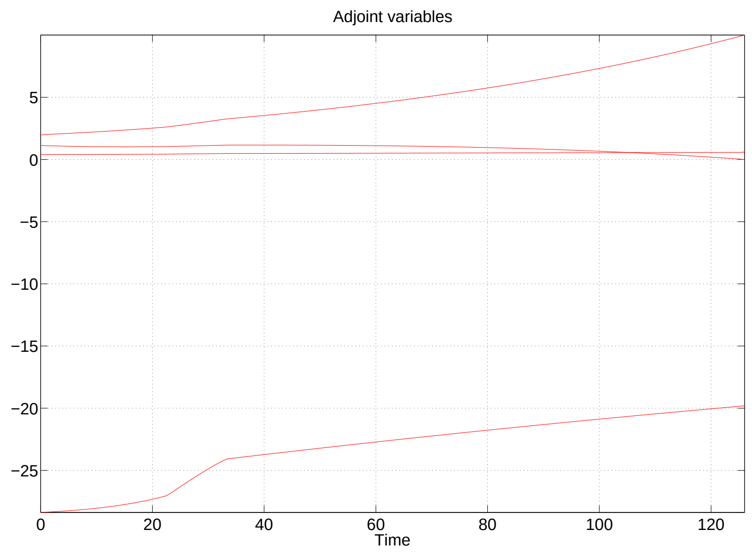

Figure 6: Adjoint variables for Fed-batch fermentor problem

## 4.3 Example 3: Bryson-Denham optimal control problem

Find the final time $t_f$ and control input $u \in \mathbb{R}$ over $t$ in $[0,\ t_f]$ solving the following optimisation problem [2]

$$\min_{u(\cdot),\ t_f} x_3(t_f)$$

subject to

$$\begin{cases} \dot{x}_1 = x_2 \\ \dot{x}_2 = u \\ \dot{x}_3 = \dfrac{u^2}{2} \end{cases}$$

$$x(0) = [0,\ 1,\ 0]$$
$$x_1(t_f) = 0,\ x_2(t_f) = -1$$

$$-10 \leq u(t) \leq 10$$
$$\begin{bmatrix} 0 \\ -20 \\ -20 \end{bmatrix} \leq \begin{bmatrix} x_1(t) \\ x_2(t) \\ x_3(t) \end{bmatrix} \leq \begin{bmatrix} 1/9 \\ 20 \\ 20 \end{bmatrix} \quad \forall t \in [0,\ t_f]$$
$$-20 \leq x_3(t_f) \leq 20$$

**Problem setup**

- The Optimal Control Problem is defined in the file `BrysonDenham.m` in the following way:

%Initial time. $t_0 < t_f$
```
problem.time.t0=0;
```

% Final time. Let tf_min=tf_max if $t_f$ is fixed.
```
problem.time.tf_min=0.1;
problem.time.tf_max=100;
guess.tf=1;
```

% Parameters bounds. pl=< p <=pu
```
problem.parameters.pl=[];
problem.parameters.pu=[];
guess.parameters=[];
```

% Initial condition $x_0$ and its bounds
```
problem.states.x0=[0 1 0];
problem.states.x0l=[0 1 0];
problem.states.x0u=[0 1 0];
```

% State bounds: xl $\leq x(t) \leq$xu

```
problem.states.xl=[0 -20 -20];
problem.states.xu=[1/9 20 20];
```

% Terminal state bounds: xfl $\leq x(t_f) \leq$ xfu

```
problem.states.xfl=[0 -1 -20];
problem.states.xfu=[0 -1 20];
```

% Guess the state trajectories with $[x(t_0),\ x(t_f)]$

```
guess.states(:,1)=[0 0];
guess.states(:,2)=[1 -1];
guess.states(:,3)=[0 0];
```

Number of control actions $N$. If $N$ is equal to the number of integration steps, problem.inputs.N can be set to 0

```
problem.inputs.N=0;
```

% Input bounds: ul $\leq u(t) \leq$ uu

```
problem.inputs.ul=[-10];
problem.inputs.uu=[10];
```

% Guess the input sequences with $[u(t_0),\ u(t_f)]$

```
guess.inputs(:,1)=[-5 -5];
```

% Choose the set-points if required

```
problem.setpoints.states=[];
problem.setpoints.inputs=[];
```

% Bounds for path constraint function gl $\leq g(x,u,p,t) \leq$ gu

```
problem.constraints.gl=[];
problem.constraints.gu=[];
```

% Bounds for boundary constraints bl $\leq b(x(t_0),x(t_f),u(t_0),u(t_f),p,t_0,t_f) \leq$ bu

```
problem.constraints.bl=[];
problem.constraints.bu=[];
```

% store the necessary problem parameters used in the functions

```
problem.data=[];


problem.functions={@L,@E,@f,@g,@b};


function stageCost=L(x,xr,u,ur,p,t,data)
```

```
    stageCost = 0*t;

function boundaryCost=E(x0,xf,u0,uf,p,tf,data)

    boundaryCost=xf(3);

function dx = f(x,u,p,t,data)

    x1 = x(:,1);x2 = x(:,2);u1 = u(:,1);
    dx(:,1) = x2;
    dx(:,2) = u1;
    dx(:,3)=u1.*u1/2;

function c=g(x,u,p,t,data)

    c=[];

function bc=b(x0,xf,u0,uf,p,tf,data)

    bc=[];
```

- The solution method and solver settings are set in `settings.m` . See the file included in the directory *../ICLOCS-*.*/examples/BrysonDenham*

- The solution of the optimisation problem is computed running the file `main.m` inside the directory *../ICLOCS-*.*/examples/BrysonDenham.* The state, control and adjoint variables solution to this problem using the Optimal Control Toolbox are shown in Figs. 7, 8 and 9.

## 4.4 Example 4: Optimal Mixing of a Catalyst

This mixing problem considers a plug-flow reactor, packed with two catalysts, involving the reactions

$$S_1 \leftrightarrow S_2 \rightarrow S_3$$

The optimal mixing policy of the two catalysts has to be determined in order to maximise the production of species $S_3$ [12]. The optimisation problem is formulated as follow

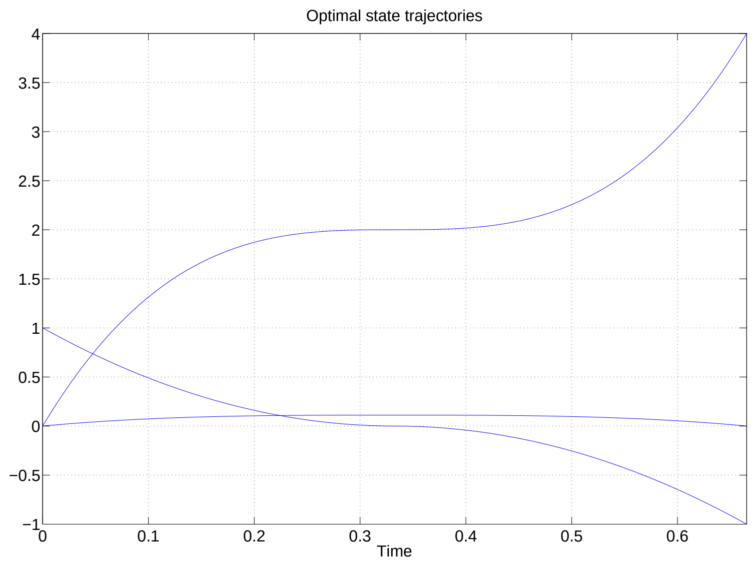

Figure 7: State trajectories for Bryson-Denham problem

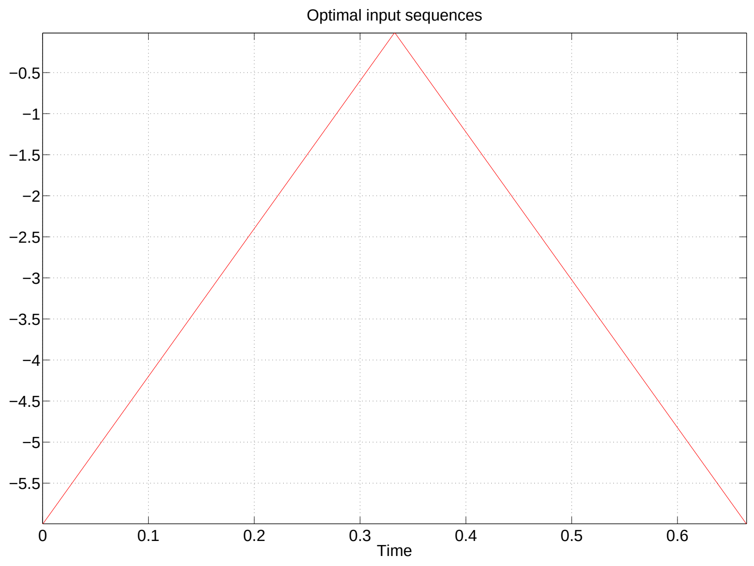

Figure 8: Input trajectory for Bryson-Denham problem

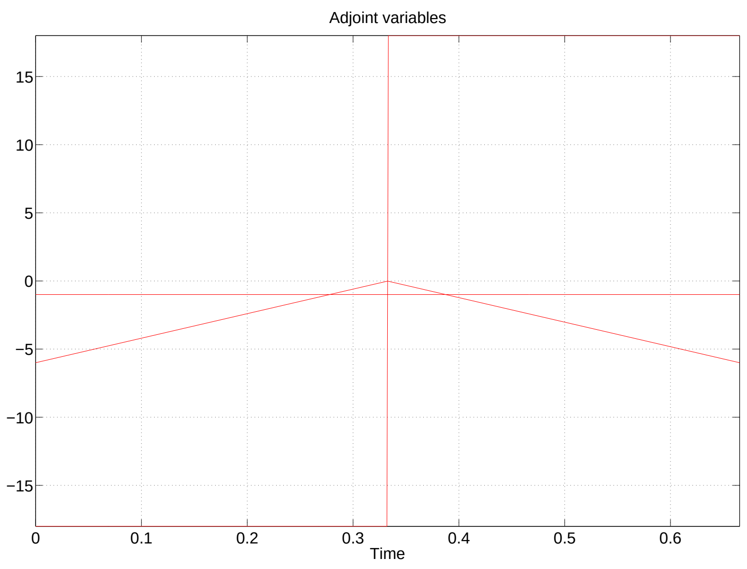

Figure 9: Adjoint variables for Bryson-Denham problem

$$\min_{u(\cdot)} -1 + x_1(t_f) + x_2(t_f);$$

subject to

$$\begin{cases} \dot{x}_1 = u(10x_2 - x_1) \\ \dot{x}_2 = u(x_1 - 10x_2) - (1-u)x_2; \end{cases}$$

$$x(0) = [1,\ 0]$$
$$t_f = 1$$

$$0 \le u(t) \le 1$$
$$\begin{bmatrix} 0.8 \\ 0 \end{bmatrix} \le \begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} \le \begin{bmatrix} 1 \\ 0.1 \end{bmatrix} \quad \forall t \in [0,\ t_f]$$
$$0.8 \le x_1(t_f) \le 0.95$$

## Problem setup

- The Optimal Control Problem is defined in the file `CatalystMixing.m` in the following way:

  % Initial time $t_0 < t_f$
  ```
  problem.time.t0=0;
  ```

  % Final time. Let tf_min=tf_max if tf is fixed.
  ```
  problem.time.tf_min=1;
  ```

```
problem.time.tf_max=1;
guess.tf=1;
```

% Parameters bounds: pl $\leq p \leq$ pu
```
problem.parameters.pl=[];
problem.parameters.pu=[];
guess.parameters=[];
```

% Initial conditions $x_0$ and its bounds
```
problem.states.x0=[1 0];
problem.states.x0l=[1 0];
problem.states.x0u=[1 0];
```

% State bounds: xl $\leq x(t) \leq$ xu
```
problem.states.xl=[0.8 0];
problem.states.xu=[1 0.1];
```

% Terminal state bounds: xfl $\leq x(t_f) \leq$ xfu
```
problem.states.xfl=[0.8 0];
problem.states.xfu=[0.95 0.1];
```

% Guess the state trajectories with $[x(t_0),\ x(t_f)]$
```
guess.states(:,1)=[1 0.915];
guess.states(:,2)=[0 0.05];
```

Number of control actions $N$. If $N$ is equal to the number of integration steps, problem.inputs.N can be set to 0
```
problem.inputs.N=0;
```

% Input bounds: ul $\leq u(t) \leq$ uu
```
problem.inputs.ul=[0];
problem.inputs.uu=[1];
```

% Guess the input sequences with $[u(t_0),\ u(t_f)]$
```
guess.inputs(:,1)=[1 0];
```

% Choose the set-points if required
```
problem.setpoints.states=[];
problem.setpoints.inputs=[];
```

% Bounds for path constraint function gl $\leq g(x,u,p,t) \leq$ gu
```
problem.constraints.gl=[];
problem.constraints.gu=[];
```

```
% Bounds for boundary constraints bl ≤ b(x(t_0), x(t_f), u(t_0), u(t_f), p, t_0, t_f) ≤ bu
problem.constraints.bl=[];
problem.constraints.bu=[];

% store the necessary problem parameters used in the functions
problem.data=[];

problem.functions={@L,@E,@f,@g,@b};

function stageCost=L(x,xr,u,ur,p,t,data)

    stageCost = t*0;

function boundaryCost=E(x0,xf,u0,uf,p,tf,data)

    boundaryCost=-1+xf(1)+xf(2);

function dx = f(x,u,p,t,data)

    x1 = x(:,1); x2 = x(:,2); u = u(:,1);

    dx(:,1) = u.*(10*x2-x1);
    dx(:,2) = u.*(x1-10*x2)-(1-u).*x2;

function c=g(x,u,p,t,data)

    c=[];

function bc=b(x0,xf,u0,uf,p,tf,data)

    bc=[];
```

- The solution method and solver settings are set in `settings.m` . See the file included in the directory *../ICLOCS-*.*/examples/CatalystMixing*

- The solution of the optimisation problem is computed running the file `main.m` inside the directory *../ICLOCS-*.*/examples/CatalystMixing* The state, control and adjoint variables solution to this problem using the Optimal Control Toolbox are shown in Figs. 10, 11 and 12.

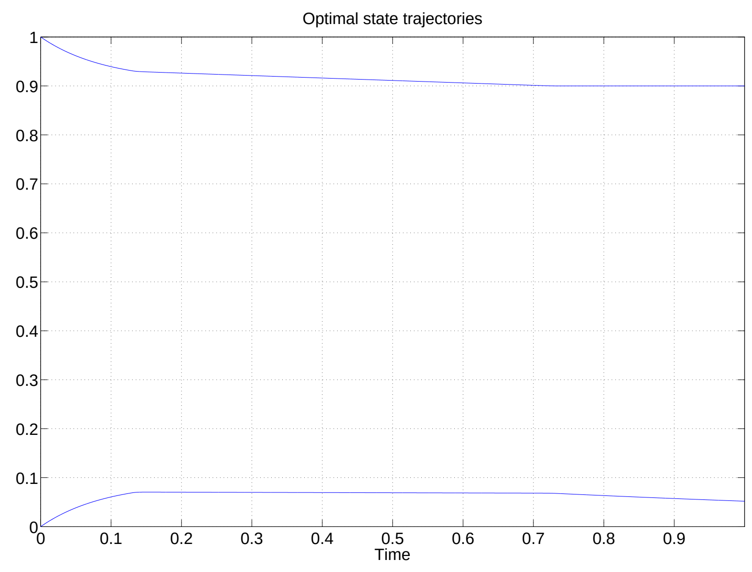

Figure 10: State trajectories for the Optimal Mixing of a Catalyst

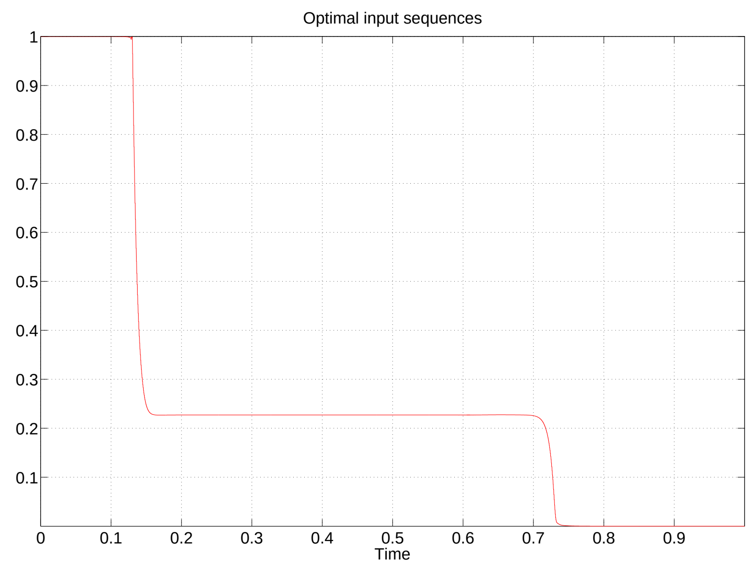

Figure 11: Input trajectory for the Optimal Mixing of a Catalyst

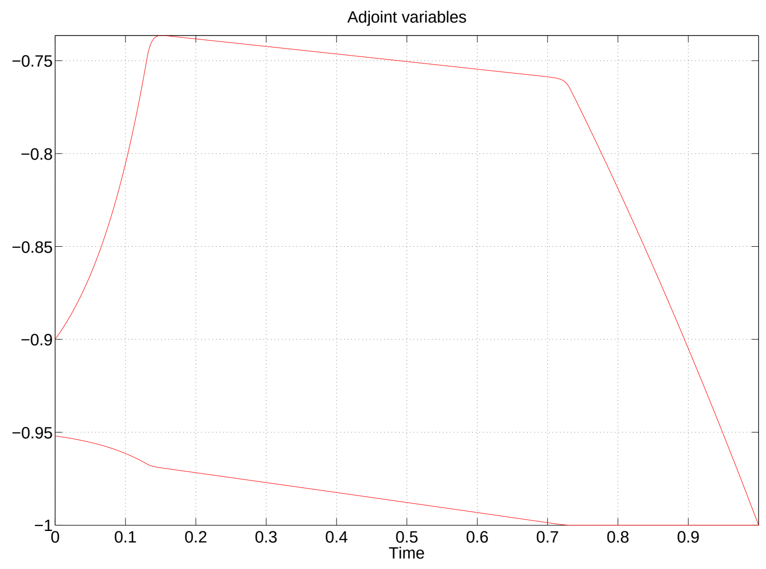

Figure 12: Adjoint variables for the Optimal Mixing of a Catalyst

## 4.5 Example 5: Discrete-time optimisation problem

At each time instant $t$, measure the current state $x$ and compute the control input sequence $\mathbf{u} \triangleq [u(0),\ u(1), \ldots, u(N-1)] \in \mathbb{R}^N$ solving the following optimisation problem

$$\min_{\mathbf{u}} \sum_{k=0}^{N-1} x(k)'Qx(k) + u(k)'Ru(k) + x(N)'Px(N);$$

subject to

$$\begin{cases} x_1(k+1) = x_1(k) + 0.1x_2(k) \\ x_2(k+1) = 1.8x_2(k) + 0.0787u(k) \end{cases}$$

$$-2 \le u(t) \le 2$$
$$x(N)'Px(N) \le gm$$

where x(0)=[0.3, 0.1], $N = 20$, $R = 10^{-4}$ and $Q$ is the identity matrix. The parameters $P$ and $gm$ define a terminal invariant set where $P$ is the positive-definite solution of the algebraic Riccati equation for the selected $Q$ and $R$.

### Problem setup

- The Optimal Control Problem is defined in the file `Discrete_Sys.m` in the following way:

% Initial time $t_0 < t_f$. For discrete-time systems is the initial index
```
problem.time.t0=0;
```

% Final time. For discrete-time systems tf_min=tf_max=[ ]
```
problem.time.tf_min=[];
problem.time.tf_max=[];
guess.tf=[];
```

% Parameter bounds: pl$\leq p \leq$ pu
```
problem.parameters.pl=[];
problem.parameters.pu=[];
guess.parameters=[];
```

% Initial conditions $x_0$ and its bounds
```
problem.states.x0=[0.3 0.1];
problem.states.x0l=problem.states.x0;
problem.states.x0u=problem.states.x0l;
```

% State bounds. xl $\leq x(t) \leq$ xu
```
problem.states.xl=[-inf -inf];
problem.states.xu=[+inf +inf];
```

% Terminal state bounds: xfl $\leq x(t_f) \leq$ xfu
```
problem.states.xfl=[-inf -inf];
problem.states.xfu=[+inf +inf];
```

% Guess the state trajectories with $[x(t_0),\ x(t_f)]$
```
guess.states(:,1)=[problem.states.x0(1) 0];
guess.states(:,2)=[problem.states.x0(2) 0];
```

Number of control actions $N$. If $N$ is equal to the number of integration steps, problem.inputs.N can be set to 0
```
problem.inputs.N=0;
```

% Input bounds: ul $\leq u(t_f) \leq$ uu
```
problem.inputs.ul=[-2];
problem.inputs.uu=[2];
```

% Guess the input sequences with $[u(t_0),\ u(t_f)]$
```
guess.inputs(:,1)=[-2 0];
```

% Bounds for path constraint function gl $\leq g(x,u,p,t) \leq$ gu
```
problem.constraints.gl=[];
problem.constraints.gu=[];
```

Load information about the terminal set
```
load termset
```

% Bounds for boundary constraints bl $\leq b(x(t_0), x(t_f), u(t_0), u(t_f), p, t_0, t_f) \leq$ bu
```
problem.constraints.bl=0;
problem.constraints.bu=gm;
```

% Choose the set-points
```
problem.setpoints.states=[0 0];
problem.setpoints.inputs=[0];


Rd=10^(-4);
Qd=[1 0; 0 1];
```

% store the problem parameters used in the functions
```
problem.data.P=P;
problem.data.R=Rd;
problem.data.Q=Qd;


problem.functions={@L,@E,@f,@g,@b};


function stageCost=L(x,xr,u,ur,p,t,data)


    RW=data.R;
    QW=data.Q;


    stagex=((x-xr)*QW).*(x-xr);
    stageu=((u-ur)*RW).*(u-ur);
    stageCost=sum(stagex,2)+sum(stageu,2);


function boundaryCost=E(x0,xf,u0,uf,p,tf,data)


    P=data.P;
    boundaryCost=((xf).'*P*(xf));


function dx = f(x,u,p,t,data)


    x1=x(:,1); x2=x(:,2);u=u(:,1);
```

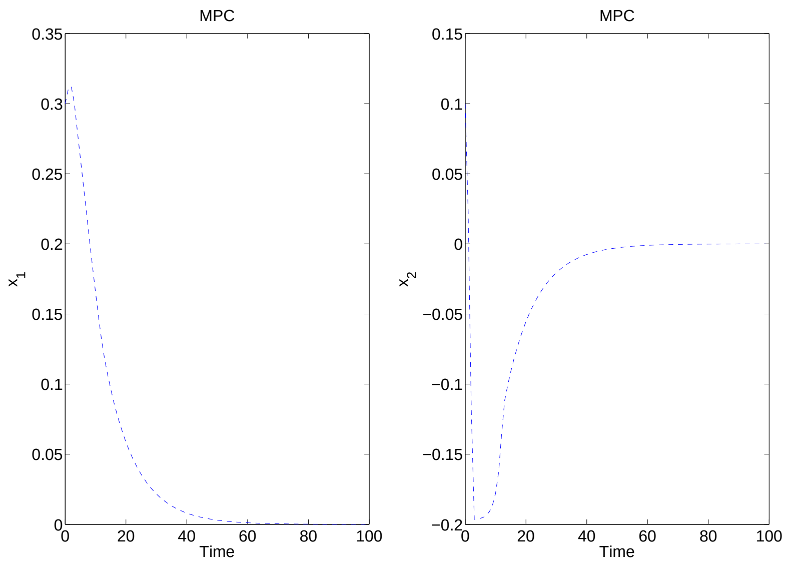

Figure 13: State trajectories for the discrete-time optimization problem

```
    dx(:,1)=x1+0.1*x2;
    dx(:,2)=1.8*x2+0.0787*u;


function c=g(x,u,p,t,data)


    c=[];


function bc=b(x0,xf,u0,uf,p,tf,data)


    P=data.P;
    bc=(xf).'*P*(xf);
```

- The solution method and solver settings are set in `settings_Dis.m` . See the file included in the directory *../ICLOCS-*.*/examples/DiscreteTimeMPC*

- The solution of the optimisation problem is computed running the file `main.m`. inside the directory *../ICLOCS-*.*/examples/DiscreteTimeMPC* The state, control and adjoint variables solution to this problem using the Optimal Control Toolbox are shown in Figs. 13, 14 and 15.

### 4.6 Example 6: Minimum fuel orbit raising problem

Find the control $u(t)$ over $t$ in $[t_0,\ t_f]$ solving the following optimisation problem [3]:

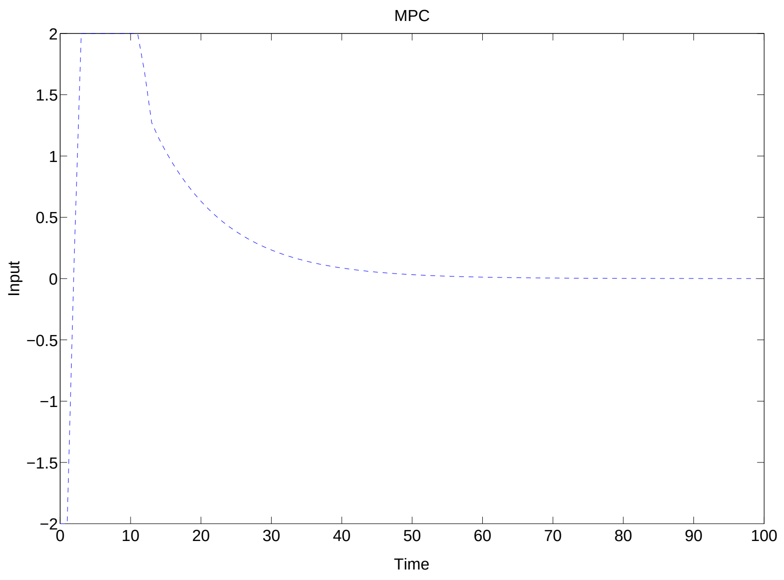

Figure 14: Input trajectory for the discrete-time optimization problem

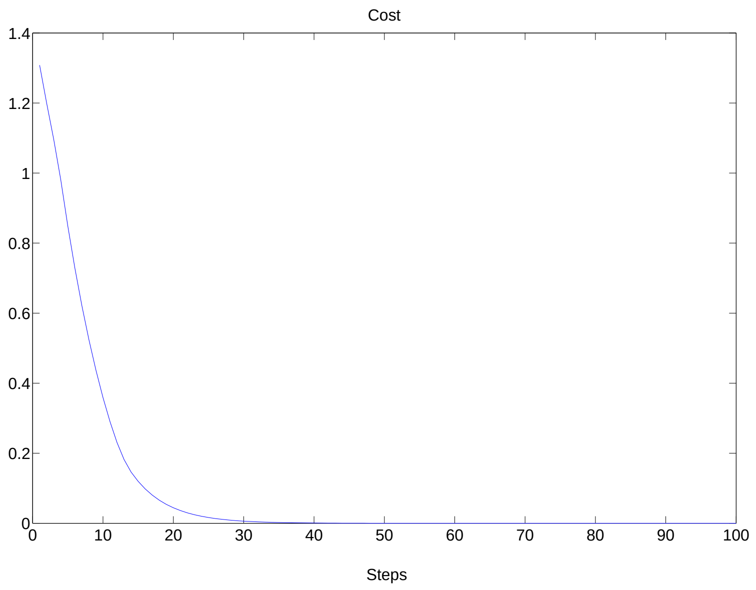

Figure 15: Cost for the discrete-time optimization problem

$$\min_{u(\cdot)} -\int_0^{t_f} x_2(t)dt$$

subject to

$$\begin{cases} \dot{x}_1 = x_2 \\ \dot{x}_2 = \dfrac{x_3^2}{x_1} - \dfrac{1}{x_1^2} + \dfrac{T\sin(u)}{(1-md\ t)} \\ \dot{x}_3 = -\dfrac{x_2 x_3}{x_1} + \dfrac{T\cos(u)}{(1-md\ t)} \end{cases}$$

$$x(0) = [1,\ 0,\ 1]$$
$$x_3(t_f) = \frac{1}{\sqrt{x_1(t_f)}},\ x_2(t_f) = 0$$
$$t_f = 3.32$$

$$0 \le u(t) \le 2\pi$$
$$\begin{bmatrix} 0.5 \\ 0 \\ 0.5 \end{bmatrix} \le \begin{bmatrix} x_1(t) \\ x_2(t) \\ x_3(t) \end{bmatrix} \le \begin{bmatrix} 2 \\ 1 \\ 2 \end{bmatrix} \quad \forall t \in [0,\ t_f]$$

with $T = 0.1405$ and $md = 0.0749$

**Problem setup**

- The Optimal Control Problem is defined in the file `OrbitRaising.m` in the following way:

  % Initial Time. $t_0 < t_f$
  ```
  problem.time.t0=0;
  ```

  % Final time. Let tf_min=tf_max if tf is fixed.
  ```
  problem.time.tf_min=3.32;
  problem.time.tf_max=3.32;
  guess.tf=3.32;
  ```

  % Parameters bounds: pl $\le p \le$ pu
  ```
  problem.parameters.pl=[];
  problem.parameters.pu=[];
  guess.parameters=[];
  ```

  % Initial condition $x_0$ and its bounds
  ```
  problem.states.x0=[1 0 1];
  problem.states.x0l=[1 0 1];
  problem.states.x0u=[1 0 1];
  ```

% State bounds: xl $\leq x(t) \leq$ xu
```
problem.states.xl=[0.5 0 0.5];
problem.states.xu=[2 1 2];
```

% Terminal state bounds: xfl $\leq x(t) \leq$ xfu
```
problem.states.xfl=[0 0 0];
problem.states.xfu=[2 1 2];
```

% Guess the state trajectories with $[x(t_0),\ x(t_f)]$
```
guess.states(:,1)=[1 1.4];
guess.states(:,2)=[0 0.8];
guess.states(:,3)=[1 0.05];
```

Number of control actions $N$. If $N$ is equal to the number of integration steps, problem.inputs.N can be set to 0
```
problem.inputs.N=0;
```

% Input bounds: ul $\leq u(t) \leq$ uu
```
problem.inputs.ul=[0];
problem.inputs.uu=[2*pi];
```

% Guess the input sequences with $[u(t_0),\ u(t_f)]$
```
guess.inputs(:,1)=[0 2*pi];
```

% Choose the set-points if required
```
problem.setpoints.states=[];
problem.setpoints.inputs=[];
```

% Bounds for path constraint function gl $\leq g(x,u,p,t) \leq$ gu
```
problem.constraints.gl=[];
problem.constraints.gu=[];
```

% Bounds for boundary constraints bl $\leq b(x(t_0),x(t_f),u(t_0),u(t_f),p,t_0,t_f) \leq$ bu
```
problem.constraints.bl=[0 0];
problem.constraints.bu=[0 0];
```

% Store the problem parameters used in the functions
```
problem.data.T1=0.1405;
problem.data.md=0.0749;
```

```
problem.functions={@L,@E,@f,@g,@b};
```

```
function stageCost=L(x,xr,u,ur,p,t,data)
```

```
    stageCost = -x(:,2);

function boundaryCost=E(x0,xf,u0,uf,p,tf,data)

    boundaryCost=0;

function dx = f(x,u,p,t,data)

    r=x(:,1);q=x(:,2);v=x(:,3);phi=u(:,1);

    T1=data.T1;
    md=data.md;
    dx=[q,...
    (v.*v)./r-r.^(-2)+(T1*sin(phi)./(1-md*t)),...
    (-q.*v)./r+(T1*cos(phi)./(1-md*t))];

function c=g(x,u,p,t,data)

    c=[];

function bc=b(x0,xf,u0,uf,p,tf,data)

    bc=[xf(3)-sqrt(1/xf(1));xf(2)];
```

- The solution method and solver settings are set in `settings.m` . See the file included in the directory *../ICLOCS-*.*/examples/OrbitRaising*

- The files `gradCost.m`, `jacConst.m` and `hessianLagrangian.m` for this example are supplied. See inside the directory *../ICLOCS-*.*/examples/OrbitRaising*.

  - `gradCost.m`:

```
    function [dL,dE]=gradCost(L,X,Xr,U,Ur,P,t,E,x0,xf,u0,uf,p,tf,data)

    Lt=ones(size(t));
```

% `dL` - Gradient of the stage cost $L(\cdot)$ wrt. $t$, $p$, $x$, $u$

```
    dL.flag=1;
    dL.dp=[];
    dL.dt=[];
    dL.dx=[0*Lt, -Lt, 0*Lt];
```

```
dL.du=0*Lt;
```

% `dE` - Gradient of $E(\cdot)$ with respect to $t_f,\ p,\ x_0,\ u_0,\ u_f,\ x_f$

```
dE.flag=1;
dE.dtf=[];
dE.dp=[];
dE.dx0=[];
dE.du0=[];
dE.dxf=[];
dE.duf=[];
```

The gradient of the stage cost and the boundary cost are supplied and so `dL.flag=1` and `dE.flag=1`. The final time $t_f$ and $p$ are not decision variables and then the derivatives `dL.dp`, `dL.dt`, `dE.dtf`, and `dE.dp` are set to be empty matrices. The derivative of the boundary cost does not depend on any variable and then all variables containing the derivatives are set to be empty matrices.

- `jacConst.m:`

```
function [df,dg,db]=jacConst(f,g,X,U,P,t,b,x0,xf,u0,uf,p,tf,t0,data)

Lt=ones(size(t));
coef=data.data.T1./(1-data.data.md*t);

df.flag=1;
```

`df.dp{1}=[];` $\partial f(x,u,p,t)/\partial p$

`df.dt{1}=[];` $\partial f(x,u,p,t)/\partial t$

$\partial f(x,u,p,t)/\partial x_1$

```
df.dx{1}=[0*Lt, -(X(:,3).^2)./(X(:,1).  ^2)+2./(X(:,1).^3),...
        X(:,3).*X(:,2)./(X(:,1).^2) ];
```

$\partial f(x,u,p,t)/\partial x_2$

```
df.dx{2}=[1*Lt, 0*Lt, -X(:,3)./X(:,1)];
```

$\partial f(x,u,p,t)/\partial x_3$

```
df.dx{3}=[0*Lt, 2*X(:,3)./X(:,1), -X(:,2)./X(:,1)];
```

$\partial f(x,u,p,t)/\partial u$

```
df.du{1}=[0*Lt, coef.*cos(U(:,1)), -coef.*sin(U(:,1))];

dg.flag=0;

db.flag=1;
db.dtf=[];
db.dp=[];
db.dx0=[];
db.du0=[];
db.duf=[];
% Derivatives with respect to [x1(tf), x2(tf), x3(tf)]
db.dxf=[1./sqrt(xf(1))./xf(1)/2, 0, 1; 0, 1, 0];
```

The path constraints are not present and then their derivatives have not be supplied (`dg.flag=0`). Instead the derivatives of the boundary constraints are supplied in the structured variable `db` as shown in the illustrated example. The boundary constraints $b(t_f,\ p,\ x_0,\ u_0,\ u_f,\ x_f)$ depends only on the final state and then the variables containing the other derivatives are set to be empty matrices. Each row of `db.dxf` corresponds to the evaluated derivative of a terminal constraint with respect to $[x_1(t_f),\ x_2(t_f),\ x_3(t_f)]$.

- `hessianLagrangian.m:`

```
function [HL,HE,Hf,Hg,Hb]=hessianLagrangian(X,U,P,t,E,x0,xf,u0,uf,p,tf,data)
```

The Hessian of the different parts of the Lagrangian (`HL, HE, Hf, Hg, Hb`) must be supplied in cell arrays as follows:

```
[nt,np,n,m,ng,nb]=deal(data.sizes{1:6});
nfz=nt+np+n+m;

Hf=[];

Lz=zeros(nfz,nfz);
HL=num2cell(Lz);

HE=[];
Hg=[];

Bz=zeros(n,n);
Hb=num2cell(Bz);
Hb{1,1}=[-3./sqrt(xf(1))./(xf(1).^2)/4; 0];
```

The Hessian of the dynamical system and the path constraints have not been specified and then the respective variables are set to be empty matrices (`Hf=[];` and `Hg=[];`). The Hessian of the stage cost is specified in a two dimensional cell array of dimension given by the size of $y = [x,\ u]$ while the Hessian of the boundary constraints is given with respect to $y = [x_f]$. Notice that `dE.flag=1` in `gradCost.m:` and $E(\cdot)$ does not depend on any variable so the Hessian of $E(\cdot)$ is not evaluated and it has

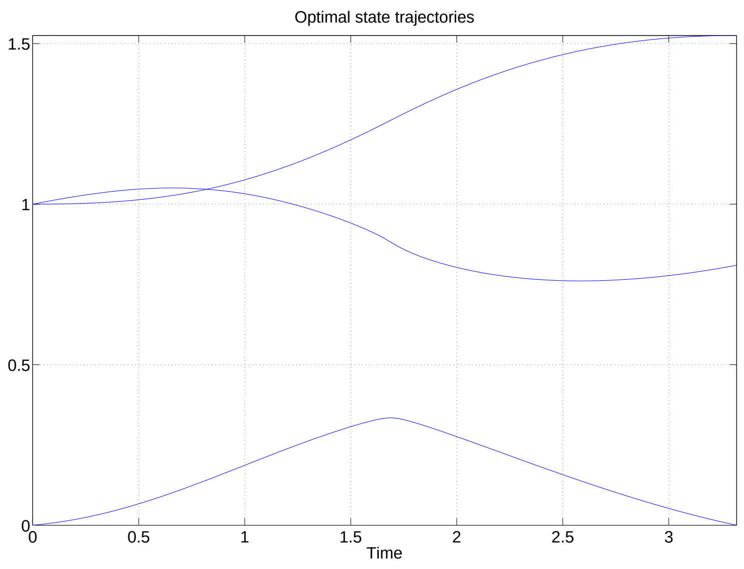

Figure 16: State trajectories for minimum fuel orbit raising problem

not be specified. If `dE.flag=0`, whenever `HE` is not set to be empty, the entries for the Hessian with respect to $[x(t_0), u(t_0), u(t_f), x(t_f)]$ are specified in the following way (`nt=0`, `np=0`):

```
nE=nt+np+2*n+2*m;
Ez=zeros(nE,nE);
HE=num2cell(Ez);
```

If db.flag=1 it is necessary to check on which variables the boundary constraints $b(t_f, p, x(t_0), u(t_0), x(t_f), u(t_f))$ depends on. In this example it depends only on $x(t_f)$ and then it is necessary to specify only the Hessian with respect to $x(t_f)$. If db.flag=0, whenever `Hb` is not set to be empty, it needs to specify at least the entries for the Hessian with respect to $[x(t_0), u(t_0), u(t_f), x(t_f)]$ as follow (here $t_f$ and $p$ are not decision variables and they must not be considered):

```
Bz=zeros(nE,nE);
Hb=num2cell(Bz);
Hb{1,6}=[-3./sqrt(xf(1))./(xf(1).^2)/4; 0];
```

- The solution of the optimisation problem is computed running the file `main.m` inside the directory *../ICLOCS-*.*/examples/OrbitRaising* The state, control and adjoint variables solution to this problem using the Optimal Control Toolbox are shown in Figs. 16, 17 and 18.

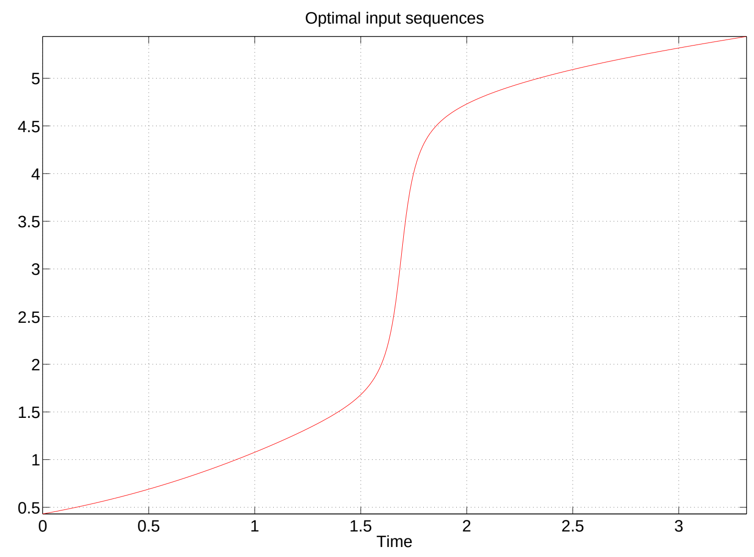

Figure 17: Input trajectory for minimum fuel orbit raising problem

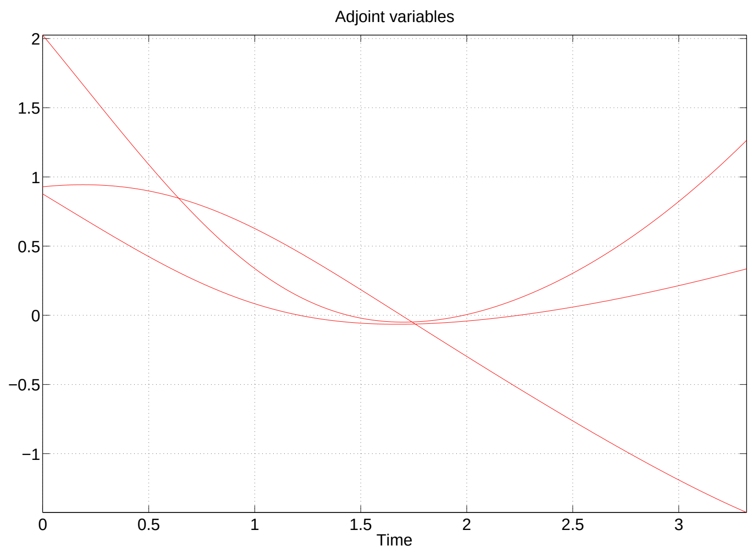

Figure 18: Adjoint variables for minimum fuel orbit raising problem

## 4.7 Example 7: Path-constrained optimal control problem

The following optimisation problem considers a continuous state constraint [7]

$$\min_{u(\cdot)} \int_0^1 x_1^2(t) + x_2^2(t) + 0.005u^2(t)dt$$

subject to

$$\begin{cases} \dot{x}_1 = x_2 \\ \dot{x}_2 = -x_2 + u \end{cases}$$

$$x(0) = [0,\ -1]$$

$$-20 \leq u(t) \leq 20$$
$$\begin{bmatrix} -10 \\ -10 \end{bmatrix} \leq \begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} \leq \begin{bmatrix} 10 \\ 10 \end{bmatrix} \ \forall t \in [0,\ 1]$$
$$8(t-0.5)^2 - 0.5 - x_2(t) \geq 0$$

### Problem setup

- The Optimal Control Problem is defined in the file `PathConstraint.m` in the following way:

  % Initial time $t_0 < t_f$
  ```
  problem.time.t0=0;
  ```

  % Final time. Let tf_min=tf_max if tf is fixed.
  ```
  problem.time.tf_min=1;
  problem.time.tf_max=1;
  guess.tf=1;
  ```

  % Parameters bounds pl $\leq p \leq$ pu
  ```
  problem.parameters.pl=[];
  problem.parameters.pu=[];
  guess.parameters=[];
  ```

  % Initial condition $x(t_0)$ and its bounds
  ```
  problem.states.x0=[0 -1];
  problem.states.x0l=[0 -1];
  problem.states.x0u=[0 -1];
  ```

  % State bounds xl $\leq x(t) \leq$ xu
  ```
  problem.states.xl=[-10 -10];
  problem.states.xu=[10 10];
  ```

% Terminal state bounds. xfl $\leq x(t_f) \leq$ xfu
```
problem.states.xfl=[-10 -10];
problem.states.xfu=[10 10];
```

% Guess the state trajectories with $[x(t_0),\ x(t_f)]$
```
guess.states(:,1)=[0 0];
guess.states(:,2)=[-1 -1];
```

% Number of control actions N. Set problem.inputs.N=0 if N is equal to the number of integration steps.
```
problem.inputs.N=0;
```

% Input bounds ul $\leq u(t) \leq$ uu
```
problem.inputs.ul=[-20];
problem.inputs.uu=[20];
```

% Guess the input sequences with $[u(t_0),\ u(t_f)]$
```
guess.inputs(:,1)=[0 0];
```

% Choose the set-points if required
```
problem.setpoints.states=[];
problem.setpoints.inputs=[];
```

% Bounds for path constraint function gl $\leq g(x, u, p, t) \leq$ gu
```
problem.constraints.gl=[0, -10];
problem.constraints.gu=[inf, 10];
```

% Bounds for boundary constraints
```
problem.constraints.bl=[];
problem.constraints.bu=[];
```

% store the necessary problem parameters used in the functions
```
problem.data=[];


problem.functions={@L,@E,@f,@g,@b};


function stageCost=L(x,xr,u,ur,p,t,data)

    x1 = x(:,1);x2 = x(:,2);u1 = u(:,1);

    stageCost = x1.^2+x2.^2+0.005*u1.^2;
```

```
function boundaryCost=E(x0,xf,u0,uf,p,tf,data)

    boundaryCost=0;

function dx = f(x,u,p,t,data)

    x1 = x(:,1);x2 = x(:,2);u1 = u(:,1);

    dx(:,1) = x2;
    dx(:,2) = -x2+u1;

function c=g(x,u,p,t,data)

    x2 = x(:,2);

    c=[8*(t-0.5).^2-0.5-x2, x(:,1)];

function bc=b(x0,xf,u0,uf,p,tf,data)

    bc=[];
```

- The solution method and solver settings are set in `settings.m` . See the file included in the directory *../ICLOCS-*.*/examples/PathConstraint*

- The files `gradCost.m`, `jacConst.m` and `hessianLagrangian.m` for this example are supplied. See inside the directory *../ICLOCS-*.*/examples/PathConstraint.*

  - `gradCost.m`:

```
function [dL,dE]=gradCost(L,X,Xr,U,Ur,P,t,E,x0,xf,u0,uf,p,tf,data)
```

% `dL` - Gradient of the stage cost $L(\cdot)$ wrt. $t,\ p,\ x,\ u$

```
dL.flag=1;
dL.dp=[];
dL.dt=[];
dL.dx=[2*X(:,1), 2*X(:,2)];
dL.du=2*0.005*U(:,1);
```

% `dE` - Gradient of $E(\cdot)$ with respect to $t_f,\ p,\ x_0,\ u_0,\ u_f,\ x_f$

```
dE.flag=1;
dE.dtf=[];
dE.dp=[];
dE.dx0=[];
dE.du0=[];
dE.dxf=[];
dE.duf=[];
```

- jacConst.m:

```
function [df,dg,db]=jacConst(f,g,X,U,P,t,b,x0,xf,u0,uf,p,tf,t0,data)

Lt=ones(size(t));

df.flag=1;
df.dp{1}=[];
df.dt{1}=[];
df.dx{1}=[0*Lt 0*Lt];
df.dx{2}=[1*Lt , -1*Lt];
df.du{1}=[0*Lt, 1*Lt];

dg.flag=1;
dg.dp{1}=[];
dg.dt{1}=[16*(t-0.5), 0*t];
dg.dx{1}=[0*Lt, 1*Lt];
dg.dx{2}=[-1*Lt, 0*Lt];
dg.du{1}=[0*Lt, 0*Lt];

db.flag=1;
db.dtf=[];
db.dp=[];
db.dx0=[];
db.du0=[];
db.dxf=[];
db.duf=[];
```

The path constraints are present and their derivatives have been supplied (`dg.flag=1`) . The derivatives of $g(x,u,p,t) = [g_1(x,u,p,t),\dots,g_{n_g}(x,u,p,t)]$ are supplied in the structured variable `dg` as shown in the illustrated example. The derivative of $g(x,u,p,t)$ with respect to $x$ are stored in dg.dx which is a cell array where each entry corresponds to the derivative with respect to a state variable. For instance the derivative of $g(x,u,p,t)$ with respect to $x_i$ is stored in `dg.dx{i}`. `dg.dx{i}` is a matrix with $n_g$ columns and a number of rows given by the different evaluation times. Notice that the derivative of the path constraints with respect to the time has been specified but it was not necessary since the final time $t_f$ is not a decision variable.

- hessianLagrangian.m:

```
function [HL,HE,Hf,Hg,Hb]=hessianLagrangian(X,U,P,t,E,x0,xf,u0,uf,p,tf,data)
```

The Hessian of the different parts of the Lagrangian (`HL, HE, Hf, Hg, Hb`) must be supplied in cell arrays as follows:

```
[nt,np,n,m]=deal(data.sizes{1:4});
nfz=nt+np+n+m;


Fz=zeros(nfz,nfz);
Hf=num2cell(Fz);


Hz=zeros(nfz, nfz);
HL=num2cell(Hz);


HL{1,1}=2;
HL{2,2}=2;
HL{3,3}=2*0.005;


HE=[];


Gz=zeros(nfz,nfz);
Hg=num2cell(Gz);


Hb=[];
```

- The solution of the optimisation problem is computed running the file `main.m` inside the directory *../ICLOCS-*.*/examples/PathConstraint* The state, control and adjoint variables solution to this problem using the Optimal Control Toolbox are shown in Figs. 19, 20 and 21.

## 4.8 Example 8: Continuously-stirred tank reactor

The following optimisation problem is based on the model of a continuously-stirred tank reactor proposed in [6] and modified in [8]

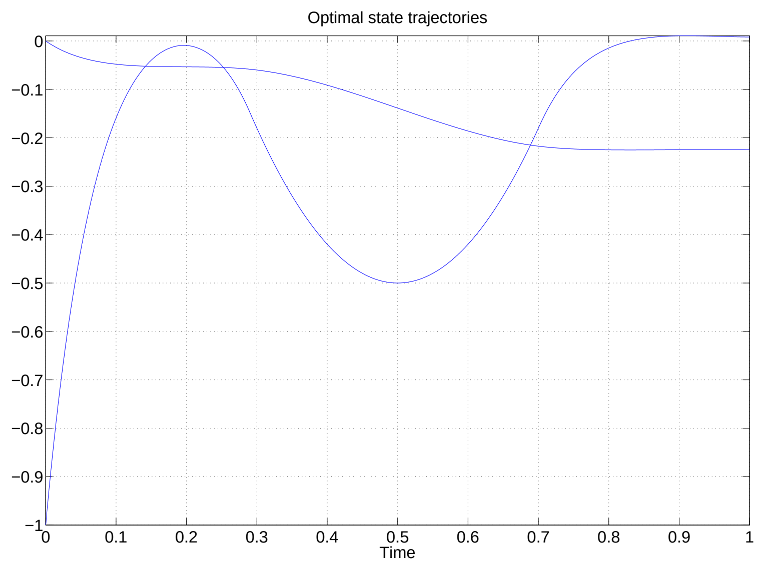

Figure 19: State trajectories for the path-constrained optimal control problem

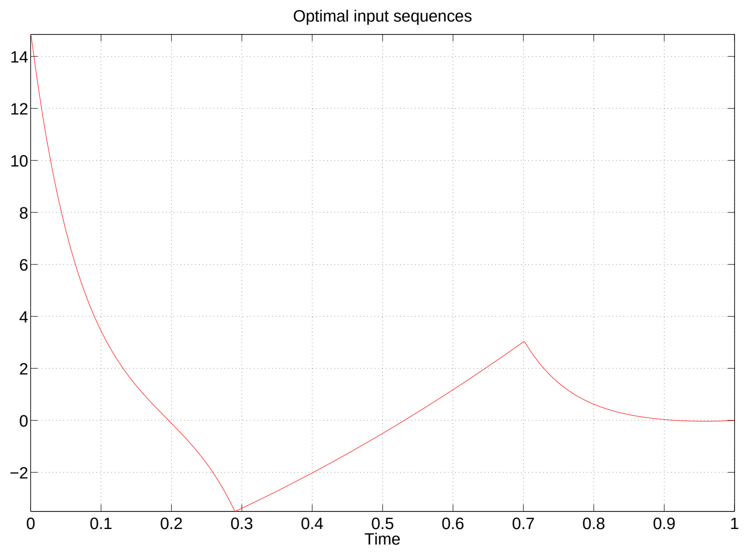

Figure 20: Input trajectory for the path-constrained optimal control problem

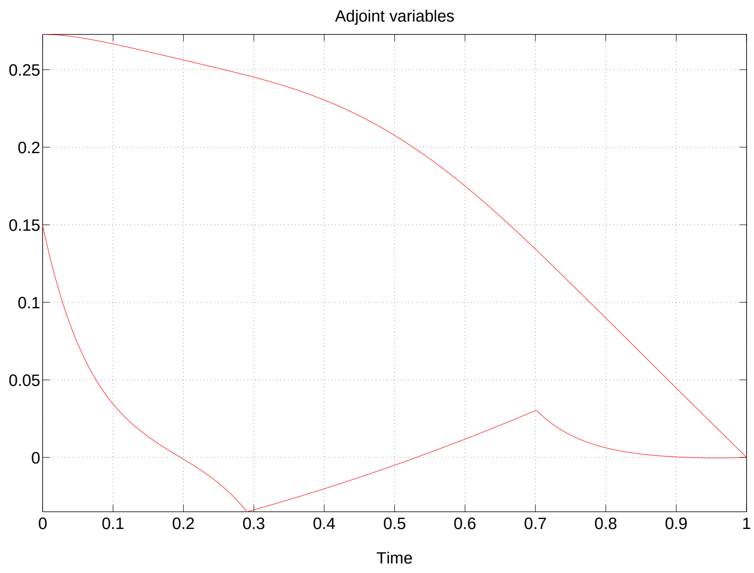

Figure 21: Adjoint variables for the path-constrained optimal control problem

$$\min_{u(\cdot),t_f} \int_0^{t_f} (x_1(t) - x_1(tf))^2 + (x_2(t) - x_2(tf))^2 + (u(t) - u(t_f))^2 dt$$

subject to

$$\begin{cases} \dot{x}_1 = \dfrac{1 - x_1}{q} - kx_1 e^{(-\frac{En}{x_2})} \\ \dot{x}_2 = \dfrac{Tf - x_2}{q} + kx_1 e^{(-\frac{En}{x_2})} - au(x_2 - T_c) \end{cases}$$

$x(0) = [0.98,\ 0.39]$
$x_1(tf) = 0.26,\ x_2(tf) = 0.65,\ u(t_f) = 0.76$

$0 \leq u(t) \leq 2$
$\begin{bmatrix} 0 \\ 0 \end{bmatrix} \leq \begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} \leq \begin{bmatrix} 1 \\ 1 \end{bmatrix} \ \forall t \in [0,\ t_f]$
$10 \leq t_f$

where $a = 0.000195 * 600$, $q = 20$, $En = 5$, $k = 300$, $T_c = 0.38158$ and $Tf = 0.3947$. The variable $x_1$ is the product concentration, $x_2$ is the temperature and $u$ is the coolant flow rate. In the described optimisation problem, the reactor undergoes a slow set-point change from the stable steady-state $x(0) = [0.98,\ 0.39]$ to the unstable steady state $x(t_f) = [0.26,\ 0.65]$.

**Problem setup**

- The Optimal Control Problem is defined in the file `RayHicksCSTR.m` in the following way:

  %Initial time $t_0 < t_f$
  ```
  problem.time.t0=0;
  ```

  % Final time. Let tf_min=tf_max if $t_f$ is fixed.
  ```
  problem.time.tf_min=10;
  problem.time.tf_max=inf;
  guess.tf=120;
  ```

  % Parameters bounds pl $\leq p \leq$ pu
  ```
  problem.parameters.pl=[];
  problem.parameters.pu=[];
  guess.parameters=[];
  ```

  % Initial condition $x(t_0)$ and its bounds
  ```
  problem.states.x0=[0.9831 0.3918];
  problem.states.x0l=problem.states.x0;
  problem.states.x0u=problem.states.x0l;
  ```

  % State bounds xl $\leq x(t) \leq$ xu
  ```
  problem.states.xl=[0 0];
  problem.states.xu=[1 1];
  ```

  % Terminal state bounds xfl $\leq x(t_f) \leq$ xfu
  ```
  problem.states.xfl=[0.2632 0.6519];
  problem.states.xfu=[0.2632 0.6519];
  ```

  % Guess the state trajectories with $[x(t_0),\ x(t_f)]$
  ```
  guess.states(:,1)=[0.9831 0.2632];
  guess.states(:,2)=[0.3918 0.6519];
  ```

  % Number of control actions N. Set problem.inputs.N=0 if N is equal to the number of integration steps.
  ```
  problem.inputs.N=40;
  ```

  % Input bounds ul $\leq u(t) \leq$ uu
  ```
  problem.inputs.ul=[0];
  problem.inputs.uu=[2];
  ```

  % Guess the input sequences with $[u(t_0),\ u(t_f)]$
  ```
  guess.inputs(:,1)=[0.0 455/600];
  ```

```
% Bounds for path constraint function gl ≤ g(x,u,p,t) ≤ gu
problem.constraints.gl=[];
problem.constraints.gu=[];

% Bounds for boundary constraints bl ≤ b(x(t0),x(tf),u(t0),u(tf),p,t0,tf) ≤ bu
problem.constraints.bl=[];
problem.constraints.bu=[];

% Choose the set-points
problem.setpoints.states=[0.2632 0.6519];
problem.setpoints.inputs=[455/600];

% store the necessary problem parameters used in the functions
problem.data=[];

problem.functions={@L,@E,@f,@g,@b};

function stageCost=L(x,xr,u,ur,p,t,data)

    c=x(:,1);T=x(:,2);u=u(:,1);
    cr=xr(:,1);Tr=xr(:,2);

    stageCost = 0.5*((c-cr).*(c-cr)+(T-Tr).*(T-Tr))+0.5*(u-ur).*(u-ur);

function boundaryCost=E(x0,xf,u0,uf,p,tf,data)

    boundaryCost=0;

function dx = f(x,u,p,t,data)

    % Biegler's Model coefficient a=0.000195*600;q=20;En=5;k=300;Tc=0.38158;Tf=0.3947;

    c=x(:,1);T=x(:,2);u=u(:,1);

    dx(:,1)=(1-c)/q-k*c.*exp(-En./T);
    dx(:,2)=(Tf-T)/q+k*c.*exp(-En./T)-a*u.*(T-Tc);

function c=g(x,u,p,t,data) function bc=b(x0,xf,u0,uf,p,tf,data)
```

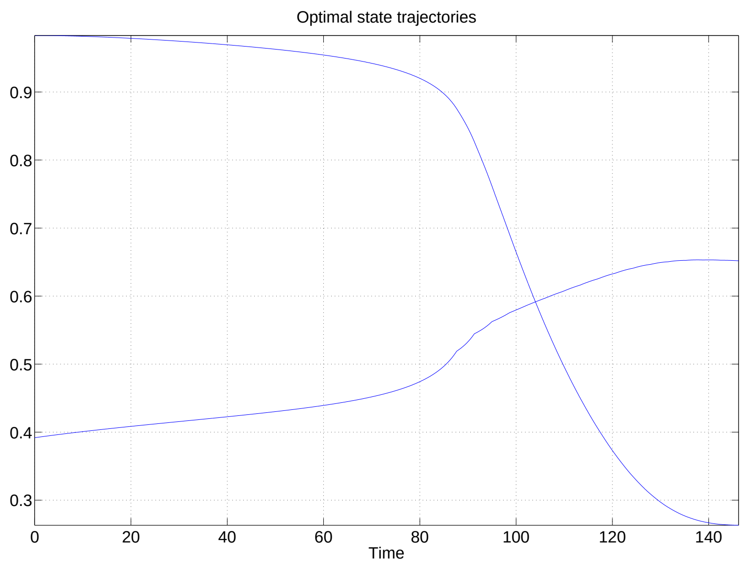

Figure 22: State trajectories for the continuous-stirred tank reactor

```
    c=[];

function bc=b(x0,xf,u0,uf,p,tf,data)

    bc=[];
```

- The solution method and solver settings are set in `settings.m` . See the file included in the directory *../ICLOCS-*.*/examples/RayHicksCSTR*

- The solution of the optimisation problem is computed running the file `main.m` inside the directory *../ICLOCS-*.*/examples/RayHicksCSTR* The state, control and adjoint variables solution to this problem using the Optimal Control Toolbox are shown in Figs. 22, 23 and 24.

## 4.9 Example 9: Continuous-time nonlinear Model Predictive Control example

At any time instant $t_k$ for $k = 0, 1, 2, \dots$ measure the state $x(t_k)$, solve the following optimal control problem

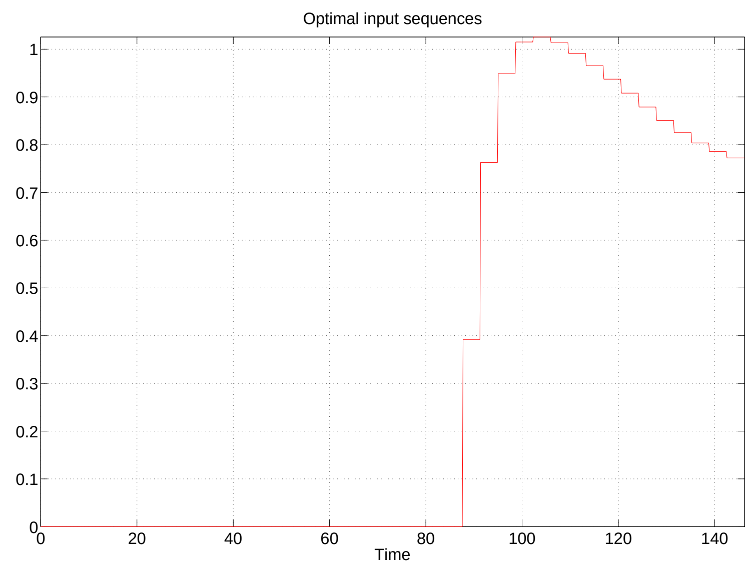

Figure 23: Input trajectory for the continuous-stirred tank reactor

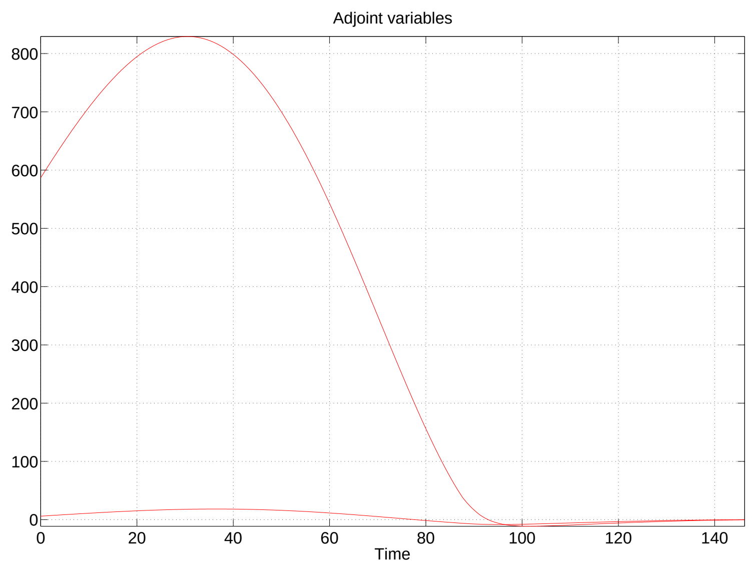

Figure 24: Adjoint variables for the continuous-stirred tank reactor

$$\min_{u(\cdot)} \int_0^{10} x_1(t)^2 + x_2(t)^2 + u(t)^2 dt$$

subject to

$$\begin{cases} \dot{x}_1 = x_2 \\ \dot{x}_2 = \sin(x_1) + u \end{cases}$$

$$x(0) = x(t_k)$$

$$-10 \leq u(t) \leq 10$$
$$\begin{bmatrix} -2 \\ -2 \end{bmatrix} \leq \begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} \leq \begin{bmatrix} 2 \\ 2 \end{bmatrix} \; \forall t \in [0,\ 10]$$

and apply the first control action u(0) for the time window $[t_k, t_{k+1}]$. The measures are collected from the plant described by the following ODE

$$\dot{x}_1 = 1.2x_2 + 0.1sin(t)$$
$$\dot{x}_2 = 0.2sin(x_1) + u$$

with $x(0) = [1,\ 1]$.

**Problem setup**

- The Optimal Control Problem is defined in the file `testProblem.m` in the following way:

  % Initial time $t_0 < t_f$
  ```
  problem.time.t0=0;
  ```

  % Final time. Let tf_min=tf_max if $t_f$ is fixed.
  ```
  problem.time.tf_min=10;
  problem.time.tf_max=10;
  guess.tf=10;
  ```

  % Parameters bounds pl $\leq p \leq$ pu
  ```
  problem.parameters.pl=[];
  problem.parameters.pu=[];
  guess.parameters=[];
  ```

  % Initial condition $x(0)$ and its bounds
  ```
  problem.states.x0=[1 1];
  problem.states.x0l=problem.states.x0;
  problem.states.x0u=problem.states.x0l;
  ```

  % State bounds xl $\leq x(t) \leq$ xu
  ```
  problem.states.xl=[-2 -2];
  ```

```
problem.states.xu=[2 2];
```

% Terminal state bounds xfl$\leq x(t_f) \leq$ xfu
```
problem.states.xfl=[0 0];
problem.states.xfu=[0 0];
```

% Guess the state trajectories with $[x(0),\ x(t_f)]$
```
guess.states(:,1)=[1 0];
guess.states(:,2)=[1 0];
```

% Number of control actions N
```
problem.inputs.N=10;
```

% Input bounds ul $\leq u(t) \leq$ uu
```
problem.inputs.ul=[-10];
problem.inputs.uu=[10];
```

% Guess the input sequences with $[u(0),\ u(t_f)]$
```
guess.inputs(:,1)=[-2 0];
```

% Choose the set-points if required
```
problem.setpoints.states=[];
problem.setpoints.inputs=[];
```

% Bounds for path constraint function gl $\leq g(x,u,p,t) \leq$ gu
```
problem.constraints.gl=[];
problem.constraints.gu=[];
```

% Bounds for boundary constraints bl $\leq b(x(t_0),x(t_f),u(t_0),u(t_f),p,t_0,t_f) \leq$ bu
```
problem.constraints.bl=[];
problem.constraints.bu=[];
```

% Problem data
```
problem.data=[];


problem.functions={@L,@E,@f,@g,@b};


function stageCost=L(x,xr,u,ur,p,t,data)


    x1=x(:,1); x2=x(:,2); u1=u(:,1);
    stageCost = x1.*x1+x2.*x2+u1.*u1;

function boundaryCost=E(x0,xf,u0,uf,p,tf,data)

    boundaryCost=0;

function dx = f(x,u,p,t,data)

    x1=x(:,1); x2=x(:,2); u=u(:,1);

    dx(:,1)=x2;
    dx(:,2)=sin(x1)+u;

function c=g(x,u,p,t,data)

    c=[];

function bc=b(x0,xf,u0,uf,p,tf,data)

    bc=[];
```

- The solution method and solver settings are set in `settings.m` . See the file included in the directory *../ICLOCS-*.*/examples/SimpleMPC*

- The solution of the optimisation problem is computed running the file `mainMPC.m` inside the directory *../ICLOCS-*.*/examples/SimpleMPC*
  The following lines are executed:

```
clear all
format compact

[problem,guess]= testProblem; % Fetch the problem definition
options= settings; % Get options and solver settings
plant=@testPlant; % Get function handle of plant model

[infoNLP,data]=transcribeOCP(problem,guess,options); % Format for NLP solver
[nt,np,n,m,ng,nb,M,N,ns]=deal(data.sizes{:});
time=[];states=[];inputs=[];

P=20; % Number of MPC iterations
```

```
% — Begin MPC loop —
for i=1:P
disp('Compute Control Action');disp(i);

solution = solveNLP(infoNLP,data); % Solve the NLP

tc=solution.tf/N; % Control horizon

% Apply control
disp('Apply Control')
[x0,tv,xv,uv]=applyControl(tc,solution,plant,data,i);

% Store results
time=[time;tv];
states=[states;xv];
inputs=[inputs;uv];

% Update initial condition
infoNLP.zl(nt+np+1:nt+np+n)=x0;
infoNLP.zu(nt+np+1:nt+np+n)=x0;
data.x0t=x0.';

% Update initial guess
infoNLP.z0=solution.z;

end
% — End MPC loop —

% Plot the solutions
figure(1)
hold on
plot(time,states)
title('States vs.  time');
xlabel('Time')

figure(2)
hold on
plot(time,inputs)
title('Inputs vs.  Time');
xlabel('Time')

figure(3)
hold on
plot(states(:,1),states(:,2))
```

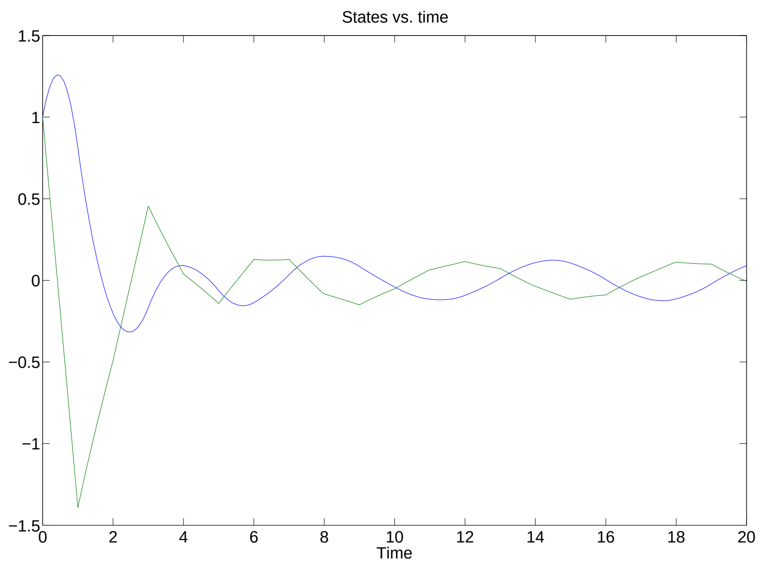

Figure 25: State trajectories for the continuous-time nonlinear Model Predictive Control example

```
title('Optimal state trajectory');
xlabel('x_1')
ylabel('x_2')
```

Notice that `mainMPC.m` calls the following two additional files:

- `testPlant.m`: It returns the ODE right hand side of the model describing the plant.
- `applyControl.m`: It applies the optimal control stored in the variable `solution` to the system described in the file `testPlant.m`.

The state and control variables solution to this problem using the Optimal Control Toolbox are shown in Figs. 25, 26 and 27.

## 4.10 Example 10: Singular Arc problem

Find the control input $u \in \mathbb{R}$ over $t$ in $[0,\ t_f]$ solving the following optimisation problem [7], [5] (p.57).

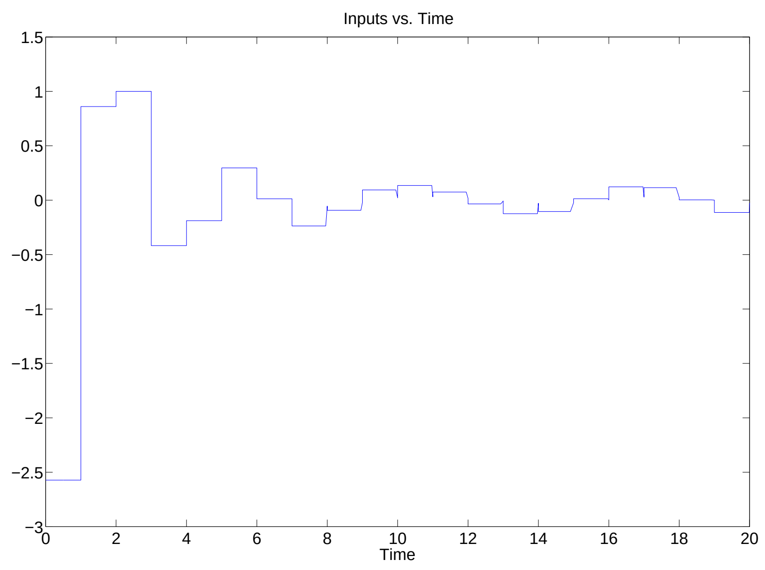

Figure 26: Input trajectory for the Continuous time nonlinear Model Predictive Control example

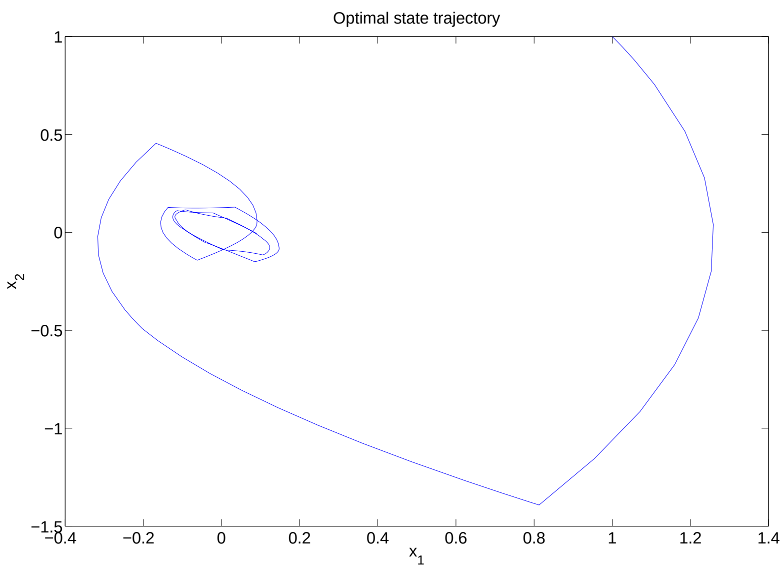

Figure 27: Phase plane for the continuous-time nonlinear Model Predictive Control example

$$\min_{u(\cdot),t_f} \; t_f$$

subject to

$$\begin{cases} \dot{x}_1 = u \\ \dot{x}_2 = \cos(x_1) \\ \dot{x}_3 = \sin(x_1) \end{cases}$$

$$x(0) = [\tfrac{\pi}{2},\ 4,\ 0]$$
$$x_3(tf) = 0,\ x_2(tf) = 0$$

$$-2 \leq u(t) \leq 2$$
$$\begin{bmatrix} -10 \\ -10 \\ -10 \end{bmatrix} \leq \begin{bmatrix} x_1(t) \\ x_2(t) \\ x_3(t) \end{bmatrix} \leq \begin{bmatrix} 10 \\ 10 \\ 10 \end{bmatrix} \quad \forall t \in [0,\ t_f]$$
$$1 \leq t_f \leq 100$$

**Problem setup**

- The Optimal Control Problem is defined in the file `SingularArc.m` in the following way:

% Initial time $t_0 < t_f$
```
problem.time.t0=0;
```

% Final time. Let tf_min=tf_max if $t_f$ is fixed.
```
problem.time.tf_min=1;
problem.time.tf_max=100;
guess.tf=4;
```

% Parameters bounds. pl $\leq p \leq$ pu
```
problem.parameters.pl=[];
problem.parameters.pu=[];
guess.parameters=[];
```

% Initial condition $x(t_0)$ and its bounds
```
problem.states.x0=[pi/2 4 0];
problem.states.x0l=[pi/2 4 0];
problem.states.x0u=[pi/2 4 0];
```

% State bounds xl $\leq x(t) \leq$ xu
```
problem.states.xl=[-10 -10 -10];
problem.states.xu=[10 10 10];
```

```
% Terminal state bounds xfl ≤ x(t) ≤ xfu
problem.states.xfl=[-10 0 0];
problem.states.xfu=[10 0 0];

% Guess the state trajectories with [x(t_0), x(t_f)]
guess.states(:,1)=[pi/2 pi/2];
guess.states(:,2)=[4 0];
guess.states(:,3)=[0 0];

% Number of control actions N.
% Set problem.inputs.N=0 if N is equal to the number of integration steps.
problem.inputs.N=0;

% Input bounds ul ≤ u(t) ≤ uu
problem.inputs.ul=-2;
problem.inputs.uu=2;

% Guess the input sequences with [u(t_0), u(t_f)]
guess.inputs(:,1)=[0 0];

% Choose the set-points if required
problem.setpoints.states=[];
problem.setpoints.inputs=[];

% Bounds for path constraint function gl ≤ g(x,u,p,t) ≤ gu
problem.constraints.gl=[];
problem.constraints.gu=[];

% Bounds for boundary constraints bl ≤ b(x(t_0),x(t_f),u(t_0),u(t_f),p,t_0,t_f) ≤ bu
problem.constraints.bl=[];
problem.constraints.bu=[];

% Problem data
problem.data=[];

problem.functions={@L,@E,@f,@g,@b};

function stageCost=L(x,xr,u,ur,p,t,data)

    stageCost = 0*t;

function boundaryCost=E(x0,xf,u0,uf,p,tf,data)
```

```
    boundaryCost=tf;

function dx = f(x,u,p,t,data)

    x1 = x(:,1); u1 = u(:,1);

    dx(:,1) = u1;
    dx(:,2) = cos(x1);
    dx(:,3) = sin(x1);

function c=g(x,u,p,t,data)

    c=[];

function bc=b(x0,xf,u0,uf,p,tf,data)

    bc=[];
```

- The solution method and solver settings are set in `settings.m` . See the file included in the directory *../ICLOCS-*.*/examples/SingularArc*

- The solution of the optimisation problem is computed running the file `main.m` inside the directory *../ICLOCS-*.*/examples/SingularArc* The state, control and adjoint variables solution to this problem using the Optimal Control Toolbox are shown in Figs. 28, 29 and 30.

## 4.11 Example 11: Speyer's problem

The Speyer's problem consists in the following periodic optimal control problem of a sailboat [11]

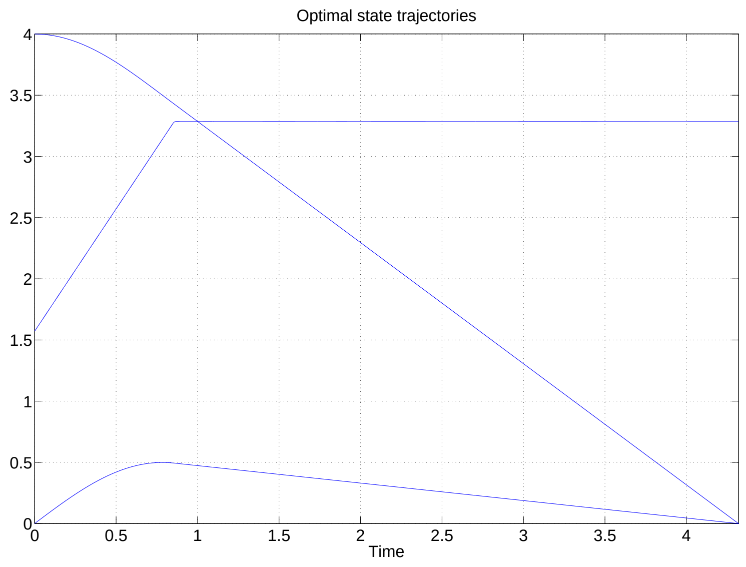

Figure 28: State trajectories for the singular arc problem

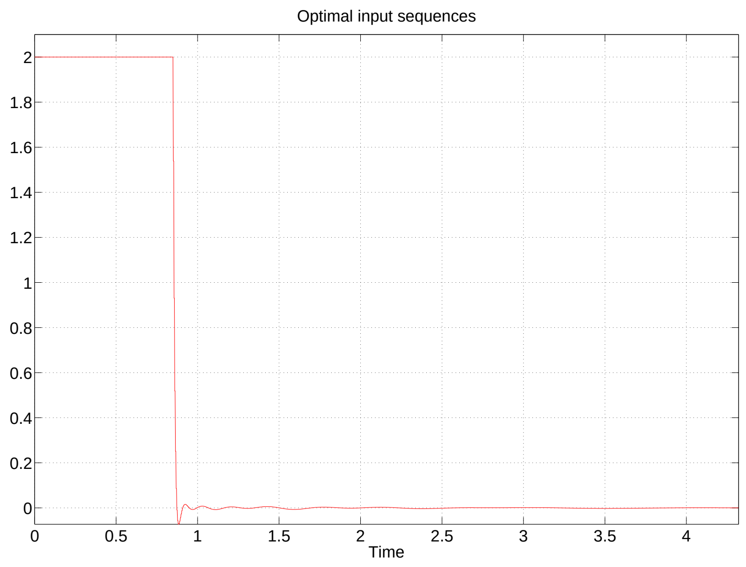

Figure 29: Input trajectory for the singular arc problem

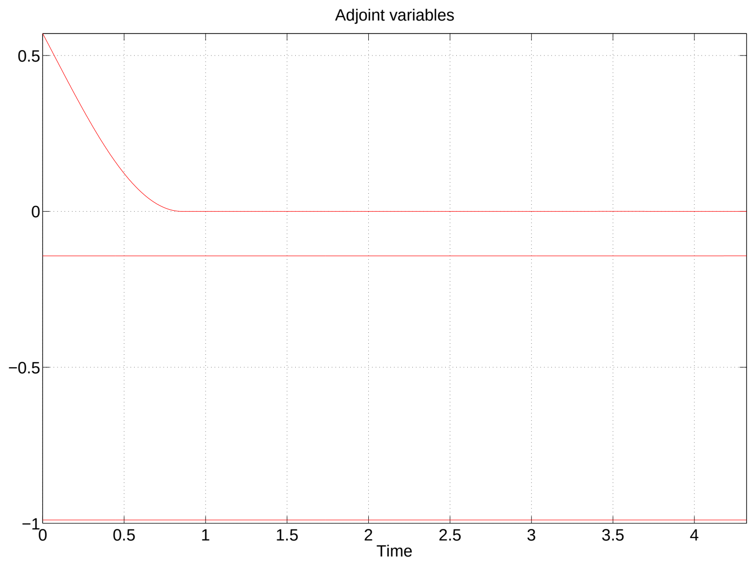

Figure 30: Adjoint variables for the singular arc problem

$$\min_{u(\cdot)} \int_0^1 (x_1^2 - x_2^2 + x_2^4 + 0.00001u^2)dt$$

subject to

$$\begin{cases} \dot{x}_1 = x_2 \\ \dot{x}_2 = u \end{cases}$$

$$x_1(1) = x_1(0),\ x_2(1) = x_2(0)$$

$$-1000 \le u(t) \le 1000$$
$$\begin{bmatrix} -100 \\ -100 \end{bmatrix} \le \begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} \le \begin{bmatrix} 100 \\ 100 \end{bmatrix} \quad \forall t \in [0,\ 1]$$

**Problem setup**

- The Optimal Control Problem is defined in the file `Speyer.m` in the following way:

% Initial time $t_0 < t_f$
```
problem.time.t0=0;
```

% Final time. Let tf_min=tf_max if $t_f$ is fixed.
```
problem.time.tf_min=1;
problem.time.tf_max=1;
```

```
guess.tf=1;
```

% Parameters bounds pl $\leq p \leq$ pu
```
problem.parameters.pl=[];
problem.parameters.pu=[];
guess.parameters=[];
```

% Initial condition $x(t_0)$ and its bounds
```
problem.states.x0=[ ];
problem.states.x0l=[-100 -100];
problem.states.x0u=[100 100];
```

% State bounds. xl $\leq x(t) \leq$ xu
```
problem.states.xl=[-100 -100];
problem.states.xu=[100 100];
```

% Terminal state bounds xfl $\leq x(t_f) \leq$ xfu
```
problem.states.xfl=[-100 -100];
problem.states.xfu=[100 100];
```

% Guess the state trajectories with $[x(t_0),\ x(t_f)]$
```
guess.states(:,1)=[1 1];
guess.states(:,2)=[1 1];
```

% Number of control actions N
% Set problem.inputs.N=0 if N is equal to the number of integration steps.
```
problem.inputs.N=0;
```

% Input bounds ul $\leq u(t) \leq$ uu
```
problem.inputs.ul=[-1000];
problem.inputs.uu=[1000];
```

% Guess the input sequences with $[u(t_0),\ u(t_f)]$
```
guess.inputs(:,1)=[1 -1];
```

% Choose the set-points if required
```
problem.setpoints.states=[];
problem.setpoints.inputs=[];
```

% Bounds for path constraint function gl $\leq g(x,u,p,t) \leq$ gu
```
problem.constraints.gl=[];
problem.constraints.gu=[];
```

```
% Bounds for boundary constraints bl ≤ b(x(t_0), x(t_f), u(t_0), u(t_f), p, t_0, t_f) ≤ bu
problem.constraints.bl=[0 0];
problem.constraints.bu=[0 0];

% Store the necessary problem parameters used in the functions
problem.data=[];

problem.functions={@L,@E,@f,@g,@b};

function stageCost=L(x,xr,u,ur,p,t,data)

    x1=x(:,1); x2=x(:,2); u=u(:,1);
    a=1;

    stageCost = 0.5*(x1.^2-a.*x2.^2+x2.^4+0.00001*u.^2);

function boundaryCost=E(x0,xf,u0,uf,p,tf,data)

    boundaryCost=0;

function dx = f(x,u,p,t,data)

    x1=x(:,1); x2=x(:,2); u=u(:,1);

    dx(:,1)=x2;
    dx(:,2)=u;

function c=g(x,u,p,t,data)

    c=[];

function bc=b(x0,xf,u0,uf,p,tf,data)
  bc=[x0-xf];
```

- The solution method and solver settings are set in `settings.m` . See the file included in the directory *../ICLOCS-*.*/examples/Speyersproblem*

- The solution of the optimisation problem is computed running the file `main.m` inside the directory *../ICLOCS-*.*/examples/Speyersproblem* The state, control and adjoint variables solution to this problem using the Optimal Control Toolbox are shown in Figs. 31, 32 and 33.

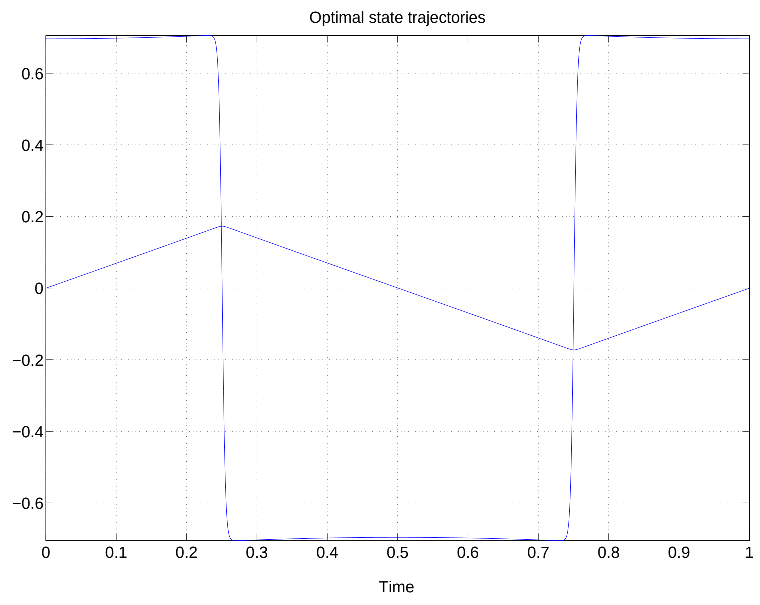

Figure 31: State trajectories for Speyer's problem

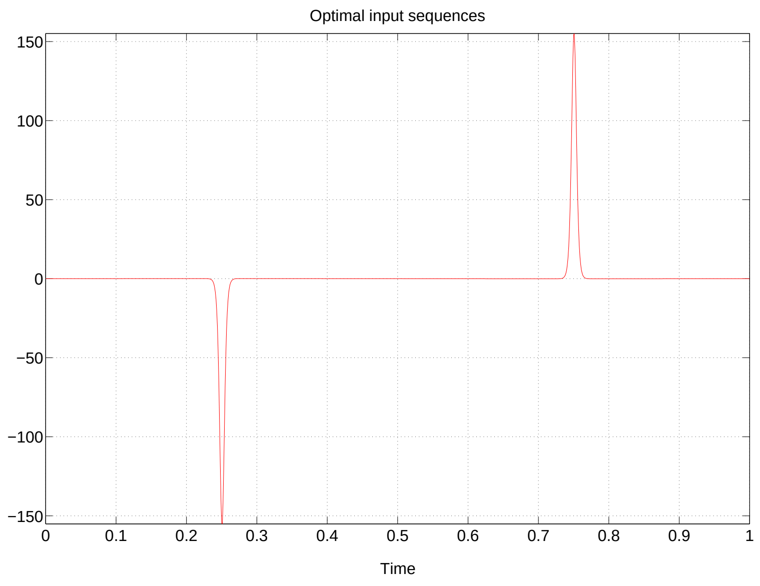

Figure 32: Input trajectory for Speyer's problem

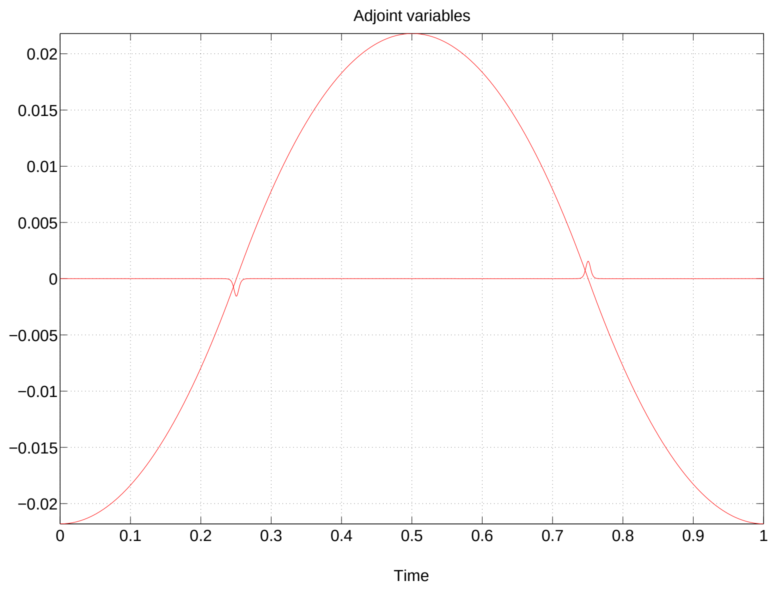

Figure 33: Adjoint variables for Speyer's problem

## 4.12 Example 12: Two-link Robot Arm

Find $t_f$ and $u(\cdot)$ over $t \in [0, t_f]$ solving the following optimal control problem of a two-link robot arm ( Section 12.4.2, Example 2 in [10])

$$\min_{u(\cdot),\ t_f} t_f + 0.01 \int_0^{tf} (u_1^2(t) + u_2^2(t))dt$$

subject to

$$\begin{cases} \dot{x}_1 = \dfrac{\sin(x_3)(\frac{9}{4}\cos(x_3)x_1^2) + 2x_2^2 + \frac{4}{3}(u_1 - u_2) - \frac{3}{2}\cos(x_3)u_2}{\frac{31}{36} + \frac{9}{4}\sin(x_3)^2} \\ \dot{x}_2 = -\dfrac{\sin(x_3)(\frac{9}{4}\cos(x_3)x_2^2) + \frac{7}{2}x_1^2 - \frac{7}{3}u_2 + \frac{3}{2}\cos(x_3)(u_1 - u_2)}{\frac{31}{36} + \frac{9}{4}\sin(x_3)^2} \\ \dot{x}_3 = x_2 - x_1 \\ \dot{x}_4 = x_1 \end{cases}$$

$x(0) = [0,\ 0,\ 0,\ 0]$
$x(t_f) = [0,\ 0,\ 0.5,\ 0.522]$

$-1 \le u(t) \le 1$
$t_f \ge 0.1$

### Problem setup

- The Optimal Control Problem is defined in the file `TwoLinkRobotArm.m` in the following way:

  % Initial time $t_0 < t_f$
  ```
  problem.time.t0=0;
  ```

  % Final time. Let tf_min=tf_max if $t_f$ is fixed.
  ```
  problem.time.tf_min=0.1;
  problem.time.tf_max=inf;
  guess.tf=3.1;
  ```

  % Parameters bounds pl $\leq p \leq$ pu
  ```
  problem.parameters.pl=[];
  problem.parameters.pu=[];
  guess.parameters=[];
  ```

  % Initial condition $x(t_0)$ and its bounds
  ```
  problem.states.x0=[0 0 0 0];
  problem.states.x0l=[0 0 0 0];
  problem.states.x0u=[0 0 0 0];
  ```

  % State bounds. xl $\leq x(t) \leq$ xu
  ```
  problem.states.xl=[-inf -inf -inf -inf];
  problem.states.xu=[inf inf inf inf];
  ```

  % Terminal state bounds xfl $\leq x(t) \leq$ xfu
  ```
  problem.states.xfl=[0 0 0.5 0.522];
  problem.states.xfu=[0 0 0.5 0.522];
  ```

  % Guess the state trajectories with $[x(t_0),\ x(t_f)]$
  ```
  guess.states(:,1)=[0.1 0];
  guess.states(:,2)=[0.1 0];
  guess.states(:,3)=[0 0.5];
  guess.states(:,4)=[0 0.522];
  ```

  % Number of control actions N.
  % Set problem.inputs.N=0 if N is equal to the number of integration steps.
  ```
  problem.inputs.N=0;
  ```

  % Input bounds ul $\leq u(t) \leq$ uu
  ```
  problem.inputs.ul=[-1 -1];
  ```

```
problem.inputs.uu=[1 1];
```

% Guess the input sequences with $[u(t_0),\ u(t_f)]$
```
guess.inputs=[];
```

% Choose the set-points if required
```
problem.setpoints.states=[];
problem.setpoints.inputs=[];
```

% Bounds for path constraint function gl $\leq g(x,u,p,t) \leq$ gu
```
problem.constraints.gl=[];
problem.constraints.gu=[];
```

% Bounds for boundary constraints bl $\leq b(x(t_0),x(t_f),u(t_0),u(t_f),p,t_0,t_f) \leq$ bu
```
problem.constraints.bl=[];
problem.constraints.bu=[];
```

% Store the necessary problem parameters used in the functions
```
problem.data=[];


problem.functions={@L,@E,@f,@g,@b};


function stageCost=L(x,xr,u,ur,p,t,data)
 stageCost = 0.01*(u(:,1).*u(:,1)+u(:,2).*u(:,2));


function boundaryCost=E(x0,xf,u0,uf,p,tf,data)
 boundaryCost=tf;


function dx = f(x,u,p,t,data)
 x1 = x(:,1); x2 = x(:,2); x3 = x(:,3);
    u1 = u(:,1); u2 = u(:,2);


    dx(:,1) = ( sin(x3).*(9/4*cos(x3).*x1.^2)+2*x2.^2 + 4/3*(u1-u2) ...
            - 3/2*cos(x3).*u2 )./ (31/36 + 9/4*sin(x3).^2);


    dx(:,2) = -( sin(x3).*(9/4*cos(x3).*x2.^2)+7/2*x1.^2 - 7/3*u2 ...
            + 3/2*cos(x3).*(u1-u2) )./ (31/36 + 9/4*sin(x3).^2);


    dx(:,3) = x2-x1;
```

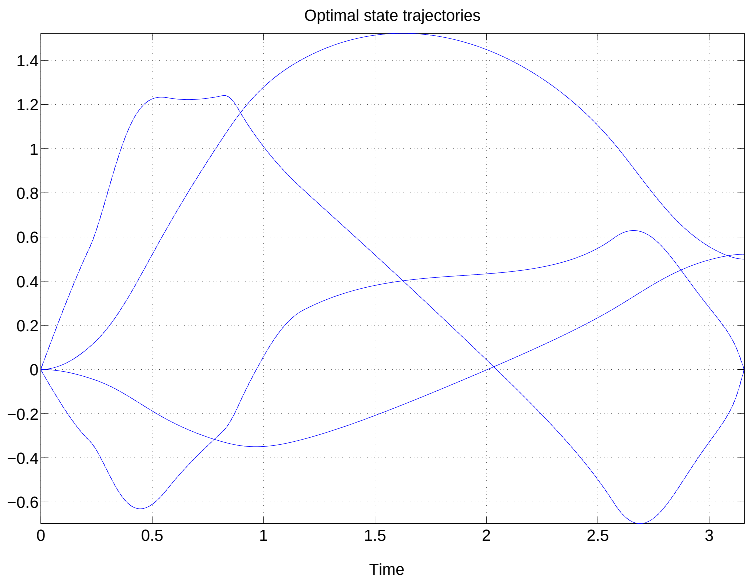

Figure 34: State trajectories for the two-link robot arm problem

```
    dx(:,4) = x1;

function c=g(x,u,p,t,data)
 c=[];

function bc=b(x0,xf,u0,uf,p,tf,data)
 bc=[];
```

- The solution method and solver settings are set in `settings.m` . See the file included in the directory *../ICLOCS-*.*/examples/TwoLinkRobotArm*

- The solution of the optimisation problem is computed running the file `main.m`. inside the directory *../ICLOCS-*.*/examples/TwoLinkRobotArm* The state, control and adjoint variables solution to this problem using the Optimal Control Toolbox are shown in Figs. 34, 35 and 36.

## References

[1] John T. Betts. *Practical Methods for Optimal Control Using Nonlinear Programming.* Advances in Design and Control. SIAM, Philadelphia, PA, 2001.

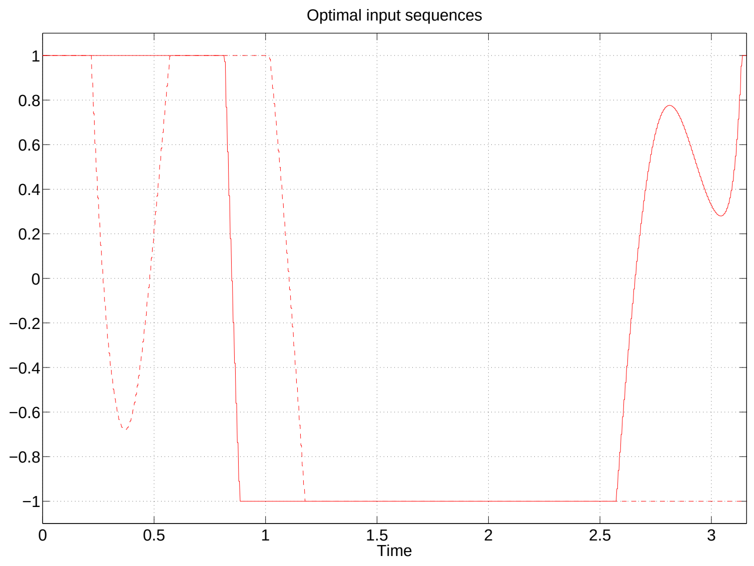

Figure 35: Input trajectory for the two-link robot arm problem

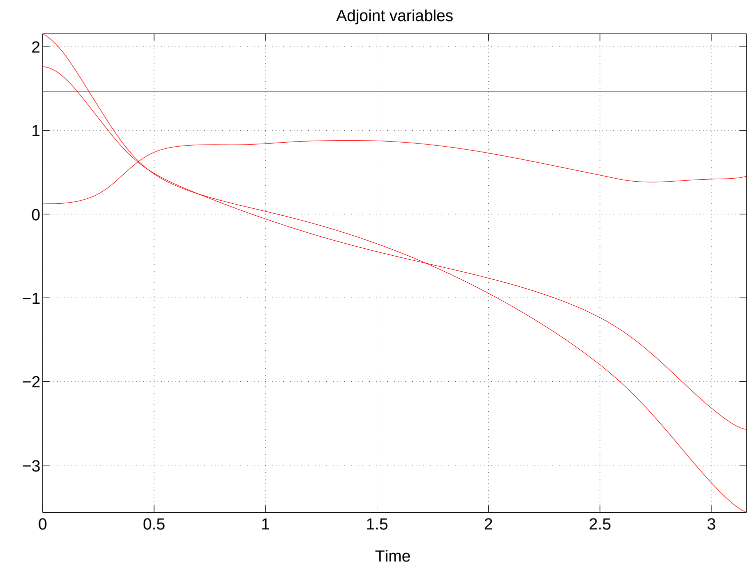

Figure 36: Adjoint variables for the two-link robot arm problem

[2] A. E. Bryson. *Dynamic Optimization.* Addison Wesley Longman, Menlo Park, CA, USA, 1999.

[3] A. E. Bryson and Y.C. Ho. *Applied optimal control.* John Wiley & Sons., 1975.

[4] J.E. Cuthrell and L. T. Biegler. Simultaneous optimization and solution methods for batch reactor control profiles. *Comput. Chem. Eng.*, 13:49–62, 1989.

[5] Wendell H. Fleming and Raymond W. Rishel. *Deterministic and stochastic optimal control.* Springer-Verlag, Berlin, New York, 1975.

[6] G. Hicks and W. Ray. Approximation methods for optimal control synthesis. *The Canadian Journal of Chemical Engineering*, 49:522–528, 1971.

[7] L. S. Jennings and M. E. Fisher. *MISER3 optimal control toolbox, User Manual.* Department of Mathematics, The University of Western Australia.

[8] S. Kameswaran and L. T. Biegler. Simultaneous dynamic optimization strategies: recent advances and challenges. *Computers and Chemical Engineering*, 30:1560–1575, 2006.

[9] Y. Chen Liang, M. Meng and R. Fullmer. Solving tough optimal control problems by network enabled optimization server (neos). Technical report, School of Engineering, Utah State University USA, Chinene University of Hong Kong China, 2003.

[10] Rein Luus. *Iterative Dynamic Programming.* Chapman & Hall/CRC Monographs and Surveys in Pure and Applied Mathematics, 2000.

[11] J.L. Speyer. Periodic optimal flight. *Journal of Guidance, Control, and Dynamics*, 61:745–755, 1996.

[12] J. R. Banga V. S. Vassiliadis and E. Balsa-Canto. Second-order sensitivities of general dynamic systems with application to optimal control problems. *Chemical Engineering Science*, 54:3851–3860, 1999.