PSOPT Application Examples Document

Victor Becerra
Email: v.m.becerra@ieee.org
www: http://www.psopt.net
March 22, 2025

Contents

$\mathcal{PSOPT}$ Application Examples

Victor Becerra
Email: v.m.becerra@ieee.org

March 22, 2025

Most of the following examples have been selected from the literature such that their solutions can be compared with published results by consulting the references provided. Although source code is only shown here for a selection of the examples, the source code for all examples can be found in the $\mathcal{PSOPT}$ software distribution. Users are advised to study the source code of some of the examples before attempting to code their own problems. Note that not all examples available the distribution are included in this document.

1 Alp rider problem

Consider the following optimal control problem, which is known in the literature as the Alp rider problem [3]. It is known as Alp rider because the minimum of the objective function forces the states to ride the path constraint. Minimize the cost functional

$$J = \int_0^{20} (100(x_1^2 + x_2^2 + x_3^2 + x_4^2) + 0.01(u_1^2 + u_2^2))\mathrm{d}t \tag{1}$$

subject to the dynamic constraints

$$\begin{array}{rcl} \dot{x}_1 & = & -10x_1 + u_1 + u_2 \\ \dot{x}_2 & = & -2x_2 + u_1 + 2u_2 \\ \dot{x}_3 & = & -3x_3 + 5x_4 + u_1 - u_2 \\ \dot{x}_4 & = & 5x_3 - 3x_4 + u_1 + 3u_2 \end{array} \tag{2}$$

the path constraint

$$x_1^2 + x_2^2 + x_3^2 + x_4^2 - 3p(t,3,12) - 3p(t,6,10) - 3p(t,10,16) - 8p(t,15,4) - 0.01 \leq 0 \tag{3}$$

where the exponential peaks are $p(t,a,b) = e^{-b(t-a)^2}$, and the boundary conditions are given by:

$$\begin{aligned} x_1(0) &= 2 \\ x_2(0) &= 1 \\ x_3(0) &= 2 \\ x_4(0) &= 1 \\ x_1(20) &= 2 \\ x_2(20) &= 3 \\ x_3(20) &= 1 \\ x_4(20) &= -2 \end{aligned} \tag{4}$$

The C++ code that solves this problem is shown below.

```
//////////////////////////////////////////////////////////////////////////
////////////////        alpine_rider.cxx            /////////////////////
//////////////////////////////////////////////////////////////////////////
////////////////           PSOPT  Example           /////////////////////
//////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////
//////// Title:         Alp rider problem                ////////////////
//////// Last modified: 09 February 2009                 ////////////////
//////// Reference:     Betts (2001)                     ////////////////
//////// (See PSOPT handbook for full reference)         ////////////////
//////////////////////////////////////////////////////////////////////////
////////     Copyright (c) Victor M. Becerra, 2009        ////////////////
//////////////////////////////////////////////////////////////////////////
//////// This is part of the PSOPT software library, which////////////////
//////// is distributed under the terms of the GNU Lesser ////////////////
//////// General Public License (LGPL)                   ////////////////
//////////////////////////////////////////////////////////////////////////

#include "psopt.h"

//////////////////////////////////////////////////////////////////////////
///////////////////  Define auxiliary function    ///////////////////////
//////////////////////////////////////////////////////////////////////////

adouble pk( adouble t, double a, double b)
{
   return exp(-b*(t-a)*(t-a));
}

//////////////////////////////////////////////////////////////////////////
///////////////////  Define the end point (Mayer) cost function //////////
//////////////////////////////////////////////////////////////////////////

adouble endpoint_cost(adouble* initial_states, adouble* final_states,
                      adouble* parameters,adouble& t0, adouble& tf,
                      adouble* xad, int iphase, Workspace* workspace)
{
   return 0;
}

//////////////////////////////////////////////////////////////////////////
///////////////////  Define the integrand (Lagrange) cost function  //////
//////////////////////////////////////////////////////////////////////////

adouble integrand_cost(adouble* states, adouble* controls,
                       adouble* parameters, adouble& time, adouble* xad,
                       int iphase, Workspace* workspace)
{
    adouble x1 = states[0];
    adouble x2 = states[1];
    adouble x3 = states[2];
    adouble x4 = states[3];
    adouble u1 = controls[0];
    adouble u2 = controls[1];

    adouble L;

    L = 100.0*( x1*x1 + x2*x2 + x3*x3 + x4*x4 ) + 0.01*( u1*u1 + u2*u2 );
```

```
    return  L;
}


//////////////////////////////////////////////////////////////////////////
///////////////////  Define the DAE's ////////////////////////////////////
//////////////////////////////////////////////////////////////////////////

void dae(adouble* derivatives, adouble* path, adouble* states,
         adouble* controls, adouble* parameters, adouble& time,
         adouble* xad, int iphase, Workspace* workspace)
{
    adouble x1 = states[0];
    adouble x2 = states[1];
    adouble x3 = states[2];
    adouble x4 = states[3];
    adouble u1 = controls[0];
    adouble u2 = controls[1];
    adouble t  = time;

   derivatives[0] = -10*x1 + u1 + u2;
   derivatives[1] = -2*x2 + u1 + 2*u2;
   derivatives[2] = -3*x3 + 5*x4 + u1 - u2;
   derivatives[3] =  5*x3 - 3*x4 + u1 + 3*u2;

   path[0] = x1*x1 + x2*x2 + x3*x3 + x4*x4
             - 3*pk(t,3,12) - 3*pk(t,6,10) - 3*pk(t,10,6) - 8*pk(t,15,4)
             - 0.01;

}

////////////////////////////////////////////////////////////////////////////
///////////////////  Define the events function ////////////////////////////
////////////////////////////////////////////////////////////////////////////

void events(adouble* e, adouble* initial_states, adouble* final_states,
            adouble* parameters,adouble& t0, adouble& tf, adouble* xad,
            int iphase, Workspace* workspace)

{
   adouble x1i = initial_states[ 0 ];
   adouble x2i = initial_states[ 1 ];
   adouble x3i = initial_states[ 2 ];
   adouble x4i = initial_states[ 3 ];
   adouble x1f = final_states[ 0 ];
   adouble x2f = final_states[ 1 ];
   adouble x3f = final_states[ 2 ];
   adouble x4f = final_states[ 3 ];



   e[ 0 ] = x1i;
   e[ 1 ] = x2i;
   e[ 2 ] = x3i;
   e[ 3 ] = x4i;
   e[ 4 ] = x1f;
   e[ 5 ] = x2f;
   e[ 6 ] = x3f;
   e[ 7 ] = x4f;

}



///////////////////////////////////////////////////////////////////////////
///////////////////  Define the phase linkages function ///////////////////
///////////////////////////////////////////////////////////////////////////

void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{
  // No linkages as this is a single phase problem
}



////////////////////////////////////////////////////////////////////////////
///////////////////  Define the main routine ///////////////////////////////
////////////////////////////////////////////////////////////////////////////

int main(void)
{
```

```
////////////////////////////////////////////////////////////////////////////
//////////////////  Declare key structures ////////////////////////////////
////////////////////////////////////////////////////////////////////////////

    Alg  algorithm;
    Sol  solution;
    Prob problem;

////////////////////////////////////////////////////////////////////////////
///////////////////  Register problem name  ////////////////////////////////
////////////////////////////////////////////////////////////////////////////

    problem.name         = "Alp rider problem";
    problem.outfilename                = "alpine.txt";

////////////////////////////////////////////////////////////////////////////
////////////  Define problem level constants & do level 1 setup ////////////
////////////////////////////////////////////////////////////////////////////

    problem.nphases    = 1;
    problem.nlinkages                  = 0;

    psopt_level1_setup(problem);

/////////////////////////////////////////////////////////////////////////////
/////////   Define phase related information & do level 2 setup  ////////////
/////////////////////////////////////////////////////////////////////////////

    problem.phases(1).nstates    = 4;
    problem.phases(1).ncontrols  = 2;
    problem.phases(1).nevents    = 8;
    problem.phases(1).npath      = 1;
    problem.phases(1).nodes                    << 120;



    psopt_level2_setup(problem, algorithm);

////////////////////////////////////////////////////////////////////////////
//////////////////  Enter problem bounds information //////////////////////
////////////////////////////////////////////////////////////////////////////


    problem.phases(1).bounds.lower.states <<    -4.0, -4.0, -4.0, -4.0;

    problem.phases(1).bounds.upper.states <<     4.0,  4.0,  4.0,  4.0;

    problem.phases(1).bounds.lower.controls <<  -500.0, -500 ;

    problem.phases(1).bounds.upper.controls <<   500.0,  500 ;

    problem.phases(1).bounds.lower.events <<  2.0, 1.0, 2.0, 1.0, 2.0, 3.0, 1.0, -2.0;

    problem.phases(1).bounds.upper.events = problem.phases(1).bounds.lower.events;

    problem.phases(1).bounds.upper.path         << 100.0;

    problem.phases(1).bounds.lower.path         << 0.0;

    problem.phases(1).bounds.lower.StartTime    = 0.0;

    problem.phases(1).bounds.upper.StartTime    = 0.0;

    problem.phases(1).bounds.lower.EndTime      = 20.0;

    problem.phases(1).bounds.upper.EndTime      = 20.0;


////////////////////////////////////////////////////////////////////////////
///////////////////  Register problem functions  ///////////////////////////
////////////////////////////////////////////////////////////////////////////


    problem.integrand_cost  = &integrand_cost;
    problem.endpoint_cost     = &endpoint_cost;
    problem.dae             = &dae;
    problem.events        = &events;
    problem.linkages       = &linkages;

////////////////////////////////////////////////////////////////////////////
//////////////////  Define & register initial guess ///////////////////////
////////////////////////////////////////////////////////////////////////////
```

```
    int nnodes                    = problem.phases(1).nodes(0);

    MatrixXd x_guess                =  zeros(4,nnodes);

    x_guess.row(0)                 = linspace(2,1,nnodes);
    x_guess.row(1)                 = linspace(2,3,nnodes);
    x_guess.row(2)                 = linspace(2,1,nnodes);
    x_guess.row(3)                 = linspace(1,-2,nnodes);

    problem.phases(1).guess.controls      = zeros(2,nnodes);
    problem.phases(1).guess.states        = x_guess;
    problem.phases(1).guess.time          = linspace(0.0,20.0,nnodes+1);


////////////////////////////////////////////////////////////////////////////
///////////////////  Enter algorithm options  //////////////////////////////
////////////////////////////////////////////////////////////////////////////


    algorithm.nlp_iter_max                = 1000;
    algorithm.nlp_tolerance               = 1.e-6;
    algorithm.nlp_method                  = "IPOPT";
    algorithm.scaling                     = "automatic";
    algorithm.derivatives                 = "automatic";
    algorithm.jac_sparsity_ratio          = 0.20;
    algorithm.collocation_method          = "Legendre";
    algorithm.diff_matrix                 = "central-differences";
    algorithm.mesh_refinement             = "automatic";
    algorithm.mr_max_increment_factor     = 0.3;
    algorithm.mr_max_iterations           = 3;
    algorithm.defect_scaling              = "jacobian-based";





////////////////////////////////////////////////////////////////////////////
///////////////////  Now call PSOPT to solve the problem   //////////////////
////////////////////////////////////////////////////////////////////////////

    psopt(solution, problem, algorithm);

////////////////////////////////////////////////////////////////////////////
///////////  Extract relevant variables from solution structure   //////////
////////////////////////////////////////////////////////////////////////////


    MatrixXd x = solution.get_states_in_phase(1);
    MatrixXd u = solution.get_controls_in_phase(1);
    MatrixXd t = solution.get_time_in_phase(1);


////////////////////////////////////////////////////////////////////////////
///////////  Save solution data to files if desired ////////////////////////
////////////////////////////////////////////////////////////////////////////

   Save(x,"x.dat");
   Save(u,"u.dat");
   Save(t,"t.dat");


////////////////////////////////////////////////////////////////////////////
///////////  Plot some results if desired (requires gnuplot) ///////////////
////////////////////////////////////////////////////////////////////////////

    plot(t,x.row(0),problem.name+": state", "time (s)", "state","x1");

    plot(t,x.row(1),problem.name+": state", "time (s)", "state","x2");

    plot(t,x.row(2),problem.name+": state", "time (s)", "state","x3");

    plot(t,x.row(3),problem.name+": state", "time (s)", "state","x4");

    plot(t,u.row(0),problem.name+": control","time (s)", "control", "u1");

    plot(t,u.row(1),problem.name+": control","time (s)", "control", "u2");

    plot(t,x.row(0),problem.name+": state x1", "time (s)", "state","x1",
                                        "pdf", "alpine_state1.pdf");

    plot(t,x.row(1),problem.name+": state x2", "time (s)", "state","x2",
```

```
                                    "pdf", "alpine_state2.pdf");

    plot(t,x.row(2),problem.name+": state x3", "time (s)", "state","x3",
                                    "pdf", "alpine_state3.pdf");

    plot(t,x.row(3),problem.name+": state x4", "time (s)", "state","x4",
                                    "pdf", "alpine_state4.pdf");

    plot(t,u.row(0),problem.name+": control u1","time (s)", "control", "u1",
                                    "pdf", "alpine_control1.pdf");

    plot(t,u.row(1),problem.name+": control u1","time (s)", "control", "u2",
                                    "pdf", "alpine_control2.pdf");

}

////////////////////////////////////////////////////////////////////////////
///////////////////////      END OF FILE     ///////////////////////////////
////////////////////////////////////////////////////////////////////////////
```

The output from $\mathcal{PSOPT}$ is summarized in the box below and shown in Figures 1-4 and Figures 5-6, which contain the elements of the state and the control, respectively. Table 1 shows the mesh refinement history for this problem.

```
PSOPT results summary
=====================

Problem:  Alp rider problem
CPU time (seconds): 1.554232e+01
NLP solver used:  IPOPT
PSOPT release number:  5.0.3
Date and time of this run:  Thu Mar  6 16:49:57 2025

Optimal (unscaled) cost function value:  2.026363e+03
Phase 1 endpoint cost function value: 0.000000e+00
Phase 1 integrated part of the cost:  2.026363e+03
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 2.000000e+01
Phase 1 maximum relative local error: 3.807700e-03
NLP solver reports:  The problem has been solved!
```

2 Brachistochrone problem

Consider the following optimal control problem. Minimize the cost functional

$$J = t_f \tag{5}$$

Table 1: Mesh refinement statistics: Alp rider problem

Iter	DM	M	NV	NC	OE	CE	JE	HE	RHS	$\epsilon_{\max}$	$\mathrm{CPU_a}$
1	LGL-CD	120	722	609	510	510	135	0	61200	2.211e-03	1.206e+00
2	LGL-CD	125	752	634	3605	3606	660	0	450750	3.461e-03	6.981e+00
3	LGL-CD	126	758	639	1138	1139	352	0	143514	3.808e-03	3.089e+00
$\mathrm{CPU_b}$	-	-	-	-	-	-	-	-	-	-	4.267e+00
-	-	-	-	-	5253	5255	1147	0	655464	-	1.554e+01

Key: Iter=iteration number, DM= discretization method, M=number of nodes, NV=number of variables, NC=number of constraints, OE=objective evaluations, CE = constraint evaluations, JE = Jacobian evaluations, HE = Hessian evaluations, RHS = ODE right hand side evaluations, $\epsilon_{\max}$ = maximum relative ODE error, $\mathrm{CPU_a}$ = CPU time in seconds spent by NLP algorithm, $\mathrm{CPU_b}$ = additional CPU time in seconds spent by PSOPT

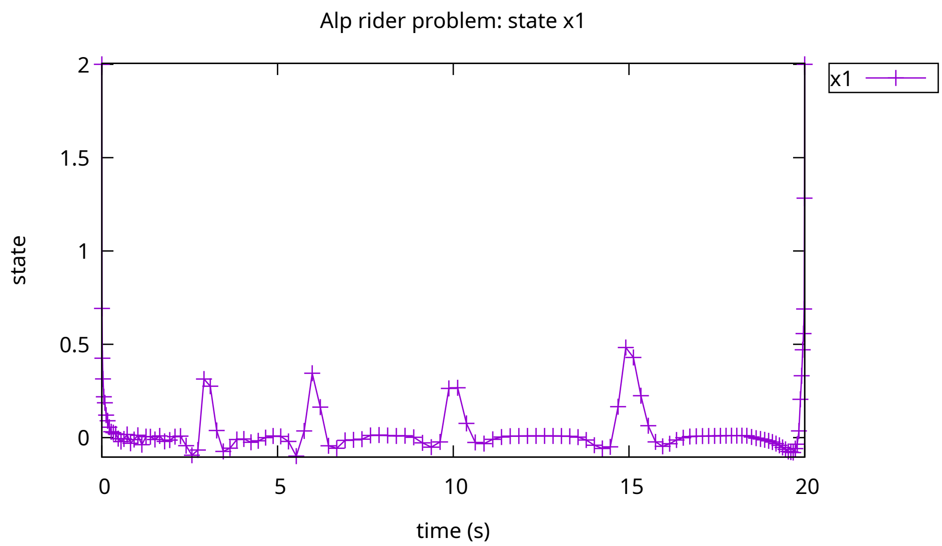

Figure 1: State $x_1(t)$ for the Alp rider problem

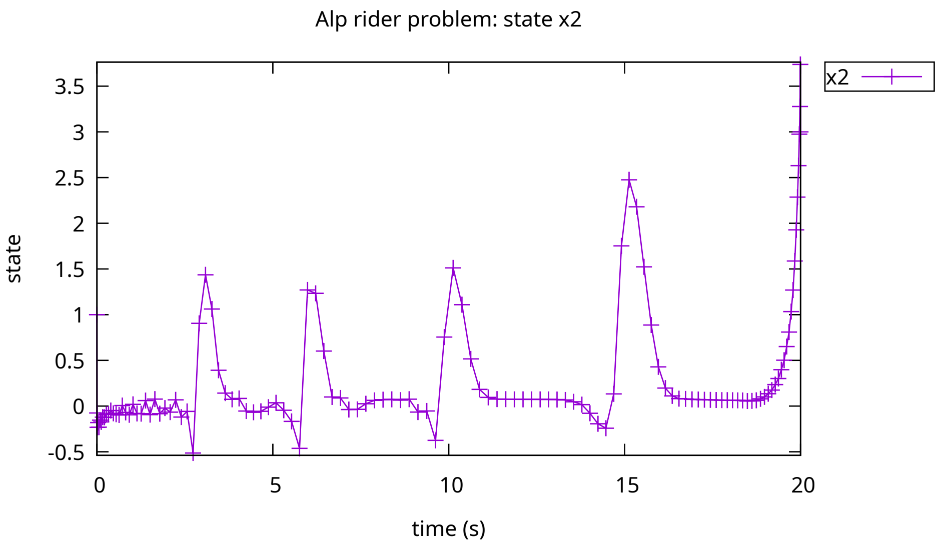

Figure 2: State $x_2(t)$ for the Alp rider problem

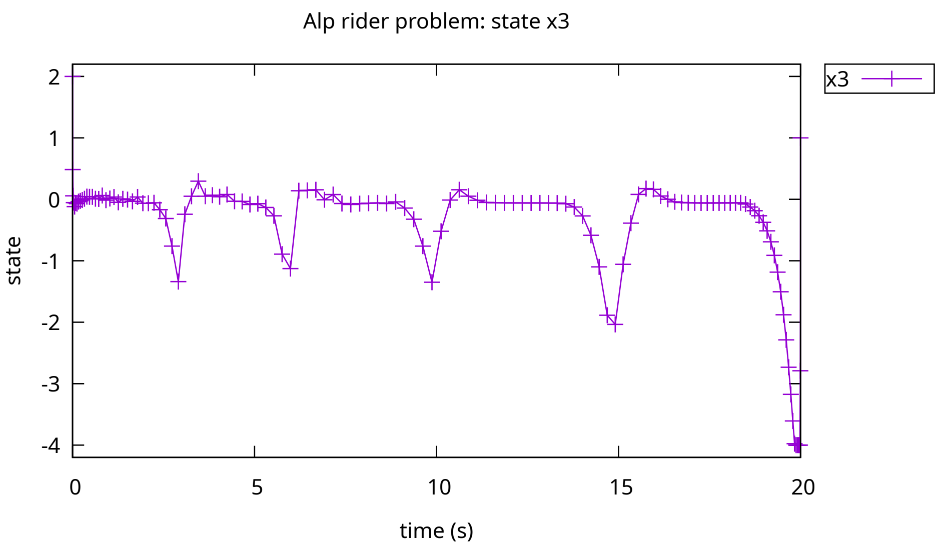

Figure 3: State $x_3(t)$ for the Alp rider problem

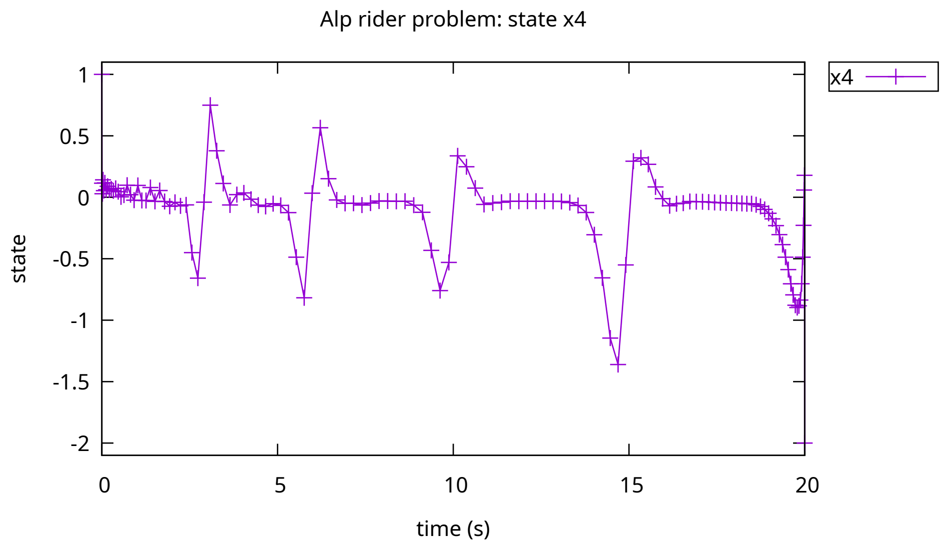

Figure 4: State $x_4(t)$ for the Alp rider problem

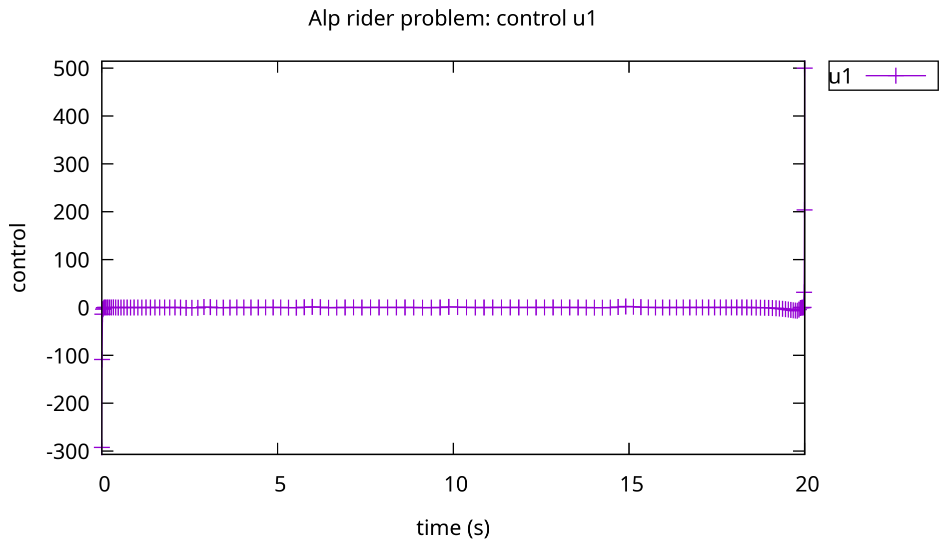

Figure 5: Control $u_1(t)$ for the Alp rider problem

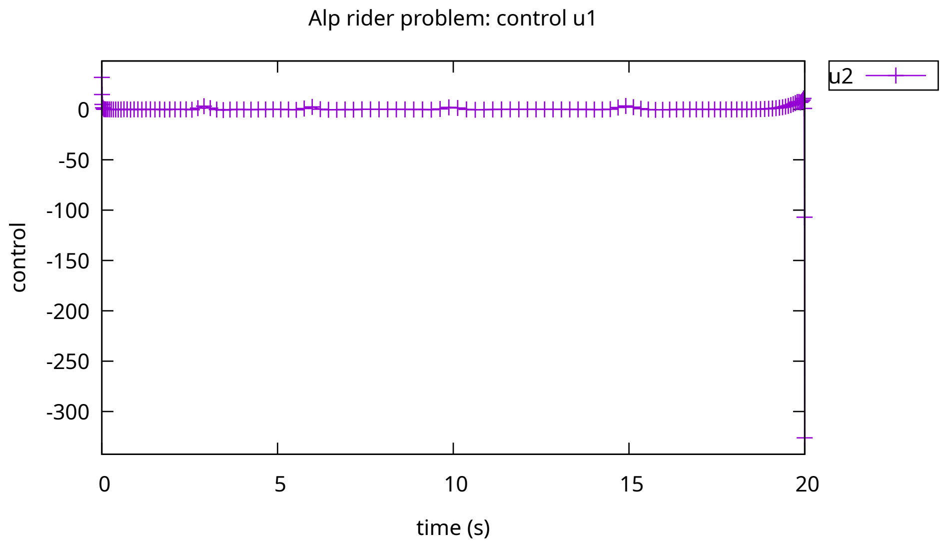

Figure 6: Control $u_2(t)$ for the Alp rider problem

subject to the dynamic constraints

$$\begin{aligned} \dot{x} &= v\sin(\theta) \\ \dot{y} &= v\cos(\theta) \\ \dot{v} &= g\cos(\theta) \end{aligned} \tag{6}$$

and the boundary conditions

$$\begin{aligned} x(0) &= 0 \\ y(0) &= 0 \\ v(0) &= 0 \\ x(t_f) &= 2 \\ y(t_f) &= 2 \end{aligned} \tag{7}$$

where $g = 9.8$. A version of this problem was originally formulated by Johann Bernoulli in 1696 and is referred to as the *Brachistochrone* problem. The C++ code that solves this problem is shown below.

```
//////////////////////////////////////////////////////////////////////////
//////////////////            brac1.cxx        ///////////////////////////
//////////////////////////////////////////////////////////////////////////
////////////////           PSOPT  Example             ////////////////////
//////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////
//////// Title:                 Brachistochrone problem   ////////////////
//////// Last modified:         04 January 2009           ////////////////
//////// Reference:             Bryson and Ho (1975)      ////////////////
//////// (See PSOPT handbook for full reference)          ////////////////
//////////////////////////////////////////////////////////////////////////
////////     Copyright (c) Victor M. Becerra, 2009        ////////////////
//////////////////////////////////////////////////////////////////////////
//////// This is part of the PSOPT software library, which ///////////////
```

```
//////// is distributed under the terms of the GNU Lesser ////////////////
//////// General Public License (LGPL)                    ////////////////
//////////////////////////////////////////////////////////////////////////

#include "psopt.h"

using namespace PSOPT;

//////////////////////////////////////////////////////////////////////////
///////////////////  Define the end point (Mayer) cost function //////////
//////////////////////////////////////////////////////////////////////////

adouble endpoint_cost(adouble* initial_states, adouble* final_states,
                      adouble* parameters,adouble& t0, adouble& tf,
                      adouble* xad, int iphase, Workspace* workspace)
{
    return tf;
}

//////////////////////////////////////////////////////////////////////////
///////////////////  Define the integrand (Lagrange) cost function  //////
//////////////////////////////////////////////////////////////////////////

adouble integrand_cost(adouble* states, adouble* controls, adouble* parameters,
                     adouble& time, adouble* xad, int iphase, Workspace* workspace)
{
    return  0.0;
}

//////////////////////////////////////////////////////////////////////////
///////////////////  Define the DAE's ////////////////////////////////////
//////////////////////////////////////////////////////////////////////////

void dae(adouble* derivatives, adouble* path, adouble* states,
         adouble* controls, adouble* parameters, adouble& time,
         adouble* xad, int iphase, Workspace* workspace)
{
   adouble xdot, ydot, vdot;

   adouble x = states[ 0 ];
   adouble y = states[ 1 ];
   adouble v = states[ 2 ];

   adouble theta = controls[ 0 ];

   xdot = v*sin(theta);
   ydot = v*cos(theta);
   vdot = 9.8*cos(theta);

   derivatives[ 0 ] = xdot;
   derivatives[ 1 ] = ydot;
   derivatives[ 2 ] = vdot;
}

////////////////////////////////////////////////////////////////////////////
///////////////////  Define the events function ////////////////////////////
////////////////////////////////////////////////////////////////////////////

void events(adouble* e, adouble* initial_states, adouble* final_states,
            adouble* parameters,adouble& t0, adouble& tf, adouble* xad,
            int iphase, Workspace* workspace)
{
   adouble x0 = initial_states[ 0 ];
   adouble y0 = initial_states[ 1 ];
   adouble v0 = initial_states[ 2 ];
   adouble xf = final_states[ 0];
   adouble yf = final_states[ 1];

   e[ 0 ] = x0;
   e[ 1 ] = y0;
   e[ 2 ] = v0;
   e[ 3 ] = xf;
   e[ 4 ] = yf;
}

///////////////////////////////////////////////////////////////////////////
///////////////////  Define the phase linkages function ///////////////////
///////////////////////////////////////////////////////////////////////////

void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{
  // No linkages as this is a single phase problem
```

```
}


////////////////////////////////////////////////////////////////////////////
//////////////////  Define the main routine ///////////////////////////////
////////////////////////////////////////////////////////////////////////////


int main(void)
{

////////////////////////////////////////////////////////////////////////////
///////////////////  Declare key structures ////////////////////////////////
////////////////////////////////////////////////////////////////////////////

    Alg  algorithm;
    Sol  solution;
    Prob problem;

////////////////////////////////////////////////////////////////////////////
///////////////////  Register problem name  ////////////////////////////////
////////////////////////////////////////////////////////////////////////////

    problem.name        = "Brachistochrone Problem";

    problem.outfilename                = "brac1.txt";

////////////////////////////////////////////////////////////////////////////
////////////  Define problem level constants & do level 1 setup ////////////
////////////////////////////////////////////////////////////////////////////

    problem.nphases   = 1;
    problem.nlinkages                  = 0;

    psopt_level1_setup(problem);


/////////////////////////////////////////////////////////////////////////////
/////////   Define phase related information & do level 2 setup ////////////
/////////////////////////////////////////////////////////////////////////////

    problem.phases(1).nstates    = 3;
    problem.phases(1).ncontrols  = 1;
    problem.phases(1).nevents    = 5;
    problem.phases(1).npath      = 0;
    problem.phases(1).nodes                    << 40;

    psopt_level2_setup(problem, algorithm);


////////////////////////////////////////////////////////////////////////////
///////////////////  Enter problem bounds information //////////////////////
////////////////////////////////////////////////////////////////////////////


    problem.phases(1).bounds.lower.states    <<  0,  0,  0;
    problem.phases(1).bounds.upper.states    << 20, 20, 20;

    problem.phases(1).bounds.lower.controls  << 0.0;
    problem.phases(1).bounds.upper.controls  << 2*pi;

    problem.phases(1).bounds.lower.events    << 0,  0,  0,  2,  2;
    problem.phases(1).bounds.upper.events    << 0,  0,  0,  2,  2;


    problem.phases(1).bounds.lower.StartTime    = 0.0;
    problem.phases(1).bounds.upper.StartTime    = 0.0;

    problem.phases(1).bounds.lower.EndTime      = 0.0;
    problem.phases(1).bounds.upper.EndTime      = 10.0;


////////////////////////////////////////////////////////////////////////////
///////////////////  Register problem functions  ///////////////////////////
////////////////////////////////////////////////////////////////////////////


    problem.integrand_cost  = &integrand_cost;
    problem.endpoint_cost  = &endpoint_cost;
    problem.dae  = &dae;
    problem.events  = &events;
    problem.linkages = &linkages;
```

```
////////////////////////////////////////////////////////////////////////////
//////////////////  set scaling factors (optional) ////////////////////////
////////////////////////////////////////////////////////////////////////////

//    problem.phases(1).scale.controls    = 1.0*ones(1,1);
//    problem.phases(1).scale.states      = 1.0*ones(3,1);
//    problem.phases(1).scale.events      = 1.0*ones(5,1);
//    problem.phases(1).scale.defects     = 1.0*ones(3,1);
//    problem.phases(1).scale.time        = 1.0;

//    problem.scale.objective             = 1.0;

////////////////////////////////////////////////////////////////////////////
//////////////////  Define & register initial guess ///////////////////////
////////////////////////////////////////////////////////////////////////////



    MatrixXd x0(3,20);

    x0.row(0) = linspace(0.0,1.0, 20);
    x0.row(1) = linspace(0.0,1.0, 20);
    x0.row(2) = linspace(0.0,1.0, 20);

    problem.phases(1).guess.controls       = ones(1,20);
    problem.phases(1).guess.states         = x0;
    problem.phases(1).guess.time           = linspace(0.0, 2.0, 20);

////////////////////////////////////////////////////////////////////////////
//////////////////  Enter algorithm options  //////////////////////////////
////////////////////////////////////////////////////////////////////////////


    algorithm.nlp_method                  = "IPOPT";
    algorithm.scaling                     = "automatic";
    algorithm.derivatives                 = "automatic";
    algorithm.nlp_iter_max                = 1000;
    algorithm.nlp_tolerance               = 1.e-6;
//    algorithm.hessian   = "exact";
    algorithm.collocation_method          = "Legendre";
//    algorithm.mesh_refinement             = "automatic";




////////////////////////////////////////////////////////////////////////////
//////////////////  Now call PSOPT to solve the problem   /////////////////
////////////////////////////////////////////////////////////////////////////

    psopt(solution, problem, algorithm);

    if (solution.error_flag) exit(0);

////////////////////////////////////////////////////////////////////////////
///////////  Extract relevant variables from solution structure   //////////
////////////////////////////////////////////////////////////////////////////

    MatrixXd x  = solution.get_states_in_phase(1);
    MatrixXd u  = solution.get_controls_in_phase(1);
    MatrixXd t  = solution.get_time_in_phase(1);
    MatrixXd H           = solution.get_dual_hamiltonian_in_phase(1);
    MatrixXd lambda      = solution.get_dual_costates_in_phase(1);

////////////////////////////////////////////////////////////////////////////
///////////  Save solution data to files if desired ////////////////////////
////////////////////////////////////////////////////////////////////////////

    Save(x,"x.dat");
    Save(u,"u.dat");
    Save(t,"t.dat");
    Save(lambda,"p.dat");


////////////////////////////////////////////////////////////////////////////
///////////  Plot some results if desired (requires gnuplot) ///////////////
////////////////////////////////////////////////////////////////////////////

    plot(t,x,problem.name + ": states", "time (s)", "states", "x y v");

    plot(t,u,problem.name + ": control", "time (s)", "control", "u");
```

```
    plot(t,H,problem.name + ": Hamiltonian", "time (s)", "H", "H");

    plot(t,lambda,problem.name + ": costates", "time (s)", "lambda", "lambda_1 lambda_2 lambda_3");

    plot(t,x,problem.name + ": states", "time (s)", "states", "x y v",
                              "pdf", "brac1_states.pdf");

    plot(t,u,problem.name + ": control", "time (s)", "control", "u",
                          "pdf", "brac1_control.pdf");

    plot(t,H,problem.name + ": Hamiltonian", "time (s)", "H", "H",
                              "pdf", "brac1_hamiltonian.pdf");

    plot(t,lambda,problem.name + ": costates", "time (s)", "lambda", "lambda_1 lambda_2 lambda_3",
                              "pdf", "brac1_costates.pdf");



}

////////////////////////////////////////////////////////////////////////////
///////////////////////      END OF FILE     ///////////////////////////////
////////////////////////////////////////////////////////////////////////////
```

The output from $\mathcal{PSOPT}$ is summarized in the box below and shown in Figures 7, 8, which contain the elements of the state, and the control respectively.

```
PSOPT results summary
=====================

Problem:  Brachistochrone Problem
CPU time (seconds): 4.764670e-01
NLP solver used:  IPOPT
PSOPT release number:  5.0.3
Date and time of this run:  Thu Mar  6 16:50:44 2025

Optimal (unscaled) cost function value:  8.247591e-01
Phase 1 endpoint cost function value: 8.247591e-01
Phase 1 integrated part of the cost:  0.000000e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 8.247591e-01
Phase 1 maximum relative local error: 2.773400e-07
NLP solver reports:  The problem has been solved!
```

3 Breakwell problem

Consider the following optimal control problem, which is known in the literature as the Breakwell problem [8]. The problem benefits from having an analytical solution, which

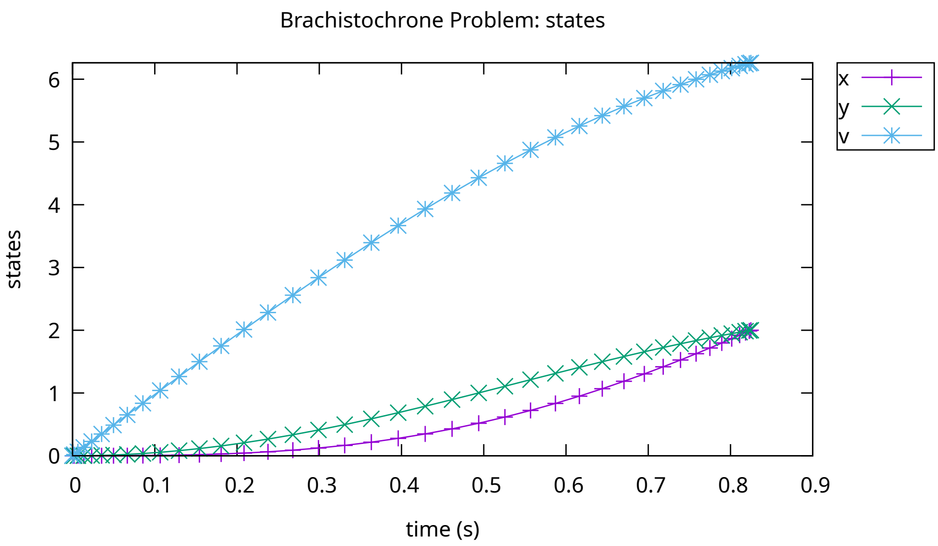

Figure 7: States for brachistochrone problem

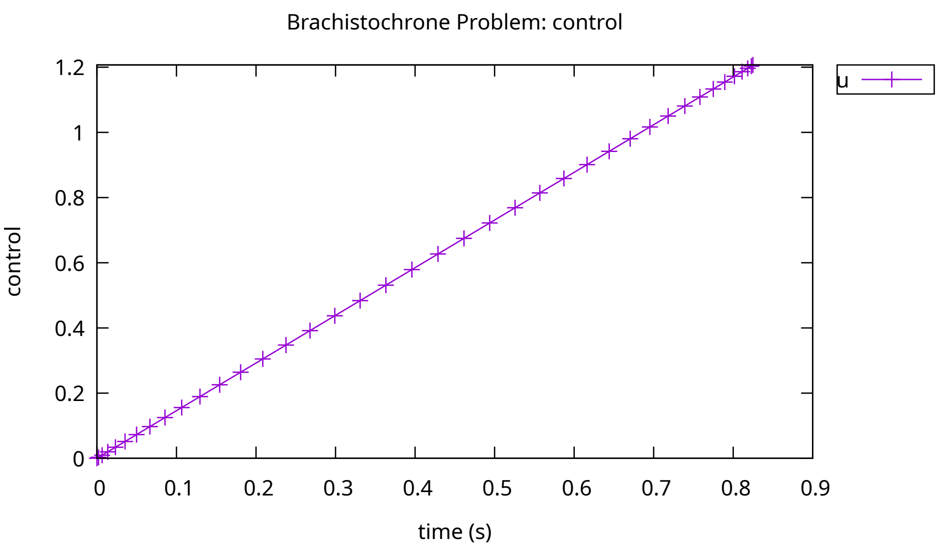

Figure 8: Control for brachistochrone problem

is reported (with some errors) in the book by Bryson and Ho (1975). Minimize the cost functional.

$$J = \int_0^{t_f} u(t)^2 \mathrm{d}t \tag{8}$$

subject to the dynamic constraints

$$\begin{aligned} \dot{x} &= v \\ \dot{v} &= u \end{aligned} \tag{9}$$

the state dependent constraint

$$x(t) \leq l \tag{10}$$

where $l = 0.1$, $t_f = 1.$ and the boundary conditions

$$\begin{aligned} x(0) &= 0 \\ v(0) &= 1 \\ x(t_f) &= 0 \\ v(t_f) &= -1 \end{aligned} \tag{11}$$

The analytical solution of the problem (valid for $0 \leq l \leq 1/6$) is given by:

$$u(t) = \begin{cases} -\frac{2}{3l}(1 - \frac{t}{3l}), & 0 \leq t \leq 3l \\ 0, & 3l \leq t \leq 1 - 3l \\ -\frac{2}{3l}(1 - \frac{1-t}{3l}), & 1 - 3l \leq t \leq 1 \end{cases} \tag{12}$$

$$x(t) = \begin{cases} l\left(1 - \left(1 - \frac{t}{3l}\right)^3\right), & 0 \leq t \leq 3l \\ l, & 3l \leq t \leq 1 - 3l \\ l\left(1 - \left(1 - \frac{1-t}{3l}\right)^3\right), & 1 - 3l \leq t \leq 1 \end{cases} \tag{13}$$

$$v(t) = \begin{cases} \left(1 - \frac{t}{3l}\right)^2, & 0 \leq t \leq 3l \\ 0, & 3l \leq t \leq 1 - 3l \\ \left(1 - \frac{1-t}{3l}\right)^2, & 1 - 3l \leq t \leq 1 \end{cases} \tag{14}$$

$$\lambda_x(t) = \begin{cases} \frac{2}{9l^2}, & 0 \leq t \leq 3l \\ 0, & 3l \leq t \leq 1 - 3l \\ -\frac{2}{9l^2}, & 1 - 3l \leq t \leq 1 \end{cases} \tag{15}$$

$$\lambda_v(t) = \begin{cases} \frac{2}{3l}(1 - \frac{t}{3l}), & 0 \leq t \leq 3l \\ 0, & 3l \leq t \leq 1 - 3l \\ \frac{2}{3l}(1 - \frac{1-t}{3l}), & 1 - 3l \leq t \leq 1 \end{cases} \tag{16}$$

where $\lambda_x(t)$ and $\lambda_v(t)$ are the costates. The analytical optimal value of the objective function is $J = 4/(9l) = 4.4444444$.

The output from $\mathcal{PSOPT}$ is summarized in the following box and shown in Figures 9 and 10, which contain the elements of the state and the control, respectively, and Figure 11 which shows the costates. The figures include curves with the analytical solution for each variable, which is very close to the computed solution.

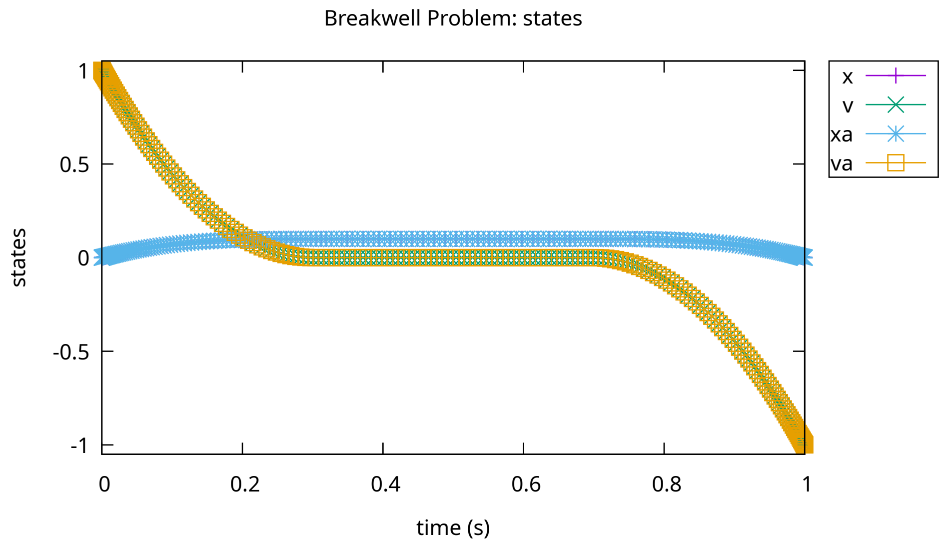

Figure 9: States for Breakwell problem

```
PSOPT results summary
=====================

Problem:  Breakwell Problem
CPU time (seconds): 1.166061e+01
NLP solver used:  IPOPT
PSOPT release number:  5.0.3
Date and time of this run:  Thu Mar  6 16:51:08 2025

Optimal (unscaled) cost function value:  4.444439e+00
Phase 1 endpoint cost function value: 0.000000e+00
Phase 1 integrated part of the cost:  4.444439e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 1.000000e+00
Phase 1 maximum relative local error: 1.487947e-06
NLP solver reports:  The problem has been solved!
```

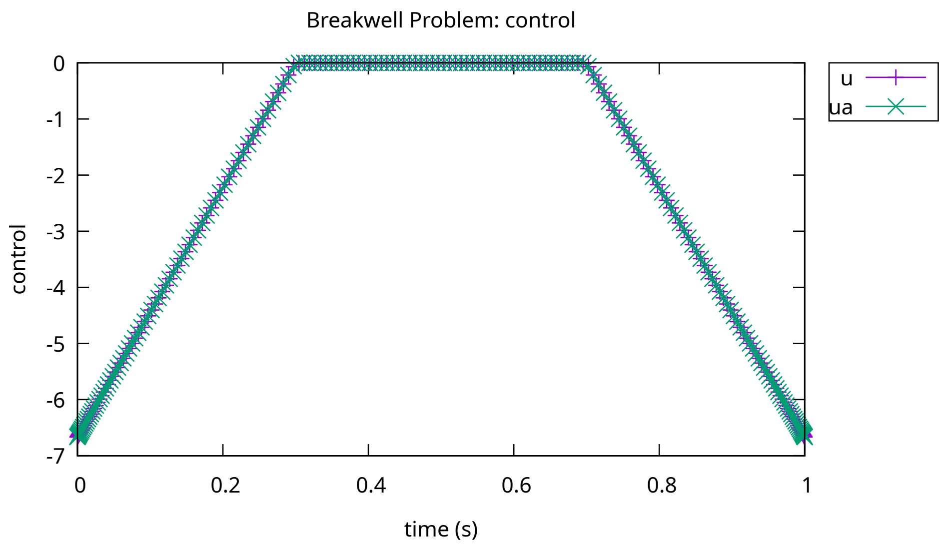

Figure 10: Control for Breakwell problem

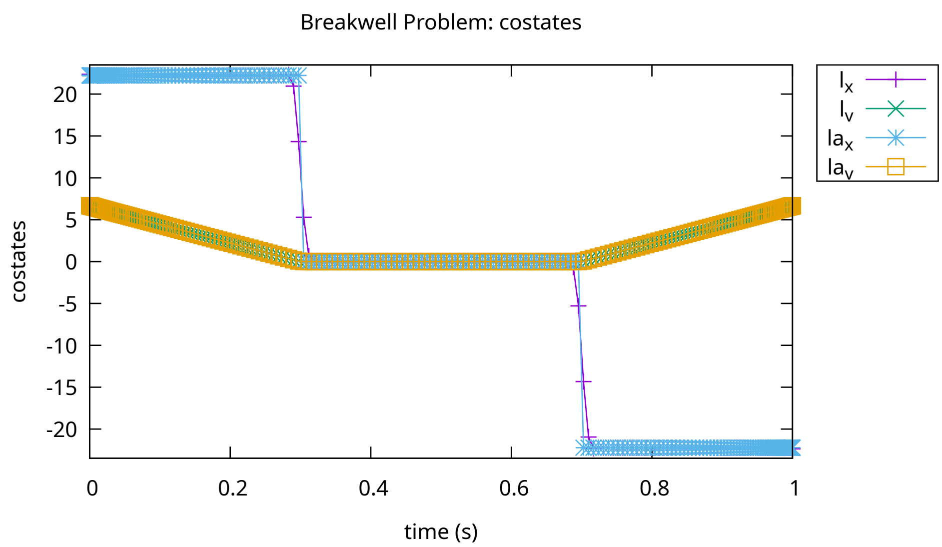

Figure 11: Costates for Breakwell problem

4 Bryson-Denham problem

Consider the following optimal control problem, which is known in the literature as the Bryson-Denham problem [7]. Minimize the cost functional

$$J = x_3(t_f) \tag{17}$$

subject to the dynamic constraints

$$\begin{aligned} \dot{x}_1 &= x_2 \\ \dot{x}_2 &= u \\ \dot{x}_3 &= \tfrac{1}{2}u^2 \end{aligned} \tag{18}$$

the state bound

$$0 \le x_1 \le 1/9 \tag{19}$$

and the boundary conditions

$$\begin{aligned} x_1(0) &= 0 \\ x_2(0) &= 1 \\ x_3(0) &= 0 \\ x_1(t_f) &= 0 \\ x_2(t_f) &= -1 \end{aligned} \tag{20}$$

The output from $\mathcal{PSOPT}$ is summarized in the following box and shown in Figures 12 and 13, which contain the elements of the state and the control, respectively.

```
PSOPT results summary
=====================

Problem:  Bryson-Denham Problem
CPU time (seconds): 6.988060e-01
NLP solver used:  IPOPT
PSOPT release number:  5.0.3
Date and time of this run:  Thu Mar  6 16:51:41 2025

Optimal (unscaled) cost function value:  3.999539e+00
Phase 1 endpoint cost function value: 3.999539e+00
Phase 1 integrated part of the cost:  0.000000e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 6.474053e-01
Phase 1 maximum relative local error: 9.530051e-06
NLP solver reports:  The problem has been solved!
```

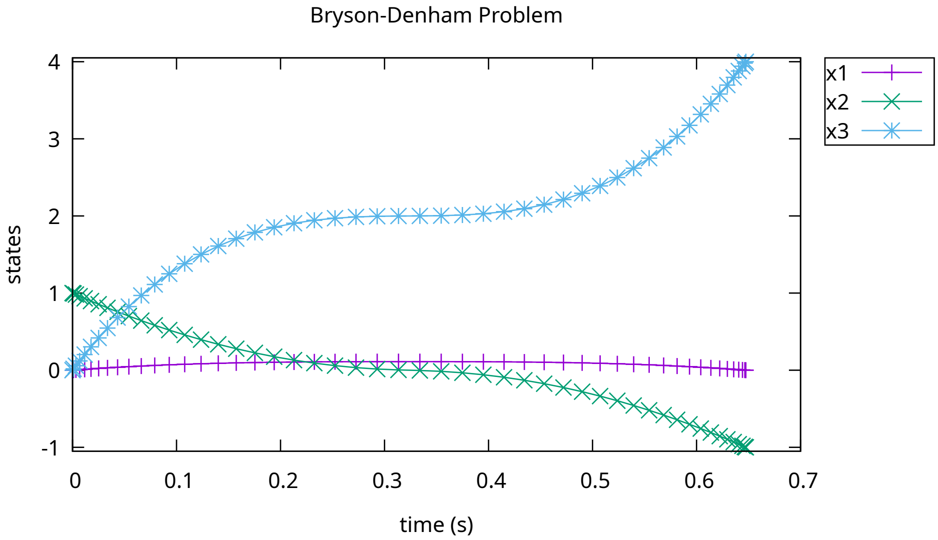

Figure 12: States for Bryson Denham problem

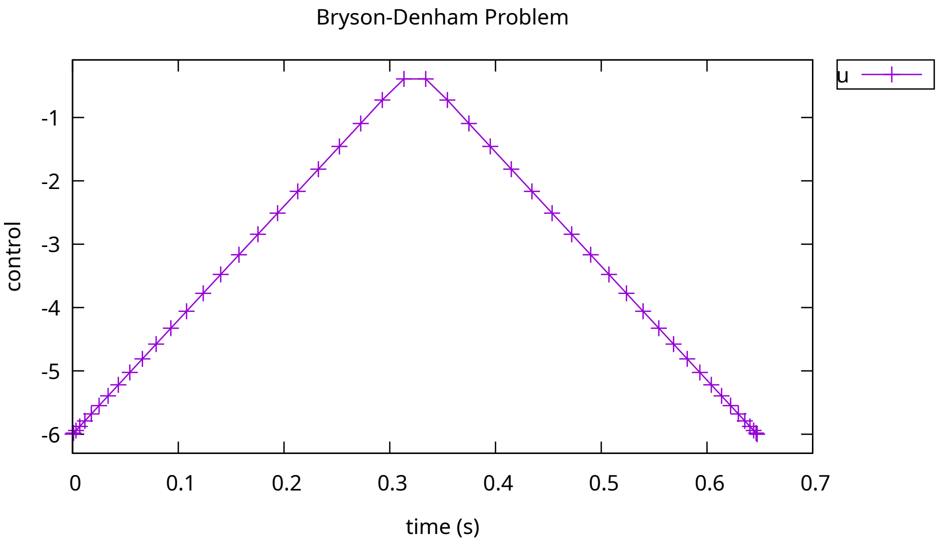

Figure 13: Control for Bryson Denham problem

5 Bryson's maximum range problem

Consider the following optimal control problem, which is known in the literature as the Bryson's maximum range problem [7]. Minimize the cost functional

$$J = -x(t_f) \tag{21}$$

subject to the dynamic constraints

$$\begin{aligned} \dot{x} &= vu_1 \\ \dot{y} &= vu_2 \\ \dot{v} &= a - gu_2 \end{aligned} \tag{22}$$

the path constraint

$$u_1^2 + u_2^2 = 1 \tag{23}$$

and the boundary conditions

$$\begin{aligned} x(0) &= 0 \\ y(0) &= 0 \\ v(0) &= 0 \\ y(t_f) &= 0.1 \end{aligned} \tag{24}$$

where $t_f = 2$, $g = 1$ and $a = 0.5g$. The C++ code that solves this problem is shown below.

```
//////////////////////////////////////////////////////////////////////////
////////////////        bryson_max_range.cxx         /////////////////////
//////////////////////////////////////////////////////////////////////////
////////////////           PSOPT  Example            /////////////////////
//////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////
//////// Title:         Bryson maximum range problem      ////////////////
//////// Last modified: 05 January 2009                   ////////////////
//////// Reference:     Bryson and Ho (1975)              ////////////////
//////// (See PSOPT handbook for full reference)          ////////////////
//////////////////////////////////////////////////////////////////////////
////////     Copyright (c) Victor M. Becerra, 2009        ////////////////
//////////////////////////////////////////////////////////////////////////
//////// This is part of the PSOPT software library, which////////////////
//////// is distributed under the terms of the GNU Lesser ////////////////
//////// General Public License (LGPL)                    ////////////////
//////////////////////////////////////////////////////////////////////////

#include "psopt.h"

//////////////////////////////////////////////////////////////////////////
//////////////////  Define the end point (Mayer) cost function //////////
//////////////////////////////////////////////////////////////////////////

adouble endpoint_cost(adouble* initial_states, adouble* final_states,
                      adouble* parameters,adouble& t0, adouble& tf,
                      adouble* xad, int iphase, Workspace* workspace)
{
   adouble x = final_states[0];

   return (-x);
}

//////////////////////////////////////////////////////////////////////////
//////////////////  Define the integrand (Lagrange) cost function  //////
//////////////////////////////////////////////////////////////////////////

adouble integrand_cost(adouble* states, adouble* controls, adouble* parameters,
                     adouble& time, adouble* xad, int iphase, Workspace* workspace)
```

```
{
    return  0.0;
}


//////////////////////////////////////////////////////////////////////////
///////////////////  Define the DAE's ////////////////////////////////////
//////////////////////////////////////////////////////////////////////////

void dae(adouble* derivatives, adouble* path, adouble* states,
         adouble* controls, adouble* parameters, adouble& time,
         adouble* xad, int iphase, Workspace* workspace)
{
   adouble xdot, ydot, vdot;

   double g = 1.0;
   double a = 0.5*g;

   adouble x = states[ 0 ];
   adouble y = states[ 1 ];
   adouble v = states[ 2 ];

   adouble u1 = controls[ 0 ];
   adouble u2 = controls[ 1 ];

   xdot = v*u1;
   ydot = v*u2;
   vdot = a-g*u2;

   derivatives[ 0 ] = xdot;
   derivatives[ 1 ] = ydot;
   derivatives[ 2 ] = vdot;

   path[ 0 ] = (u1*u1) + (u2*u2);

}

////////////////////////////////////////////////////////////////////////////
//////////////////  Define the events function /////////////////////////////
////////////////////////////////////////////////////////////////////////////

void events(adouble* e, adouble* initial_states, adouble* final_states,
            adouble* parameters,adouble& t0, adouble& tf, adouble* xad,
            int iphase, Workspace* workspace)

{
   adouble x0 = initial_states[ 0 ];
   adouble y0 = initial_states[ 1 ];
   adouble v0 = initial_states[ 2 ];
   adouble xf = final_states[ 0 ];
   adouble yf = final_states[ 1 ];

   e[ 0 ] = x0;
   e[ 1 ] = y0;
   e[ 2 ] = v0;
   e[ 3 ] = yf;

}



///////////////////////////////////////////////////////////////////////////
///////////////////  Define the phase linkages function ///////////////////
///////////////////////////////////////////////////////////////////////////

void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{
  // No linkages as this is a single phase problem
}



////////////////////////////////////////////////////////////////////////////
//////////////////  Define the main routine ////////////////////////////////
////////////////////////////////////////////////////////////////////////////

int main(void)
{

////////////////////////////////////////////////////////////////////////////
//////////////////  Declare key structures /////////////////////////////////
////////////////////////////////////////////////////////////////////////////
```

```
    Alg  algorithm;
    Sol  solution;
    Prob problem;

////////////////////////////////////////////////////////////////////////////
//////////////////  Register problem name  /////////////////////////////////
////////////////////////////////////////////////////////////////////////////

    problem.name        = "Bryson Maximum Range Problem";
    problem.outfilename                = "brymr.txt";

////////////////////////////////////////////////////////////////////////////
///////////  Define problem level constants & do level 1 setup ////////////
////////////////////////////////////////////////////////////////////////////

    problem.nphases   = 1;
    problem.nlinkages                  = 0;

    psopt_level1_setup(problem);

/////////////////////////////////////////////////////////////////////////////
/////////   Define phase related information & do level 2 setup  ////////////
/////////////////////////////////////////////////////////////////////////////

    problem.phases(1).nstates    = 3;
    problem.phases(1).ncontrols  = 2;
    problem.phases(1).nevents    = 4;
    problem.phases(1).npath      = 1;
    problem.phases(1).nodes         << 50;

    psopt_level2_setup(problem, algorithm);

////////////////////////////////////////////////////////////////////////////
//////////////////  Declare MatrixXd objects to store results //////////////
////////////////////////////////////////////////////////////////////////////

    MatrixXd x, u, t;
    MatrixXd lambda, H;

////////////////////////////////////////////////////////////////////////////
//////////////////  Enter problem bounds information ///////////////////////
////////////////////////////////////////////////////////////////////////////

    double xL = -10.0;
    double yL = -10.0;
    double vL = -10.0;
    double xU = 10.0;
    double yU = 10.0;
    double vU = 10.0;

    double u1L = -10.0;
    double u2L = -10.0;
    double u1U = 10.0;
    double u2U = 10.0;

    double x0 = 0.0;
    double y0 = 0.0;
    double v0 = 0.0;
    double yf = 0.1;


    problem.phases(1).bounds.lower.states(0) = xL;
    problem.phases(1).bounds.lower.states(1) = yL;
    problem.phases(1).bounds.lower.states(2) = vL;


    problem.phases(1).bounds.upper.states(0) = xU;
    problem.phases(1).bounds.upper.states(1) = yU;
    problem.phases(1).bounds.upper.states(2) = vU;


    problem.phases(1).bounds.lower.controls(0) = u1L;
    problem.phases(1).bounds.lower.controls(1) = u2L;
    problem.phases(1).bounds.upper.controls(0) = u1U;
    problem.phases(1).bounds.upper.controls(1) = u2U;

    problem.phases(1).bounds.lower.events(0) = x0;
    problem.phases(1).bounds.lower.events(1) = y0;
    problem.phases(1).bounds.lower.events(2) = v0;
    problem.phases(1).bounds.lower.events(3) = yf;
```

```
    problem.phases(1).bounds.upper.events(0) = x0;
    problem.phases(1).bounds.upper.events(1) = y0;
    problem.phases(1).bounds.upper.events(2) = v0;
    problem.phases(1).bounds.upper.events(3) = yf;

    problem.phases(1).bounds.upper.path(0) = 1.0;
    problem.phases(1).bounds.lower.path(0) = 1.0;



    problem.phases(1).bounds.lower.StartTime    = 0.0;
    problem.phases(1).bounds.upper.StartTime    = 0.0;

    problem.phases(1).bounds.lower.EndTime      = 2.0;
    problem.phases(1).bounds.upper.EndTime      = 2.0;



////////////////////////////////////////////////////////////////////////////
///////////////////  Register problem functions  ///////////////////////////
////////////////////////////////////////////////////////////////////////////


    problem.integrand_cost  = &integrand_cost;
    problem.endpoint_cost  = &endpoint_cost;
    problem.dae             = &dae;
    problem.events  = &events;
    problem.linkages = &linkages;

////////////////////////////////////////////////////////////////////////////
///////////////////  Define & register initial guess ///////////////////////
////////////////////////////////////////////////////////////////////////////

    int nnodes                  = problem.phases(1).nodes(0);
    int ncontrols                       = problem.phases(1).ncontrols;
    int nstates                         = problem.phases(1).nstates;

    MatrixXd x_guess    =  zeros(nstates,nnodes);

    x_guess.row(0)  = x0*ones(1,nnodes);
    x_guess.row(1)  = y0*ones(1,nnodes);
    x_guess.row(2)  = v0*ones(1,nnodes);

    problem.phases(1).guess.controls       = zeros(ncontrols,nnodes);
    problem.phases(1).guess.states         = x_guess;
    problem.phases(1).guess.time           = linspace(0.0,2.0,nnodes);


////////////////////////////////////////////////////////////////////////////
///////////////////  Enter algorithm options  //////////////////////////////
////////////////////////////////////////////////////////////////////////////


    algorithm.nlp_iter_max                = 1000;
    algorithm.nlp_tolerance               = 1.e-4;
    algorithm.nlp_method                  = "IPOPT";
    algorithm.scaling                     = "automatic";
    algorithm.derivatives                 = "automatic";
//    algorithm.mesh_refinement             = "automatic";
    algorithm.collocation_method          = "trapezoidal";
//    algorithm.defect_scaling = "jacobian-based";
    algorithm.ode_tolerance               = 1.e-6;



////////////////////////////////////////////////////////////////////////////
///////////////////  Now call PSOPT to solve the problem   /////////////////
////////////////////////////////////////////////////////////////////////////

    psopt(solution, problem, algorithm);

////////////////////////////////////////////////////////////////////////////
///////////  Extract relevant variables from solution structure   //////////
////////////////////////////////////////////////////////////////////////////


    x      = solution.get_states_in_phase(1);
    u      = solution.get_controls_in_phase(1);
    t      = solution.get_time_in_phase(1);
    lambda = solution.get_dual_costates_in_phase(1);
    H      = solution.get_dual_hamiltonian_in_phase(1);
```

```
////////////////////////////////////////////////////////////////////////////
///////////  Save solution data to files if desired ////////////////////////
////////////////////////////////////////////////////////////////////////////

    Save(x, "x.dat");
    Save(u,"u.dat");
    Save(t,"t.dat");
    Save(lambda,"lambda.dat");
    Save(H,"H.dat");

////////////////////////////////////////////////////////////////////////////
///////////  Plot some results if desired (requires gnuplot) ///////////////
////////////////////////////////////////////////////////////////////////////

    plot(t,x,problem.name+": states", "time (s)", "states","x y v");

    plot(t,u,problem.name+": controls","time (s)", "controls", "u_1 u_2");

    plot(t,x,problem.name+": states", "time (s)", "states","x y v",
                             "pdf", "brymr_states.pdf");

    plot(t,u,problem.name+": controls","time (s)", "controls", "u_1 u_2",
                             "pdf", "brymr_controls.pdf");
}

////////////////////////////////////////////////////////////////////////////
//////////////////////      END OF FILE     ////////////////////////////////
////////////////////////////////////////////////////////////////////////////
```

The output from $\mathcal{PSOPT}$ is summarized in the box below and shown in Figures 14 and 15, which contain the elements of the state and the control, respectively.

```
PSOPT results summary
=====================

Problem:  Bryson Maximum Range Problem
CPU time (seconds): 2.326190e-01
NLP solver used:  IPOPT
PSOPT release number:  5.0.3
Date and time of this run:  Thu Mar  6 16:51:58 2025

Optimal (unscaled) cost function value:  -1.712313e+00
Phase 1 endpoint cost function value: -1.712313e+00
Phase 1 integrated part of the cost:  0.000000e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 2.000000e+00
Phase 1 maximum relative local error: 1.312361e-04
NLP solver reports:  The problem has been solved!
```

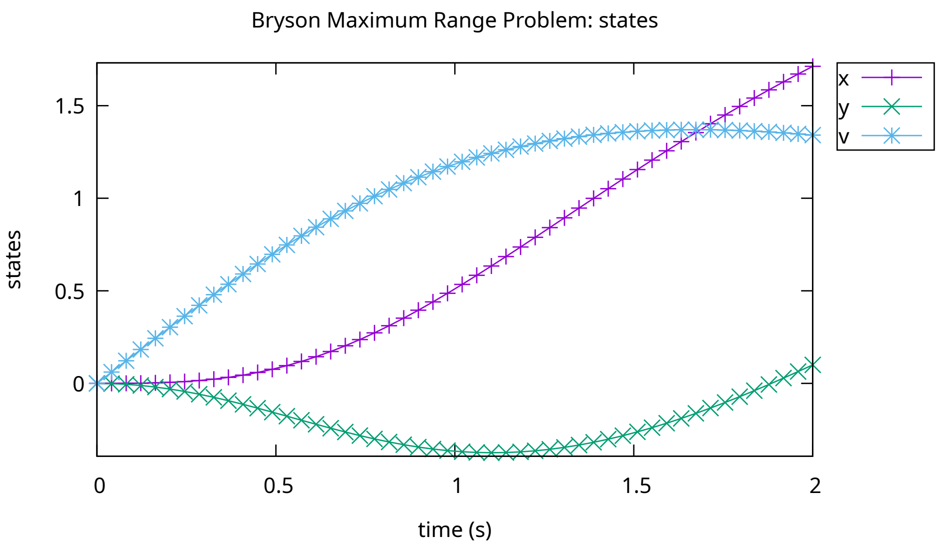

Figure 14: States for Bryson's maximum range problem

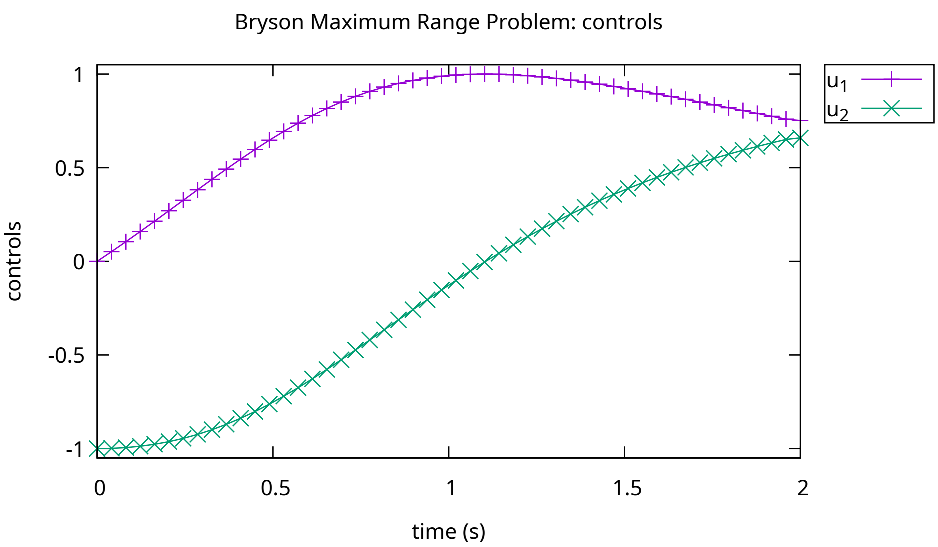

Figure 15: Controls for Bryson's maximum range problem

6 Catalyst mixing problem

Consider the following optimal control problem, which attempts to determine the optimal mixing policy of two catalysts along the length of a tubular plug flow reactor involving several reactions [20]. The catalyst mixing problem is a typical bang-singular-bang problem. Minimize the cost functional

$$J = -1 + x_1(t_f) + x_2(t_f) \tag{25}$$

subject to the dynamic constraints

$$\begin{aligned} \dot{x}_1 &= u(10x_2 - x_1) \\ \dot{x}_2 &= u(x_1 - 10x_2) - (1-u)x_2 \end{aligned} \tag{26}$$

the boundary conditions

$$\begin{aligned} x_1(0) &= 1 \\ x_2(0) &= 0 \\ x_1(t_f) &\leq 0.95 \end{aligned} \tag{27}$$

and the box constraints:

$$\begin{aligned} 0.9 &\leq x_1(t) \leq 1.0 \\ 0 &\leq x_2(t) \leq 0.1 \\ 0 &\leq u(t) \leq 1 \end{aligned} \tag{28}$$

where $t_f = 1$. The C++ code that solves this problem is shown below.

The output from $\mathcal{PSOPT}$ is summarised in the box below and shown in Figures 16 and 17, which contain the elements of the state and the control, respectively.

```
PSOPT results summary
=====================

Problem:  Catalyst mixing roblem
CPU time (seconds): 3.190900e-01
NLP solver used:  IPOPT
PSOPT release number:  5.0.3
Date and time of this run:  Thu Mar  6 16:52:11 2025

Optimal (unscaled) cost function value:  -4.805320e-02
Phase 1 endpoint cost function value: -4.805320e-02
Phase 1 integrated part of the cost:  0.000000e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 1.000000e+00
Phase 1 maximum relative local error: 1.092791e-04
NLP solver reports:  The problem has been solved!
```

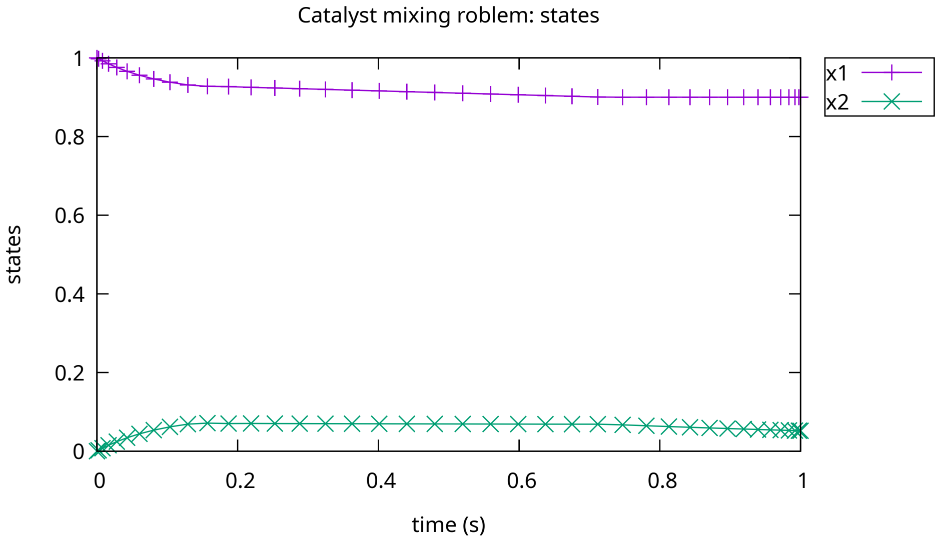

Figure 16: States for catalyist mixing problem

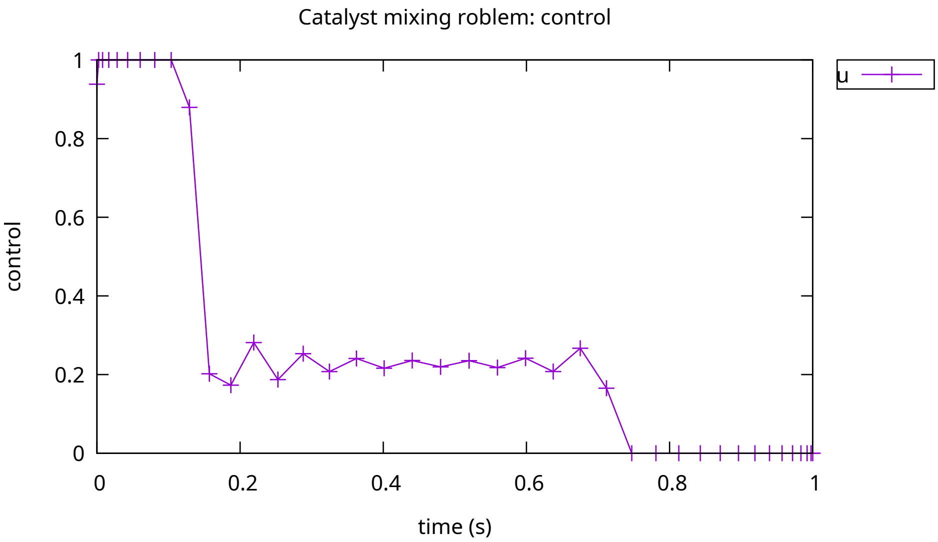

Figure 17: Control for catalyst mixing problem

7 Catalytic cracking of gas oil

Consider the following optimization problem, which involves finding optimal static parameters subject to dynamic constraints [9]. Minimize

$$J = \sum_{i=1}^{21} (y_1(t_i) - y_{m,1}(i))^2 + (y_2(t_i) - y_{m,2}(i))^2 \tag{29}$$

subject to the dynamic constraints

$$\begin{aligned} \dot{y}_1 &= -(\theta_1 + \theta_3) y_1^2 \\ \dot{y}_2 &= \theta_1 y_1^2 - \theta_2 y_2 \end{aligned} \tag{30}$$

the parameter constraint

$$\begin{aligned} &\theta_1 \geq 0 \\ &\theta_2 \geq 0 \\ &\theta_3 \geq 0 \end{aligned} \tag{31}$$

Note that, given the nature of the problem, the parameter estimation facilities of $\mathcal{PSOPT}$ are used in this example. In this case, the observations function is simple:

$$g(x(t), u(t), p, t) = [y_1 \;\; y_2]^T$$

The $\mathcal{PSOPT}$ code that solves this problem is shown below. The code includes the values of the measurement vectors $y_{m,1}$, and $y_{m,2}$, as well as the vector of sampling instants $\theta_i, i = 1, \ldots, 21$.

```
//////////////////////////////////////////////////////////////////////////
//////////////////        cracking.cxx      ///////////////////////////
//////////////////////////////////////////////////////////////////////////
////////////////          PSOPT  Example           ////////////////////
//////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////
//////// Title:  Catalytic Cracking of Gas Oil            ////////////////
//////// Last modified: 15 January 2009                   ////////////////
//////// Reference:     User's guide for DIRCOL     ////////////////
//////// (See PSOPT handbook for full reference)           ///////////////
//////////////////////////////////////////////////////////////////////////
////////     Copyright (c) Victor M. Becerra, 2009        ////////////////
//////////////////////////////////////////////////////////////////////////
//////// This is part of the PSOPT software library, which ///////////////
//////// is distributed under the terms of the GNU Lesser ////////////////
//////// General Public License (LGPL)                    ////////////////
//////////////////////////////////////////////////////////////////////////

#include "psopt.h"

//////////////////////////////////////////////////////////////////////////
//////////////////  Define the observation function //////////
//////////////////////////////////////////////////////////////////////////



void  observation_function( adouble* observations,
                            adouble* states, adouble* controls,
                            adouble* parameters, adouble& time, int k,
                            adouble* xad, int iphase, Workspace* workspace)
{

      observations[ 0 ] = states[ 0 ];
```

```
      observations[ 1 ] = states[ 1 ];
}



//////////////////////////////////////////////////////////////////////////
//////////////////  Define the DAE's /////////////////////////////////////
//////////////////////////////////////////////////////////////////////////

void dae(adouble* derivatives, adouble* path, adouble* states,
         adouble* controls, adouble* parameters, adouble& time,
         adouble* xad, int iphase, Workspace* workspace)
{

   adouble y1 = states[0];
   adouble y2 = states[1];

   adouble theta1 = parameters[ 0 ];
   adouble theta2 = parameters[ 1 ];
   adouble theta3 = parameters[ 2 ];

   derivatives[0] = -(theta1 + theta3)*y1*y1;
   derivatives[1] =  theta1*y1*y1 - theta2*y2;

}

////////////////////////////////////////////////////////////////////////////
//////////////////  Define the events function /////////////////////////////
////////////////////////////////////////////////////////////////////////////

void events(adouble* e, adouble* initial_states, adouble* final_states,
            adouble* parameters,adouble& t0, adouble& tf, adouble* xad,
            int iphase, Workspace* workspace)
{
   // No events

}


///////////////////////////////////////////////////////////////////////////
//////////////////  Define the phase linkages function ////////////////////
///////////////////////////////////////////////////////////////////////////

void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{
  // No linkages as this is a single phase problem
}


////////////////////////////////////////////////////////////////////////////
//////////////////  Define the main routine ////////////////////////////////
////////////////////////////////////////////////////////////////////////////

int main(void)
{

   MatrixXd y1meas(1,21), y2meas(1,21), tmeas(1,21);
   // Measured values of y1
   y1meas <<     1.0,0.8105,0.6208,0.5258,0.4345,0.3903,0.3342,0.3034, \
                  0.2735,0.2405,0.2283,0.2071,0.1669,0.153,0.1339,0.1265, \
                  0.12,0.099,0.087,0.077,0.069;
   // Measured values of y2
   y2meas <<  0.0,0.2,0.2886,0.301,0.3215,0.3123,0.2716,0.2551,0.2258, \
                 0.1959,0.1789,0.1457,0.1198,0.0909,0.0719,0.0561,0.046, \
                0.028,0.019,0.014,0.01;
   // Sampling instants
   tmeas  <<  0.0,0.025,0.05,0.075,0.1,0.125,0.15,0.175,0.2,0.225,0.25, \
              0.3,0.35,0.4,0.45,0.5,0.55,0.65,0.75,0.85,0.95;



////////////////////////////////////////////////////////////////////////////
//////////////////  Declare key structures /////////////////////////////////
////////////////////////////////////////////////////////////////////////////

    Alg  algorithm;
    Sol  solution;
    Prob problem;

////////////////////////////////////////////////////////////////////////////
//////////////////  Register problem name  /////////////////////////////////
////////////////////////////////////////////////////////////////////////////
```

```
    problem.name           = "Catalytic cracking of gas oil";
    problem.outfilename                 = "cracking.txt";

////////////////////////////////////////////////////////////////////////////
////////////  Define problem level constants & do level 1 setup ////////////
////////////////////////////////////////////////////////////////////////////

    problem.nphases    = 1;
    problem.nlinkages                   = 0;

    psopt_level1_setup(problem);


/////////////////////////////////////////////////////////////////////////////
/////////   Define phase related information & do level 2 setup ////////////
/////////////////////////////////////////////////////////////////////////////

    problem.phases(1).nstates    = 2;
    problem.phases(1).ncontrols  = 0;
    problem.phases(1).nevents    = 0;
    problem.phases(1).npath      = 0;
    problem.phases(1).nparameters         = 3;
    problem.phases(1).nodes          << 80;
    problem.phases(1).nobserved   = 2;
    problem.phases(1).nsamples    = 21;

    psopt_level2_setup(problem, algorithm);

////////////////////////////////////////////////////////////////////////////
////////////  Enter estimation information                      ////////////
////////////////////////////////////////////////////////////////////////////

    MatrixXd observations(2, 21);

    observations << y1meas, y2meas;

    problem.phases(1).observation_nodes      = tmeas;
    problem.phases(1).observations           = observations;
    problem.phases(1).residual_weights       = ones(2,21);


////////////////////////////////////////////////////////////////////////////
///////////////////  Declare DMatrix objects to store results //////////////
////////////////////////////////////////////////////////////////////////////

    DMatrix x, p, t;

////////////////////////////////////////////////////////////////////////////
///////////////////  Enter problem bounds information //////////////////////
////////////////////////////////////////////////////////////////////////////


    problem.phases(1).bounds.lower.states(0) =  0.0;
    problem.phases(1).bounds.lower.states(1) =  0.0;


    problem.phases(1).bounds.upper.states(0) =  2.0;
    problem.phases(1).bounds.upper.states(1) =  2.0;


    problem.phases(1).bounds.lower.parameters(0) = 0.0;
    problem.phases(1).bounds.lower.parameters(1) = 0.0;
    problem.phases(1).bounds.lower.parameters(2) = 0.0;
    problem.phases(1).bounds.upper.parameters(0) = 20.0;
    problem.phases(1).bounds.upper.parameters(1) = 20.0;
    problem.phases(1).bounds.upper.parameters(2) = 20.0;


    problem.phases(1).bounds.lower.StartTime    = 0.0;
    problem.phases(1).bounds.upper.StartTime    = 0.0;

    problem.phases(1).bounds.lower.EndTime      = 0.95;
    problem.phases(1).bounds.upper.EndTime      = 0.95;

////////////////////////////////////////////////////////////////////////////
///////////////////  Register problem functions  ///////////////////////////
////////////////////////////////////////////////////////////////////////////

    problem.dae  = &dae;
    problem.events  = &events;
    problem.linkages = &linkages;
    problem.observation_function = & observation_function;
```

```
////////////////////////////////////////////////////////////////////////////
///////////////////  Define & register initial guess ///////////////////////
////////////////////////////////////////////////////////////////////////////

    MatrixXd state_guess(2, 40);

    state_guess.row(0) =  linspace(1.0,0.069, 40);
    state_guess.row(1) =  linspace(0.30,0.01,  40);


    problem.phases(1).guess.states         = state_guess;
    problem.phases(1).guess.time           = linspace(0.0, 0.95, 40);
    problem.phases(1).guess.parameters     = zeros(3,1);


////////////////////////////////////////////////////////////////////////////
///////////////////  Enter algorithm options  //////////////////////////////
////////////////////////////////////////////////////////////////////////////

    algorithm.nlp_method                  = "IPOPT";
    algorithm.scaling                     = "automatic";
    algorithm.derivatives                 = "automatic";
    algorithm.collocation_method          = "Hermite-Simpson";
    algorithm.nlp_iter_max                = 1000;
    algorithm.nlp_tolerance               = 1.e-6;
//    algorithm.jac_sparsity_ratio          = 0.52;

////////////////////////////////////////////////////////////////////////////
///////////////////  Now call PSOPT to solve the problem   /////////////////
////////////////////////////////////////////////////////////////////////////

    psopt(solution, problem, algorithm);

////////////////////////////////////////////////////////////////////////////
///////////  Extract relevant variables from solution structure   //////////
////////////////////////////////////////////////////////////////////////////

    x = solution.get_states_in_phase(1);
    t = solution.get_time_in_phase(1);
    p = solution.get_parameters_in_phase(1);


////////////////////////////////////////////////////////////////////////////
///////////  Save solution data to files if desired ////////////////////////
////////////////////////////////////////////////////////////////////////////

    Save(x,"x.dat");
    Save(t,"t.dat");
    cout << "\n Estimated parameters\n" << p << endl;
//    Print(p,"Estimated parameters");


////////////////////////////////////////////////////////////////////////////
///////////  Plot some results if desired (requires gnuplot) ///////////////
////////////////////////////////////////////////////////////////////////////

    plot(t,x,problem.name, "time (s)", "states", "y1 y2");
    plot(t,x,problem.name, "time (s)", "states", "y1 y2",
                           "pdf", "cracking_states.pdf");

}

////////////////////////////////////////////////////////////////////////////
///////////////////////      END OF FILE     ///////////////////////////////
////////////////////////////////////////////////////////////////////////////
```

The output from $\mathcal{PSOPT}$ is summarized in the box below and shown in Figure 18, which shows the states of the system. The optimal parameters found were:

$$\begin{aligned} \theta_1 &= 11.40825702 \\ \theta_2 &= 8.123367918 \\ \theta_3 &= 1.668727477 \end{aligned} \tag{32}$$

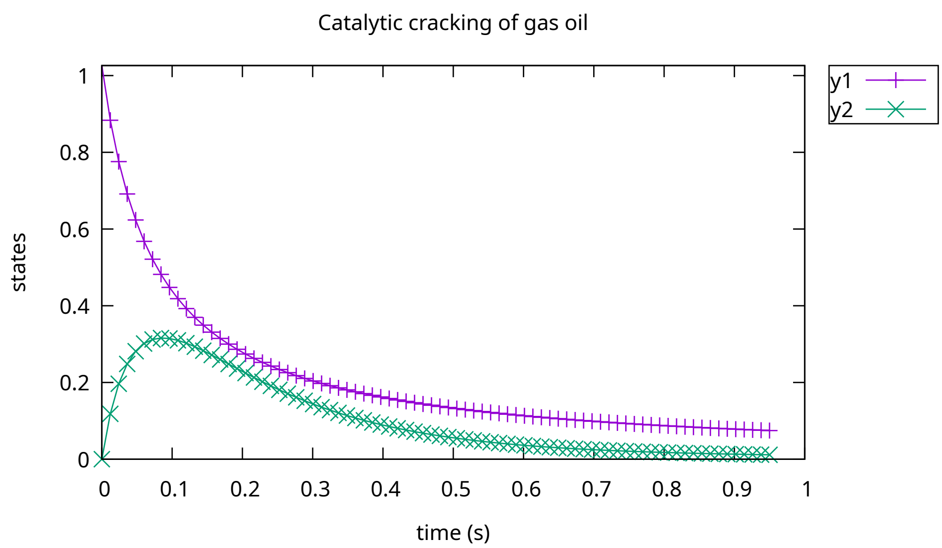

Figure 18: States for catalytic cracking of gas oil problem

```
PSOPT results summary
=====================

Problem:  Catalytic cracking of gas oil
CPU time (seconds): 2.712220e-01
NLP solver used:   IPOPT
PSOPT release number:  5.0.3
Date and time of this run:  Thu Mar  6 16:54:57 2025

Optimal (unscaled) cost function value:  4.319519e-03
Phase 1 endpoint cost function value: 4.319519e-03
Phase 1 integrated part of the cost:  0.000000e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 9.500000e-01
Phase 1 maximum relative local error: 4.414787e-04
NLP solver reports:  The problem has been solved!
```

8 Coulomb friction

Consider the following optimal control problem, which consists of a system that exhibits Coulomb friction [14]. Minimize the cost:

$$J = t_f \tag{33}$$

subject to the dynamic constraints

$$\begin{array}{rcl} \ddot{q}_1 & = & (-(k_1 - k_2)q_1 + k_2 q_2 - \mu \text{sign}(\dot{q}_1) + u_1)/m_1 \\ \ddot{q}_2 & = & (k_2 q_1 - k_2 q_2 - \mu \text{sign}(\dot{q}_2) + u_2)/m_2 \end{array} \tag{34}$$

and the boundary conditions

$$\begin{array}{rcl} q_1(0) & = & 0 \\ \dot{q}_1(0) & = & -1 \\ q_2(0) & = & 0 \\ \dot{q}_2(0) & = & -2 \\ q_1(t_f) & = & 1 \\ \dot{q}_1(t_f) & = & 0 \\ q_2(t_f) & = & 2 \\ \dot{q}_2(t_f) = 0 & & \end{array} \tag{35}$$

where $k_1 = 0.95$, k_2=0.85, $\mu = 1.0$, m_1=1.1, m_2=1.2.

The output from $\mathcal{PSOPT}$ summarised in the box below and shown in Figures 19 and 20, which contain the elements of the state and the control, respectively.

```
PSOPT results summary
=====================

Problem:  Coulomb friction problem
CPU time (seconds): 6.261420e-01
NLP solver used:  IPOPT
PSOPT release number:  5.0.3
Date and time of this run:  Thu Mar  6 16:54:41 2025

Optimal (unscaled) cost function value:  2.104992e+00
Phase 1 endpoint cost function value: 2.104992e+00
Phase 1 integrated part of the cost:  0.000000e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 2.104992e+00
Phase 1 maximum relative local error: 7.858415e-03
NLP solver reports:  The problem has been solved!
```

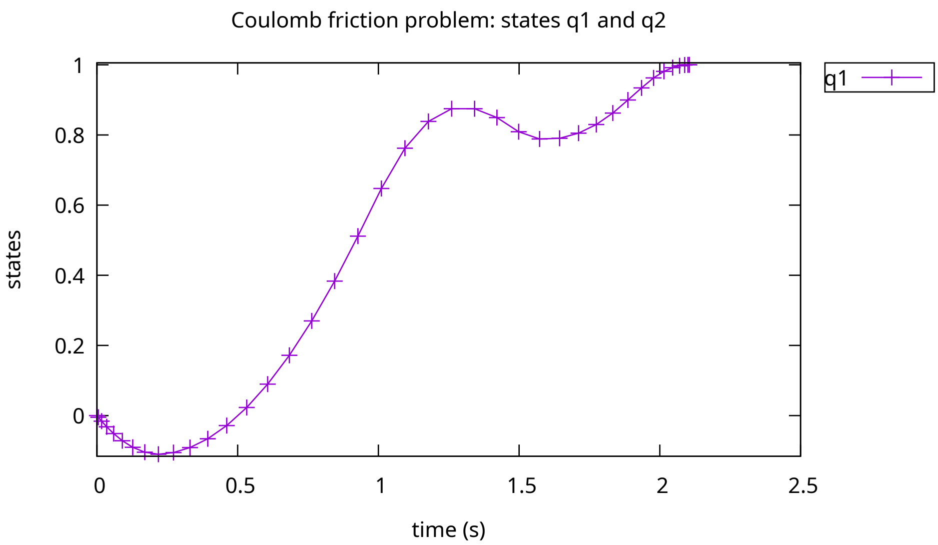

Figure 19: States for Coulomb friction problem

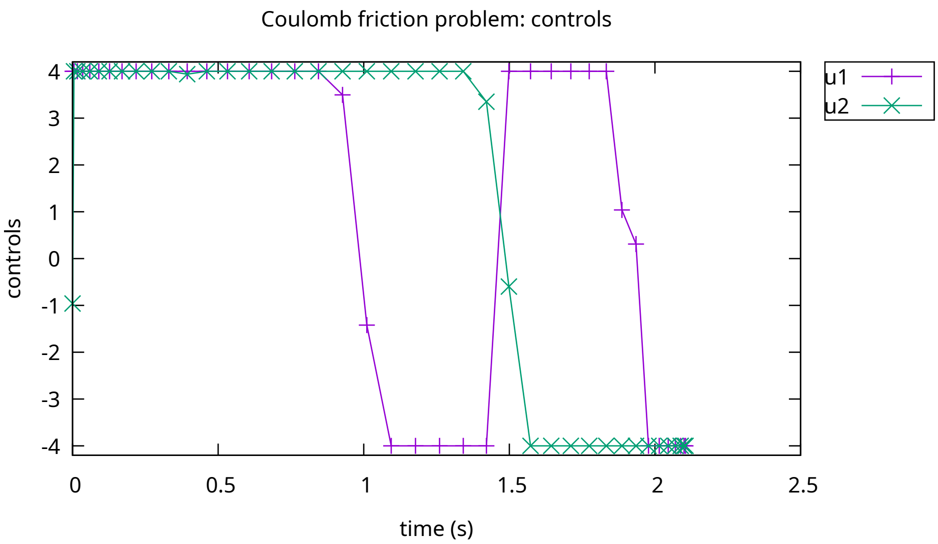

Figure 20: Controls for Coulomb friction problem

9 DAE index 3 parameter estimation problem

Consider the following parameter estimation problem, which involves a differential-algebraic equation of index 3 with four differential states and one algebraic state [19].

The dynamics consists of the differential equations

$$\begin{aligned}\dot{x}_1(t) &= x_3(t)\\ \dot{x}_2(t) &= x_4(t)\\ \dot{x}_3(t) &= \lambda(t)x_1(t)\\ \dot{x}_4(t) &= \lambda(t)x_2(t)\end{aligned} \tag{36}$$

and the algebraic equation

$$0 = L^2 - x_1(t)^2 - x_2(t)^2 \tag{37}$$

where $x_j(t), j = 1, \ldots, 4$ are the differential states, $\lambda(t)$ is an algebraic state (note that algebraic states are treated as control variables), and L is a parameter to be estimated.

The observations function is given by:

$$\begin{aligned}y_1 &= x_1\\ y_2 &= x_2\end{aligned} \tag{38}$$

And the following least squares objective is minimised:

$$J = \sum_{k=1}^{n_s} \left[(y_1(t_k) - \hat{y}_1(t_k))^2 + (y_2(t_k) - \hat{y}_2(t_k))^2 \right] \tag{39}$$

where $n_s = 20$, $t_1 = 0.5$ and $t_{20} = 10.0$.

The C++ code that solves this problem is shown below.

```
//////////////////////////////////////////////////////////////////////////
//////////////////        dae_i3.cxx      ////////////////////////////
//////////////////////////////////////////////////////////////////////////
////////////////           PSOPT  Example           ////////////////////
//////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////
//////// Title:  DAE Index 3                          ////////////////
//////// Last modified:  07 June 2011                ////////////////
//////// Reference:     Schittkowski (2002)          ////////////////
//////// (See PSOPT handbook forf full reference)           ///////////////
//////////////////////////////////////////////////////////////////////////
////////     Copyright (c) Victor M. Becerra, 2011       ////////////////
//////////////////////////////////////////////////////////////////////////
//////// This is part of the PSOPT software library, which ///////////////
//////// is distributed under the terms of the GNU Lesser ////////////////
//////// General Public License (LGPL)                    ////////////////
//////////////////////////////////////////////////////////////////////////

#include "psopt.h"

//////////////////////////////////////////////////////////////////////////
//////////////////  Define the observation function //////////
//////////////////////////////////////////////////////////////////////////
```

```
void  observation_function( adouble* observations,
                            adouble* states, adouble* controls,
                            adouble* parameters, adouble& time, int k,
                            adouble* xad, int iphase, Workspace* workspace)
{
      observations[ 0 ] = states[ 0 ];
      observations[ 1 ] = states[ 1 ];

}



//////////////////////////////////////////////////////////////////////////
//////////////////  Define the DAE's ////////////////////////////////////
//////////////////////////////////////////////////////////////////////////

void dae(adouble* derivatives, adouble* path, adouble* states,
         adouble* controls, adouble* parameters, adouble& time,
         adouble* xad, int iphase, Workspace* workspace)
{

    // Variables
       adouble x1, x2, x3, x4, L, OMEGA, LAMBDA;
       adouble dx1, dx2, dx3, dx4;

    // Differential states
       x1 = states[0];
       x2 = states[1];
       x3 = states[2];
       x4 = states[3];

    // Algebraic variables
       LAMBDA = controls[0];


    // Parameters
       L     = parameters[0];
    // Differential equations

      dx1 = x3;

      dx2 = x4;

      dx3 = LAMBDA*x1;

      dx4 = LAMBDA*x2;

      derivatives[ 0 ] = dx1;
      derivatives[ 1 ] = dx2;
      derivatives[ 2 ] = dx3;
      derivatives[ 3 ] = dx4;


     // algebraic equation

      path[ 0 ] = L*L - x1*x1 - x2*x2;

}

////////////////////////////////////////////////////////////////////////////
//////////////////  Define the events function ////////////////////////////
////////////////////////////////////////////////////////////////////////////

void events(adouble* e, adouble* initial_states, adouble* final_states,
            adouble* parameters,adouble& t0, adouble& tf, adouble* xad,
            int iphase, Workspace* workspace)
{
       // no events

       return;

}


///////////////////////////////////////////////////////////////////////////
//////////////////  Define the phase linkages function ///////////////////
///////////////////////////////////////////////////////////////////////////

void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{
  // No linkages as this is a single phase problem
}
```

```
////////////////////////////////////////////////////////////////////////////
//////////////////  Define the main routine ///////////////////////////////
////////////////////////////////////////////////////////////////////////////

int main(void)
{

////////////////////////////////////////////////////////////////////////////
///////////////////  Declare key structures ////////////////////////////////
////////////////////////////////////////////////////////////////////////////

    Alg  algorithm;
    Sol  solution;
    Prob problem;

////////////////////////////////////////////////////////////////////////////
///////////////////  Register problem name  ////////////////////////////////
////////////////////////////////////////////////////////////////////////////

    problem.name           =       "DAE Index 3";
    problem.outfilename          =       "dae_i3.txt";

////////////////////////////////////////////////////////////////////////////
////////////  Define problem level constants & do level 1 setup ////////////
////////////////////////////////////////////////////////////////////////////

    problem.nphases             = 1;
    problem.nlinkages                  = 0;

    psopt_level1_setup(problem);


/////////////////////////////////////////////////////////////////////////////
/////////   Define phase related information & do level 2 setup /////////////
/////////////////////////////////////////////////////////////////////////////

    problem.phases(1).nstates     = 4;
    problem.phases(1).ncontrols   = 1;
    problem.phases(1).nevents     = 0;
    problem.phases(1).npath       = 1;
    problem.phases(1).nparameters       = 1;
    problem.phases(1).nodes         << 30;
    problem.phases(1).nobserved         = 2;
    problem.phases(1).nsamples          = 20;

    psopt_level2_setup(problem, algorithm);

////////////////////////////////////////////////////////////////////////////
////////////  Load data for parameter estimation                ////////////
////////////////////////////////////////////////////////////////////////////

   int iphase = 1;
   load_parameter_estimation_data(problem, iphase, "../../../examples/dae_i3/dae_i3.dat");

   Print(problem.phases(1).observation_nodes, "observation nodes");
   Print(problem.phases(1).observations, "observations");
   Print(problem.phases(1).residual_weights, "weights");


////////////////////////////////////////////////////////////////////////////
//////////////////  Declare MatrixXd objects to store results /////////////
////////////////////////////////////////////////////////////////////////////

    MatrixXd x, u, p, t;

////////////////////////////////////////////////////////////////////////////
//////////////////  Enter problem bounds information //////////////////////
////////////////////////////////////////////////////////////////////////////


    problem.phases(1).bounds.lower.states(0) = -2.0;
    problem.phases(1).bounds.lower.states(1) = -2.0;
    problem.phases(1).bounds.lower.states(2) = -2.0;
    problem.phases(1).bounds.lower.states(3) = -2.0;

    problem.phases(1).bounds.upper.states(0) = 2.0;
    problem.phases(1).bounds.upper.states(1) = 2.0;
    problem.phases(1).bounds.upper.states(2) = 2.0;
    problem.phases(1).bounds.upper.states(3) = 2.0;

    problem.phases(1).bounds.lower.controls(0) = -10.0;
```

```
    problem.phases(1).bounds.upper.controls(0) =  10.0;

    problem.phases(1).bounds.lower.parameters(0)  = 0.0;
    problem.phases(1).bounds.upper.parameters(0)  = 5.0;


    problem.phases(1).bounds.lower.path(0)  = 0.0;
    problem.phases(1).bounds.upper.path(0)  = 0.0;

    problem.phases(1).bounds.lower.StartTime    = 0.5;
    problem.phases(1).bounds.upper.StartTime    = 0.5;

    problem.phases(1).bounds.lower.EndTime      = 10.0;
    problem.phases(1).bounds.upper.EndTime      = 10.0;

////////////////////////////////////////////////////////////////////////////
///////////////////  Register problem functions  ///////////////////////////
////////////////////////////////////////////////////////////////////////////

    problem.dae  = &dae;
    problem.events  = &events;
    problem.linkages = &linkages;
    problem.observation_function = & observation_function;

////////////////////////////////////////////////////////////////////////////
///////////////////  Define & register initial guess ///////////////////////
////////////////////////////////////////////////////////////////////////////

    int nnodes =     (int) problem.phases(1).nsamples;

    MatrixXd state_guess(4, nnodes);
    MatrixXd control_guess(1,nnodes);
    MatrixXd param_guess(1,1);

    state_guess <<   problem.phases(1).observations.row(0),
     problem.phases(1).observations.row(1),
     ones(1,nnodes),
     ones(1,nnodes);

    control_guess = zeros(1,nnodes);

    param_guess << 0.5;


    problem.phases(1).guess.states        = state_guess;
    problem.phases(1).guess.time          = problem.phases(1).observation_nodes;
    problem.phases(1).guess.parameters    = param_guess;
    problem.phases(1).guess.controls      = control_guess;



////////////////////////////////////////////////////////////////////////////
///////////////////  Enter algorithm options  //////////////////////////////
////////////////////////////////////////////////////////////////////////////

    algorithm.nlp_method                  = "IPOPT";
    algorithm.scaling                     = "automatic";
    algorithm.derivatives                 = "automatic";
    algorithm.collocation_method          = "Legendre";


////////////////////////////////////////////////////////////////////////////
///////////////////  Now call PSOPT to solve the problem   /////////////////
////////////////////////////////////////////////////////////////////////////

    psopt(solution, problem, algorithm);

////////////////////////////////////////////////////////////////////////////
///////////  Extract relevant variables from solution structure   //////////
////////////////////////////////////////////////////////////////////////////

    x = solution.get_states_in_phase(1);
    u = solution.get_controls_in_phase(1);
    t = solution.get_time_in_phase(1);
    p = solution.get_parameters_in_phase(1);


////////////////////////////////////////////////////////////////////////////
///////////  Save solution data to files if desired ////////////////////////
////////////////////////////////////////////////////////////////////////////

    Save(x,"x.dat");
```

```
    Save(u,"u.dat");
    Save(t,"t.dat");
    Print(p,"Estimated parameter");


////////////////////////////////////////////////////////////////////////////
///////////  Plot some results if desired (requires gnuplot) ///////////////
////////////////////////////////////////////////////////////////////////////

     MatrixXd tm;
     MatrixXd ym;

     tm = problem.phases(1).observation_nodes;
     ym = problem.phases(1).observations;


     plot(t,x.row(0) ,tm,ym.row(0) ,problem.name, "time (s)", "state x1", "x1 yhat1");
     plot(t,x.row(1) ,tm,ym.row(1) ,problem.name, "time (s)", "state x2", "x2 yhat2");
     plot(t,u,problem.name, "time (s)", "algebraic state u", "u");

     plot(t,x.row(0),tm,ym.row(0),problem.name, "time (s)", "state x1", "x1 yhat1",
  "pdf", "x1.pdf");
     plot(t,x.row(1),tm,ym.row(1),problem.name, "time (s)", "state x2", "x2 yhat2",
  "pdf", "x2.pdf");
     plot(t,u,problem.name, "time (s)", "algebraic state lambda", "lambda", "pdf", "lambda.pdf");


}





////////////////////////////////////////////////////////////////////////////
///////////////////////      END OF FILE     ///////////////////////////////
////////////////////////////////////////////////////////////////////////////
```

The output from $\mathcal{PSOPT}$ summarised in the box below and shown in Figures 21 and 22, which compare the observations with the estimated outputs, and 23, which shows the algebraic state. The exact solution to the problem is $L = 1$ and $\lambda(t) = -1$. The numerical solution obtained is $L = 1.000000188$ and $\lambda(t) = -0.999868$. The 95% confidence interval for the estimated parameter is $[0.9095289, \;\; 1.090471]$

```
PSOPT results summary
=====================

Problem:  DAE Index 3
CPU time (seconds): 1.464757e+00
NLP solver used:  IPOPT
PSOPT release number:  5.0.3
Date and time of this run:  Thu Mar  6 16:55:33 2025

Optimal (unscaled) cost function value:  2.423655e-18
Phase 1 endpoint cost function value: 2.423655e-18
Phase 1 integrated part of the cost:  0.000000e+00
Phase 1 initial time: 5.000000e-01
Phase 1 final time: 1.000000e+01
```

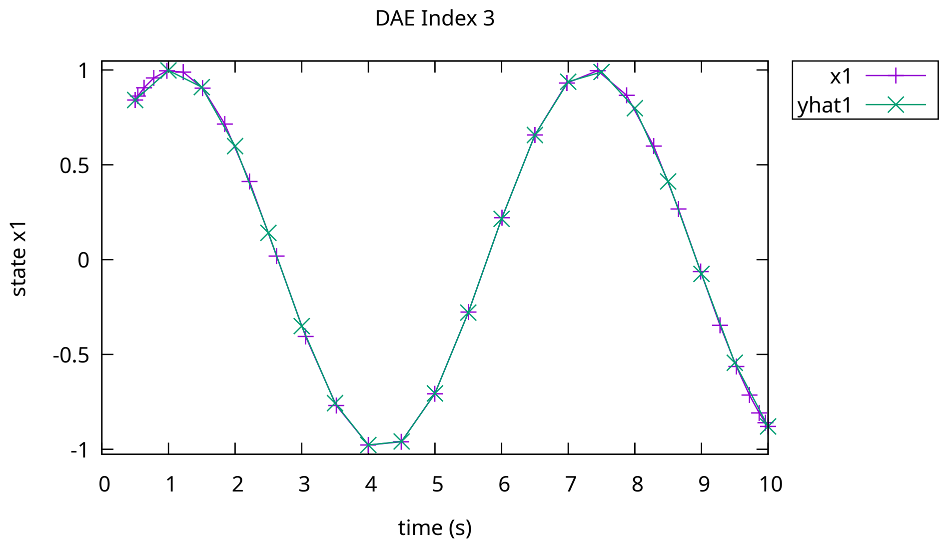

Figure 21: State x_1 and observations

```
Phase 1 maximum relative local error: 9.431335e-09
NLP solver reports:  The problem has been solved!
```

10 Delayed states problem 1

Consider the following optimal control problem, which consists of a linear system with delays in the state equations [14]. Minimize the cost functional:

$$J = x_3(t_f) \tag{40}$$

subject to the dynamic constraints

$$\begin{aligned} \dot{x}_1 &= x_2(t) \\ \dot{x}_2 &= -10x_1(t) - 5x_2(t) - 2x_1(t-\tau) - x_2(t-\tau) + u(t) \\ \dot{x}_3 &= 0.5(10x_1^2(t) + x_2^2(t) + u^2(t)) \end{aligned} \tag{41}$$

and the boundary conditions

$$\begin{aligned} x_1(0) &= 1 \\ x_2(0) &= 1 \\ x_3(0) &= 0 \end{aligned} \tag{42}$$

where $t_f = 5$ and $\tau = 0.25$.

The output from $\mathcal{PSOPT}$ summarised in the box below and shown in Figures 24 and 25, which contain the elements of the state and the control, respectively.

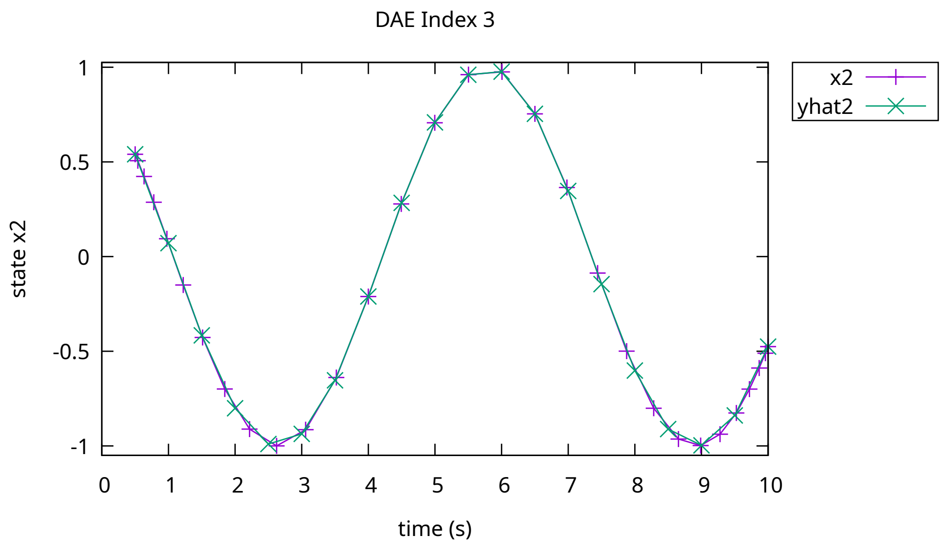

Figure 22: State x_2 and observations

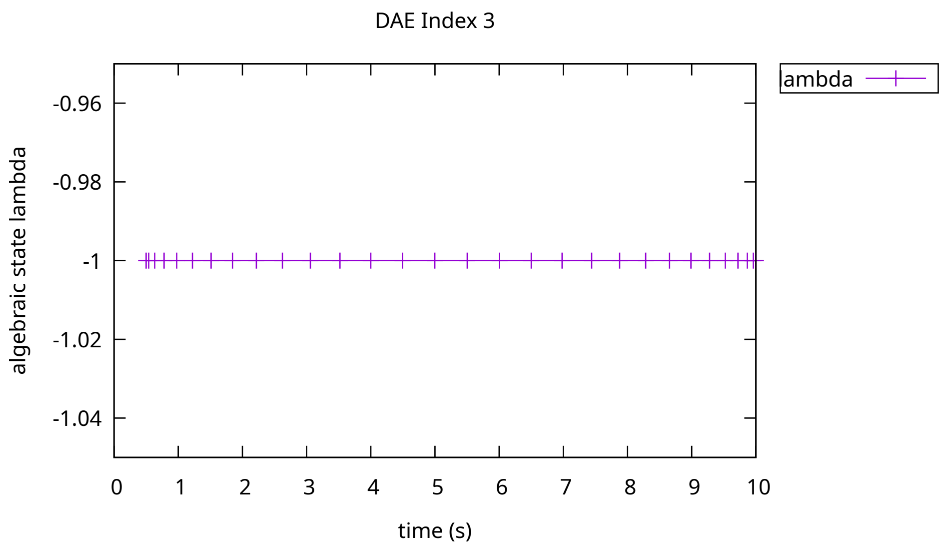

Figure 23: Algebraic state $\lambda(t)$

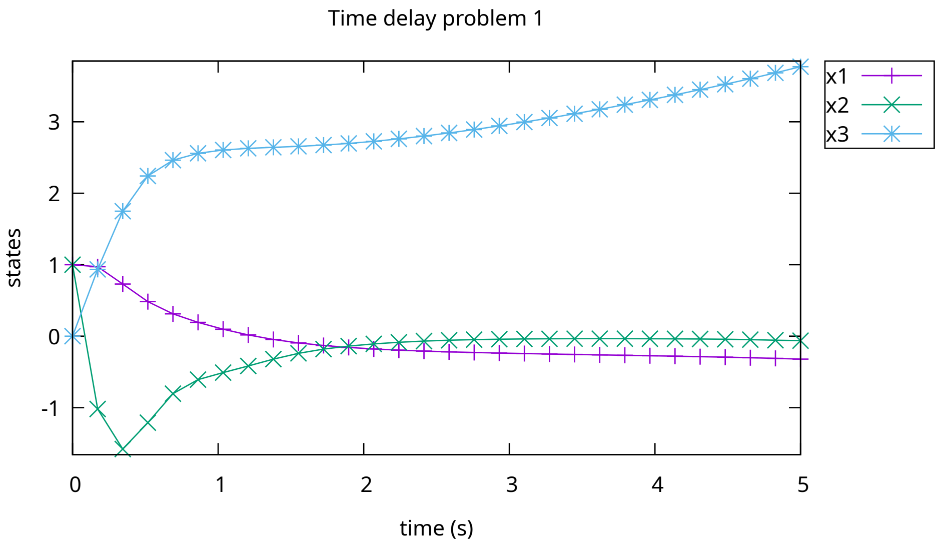

Figure 24: States for time delay problem 1

```
PSOPT results summary
=====================

Problem:  Time delay problem 1
CPU time (seconds): 4.242920e-01
NLP solver used:  IPOPT
PSOPT release number:  5.0.3
Date and time of this run:  Thu Mar  6 16:56:04 2025

Optimal (unscaled) cost function value:  3.770849e+00
Phase 1 endpoint cost function value: 3.770849e+00
Phase 1 integrated part of the cost:  0.000000e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 5.000000e+00
Phase 1 maximum relative local error: 1.838200e-02
NLP solver reports:  The problem has been solved!
```

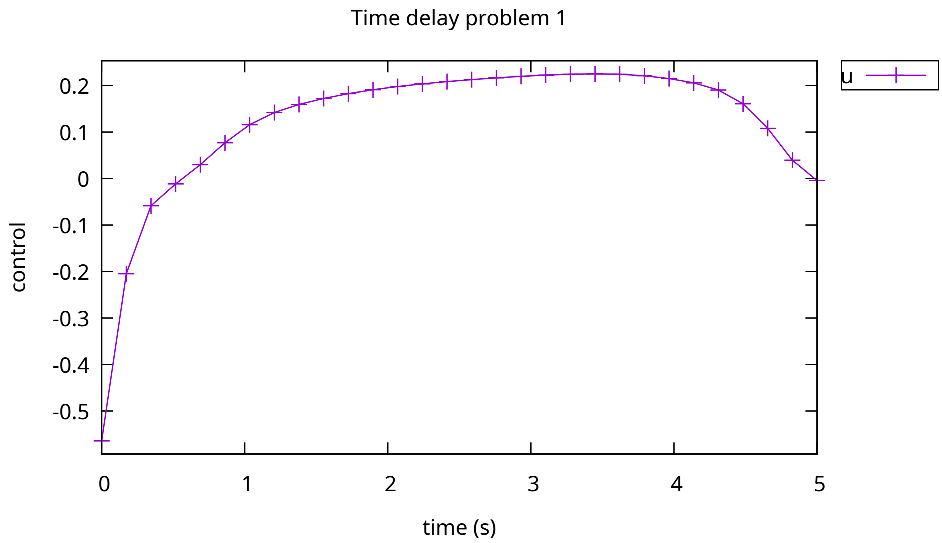

Figure 25: Control for time delay problem 1

11 Dynamic MPEC problem

Consider the following optimal control problem, which involves special handling of a system with a discontinuous right hand side [4]. Minimize the cost functional:

$$J = [y(2) - 5/3]^2 + \int_0^2 y^2(t)\mathrm{d}t \tag{43}$$

subject to

$$\dot{y} \quad = \quad 2 - \mathrm{sgn}(y) \tag{44}$$

and the boundary condition

$$y(0) \quad = \quad -1 \tag{45}$$

Note that there is no control variable, and the analytical solution of this problem satisfies $\dot{y}(t) = 3$, $0 \le t \le 1/3$, and $\dot{y}(t) = 1$, $1/3 \le t \le 2$.

In order to handle the discontinuous right hand side, the problem is converted into the following equivalent problem, which has three algebraic (control) variables. This type of problem is known in the literature as a dynamic MPEC problem.

$$J = [y(2) - 5/3]^2 + \int_0^2 \left(y^2(t) + \rho\left\{p(t)[s(t)+1] + q(t)[1 - s(t)]\right\}\right) \mathrm{d}t \tag{46}$$

subject to

$$\begin{array}{rcl} \dot{y} & = & 2 - \mathrm{sgn}(y) \\ 0 & = & -y(t) - p(t) + q(t) \end{array} \tag{47}$$

the boundary condition

$$y(0) = -1 \tag{48}$$

and the bounds:

$$\begin{array}{lll} -1 & \leq & s(t) \leq 1, \\ 0 & \leq & p(t), \\ 0 & \leq & q(t). \end{array} \tag{49}$$

The output from $\mathcal{PSOPT}$ summarised in the box below and shown in Figures 26, 27, 28, and 29.

```
PSOPT results summary
=====================

Problem:  Dynamic MPEC problem
CPU time (seconds): 1.034172e+00
NLP solver used:  IPOPT
PSOPT release number:  5.0.3
Date and time of this run:  Thu Mar  6 17:00:21 2025

Optimal (unscaled) cost function value:  1.656948e+00
Phase 1 endpoint cost function value: 3.271531e-07
Phase 1 integrated part of the cost:  1.656948e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 2.000000e+00
Phase 1 maximum relative local error: 2.178111e-06
NLP solver reports:  The problem has been solved!
```

12 Geodesic problem

This problem is about calculating the geodesic curve [1] that joins two points on Earth using optimal control. The problem is posed in the form of estimating the shortest fligh path for an airliner to fly from New Yorks's JFK to London's LHR airport.

The formulation is as follows. Find the trajectories for the elevation and azimuth angles $\theta(t)$ and $\phi(t) \in [0, t_f]$ to minimize the cost functional

$$J = \int_0^{t_f} \sqrt{\dot{x}^2 + \dot{y}^2 + \dot{z}^2} dt \tag{50}$$

[1] See http://mathworld.wolfram.com/Geodesic.html

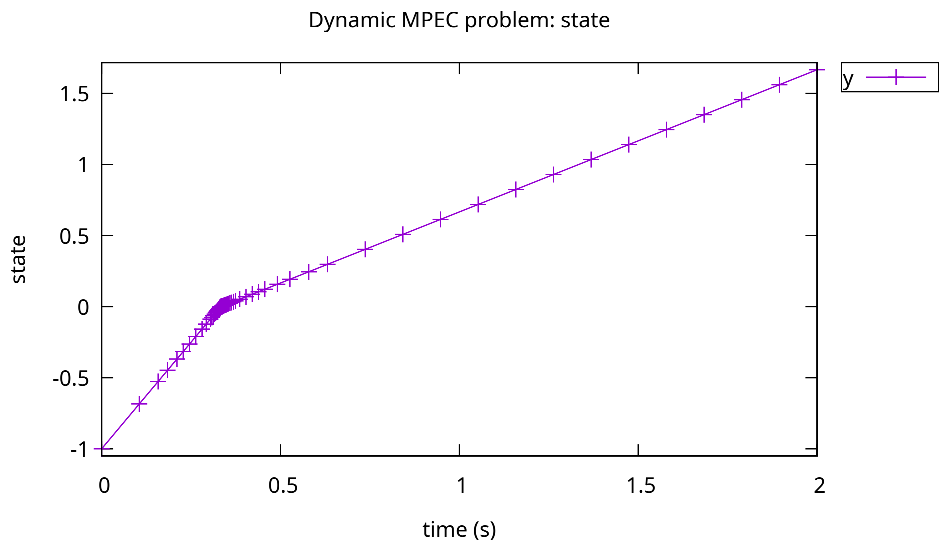

Figure 26: State y for dynamic MPEC problem

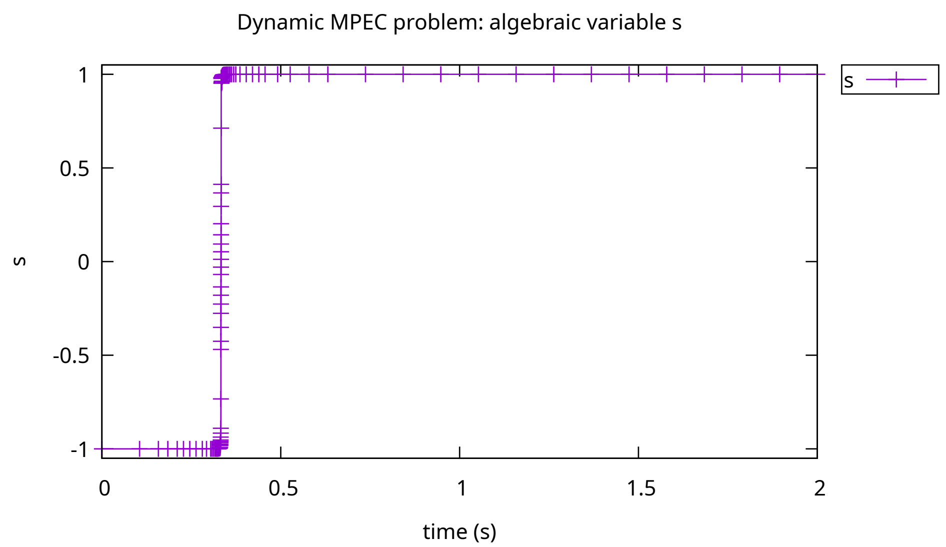

Figure 27: Algebraic variable s for dynamic MPEC problem

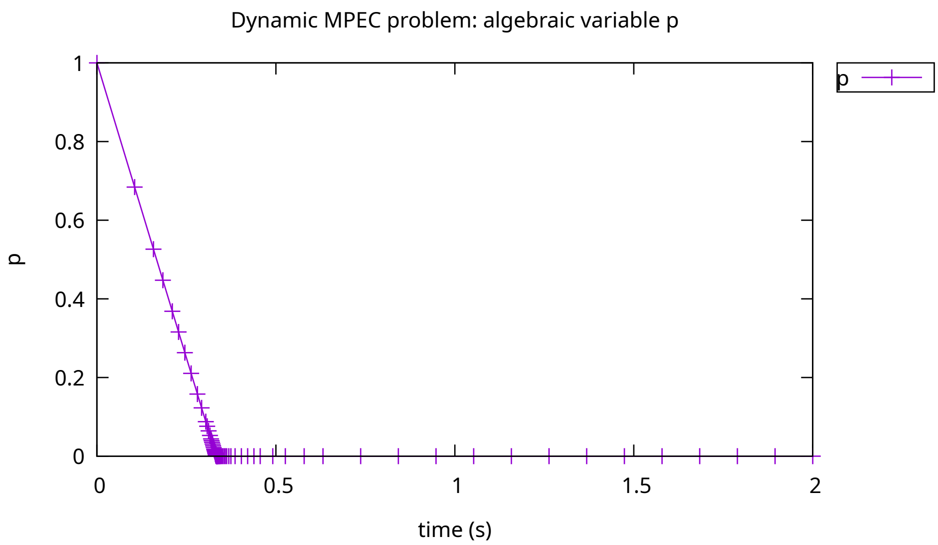

Figure 28: Algebraic variable p for dynamic MPEC problem

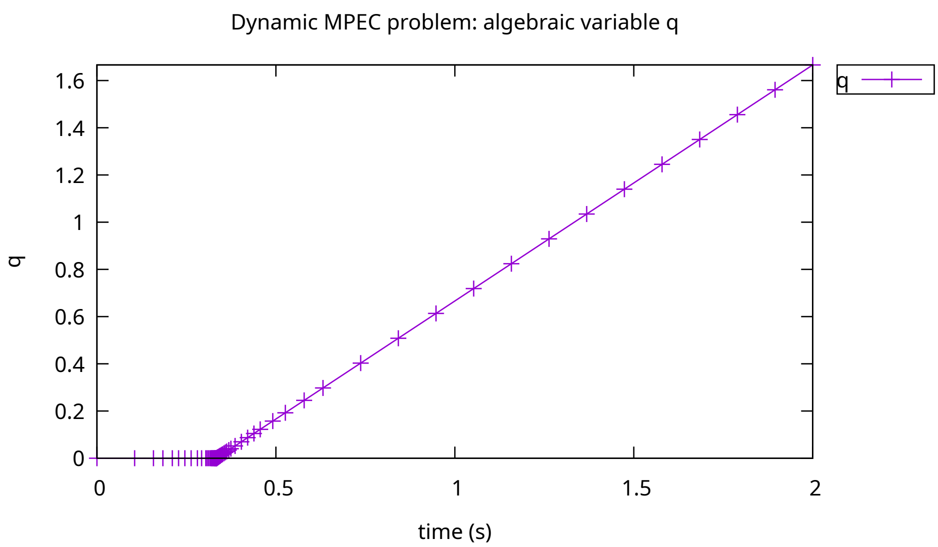

Figure 29: Algebraic variable q for dynamic MPEC problem

subject to the dynamic constraints

$$\begin{aligned} \dot{x} &= V\sin(\theta)\cos(\phi) \\ \dot{y} &= V\sin(\theta)\sin(\phi) \\ \dot{z} &= V\cos(\theta) \end{aligned} \tag{51}$$

The path constraint, which corresponds to the Earth's spheroid [2] shape:

$$\frac{x^2}{a^2} + \frac{y^2}{a^2} + \frac{z^2}{b^2} - 1.0 = 0 \tag{52}$$

the boundary conditions, which correspond to the geographical coordinates of LHR (51.4700° N, 0.4543° W) and JFK (40.6413° N, 73.7781° W)

$$\begin{aligned} x(0) &= x_0 \\ y(0) &= y_0 \\ z(0) &= z_0 \\ x(t_f) &= x_f \\ y(t_f) &= y_f \\ z(t_f) &= z_f \end{aligned} \tag{53}$$

and the control bounds

$$\begin{aligned} 0 \leq \ & \theta(t) \leq \pi \\ 0 \leq \ & \phi(t) \leq 2\pi \end{aligned} \tag{54}$$

where x, y, z are the Cartesian coordinates (in km) with origin on the centre of Earth, t is time in hours, $V = 900$ km/h corresponds to the cruising speed of a typical airliner, $a = 6384$ km is the Earth's semi-major axis, and $b = 6353$ km is the Earth's semi-minor axis, which is the length of the Earth's axis of rotation from the north pole to the south pole. For simplicity, the altitude of the aircraft is neglected.

The $\mathcal{PSOPT}$ code that solves this problem is shown below.

The output from $\mathcal{PSOPT}$ is summarised in the box below and shown in Figures 30, 31 and 32, which show the flight path, the elements of the state vector, and the elements of the control vector, respectively. Note that $\mathcal{PSOPT}$ predicts that the length of the shortest flightpath is 5,540.4 km, and the flight time is 6 hours 9 min.

```
PSOPT results summary
=====================

Problem:  Geodesic problem
CPU time (seconds): 9.437560e-01
NLP solver used:  IPOPT
PSOPT release number:  5.0.3
```

[2]See http://mathworld.wolfram.com/Ellipsoid.html

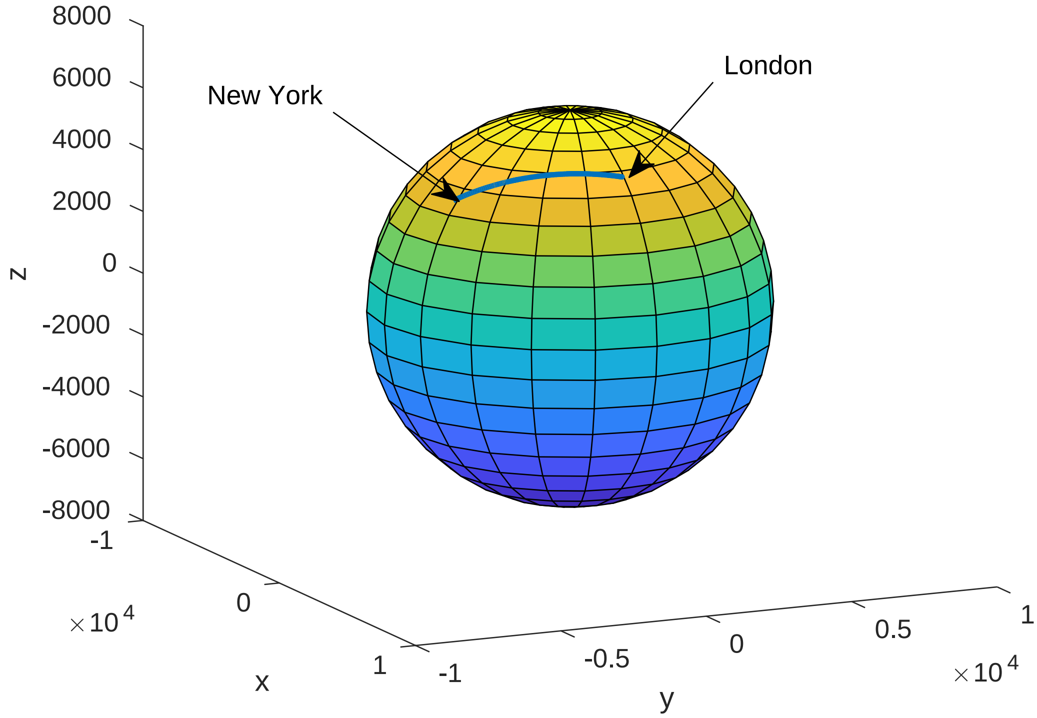

Figure 30: Flight path for geodesic problem

```
Date and time of this run:  Thu Mar  6 16:56:32 2025

Optimal (unscaled) cost function value:  5.540439e+03
Phase 1 endpoint cost function value: 0.000000e+00
Phase 1 integrated part of the cost:  5.540439e+03
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 6.156043e+00
Phase 1 maximum relative local error: 6.717589e-05
NLP solver reports:  The problem has been solved!
```

13 Goddard rocket maximum ascent problem

Consider the following optimal control problem, which is known in the literature as the Goddard rocket maximum ascent problem [6]. Find t_f and $T(t) \in [t_0, t_f]$ to minimize

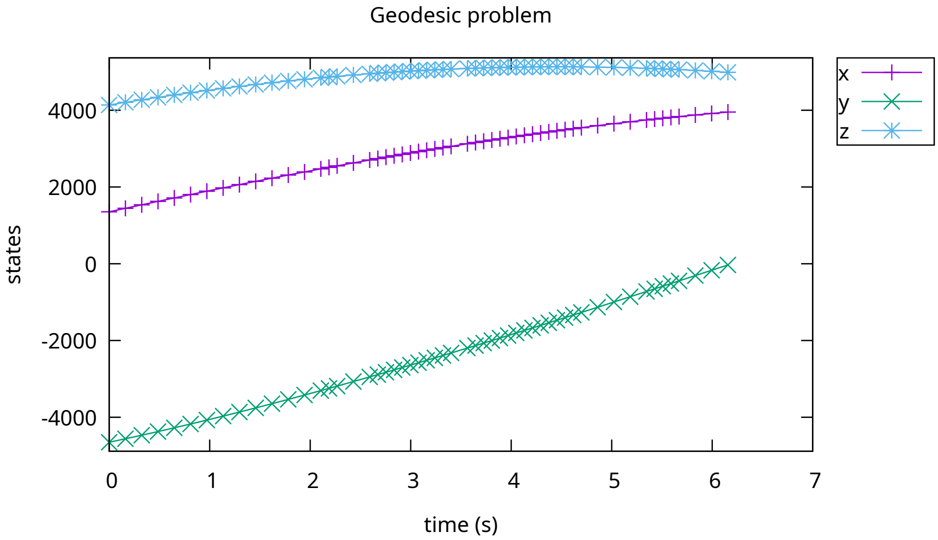

Figure 31: States for geodesic problem

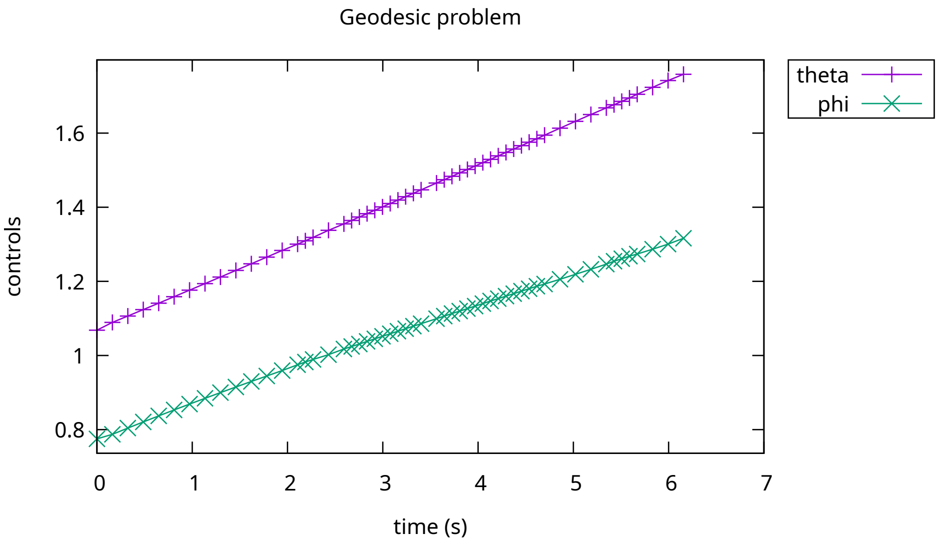

Figure 32: Controls for geodesic problem

the cost functional

$$J = h(t_f) \tag{55}$$

subject to the dynamic constraints

$$\begin{array}{rcl} \dot{v} & = & \frac{1}{m}(T - D) - g \\ \dot{h} & = & v \\ \dot{m} & = & -\frac{T}{c} \end{array} \tag{56}$$

the boundary conditions:

$$\begin{array}{rcl} v(0) & = & 0 \\ h(0) & = & 1 \\ m(0) & = & 1 \\ m(t_f) & = & 0.6 \end{array} \tag{57}$$

the state bounds:

$$\begin{array}{rcl} 0.0 \leq & v(t) & \leq 2.0 \\ 1.0 \leq & h(t) & \leq 2.0 \\ 0.6 \leq & m(t) & \leq 1.0 \end{array} \tag{58}$$

and the control bounds

$$0 \leq T(t) \leq 3.5 \tag{59}$$

where

$$\begin{array}{rcl} D & = & D_0 v^2 \exp(-\beta h) \\ g & = & 1/(h^2) \end{array}, \tag{60}$$

$D_0 = 310$, $\beta = 500$, and $c = 0.5$, $0.1 \leq t_f \leq 1$.

The output from $\mathcal{PSOPT}$ is summarised in the box below and shown in Figures 33 and 34, which contain the elements of the state and the control, respectively.

```
PSOPT results summary
=====================

Problem:  Goddard Rocket Maximum Ascent
CPU time (seconds): 1.558993e+00
NLP solver used:  IPOPT
PSOPT release number:  5.0.3
Date and time of this run:  Thu Mar  6 16:57:16 2025

Optimal (unscaled) cost function value:  -1.025336e+00
Phase 1 endpoint cost function value: -1.025336e+00
Phase 1 integrated part of the cost:  0.000000e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 2.605347e-01
Phase 1 maximum relative local error: 6.877091e-04
NLP solver reports:  The problem has been solved!
```

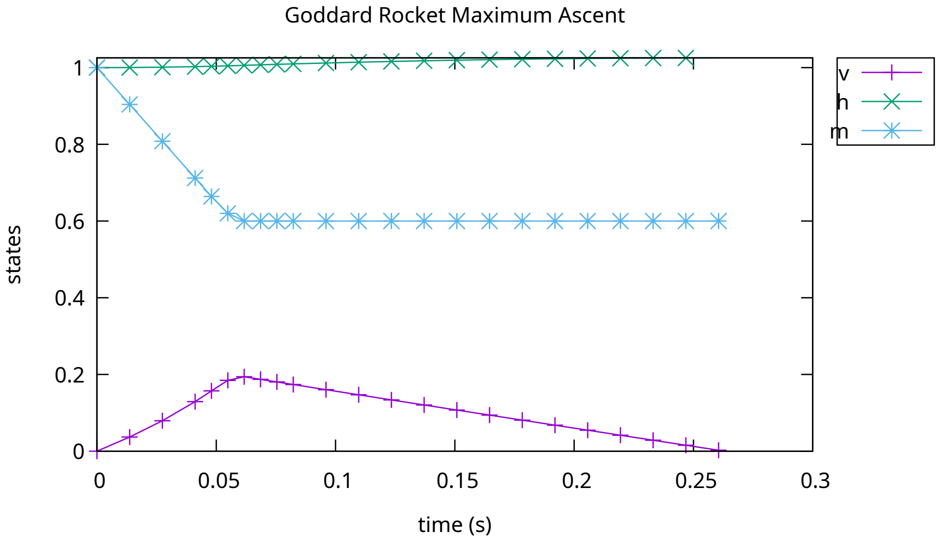

Figure 33: States for Goddard rocket problem

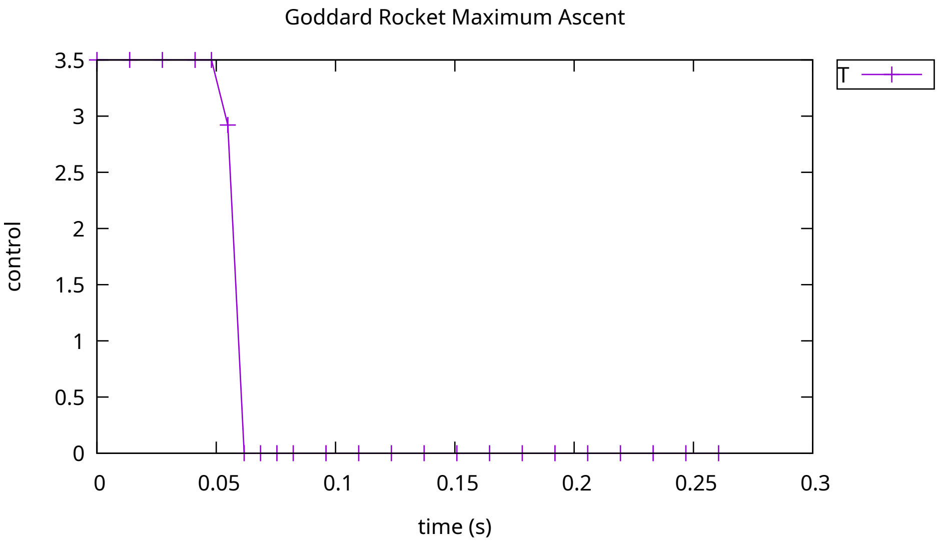

Figure 34: Control for Goddard rocket problem

14 Hang glider

This problem is about the range maximisation of a hang glider in the presence of a specified thermal draft [4]. Find t_f and $C_L(t), t \in [0, t_f]$, to minimise,

$$J = x(t_f) \tag{61}$$

subject to the dynamic constraints

$$\begin{aligned} \dot{x} &= v_x \\ \dot{y} &= v_y \\ \dot{v}_x &= \tfrac{1}{m}(-L\sin\eta - D\cos\eta) \\ \dot{v}_y &= \tfrac{1}{m}(L\cos\eta - D\sin\eta - W) \end{aligned} \tag{62}$$

where

$$\begin{aligned} C_D &= C_0 + kC_L^2 \\ v_r &= \sqrt{v_x^2 + v_y^2} \\ D &= \frac{1}{2}C_D\rho S v_r^2 \\ L &= \frac{1}{2}C_L\rho S v_r^2 \\ X &= \left(\frac{x}{R} - 2.5\right)^2 \\ u_a &= u_M(1 - X)\exp(-X) \\ V_y &= v_y - ua \\ \sin\eta &= \frac{V_y}{v_r} \\ \cos\eta &= \frac{v_x}{v_r} \\ W &= mg \end{aligned} \tag{63}$$

The control is bounded as follows:

$$0 \leq C_L \leq 1.4 \tag{64}$$

and the following boundary conditions:

$$\begin{aligned} x(0) &= 0, & x(t_f) &= \text{free} \\ y(0) &= 1000, & y(t_f) &= 900 \\ v_x(0) &= 13.227567500, & v_x(t_f) &= 13.227567500 \\ v_y(0) &= -1.2875005200, & v_y(t_f) &= -1.2875005200 \end{aligned} \tag{65}$$

With the following parameter values:

$$\begin{aligned} u_M &= 2.5, & m &= 100.0 \\ R &= 100.0, & S &= 14, \\ C_0 &= 0.034, & \rho &= 1.13 \\ k &= 0.069662, & g &= 9.80665 \end{aligned} \tag{66}$$

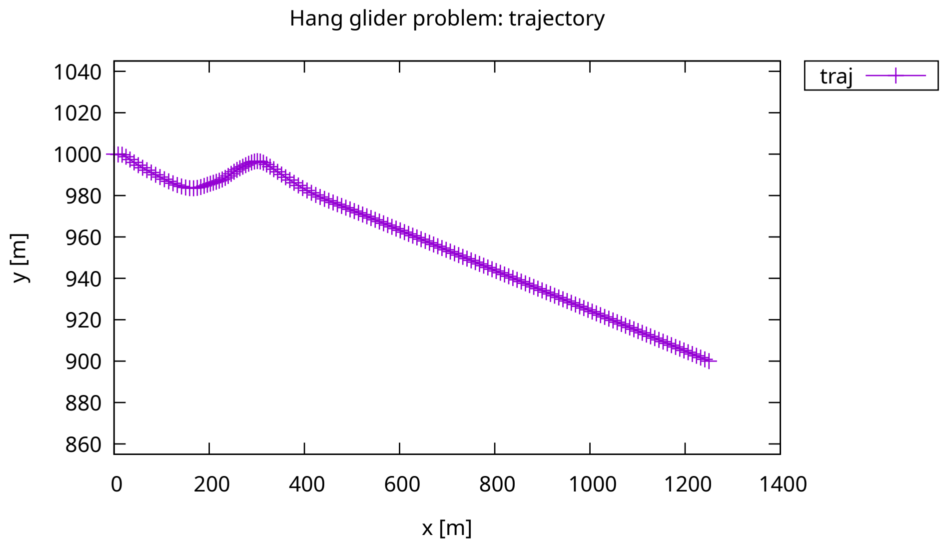

Figure 35: $x - y$ trajectory for hang glider

The output from $\mathcal{PSOPT}$ is summarised in the box below and shown in Figures 35, 36 and 37.

```
PSOPT results summary
=====================

Problem:  Hang glider problem
CPU time (seconds): 2.307024e+00
NLP solver used:  IPOPT
PSOPT release number:  5.0.3
Date and time of this run:  Thu Mar  6 16:57:01 2025

Optimal (unscaled) cost function value:  -1.250190e+03
Phase 1 endpoint cost function value: -1.250190e+03
Phase 1 integrated part of the cost:  0.000000e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 9.888716e+01
Phase 1 maximum relative local error: 1.631133e-01
NLP solver reports:  The problem has been solved!
```

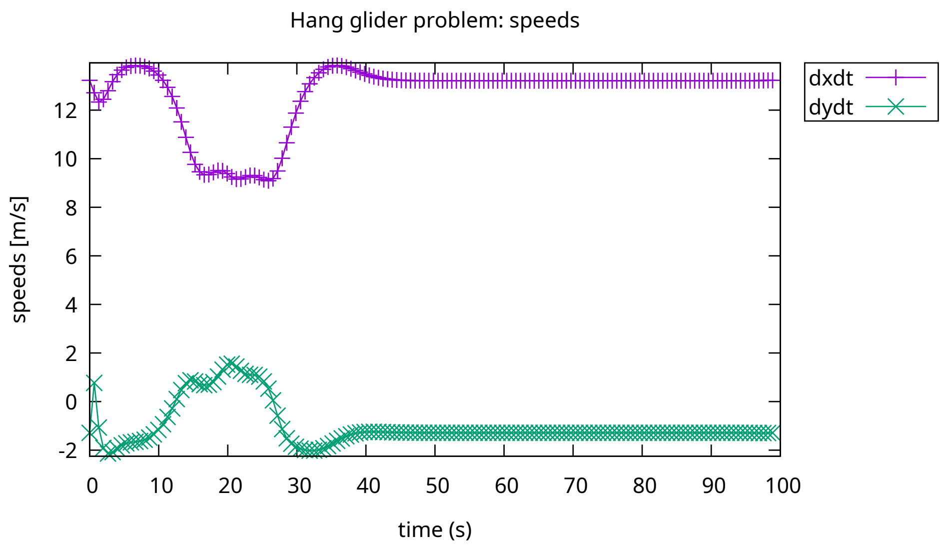

Figure 36: Velocities for hang glider

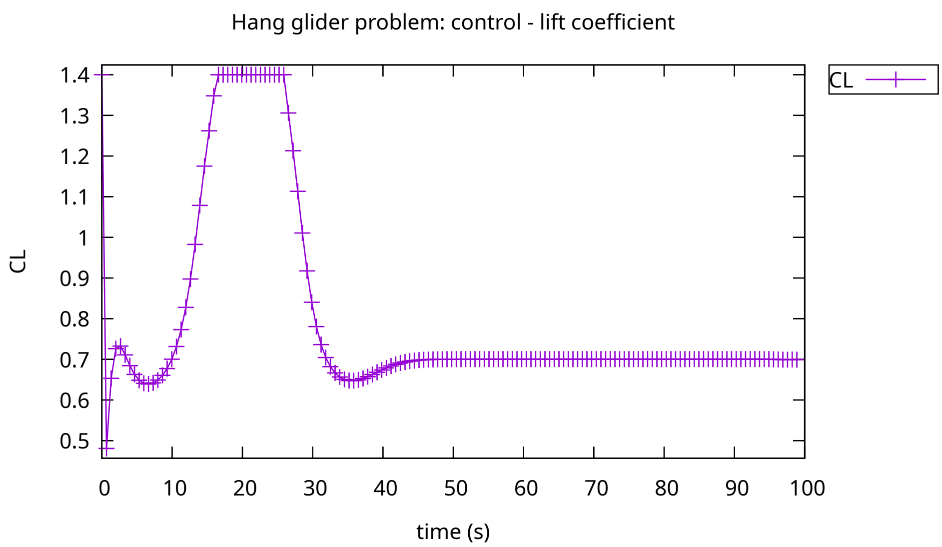

Figure 37: Lift coefficient for hang glider problem

15 Hanging chain problem

Consider the following optimal control problem, which includes an integral constraint. Minimize the cost functional

$$J = \int_0^{t_f} \left[x\sqrt{1 + (\dot{x})^2} \right] dt \tag{67}$$

subject to the dynamic constraint

$$\dot{x} = u \tag{68}$$

the integral constraint:

$$\int_0^{t_f} \left[\sqrt{1 + \left(\frac{dx}{dt}\right)^2} \right] dt = 4 \tag{69}$$

the boundary conditions

$$\begin{array}{rcl} x(0) & = & 1 \\ x(t_f) & = & 3 \end{array} \tag{70}$$

and the bounds:

$$\begin{array}{c} -20 \le u(t) \le 20 \\ -10 \le x(t) \le 10 \end{array} \tag{71}$$

where $t_f = 1$.

The output from $\mathcal{PSOPT}$ is summarized in the text box below and in Figure 38, which illustrates the shape of the hanging chain.

```
PSOPT results summary
=====================

Problem:  Hanging chain problem
CPU time (seconds): 7.381840e-01
NLP solver used:  IPOPT
PSOPT release number:  5.0.3
Date and time of this run:  Thu Mar  6 16:52:22 2025

Optimal (unscaled) cost function value:  5.068480e+00
Phase 1 endpoint cost function value: 0.000000e+00
Phase 1 integrated part of the cost:  5.068480e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 1.000000e+00
Phase 1 maximum relative local error: 1.875570e-06
NLP solver reports:  The problem has been solved!
```

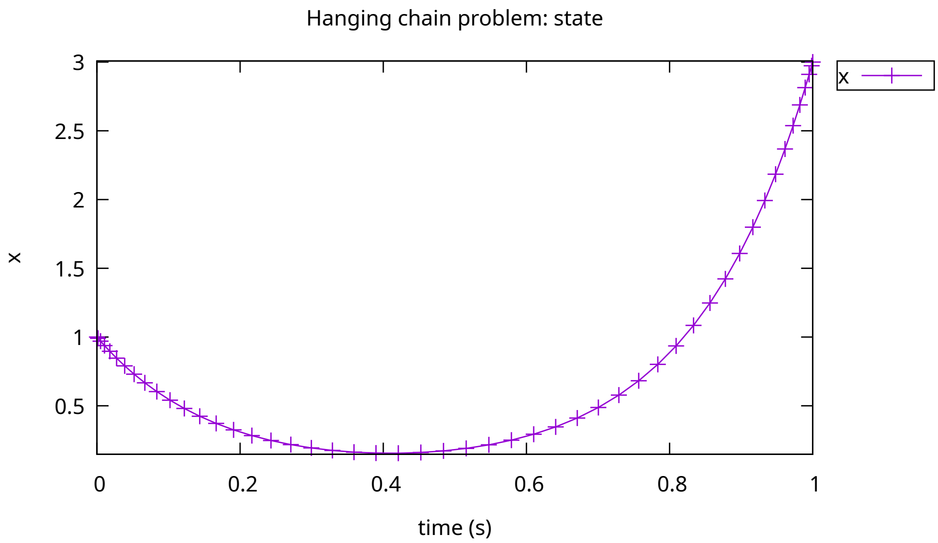

Figure 38: State for hanging chain problem

16 Heat difussion problem

This example can be viewed as a simplified model for the heating of a probe in a kiln [3]. The dynamics are a spatially discretized form of a partial differential equation, which is obtained by using the method of the lines. The problem is formulated on the basis of the state vector $\mathbf{x} = [x_1, \ldots, x_M]^T$ and the control vector $\mathbf{u} = [v_1, v_2, v_3]^T$, as follows

$$\min_{\mathbf{u}(t)} \; J = \frac{1}{2}\int_0^T \left\{(x_N(t) - x_d(t))^2 + \gamma v_1(t)^2\right\} \mathrm{d}t$$

subject to the differential constraints

$$\begin{aligned}
\dot{x}_1 &= \frac{1}{(a_1 + a_2 x_1)}\left[q_1 + \frac{1}{\delta^2}(a_3 + a_4 x_1)(x_2 - 2x_1 + v_2) + a_4\left(\frac{x_2 - x_1}{2\delta}\right)^2\right] \\
\dot{x}_i &= \frac{1}{(a_1 + a_2 x_i)}\left[q_i + \frac{1}{\delta^2}(a_3 + a_4 x_i)(x_{i+1} - 2x_i + x_{i-1}) + a_4\left(\frac{x_{i+1} - x_{i-1}}{2\delta}\right)^2\right] \\
&\quad \text{for } i = 2, \ldots, M-1 \\
\dot{x}_M &= \frac{1}{(a_1 + a_2 x_M)}\left[q_M + \frac{1}{\delta^2}(a_3 + a_4 x_M)(v_3 - 2x_N + x_{M-1}) + a_4\left(\frac{v_3 - x_{M-1}}{2\delta}\right)^2\right]
\end{aligned}$$

the path constraints

$$\begin{aligned}
0 &= g(x_1 - v_1) - \frac{1}{2\delta}(a_3 + a_4 x_1)(x_2 - v_2) \\
0 &= \frac{1}{2\delta}(a_3 + a + 4x_M)(v_3 - x_{M-1})
\end{aligned}$$

the control bounds

$$u_L \le v_1 \le u_U$$

and the initial conditions for the states:

$$x_i(0) = 2 + \cos(\pi z_i)$$

where

$$\begin{aligned}
z_i &= \frac{i-1}{M-1},\ i = 1, \ldots, M \\
x_d(t) &= 2 - e^{\rho t} \\
q(z,t) &= \left[\rho(a_1 + 2a_2) + \pi^2(a_3 + 2a_4)\right] e^{\rho t}\cos(\pi z) \\
&\quad - a_4\pi^2 e^{2\pi t} + (2a_4\pi^2 + \rho a_2)e^{2\rho t}\cos^2(\pi z) \\
q_i &\equiv q(z_i, t),\ i = 1, \ldots, M
\end{aligned}$$

with the parameter values $a_1 = 4$, $a_2 = 1$, $a_3 = 4$, $a_4 = -1$, $u_U = 0.1$, $\rho = -1$, $T = 0.5$, $\gamma = 10^{-3}$, $g = 1$, $u_L = -\infty$.

A spatial discretization given by $M = 10$ was used. The problem was solved initially by using first 50 nodes, then the mesh was refined to 60 nodes, and an interpolation of the previous solution was employed as an initial guess for the new solution.

The output from $\mathcal{PSOPT}$ is summarized the box below. Figure 39 shows the control variable v_1 as a function of time. Figure 40 shows the resulting temperature distribution.

```
PSOPT results summary
=====================

Problem:  Heat diffusion process
CPU time (seconds): 1.022578e+00
NLP solver used:  IPOPT
PSOPT release number:  5.0.3
Date and time of this run:  Thu Mar  6 16:57:26 2025

Optimal (unscaled) cost function value:  4.372837e-05
Phase 1 endpoint cost function value: 0.000000e+00
Phase 1 integrated part of the cost:  4.372837e-05
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 5.000000e-01
Phase 1 maximum relative local error: 1.749463e-04
NLP solver reports:  The problem has been solved!
```

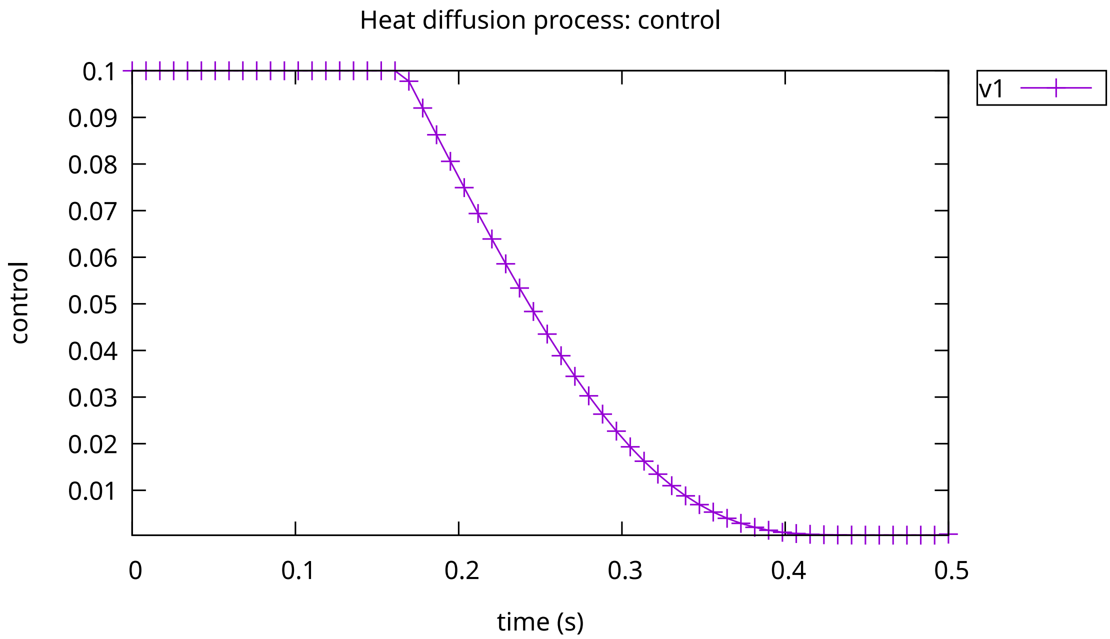

Figure 39: Optimal control distribution for the heat diffusion process

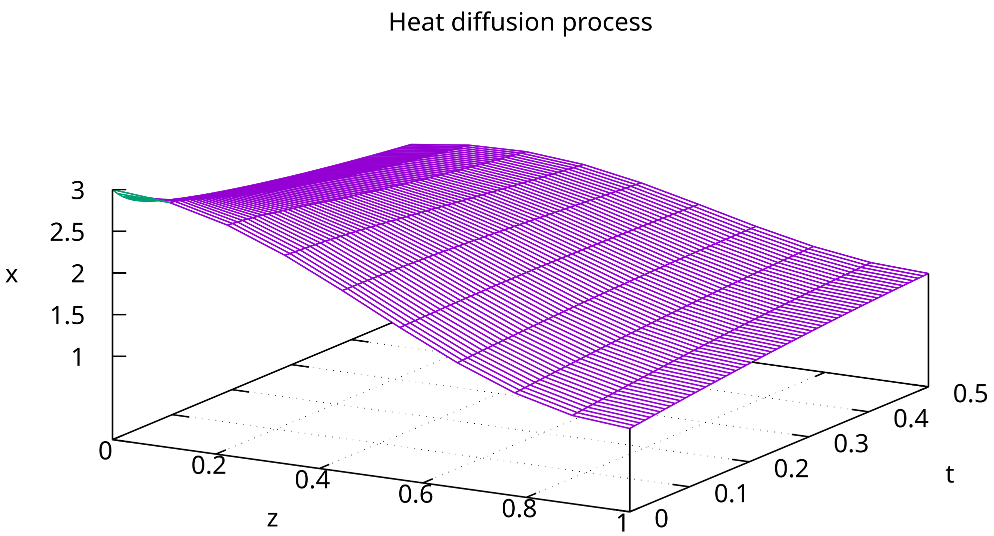

Figure 40: Optimal temperature distribution for the heat diffusion process

17 Hypersensitive problem

Consider the following optimal control problem, which is known in the literature as the hypesensitive optimal control problem [17]. Minimize the cost functional

$$J = \frac{1}{2}\int_0^{t_f} [x^2 + u^2] dt \tag{72}$$

subject to the dynamic constraint

$$\dot{x} = -x^3 + u \tag{73}$$

and the boundary conditions

$$\begin{array}{rcl} x(0) & = & 1.5 \\ x(t_f) & = & 1 \end{array} \tag{74}$$

where $t_f = 50$.

The output from $\mathcal{PSOPT}$ is summarized the box below and shown in the following plots that contain the elements of the state and the control, respectively.

```
PSOPT results summary
=====================

Problem:  Hypersensitive problem
CPU time (seconds): 3.629070e-01
NLP solver used:  IPOPT
PSOPT release number:  5.0.3
Date and time of this run:  Thu Mar  6 16:57:41 2025

Optimal (unscaled) cost function value:  1.330826e+00
Phase 1 endpoint cost function value: 0.000000e+00
Phase 1 integrated part of the cost:  1.330826e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 5.000000e+01
Phase 1 maximum relative local error: 5.728571e-04
NLP solver reports:  The problem has been solved!
```

18 Interior point constraint problem

Consider the following optimal control problem, which involves a scalar system with an interior point constraint on the state [12]. Minimize the cost functional

$$J = \int_0^1 [x^2 + u^2] dt \tag{75}$$

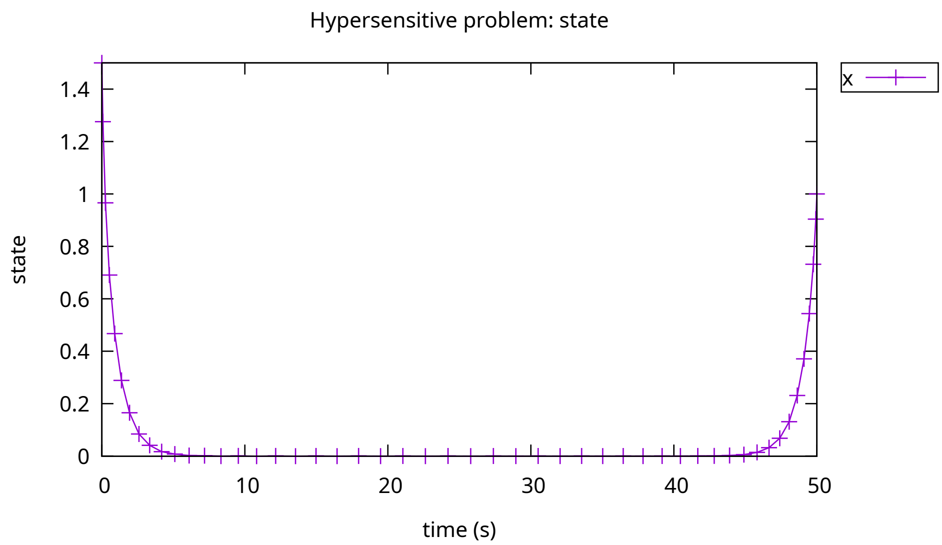

Figure 41: State for hypersensitive problem

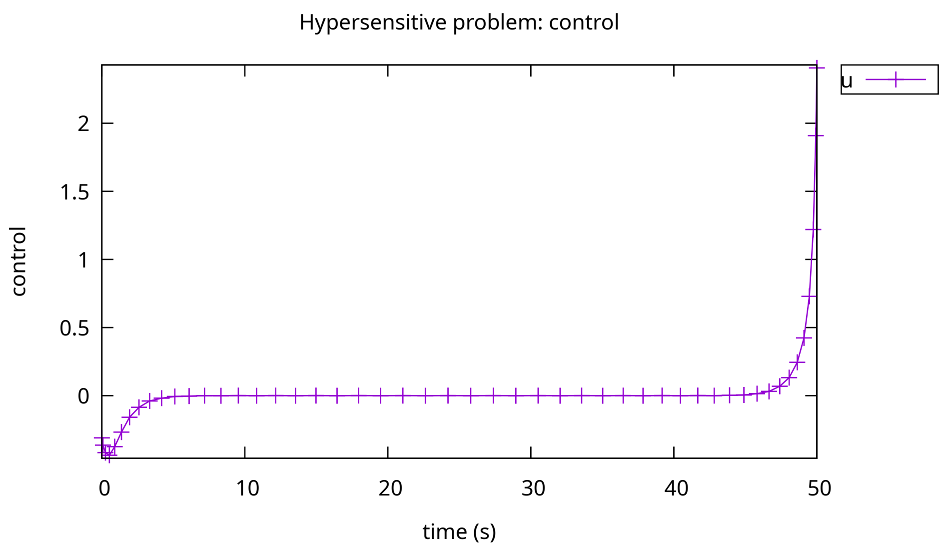

Figure 42: Control for hypersensitive problem

subject to the dynamic constraint

$$\dot{x} = u, \tag{76}$$

the boundary conditions

$$\begin{array}{rcl} x(0) & = & 1, \\ x(1) & = & 0.75, \end{array} \tag{77}$$

and the interior point constraint:

$$x(0.75) = 0.9. \tag{78}$$

The problem is divided into two phases and the interior point constraint is accommodated as an event constraint at the end of the first phase.

The output from $\mathcal{PSOPT}$ is summarized the box below and shown in the following plots that contain the elements of the state and the control, respectively.

```
PSOPT results summary
=====================

Problem:  Problem with interior point constraint
CPU time (seconds): 1.328690e-01
NLP solver used:  IPOPT
PSOPT release number:  5.0.3
Date and time of this run:  Thu Mar  6 16:58:02 2025

Optimal (unscaled) cost function value:  9.205314e-01
Phase 1 endpoint cost function value: 0.000000e+00
Phase 1 integrated part of the cost:  6.607877e-01
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 7.500000e-01
Phase 1 maximum relative local error: 2.331700e-08
Phase 2 endpoint cost function value: 0.000000e+00
Phase 2 integrated part of the cost:  2.597438e-01
Phase 2 initial time: 7.500000e-01
Phase 2 final time: 1.000000e+00
Phase 2 maximum relative local error: 6.746923e-09
NLP solver reports:  The problem has been solved!
```

19 Isoperimetric constraint problem

Consider the following optimal control problem, which includes an integral constraint. Minimize the cost functional

$$J = \int_0^{t_f} x^2(t)dt \tag{79}$$

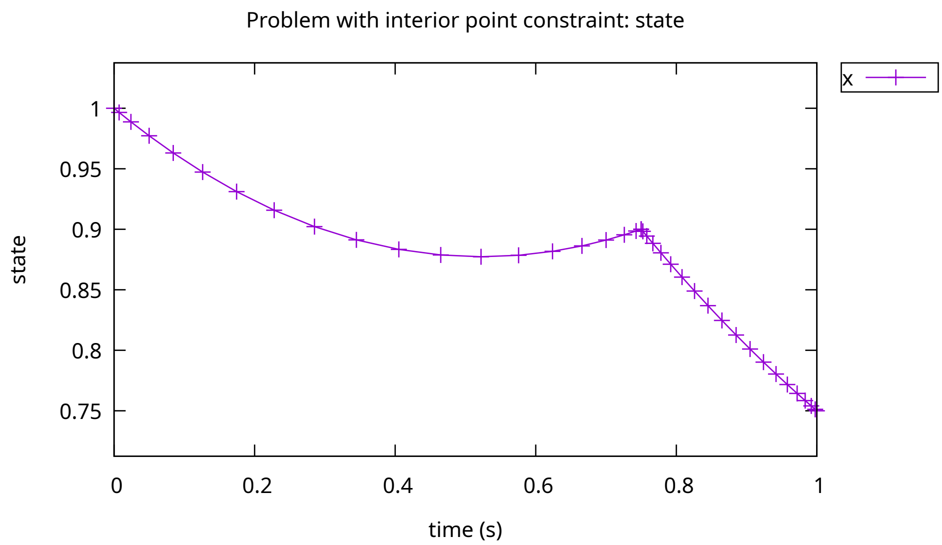

Figure 43: State for interior point constraint problem

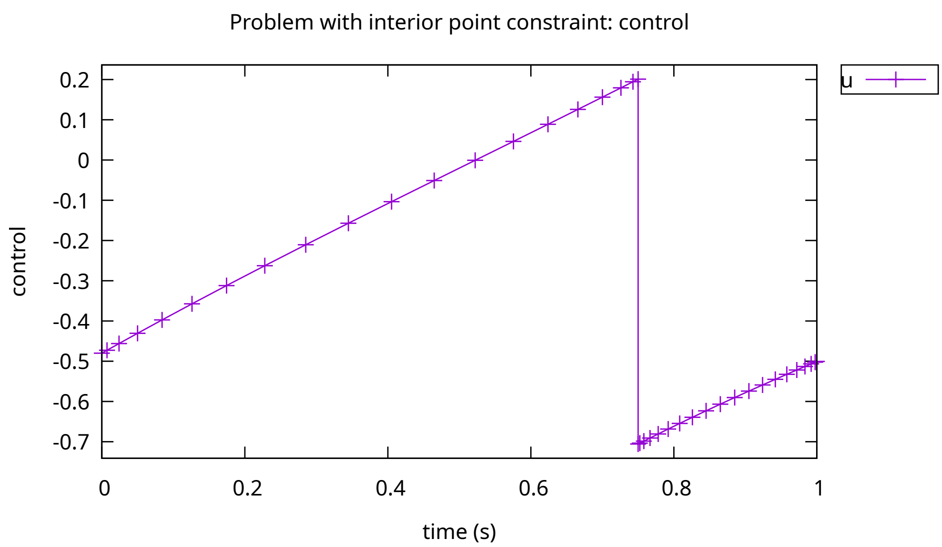

Figure 44: Control for interior point constraint problem

subject to the dynamic constraint

$$\dot{x} = -\sin(x) + u \tag{80}$$

the integral constraint:

$$\int_0^{t_f} u^2(t)dt = 10 \tag{81}$$

the boundary conditions

$$\begin{aligned} x(0) &= 1 \\ x(t_f) &= 0 \end{aligned} \tag{82}$$

and the bounds:

$$\begin{aligned} -4 \leq u(t) \leq 4 \\ -10 \leq x(t) \leq 10 \end{aligned} \tag{83}$$

where $t_f = 1$. The C++ code that solves this problem is shown below.

```
//////////////////////////////////////////////////////////////////////////
//////////////////        isoperimetric.cxx       ////////////////////////
//////////////////////////////////////////////////////////////////////////
////////////////           PSOPT  Example             ////////////////////
//////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////
//////// Title:                 Isoperimetric problem     ////////////////
//////// Last modified:         29 January 2009           ////////////////
//////// Reference:                                       ////////////////
////////                                                  ////////////////
//////////////////////////////////////////////////////////////////////////
////////     Copyright (c) Victor M. Becerra, 2009        ////////////////
//////////////////////////////////////////////////////////////////////////
//////// This is part of the PSOPT software library, which ///////////////
//////// is distributed under the terms of the GNU Lesser ////////////////
//////// General Public License (LGPL)                    ////////////////
//////////////////////////////////////////////////////////////////////////

#include "psopt.h"

//////////////////////////////////////////////////////////////////////////
//////////////////  Define the end point (Mayer) cost function //////////
//////////////////////////////////////////////////////////////////////////

adouble endpoint_cost(adouble* initial_states, adouble* final_states,
                      adouble* parameters,adouble& t0, adouble& tf,
                      adouble* xad, int iphase, Workspace* workspace)
{
    return 0.0;
}

//////////////////////////////////////////////////////////////////////////
//////////////////  Define the integrand (Lagrange) cost function  //////
//////////////////////////////////////////////////////////////////////////

adouble integrand_cost(adouble* states, adouble* controls,
                       adouble* parameters, adouble& time, adouble* xad,
                       int iphase, Workspace* workspace)
{
    adouble x = states[0];

    adouble L = x;

    return  L;
}

//////////////////////////////////////////////////////////////////////////
//////////////////  Define the DAE's ////////////////////////////////////
//////////////////////////////////////////////////////////////////////////

void dae(adouble* derivatives, adouble* path, adouble* states,
         adouble* controls, adouble* parameters, adouble& time,
```

```
         adouble* xad, int iphase, Workspace* workspace)
{
   adouble xdot, ydot, vdot;

   adouble x = states[ 0 ];

   adouble u = controls[ 0 ];

   derivatives[0] = -sin(x) + u;

}


////////////////////////////////////////////////////////////////////////////
//////////////////  Define the integrand of the integral constraint ///////
////////////////////////////////////////////////////////////////////////////

adouble integrand( adouble* states, adouble* controls, adouble* parameters,
                     adouble& time, adouble* xad, int iphase, Workspace* workspace)
{
   adouble g;
   adouble u = controls[ 0 ];

   g =  u*u ;

   return g;

}


////////////////////////////////////////////////////////////////////////////
///////////////////  Define the events function ////////////////////////////
////////////////////////////////////////////////////////////////////////////

void events(adouble* e, adouble* initial_states, adouble* final_states,
            adouble* parameters,adouble& t0, adouble& tf, adouble* xad,
            int iphase, Workspace* workspace)
{
   adouble x0 = initial_states[ 0 ];
   adouble xf = final_states[   0 ];
   adouble Q;

   // Compute the integral to be constrained
   Q = integrate( integrand, xad, iphase, workspace );

   e[ 0 ] = x0;
   e[ 1 ] = xf;
   e[ 2 ] = Q;

}



///////////////////////////////////////////////////////////////////////////
//////////////////  Define the phase linkages function ///////////////////
///////////////////////////////////////////////////////////////////////////

void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{
  // No linkages as this is a single phase problem
}


////////////////////////////////////////////////////////////////////////////
//////////////////  Define the main routine ////////////////////////////////
////////////////////////////////////////////////////////////////////////////


int main(void)
{

////////////////////////////////////////////////////////////////////////////
//////////////////  Declare key structures /////////////////////////////////
////////////////////////////////////////////////////////////////////////////

    Alg  algorithm;
    Sol  solution;
    Prob problem;

////////////////////////////////////////////////////////////////////////////
//////////////////  Register problem name  /////////////////////////////////
////////////////////////////////////////////////////////////////////////////
```

```
    problem.name        = "Isoperimetric constraint problem";

    problem.outfilename                = "isoperimetric.txt";

////////////////////////////////////////////////////////////////////////////
////////////  Define problem level constants & do level 1 setup ////////////
////////////////////////////////////////////////////////////////////////////

    problem.nphases          = 1;
    problem.nlinkages               = 0;

    psopt_level1_setup(problem);


/////////////////////////////////////////////////////////////////////////////
/////////   Define phase related information & do level 2 setup /////////////
/////////////////////////////////////////////////////////////////////////////


    problem.phases(1).nstates    = 1;
    problem.phases(1).ncontrols  = 1;
    problem.phases(1).nevents    = 3;
    problem.phases(1).npath      = 0;
    problem.phases(1).nodes << 50;

    psopt_level2_setup(problem, algorithm);



////////////////////////////////////////////////////////////////////////////
///////////////////  Enter problem bounds information //////////////////////
////////////////////////////////////////////////////////////////////////////

    problem.phases(1).bounds.lower.states << -10;
    problem.phases(1).bounds.upper.states << 10;

    problem.phases(1).bounds.lower.controls << -4.0;
    problem.phases(1).bounds.upper.controls <<  4.0;

    problem.phases(1).bounds.lower.events    <<  1.0, 0.0, 10.0;


    problem.phases(1).bounds.upper.events    <<  1.0, 0.0, 10.0;


    problem.phases(1).bounds.lower.StartTime    = 0.0;
    problem.phases(1).bounds.upper.StartTime    = 0.0;

    problem.phases(1).bounds.lower.EndTime      = 1.0;
    problem.phases(1).bounds.upper.EndTime      = 1.0;


////////////////////////////////////////////////////////////////////////////
///////////////////  Register problem functions  ///////////////////////////
////////////////////////////////////////////////////////////////////////////


    problem.integrand_cost                    = &integrand_cost;
    problem.endpoint_cost                       = &endpoint_cost;
    problem.dae                               = &dae;
    problem.events                            = &events;
    problem.linkages                           = &linkages;




////////////////////////////////////////////////////////////////////////////
///////////////////  Define & register initial guess ///////////////////////
////////////////////////////////////////////////////////////////////////////

    problem.phases(1).guess.controls       = zeros(1,30);
    problem.phases(1).guess.states         = zeros(1,30);
    problem.phases(1).guess.time           = linspace(0.0,1.0,30);

////////////////////////////////////////////////////////////////////////////
///////////////////  Enter algorithm options  //////////////////////////////
////////////////////////////////////////////////////////////////////////////


    algorithm.nlp_method                  = "IPOPT";
    algorithm.scaling                     = "automatic";
    algorithm.derivatives                 = "automatic";
```

```
    algorithm.nlp_iter_max              = 1000;
    algorithm.nlp_tolerance             = 1.e-6;

////////////////////////////////////////////////////////////////////////////
//////////////////  Now call PSOPT to solve the problem   ////////////////
////////////////////////////////////////////////////////////////////////////

    psopt(solution, problem, algorithm);

    if (solution.error_flag) exit(0);

////////////////////////////////////////////////////////////////////////////
///////////  Extract relevant variables from solution structure   //////////
////////////////////////////////////////////////////////////////////////////

    MatrixXd x, u, t;
    x  = solution.get_states_in_phase(1);
    u  = solution.get_controls_in_phase(1);
    t       = solution.get_time_in_phase(1);

////////////////////////////////////////////////////////////////////////////
///////////  Save solution data to files if desired ///////////////////////
////////////////////////////////////////////////////////////////////////////

    Save(x,"x.dat");
    Save(u,"u.dat");
    Save(t,"t.dat");


////////////////////////////////////////////////////////////////////////////
///////////  Plot some results if desired (requires gnuplot) //////////////
////////////////////////////////////////////////////////////////////////////

    plot(t,x,problem.name + ": state", "time (s)", "x", "x");

    plot(t,u,problem.name + ": control", "time (s)", "u", "u");

    plot(t,x,problem.name + ": state", "time (s)", "x", "x",
                            "pdf", "isop_state.pdf");

    plot(t,u,problem.name + ": control", "time (s)", "u", "u",
                            "pdf", "isop_control.pdf");



}

////////////////////////////////////////////////////////////////////////////
///////////////////////      END OF FILE     ///////////////////////////////
////////////////////////////////////////////////////////////////////////////
```

The output from $\mathcal{PSOPT}$ is summarized in the text box below and in Figures 45 and 46, which show the optimal state and control, respectively.

```
PSOPT results summary
=====================

Problem:  Isoperimetric constraint problem
CPU time (seconds): 5.948190e-01
NLP solver used:  IPOPT
PSOPT release number:  5.0.3
Date and time of this run:  Thu Mar  6 16:58:16 2025

Optimal (unscaled) cost function value:  -3.755058e-01
Phase 1 endpoint cost function value: 0.000000e+00
```

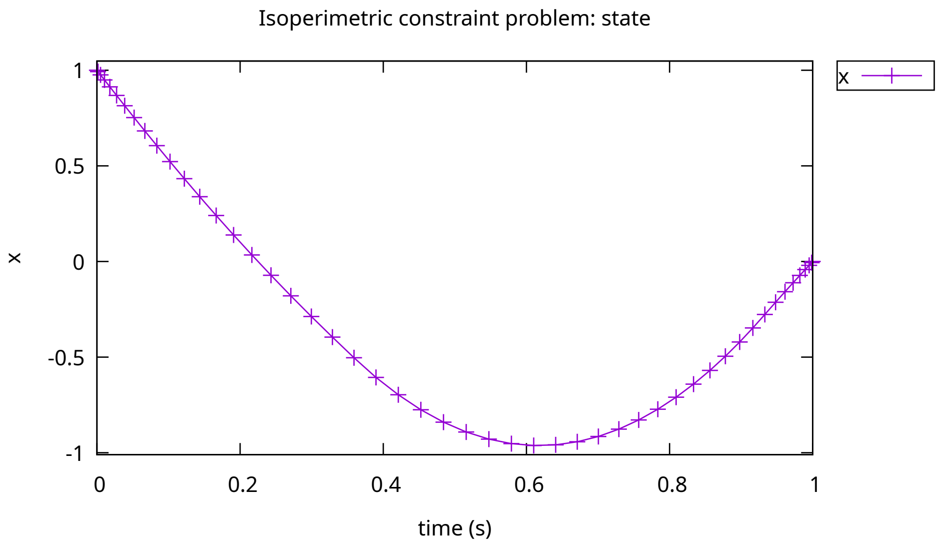

Figure 45: State for isoperimetric constraint problem

```
Phase 1 integrated part of the cost:  -3.755058e-01
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 1.000000e+00
Phase 1 maximum relative local error: 3.106345e-05
NLP solver reports:  The problem has been solved!
```

20 Lambert's problem

This example demonstrates the use of the $\mathcal{PSOPT}$ for a classical orbit determination problem, namely the determination of an orbit from two position vectors and time (Lambert's problem) [23]. The problem is formulated as follows. Find $\mathbf{r}(t) \in [0, t_f]$ and $\mathbf{v}(t) \in [0, t_f]$ to minimise:

$$J = 0 \tag{84}$$

subject to

$$\begin{aligned} \dot{\mathbf{r}} &= \mathbf{v} \\ \dot{\mathbf{v}} &= -\mu \frac{\mathbf{r}}{||\mathbf{r}||^3} \end{aligned} \tag{85}$$

with the boundary conditions:

$$\begin{aligned} \mathbf{r}(\mathbf{0}) &= [15945.34\text{E}3, 0.0, 0.0]^T \\ \mathbf{r}(t_f) &= [12214.83899\text{E}3, 10249.46731\text{E}3, 0.0]^T \end{aligned} \tag{86}$$

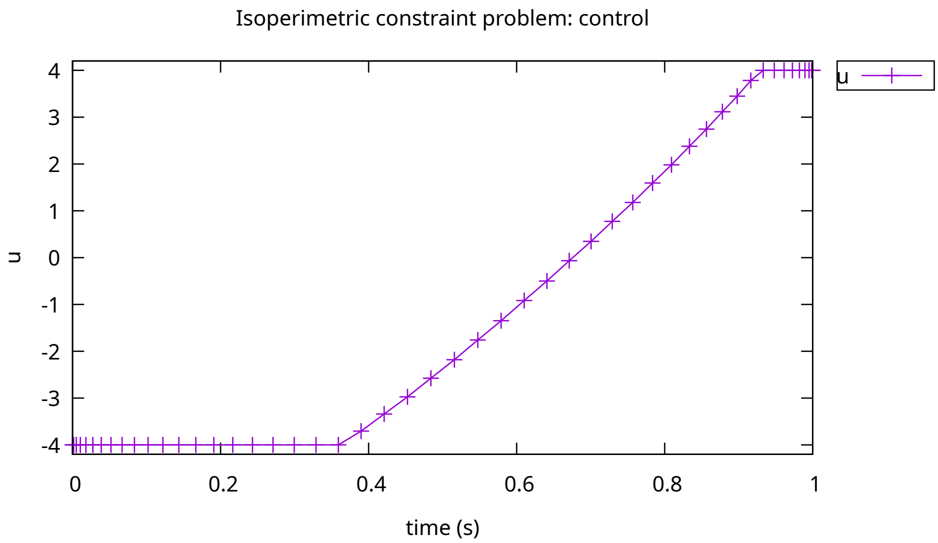

Figure 46: Control for isoperimetric constraint problem

where $\mathbf{r} = [x, y, z]^T$ (m) is a cartesian position vector, and $\mathbf{v} = [v_x, v_z, v_z]^T$ is the corresponding velocity vector, $\mu = GM_e$, G ($\text{m}^3/(\text{kg s}^2)$) is the universal gravitational constant and M_e (kg) is the mass of Earth.

The C++ code that solves this problem is shown below.

```
//////////////////////////////////////////////////////////////////////////
////////////////        lambert.cxx              /////////////////////
//////////////////////////////////////////////////////////////////////////
////////////////           PSOPT  Example             ////////////////////
//////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
//////// Title: Lambert problem                           ////////////////
//////// Last modified: 27 December 2009                  ////////////////
//////// Reference:     Vallado (2001), page 470          ////////////////
//////// (See PSOPT handbook for full reference)           ////////////////
//////////////////////////////////////////////////////////////////////////
////////     Copyright (c) Victor M. Becerra, 2009        ////////////////
//////////////////////////////////////////////////////////////////////////
//////// This is part of the PSOPT software library, which ////////////////
//////// is distributed under the terms of the GNU Lesser ////////////////
//////// General Public License (LGPL)                    ////////////////
//////////////////////////////////////////////////////////////////////////

#include "psopt.h"


//////////////////////////////////////////////////////////////////////////
///////////////////  Define the end point (Mayer) cost function //////////
//////////////////////////////////////////////////////////////////////////

adouble endpoint_cost(adouble* initial_states, adouble* final_states,
                      adouble* parameters,adouble& t0, adouble& tf,
                      adouble* xad, int iphase, Workspace* workspace)
{
   return 0.0;
}

//////////////////////////////////////////////////////////////////////////
///////////////////  Define the integrand (Lagrange) cost function  //////
//////////////////////////////////////////////////////////////////////////
```

```
adouble integrand_cost(adouble* states, adouble* controls, adouble* parameters,
                     adouble& time, adouble* xad, int iphase, Workspace* workspace)
{
    return  0.0;
}


//////////////////////////////////////////////////////////////////////////
///////////////////  Define the DAE's ////////////////////////////////////
//////////////////////////////////////////////////////////////////////////

void dae(adouble* derivatives, adouble* path, adouble* states,
         adouble* controls, adouble* parameters, adouble& time,
         adouble* xad, int iphase, Workspace* workspace)
{

   // Define constants:

   double G =   6.6720e-11; // Universal gravity constant [m^3/(kg s^2)]
   double Me =  5.9742e24;   // Mass of earth;[kg]

   double mu   = G*Me;       // [m^3/sec^2]

   adouble r[3];
   adouble v[3];

   // Extract individual variables

   r[0] = states[ 0 ];
   r[1] = states[ 1 ];
   r[2] = states[ 2 ];

   v[0] = states[ 3 ];
   v[1] = states[ 4 ];
   v[2] = states[ 5 ];


   adouble rdd[3];

   adouble rr = sqrt( r[0]*r[0]+r[1]*r[1]+r[2]*r[2] );

   adouble r3 = pow(rr,3.0);

   rdd[0] = -mu*r[0]/r3;
   rdd[1] = -mu*r[1]/r3;
   rdd[2] = -mu*r[2]/r3;

   derivatives[ 0 ] = v[0];
   derivatives[ 1 ] = v[1];
   derivatives[ 2 ] = v[2];
   derivatives[ 3 ] = rdd[0];
   derivatives[ 4 ] = rdd[1];
   derivatives[ 5 ] = rdd[2];

}

////////////////////////////////////////////////////////////////////////////
///////////////////  Define the events function ////////////////////////////
////////////////////////////////////////////////////////////////////////////

void events(adouble* e, adouble* initial_states, adouble* final_states,
            adouble* parameters,adouble& t0, adouble& tf, adouble* xad,
            int iphase, Workspace* workspace)

{

   adouble ri1 = initial_states[ 0 ];
   adouble ri2 = initial_states[ 1 ];
   adouble ri3 = initial_states[ 2 ];




   adouble rf1 = final_states[ 0 ];
   adouble rf2 = final_states[ 1 ];
   adouble rf3 = final_states[ 2 ];

   e[ 0 ] = ri1;
   e[ 1 ] = ri2;
   e[ 2 ] = ri3;

   e[ 3 ] = rf1;
```

```
   e[ 4 ] = rf2;
   e[ 5 ] = rf3;

}



////////////////////////////////////////////////////////////////////////////
///////////////////  Define the phase linkages function ///////////////////
////////////////////////////////////////////////////////////////////////////

void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{
 //  auto_link_multiple(linkages, xad, N_PHASES);
}



////////////////////////////////////////////////////////////////////////////
///////////////////  Define the main routine ///////////////////////////////
////////////////////////////////////////////////////////////////////////////

int main(void)
{

////////////////////////////////////////////////////////////////////////////
///////////////////  Declare key structures ////////////////////////////////
////////////////////////////////////////////////////////////////////////////

    Alg  algorithm;
    Sol  solution;
    Prob problem;

    MSdata msdata;

////////////////////////////////////////////////////////////////////////////
///////////////////  Register problem name  ////////////////////////////////
////////////////////////////////////////////////////////////////////////////

    problem.name                   = "Lambert problem";
    problem.outfilename                 = "lambert.txt";

////////////////////////////////////////////////////////////////////////////
////////////  Define problem level constants & do level 1 setup ////////////
////////////////////////////////////////////////////////////////////////////

    problem.nphases              = 1;
    problem.nlinkages                   = 0;

    psopt_level1_setup(problem);

/////////////////////////////////////////////////////////////////////////////
/////////   Define phase related information &  do level 2 setup ////////////
/////////////////////////////////////////////////////////////////////////////


    problem.phases(1).nstates    = 6;
    problem.phases(1).ncontrols  = 0;
    problem.phases(1).nevents    = 6;
    problem.phases(1).nparameters   = 6;
    problem.phases(1).npath      = 0;
    problem.phases(1).nodes         << 100;

    psopt_level2_setup(problem, algorithm);


////////////////////////////////////////////////////////////////////////////
///////////////////  Enter problem bounds information //////////////////////
////////////////////////////////////////////////////////////////////////////

    double r1i = 15945.34e3; // m
    double r2i = 0.0;
    double r3i = 0.0;

    double r1f = 12214.83899e3; //m
    double r2f = 10249.46731e3; //m
    double r3f = 0.0;

    double TF = 76.0*60.0; // seconds

    problem.phases(1).bounds.lower.states(0) = -10*max(r1i,r1f);
    problem.phases(1).bounds.lower.states(1) = -10*max(r2i,r2f);
```

```
    problem.phases(1).bounds.lower.states(2) = -10*max(r3i,r3f);
    problem.phases(1).bounds.upper.states(0) = 10*max(r1i,r1f);
    problem.phases(1).bounds.upper.states(1) = 10*max(r2i,r2f);
    problem.phases(1).bounds.upper.states(2) = 10*max(r3i,r3f);

    problem.phases(1).bounds.lower.states(3) = -10*max(r1i,r1f)/TF;
    problem.phases(1).bounds.lower.states(4) = -10*max(r1i,r1f)/TF;;
    problem.phases(1).bounds.lower.states(5) = -10*max(r1i,r1f)/TF;;
    problem.phases(1).bounds.upper.states(3) = 10*max(r1i,r1f)/TF;
    problem.phases(1).bounds.upper.states(4) = 10*max(r2i,r2f)/TF;
    problem.phases(1).bounds.upper.states(5) = 10*max(r3i,r3f)/TF;


    problem.phases(1).bounds.lower.events(0) = r1i;
    problem.phases(1).bounds.upper.events(0) = r1i;

    problem.phases(1).bounds.lower.events(1) = r2i;
    problem.phases(1).bounds.upper.events(1) = r2i;

    problem.phases(1).bounds.lower.events(2) = r3i;
    problem.phases(1).bounds.upper.events(2) = r3i;

    problem.phases(1).bounds.lower.events(3) = r1f;
    problem.phases(1).bounds.upper.events(3) = r1f;

    problem.phases(1).bounds.lower.events(4) = r2f;
    problem.phases(1).bounds.upper.events(4) = r2f;

    problem.phases(1).bounds.lower.events(5) = r3f;
    problem.phases(1).bounds.upper.events(5) = r3f;


    problem.phases(1).bounds.lower.StartTime    = 0.0;
    problem.phases(1).bounds.upper.StartTime    = 0.0;

    problem.phases(1).bounds.lower.EndTime      = TF;
    problem.phases(1).bounds.upper.EndTime      = TF;


////////////////////////////////////////////////////////////////////////////
///////////////////  Enter problem names and units    //////////////////////
////////////////////////////////////////////////////////////////////////////


    problem.phases(1).name.states(1) = "x position";
    problem.phases(1).name.states(2) = "y position";
    problem.phases(1).name.states(3) = "z position";
    problem.phases(1).name.states(4) = "x velocity";
    problem.phases(1).name.states(5) = "y velocity";
    problem.phases(1).name.states(6) = "z velocity";

    problem.phases(1).units.states(1) = "m";
    problem.phases(1).units.states(2) = "m";
    problem.phases(1).units.states(3) = "m";
    problem.phases(1).units.states(4) = "m";
    problem.phases(1).units.states(5) = "m/s";
    problem.phases(1).units.states(6) = "m/s";




////////////////////////////////////////////////////////////////////////////
///////////////////  Register problem functions  ///////////////////////////
////////////////////////////////////////////////////////////////////////////


    problem.integrand_cost  = &integrand_cost;
    problem.endpoint_cost   = &endpoint_cost;
    problem.dae             = &dae;
    problem.events    = &events;
    problem.linkages = &linkages;

////////////////////////////////////////////////////////////////////////////
///////////////////  Define & register initial guess ///////////////////////
////////////////////////////////////////////////////////////////////////////

    int nnodes      = 20;
    int ncontrols                       = problem.phases(1).ncontrols;
    int nstates                         = problem.phases(1).nstates;


    MatrixXd x_guess    =  zeros(nstates,nnodes);
```

```
    MatrixXd time_guess =  linspace(0.0,TF,nnodes);



    x_guess << linspace(r1i,r1f, nnodes),
               linspace(r2i,r2f,nnodes),
      linspace(r3i,r3f,nnodes),
     linspace(r1i,r1f,nnodes)/TF,
     linspace(r2i,r2f,nnodes)/TF,
     linspace(r3i,r3f, nnodes)/TF;


    problem.phases(1).guess.states         = x_guess;
    problem.phases(1).guess.time           = time_guess;


////////////////////////////////////////////////////////////////////////////
///////////////////  Enter algorithm options  //////////////////////////////
////////////////////////////////////////////////////////////////////////////


    algorithm.nlp_iter_max                = 1000;
    algorithm.nlp_tolerance               = 1.e-6;
    algorithm.nlp_method                  = "IPOPT";
    algorithm.scaling                     = "automatic";
    algorithm.derivatives                 = "automatic";
    algorithm.defect_scaling              = "jacobian-based";
    algorithm.collocation_method          = "Hermite-Simpson";




////////////////////////////////////////////////////////////////////////////
///////////////////  Now call PSOPT to solve the problem   //////////////////
////////////////////////////////////////////////////////////////////////////

    psopt(solution, problem, algorithm);

////////////////////////////////////////////////////////////////////////////
///////////  Extract relevant variables from solution structure   //////////
////////////////////////////////////////////////////////////////////////////


    MatrixXd x, u, t, xi, ui, ti;

    x      = solution.get_states_in_phase(1);
    u      = solution.get_controls_in_phase(1);
    t      = solution.get_time_in_phase(1);



////////////////////////////////////////////////////////////////////////////
///////////  Save solution data to files if desired ////////////////////////
////////////////////////////////////////////////////////////////////////////

    Save(x,"x.dat");
    Save(u,"u.dat");
    Save(t,"t.dat");

////////////////////////////////////////////////////////////////////////////
///////////  Plot some results if desired (requires gnuplot) ///////////////
////////////////////////////////////////////////////////////////////////////

    MatrixXd r1 = x.row(0);
    MatrixXd r2 = x.row(1);
    MatrixXd r3 = x.row(2);

    MatrixXd v1 = x.row(3);
    MatrixXd v2 = x.row(4);
    MatrixXd v3 = x.row(5);

    MatrixXd vi(3,1), vf(3,1);

    vi(0) = v1(0);
    vi(1) = v2(1);
    vi(2) = v3(2);

    vf(0) = v1(length(v1)-1);
    vf(1) = v2(length(v1)-1);
    vf(2) = v3(length(v1)-1);

    Print(vi,"Initial velocity vector [m/s]");
```

```
    Print(vf,"Final velocity vector [m/s]");


    plot(t,r1,problem.name+": x", "time (s)", "x","x");

    plot(t,r2,problem.name+": y", "time (s)", "x","y");

    plot(t,r3,problem.name+": z", "time (s)", "z","z");

    plot(r1,r2,problem.name+": x-y", "x (m)", "y (m)","y");

    plot(r1,r2,problem.name+": x-y trajectory", "x (m)", "y (m)","y", "pdf", "lambert_xy.pdf");

    plot3(r1,r2,r3,problem.name+": trajectory","x (m)", "y (m)", "z (m)");


}

////////////////////////////////////////////////////////////////////////////
//////////////////////      END OF FILE     ///////////////////////////////
////////////////////////////////////////////////////////////////////////////
```

The output from $\mathcal{PSOPT}$ is summarized in the text box below and in Figure 47, which show the trajectory from $\mathbf{r}(0)$ to $\mathbf{r}(t_f)$, respectively.

```
PSOPT results summary
=====================

Problem:  Lambert problem
CPU time (seconds): 6.180490e-01
NLP solver used:  IPOPT
PSOPT release number:  5.0.3
Date and time of this run:  Thu Mar  6 16:58:29 2025

Optimal (unscaled) cost function value:  0.000000e+00
Phase 1 endpoint cost function value: 0.000000e+00
Phase 1 integrated part of the cost:  0.000000e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 4.560000e+03
Phase 1 maximum relative local error: 3.433534e-05
NLP solver reports:  The problem has been solved!
```

The resulting initial and final velocity vectors are:

$$\begin{aligned} \mathbf{v}(0) &= [2058.902605, 2915.961924, -6.878790137\text{E}-13]^T \\ \mathbf{v}(t_f) &= [-3451.55505, 910.3192974, -6.878787164\text{E}-13]^T \end{aligned} \tag{87}$$

21 Lee-Ramirez bioreactor

Consider the following optimal control problem, which is known in the literature as the Lee-Ramirez bioreactor [14, 18]. Find t_f and $u(t) \in [0, t_f]$ to minimize the cost functional

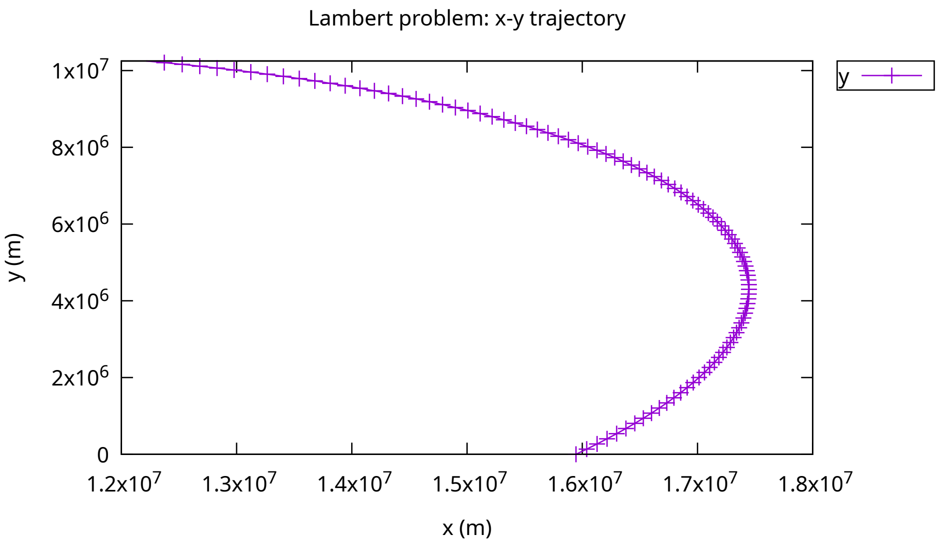

Figure 47: Trajectory between the initial and final positions for Lambert's problem

$$J = -x_1(t_f)x_4(t_f) + \int_0^{t_f} \rho[\dot{u}_1(t)^2 + \dot{u}_2(t)^2]dt \tag{88}$$

subject to the dynamic constraints

$$\begin{aligned}
\dot{x}_1 &= u_1 + u_2; \\
\dot{x}_2 &= g_1 x_2 - \frac{u_1 + u_2}{x_1} x_2; \\
\dot{x}_3 &= 100\frac{u_1}{x_1} - \frac{u_1 + u_2}{x_1} x_3 - (g_1/0.51)x_2; \\
\dot{x}_4 &= R_{fp} x_2 - \frac{u_1 + u_2}{x_1} x_4; \\
\dot{x}_5 &= 4\frac{u_2}{x_1} - \frac{u_1 + u_2}{x_1} x_5; \\
\dot{x}_6 &= -k_1 x_6; \\
\dot{x}_7 &= k_2(1 - x_7).
\end{aligned} \tag{89}$$

where $t_f = 10$, $\rho = 1/N$, and N is the number of discretization nodes,

$$\begin{aligned}
k_1 &= 0.09x_5/(0.034 + x_5); \\
k_2 &= k_1; \\
g_1 &= (x_3/(14.35 + x_3(1.0 + x_3/111.5)))(x_6 + 0.22x_7/(0.22 + x_5)); \\
R_{fp} &= (0.233x_3/(14.35 + x_3(1.0 + x_3/111.5)))((0.0005 + x_5)/(0.022 + x_5));
\end{aligned} \tag{90}$$

the initial conditions:

$$\begin{array}{rcl} x_1(0) & = & 1 \\ x_2(0) & = & 0.1 \\ x_3(0) & = & 40 \\ x_4(0) & = & 0 \\ x_5(0) & = & 0 \\ x_6(0) & = & 1.0 \\ x_7(0) & = & 0 \end{array} \tag{91}$$

The output from $\mathcal{PSOPT}$ is summarised in the box below and shown in Figures 48 and 49, which contain the elements of the state and the control, respectively.

```
PSOPT results summary
=====================

Problem:  Lee-Ramirez bioreactor
CPU time (seconds): 1.358851e+01
NLP solver used:  IPOPT
PSOPT release number:  5.0.3
Date and time of this run:  Thu Mar  6 16:50:29 2025

Optimal (unscaled) cost function value:  -6.163108e+00
Phase 1 endpoint cost function value: -6.166015e+00
Phase 1 integrated part of the cost:  2.906354e-03
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 1.000000e+01
Phase 1 maximum relative local error: 8.608525e-02
NLP solver reports:  The problem has been solved!
```

22 Li's parameter estimation problem

This is a parameter estimation problem with two parameters and three observed variables, which is presented b Li *et. al* [13].

The dynamic equations are given by:

$$\frac{dx}{dt} = M(t,p)x + f(t), \; t \in [0, \pi] \tag{92}$$

with boundary condition:

$$x(0) + x(\pi) = (1 + e^{\pi})\,[1,\,1,\,1]^T$$

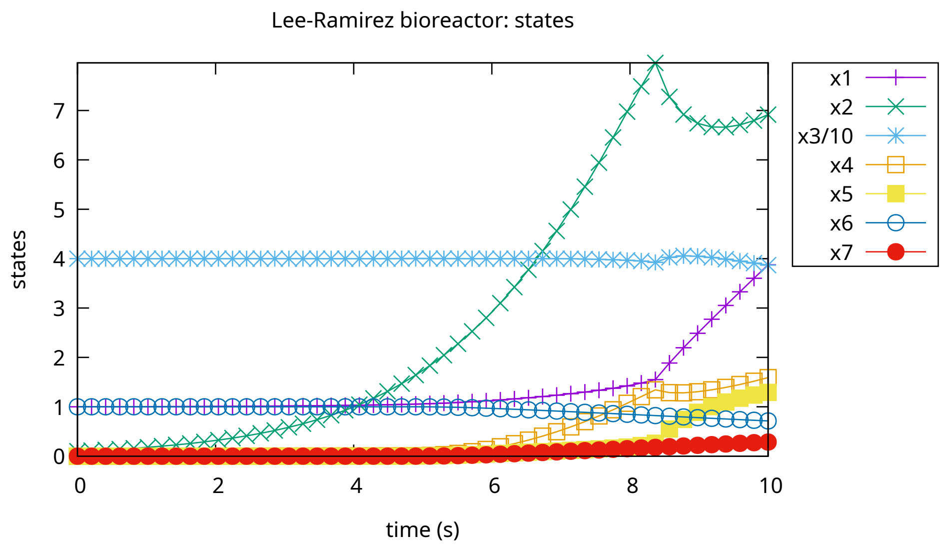

Figure 48: States for the Lee-Ramirez bioreactor problem

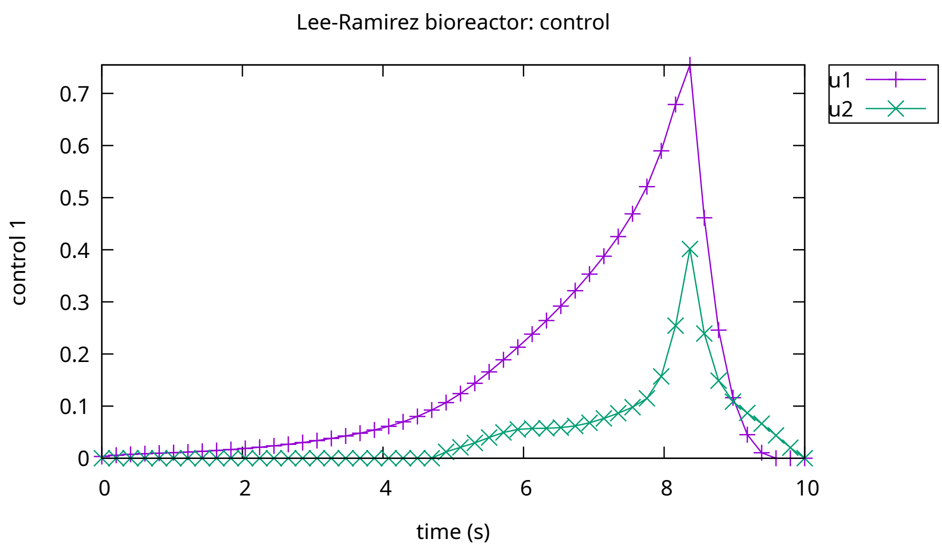

Figure 49: Control for the Lee-Ramirez bioreactor problem

Table 2: Estimated parameter values and 95 percent statistical confidence limits on estimated parameters

Parameter	Low Confidence Limit	Value	High Confidence Limit
p_1	1.907055e+01	1.907712e+01	1.908369e+01
p_2	9.984900e-01	9.984990e-01	9.985080e-01

where

$$M(t,p) = \begin{bmatrix} p_2 - p_1\cos(p_2 t) & 0 & p_2 + p_1\sin(p_2 t) \\ 0 & p_1 & 0 \\ -p_2 + p_1\sin(p_2 t) & 0 & p_2 + p_1\cos(p_2 t) \end{bmatrix} \tag{93}$$

and

$$f(t) = \begin{bmatrix} -1 + 19(\cos(t) - \sin(t)) \\ -18 \\ 1 - 19(\cos(t) + \sin(t)) \end{bmatrix} \tag{94}$$

and the observation functions are:

$$\begin{aligned} g_1 &= x_1 \\ g_2 &= x_2 \\ g_3 &= x_3 \end{aligned} \tag{95}$$

The trajectories of the dynamic system is characterised by rapidly varying fast and slow components if the difference between the two parameters p_1 and p_2 is large, which may cause numerical problems to some ODE solvers.

The estimation data set is generated by adding Gaussian noise with standard deviation 1 around the solution $[x_1(t), x_2(t), x_3(t)]^T = [e^t,\, e^t,\, e^t]^T$, with $N = 33$ equidistant samples within the interval $t = [0,\, \pi]$. The true values of the parameters are $p_1 = 19$ and $p_2 = 1$. The weights of the three observations are the same and equal to one.

The solution is found using Legendre discretisation with 40 grid points. The estimated parameter values and their 95% confidence limits for $n_s = 129$ samples are shown in Table 22. Figure 50 shows the observations as well as the estimated values of variable x_1.

23 Linear tangent steering problem

Consider the following optimal control problem, which is known in the literature as the linear tangent steering problem [3]. Find t_f and $u(t) \in [0, t_f]$ to minimize the cost functional

$$J = t_f \tag{96}$$

subject to the dynamic constraints

$$\begin{aligned} \dot{x}_1 &= x_2 \\ \dot{x}_2 &= a\cos(u) \\ \dot{x}_3 &= x_4 \\ \dot{x}_4 &= a\sin(u) \end{aligned} \tag{97}$$

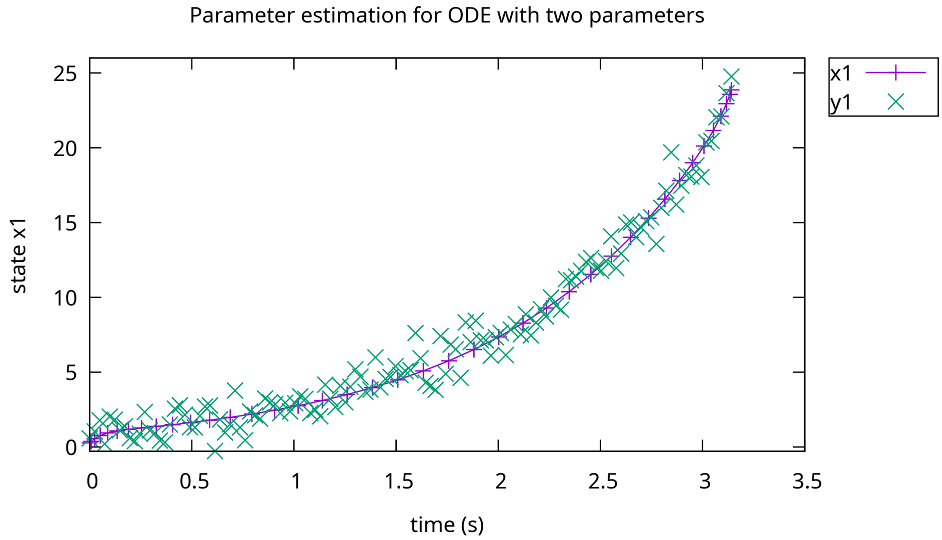

Figure 50: Observations and estimated state $x_1(t)$

the boundary conditions:

$$\begin{aligned}
x_1(0) &= 0 \\
x_2(0) &= 0 \\
x_3(0) &= 0 \\
x_4(0) &= 0 \\
x_2(t_f) &= 45.0 \\
x_3(t_f) &= 5.0 \\
x_4(t_f) &= 0.0
\end{aligned} \tag{98}$$

The output from $\mathcal{PSOPT}$ is summarised in the box below and shown in Figures 51 and 52, which contain the elements of the state and the control, respectively.

```
PSOPT results summary
=====================

Problem:  Linear Tangent Steering Problem
CPU time (seconds): 2.376000e-01
NLP solver used:  IPOPT
PSOPT release number:  5.0.3
Date and time of this run:  Thu Mar  6 16:59:26 2025

Optimal (unscaled) cost function value:  5.545709e-01
Phase 1 endpoint cost function value: 5.545709e-01
```

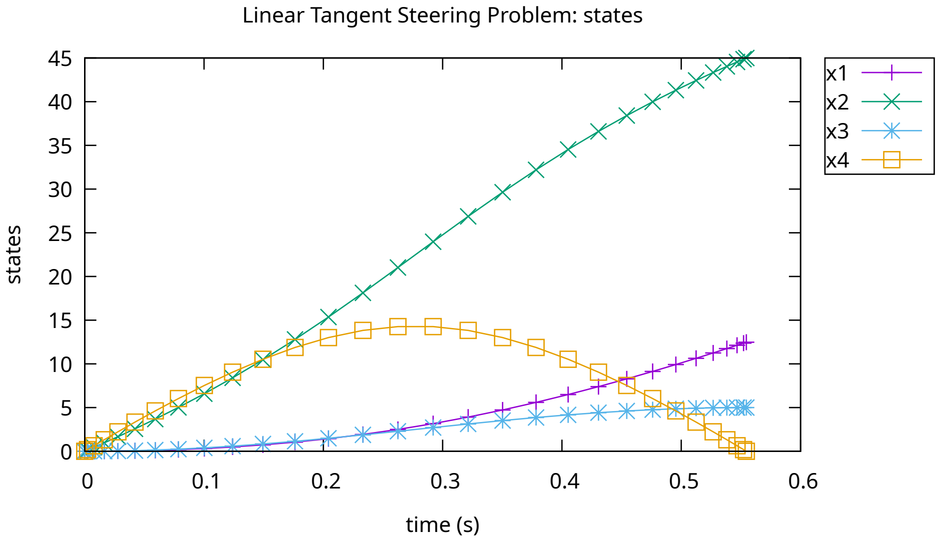

Figure 51: States for the linear tangent steering problem

```
Phase 1 integrated part of the cost:  0.000000e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 5.545709e-01
Phase 1 maximum relative local error: 1.890010e-07
NLP solver reports:  The problem has been solved!
```

24 Low thrust orbit transfer

The goal of this problem is to compute an optimal low thrust policy for an spacecraft to go from a standard space shuttle park orbit to a specified final orbit, while maximising the final weight of the spacecraft. The problem is described in detail by Betts [3]. The problem is formulated as follows. Find $\mathbf{u}(t) = [u_r(t), u_\theta(t), u_h(t)]^T, t \in [0, t_f]$, the unknown throtle parameter τ, and the final time t_f, such that the following objective function is minimised:

$$J = -w(t_f) \tag{99}$$

subject to the dynamic constraints:

$$\begin{aligned} \dot{\mathbf{y}} &= \mathbf{A}(\mathbf{y})\Delta + \mathbf{b} \\ \dot{w} &= -T[1 + 0.01\tau]/I_{sp} \end{aligned} \tag{100}$$

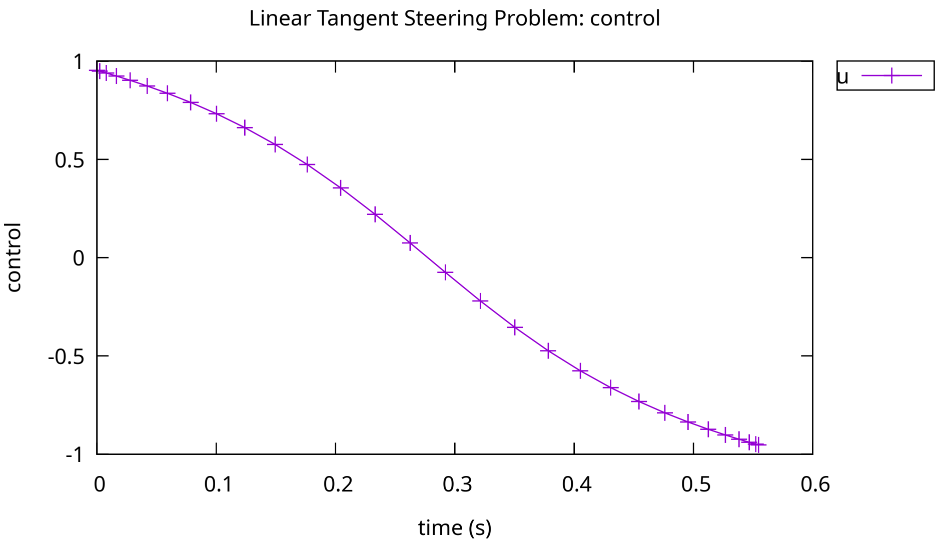

Figure 52: Control for the linear tangent steering problem

the path constraint:

$$||u(t)||^2 = 1 \tag{101}$$

and the parameter bounds:

$$\tau_L \leq \tau \leq 0 \tag{102}$$

where $\mathbf{y} = [p, f, g, h, k, L, w]^T$ is the vector of modified equinoctial elements, $w(t)$ is the weight of the spacecraft, I_{sp} is the specific impulse of the engine, expressions for $\mathbf{A}(\mathbf{y})$ and $\mathbf{b}$ are given in [3], the disturbing acceleration Δ is given by:

$$\Delta = \Delta_g + \Delta_T \tag{103}$$

where Δ_g is the gravitational disturbing acceleration due to the oblatness of Earth (given in [3]), and Δ_T is the thurst acceleration, given by:

$$\Delta_T = \frac{g_0 T[1 + 0.01\tau]}{w}\mathbf{u}$$

where T is the maximum thrust, and g_0 is the mass to weight conversion factor.

The boundary conditions of the problem are given by:

$$\begin{aligned}
p(t_f) &= 40007346.015232 \text{ ft} \\
\sqrt{f(t_f)^2 + g(t_f)^2} &= 0.73550320568829 \\
\sqrt{h(t_f)^2 + k(t_f)^2} &= 0.61761258786099 \\
f(t_f)h(t_f) + g(t_f)k(t_f) &= 0 \\
g(t_f)h(t_f) - k(t_f)f(t_f) &= 0 \\
p(0) &= 21837080.052835\text{ft} \\
f(0) &= 0 \\
g(0) &= 0 \\
h(0) &= 0 \\
h(0) &= 0 \\
k(0) &= 0 \\
L(0) &= \pi \text{ (rad)} \\
w(0) &= 1 \text{ (lb)}
\end{aligned} \tag{104}$$

and the values of the parameters are: $g_0 = 32.174$ (ft/sec^2), $I_{sp} = 450$ (sec), $T = 4.446618 \times 10^{-3}$ (lb), $\mu = 1.407645794 \times 10^{16}$ (ft^3/sec^2), $R_e = 20925662.73$ (ft), $J_2 = 1082.639 \times 10^{-6}$, $J_3 = -2.565 \times 10^{-6}$, $J_4 = -1.608 \times 10^{-6}$, $\tau_L = -50$.

An initial guess was computed by forward propagation from the initial conditions, assuming that the direction of the thrust vector is parallel to the cartesian velocity vector, such that the initial control input was computed as follows:

$$\mathbf{u}(t) = \mathbf{Q}_r^T \frac{\mathbf{v}}{||\mathbf{v}||} \tag{105}$$

where $\mathbf{Q}_r$ is a matrix whose columns are the directions of the rotating radial frame:

$$\mathbf{Q}_r = \begin{bmatrix} \mathbf{i}_r & \mathbf{i}_\theta & \mathbf{i}_h \end{bmatrix} = \begin{bmatrix} \frac{\mathbf{r}}{||\mathbf{r}||} & \frac{(\mathbf{r}\times\mathbf{v})\times\mathbf{r}}{||\mathbf{r}\times\mathbf{v}||||\mathbf{r}||} & \frac{(\mathbf{r}\times\mathbf{v})}{||\mathbf{r}\times\mathbf{v}||} \end{bmatrix} \tag{106}$$

The problem was solved using local collocation (trapezoidal followed by Hermite-Simpson) with automatic mesh refinement. The C++ code that solves the problem is shown below.

```
//////////////////////////////////////////////////////////////////////////
////////////////        low_thrust.cxx              /////////////////////
//////////////////////////////////////////////////////////////////////////
////////////////           PSOPT  Example            ////////////////////
//////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////
//////// Title: Low thrust orbit transfer problem         ////////////////
//////// Last modified: 16 February 2009                  ////////////////
//////// Reference:     Betts  (2001)                 ////////////////
//////// (See PSOPT handbook for full reference)           ///////////////
//////////////////////////////////////////////////////////////////////////
////////     Copyright (c) Victor M. Becerra, 2009        ////////////////
//////////////////////////////////////////////////////////////////////////
```

```
//////// This is part of the PSOPT software library, which ///////////////
//////// is distributed under the terms of the GNU Lesser ///////////////
//////// General Public License (LGPL)                    ///////////////
////////////////////////////////////////////////////////////////////////////

#include "psopt.h"

using namespace PSOPT;



////////////////////////////////////////////////////////////////////////////
///////////////////  Define auxiliary functions                 //////////
////////////////////////////////////////////////////////////////////////////

adouble legendre_polynomial( adouble x, int n)
{
// This function computes the value of the legendre polynomials
// for a given value of the argument x and for n=0...5 only

  adouble retval=0.0;

  switch(n) {
     case 0:
         retval=1.0; break;
     case 1:
         retval= x; break;
     case 2:
         retval= 0.5*(3.0*pow(x,2)-1.0); break;
     case 3:
         retval= 0.5*(5.0*pow(x,3)- 3*x); break;
     case 4:
         retval= (1.0/8.0)*(35.0*pow(x,4) - 30.0*pow(x,2) + 3.0); break;
     case 5:
         retval= (1.0/8.0)*(63.0*pow(x,5) - 70.0*pow(x,3) + 15.0*x); break;
     default:
         error_message("legendre_polynomial(x,n) is limited to n=0...5");
  }

  return retval;

}

adouble legendre_polynomial_derivative( adouble x, int n)
{
// This function computes the value of the legendre polynomial derivatives
// for a given value of the argument x and for n=0...5 only.

  adouble retval=0.0;

  switch(n) {
     case 0:
         retval=0.0; break;
     case 1:
         retval= 1.0; break;
     case 2:
         retval= 0.5*(2.0*3.0*x); break;
     case 3:
         retval= 0.5*(3.0*5.0*pow(x,2)-3.0); break;
     case 4:
         retval= (1.0/8.0)*(4.0*35.0*pow(x,3) - 2.0*30.0*x ); break;
     case 5:
         retval= (1.0/8.0)*(5.0*63.0*pow(x,4) - 3.0*70.0*pow(x,2) + 15.0); break;
     default:
         error_message("legendre_polynomial_derivative(x,n) is limited to n=0...5");
  }

  return retval;

}


void compute_cartesian_trajectory(const MatrixXd& x, MatrixXd& xyz )
{

   int npoints = x.cols();
   xyz.resize(3,npoints);

   for(int i=0; i<npoints;i++) {


   double p = x(0,i);
```

```
   double f = x(1,i);
   double g = x(2,i);
   double h = x(3,i);
   double k = x(4,i);
   double L = x(5,i);

   double q      =  1.0 + f*cos(L) + g*sin(L);
   double r      =  p/q;
   double alpha2 = h*h - k*k;
   double X      = sqrt( h*h + k*k );
   double s2     = 1 + X*X;

   double r1 = r/s2*( cos(L) + alpha2*cos(L) + 2*h*k*sin(L));
   double r2 = r/s2*( sin(L) - alpha2*sin(L) + 2*h*k*cos(L));
   double r3 = 2*r/s2*( h*sin(L) - k*cos(L) );

   xyz(0,i) = r1;
   xyz(1,i) = r2;
   xyz(2,i) = r3;

   }

}

//////////////////////////////////////////////////////////////////////////
///////////////////  Define the end point (Mayer) cost function //////////
//////////////////////////////////////////////////////////////////////////

adouble endpoint_cost(adouble* initial_states, adouble* final_states,
                      adouble* parameters,adouble& t0, adouble& tf,
                      adouble* xad, int iphase, Workspace* workspace)
{

   if (iphase == 1) {
   adouble w = final_states[6];
   return (-w);
   }
   else {
   return (0);
   }
}

//////////////////////////////////////////////////////////////////////////
///////////////////  Define the integrand (Lagrange) cost function  //////
//////////////////////////////////////////////////////////////////////////

adouble integrand_cost(adouble* states, adouble* controls, adouble* parameters,
                     adouble& time, adouble* xad, int iphase, Workspace* workspace)
{
    return  0.0;
}


//////////////////////////////////////////////////////////////////////////
///////////////////  Define the DAE's ////////////////////////////////////
//////////////////////////////////////////////////////////////////////////

void dae(adouble* derivatives, adouble* path, adouble* states,
         adouble* controls, adouble* parameters, adouble& time,
         adouble* xad, int iphase, Workspace* workspace)
{
   // Local integers
   int i, j;
   // Define constants:
   double Isp  = 450.0;            // [sec]
   double mu   = 1.407645794e16;   // [f2^2/sec^2]
   double g0   = 32.174;           // [ft/sec^2]
   double T    = 4.446618e-3;      // [lb]
   double Re   = 20925662.73;      // [ft]
   double J[5];
   J[2] =  1082.639e-6;
   J[3] = -2.565e-6;
   J[4] = -1.608e-6;

   // Extract individual variables

   adouble p = states[ 0 ];
   adouble f = states[ 1 ];
   adouble g = states[ 2 ];
   adouble h = states[ 3 ];
   adouble k = states[ 4 ];
   adouble L = states[ 5 ];
```

```
adouble w = states[ 6 ];

adouble* u  = controls;

adouble tau  = parameters[ 0 ];

// Define some dependent variables

adouble q      =  1.0 + f*cos(L) + g*sin(L);
adouble r      =  p/q;
adouble alpha2 = h*h - k*k;
adouble X      = sqrt( h*h + k*k );
adouble s2     = 1 + X*X;

// r and v

adouble r1 = r/s2*( cos(L) + alpha2*cos(L) + 2*h*k*sin(L));
adouble r2 = r/s2*( sin(L) - alpha2*sin(L) + 2*h*k*cos(L));
adouble r3 = 2*r/s2*( h*sin(L) - k*cos(L) );

adouble rvec[3];

rvec[ 0 ] = r1; rvec[ 1] = r2; rvec[ 2 ] = r3;

adouble v1 = -(1.0/s2)*sqrt(mu/p)*(  sin(L) + alpha2*sin(L) - 2*h*k*cos(L) + g - 2*f*h*k + alpha2*g);
adouble v2 = -(1.0/s2)*sqrt(mu/p)*( -cos(L) + alpha2*cos(L) + 2*h*k*sin(L) - f + 2*g*h*k + alpha2*f);
adouble v3 =  (2.0/s2)*sqrt(mu/p)*(h*cos(L) +      k*sin(L) + f*h + g*k);

adouble vvec[3];

vvec[ 0 ] = v1; vvec[ 1 ] = v2; vvec[ 2 ] = v3;

// compute Qr

adouble ir[3], ith[3], ih[3];
adouble rv[3];
adouble rvr[3];

cross( rvec, vvec, rv );

cross( rv, rvec, rvr );

adouble norm_r = sqrt( dot(rvec, rvec, 3) );

adouble norm_rv = sqrt( dot(rv, rv, 3) );

for (i=0; i<3; i++) {

   ir[i]  = rvec[i]/norm_r;

   ith[i] = rvr[i]/( norm_rv*norm_r );

   ih[i]  = rv[i]/norm_rv;

}

adouble Qr1[3], Qr2[3], Qr3[3];

for(i=0; i< 3; i++)
{
     // Columns of matrix Qr
     Qr1[i] = ir[i];
     Qr2[i] = ith[i];
     Qr3[i] = ih[i];
}

// Compute in

adouble en[3];
en[ 0 ] = 0.0; en[ 1 ]= 0.0; en[ 2 ] = 1.0;

adouble enir = dot(en,ir,3);

adouble in[3];

for(i=0;i<3;i++) {
   in[i] = en[i] - enir*ir[i];
}

adouble norm_in = sqrt( dot( in, in, 3 ) );

for(i=0;i<3;i++) {
```

```
      in[i] = in[i]/norm_in;
   }

   // Geocentric latitude angle:

   adouble sin_phi =  rvec[ 2 ]/ sqrt( dot(rvec,rvec,3) ) ;
   adouble cos_phi =  sqrt(1.0- pow(sin_phi,2.0));

   adouble deltagn = 0.0;
   adouble deltagr = 0.0;
   for (j=2; j<=4;j++) {
     adouble Pdash_j = legendre_polynomial_derivative( sin_phi, j );
     adouble P_j     = legendre_polynomial( sin_phi, j );
     deltagn += -mu*cos_phi/(r*r)*pow(Re/r,j)*Pdash_j*J[j];
     deltagr += -mu/(r*r)* (j+1)*pow( Re/r,j)*P_j*J[j];
   }

   // Compute vector delta_g

   adouble delta_g[3];
   for (i=0;i<3;i++) {
       delta_g[i] = deltagn*in[i] - deltagr*ir[i];
   }

   // Compute vector DELTA_g

   adouble DELTA_g[3];

   DELTA_g[ 0 ] = dot(Qr1, delta_g,3);
   DELTA_g[ 1 ] = dot(Qr2, delta_g,3);
   DELTA_g[ 2 ] = dot(Qr3, delta_g,3);

   // Compute DELTA_T

   adouble DELTA_T[3];



   for(i=0;i<3;i++) {
      DELTA_T[i] =  g0*T*(1.0+0.01*tau)*u[i]/w;
   }

   // Compute DELTA

   adouble DELTA[3];

   for(i=0;i<3;i++) {
      DELTA[i] =  DELTA_g[i] + DELTA_T[i];
   }

   adouble delta1= DELTA[ 0 ];
   adouble delta2= DELTA[ 1 ];
   adouble delta3= DELTA[ 2 ];


  // derivatives

   adouble pdot =                                      2*p/q*sqrt(p/mu)               * delta2;
   adouble fdot =  sqrt(p/mu)*sin(L) * delta1 + sqrt(p/mu)*(1.0/q)*((q+1.0)*cos(L)+f) * delta2
                 -  sqrt(p/mu)*(g/q)*(h*sin(L)-k*cos(L))  * delta3;
   adouble gdot = -sqrt(p/mu)*cos(L) * delta1 + sqrt(p/mu)*(1.0/q)*((q+1.0)*sin(L)+g) * delta2
                 +  sqrt(p/mu)*(f/q)*(h*sin(L)-k*cos(L))  * delta3;
   adouble hdot =  sqrt(p/mu)*s2*cos(L)/(2.0*q)             * delta3;
   adouble kdot =  sqrt(p/mu)*s2*sin(L)/(2.0*q)             * delta3;
   adouble Ldot =  sqrt(p/mu)*(1.0/q)*(h*sin(L)-k*cos(L))* delta3     + sqrt(mu*p)*pow( (q/p),2.);

   adouble wdot = -T*(1.0+0.01*tau)/Isp;

   derivatives[ 0 ] = pdot;
   derivatives[ 1 ] = fdot;
   derivatives[ 2 ] = gdot;
   derivatives[ 3 ] = hdot;
   derivatives[ 4 ] = kdot;
   derivatives[ 5 ] = Ldot;
   derivatives[ 6 ] = wdot;

   path[ 0 ] = pow( u[0] , 2)  + pow( u[1], 2) + pow( u[2], 2);

}

////////////////////////////////////////////////////////////////////////////
///////////////////  Define the events function ////////////////////////////
```

```
////////////////////////////////////////////////////////////////////////////

void events(adouble* e, adouble* initial_states, adouble* final_states,
            adouble* parameters,adouble& t0, adouble& tf, adouble* xad,
            int iphase, Workspace* workspace)

{


   int offset;

   adouble pti = initial_states[ 0 ];
   adouble fti = initial_states[ 1 ];
   adouble gti = initial_states[ 2 ];
   adouble hti = initial_states[ 3 ];
   adouble kti = initial_states[ 4 ];
   adouble Lti = initial_states[ 5 ];
   adouble wti = initial_states[ 6 ];


   adouble ptf = final_states[ 0 ];
   adouble ftf = final_states[ 1 ];
   adouble gtf = final_states[ 2 ];
   adouble htf = final_states[ 3 ];
   adouble ktf = final_states[ 4 ];
   adouble Ltf = final_states[ 5 ];


   if (iphase==1) {
    e[ 0 ]  = pti;
    e[ 1 ]  = fti;
    e[ 2 ]  = gti;
    e[ 3 ]  = hti;
    e[ 4 ]  = kti;
    e[ 5 ]  = Lti;
    e[ 6 ]  = wti;
   }

   if (1 == 1) offset = 7;
   else offset = 0;

   if (iphase == 1 ) {
    e[ offset + 0 ]  = ptf;
    e[ offset + 1 ]  = sqrt( ftf*ftf + gtf*gtf );
    e[ offset + 2 ]  = sqrt( htf*htf + ktf*ktf );
    e[ offset + 3 ] = ftf*htf + gtf*ktf;
    e[ offset + 4 ] = gtf*htf - ktf*ftf;
   }

}



///////////////////////////////////////////////////////////////////////////
//////////////////  Define the phase linkages function ///////////////////
///////////////////////////////////////////////////////////////////////////

void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{
 //  auto_link_multiple(linkages, xad, 1);
}



////////////////////////////////////////////////////////////////////////////
///////////////////  Define the main routine ///////////////////////////////
////////////////////////////////////////////////////////////////////////////

int main(void)
{

////////////////////////////////////////////////////////////////////////////
///////////////////  Declare key structures ////////////////////////////////
////////////////////////////////////////////////////////////////////////////

    Alg  algorithm;
    Sol  solution;
    Prob problem;

    MSdata msdata;

////////////////////////////////////////////////////////////////////////////
```

```
////////////////////  Register problem name  ////////////////////////////////
////////////////////////////////////////////////////////////////////////////

    problem.name          = "Low thrust transfer problem";
    problem.outfilename                 = "lowthrust.txt";

////////////////////////////////////////////////////////////////////////////
////////////  Define problem level constants & do level 1 setup ////////////
////////////////////////////////////////////////////////////////////////////

    problem.nphases             = 1;
    problem.nlinkages                   = 0;

    psopt_level1_setup(problem);

/////////////////////////////////////////////////////////////////////////////
/////////   Define phase related information & do level 2 setup  ////////////
/////////////////////////////////////////////////////////////////////////////

    problem.phases(1).nstates    = 7;
    problem.phases(1).ncontrols  = 3;
    problem.phases(1).nparameters         = 1;
    problem.phases(1).nevents             = 12;
    problem.phases(1).npath      = 1;
    problem.phases(1).nodes               << 80;

    psopt_level2_setup(problem, algorithm);


////////////////////////////////////////////////////////////////////////////
///////////////////  Enter problem bounds information //////////////////////
////////////////////////////////////////////////////////////////////////////

    double tauL = -50.0;
    double tauU =   0.0;


    double pti = 21837080.052835;
    double fti = 0.0;
    double gti = 0.0;
    double hti = -0.25396764647494;
    double kti = 0.0;
    double Lti = pi;
    double wti = 1.0;

    double wtf_guess;

    double  SISP = 450.0;
    double  DELTAV = 22741.1460;
    double  CM2W = 32.174;

    wtf_guess  = wti*exp(-DELTAV/(CM2W*SISP));

    double ptf            = 40007346.015232;
    double event_final_9  = 0.73550320568829;
    double event_final_10  = 0.61761258786099;
    double event_final_11 = 0.0;
    double event_final_12_upper = 0.0;
    double event_final_12_lower = -10.0;



    problem.phases(1).bounds.lower.parameters <<  tauL;

    problem.phases(1).bounds.upper.parameters <<  tauU;

    problem.phases(1).bounds.lower.states << 10.e6, -0.20, -0.10, -1.0, -0.20, pi, 0.0;

    problem.phases(1).bounds.upper.states << 60.e6, 0.20, 1.0, 1.0, 0.20, 20*pi, 2.0;

    problem.phases(1).bounds.lower.controls << -1.0, -1.0, -1.0;

    problem.phases(1).bounds.upper.controls << 1.0, 1.0, 1.0;

    problem.phases(1).bounds.lower.events << pti, fti, gti, hti, kti, Lti, wti, ptf, event_final_9, event_final_10, event_final_11, event_final_12_lower;

    problem.phases(1).bounds.upper.events << pti, fti, gti, hti, kti, Lti, wti, ptf, event_final_9, event_final_10, event_final_11, event_final_12_upper;

    double EQ_TOL = 0.001;

    problem.phases(1).bounds.upper.path << 1.0+EQ_TOL;
    problem.phases(1).bounds.lower.path << 1.0-EQ_TOL;
```

```
    problem.phases(1).bounds.lower.StartTime    = 0.0;
    problem.phases(1).bounds.upper.StartTime    = 0.0;

    problem.phases(1).bounds.lower.EndTime      = 50000.0;
    problem.phases(1).bounds.upper.EndTime      = 100000.0;


////////////////////////////////////////////////////////////////////////////
///////////////////  Register problem functions  ///////////////////////////
////////////////////////////////////////////////////////////////////////////


    problem.integrand_cost  = &integrand_cost;
    problem.endpoint_cost  = &endpoint_cost;
    problem.dae             = &dae;
    problem.events  = &events;
    problem.linkages = &linkages;

////////////////////////////////////////////////////////////////////////////
///////////////////  Define & register initial guess ///////////////////////
////////////////////////////////////////////////////////////////////////////

    int nnodes     = 141;
    int ncontrols          = problem.phases(1).ncontrols;
    int nstates            = problem.phases(1).nstates;

    MatrixXd u_guess    =  zeros(ncontrols,nnodes);
    MatrixXd x_guess    =  zeros(nstates,nnodes);
    MatrixXd time_guess =  linspace(0.0,86810.0,nnodes);

    MatrixXd param_guess = -25.0*ones(1,1);

    u_guess    = load_data("../../../examples/low_thrust/U0.dat",ncontrols, nnodes );
    x_guess    = load_data("../../../examples/low_thrust/X0.dat",nstates  , nnodes );
    time_guess = load_data("../../../examples/low_thrust/T0.dat",1        , nnodes );

    auto_phase_guess(problem, u_guess, x_guess, param_guess, time_guess);


////////////////////////////////////////////////////////////////////////////
///////////////////  Enter algorithm options  //////////////////////////////
////////////////////////////////////////////////////////////////////////////


    algorithm.nlp_iter_max                = 1000;
    algorithm.nlp_tolerance               = 1.e-6;
    algorithm.nlp_method                  = "IPOPT";
    algorithm.scaling                     = "automatic";
    algorithm.derivatives                 = "automatic";
    algorithm.defect_scaling              = "jacobian-based";
    algorithm.jac_sparsity_ratio          =  0.11; // 0.05;
    algorithm.collocation_method          = "trapezoidal";
    algorithm.mesh_refinement             = "automatic";
    algorithm.mr_max_increment_factor     = 0.2;




////////////////////////////////////////////////////////////////////////////
///////////////////  Now call PSOPT to solve the problem   //////////////////
////////////////////////////////////////////////////////////////////////////

    psopt(solution, problem, algorithm);

////////////////////////////////////////////////////////////////////////////
///////////  Extract relevant variables from solution structure   //////////
////////////////////////////////////////////////////////////////////////////


    MatrixXd x, u, t;

    x      = solution.get_states_in_phase(1);
    u      = solution.get_controls_in_phase(1);
    t      = solution.get_time_in_phase(1);


    t = t/3600.0;


    MatrixXd tau = solution.get_parameters_in_phase(1);
```

```
    Print(tau,"tau");

////////////////////////////////////////////////////////////////////////////
///////////  Save solution data to files if desired ////////////////////////
////////////////////////////////////////////////////////////////////////////

    Save(x,"x.dat");
    Save(u,"u.dat");
    Save(t,"t.dat");

////////////////////////////////////////////////////////////////////////////
///////////  Plot some results if desired (requires gnuplot) ///////////////
////////////////////////////////////////////////////////////////////////////

    MatrixXd x1 = x.row(0)/1.e6;
    MatrixXd x2 = x.row(1);
    MatrixXd x3 = x.row(2);
    MatrixXd x4 = x.row(3);
    MatrixXd x5 = x.row(4);
    MatrixXd x6 = x.row(5);
    MatrixXd x7 = x.row(6);
    MatrixXd u1 = u.row(0);
    MatrixXd u2 = u.row(1);
    MatrixXd u3 = u.row(2);

    plot(t,x1,problem.name+": states", "time (h)", "p (1000000 ft)","p (1000000 ft)");

    plot(t,x2,problem.name+": states", "time (h)", "f","f");

    plot(t,x3,problem.name+": states", "time (h)", "g","g");

    plot(t,x4,problem.name+": states", "time (h)", "h","h");

    plot(t,x5,problem.name+": states", "time (h)", "k","k");

    plot(t,x6,problem.name+": states", "time (h)", "L (rev)","L (rev)");

    plot(t,x7,problem.name+": states", "time (h)", "w (lb)","w (lb)");

    plot(t,u1,problem.name+": controls","time (h)", "ur", "ur");

    plot(t,u2,problem.name+": controls","time (h)", "ut", "ut");

    plot(t,u3,problem.name+": controls","time (h)", "uh", "uh");

    plot(t,x1,problem.name+": states", "time (h)", "p (1000000 ft)","p (1000000 ft)",
         "pdf","lowthr_x1.pdf");

    plot(t,x2,problem.name+": states", "time (h)", "f","f",
   "pdf","lowthr_x2.pdf");

    plot(t,x3,problem.name+": states", "time (h)", "g","g",
   "pdf","lowthr_x3.pdf");

    plot(t,x4,problem.name+": states", "time (h)", "h","h",
   "pdf","lowthr_x4.pdf");

    plot(t,x5,problem.name+": states", "time (h)", "k","k",
             "pdf","lowthr_x5.pdf");

    plot(t,x6,problem.name+": states", "time (h)", "L (rev)","L (rev)",
   "pdf","lowthr_x6.pdf");

    plot(t,x7,problem.name+": states", "time (h)", "w (lb)","w (lb)",
   "pdf","lowthr_x7.pdf");


    plot(t,u1,problem.name+": controls","time (h)", "ur", "ur",
   "pdf","lowthr_u1.pdf");

    plot(t,u2,problem.name+": controls","time (h)", "ut", "ut",
   "pdf","lowthr_u2.pdf");

    plot(t,u3,problem.name+": controls","time (h)", "uh", "uh",
   "pdf","lowthr_u3.pdf");

    MatrixXd r;

    compute_cartesian_trajectory(x,r);

    double ft2km = 0.0003048;
```

```
    r = r*ft2km;

    MatrixXd rnew, tnew;

    tnew = linspace(0.0, t(length(t)-1), 1000);

    resample_trajectory( rnew, tnew, r, t );


    plot3(rnew.row(0), rnew.row(1), rnew.row(2),
        "Low thrust transfer trajectory", "x (km)", "y (km)", "z (km)",
         NULL, NULL, "30,97");

    plot3(rnew.row(0), rnew.row(1), rnew.row(2),
       "Low thrust transfer trajectory", "x (km)", "y (km)", "z (km)",
       "pdf", "trajectory.pdf", "30,97");


}

////////////////////////////////////////////////////////////////////////////
///////////////////////      END OF FILE     ///////////////////////////////
////////////////////////////////////////////////////////////////////////////
```

The output from $\mathcal{PSOPT}$ is summarised in the box below and shown in Figures 53, to 58 and 59 to 61, which contain the modified equinoctial elements and the controls, respectively.

```
PSOPT results summary
=====================

Problem:  Low thrust transfer problem
CPU time (seconds): 1.142504e+01
NLP solver used:  IPOPT
PSOPT release number:  5.0.3
Date and time of this run:  Thu Mar  6 16:58:56 2025

Optimal (unscaled) cost function value:  -2.203381e-01
Phase 1 endpoint cost function value: -2.203381e-01
Phase 1 integrated part of the cost:  0.000000e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 8.683818e+04
Phase 1 maximum relative local error: 3.500641e-04
NLP solver reports:  The problem has been solved!
```

25 Manutec R3 robot

The DLR model 2 of the Manutec r3 robot, reported and validated by Otter and co-workers [15, 10], describes the motion of three links of the robot as a function of the control input signals of the robot drive:

Table 3: Mesh refinement statistics: Low thrust transfer problem

Iter	DM	M	NV	NC	OE	CE	JE	HE	RHS	$\epsilon_{\max}$	$\mathrm{CPU_a}$
1	TRP	80	803	653	364	364	173	0	57876	2.211e-03	1.983e+00
2	TRP	96	963	781	142	143	132	0	27313	2.266e-03	1.546e+00
3	H-S	106	1378	966	112	113	106	0	35708	1.179e-03	2.098e+00
4	H-S	113	1469	1029	142	143	125	0	48191	3.501e-04	2.786e+00
$\mathrm{CPU_b}$	-	-	-	-	-	-	-	-	-	-	3.011e+00
-	-	-	-	-	760	763	536	0	169088	-	1.143e+01

Key: Iter=iteration number, DM= discretization method, M=number of nodes, NV=number of variables, NC=number of constraints, OE=objective evaluations, CE = constraint evaluations, JE = Jacobian evaluations, HE = Hessian evaluations, RHS = ODE right hand side evaluations, $\epsilon_{\max}$ = maximum relative ODE error, $\mathrm{CPU_a}$ = CPU time in seconds spent by NLP algorithm, $\mathrm{CPU_b}$ = additional CPU time in seconds spent by PSOPT

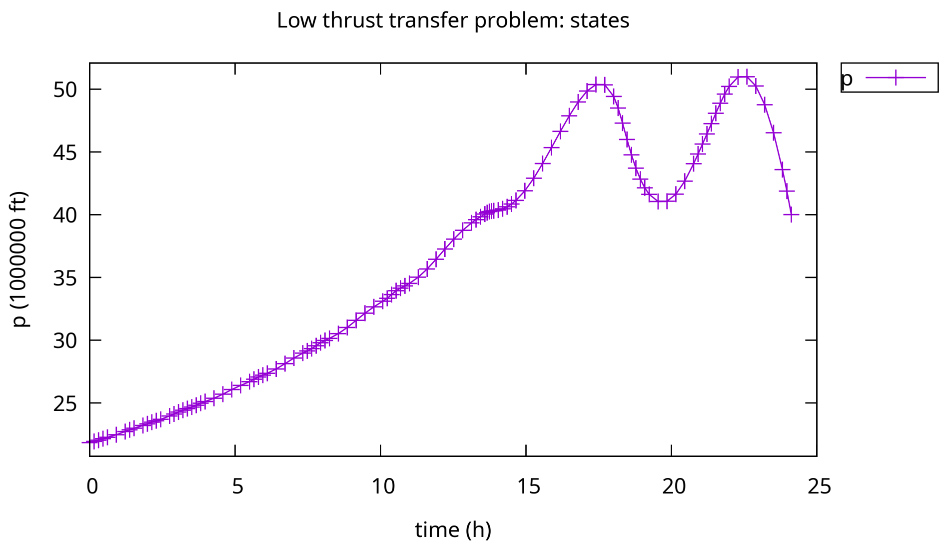

Figure 53: Modified equinoctial element p

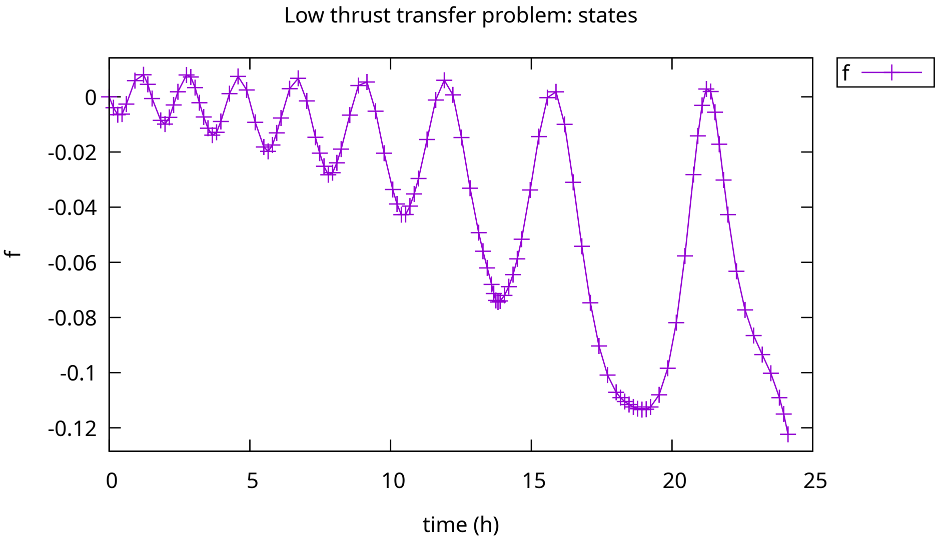

Figure 54: Modified equinoctial element f

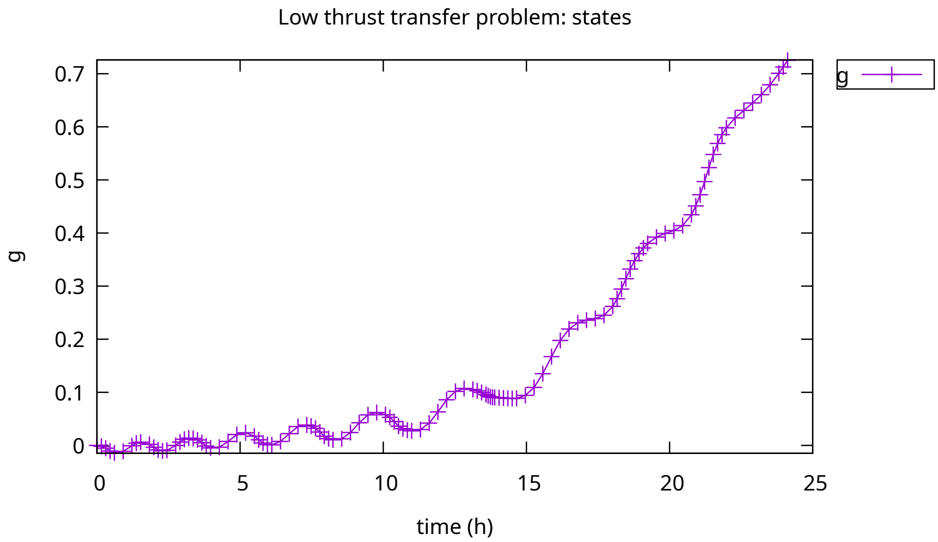

Figure 55: Modified equinoctial element g

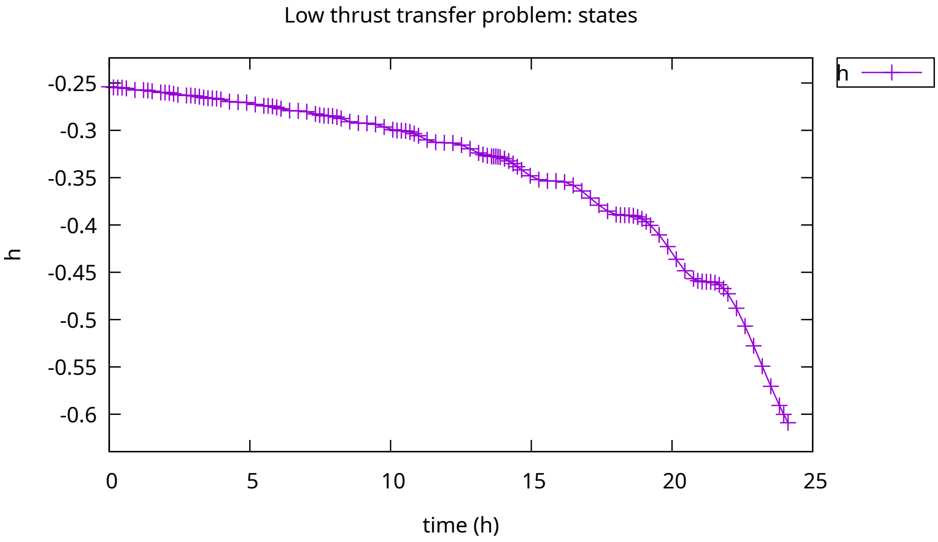

Figure 56: Modified equinoctial element h

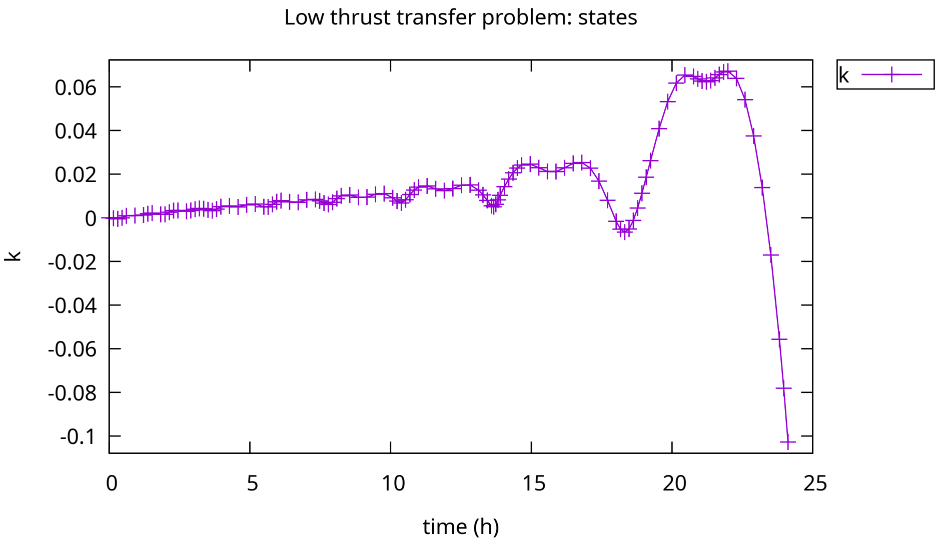

Figure 57: Modified equinoctial element k

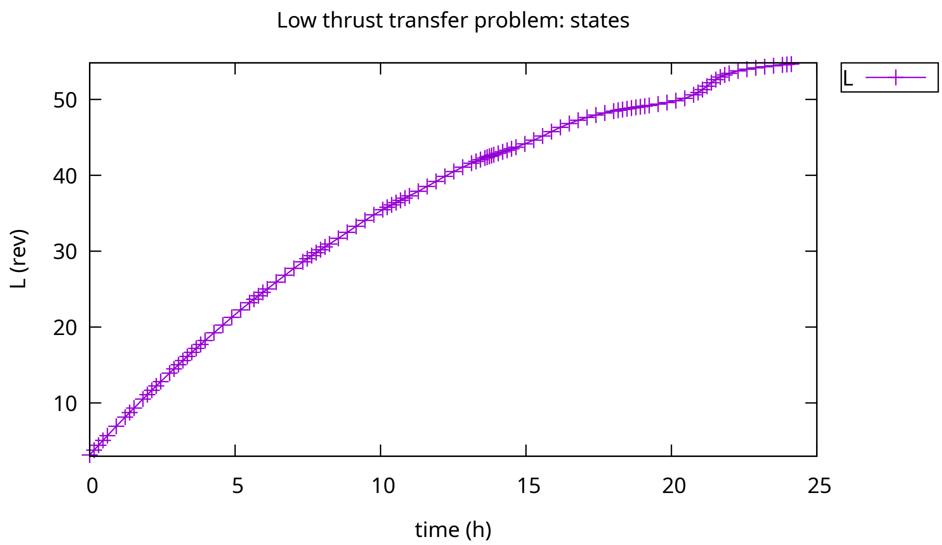

Figure 58: Modified equinoctial element L

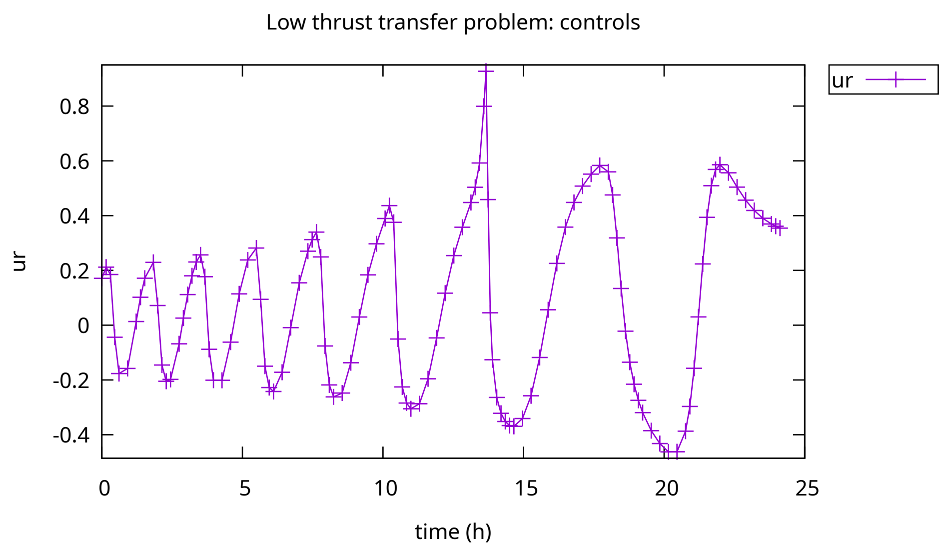

Figure 59: Radial component of the thrust direction vector, u_r

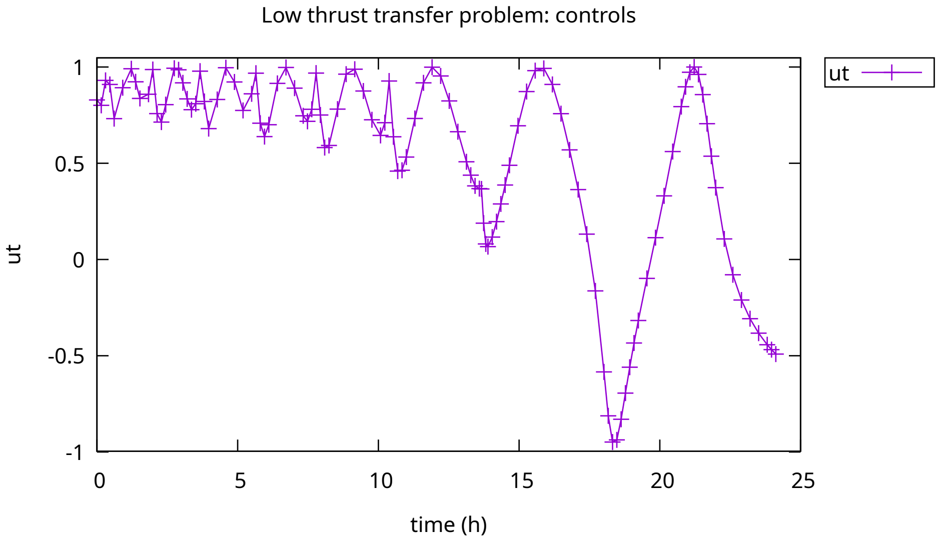

Figure 60: Tangential component of the thrust direction vector, u_t

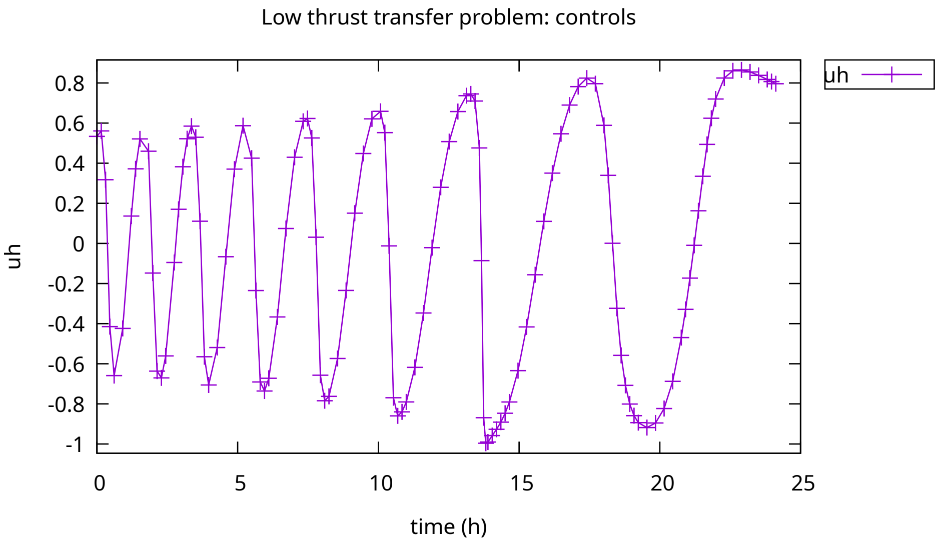

Figure 61: Normal component of the thrust direction vector, u_h

$$\mathbf{M}(\mathbf{q}(t))\ddot{\mathbf{q}}(t) = \mathbf{V}(\mathbf{q}(t), \dot{\mathbf{q}}(t)) + \mathbf{G}(\mathbf{q}(t)) + \mathbf{D}\mathbf{u}(t)$$

where $\mathbf{q} = [q_1(t), q_2(t), q_3(t)]^T$ is the vector of relative angles between the links, the normalized torque controls are $\mathbf{u}(t) = [u_1(t), u_2(t), u_3(t)]^T$, $\mathbf{D}$ is a diagonal matrix with constant values, $\mathbf{M}(\mathbf{q})$ is a symmetric inertia matrix, $\mathbf{V}(\mathbf{q}(t), \dot{\mathbf{q}}(t))$ are the torques caused by coriolis and centrifugal forces, $\mathbf{G}(\mathbf{q}(t))$ are gravitational torques. The model is described in detail in [15] and is fully included in the code for this example[3].

The example reported here consists of a minimum energy point to point trajectory, so that the objective is to find t_f and $\mathbf{u}(t) = [u_1(t), u_2(t), u_3(t)]^T$, $t \in [0, t_f]$ to minimise:

$$J = \int_0^{t_f} \mathbf{u}(t)^T \mathbf{u}(t) \mathrm{d}t \tag{107}$$

The boundary conditions associated with the problem are:

$$\begin{aligned}
\mathbf{q}(0) &= \begin{bmatrix} 0 & -1.5 & 0 \end{bmatrix}^T \\
\dot{\mathbf{q}}(0) &= \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}^T \\
\mathbf{q}(t_f) &= \begin{bmatrix} 1.0 & -1.95 & 1.0 \end{bmatrix}^T \\
\dot{\mathbf{q}}(t_f) &= \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}^T
\end{aligned} \tag{108}$$

The output from $\mathcal{PSOPT}$ is summarised in the box below and shown in Figures 62, 63 and 64, which contain the elements of the position vector $\mathbf{q}(t)$, the velocity vector $\dot{\mathbf{q}}(t)$, and the controls $\mathbf{u}(t)$, respectively. The mesh refinement process is described in Table 4.

```
PSOPT results summary
=====================

Problem:  Manutec R3 robot problem
CPU time (seconds): 1.534066e+00
NLP solver used:  IPOPT
PSOPT release number:  5.0.3
Date and time of this run:  Thu Mar  6 16:59:38 2025

Optimal (unscaled) cost function value:  2.040420e+01
Phase 1 endpoint cost function value: 0.000000e+00
Phase 1 integrated part of the cost:  2.040420e+01
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 5.300000e-01
```

[3] Dr. Martin Otter from DLR, Germany, has kindly authorised the author to publish a translated form of subroutine R3M2SI as part of the $\mathcal{PSOPT}$ distribution.

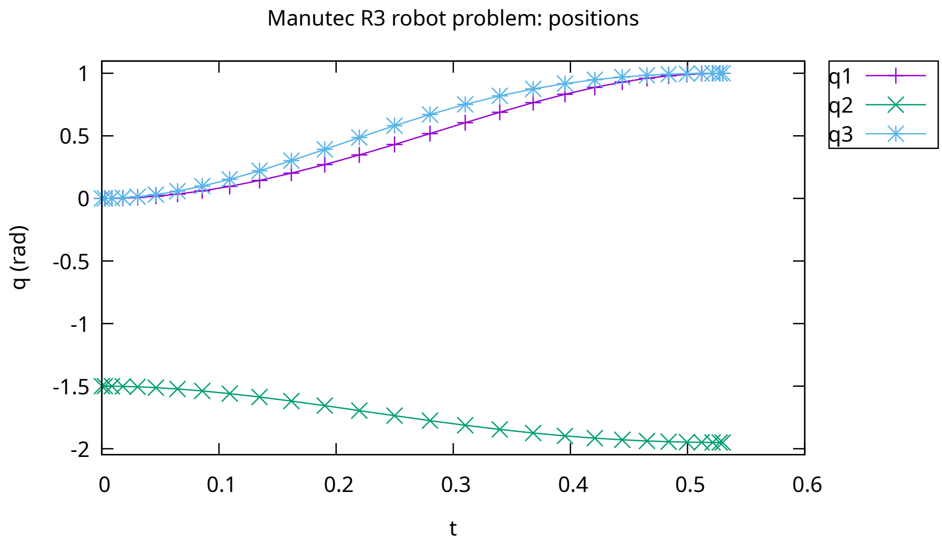

Figure 62: States q_1, q_2 and q_3 for the Manutec R3 robot minimum energy problem

```
Phase 1 maximum relative local error: 2.598526e-05
NLP solver reports:  The problem has been solved!
```

26 Minimum swing control for a container crane

Consider the following optimal control problem [22], which seeks to minimise the load swing of a container crane, while the load is transferred from one location to another. Find $u(t) \in [0, t_f]$ to minimize the cost functional

$$J = 4.5 \int_0^{t_f} \left[x_3^2(t) + x_6^2(t) \right] dt \tag{109}$$

subject to the dynamic constraints

$$\begin{aligned} \dot{x}_1 &= 9x_4 \\ \dot{x}_2 &= 9x_5 \\ \dot{x}_3 &= 9x_6 \\ \dot{x}_4 &= 9(u_1 + 17.2656x_3) \\ \dot{x}_5 &= 9u_2 \\ \dot{x}_6 &= -\frac{9}{x_2}\left[u_1 + 27.0756x_3 + 2x_5x_6\right] \end{aligned} \tag{110}$$

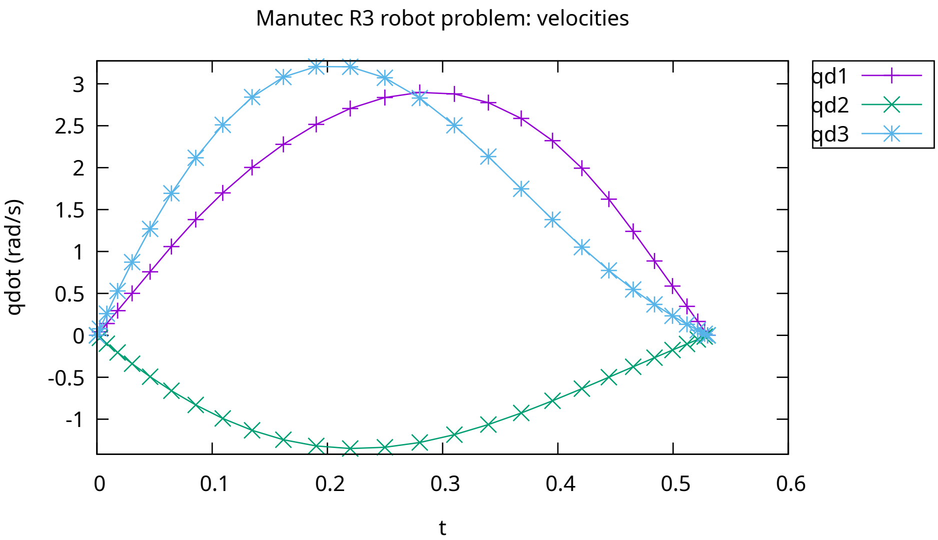

Figure 63: States $\dot{q}_1, \dot{q}_2$ and $\dot{q}_3$ for the Manutec R3 robot minimum energy problem

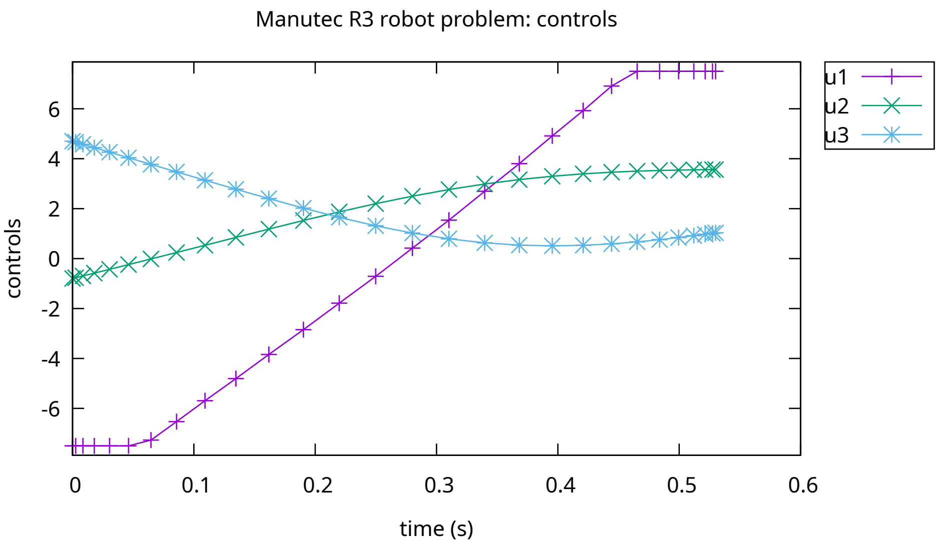

Figure 64: Controls u_1, u_2 and u_3 for the Manutec R3 robot minimum energy problem

Table 4: Mesh refinement statistics: Manutec R3 robot problem

Iter	DM	M	NV	NC	OE	CE	JE	HE	RHS	$\epsilon_{\max}$	$\mathrm{CPU_a}$
1	LGL-ST	20	182	133	43	43	35	0	860	4.676e-05	1.766e-01
2	LGL-ST	25	227	163	29	30	28	0	750	3.636e-05	1.815e-01
3	LGL-ST	26	236	169	34	35	31	0	910	2.947e-05	2.111e-01
4	LGL-ST	27	245	175	20	21	19	0	567	5.015e-05	1.461e-01
5	LGL-ST	28	254	181	19	20	18	0	560	2.599e-05	1.409e-01
$\mathrm{CPU_b}$	-	-	-	-	-	-	-	-	-	-	6.779e-01
-	-	-	-	-	145	149	131	0	3647	-	1.534e+00

Key: Iter=iteration number, DM= discretization method, M=number of nodes, NV=number of variables, NC=number of constraints, OE=objective evaluations, CE = constraint evaluations, JE = Jacobian evaluations, HE = Hessian evaluations, RHS = ODE right hand side evaluations, $\epsilon_{\max}$ = maximum relative ODE error, $\mathrm{CPU_a}$ = CPU time in seconds spent by NLP algorithm, $\mathrm{CPU_b}$ = additional CPU time in seconds spent by PSOPT

the boundary conditions

$$\begin{array}{rclcrcl} x_1(0) &=& 0 & & x_1(t_f) &=& 10 \\ x_2(0) &=& 22 & & x_2(t_f) &=& 14 \\ x_3(0) &=& 0 & & x_3(t_f) &=& 0 \\ x_4(0) &=& 0 & & x_4(t_f) &=& 2.5 \\ x_5(0) &=& -1 & & x_5(t_f) &=& 0 \\ x_6(0) &=& 0 & & x_6(t_f) &=& 0 \end{array}, \tag{111}$$

and the bounds

$$\begin{aligned} -2.83374 \leq & u_1(t) \leq 2.83374, \\ -0.80865 \leq & u_2(t) \leq 0.71265, \\ -2.5 \leq & x_4(t) \leq 2.5, \\ -1 \leq & x_5(t) \leq 1. \end{aligned} \tag{112}$$

The output from $\mathcal{PSOPT}$ is summarised in the box below and shown in Figures 65, 66 and 67, which contain the elements of the state x_1 to x_3, x_4 to x_6, and the controls, respectively.

```
PSOPT results summary
=====================

Problem:  Minimum swing control for a container crane
CPU time (seconds): 9.554088e+00
NLP solver used:  IPOPT
PSOPT release number:  5.0.3
Date and time of this run:  Thu Mar  6 16:55:16 2025
```

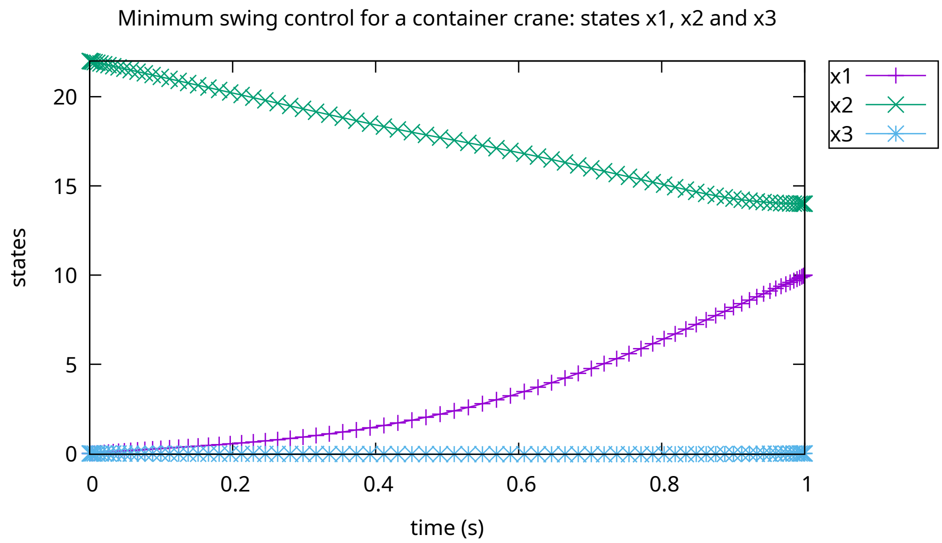

Figure 65: States x_1, x_2 and x_3 for minimum swing crane control problem

```
Optimal (unscaled) cost function value:  5.151388e-03
Phase 1 endpoint cost function value: 0.000000e+00
Phase 1 integrated part of the cost:  5.151388e-03
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 1.000000e+00
Phase 1 maximum relative local error: 1.010589e-04
NLP solver reports:  The problem has been solved!
```

27 Minimum time to climb for a supersonic aircraft

Consider the following optimal control problem, which finds the minimum time to climb to a given altitude for a supersonic aircraft [4]. Minimize the cost functional

$$J = t_f \tag{113}$$

subject to the dynamic constraints

$$\begin{aligned}
\dot{h} &= v \sin\gamma \\
\dot{v} &= \tfrac{1}{m}\left[T(M,h)\cos\alpha - D\right] - \tfrac{\mu}{(R_e+h)^2}\sin\gamma \\
\dot{\gamma} &= \tfrac{1}{mv}\left[T(M,h)\sin\alpha + L\right] + \cos\gamma\left[\tfrac{v}{(R_e+h)} - \tfrac{\mu}{v(R_e+h)^2}\right] \\
\dot{w} &= \tfrac{-T(M,h)}{I_{sp}}
\end{aligned} \tag{114}$$

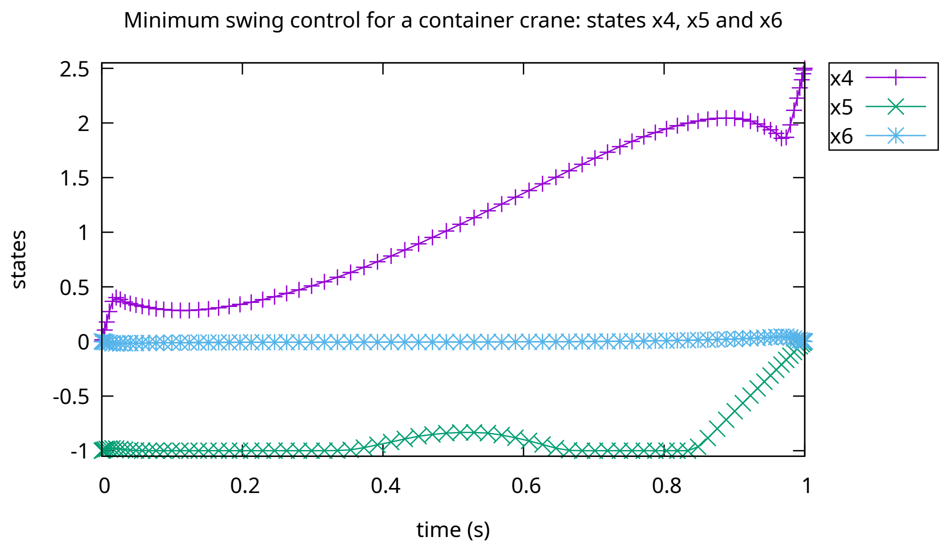

Figure 66: States x_4, x_5 and x_6 for minimum swing crane control problem

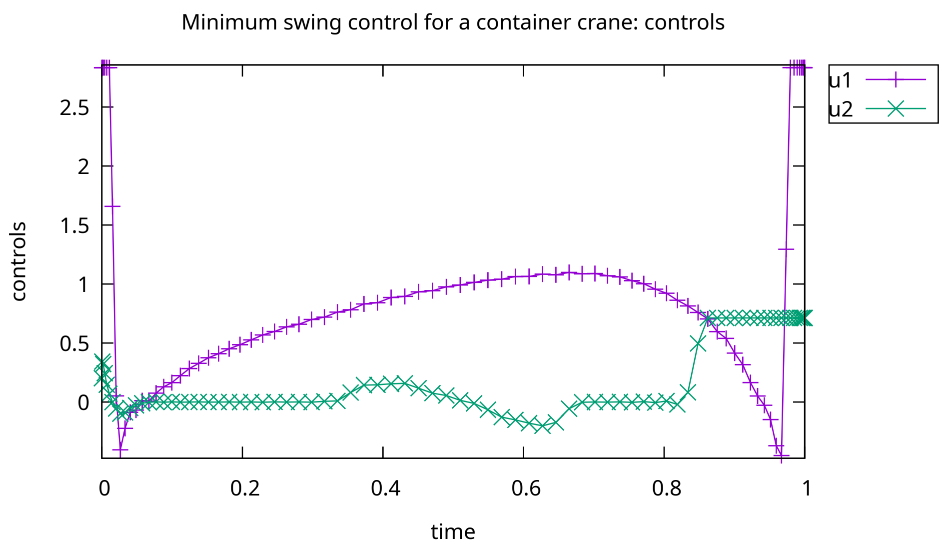

Figure 67: Controls for minimum swing crane control problem

where h is the altitude (ft), v is the velocity (ft/s), γ is the flight path angle (rad), w is the weight (lb), L is the lift force, D is the drag force (lb), T is the thrust (lb), $M = v/c$ is the mach number, $m = w/g_0$ (slug) is the mass, $c(h)$ is the speed of sound (ft/s), R_e is the radious of Earth, and μ is the gravitational constant. The control input α is the angle of attack (rad).

The speed of sound is given by:

$$c = 20.0468\sqrt{\theta} \tag{115}$$

where $\theta = \theta(h)$ is the atmospheric temperature (K).

The aerodynamic forces are given by:

$$\begin{aligned} D &= \frac{1}{2} C_D S \rho v^2 \\ L &= \frac{1}{2} C_L S \rho v^2 \end{aligned} \tag{116}$$

where

$$\begin{aligned} C_L &= c_{L\alpha}(M)\alpha \\ C_D &= c_{D0}(M) + \eta(M) c_{L\alpha}(M)\alpha^2 \end{aligned} \tag{117}$$

where C_L and C_D are aerodynamic lift and drag coefficients, S is the aerodynamic reference area of the aircraft, and $\rho = \rho(h)$ is the air density.

The boundary conditions are given by:

$$\begin{aligned} h(0) &= 0 \text{ (ft)}, \\ h(t_f) &= 65600.0 \text{ (ft)} \\ v(0) &= 424.260 \text{ (ft/s)}, \\ v(t_f) &= 968.148 \text{ (ft/s)} \\ \gamma(0) &= \gamma(t_f) = 0 \text{ (rad)} \\ w(0) &= 42000.0 \text{ lb} \end{aligned} \tag{118}$$

The parameter values are given by:

$$\begin{aligned} S &= 530 \text{ (ft}^2\text{)}, \\ I_{sp} &= 1600.0 \text{ (sec)} \\ \mu &= 0.14046539 \times 10^{17} \text{ (ft}^3/\text{s}^2\text{)}, \\ g_0 &= 32.174 \text{ (ft/s}^2\text{)} \\ R_e &= 20902900 \text{ (ft)} \end{aligned} \tag{119}$$

The variables $c_{L\alpha}(M)$, $c_{D0}(M)$, $\eta(M)$ are interpolated from 1-D tabular data which is given in the code and also in [4], using spline interpolation, while the thrust $T(M, h)$ is interpolated from 2-D tabular data given in the code and in [4], using 2D spline interpolation.

The air density ρ and the atmospheric temperature θ were calculated using the US Standard Atmosphere Model 1976[4], based on the standard temperature of 15 (deg C) at zero altitude and the standard air density of 1.22521 (slug/ft^3) at zero altitude.

The C++ code that solves this problem is shown below.

```
//////////////////////////////////////////////////////////////////////////
//////////////////            climb.cxx             /////////////////////
//////////////////////////////////////////////////////////////////////////
////////////////           PSOPT  Example            ////////////////////
//////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////
//////// Title: Minimum time to climb of a supersonic aircraft ///////////
//////// Last modified: 12 January 2009                  ////////////////
//////// Reference:     GPOPS Manual   ////////////////
//////// (See PSOPT handbook for full reference)           ///////////////
//////////////////////////////////////////////////////////////////////////
////////     Copyright (c) Victor M. Becerra, 2009        ////////////////
//////////////////////////////////////////////////////////////////////////
//////// This is part of the PSOPT software library, which ///////////////
//////// is distributed under the terms of the GNU Lesser ////////////////
//////// General Public License (LGPL)                    ////////////////
//////////////////////////////////////////////////////////////////////////

#include "psopt.h"

using namespace PSOPT;



//////////////////////////////////////////////////////////////////////////
/////////  Declare an auxiliary structure to hold local constants ////////
//////////////////////////////////////////////////////////////////////////

struct Constants {
  double g0;
  double S;
  double Re;
  double Isp;
  double mu;
  MatrixXd* CLa_table;
  MatrixXd* CD0_table;
  MatrixXd* eta_table;
  MatrixXd* T_table;
  MatrixXd* M1;
  MatrixXd* M2;
  MatrixXd* h1;
  MatrixXd* htab;
  MatrixXd* ttab;
  MatrixXd* ptab;
  MatrixXd* gtab;
};

typedef struct Constants Constants_;

void atmosphere(adouble* alt,adouble* sigma,adouble* delta,adouble* theta, Constants_& CONSTANTS);

void atmosphere_model(adouble* rho, adouble* M, adouble v, adouble h, Constants_& CONSTANTS);


//////////////////////////////////////////////////////////////////////////
//////////////////  Define the end point (Mayer) cost function //////////
//////////////////////////////////////////////////////////////////////////

adouble endpoint_cost(adouble* initial_states, adouble* final_states,
                      adouble* parameters,adouble& t0, adouble& tf,
                      adouble* xad, int iphase, Workspace* workspace)
{
return tf;
}

//////////////////////////////////////////////////////////////////////////
//////////////////  Define the integrand (Lagrange) cost function  //////
//////////////////////////////////////////////////////////////////////////
```

[4] see http://www.pdas.com/programs/atmos.f90

```
adouble integrand_cost(adouble* states, adouble* controls, adouble* parameters,
                     adouble& time, adouble* xad, int iphase, Workspace* workspace)

{
    return  0.0;
}



//////////////////////////////////////////////////////////////////////////
///////////////////  Define the DAE's ////////////////////////////////////
//////////////////////////////////////////////////////////////////////////

void dae(adouble* derivatives, adouble* path, adouble* states,
         adouble* controls, adouble* parameters, adouble& time,
         adouble* xad, int iphase, Workspace* workspace)
{

  Constants_& CONSTANTS = *( (Constants_ *) workspace->problem->user_data );

  adouble alpha = controls[ 0]; // Angle of attack (rad)


  adouble h     =   states[ 0 ]; // Altitude (ft)
  adouble v     =   states[ 1 ]; // Velocity (ft/s)
  adouble gamma =   states[ 2 ]; // Flight path angle (rad)
  adouble w     =   states[ 3 ]; // weight (lb)


  double g0   = CONSTANTS.g0;
  double S    = CONSTANTS.S;
  double Re   = CONSTANTS.Re;
  double Isp  = CONSTANTS.Isp;
  double mu   = CONSTANTS.mu;

  MatrixXd& M1         = *CONSTANTS.M1;
  MatrixXd& M2         = *CONSTANTS.M2;
  MatrixXd& h1         = *CONSTANTS.h1;
  MatrixXd& CLa_table  = *CONSTANTS.CLa_table;
  MatrixXd& CD0_table  = *CONSTANTS.CD0_table;
  MatrixXd& eta_table  = *CONSTANTS.eta_table;
  MatrixXd& T_table    = *CONSTANTS.T_table;

  int lM1 = length(M1);

  adouble rho;
  adouble   m = w/g0;
  adouble   M;

  atmosphere_model( &rho, &M, v, h, CONSTANTS);

  adouble CL_a, CD0, eta, T;


  spline_interpolation( &CL_a, M, M1, CLa_table,  lM1);
  spline_interpolation( &CD0,  M, M1, CD0_table, lM1);
  spline_interpolation( &eta,  M, M1, eta_table, lM1);
  spline_2d_interpolation(&T, M, h, M2, h1, T_table, workspace);

//  smooth_linear_interpolation( &CL_a, M, M1, CLa_table,  lM1);
//  smooth_linear_interpolation( &CD0,  M, M1, CD0_table, lM1);
//  smooth_linear_interpolation( &eta,  M, M1, eta_table, lM1);
//  smooth_bilinear_interpolation(&T, M, h, M2, h1, T_table);


//  linear_interpolation( &CL_a, M, M1, CLa_table,  lM1);
//  linear_interpolation( &CD0,  M, M1, CD0_table, lM1);
//  linear_interpolation( &eta,  M, M1, eta_table, lM1);
//  bilinear_interpolation(&T, M, h, M2, h1, T_table);


   adouble CL = CL_a*alpha;
   adouble CD = CD0 + eta*CL_a*alpha*alpha;

   adouble D = 0.5*CD*S*rho*v*v;
   adouble L = 0.5*CL*S*rho*v*v;


   adouble hdot     = v*sin(gamma);
   adouble vdot     = 1.0/m*(T*cos(alpha)-D) - mu/pow(Re+h,2.0)*sin(gamma);
   adouble gammadot = (1.0/(m*v))*(T*sin(alpha)+L) + cos(gamma)*(v/(Re+h)-mu/(v*pow(Re+h,2.0)));
   adouble wdot     = -T/Isp;
```

```
   derivatives[ 0 ] = hdot;
   derivatives[ 1 ] = vdot;
   derivatives[ 2 ] = gammadot;
   derivatives[ 3 ] = wdot;


}

////////////////////////////////////////////////////////////////////////////
//////////////////  Define the events function ////////////////////////////
////////////////////////////////////////////////////////////////////////////

void events(adouble* e, adouble* initial_states, adouble* final_states,
            adouble* parameters,adouble& t0, adouble& tf, adouble* xad,
            int iphase, Workspace* workspace)
{



      adouble h0     = initial_states[0];
      adouble v0     = initial_states[1];
      adouble gamma0 = initial_states[2];
      adouble w0     = initial_states[3];

      adouble hf     = final_states[0];
      adouble vf     = final_states[1];
      adouble gammaf = final_states[2];

      e[ 0 ] = h0;
      e[ 1 ] = v0;
      e[ 2 ] = gamma0;
      e[ 3 ] = w0;
      e[ 4 ] = hf;
      e[ 5 ] = vf;
      e[ 6 ] = gammaf;

}


///////////////////////////////////////////////////////////////////////////
//////////////////  Define the phase linkages function ///////////////////
///////////////////////////////////////////////////////////////////////////

void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{

  // Single phase problem

}



////////////////////////////////////////////////////////////////////////////
//////////////////  Define the main routine ///////////////////////////////
////////////////////////////////////////////////////////////////////////////

int main(void)
{
////////////////////////////////////////////////////////////////////////////
//////////////////  Declare key structures ////////////////////////////////
////////////////////////////////////////////////////////////////////////////

    Alg  algorithm;
    Sol  solution;
    Prob problem;

////////////////////////////////////////////////////////////////////////////
//////////////////  Register problem name  ////////////////////////////////
////////////////////////////////////////////////////////////////////////////

    problem.name = "Minimum time to climb for a supersonic aircraft";
    problem.outfilename = "climb.txt";

////////////////////////////////////////////////////////////////////////////
///////////  Define problem level constants & do level 1 setup ////////////
////////////////////////////////////////////////////////////////////////////

    problem.nphases    = 1;
    problem.nlinkages                  = 0;
```

```
    psopt_level1_setup(problem);

////////////////////////////////////////////////////////////////////////////
/////////   Define phase related information & do level 2 setup  ////////////
////////////////////////////////////////////////////////////////////////////

    problem.phases(1).nstates   = 4;
    problem.phases(1).ncontrols = 1;
    problem.phases(1).nevents   = 7;
    problem.phases(1).npath     = 0;

    problem.phases(1).nodes     = (RowVectorXi(1) << 30).finished();


    psopt_level2_setup(problem, algorithm);

////////////////////////////////////////////////////////////////////////////
///////////////////  Declare an instance of Constants structure /////////////
////////////////////////////////////////////////////////////////////////////


    Constants_ CONSTANTS;

    problem.user_data = (void*) &CONSTANTS;

////////////////////////////////////////////////////////////////////////////
///////////////////  Declare MatrixXd objects to store results //////////////
////////////////////////////////////////////////////////////////////////////

    MatrixXd x, u, t, H;



////////////////////////////////////////////////////////////////////////////
///////////////////  Initialize CONSTANTS and //////////////////////////////
///////////////////  declare local variables  //////////////////////////////
////////////////////////////////////////////////////////////////////////////

   CONSTANTS.g0   = 32.174; // ft/s^2
   CONSTANTS.S    = 530.0; // ft^2
   CONSTANTS.Re   = 20902900.0; // ft
   CONSTANTS.Isp  = 1600.00; //s
   CONSTANTS.mu   = 0.14076539E17; // ft^3/s^2


   MatrixXd M1(1,9);
   M1 << 0.E0, .4E0, .8E0, .9E0, 1.E0, 1.2E0, 1.4E0, 1.6E0, 1.8E0;

   MatrixXd M2(1,10);
   M2 << 0.E0, .2E0, .4E0, .6E0, .8E0, 1.E0, 1.2E0, 1.4E0, 1.6E0, 1.8E0;

   MatrixXd h1(1,10);
   h1 << 0.E0, 5E3, 10.E3, 15.E3, 20.E3, 25.E3, 30.E3, 40.E3, 50.E3, 70.E3;

   MatrixXd CLa_table(1,9);
   CLa_table << 3.44E0, 3.44E0, 3.44E0, 3.58E0, 4.44E0, 3.44E0, 3.01E0, 2.86E0, 2.44E0;

   MatrixXd CD0_table(1,9);
   CD0_table << .013E0, .013E0, .013E0, .014E0, .031E0, 0.041E0, .039E0, .036E0, .035E0;

   MatrixXd eta_table(1,9);
   eta_table << .54E0, .54E0, .54E0, .75E0, .79E0, .78E0, .89E0, .93E0, .93E0;

   MatrixXd T_table(10,10);
   T_table << 24200., 24000.,  20300.,  17300.,14500.,12200.,10200.,5700.,3400.,100.,
   28000., 24600.,  21100.,  18100.,15200.,12800.,10700.,6500.,3900.,200.,
   28300., 25200.,  21900.,  18700.,15900.,13400.,11200.,7300.,4400.,400.,
   30800., 27200.,  23800.,  20500.,17300.,14700.,12300.,8100.,4900.,800.,
   34500., 30300.,  26600.,  23200.,19800.,16800.,14100.,9400.,5600.,1100.,
   37900., 34300.,  30400.,  26800.,23300.,19800.,16800.,11200.,6800.,1400.,
   36100., 38000.,  34900.,  31300.,27300.,23600.,20100.,13400.,8300.,1700.,
   34300., 36600.,  38500.,  36100.,31600.,28100.,24200.,16200.,10000.,2200.,
   32500., 35200.,  42100.,  38700.,35700.,32000.,28100.,19300.,11900.,2900.,
   30700., 33800.,  45700.,  41300.,39800.,34600.,31100.,21700.,13300.,3100. ;

   MatrixXd htab(1,8);
   htab <<      0.0, 11.0, 20.0, 32.0, 47.0, 51.0, 71.0, 84.852;
   MatrixXd ttab(1,8);
   ttab <<      288.15, 216.65, 216.65, 228.65, 270.65, 270.65, 214.65, 186.946;
   MatrixXd ptab(1,8);
   ptab <<      1.0, 2.233611E-1, 5.403295E-2, 8.5666784E-3, 1.0945601E-3,
                               6.6063531E-4, 3.9046834E-5, 3.68501E-6;
```

```
   MatrixXd gtab(1,8);
   gtab << -6.5, 0.0, 1.0, 2.8, 0.0, -2.8, -2.0, 0.0;

//   M1.Print("M1");

//   M2.Print("M2");

//   h1.Print("h1");

//   CLa_table.Print("CLa_table");

//   CD0_table.Print("CD0_table");

//   eta_table.Print("eta_table");

//   T_table.Print("T_table");


   CONSTANTS.M1         = &M1;
   CONSTANTS.M2         = &M2;
   CONSTANTS.h1         = &h1;
   CONSTANTS.CLa_table  = &CLa_table;
   CONSTANTS.CD0_table  = &CD0_table;
   CONSTANTS.eta_table  = &eta_table;
   CONSTANTS.T_table    = &T_table;
   CONSTANTS.htab       = &htab;
   CONSTANTS.ttab       = &ttab;
   CONSTANTS.ptab       = &ptab;
   CONSTANTS.gtab       = &gtab;

   double h0     = 0.0;
   double hf     = 65600.0;
   double v0     = 424.26;
   double vf     = 968.148;
   double gamma0 = 0.0;
   double gammaf = 0.0;
   double w0     = 42000.0;

   double hmin =  0;
   double hmax =  69000.0;
   double vmin =  1.0;
   double vmax =  2000.0;
   double gammamin = -89.0*pi/180.0; // -89.0*pi/180.0;
   double gammamax =  89.0*pi/180.0; //  89.0*pi/180.0;
   double wmin     = 0.0;
   double wmax     = 45000.0;
   double alphamin = -20.0*pi/180.0;
   double alphamax =  20.0*pi/180.0;

   double t0min = 0.0;
   double t0max = 0.0;
   double tfmin = 200.0;
   double tfmax = 500.0;


////////////////////////////////////////////////////////////////////////////
///////////////////  Enter problem bounds information //////////////////////
////////////////////////////////////////////////////////////////////////////

    int iphase =  1;

    problem.phases(iphase).bounds.lower.StartTime    = t0min;
    problem.phases(iphase).bounds.upper.StartTime    = t0max;

    problem.phases(iphase).bounds.lower.EndTime      = tfmin;
    problem.phases(iphase).bounds.upper.EndTime      = tfmax;


    problem.phases(iphase).bounds.lower.states(0) = hmin;
    problem.phases(iphase).bounds.upper.states(0) = hmax;
    problem.phases(iphase).bounds.lower.states(1) = vmin;
    problem.phases(iphase).bounds.upper.states(1) = vmax;
    problem.phases(iphase).bounds.lower.states(2) = gammamin;
    problem.phases(iphase).bounds.upper.states(2) = gammamax;
    problem.phases(iphase).bounds.lower.states(3) = wmin;
    problem.phases(iphase).bounds.upper.states(3) = wmax;


    problem.phases(iphase).bounds.lower.controls(0) = alphamin;
    problem.phases(iphase).bounds.upper.controls(0) = alphamax;

    // The following bounds fix the initial and final state conditions
```

```
    problem.phases(iphase).bounds.lower.events(0) = h0;
    problem.phases(iphase).bounds.upper.events(0) = h0;
    problem.phases(iphase).bounds.lower.events(1) = v0;
    problem.phases(iphase).bounds.upper.events(1) = v0;
    problem.phases(iphase).bounds.lower.events(2) = gamma0;
    problem.phases(iphase).bounds.upper.events(2) = gamma0;
    problem.phases(iphase).bounds.lower.events(3) = w0;
    problem.phases(iphase).bounds.upper.events(3) = w0;
    problem.phases(iphase).bounds.lower.events(4) = hf;
    problem.phases(iphase).bounds.upper.events(4) = hf;
    problem.phases(iphase).bounds.lower.events(5) = vf;
    problem.phases(iphase).bounds.upper.events(5) = vf;
    problem.phases(iphase).bounds.lower.events(6) = gammaf;
    problem.phases(iphase).bounds.upper.events(6) = gammaf;


////////////////////////////////////////////////////////////////////////////
///////////////////  Define & register initial guess ///////////////////////
////////////////////////////////////////////////////////////////////////////

    int nnodes = problem.phases(iphase).nodes(0);


    MatrixXd stateGuess(4,nnodes);

    stateGuess.row(0) = linspace(h0,hf,nnodes);
    stateGuess.row(1) = linspace(v0,vf,nnodes);
    stateGuess.row(2) = linspace(gamma0,gammaf,nnodes);
    stateGuess.row(3) = linspace(w0,0.8*w0,nnodes);



    problem.phases(1).guess.controls       = zeros(1,nnodes);
    problem.phases(1).guess.states         = stateGuess;
    problem.phases(1).guess.time           = linspace(t0min, tfmax, nnodes);


////////////////////////////////////////////////////////////////////////////
///////////////////  Register problem functions  ///////////////////////////
////////////////////////////////////////////////////////////////////////////


    problem.integrand_cost  = &integrand_cost;
    problem.endpoint_cost  = &endpoint_cost;
    problem.dae  = &dae;
    problem.events  = &events;
    problem.linkages = &linkages;

////////////////////////////////////////////////////////////////////////////
///////////////////  Enter algorithm options  //////////////////////////////
////////////////////////////////////////////////////////////////////////////


    algorithm.nlp_method                   = "IPOPT";
    algorithm.scaling                      = "automatic";
    algorithm.derivatives                  = "numerical";
    algorithm.collocation_method            = "trapezoidal";
    algorithm.nlp_iter_max                 = 1000;
    algorithm.nlp_tolerance                = 1.e-6;
    algorithm.mesh_refinement               = "automatic";
    algorithm.mr_max_iterations             = 4;
    algorithm.defect_scaling                = "jacobian-based";




////////////////////////////////////////////////////////////////////////////
///////////////////  Now call PSOPT to solve the problem   /////////////////
////////////////////////////////////////////////////////////////////////////


    psopt(solution, problem, algorithm);


////////////////////////////////////////////////////////////////////////////
///////////  Extract relevant variables from solution structure   //////////
////////////////////////////////////////////////////////////////////////////

    x  = solution.get_states_in_phase(1);
    u  = solution.get_controls_in_phase(1);
    t  = solution.get_time_in_phase(1);
    H       = solution.get_dual_hamiltonian_in_phase(1);
```

```
    MatrixXd h     = x.row(0);
    MatrixXd v     = x.row(1);
    MatrixXd gamma = x.row(2);
    MatrixXd w     = x.row(3);

////////////////////////////////////////////////////////////////////////////
///////////  Save solution data to files if desired ////////////////////////
////////////////////////////////////////////////////////////////////////////

    Save(x,"x.dat");
    Save(u,"u.dat");
    Save(t,"t.dat");


////////////////////////////////////////////////////////////////////////////
///////////  Plot some results if desired (requires gnuplot) ///////////////
////////////////////////////////////////////////////////////////////////////


    plot(t,h/1000.0,problem.name + ": altitude", "time (s)", "altitude (x1,000 ft)", "h");
    plot(t,v/100.0,problem.name + ": velocity", "time (s)", "velocity (x100 ft/s)", "v");
    plot(t,gamma*180/pi,problem.name + ": flight path angle", "time (s)", "gamma (deg)", "gamma");
    plot(t,w/10000.0,problem.name + ": weight", "time (s)", "w (x10,000 lb)", "w");
    plot(t,u*180/pi,problem.name + ": angle of attack", "time (s)", "alpha (deg)", "alpha");

    plot(t,h/1000.0,problem.name + ": altitude", "time (s)", "altitude (x1,000 ft", "h",
                                   "pdf","climb_altitude.pdf");
    plot(t,v/100.0,problem.name + ": velocity", "time (s)", "velocity (x100 ft/s)", "v",
                             "pdf","climb_velocity.pdf");
    plot(t,gamma*180/pi,problem.name + ": flight path angle", "time (s)", "gamma (deg)", "gamma",
                                     "pdf","climb_fpa.pdf");
    plot(t,w/10000.0,problem.name + ": weight", "time (s)", "w (x10,000 lb)", "w", "pdf",
                                      "weight.pdf");
    plot(t,u*180/pi,problem.name + ": angle of attack", "time (s)", "alpha (deg)", "alpha",
                                      "pdf", "alpha.pdf");


}


void atmosphere(adouble* alt,adouble* sigma,adouble* delta,adouble* theta, Constants_& CONSTANTS)
// US Standard Atmosphere Model 1976
// Adopted from original Fortran 90 code by Ralph Carmichael
// Fortran code located at: http://www.pdas.com/programs/atmos.f90
{
/*!   ------------------------------------------------------------------------
! PURPOSE - Compute the properties of the 1976 standard atmosphere to 86 km.
! AUTHOR - Ralph Carmichael, Public Domain Aeronautical Software
! NOTE - If alt > 86, the values returned will not be correct, but they will
!   not be too far removed from the correct values for density.
!   The reference document does not use the terms pressure and temperature
!   above 86 km.
  IMPLICIT NONE
!============================================================================
!     A R G U M E N T S                                                      |
!============================================================================
  alt        ! geometric altitude, km.
  sigma      ! density/sea-level standard density
  delta      ! pressure/sea-level standard pressure
  theta      ! temperature/sea-level standard temperature
*/

/*!===========================================================================
!     L O C A L   C O N S T A N T S                                         |
!============================================================================
*/
  double REARTH = 6369.0;                 // radius of the Earth (km)
  double GMR = 34.163195;                 // hydrostatic constant
  int NTAB=8;       // number of entries in the defining tables
/*!===========================================================================
!     L O C A L   V A R I A B L E S                                         |
!============================================================================
*/
  int i,j,k;                                                  // counters
  adouble h;                                         // geopotential altitude (km)
  adouble tgrad, tbase;      // temperature gradient and base temp of this layer
  adouble tlocal;                                           // local temperature
  adouble deltah;                                   // height above base of this layer
/*!===========================================================================
!     L O C A L   A R R A Y S   ( 1 9 7 6   S T D.  A T M O S P H E R E )   |
!============================================================================
*/
```

```
  MatrixXd& htab = *CONSTANTS.htab;
  MatrixXd& ttab = *CONSTANTS.ttab;
  MatrixXd& ptab = *CONSTANTS.ptab;
  MatrixXd& gtab = *CONSTANTS.gtab;


//!----------------------------------------------------------------------------
  h=(*alt)*REARTH/((*alt)+REARTH);      //convert geometric to geopotential altitude

  i=1;
  j=NTAB;                                       // setting up for binary search
  while (j<=i+1) {
    k=(i+j)/2;                                              // integer division
    if (h < htab(k-1)) {
      j=k;
    } else {
       i=k;
    }
  }

  tgrad=gtab(i-1);                                     // i will be in 1...NTAB-1
  tbase=ttab(i-1);
  deltah=h-htab(i-1);
  tlocal=tbase+tgrad*deltah;
  *theta=tlocal/ttab(0);                                    // temperature ratio

  if (tgrad == 0.0) {                                  //  pressure ratio
    *delta=ptab(i-1)*exp(-GMR*deltah/tbase);
  } else {
    *delta=ptab(i-1)*pow(tbase/tlocal, GMR/tgrad);
  }

  *sigma=(*delta)/(*theta);                                          // density ratio
  return;

}


void atmosphere_model(adouble* rho, adouble* M, adouble v, adouble h, Constants_& CONSTANTS)
{
   double feet2meter = 0.3048;
   double kgperm3_to_slug_per_feet3 = 0.062427960841/32.174049;
   adouble alt, sigma, delta, theta;
   alt = h.value()*feet2meter/1000.0;

   // Call the standard atmosphere model 1976

   atmosphere(&alt, &sigma, &delta, &theta, CONSTANTS);

   adouble rho1 = 1.22521 * sigma; // Multiply by standard density at zero altitude and 15 deg C.

   rho1 = rho1*kgperm3_to_slug_per_feet3;

   *rho = rho1;

   adouble T;
   adouble mach;

   double TempStandardSeaLevel = 288.15; // in K, or 15 deg C.

   T = theta*TempStandardSeaLevel;

   adouble a = 20.0468 * sqrt(T); // Speed of sound in m/s.

   a = a/feet2meter;  // Speed of sound in ft/s

   mach = v/a;

   *M = mach;


  return;
}

////////////////////////////////////////////////////////////////////////////
///////////////////////      END OF FILE     ///////////////////////////////
////////////////////////////////////////////////////////////////////////////
```

The output from $\mathcal{PSOPT}$ is summarized in the box below and Figures 68, to 72. The

Table 5: Mesh refinement statistics: Minimum time to climb for a supersonic aircraft

Iter	DM	M	NV	NC	OE	CE	JE	HE	RHS	ϵ_{max}	CPU_a
1	TRP	30	152	128	14008	2506	46	0	147854	3.968e-02	4.433e+00
2	TRP	42	212	176	25812	3406	61	0	282698	2.070e-02	8.506e+00
3	H-S	58	349	240	40904	4684	59	0	805648	1.139e-02	2.425e+01
4	H-S	77	463	316	63461	5992	69	0	1372168	1.899e-03	4.197e+01
CPU_b	-	-	-	-	-	-	-	-	-	-	1.095e+01
-	-	-	-	-	144185	16588	235	0	2608368	-	9.012e+01

Key: Iter=iteration number, DM= discretization method, M=number of nodes, NV=number of variables, NC=number of constraints, OE=objective evaluations, CE = constraint evaluations, JE = Jacobian evaluations, HE = Hessian evaluations, RHS = ODE right hand side evaluations, ϵ_{max} = maximum relative ODE error, CPU_a = CPU time in seconds spent by NLP algorithm, CPU_b = additional CPU time in seconds spent by PSOPT

results can be compared with those presented in [4]. Table 1 shows the mesh refinement history for this problem.

```
PSOPT results summary
=====================

Problem:  Minimum time to climb for a supersonic aircraft
CPU time (seconds): 9.011813e+01
NLP solver used:  IPOPT
PSOPT release number:  5.0.3
Date and time of this run:  Thu Mar  6 16:54:06 2025

Optimal (unscaled) cost function value:  3.188146e+02
Phase 1 endpoint cost function value: 3.188146e+02
Phase 1 integrated part of the cost:  0.000000e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 3.188146e+02
Phase 1 maximum relative local error: 1.898667e-03
NLP solver reports:  The problem has been solved!
```

28 Missile terminal burn maneouvre

This example illustrates the design of a missile trajectory to strike a specified target from given initial conditions in minimum time [21]. Figure 28 shows the variables associated with the dynamic model of the missile employed in this example, where γ is the flight

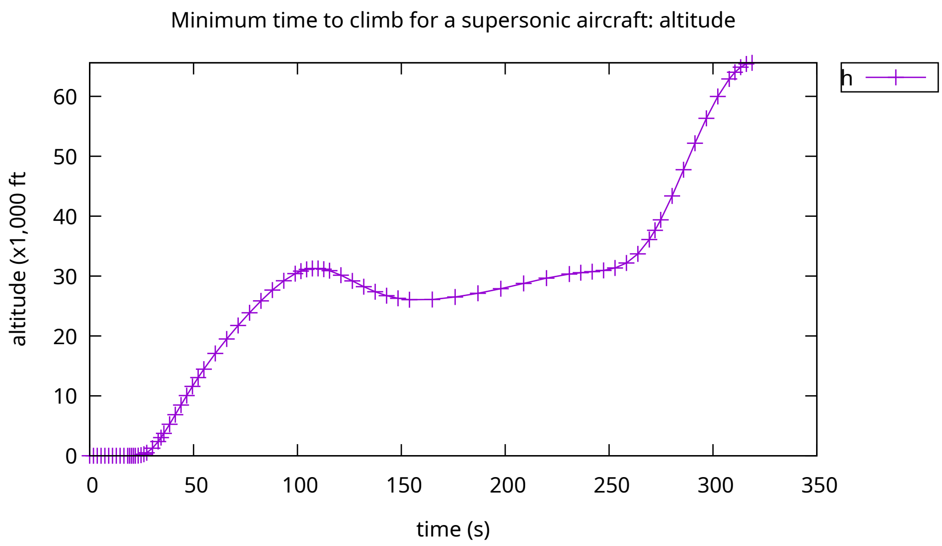

Figure 68: Altitude for minimum time to climb problem

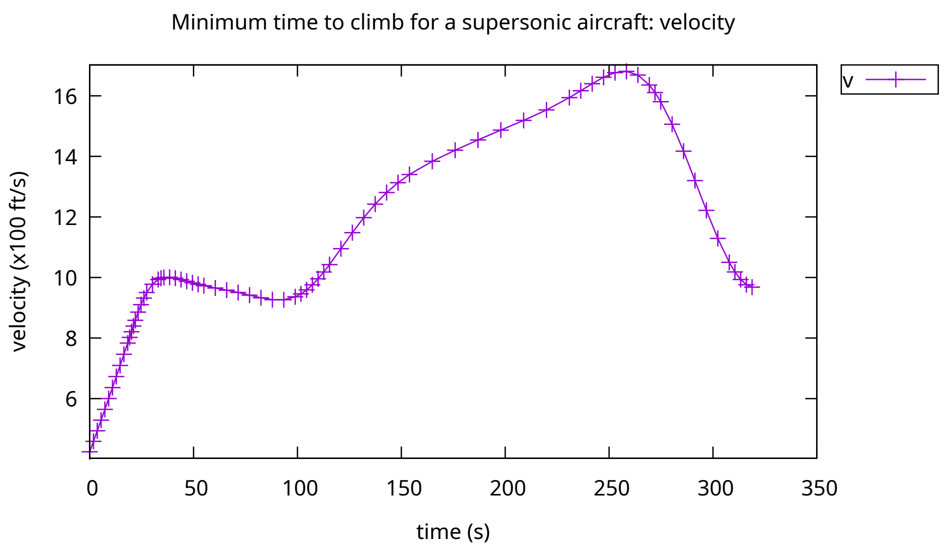

Figure 69: Velocity for minimum time to climb problem

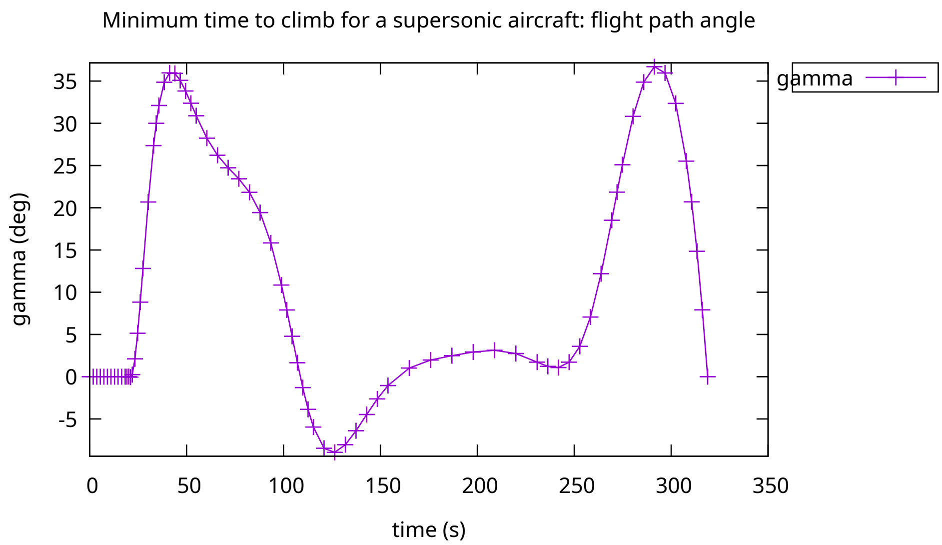

Figure 70: Flight path angle for minimum time to climb problem

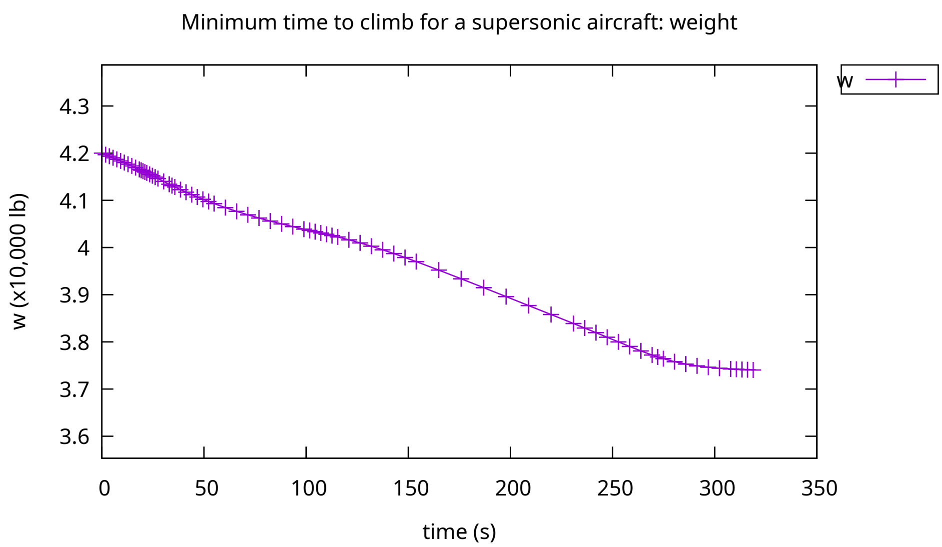

Figure 71: Weight for minimum time to climb problem

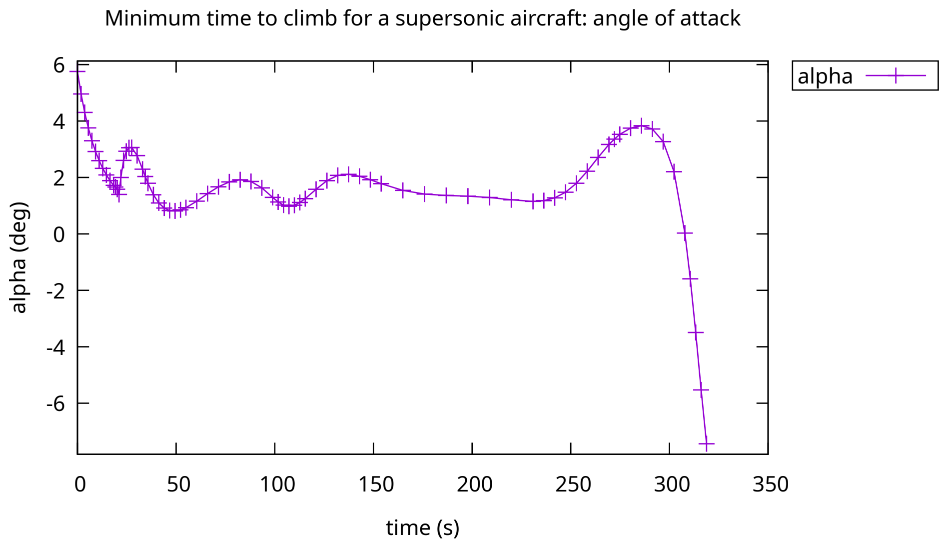

Figure 72: Angle of attack (α) for minimum time to climb problem

path angle, α is the angle of attack, V is the missile speed, x is the longitudinal position, h is the altitude, D is the axial aerodynamic force, L is the normal aerodynamic force, and T is the thrust.

The equations of motion of the missile are given by:

$$\begin{aligned}
\dot{\gamma} &= \frac{T-D}{mg}\sin\alpha + \frac{L}{mV}\cos\alpha - \frac{g\cos\gamma}{V} \\
\dot{V} &= \frac{T-D}{m}\cos\alpha - \frac{L}{m}\sin\alpha - g\cos\gamma \\
\dot{x} &= V\cos\gamma \\
\dot{h} &= V\sin\gamma
\end{aligned}$$

where

$$\begin{aligned}
D &= \frac{1}{2}C_d\rho V^2 Sref \\
C_d &= A_1\alpha^2 + A_2\alpha + A_3 \\
L &= \frac{1}{2}C_l\rho V^2 Sref \\
C_l &= B_1\alpha + B_2 \\
\rho &= C_1h^2 + C_2h + C_3
\end{aligned}$$

where all the model parameters are given in Table 6. The initial conditions for the state variables are:

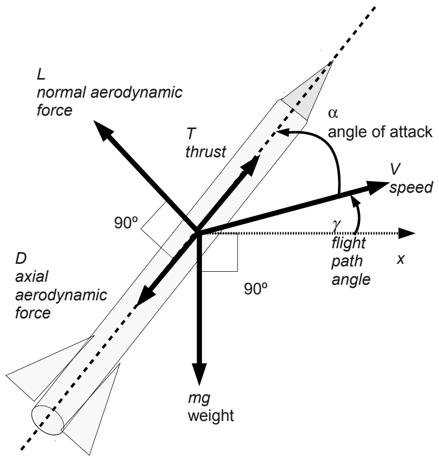

Figure 73: Ilustration of the variables associated with the missile model

Table 6: Parameters values of the missile model

Parameter	Value	Units
m	1005	kg
g	9.81	$\mathrm{m/s^2}$
S_{ref}	0.3376	$\mathrm{m^2}$
A_1	-1.9431	
A_2	-0.1499	
A_3	0.2359	
B_1	21.9	
B_2	0	
C_1	3.312×10^{-9}	$\mathrm{kg/m^5}$
C_2	-1.142×10^{-4}	$\mathrm{kg/m^4}$
C_3	1.224	$\mathrm{kg/m^3}$

$$
\begin{aligned}
\gamma(0) &= 0 \\
V(0) &= 272\mathrm{m/s} \\
x(0) &= 0\mathrm{m} \\
h(0) &= 30\mathrm{m}
\end{aligned}
$$

The terminal conditions on the states are:

$$
\begin{aligned}
\gamma(t_f) &= -\pi/2 \\
V(t_f) &= 310\mathrm{m/s} \\
x(t_f) &= 10000\mathrm{m} \\
h(t_f) &= 0\mathrm{m}
\end{aligned}
$$

The problem constraints are given by:

$$
\begin{aligned}
200 \leq & V \leq 310 \\
1000 \leq & T \leq 6000 \\
-0.3 \leq & \alpha \leq 0.3 \\
-4 \leq & \frac{L}{mg} \leq 4 \\
& h \geq 30 \text{ (for } x \leq 7500\mathrm{m}) \\
& h \geq 0 \text{ (for } x > 7500\mathrm{m})
\end{aligned}
$$

Note that the path constraints on the altitude are non-smooth. Given that non-smoothness causes problems with nonlinear programming, the constraints on the altitude were approximated by a single smooth constraint:

$$\mathcal{H}_\epsilon(x - 7500))h(t) + [1 - \mathcal{H}_\epsilon(x - 7500)][h(t) - 30] \geq 0$$

where $\mathcal{H}_\epsilon(z)$ is a smooth version of the Heaviside function, which is computed as follows:

$$\mathcal{H}_\epsilon(z) = 0.5(1 + \tanh(z/\epsilon))$$

where $\epsilon > 0$ is a small number.

The problem is solved by using automatic mesh refinement starting with 50 nodes. The final solution, which is found after six mesh refinement iterations, has 85 nodes. Figure 74 shows the missile altitude as a function of the longitudinal position. Figures 75 and 76 show, respectively, the missile speed and angle of attack as functions of time. The output from $\mathcal{PSOPT}$ is summarised in the box below.

```
PSOPT results summary
=====================

Problem:  Missile problem
CPU time (seconds): 9.833320e-01
NLP solver used:  IPOPT
PSOPT release number:  5.0.3
Date and time of this run:  Thu Mar  6 16:59:53 2025

Optimal (unscaled) cost function value:  4.091755e+01
Phase 1 endpoint cost function value: 4.091755e+01
Phase 1 integrated part of the cost:  0.000000e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 4.091755e+01
Phase 1 maximum relative local error: 4.934680e-04
NLP solver reports:  The problem has been solved!
```

29 Moon lander problem

Consider the following optimal control problem, which is known in the literature as the moon lander problem [18]. Find t_f and $T(t) \in [0, t_f]$ to minimize the cost functional

$$J = \int_0^{t_f} T(t)dt \tag{120}$$

subject to the dynamic constraints

$$\begin{array}{rcl} \dot{h} & = & v \\ \dot{v} & = & -g + T/m \\ \dot{m} & = & -T/E \end{array} \tag{121}$$

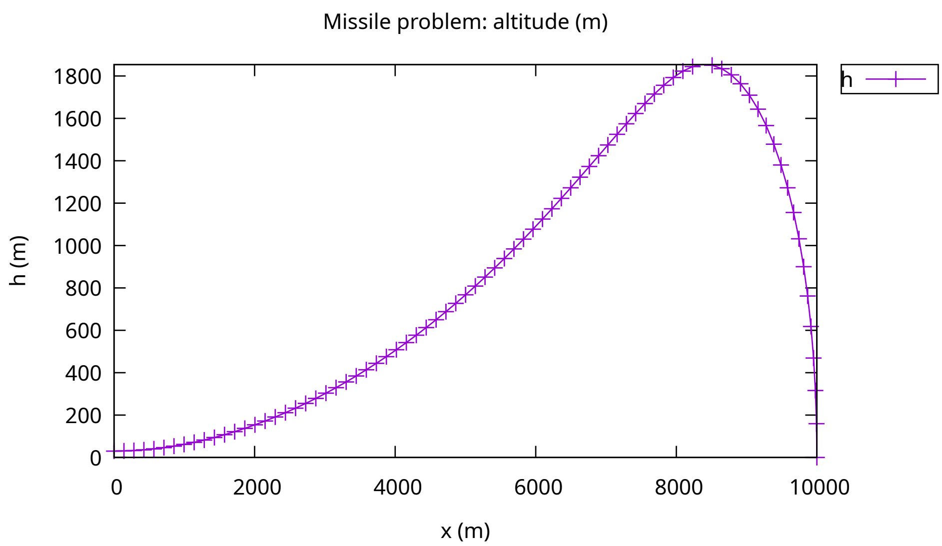

Figure 74: Missile altitude and a function of the longitudinal position

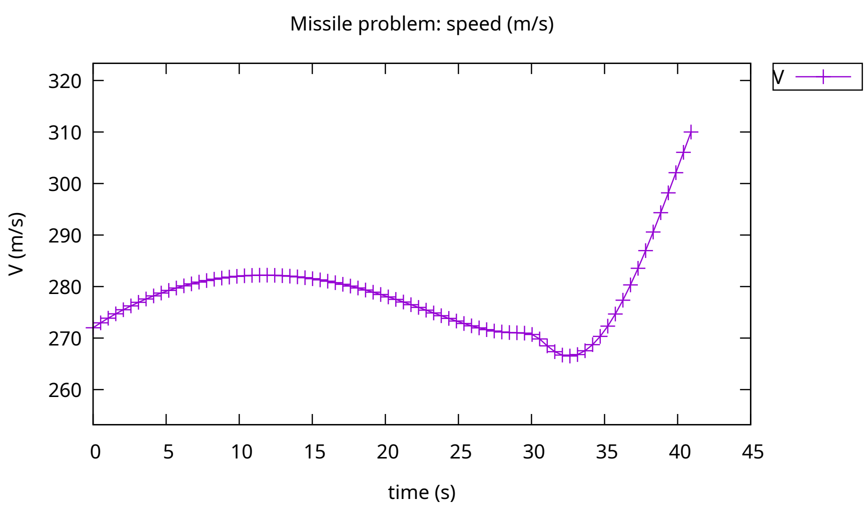

Figure 75: Missile speed as a function of time

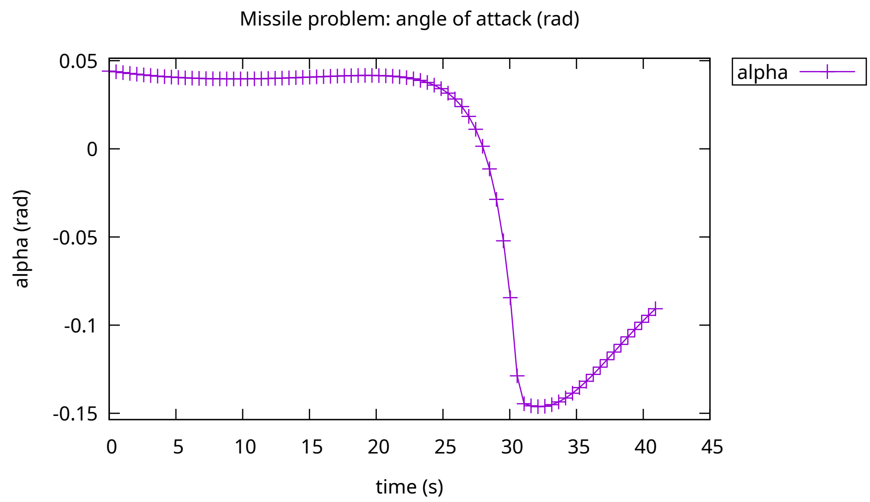

Figure 76: Missile angle of attack as a function of time

the boundary conditions:

$$\begin{aligned} h(0) &= 1 \\ v(0) &= -0.783 \\ m(0) &= 1 \\ h(t_f) &= 0.0 \\ v(t_f) &= 0.0 \end{aligned} \tag{122}$$

and the bounds

$$\begin{aligned} 0 \leq T(t) &\leq 1.227 \\ -20 \leq h(t) &\leq 20 \\ -20 \leq v(t) &\leq 20 \\ 0.01 \leq m(t) &\leq 1 \\ 0 \leq t_f &\leq 1000 \end{aligned} \tag{123}$$

where $g = 1.0$, and $E = 2.349$.

The output from $\mathcal{PSOPT}$ is summarised in the box below and shown in Figures 77 and 78, which contain the elements of the state and the control, respectively.

```
PSOPT results summary
=====================

Problem:  Moon Lander Problem
CPU time (seconds): 7.036530e-01
NLP solver used:  IPOPT
PSOPT release number:  5.0.3
```

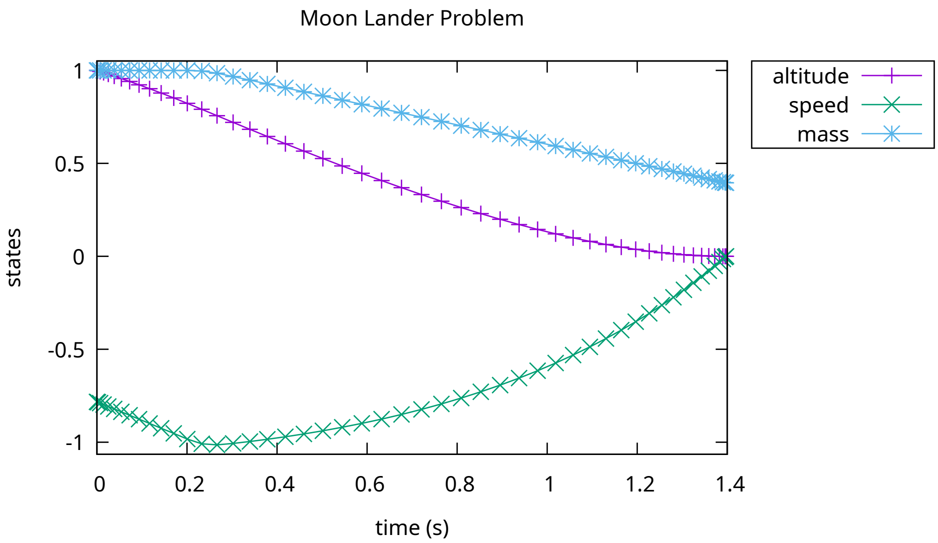

Figure 77: States for moon lander problem

```
Date and time of this run:  Thu Mar  6 17:00:07 2025

Optimal (unscaled) cost function value:  1.420484e+00
Phase 1 endpoint cost function value: 0.000000e+00
Phase 1 integrated part of the cost:  1.420484e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 1.397241e+00
Phase 1 maximum relative local error: 9.162886e-05
NLP solver reports:  The problem has been solved!
```

30 Multi-segment problem

Consider the following optimal control problem, where the optimal control has a characteristic stepped shape [11]. Find $u(t) \in [0, 3]$ to minimize the cost functional

$$J = \int_0^3 x(t)dt \tag{124}$$

subject to the dynamic constraints

$$\dot{x} = u \tag{125}$$

the boundary conditions:

$$\begin{aligned} x(0) &= 1 \\ x(3) &= 1 \end{aligned} \tag{126}$$

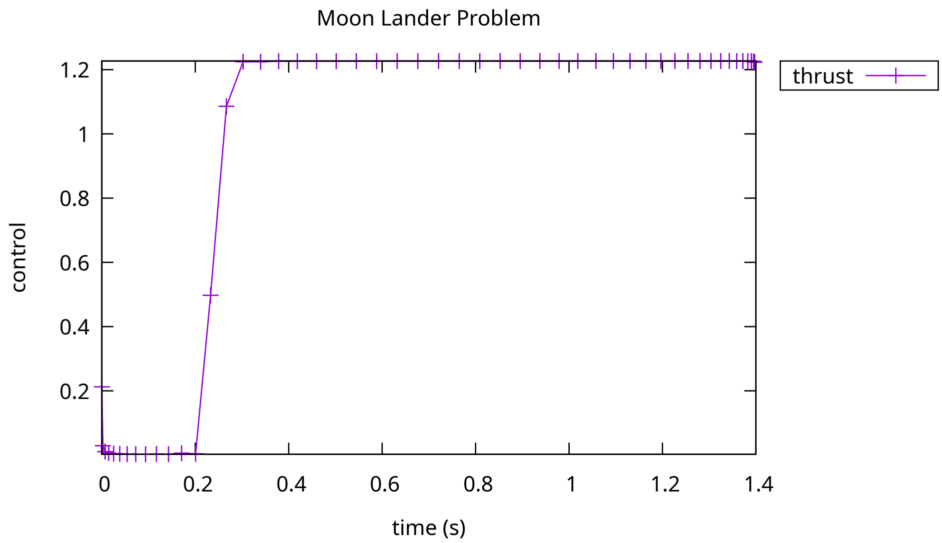

Figure 78: Control for moon lander problem

and the bounds

$$\begin{aligned} -1 \leq u(t) \quad &\leq 1 \\ x(t) \geq 0 & \end{aligned} \tag{127}$$

The analytical optimal control is given by:

$$u(t) = \begin{cases} -1, & t \in [0, 1) \\ 0, & t \in [1, 2] \\ 1, & t \in (2, 3] \end{cases} \tag{128}$$

The problem has been solved using the multi-segment paradigm. Three segments are defined in the code, such that the initial time is fixed at $t_0^{(1)} = 0$, the final time is fixed at $t_f^{(3)} = 3$, and the intermediate junction times are $t_f^{(1)} = 1$, and $t_f^{(2)} = 2$.

The C++ code that solves this problem is shown below.

```
//////////////////////////////////////////////////////////////////////////
////////////////           steps.cxx                 /////////////////////
//////////////////////////////////////////////////////////////////////////
////////////////           PSOPT example             /////////////////////
//////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////
//////// Title:  Steps problem                     ////////////////
//////// Last modified: 12 July 2009                    ////////////////
//////// Reference:      Gong, Farhoo, and Ross (2008)    ////////////////
////////                                                ////////////////
//////////////////////////////////////////////////////////////////////////
////////     Copyright (c) Victor M. Becerra, 2009       ////////////////
//////////////////////////////////////////////////////////////////////////
//////// This is part of the PSOPT software library, which ///////////////
//////// is distributed under the terms of the GNU Lesser ///////////////
//////// General Public License (LGPL)                  ////////////////
//////////////////////////////////////////////////////////////////////////
```

```
#include "psopt.h"


//////////////////////////////////////////////////////////////////////////
//////////////////  Define the end point (Mayer) cost function //////////
//////////////////////////////////////////////////////////////////////////


adouble endpoint_cost(adouble* initial_states, adouble* final_states,
                      adouble* parameters,adouble& t0, adouble& tf,
                      adouble* xad, int iphase, Workspace* workspace)
{
    return 0.0;
}

//////////////////////////////////////////////////////////////////////////
//////////////////  Define the integrand (Lagrange) cost function  //////
//////////////////////////////////////////////////////////////////////////

adouble integrand_cost(adouble* states, adouble* controls, adouble* parameters,
                     adouble& time, adouble* xad, int iphase, Workspace* workspace)
{
   adouble x = states[ 0 ];

   return (x);
}


//////////////////////////////////////////////////////////////////////////
//////////////////  Define the DAE's ////////////////////////////////////
//////////////////////////////////////////////////////////////////////////


void dae(adouble* derivatives, adouble* path, adouble* states,
         adouble* controls, adouble* parameters, adouble& time,
         adouble* xad, int iphase, Workspace* workspace)
{

    adouble u = controls[CINDEX(1)];

    derivatives[ 0 ]  = u;

}

////////////////////////////////////////////////////////////////////////////
//////////////////  Define the events function ////////////////////////////
////////////////////////////////////////////////////////////////////////////

void events(adouble* e, adouble* initial_states, adouble* final_states,
            adouble* parameters,adouble& t0, adouble& tf, adouble* xad,
            int iphase, Workspace* workspace)

{

    adouble x1_i  = initial_states[ 0 ];

    adouble x1_f  = final_states[   0 ];


   if ( iphase==1 ) {
    e[ 0 ] =  x1_i;
   }
   else if ( iphase==3 ) {
    e[ 0 ] =  x1_f;
   }

}



///////////////////////////////////////////////////////////////////////////
//////////////////  Define the phase linkages function ///////////////////
///////////////////////////////////////////////////////////////////////////

void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{

}



////////////////////////////////////////////////////////////////////////////
```

```
//////////////////  Define the main routine ///////////////////////////////
///////////////////////////////////////////////////////////////////////////

int main(void)
{

///////////////////////////////////////////////////////////////////////////
//////////////////  Declare key structures ////////////////////////////////
///////////////////////////////////////////////////////////////////////////

    Alg  algorithm;
    Sol  solution;
    Prob problem;
    MSdata msdata;

///////////////////////////////////////////////////////////////////////////
//////////////////  Register problem name  ////////////////////////////////
///////////////////////////////////////////////////////////////////////////

    problem.name                  = "Steps problem";
    problem.outfilename                = "steps.txt";

///////////////////////////////////////////////////////////////////////////
///////////  Define problem level constants & do level 1 setup ///////////
///////////////////////////////////////////////////////////////////////////

    msdata.nsegments       = 3;
    msdata.nstates         = 1;
    msdata.ncontrols       = 1;
    msdata.nparameters     = 0;
    msdata.npath           = 0;
    msdata.ninitial_events = 1;
    msdata.nfinal_events   = 1;
    msdata.nodes           << 20; // nodes per segment

    multi_segment_setup(problem, algorithm, msdata );

///////////////////////////////////////////////////////////////////////////
//////////////////  Enter problem bounds information //////////////////////
///////////////////////////////////////////////////////////////////////////


    problem.phases(1).bounds.lower.controls(0) = -1.0;
    problem.phases(1).bounds.upper.controls(0) =  1.0;
    problem.phases(1).bounds.lower.states(0) = 0.0;
    problem.phases(1).bounds.upper.states(0) = 5.0;
    problem.phases(1).bounds.lower.events(0) = 1.0;
    problem.phases(3).bounds.lower.events(0) = 1.0;


    problem.phases(1).bounds.upper.events=problem.phases(1).bounds.lower.events;
    problem.phases(3).bounds.upper.events=problem.phases(3).bounds.lower.events;



    problem.phases(1).bounds.lower.StartTime    = 0.0;
    problem.phases(1).bounds.upper.StartTime    = 0.0;

    problem.phases(3).bounds.lower.EndTime      = 3.0;
    problem.phases(3).bounds.upper.EndTime      = 3.0;

//    problem.bounds.lower.times = "[0.0, 1.0, 2.0, 3.0]";
//    problem.bounds.upper.times = "[0.0, 1.0, 2.0, 3.0]";

problem.bounds.lower.times.resize(1,4);
problem.bounds.upper.times.resize(1,4);

    problem.bounds.lower.times << 0.0, 1.0, 2.0, 3.0;
    problem.bounds.upper.times << 0.0, 1.0, 2.0, 3.0;

    auto_phase_bounds(problem);



///////////////////////////////////////////////////////////////////////////
//////////////////  Register problem functions  ///////////////////////////
///////////////////////////////////////////////////////////////////////////


    problem.integrand_cost  = &integrand_cost;
    problem.endpoint_cost  = &endpoint_cost;
    problem.dae             = &dae;
```

```
    problem.events  = &events;
    problem.linkages = &linkages;

////////////////////////////////////////////////////////////////////////////
///////////////////  Define & register initial guess ///////////////////////
////////////////////////////////////////////////////////////////////////////

    int nnodes      = problem.phases(1).nodes(0);
    int ncontrols                       = problem.phases(1).ncontrols;
    int nstates                         = problem.phases(1).nstates;

    MatrixXd state_guess     =  zeros(nstates,nnodes);
    MatrixXd control_guess   =  zeros(ncontrols,nnodes);
    MatrixXd time_guess     =  linspace(0.0,3.0,nnodes);
    MatrixXd param_guess;



    state_guess   = linspace(1.0, 1.0, nnodes);

    control_guess = zeros(1,nnodes);


    auto_phase_guess(problem, control_guess, state_guess, param_guess, time_guess);

////////////////////////////////////////////////////////////////////////////
///////////////////  Enter algorithm options  //////////////////////////////
////////////////////////////////////////////////////////////////////////////


    algorithm.nlp_iter_max                = 1000;
    algorithm.nlp_tolerance               = 1.e-6;
    algorithm.nlp_method                  = "IPOPT";
    algorithm.scaling                     = "automatic";
    algorithm.derivatives                 = "automatic";
    algorithm.hessian                     = "exact";
    algorithm.mesh_refinement             = "automatic";
    algorithm.ode_tolerance               = 1.e-5;


////////////////////////////////////////////////////////////////////////////
///////////////////  Now call PSOPT to solve the problem   //////////////////
////////////////////////////////////////////////////////////////////////////

    psopt(solution, problem, algorithm);

////////////////////////////////////////////////////////////////////////////
///////////  Extract relevant variables from solution structure   //////////
////////////////////////////////////////////////////////////////////////////


    MatrixXd x, u, t, x_ph1, u_ph1, t_ph1, x_ph2, u_ph2, t_ph2, x_ph3, u_ph3, t_ph3;

    x_ph1      = solution.get_states_in_phase(1);
    u_ph1      = solution.get_controls_in_phase(1);
    t_ph1      = solution.get_time_in_phase(1);

    x_ph2      = solution.get_states_in_phase(2);
    u_ph2      = solution.get_controls_in_phase(2);
    t_ph2      = solution.get_time_in_phase(2);

    x_ph3      = solution.get_states_in_phase(3);
    u_ph3      = solution.get_controls_in_phase(3);
    t_ph3      = solution.get_time_in_phase(3);


    x.resize(1, length(t_ph1)+length(t_ph2)+length(t_ph3) );
    u.resize(1, length(t_ph1)+length(t_ph2)+length(t_ph3) );
    t.resize(1, length(t_ph1)+length(t_ph2)+length(t_ph3) );

    x << x_ph2, x_ph2, x_ph3;
    u << u_ph1, u_ph2, u_ph3;
    t << t_ph1, t_ph2, t_ph3;


////////////////////////////////////////////////////////////////////////////
///////////  Save solution data to files if desired ////////////////////////
////////////////////////////////////////////////////////////////////////////

    Save(x,"x.dat");
    Save(u,"u.dat");
    Save(t,"t.dat");
```

```
////////////////////////////////////////////////////////////////////////////
///////////  Plot some results if desired (requires gnuplot) ///////////////
////////////////////////////////////////////////////////////////////////////

    plot(t,x,problem.name+": state","time (s)", "x", "x");

    plot(t,u,problem.name+": control","time (s)", "u", "u");

    plot(t,x,problem.name+": state","time (s)", "x", "x",
                                "pdf", "steps_state.pdf");


    plot(t,u,problem.name+": control","time (s)", "u", "u",
                                "pdf", "steps_control.pdf");


}

////////////////////////////////////////////////////////////////////////////
///////////////////////      END OF FILE     ///////////////////////////////
////////////////////////////////////////////////////////////////////////////
```

The output from $\mathcal{PSOPT}$ is summarised in the box below and shown in Figures 79 and 80, which contain the elements of the state and the control, respectively.

```
PSOPT results summary
=====================

Problem:  Steps problem
CPU time (seconds): 1.039730e-01
NLP solver used:  IPOPT
PSOPT release number:  5.0.3
Date and time of this run:  Thu Mar  6 17:04:00 2025

Optimal (unscaled) cost function value:  1.000000e+00
Phase 1 endpoint cost function value: 0.000000e+00
Phase 1 integrated part of the cost:  5.000001e-01
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 1.000000e+00
Phase 1 maximum relative local error: 4.163987e-08
Phase 2 endpoint cost function value: 0.000000e+00
Phase 2 integrated part of the cost:  5.787927e-08
Phase 2 initial time: 1.000000e+00
Phase 2 final time: 2.000000e+00
Phase 2 maximum relative local error: 2.914144e-07
Phase 3 endpoint cost function value: 0.000000e+00
Phase 3 integrated part of the cost:  5.000001e-01
Phase 3 initial time: 2.000000e+00
Phase 3 final time: 3.000000e+00
Phase 3 maximum relative local error: 4.164124e-08
NLP solver reports:  The problem has been solved!
```

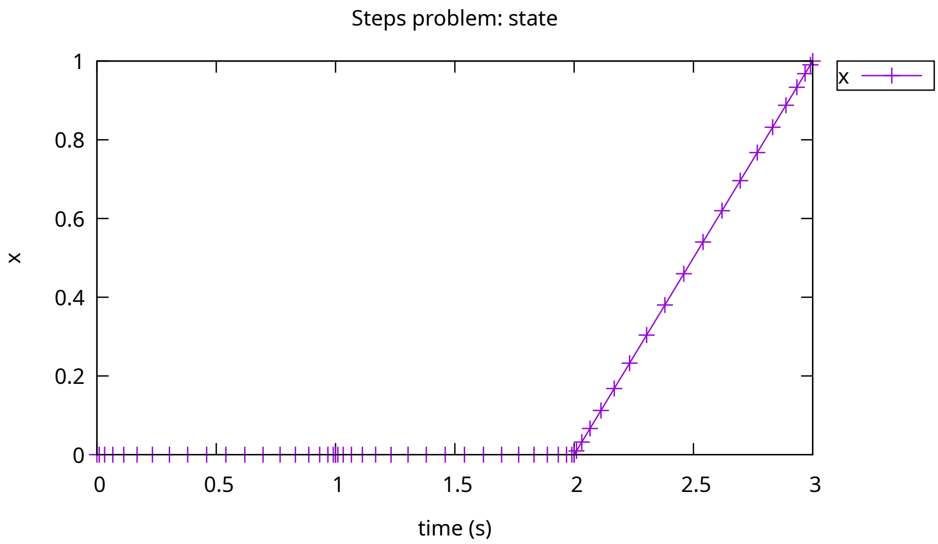

Figure 79: State trajectory for the multi-segment problem

31 Notorious parameter estimation problem

Consider the following parameter estimation problem, which is known to be challenging to single-shooting methods because of internal instability of the differential equations [19]. Find $p \in \Re$ to minimize

$$J = \sum_{i=1}^{200} (y_1(t_i) - \tilde{y}_1(i))^2 + (y_2(t_i) - \tilde{y}_2(i))^2 \tag{129}$$

subject to the dynamic constraints

$$\begin{aligned} \dot{y}_1 &= y_2 \\ \dot{y}_2 &= \mu^2 y_1 - (\mu^2 + p^2)\sin(pt) \end{aligned} \tag{130}$$

where $\mu = 60.0$, $y_1(0) = 0$, $y_2(0) = \pi$. The parameter estimation facilities of $\mathcal{PSOPT}$ are used in this example. In this case, the observations function is:

$$g(x(\theta_k), u(\theta_k), p, \theta_k) = [y_1(\theta_k) \;\; y_2(\theta_k)]^T$$

The C++ code that solves this problem is shown below. The code includes the generation of the measurement vectors $\tilde{y}_1$, and $\tilde{y}_2$ by adding Gaussian noise with standard deviation

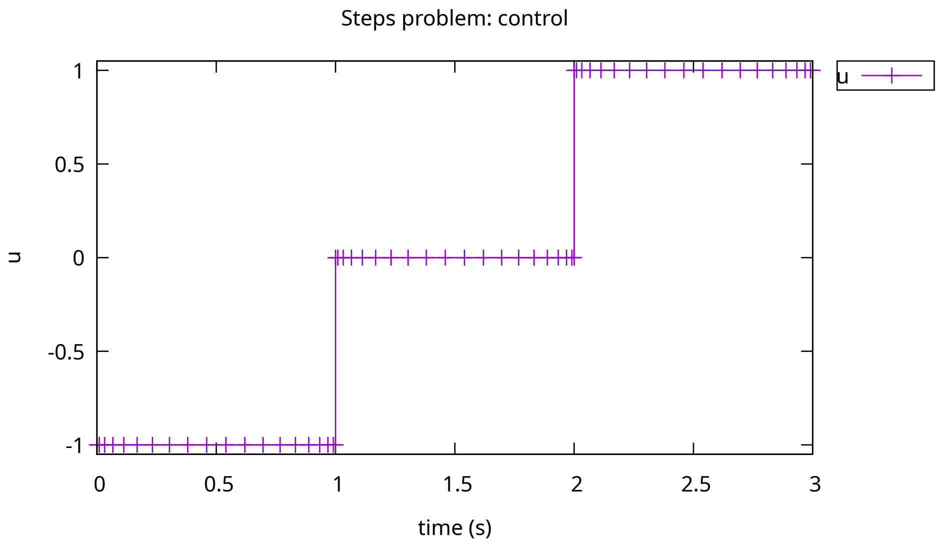

Figure 80: Control trajectory for the multi-segment problem

0.05 to the exact solution of the problem with $p = \pi$, which is given by:

$$y_1(t) = \sin(\pi t)$$
$$y_2(t) = \pi \cos(\pi t)$$

The code also defines the vector of sampling instants $\theta_i, i = 1, \ldots, 200$ as a uniform random samples in the interval $[0, 1]$.

```
//////////////////////////////////////////////////////////////////////////
//////////////////        notorious.cxx      ///////////////////////////
//////////////////////////////////////////////////////////////////////////
////////////////          PSOPT  Example           ////////////////////
//////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////
//////// Title:  Bock's notorious parameter estimation problem ///////////
//////// Last modified: 08 April 2011                  ////////////////
//////// Reference:     Schittkowskit (2002)          ////////////////
//////// (See PSOPT handbook for full reference)         ///////////////
//////////////////////////////////////////////////////////////////////////
////////     Copyright (c) Victor M. Becerra, 2009       ////////////////
//////////////////////////////////////////////////////////////////////////
//////// This is part of the PSOPT software library, which ///////////////
//////// is distributed under the terms of the GNU Lesser ////////////////
//////// General Public License (LGPL)                  ////////////////
//////////////////////////////////////////////////////////////////////////

#include "psopt.h"

using namespace PSOPT;



//////////////////////////////////////////////////////////////////////////
//////////////////  Define the observation function //////////
//////////////////////////////////////////////////////////////////////////
```

```
void  observation_function( adouble* observations,
                            adouble* states, adouble* controls,
                            adouble* parameters, adouble& time, int k,
                            adouble* xad, int iphase, Workspace* workspace)
{

      observations[ 0 ] = states[ 0 ];
      observations[ 1 ] = states[ 1 ];
}




//////////////////////////////////////////////////////////////////////////
//////////////////  Define the DAE's ////////////////////////////////////
//////////////////////////////////////////////////////////////////////////


void dae(adouble* derivatives, adouble* path, adouble* states,
         adouble* controls, adouble* parameters, adouble& time,
         adouble* xad, int iphase, Workspace* workspace)
{

   adouble x1 = states[ 0 ];
   adouble x2 = states[ 1 ];


   adouble p  = parameters[ 0 ];
   adouble t  = time;

   double mu = 60.0;

   derivatives[0] =  x2;
   derivatives[1] =  mu*mu*x1 - (mu*mu + p*p)*sin(p*t);


}

////////////////////////////////////////////////////////////////////////////
//////////////////  Define the events function ////////////////////////////
////////////////////////////////////////////////////////////////////////////

void events(adouble* e, adouble* initial_states, adouble* final_states,
            adouble* parameters,adouble& t0, adouble& tf, adouble* xad,
            int iphase, Workspace* workspace)
{

   e[ 0 ] =  initial_states[0];
   e[ 1 ] =  initial_states[1];


}


///////////////////////////////////////////////////////////////////////////
//////////////////  Define the phase linkages function ///////////////////
///////////////////////////////////////////////////////////////////////////

void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{
  // No linkages as this is a single phase problem
}

using namespace std;

////////////////////////////////////////////////////////////////////////////
//////////////////  Define the main routine ///////////////////////////////
////////////////////////////////////////////////////////////////////////////

int main(void)
{
   int nobs =200;

   // Generate true solution at sampling points and add noise

   double sigma = 0.05;

   MatrixXd y1m, y2m;


//   theta = randu(1,nobs);
```

```
   MatrixXd noise1= GaussianRandom(1,nobs);

   MatrixXd noise2= GaussianRandom(1,nobs);

   MatrixXd theta = (MatrixXd::Random(1,nobs)+ones(1,nobs))/2.0;

   sort(theta);

   MatrixXd ss = (pi*theta).array().sin();
   MatrixXd cc = (pi*theta).array().cos();

   y1m =  ss + sigma*noise1;

   y2m =  pi*cc + sigma*noise2;



////////////////////////////////////////////////////////////////////////////
///////////////////  Declare key structures ////////////////////////////////
////////////////////////////////////////////////////////////////////////////

    Alg  algorithm;
    Sol  solution;
    Prob problem;

////////////////////////////////////////////////////////////////////////////
///////////////////  Register problem name  ////////////////////////////////
////////////////////////////////////////////////////////////////////////////

    problem.name           = "Bocks notorious parameter estimation problem";
    problem.outfilename                 = "notorious.txt";

////////////////////////////////////////////////////////////////////////////
////////////  Define problem level constants & do level 1 setup ////////////
////////////////////////////////////////////////////////////////////////////

    problem.nphases    = 1;
    problem.nlinkages                    = 0;

    psopt_level1_setup(problem);


/////////////////////////////////////////////////////////////////////////////
/////////   Define phase related information & do level 2 setup ////////////
/////////////////////////////////////////////////////////////////////////////

    problem.phases(1).nstates    = 2;
    problem.phases(1).ncontrols  = 0;
    problem.phases(1).nevents    = 2;
    problem.phases(1).npath      = 0;
    problem.phases(1).nparameters         = 1;
    problem.phases(1).nodes          << 80;
    problem.phases(1).nobserved             = 2;
    problem.phases(1).nsamples              = nobs;


    psopt_level2_setup(problem, algorithm);

////////////////////////////////////////////////////////////////////////////
////////////  Enter estimation information                      ////////////
////////////////////////////////////////////////////////////////////////////


    problem.phases(1).observation_nodes      = theta;
    problem.phases(1).observations           << y1m,
                                                y2m;



////////////////////////////////////////////////////////////////////////////
///////////////////  Enter problem bounds information //////////////////////
////////////////////////////////////////////////////////////////////////////


    problem.phases(1).bounds.lower.states(0) =  -10.0;
    problem.phases(1).bounds.lower.states(1) =  -100.0;



    problem.phases(1).bounds.upper.states(0) =  10.0;
    problem.phases(1).bounds.upper.states(1) =  100.0;
```

```
    problem.phases(1).bounds.lower.parameters(0) =  -10.0;
    problem.phases(1).bounds.upper.parameters(0) =   10.0;

    problem.phases(1).bounds.lower.events(0) =  0.0;
    problem.phases(1).bounds.upper.events(0) =  0.0;

    problem.phases(1).bounds.lower.events(1) =  pi;
    problem.phases(1).bounds.upper.events(1) =  pi;



    problem.phases(1).bounds.lower.StartTime    = 0.0;
    problem.phases(1).bounds.upper.StartTime    = 0.0;

    problem.phases(1).bounds.lower.EndTime      = 1.0;
    problem.phases(1).bounds.upper.EndTime      = 1.0;

////////////////////////////////////////////////////////////////////////////
///////////////////  Register problem functions  ///////////////////////////
////////////////////////////////////////////////////////////////////////////

    problem.dae  = &dae;
    problem.events  = &events;
    problem.linkages = &linkages;
    problem.observation_function  = & observation_function;

////////////////////////////////////////////////////////////////////////////
///////////////////  Define & register initial guess ///////////////////////
////////////////////////////////////////////////////////////////////////////

    int nnodes = problem.phases(1).nodes(0);


    problem.phases(1).guess.states         = zeros(2,nnodes);
    problem.phases(1).guess.time           = linspace(0.0, 1.0, nnodes);
    problem.phases(1).guess.parameters(0)  = 0.0;


////////////////////////////////////////////////////////////////////////////
///////////////////  Enter algorithm options  //////////////////////////////
////////////////////////////////////////////////////////////////////////////

    algorithm.nlp_method                  = "IPOPT";
    algorithm.scaling                     = "automatic";
    algorithm.derivatives                 = "automatic";
    algorithm.collocation_method          = "trapezoidal";
    algorithm.nlp_iter_max                =  200;
    algorithm.nlp_tolerance               = 1.e-4;
//    algorithm.mesh_refinement             = "automatic";
//    algorithm.ode_tolerance               = 1.e-6;



////////////////////////////////////////////////////////////////////////////
///////////////////  Now call PSOPT to solve the problem   /////////////////
////////////////////////////////////////////////////////////////////////////


    psopt(solution, problem, algorithm);

////////////////////////////////////////////////////////////////////////////
///////////////////  Declare DMatrix objects to store results //////////////
////////////////////////////////////////////////////////////////////////////

    DMatrix states, x1, x2, p, t;


////////////////////////////////////////////////////////////////////////////
///////////  Extract relevant variables from solution structure   //////////
////////////////////////////////////////////////////////////////////////////

    states = solution.get_states_in_phase(1);
    t      = solution.get_time_in_phase(1);
    p      = solution.get_parameters_in_phase(1);
    x1     = states.row(0);
    x2     = states.row(1);


////////////////////////////////////////////////////////////////////////////
///////////  Save solution data to files if desired ////////////////////////
////////////////////////////////////////////////////////////////////////////
```

```
    Save(states,"states.dat");
    Save(t,"t.dat");
    Print(p,"Estimated parameter");
    printf( "\nParameter error %e\n", abs(p(0)-pi) );


////////////////////////////////////////////////////////////////////////////
//////////  Plot some results if desired (requires gnuplot) ///////////////
////////////////////////////////////////////////////////////////////////////

    plot(theta,y1m,t,x1,problem.name, "time (s)", "observed x1", "x1m x1");
    plot(theta,y2m,t,x2,problem.name, "time (s)", "observed x2", "x2m x2");



}

////////////////////////////////////////////////////////////////////////////
///////////////////////      END OF FILE     ///////////////////////////////
////////////////////////////////////////////////////////////////////////////
```

The output from $\mathcal{PSOPT}$ is summarized in the box below. The optimal parameter found was $p = 3.141180$, which is an approximation of π with an error of the order of 10^{-4}. The 95% confidence interval of the estimated parameter is $[3.132363, 3.149998]$.

```
PSOPT results summary
=====================

Problem:  Bocks notorious parameter estimation problem
CPU time (seconds): 3.488590e-01
NLP solver used:  IPOPT
PSOPT release number:  5.0.3
Date and time of this run:  Thu Mar  6 17:00:34 2025

Optimal (unscaled) cost function value:  8.806615e-01
Phase 1 endpoint cost function value: 8.806615e-01
Phase 1 integrated part of the cost:  0.000000e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 1.000000e+00
Phase 1 maximum relative local error: 5.104843e-04
NLP solver reports:  The problem has been solved!
```

32 Predator-prey parameter estimation problem

This is a well known model that describes the behaviour of predator and prey species of an ecological system. The Letka-Volterra model system consist of two differential equations [19].

Table 7: Estimated parameter values and 95 percent statistical confidence limits on estimated parameters

Parameter	Low Confidence Limit	Value	High Confidence Limit
p_1	7.166429e-01	9.837490e-01	1.250855e+00
p_2	7.573469e-01	9.803930e-01	1.203439e+00
p_3	7.287846e-01	1.016900e+00	1.305015e+00
p_4	6.914964e-01	1.022702e+00	1.353909e+00

The dynamic equations are given by:

$$\begin{aligned} \dot{x}_1 &= -p_1 x_1 + p_2 x_1 x_2 \\ \dot{x}_2 &= p_3 x_2 - p_4 x_1 x_2 \end{aligned} \tag{131}$$

with boundary condition:

$$\begin{aligned} x_1(0) &= 0.4 \\ x_2(0) &= 1 \end{aligned}$$

The observation functions are:

$$\begin{aligned} g_1 &= x_1 \\ g_2 &= x_2 \end{aligned} \tag{132}$$

The measured data, with consists of $n_s = 10$ samples over the interval $t \in [0, 10]$, was constructed from simulations with parameter values $[p_1, p_2, p_3, p_4] = [1, 1, 1, 1]$ with added noise. The weights of both observations are the same and equal to one.

The solution is found using local discretisation (trapezoidal, Hermite-Simpson) and automatic mesh refinement, starting with 20 grid points with ODE tolerance 10^{-4}. The estimated parameter values and their 95% confidence limits are shown in Table 32. Figure 81 shows the observations as well as the estimated values of variables x_1 and x_2. The mesh statistics can be seen in Table 8

33 Rayleigh problem with mixed state-control path constraints

Consider the following optimal control problem, which involves a path constraint in which the control and the state appear explicitly [4]. Find $u(t) \in [0, t_f]$ to minimize the cost functional

$$J = \int_0^{t_f} \left[x_1(t)^2 + u(t)^2 \right] \mathrm{d}t \tag{133}$$

subject to the dynamic constraints

$$\begin{aligned} \dot{x}_1 &= x_2 \\ \dot{x}_2 &= -x_1 + x_2(1.4 - p x_2^2) + 4u \sin(\theta) \end{aligned} \tag{134}$$

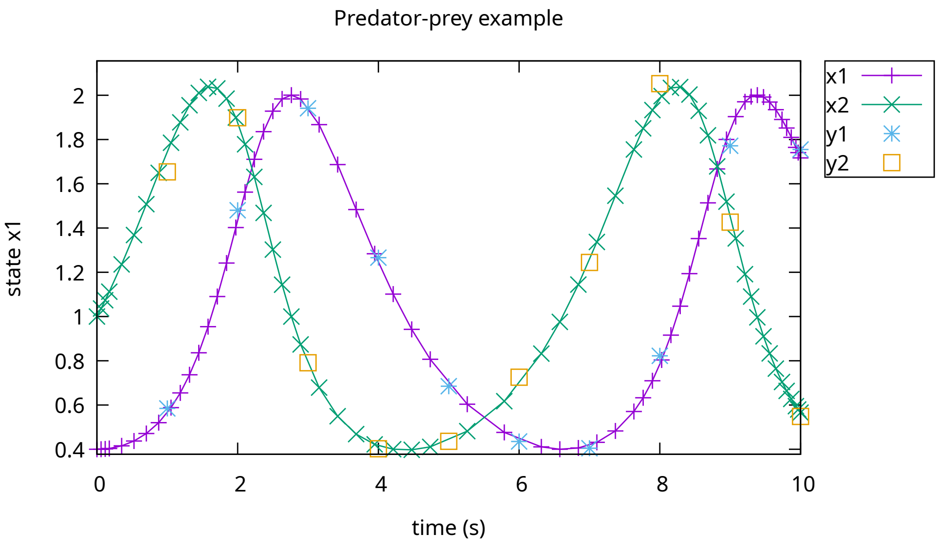

Figure 81: Observations y_1, y_2 and estimated states $x_1(t)$ and $x_2(t)$

Table 8: Mesh refinement statistics: Predator-prey example

Iter	DM	M	NV	NC	OE	CE	JE	HE	RHS	$\epsilon_{\max}$	$\mathrm{CPU_a}$
1	TRP	20	46	43	20	20	20	0	780	1.615e-02	4.367e-02
2	TRP	28	62	59	11	11	11	0	605	8.919e-03	2.150e-02
3	H-S	39	84	81	11	11	11	0	1265	1.670e-03	2.211e-02
4	H-S	54	114	111	16	16	14	0	2560	1.268e-04	3.434e-02
5	H-S	61	128	125	9	9	9	0	1629	4.884e-05	2.119e-02
$\mathrm{CPU_b}$	-	-	-	-	-	-	-	-	-	-	1.777e-01
-	-	-	-	-	67	67	65	0	6839	-	3.205e-01

Key: Iter=iteration number, DM= discretization method, M=number of nodes, NV=number of variables, NC=number of constraints, OE=objective evaluations, CE = constraint evaluations, JE = Jacobian evaluations, HE = Hessian evaluations, RHS = ODE right hand side evaluations, $\epsilon_{\max}$ = maximum relative ODE error, $\mathrm{CPU_a}$ = CPU time in seconds spent by NLP algorithm, $\mathrm{CPU_b}$ = additional CPU time in seconds spent by PSOPT

The path constraint:

$$u + \frac{x_1}{6} \leq 0 \tag{135}$$

and the boundary conditions:

$$\begin{aligned} x_1(0) &= -5 \\ x_2(0) &= -5 \end{aligned} \tag{136}$$

where $t_f = 4.5$, and $p = 0.14$.

The output from $\mathcal{PSOPT}$ is summarised in the box below and shown in Figures 82, 83, 85, 85, 85, which show, respectively, the trajectories of the states, control, costates and path constraint multiplier. The results are comparable to those presented by [4].

```
PSOPT results summary
=====================

Problem:  Rayleigh problem
CPU time (seconds): 2.895580e-01
NLP solver used:  IPOPT
PSOPT release number:  5.0.3
Date and time of this run:  Thu Mar  6 17:02:06 2025

Optimal (unscaled) cost function value:  4.477625e+01
Phase 1 endpoint cost function value: 0.000000e+00
Phase 1 integrated part of the cost:  4.477625e+01
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 4.500000e+00
Phase 1 maximum relative local error: 2.329140e-03
NLP solver reports:  The problem has been solved!
```

34 Obstacle avoidance problem

Consider the following optimal control problem, which involves finding an optimal trajectory for a particle to travel from A to B while avoiding two forbidden regions [18]. Find $\theta(t) \in [0, t_f]$ to minimize the cost functional

$$J = \int_0^{t_f} \left[\dot{x}(t)^2 + \dot{y}(t)^2\right] dt \tag{137}$$

subject to the dynamic constraints

$$\begin{aligned} \dot{x} &= V\cos(\theta) \\ \dot{y} &= V\sin(\theta) \end{aligned} \tag{138}$$

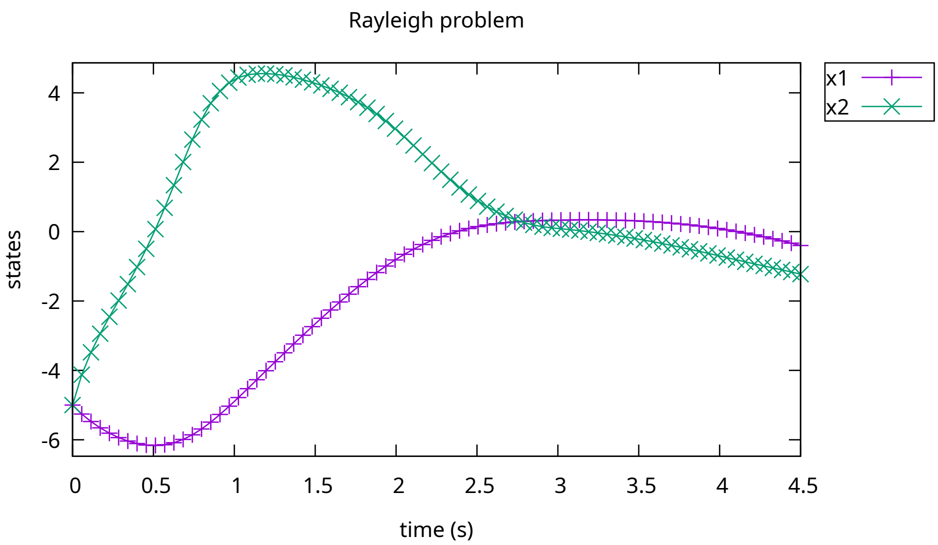

Figure 82: States for Rayleigh problem

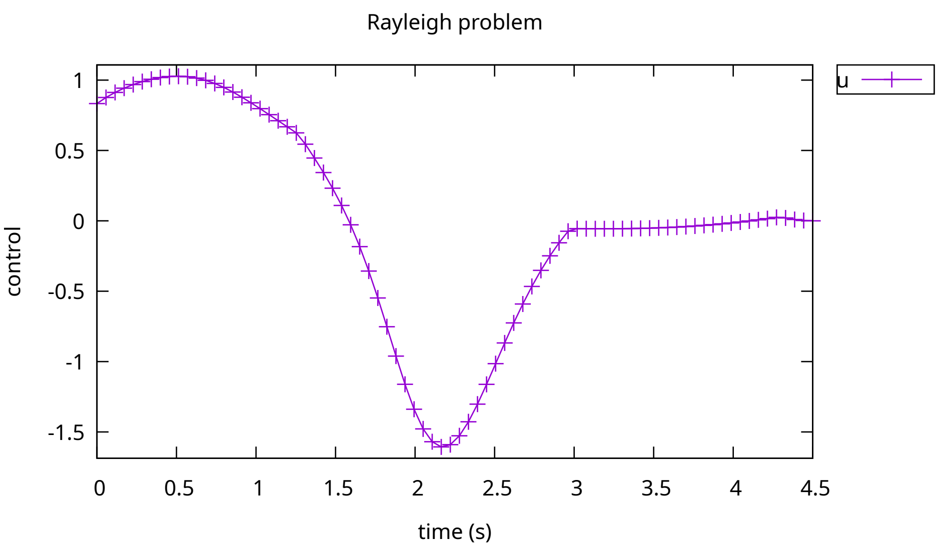

Figure 83: Optimal control for Rayleigh problem

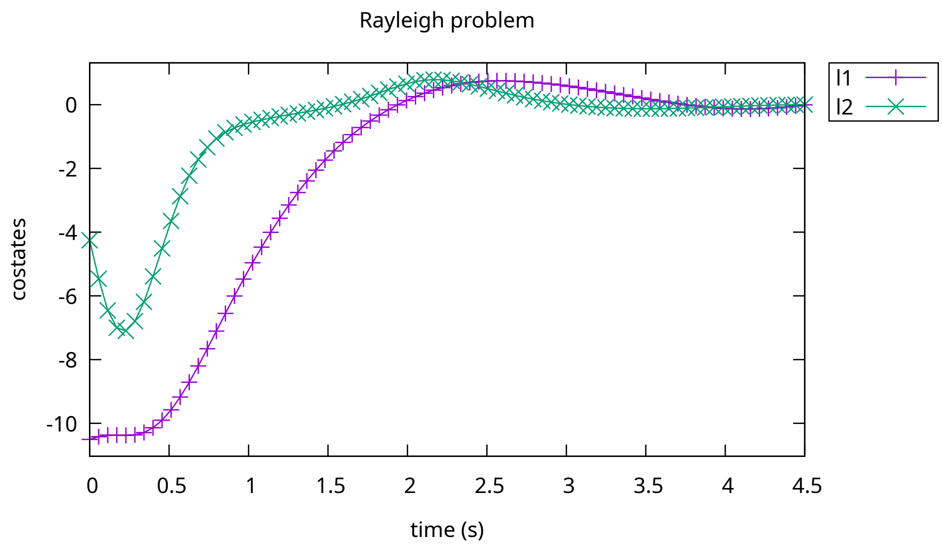

Figure 84: Costates for Rayleigh problem

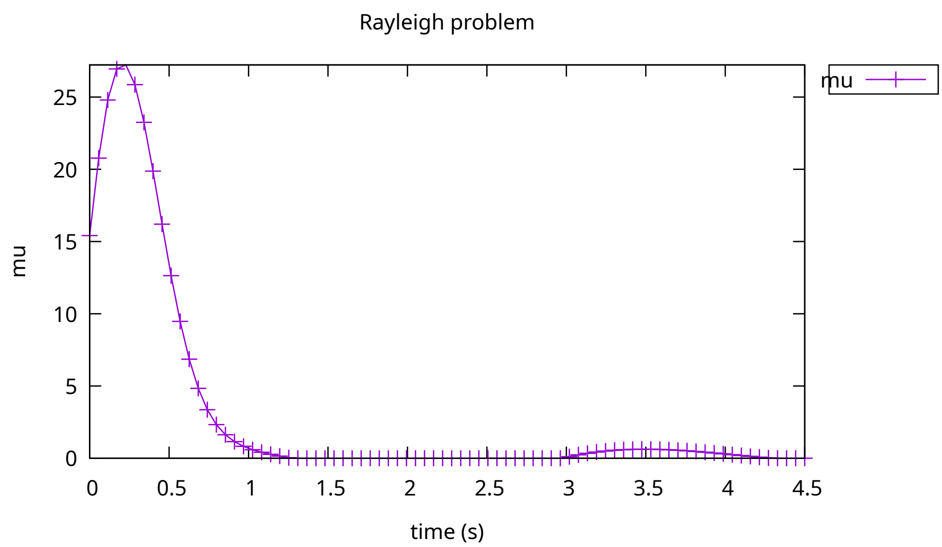

Figure 85: Path constraint multiplier for Rayleigh problem

The path constraints:

$$
\begin{aligned}
(x(t)-0.4)^2+(y(t)-0.5)^2 &\geq 0.1 \\
(x(t)-0.8)^2+(y(t)-1.5)^2 &\geq 0.1,
\end{aligned} \tag{139}
$$

and the boundary conditions:

$$
\begin{array}{rcl}
x(0) & = & 0 \\
y(0) & = & 0 \\
x(t_f) & = & 1.2 \\
y(t_f) & = & 1.6
\end{array} \tag{140}
$$

where $t_f = 1.0$, and $V = 2.138$.

The output from $\mathcal{PSOPT}$ is summarised in the box below and shown in Figure 86, which illustrates the optimal (x, y) trajectory of the particle.

```
PSOPT results summary
=====================

Problem:  Obstacle avoidance problem
CPU time (seconds): 1.729016e+01
NLP solver used:  IPOPT
PSOPT release number:  5.0.3
Date and time of this run:  Thu Mar  6 17:01:05 2025

Returned (unscaled) cost function value:  4.571044e+00
Phase 1 endpoint cost function value:  0.000000e+00
Phase 1 integrated part of the cost:  4.571044e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 1.000000e+00
Phase 1 maximum relative local error 1.593798e-02
NLP solver reports:  *** The problem FAILED! - see screen output
```

35 Reorientation of an asymmetric rigid body

Consider the following optimal control problem, which consists of the reorientation of an asymmetric rigid body in minimum time [4]. Find t_f, $\hat{\mathbf{u}}(t) = [u_1(t), u_2(t), u_3(t), q_4(t)]^T$ to minimize the cost functional

$$
J = t_f \tag{141}
$$

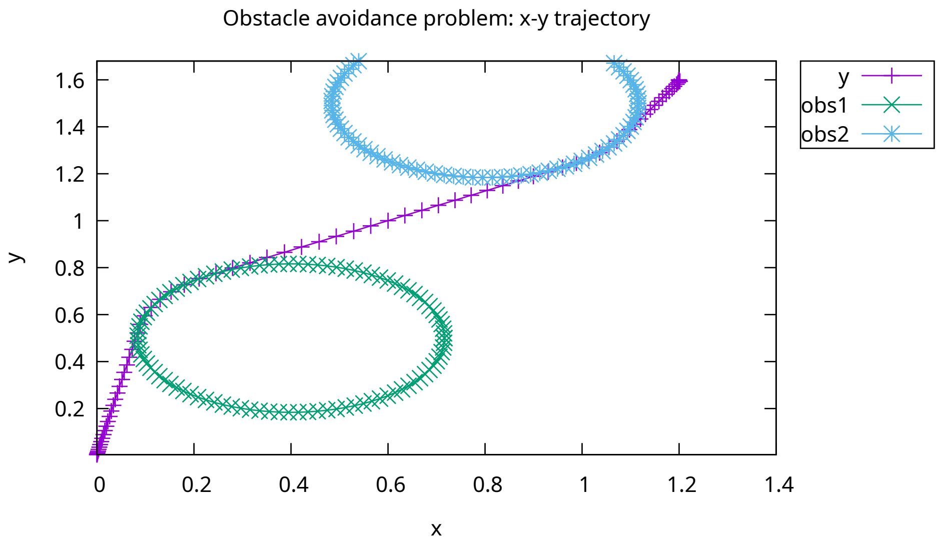

Figure 86: Optimal (x, y) trajectory for obstacle avoidance problem

subject to the dynamic constraints

$$\begin{aligned}
\dot{q}_1 &= \frac{1}{2}\left[\omega_1 q_4 - \omega_2 q_3 + \omega_3 q_2\right] \\
\dot{q}_2 &= \frac{1}{2}\left[\omega_1 q_3 + \omega_2 q_4 - \omega_3 q_1\right] \\
\dot{q}_3 &= \frac{1}{2}\left[-\omega_1 q_2 + \omega_2 q_1 + \omega_3 q_4\right] \\
\dot{\omega}_1 &= \frac{u_1}{I_x} - \left[\frac{I_z - I_y}{I_x}\omega_2\omega_3\right] \\
\dot{\omega}_2 &= \frac{u_2}{I_y} - \left[\frac{I_x - I_z}{I_y}\omega_1\omega_3\right] \\
\dot{\omega}_3 &= \frac{u_3}{I_z} - \left[\frac{I_y - I_x}{I_z}\omega_1\omega_2\right]
\end{aligned} \tag{142}$$

The path constraint:

$$0 = q_1^2 + q_2^2 + q_3^2 + q_4^2 - 1 \tag{143}$$

the boundary conditions:

$$\begin{aligned}
q_1(0) &= 0, \\
q_2(0) &= 0, \\
q_3(0) &= 0, \\
q_4(0) &= 1.0 \\
q_1(t_f) &= \sin\frac{\phi}{2}, \\
q_2(t_f) &= 0, \\
q_3(t_f) &= 0, \\
q_4(t_f) &= \cos\frac{\phi}{2} \\
\omega_1(0) &= 0, \\
\omega_2(0) &= 0, \\
\omega_3(0) &= 0, \\
\omega_1(t_f) &= 0, \\
\omega_2(t_f) &= 0, \\
\omega_3(t_f) &= 0,
\end{aligned} \tag{144}$$

where $\phi = 150\,\text{deg}$ is the Euler axis rotation angle, $\mathbf{q} = [q_1, q_2, q_3, q_4]^T$ is the quarternion vector, $\omega = [\omega_1, \omega_2, \omega_3]^T$ is the angular velocity vector, and $\mathbf{u} = [u_1, u_2, u_3]^T$ is the control vector. Note that in the implementation, variable $q_4(t)$ is treated as an algebraic variable (i.e. as a control variable).

The variable bounds and other parameters are given in the code.

The output from $\mathcal{PSOPT}$ is summarised in the box below and shown in Figures 87 to 88, which contain the elements of the quarternion vector $\mathbf{q}$, and the control vector $\mathbf{u} = [u_1, u_2, u_3]^T$, respectively.

```
PSOPT results summary
=====================

Problem:  Reorientation of a rigid body
CPU time (seconds): 1.358960e+01
NLP solver used:  IPOPT
PSOPT release number:  5.0.3
Date and time of this run:  Thu Mar  6 17:02:31 2025

Optimal (unscaled) cost function value:  2.863042e+01
Phase 1 endpoint cost function value: 2.863042e+01
Phase 1 integrated part of the cost:  0.000000e+00
Phase 1 initial time: 0.000000e+00
```

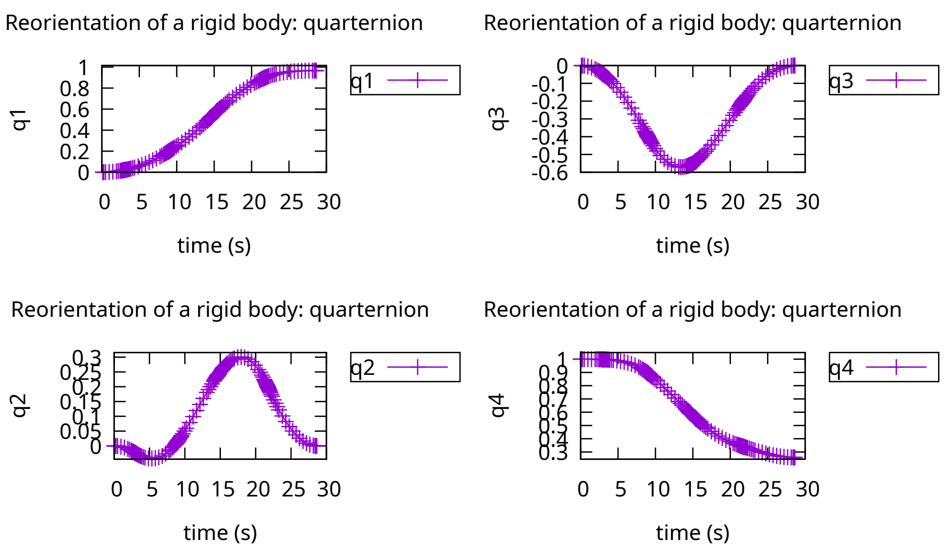

Figure 87: Quarternion vector elements for the reorientation problem

```
Phase 1 final time: 2.863042e+01
Phase 1 maximum relative local error: 9.888943e-06
NLP solver reports:  The problem has been solved!
```

36 Shuttle re-entry problem

Consider the following optimal control problem, which is known in the literature as the shuttle re-entry problem [3]. Find t_f, $\alpha(t)$ and $\beta(t) \in [0, t_f]$ to minimize the cost functional

$$J = -\frac{180}{\pi}\theta(t_f) \tag{145}$$

subject to the dynamic constraints

$$\begin{aligned}
\dot{h} &= v\sin(\gamma) \\
\dot{\phi} &= \tfrac{v}{r}\cos(\gamma)\sin(\psi)/\cos(\theta) \\
\dot{m} &= \tfrac{v}{r}\cos(\gamma)\cos(\psi) \\
\dot{v} &= -\tfrac{D}{m} - g\sin(\gamma) \\
\dot{\gamma} &= \tfrac{L}{mv}\cos(\beta) + \cos(\gamma)(\tfrac{v}{r} - \tfrac{g}{v}) \\
\dot{\psi} &= \tfrac{1}{mv\cos(\gamma)}L\sin(\beta) + \tfrac{v}{r\cos(\theta)}\cos(\gamma)\sin(\psi)\sin(\theta)
\end{aligned} \tag{146}$$

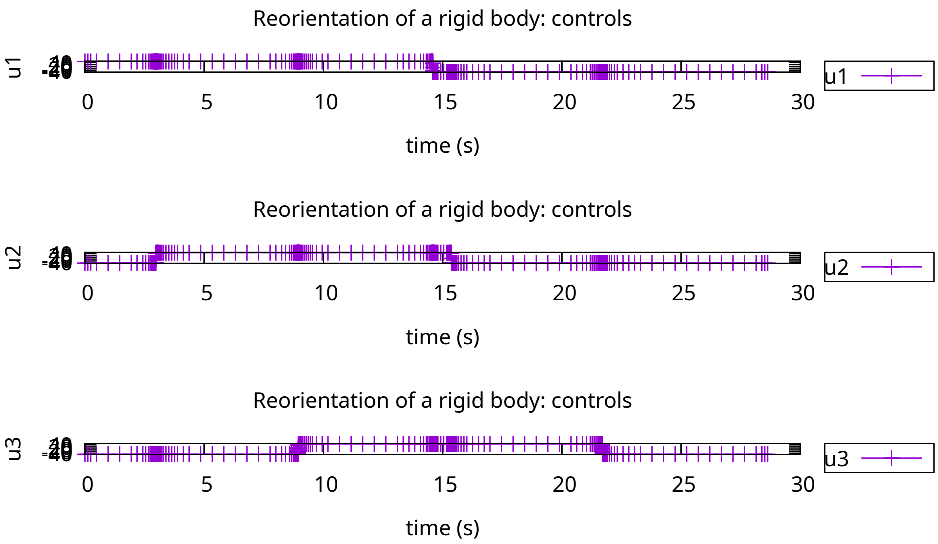

Figure 88: Control vector elements for the reorientation problem

the boundary conditions:

$$\begin{array}{rcl} h(0) & = & 260000.0 \\ \phi(0) & = & -0.6572 \\ \theta(0) & = & 0.0 \\ v(0) & = & 25600.0 \\ \gamma(0) & = & -0.0175 \\ h(t_f) & = & 80000.0 \\ v(t_f) & = & 2500.0 \\ \gamma(t_f) & = & -0.0873 \end{array} \tag{147}$$

The variable bounds and other parameters are given in the code.

The output from $\mathcal{PSOPT}$ is summarised in the box below and shown in Figures 89 to 96, which contain the elements of the state and the control vectors.

```
PSOPT results summary
=====================

Problem:  Shuttle re-entry problem
CPU time (seconds): 4.044636e+00
NLP solver used:  IPOPT
PSOPT release number:  5.0.3
Date and time of this run:  Thu Mar  6 17:02:50 2025

Optimal (unscaled) cost function value:  -3.414118e+01
```

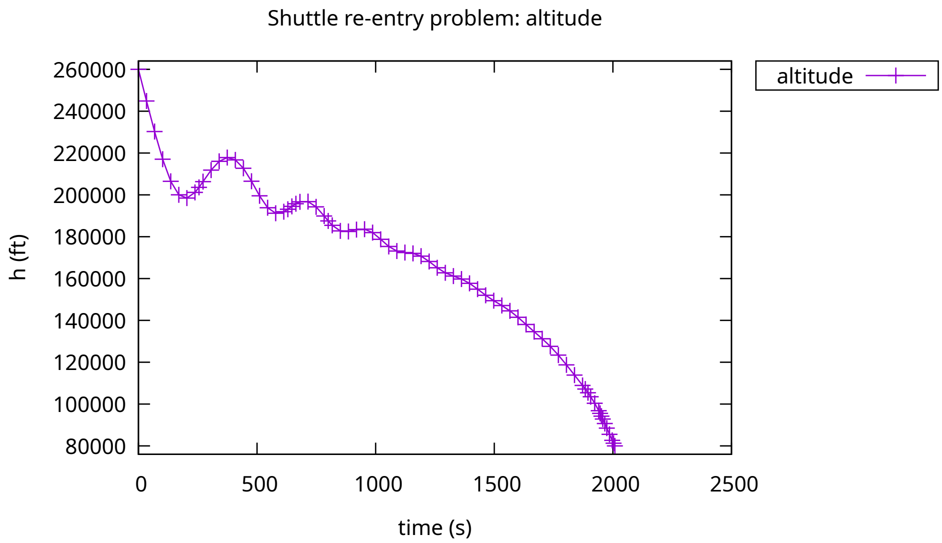

Figure 89: Altitude $h(t)$ for the shuttle re-entry problem

```
Phase 1 endpoint cost function value: -3.414118e+01
Phase 1 integrated part of the cost:  0.000000e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 2.008395e+03
Phase 1 maximum relative local error: 4.517636e-04
NLP solver reports:  The problem has been solved!
```

37 Singular control problem

Consider the following optimal control problem, whose solution is known to have a singular arc [14, 18]. Find $u(t), t \in [0,1]$ to minimize the cost functional

$$J = \int_0^1 [x_1^2 + x_2^2 + 0.0005(x_2 + 16x_5 - 8 - 0.1x_3u^2)^2]dt \tag{148}$$

subject to the dynamic constraints

$$\begin{aligned} \dot{x}_1 &= x_2 \\ \dot{x}_2 &= -x_3u + 16t - 8 \\ \dot{x}_3 &= u \end{aligned} \tag{149}$$

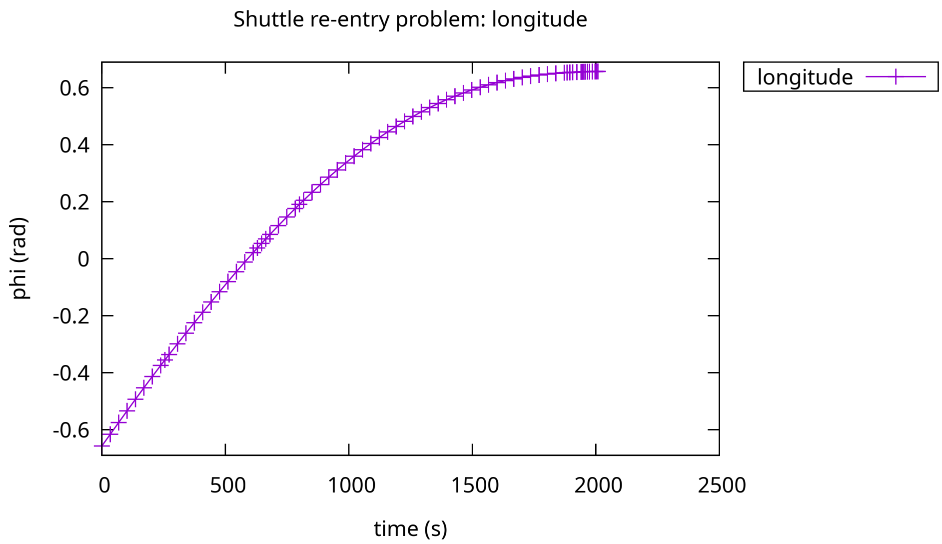

Figure 90: Longitude $\phi(t)$ for the shuttle re-entry problem

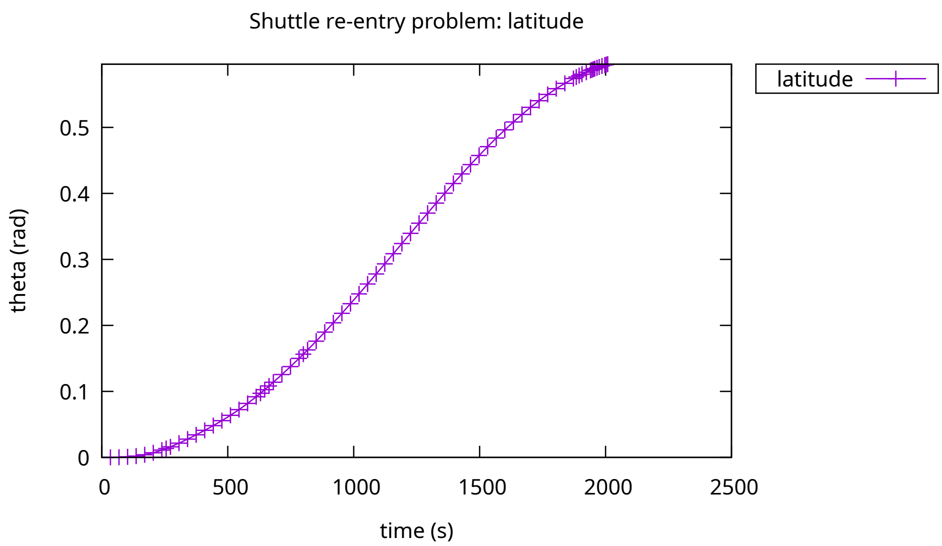

Figure 91: Latitude $\theta(t)$ for the shuttle re-entry problem

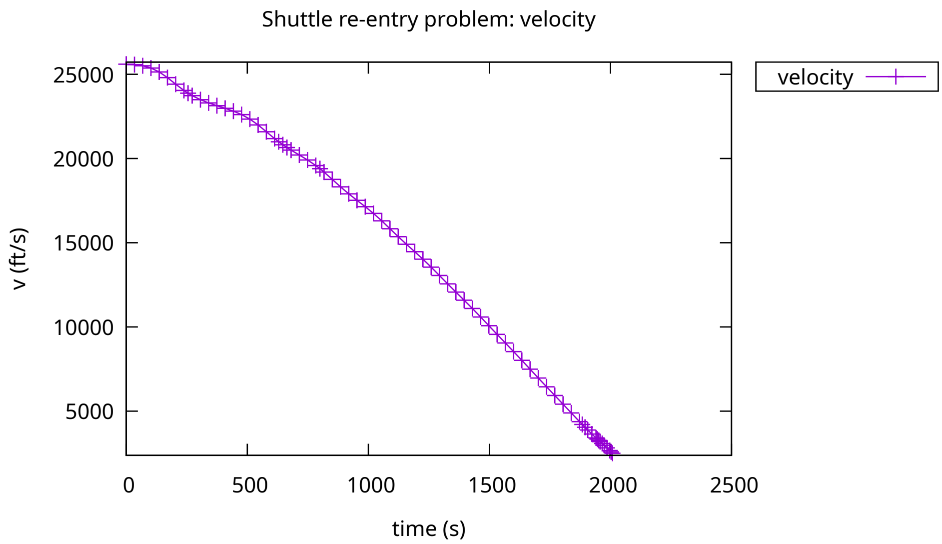

Figure 92: Velocity $v(t)$ for the shuttle re-entry problem

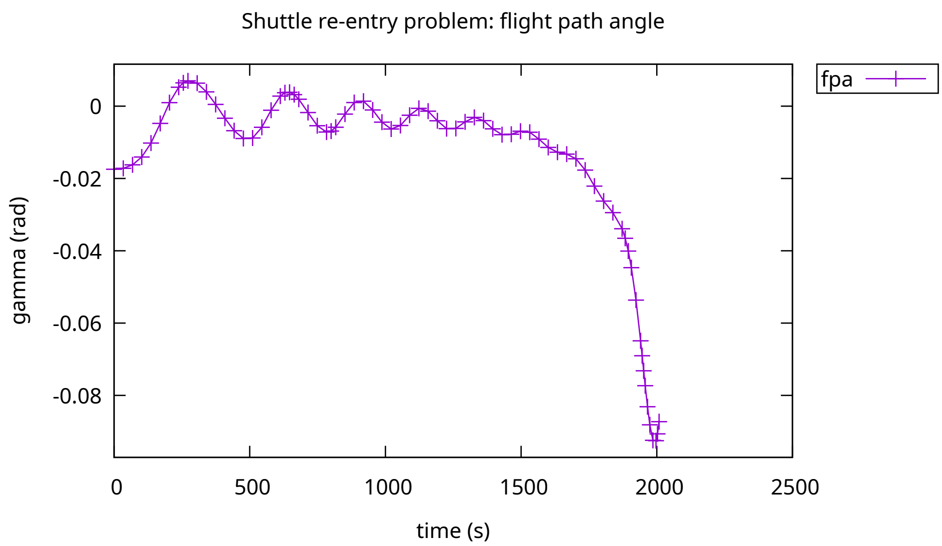

Figure 93: Flight path angle $\gamma(t)$ for the shuttle re-entry problem

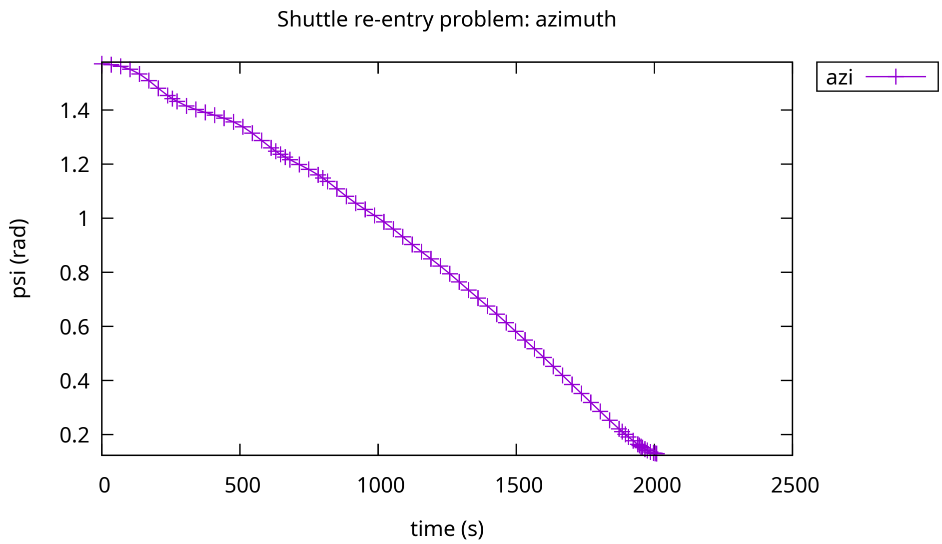

Figure 94: Azimuth $\psi(t)$ for the shuttle re-entry problem

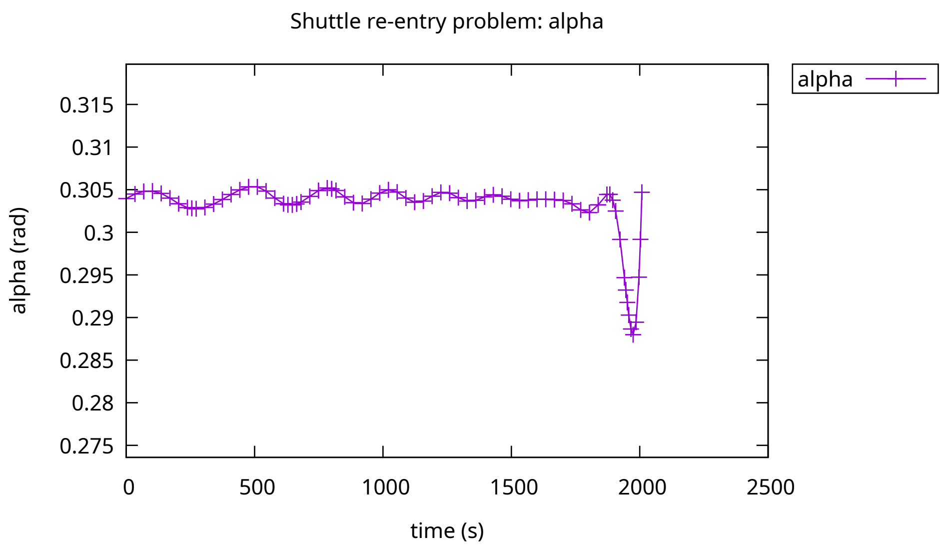

Figure 95: Angle of attack $\alpha(t)$ for the shuttle re-entry problem

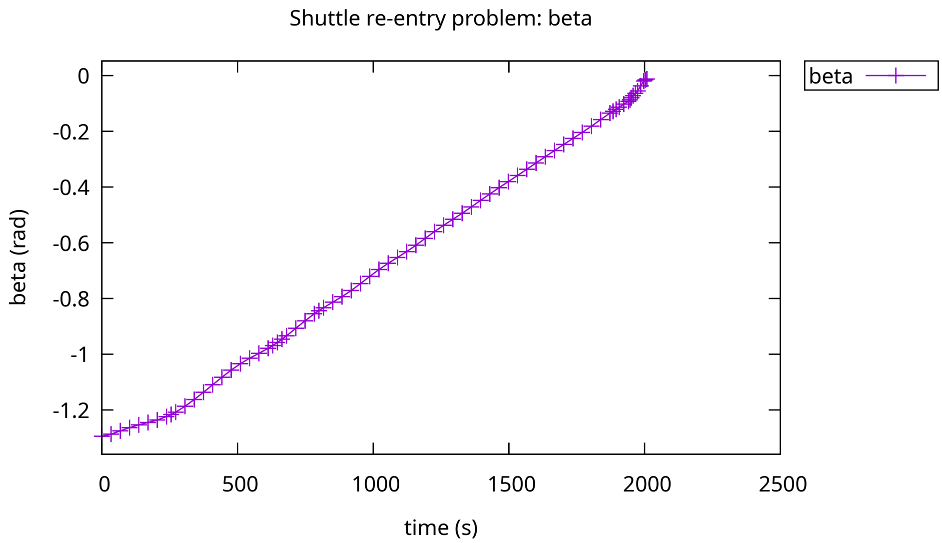

Figure 96: Bank angle $\beta(t)$ for the shuttle re-entry problem

the boundary conditions:

$$\begin{array}{rcl} x_1(0) & = & 0 \\ x_2(0) & = & -1 \\ x_3(0) & = & \sqrt{5} \end{array} \tag{150}$$

and the control bounds

$$-4 \leq u(t) \quad \leq 10 \tag{151}$$

The output from $\mathcal{PSOPT}$ is summarised in the box below and shown in Figures 97 and 98, which contain the elements of the state and the control, respectively.

```
PSOPT results summary
=====================

Problem:  Singular control 5
CPU time (seconds): 5.170623e+00
NLP solver used:  IPOPT
PSOPT release number:  5.0.3
Date and time of this run:  Thu Mar  6 17:03:08 2025

Optimal (unscaled) cost function value:  1.192620e-01
Phase 1 endpoint cost function value: 0.000000e+00
Phase 1 integrated part of the cost:  1.192620e-01
Phase 1 initial time: 0.000000e+00
```

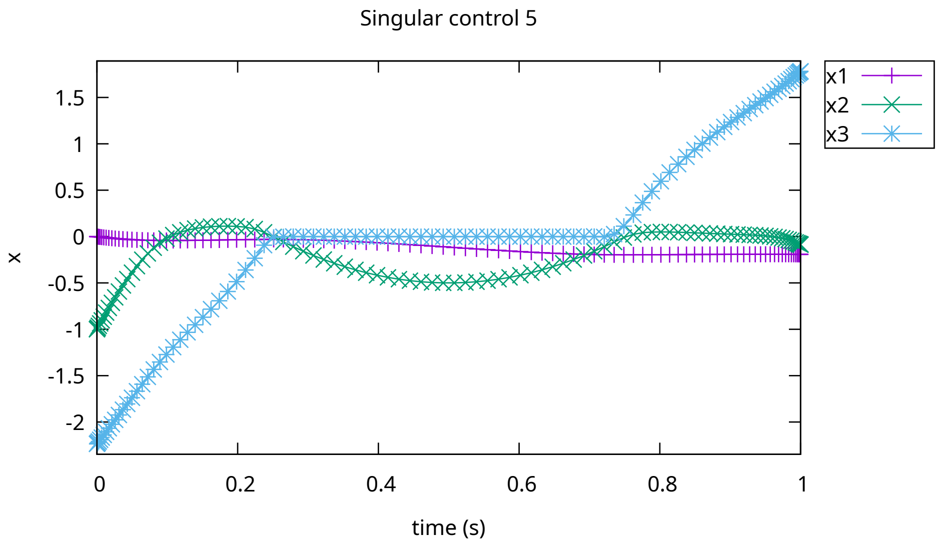

Figure 97: States for singular control problem

```
Phase 1 final time: 1.000000e+00
Phase 1 maximum relative local error: 2.551685e-04
NLP solver reports:  The problem has been solved!
```

38 Time varying state constraint problem

Consider the following optimal control problem, which involves a time varying state constraint [22]. Find $u(t) \in [0, 1]$ to minimize the cost functional

$$J = \int_0^1 [x_1^2(t) + x_2^2(t) + 0.005u^2(t)]dt \tag{152}$$

subject to the dynamic constraints

$$\begin{aligned} \dot{x}_1 &= x_2 \\ \dot{x}_2 &= -x_2 + u \end{aligned} \tag{153}$$

the boundary conditions:

$$\begin{aligned} x_1(0) &= 0 \\ x_2(0) &= -1 \end{aligned} \tag{154}$$

and the path constraint

$$x_2 \leq 8(t - 0.5)^2 - 0.5 \tag{155}$$

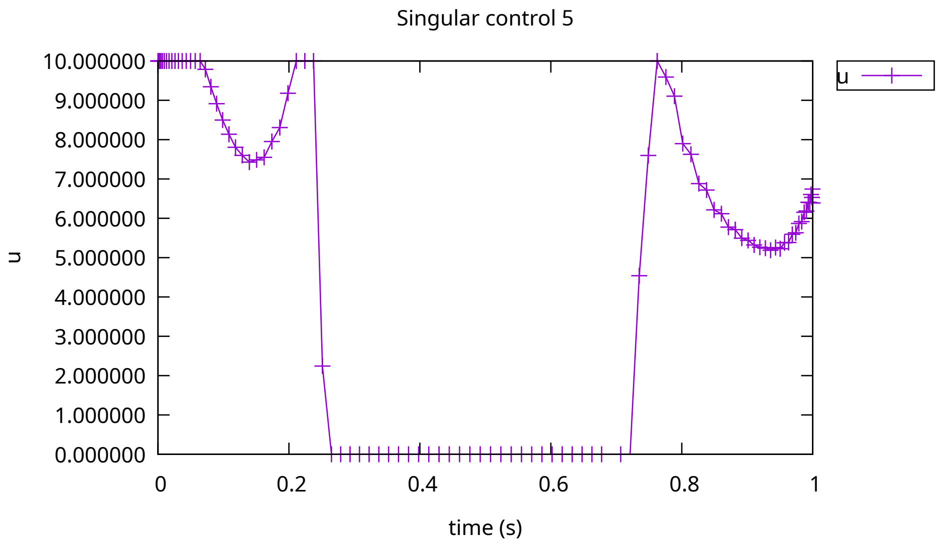

Figure 98: Control for singular control problem

The output from $\mathcal{PSOPT}$ is summarised in the box below and shown in Figures 99 and 100, which contain the elements of the states with the boundary of the constraint on x_2, and the control, respectively.

```
PSOPT results summary
=====================

Problem:  Time varying state constraint problem
CPU time (seconds): 4.926420e-01
NLP solver used:  IPOPT
PSOPT release number:  5.0.3
Date and time of this run:  Thu Mar  6 17:03:19 2025

Optimal (unscaled) cost function value:  1.698180e-01
Phase 1 endpoint cost function value: 0.000000e+00
Phase 1 integrated part of the cost:  1.698180e-01
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 1.000000e+00
Phase 1 maximum relative local error: 3.218028e-05
NLP solver reports:  The problem has been solved!
```

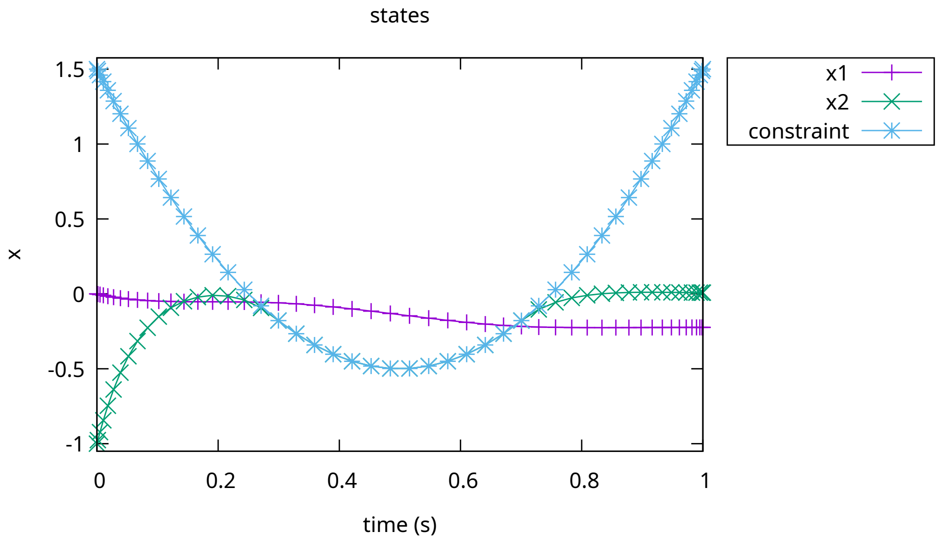

Figure 99: States for time-varying state constraint problem

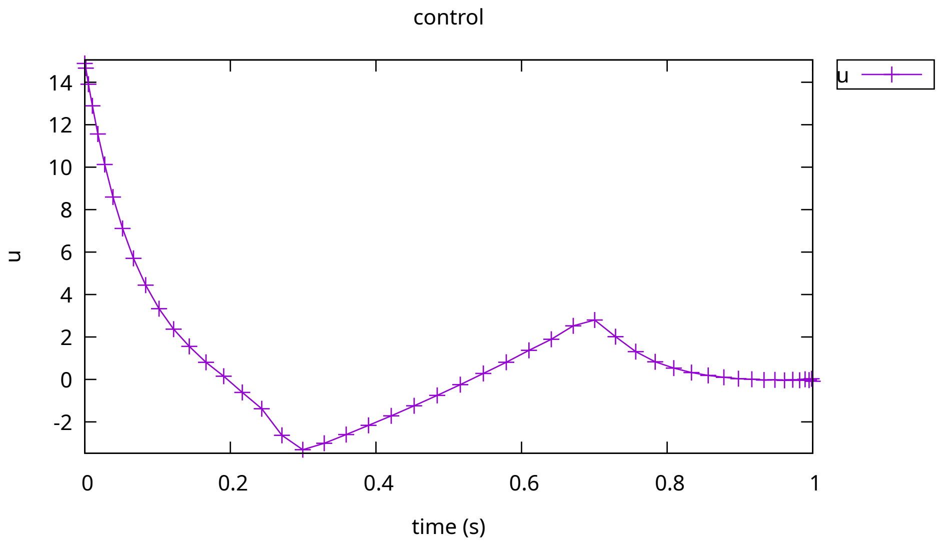

Figure 100: Control for time-varying state constraint problem

39 Two burn orbit transfer

The goal of this problem is to compute a trajectory for an spacecraft to go from a standard space shuttle park orbit to a geosynchronous final orbit. It is assumed that the engines operate over two short periods during the mission, and it is desired to compute the timing and duration of the burn periods, as well as the instantaneous direction of the thrust during these two periods, to maximise the final weight of the spacecraft. The problem is described in detail by Betts [3]. The mission then involves four phases: coast, burn, coast and burn. The problem is formulated as follows. Find $\mathbf{u}(t) = [\theta(t), \phi(t)]^T, t \in [t_f^{(1)}, t_f^{(2)}]$ and $t \in [t_f^{(3)}, t_f^{(4)}]$, and the instants $t_f^{(1)}, t_f^{(2)}, t_f^{(3)}, t_f^{(4)}$ such that the following objective function is minimised:

$$J = -w(t_f) \tag{156}$$

subject to the dynamic constraints for phases 1 and 3:

$$\dot{\mathbf{y}} = \mathbf{A}(\mathbf{y})\Delta_g + \mathbf{b} \tag{157}$$

the following dynamic constraints for phases 2 and 4:

$$\begin{aligned} \dot{\mathbf{y}} &= \mathbf{A}(\mathbf{y})\Delta + \mathbf{b} \\ \dot{w} &= -T/I_{sp} \end{aligned} \tag{158}$$

and the following linkages between phases

$$\begin{aligned} \mathbf{y}(t_f^{(1)}) &= \mathbf{y}(t_0^{(2)}) \\ \mathbf{y}(t_f^{(2)}) &= \mathbf{y}(t_0^{(3)}) \\ \mathbf{y}(t_f^{(3)}) &= \mathbf{y}(t_0^{(4)}) \\ t_f^{(1)} &= t_0^{(2)} \\ t_f^{(2)} &= t_0^{(3)} \\ t_f^{(3)} &= t_0^{(4)} \\ w(t_f^{(2)}) &= w(t_0^{(4)}) \end{aligned} \tag{159}$$

where $\mathbf{y} = [p, f, g, h, k, L, w]^T$ is the vector of modified equinoctial elements, w is the spacecraft weight, I_{sp} is the specific impulse of the engine, T is the maximum thrust, expressions for $\mathbf{A}(\mathbf{y})$ and $\mathbf{b}$ are given in [3]. the disturbing acceleration is $\Delta = \Delta_g + \Delta_T$, where Δ_g is the gravitational disturbing acceleration due to the oblatness of Earth (given in [3]), and Δ_T is the thurst acceleration, given by:

$$\Delta_T = \mathbf{Q}_r \mathbf{Q}_v \begin{bmatrix} T_a \cos\theta \cos\phi \\ T_a \cos\theta \sin\phi \\ T_a \sin\theta \end{bmatrix} \tag{160}$$

where $T_a(t) = g_0 T/w(t)$, g_0 is a constant, θ is the pitch angle and ϕ is the yaw angle of the thurst, matrix $\mathbf{Q}_v$ is given by:

$$\mathbf{Q}_v = \left[\frac{\mathbf{v}}{||\mathbf{v}||}, \frac{\mathbf{v} \times r}{||\mathbf{v} \times \mathbf{r}||}, \frac{\mathbf{v}}{||\mathbf{v}||} \times \frac{\mathbf{v} \times r}{||\mathbf{v} \times \mathbf{r}||} \right] \tag{161}$$

matrix $\mathbf{Q}_r$ is given by:

$$\mathbf{Q}_r = \begin{bmatrix} \mathbf{i}_r & \mathbf{i}_\theta & \mathbf{i}_h \end{bmatrix} = \begin{bmatrix} \frac{\mathbf{r}}{||\mathbf{r}||} & \frac{(\mathbf{r} \times \mathbf{v}) \times \mathbf{r}}{||\mathbf{r} \times \mathbf{v}|| ||\mathbf{r}||} & \frac{(\mathbf{r} \times \mathbf{v})}{||\mathbf{r} \times \mathbf{v}||} \end{bmatrix} \tag{162}$$

The boundary conditions of the problem are given by:

$$\begin{aligned} p(0) &= 218327080.052835 \\ f(0) &= 0 \\ g(0) &= 0 \\ h(0) &= 0 \\ h(0) &= 0 \\ k(0) &= 0 \\ L(0) &= \pi \text{ (rad)} \\ w(0) &= 1 \text{ (lb)} \\ p(t_f) &= 19323/\sigma + R_e \\ f(t_f) &= 0 \\ g(t_f) &= 0 \\ h(t_f) &= 0 \\ k(t_f) &= 0 \end{aligned} \tag{163}$$

and the values of the parameters are: $g_0 = 32.174$ (ft/sec^2), $I_{sp} = 300$ (sec), $T = 1.2$ (lb), $\mu = 1.407645794 \times 10^{16}$ (ft^3/sec^2), $R_e = 20925662.73$ (ft), $\sigma = 1.0/6076.1154855643$, $J_2 = 1082.639 \times 10^{-6}$, $J_3 = -2.565 \times 10^{-6}$, $J_4 = -1.608 \times 10^{-6}$.

An initial guess was computed by forward propagation from the initial conditions, assuming the following guesses for the controls and burn periods [3]:

$$\begin{aligned} \mathbf{u}(t) &= \begin{bmatrix} 0.148637 \times 10^{-2}, & -9.08446 \end{bmatrix}^T \quad & t \in [2840, 21650] \\ \mathbf{u}(t) &= \begin{bmatrix} -0.136658 \times 10^{-2}, & 49.7892 \end{bmatrix} \quad & t \in [21650, 21700] \end{aligned} \tag{164}$$

The problem was solved using local collocation (trapezoidal followed by Hermite-Simpson) with automatic mesh refinement.

The output from $\mathcal{PSOPT}$ is summarised in the box below. The controls during the burn periods are shown Figures 101 to 104, which show the control variables during phases 2 and 4, and Figure 105, which shows the trajectory in cartesian co-ordinates.

```
PSOPT results summary
=====================

Problem:  Two burn transfer problem
CPU time (seconds): 1.005158e+01
NLP solver used:  IPOPT
PSOPT release number:  5.0.3
Date and time of this run:  Thu Mar  6 17:04:19 2025

Optimal (unscaled) cost function value:  -2.367249e-01
Phase 1 endpoint cost function value: 0.000000e+00
Phase 1 integrated part of the cost:  0.000000e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 2.609964e+03
Phase 1 maximum relative local error: 7.747257e-07
Phase 2 endpoint cost function value: 0.000000e+00
Phase 2 integrated part of the cost:  0.000000e+00
Phase 2 initial time: 2.609964e+03
Phase 2 final time: 2.751426e+03
Phase 2 maximum relative local error: 3.260256e-03
Phase 3 endpoint cost function value: 0.000000e+00
Phase 3 integrated part of the cost:  0.000000e+00
Phase 3 initial time: 2.751426e+03
Phase 3 final time: 2.163411e+04
Phase 3 maximum relative local error: 5.773065e-06
Phase 4 endpoint cost function value: -2.367249e-01
Phase 4 integrated part of the cost:  0.000000e+00
Phase 4 initial time: 2.163411e+04
Phase 4 final time: 2.168347e+04
Phase 4 maximum relative local error: 2.500652e-05
NLP solver reports:  The problem has been solved!
```

40 Two link robotic arm

Consider the following optimal control problem [14]. Find t_f, and $u(t) \in [0, t_f]$ to minimize the cost functional

$$J = t_f \tag{165}$$

Table 9: Mesh refinement statistics: Two burn transfer problem

Iter	DM	M	NV	NC	OE	CE	JE	HE	RHS	$\epsilon_{\max}$	$\mathrm{CPU_a}$
1	TRP	40	308	298	1399	1400	426	0	106400	5.447e-02	2.676e+00
2	TRP	56	428	402	28	29	27	0	3132	2.280e-02	1.806e-01
3	H-S	76	650	532	54	55	54	0	12100	1.503e-02	5.815e-01
4	H-S	104	888	714	58	59	56	0	17936	7.752e-03	9.029e-01
5	H-S	144	1228	974	140	141	119	0	59784	3.268e-02	2.518e+00
6	H-S	191	1650	1284	69	70	68	0	39550	3.260e-03	1.793e+00
$\mathrm{CPU_b}$	-	-	-	-	-	-	-	-	-	-	1.399e+00
-	-	-	-	-	1748	1754	750	0	238902	-	1.005e+01

Key: Iter=iteration number, DM= discretization method, M=number of nodes, NV=number of variables, NC=number of constraints, OE=objective evaluations, CE = constraint evaluations, JE = Jacobian evaluations, HE = Hessian evaluations, RHS = ODE right hand side evaluations, $\epsilon_{\max}$ = maximum relative ODE error, $\mathrm{CPU_a}$ = CPU time in seconds spent by NLP algorithm, $\mathrm{CPU_b}$ = additional CPU time in seconds spent by PSOPT

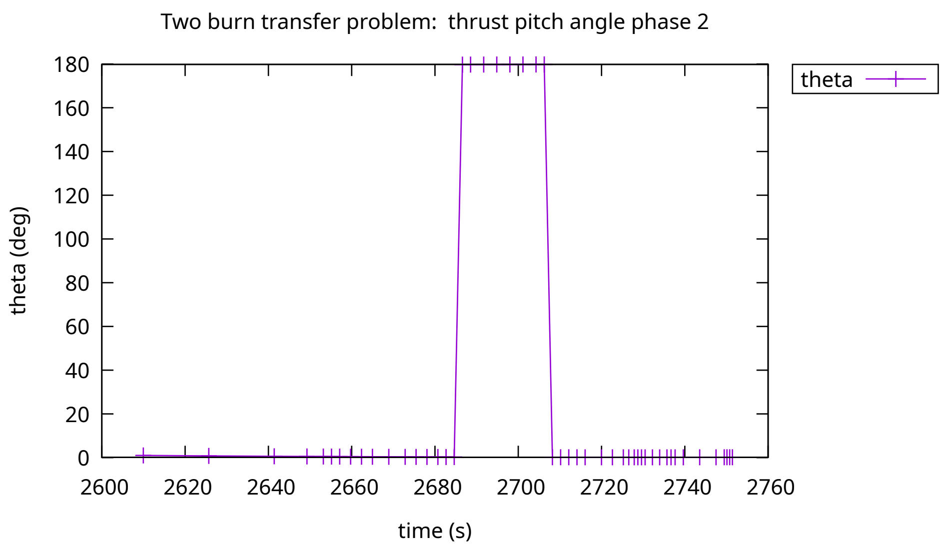

Figure 101: Pitch angle during phase 2

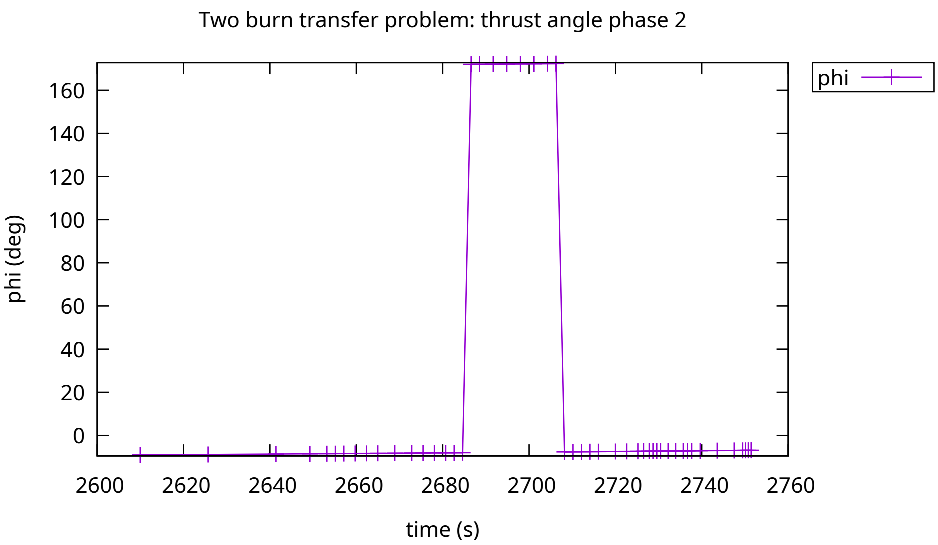

Figure 102: Yaw angle during phase 2

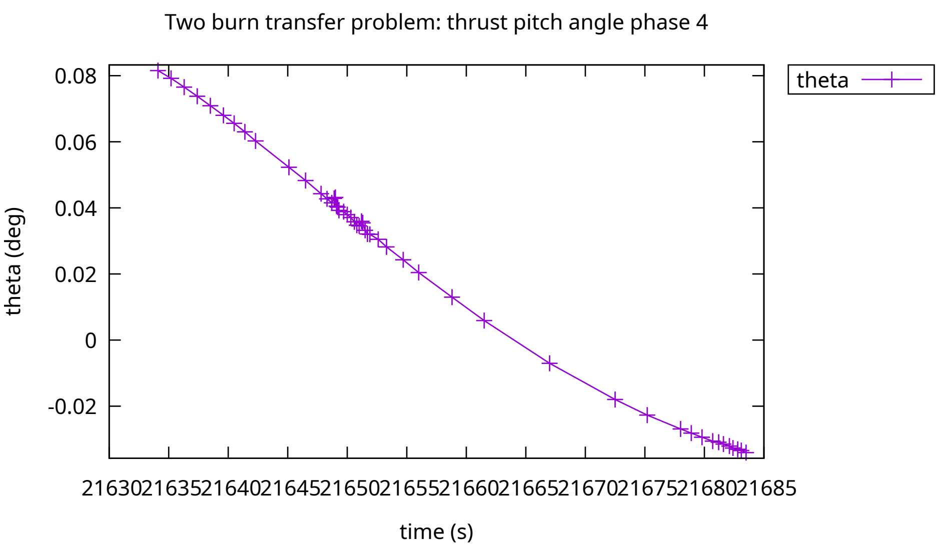

Figure 103: Pitch angle during phase 4

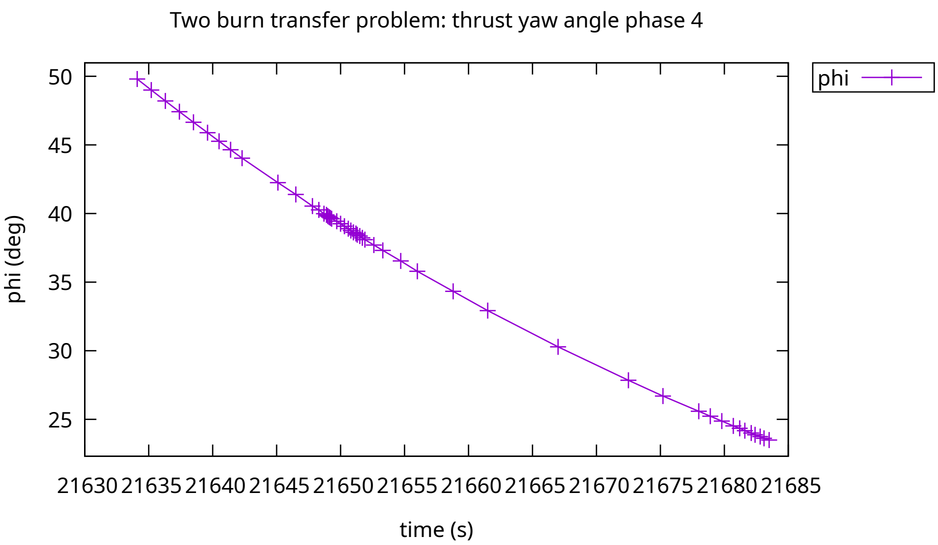

Figure 104: Yaw angle during phase 4

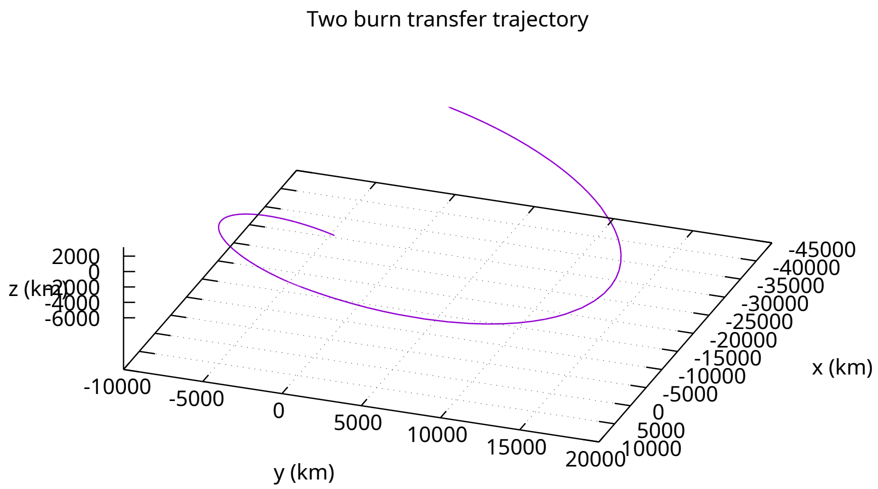

Figure 105: Two burn transfer trajectory

subject to the dynamic constraints

$$
\begin{aligned}
\dot{x}_1 &= \frac{\sin(x_3)(\frac{9.0}{4.0}\cos(x_3)x_1^2+2*x_2^2)+\frac{4.0}{3.0}(u_1-u_2)-\frac{3.0}{2.0}\cos(x_3)u_2}{\frac{31.0}{36.0}+\frac{9.0}{4.0\sin^2(x_3)}} \\
\dot{x}_2 &= \frac{-(\sin(x_3)*(\frac{7.0}{2.0}*x_1^2+\frac{9.0}{4.0}\cos(x_3)x_2^2)-\frac{7.0}{3.0}u_2+\frac{3.0}{2.0}\cos(x_3)(u_1-u_2))}{\frac{31.0}{36.0}+\frac{9.0}{4.0\sin^2(x_3)}} \\
\dot{x}_3 &= x_2 - x_1 \\
\dot{x}_4 &= x_1
\end{aligned} \tag{166}
$$

the boundary conditions:

$$
\begin{array}{rclrcl}
x_1(0) &=& 0 & x_1(t_f) &=& 0 \\
x_2(0) &=& 0 & x_2(t_f) &=& 0 \\
x_3(0) &=& 0.5 & x_3(t_f) &=& 0.5 \\
x_4(0) &=& 0.0 & x_4(t_f) &=& 0.522
\end{array} \tag{167}
$$

The control bounds:

$$
\begin{aligned}
-1 \leq & u_1(t) \leq 1 \\
-1 \leq & u_2(t) \leq 1
\end{aligned} \tag{168}
$$

The output from $\mathcal{PSOPT}$ is summarised in the box below and shown in Figures 106 and 107, which contain the elements of the state and the control, respectively.

```
PSOPT results summary
=====================

Problem:  Two link robotic arm
CPU time (seconds): 4.981430e-01
NLP solver used:  IPOPT
PSOPT release number:  5.0.3
Date and time of this run:  Thu Mar  6 17:04:38 2025

Optimal (unscaled) cost function value:  2.988662e+00
Phase 1 endpoint cost function value: 2.988662e+00
Phase 1 integrated part of the cost:  0.000000e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 2.988662e+00
Phase 1 maximum relative local error: 3.815629e-04
NLP solver reports:  The problem has been solved!
```

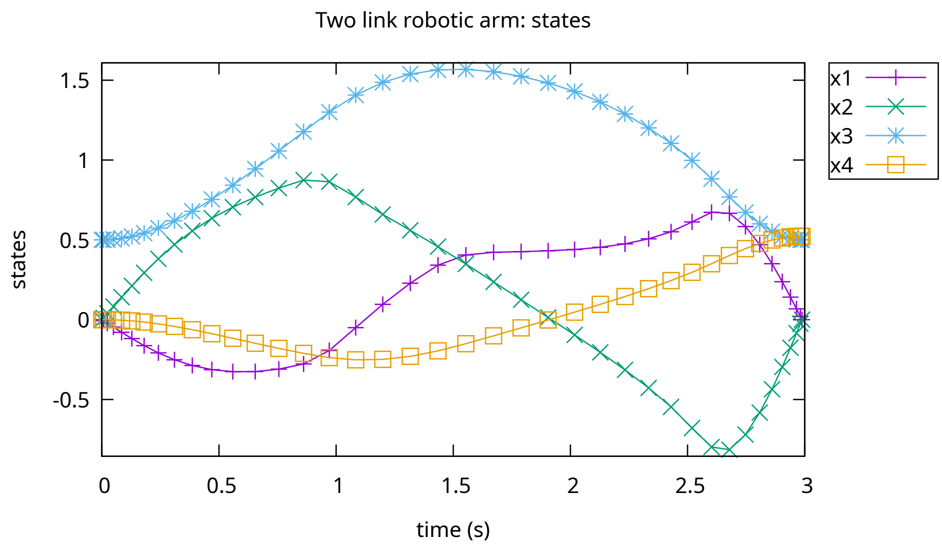

Figure 106: States for two-link robotic arm problem

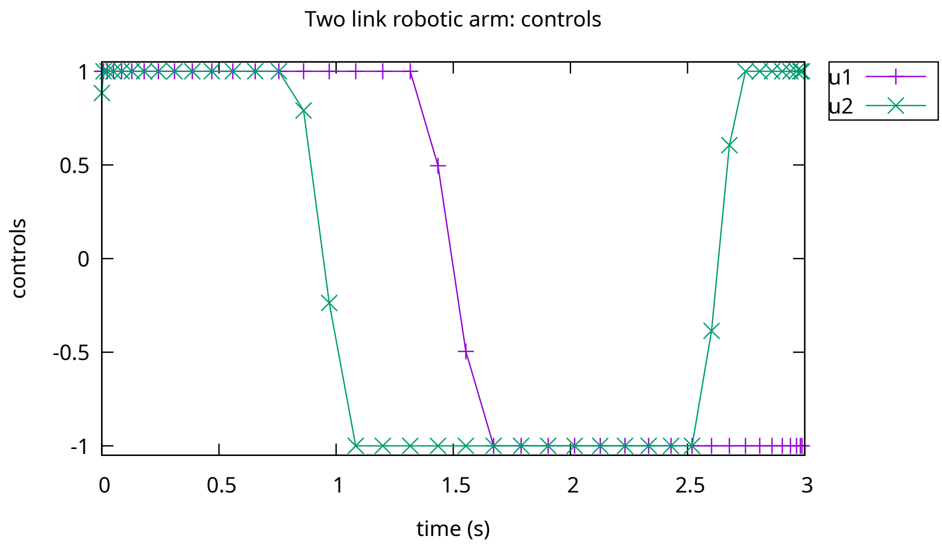

Figure 107: Controls for two link robotic arm problem

41 Two-phase path tracking robot

Consider the following two-phase optimal control problem, which consists of a robot following a specified path [20, 18]. Find $u(t) \in [0, 2]$ to minimize the cost functional

$$\begin{aligned} J = \int_0^2 & [100(x_1 - x_{1,ref})^2 + 100(x_2 - x_{2,ref})^2 \\ & +500(x_3 - x_{3,ref})^2 + 500(x_4 - x_{4,ref})^2]dt \end{aligned} \tag{169}$$

subject to the dynamic constraints

$$\begin{aligned} \dot{x}_1 &= x_3 \\ \dot{x}_2 &= x_4 \\ \dot{x}_3 &= u_1 \\ \dot{x}_4 &= u_2 \end{aligned} \tag{170}$$

the boundary conditions:

$$\begin{array}{lclclcl} x_1(0) &=& 0 & x_1(2) &=& 0.5 \\ x_2(0) &=& 0 & x_2(2) &=& 0.5 \\ x_3(0) &=& 0.5 & x_3(2) &=& 0 \\ x_4(0) &=& 0.0 & x_4(2) &=& 0.5 \end{array} \tag{171}$$

where the reference signals are given by:

$$\begin{aligned} x_{1,ref} &= \frac{t}{2}\,(0 \le t < 1), \frac{1}{2}\,(1 \le t \le 2) \\ x_{2,ref} &= 0\,(0 \le t < 1), \frac{t-1}{2}\,(1 \le t \le 2) \\ x_{3,ref} &= \frac{1}{2}\,(0 \le t < 1), 0\,(1 \le t \le 2) \\ x_{4,ref} &= 0\,(0 \le t < 1), \frac{1}{2}\,(1 \le t \le 2) \end{aligned} \tag{172}$$

Note that the first phase covers the period $t \in [0, 1]$, while the second phase covers the period $t \in [1, 2]$.

The output from $\mathcal{PSOPT}$ is summarised in the box below and shown in Figures 108 and 109, which contain the elements of the state and the control, respectively.

```
PSOPT results summary
=====================

Problem:  Two phase path tracking robot
CPU time (seconds): 3.846540e-01
NLP solver used:  IPOPT
```

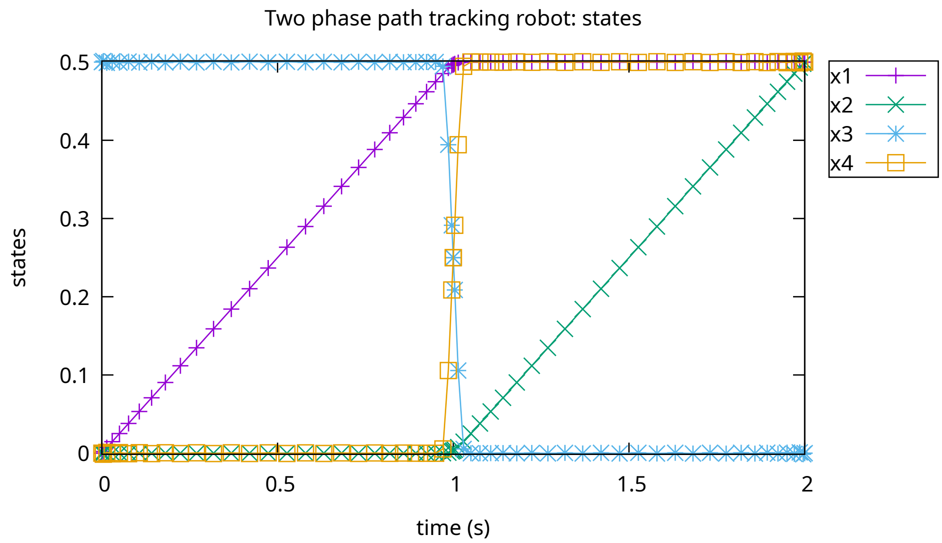

Figure 108: States for two-phase path tracking robot problem

```
PSOPT release number:  5.0.3
Date and time of this run:  Thu Mar  6 17:04:52 2025

Optimal (unscaled) cost function value:  1.042568e+00
Phase 1 endpoint cost function value: 0.000000e+00
Phase 1 integrated part of the cost:  5.212840e-01
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 1.000000e+00
Phase 1 maximum relative local error: 3.443286e-04
Phase 2 endpoint cost function value: 0.000000e+00
Phase 2 integrated part of the cost:  5.212840e-01
Phase 2 initial time: 1.000000e+00
Phase 2 final time: 2.000000e+00
Phase 2 maximum relative local error: 3.443298e-04
NLP solver reports:  The problem has been solved!
```

42 Two-phase Schwartz problem

Consider the following two-phase optimal control problem [18]. Find $u(t) \in [0, 2.9]$ to minimize the cost functional

$$J = 5(x_1(t_f)^2 + x_2(t_f)^2) \tag{173}$$

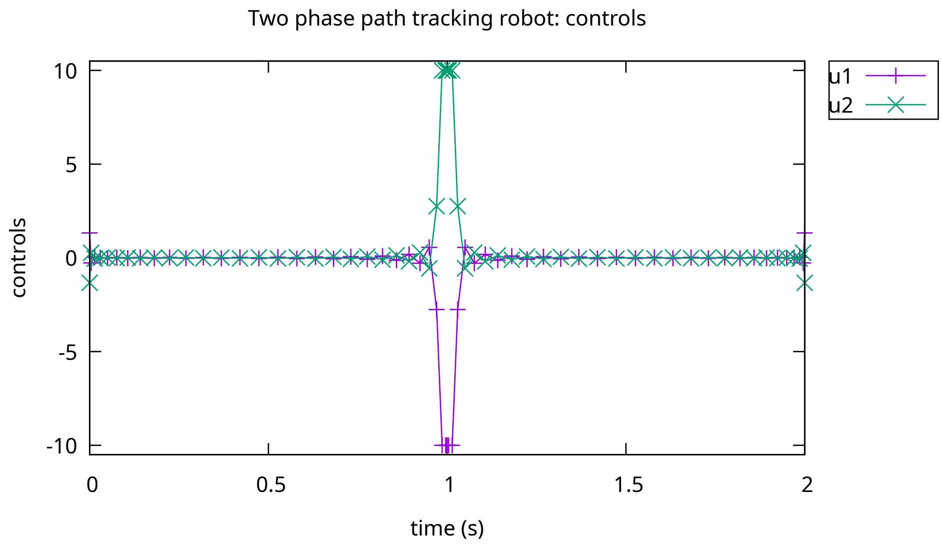

Figure 109: Control for two phase path tracking robot problem

subject to the dynamic constraints

$$\begin{aligned} \dot{x}_1 &= x_2 \\ \dot{x}_2 &= u - 0.1(1 + 2x_1^2)x_2 \end{aligned} \tag{174}$$

the boundary conditions:

$$\begin{aligned} x_1(0) &= 1 \\ x_2(0) &= 1 \end{aligned} \tag{175}$$

and the constraints for $t < 1$:

$$\begin{aligned} 1-9(x_1-1)^2 - \left(\frac{x_2 - 0.4}{0.3}\right)^2 &\leq 0 \\ -0.8 \leq x_2 & \\ -1 \leq u \leq 1 & \end{aligned} \tag{176}$$

The problem has been divided into two phases. The first phase covers the period $t \in [0, 1]$, while the second phase covers the period $t \in [1, 2.9]$.

The output from $\mathcal{PSOPT}$ is summarised in the box below and shown in Figures 110 and 111, which contain the elements of the state and the control, respectively.

```
PSOPT results summary
=====================
```

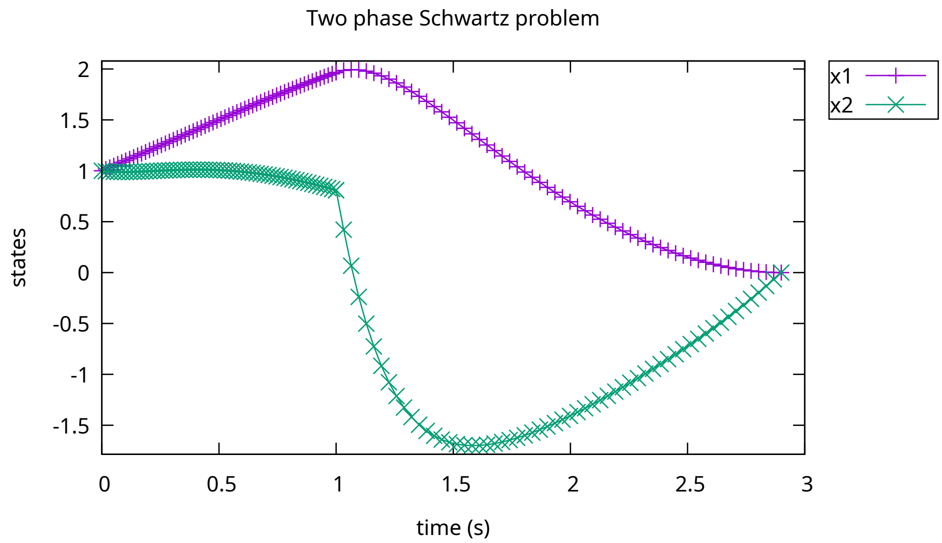

Figure 110: States for two-phase Schwartz problem

```
Problem:  Two phase Schwartz problem
CPU time (seconds): 2.101310e-01
NLP solver used:  IPOPT
PSOPT release number:  5.0.3
Date and time of this run:  Thu Mar  6 17:05:05 2025

Optimal (unscaled) cost function value:  3.138721e-16
Phase 1 endpoint cost function value: 0.000000e+00
Phase 1 integrated part of the cost:  0.000000e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 1.000000e+00
Phase 1 maximum relative local error: 3.948801e-03
Phase 2 endpoint cost function value: 3.138721e-16
Phase 2 integrated part of the cost:  0.000000e+00
Phase 2 initial time: 1.000000e+00
Phase 2 final time: 2.900000e+00
Phase 2 maximum relative local error: 2.254848e-02
NLP solver reports:  The problem has been solved!
```

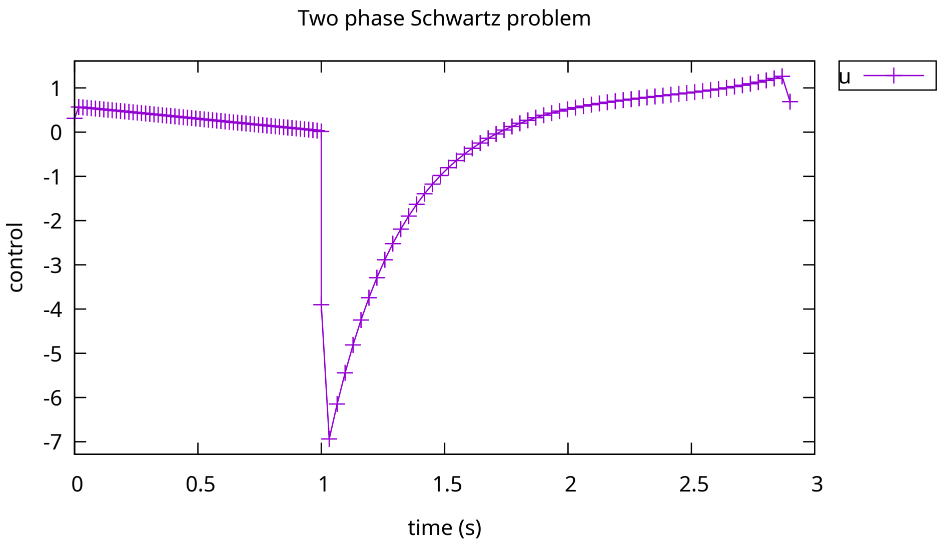

Figure 111: Control for two-phase Schwartz problem

43 Vehicle launch problem

This problem consists of the launch of a space vehicle. See [16, 2] for a full description of the problem. Only a brief description is given here. The flight of the vehicle can be divided into four phases, with dry masses ejected from the vehicle at the end of phases 1, 2 and 3. The final times of phases 1, 2 and 3 are fixed, while the final time of phase 4 is free. The optimal control problem is to find the control, $\mathbf{u}$, that minimizes the cost function

$$J = -m^{(4)}(t_f) \tag{177}$$

In other words, it is desired to maximise the vehicle mass at the end of phase 4. The dynamics are given by:

$$\begin{aligned} \dot{\mathbf{r}} &= \mathbf{v} \\ \dot{\mathbf{v}} &= -\frac{\mu}{\|\mathbf{r}\|^3}\mathbf{r} + \frac{T}{m}\mathbf{u} + \frac{\mathbf{D}}{m} \\ \dot{m} &= -\frac{T}{g_0 I_{sp}} \end{aligned} \tag{178}$$

where $\mathbf{r}(t) = \begin{bmatrix} x(t) & y(t) & z(t) \end{bmatrix}^T$ is the position, $\mathbf{v} = \begin{bmatrix} v_x(t) & v_y(t) & v_z(t) \end{bmatrix}^T$ is the Cartesian ECI velocity, μ is the gravitational parameter, T is the vacuum thrust, m is the mass, g_0 is the acceleration due to gravity at sea level, I_{sp} is the specific impulse of the engine, $\mathbf{u} = \begin{bmatrix} u_x & u_y & u_z \end{bmatrix}^T$ is the thrust direction, and $\mathbf{D} = \begin{bmatrix} D_x & D_y & D_z \end{bmatrix}^T$ is the drag force, which is given by:

$$\mathbf{D} = -\frac{1}{2} C_D A_{ref} \rho \|\mathbf{v}_{rel}\| \mathbf{v}_{rel} \tag{179}$$

where C_D is the drag coefficient, A_{ref} is the reference area, ρ is the atmospheric density, and $\mathbf{v}_{rel}$ is the Earth relative velocity, where $\mathbf{v}_{rel}$ is given as

$$\mathbf{v}_{rel} = \mathbf{v} - \boldsymbol{\omega} \times \mathbf{r} \tag{180}$$

where $\boldsymbol{\omega}$ is the angular velocity of the Earth relative to inertial space. The atmospheric density is modeled as follows

$$\rho = \rho_0 \exp[-h/h_0] \tag{181}$$

where ρ_0 is the atmospheric density at sea level, $h = \|\mathbf{r}\| - R_e$ is the altitude, R_e is the equatorial radius of the Earth, and h_0 is the density scale height. The numerical values for these constants can be found in the code.

The vehicle starts on the ground at rest (relative to the Earth) at time t_0, so that the initial conditions are

$$\begin{aligned} \mathbf{r}(t_0) &= \mathbf{r}_0 = \begin{bmatrix} 5605.2 & 0 & 3043.4 \end{bmatrix}^T \quad \text{km} \\ \mathbf{v}(t_0) &= \mathbf{v}_0 = \begin{bmatrix} 0 & 0.4076 & 0 \end{bmatrix}^T \quad \text{km/s} \\ m(t_0) &= m_0 = 301454 \quad \text{kg} \end{aligned} \tag{182}$$

The terminal constraints define the target transfer orbit, which is defined in orbital elements as

$$\begin{aligned} a_f &= 24361.14\,\text{km}, \\ e_f &= 0.7308, \\ i_f &= 28.5\,\text{deg}, \\ \Omega_f &= 269.8\,\text{deg}, \\ \omega_f &= 130.5\,\text{deg} \end{aligned} \tag{183}$$

There is also a path constraint associated with this problem:

$$||\mathbf{u}||^2 = 1 \tag{184}$$

The following linkage constraints force the position and velocity to be continuous and also account for discontinuity in the mass state due to the ejections at the end of phases 1, 2 and 3:

$$\begin{aligned} \mathbf{r}^{(p)}(t_f) - \mathbf{r}^{(p+1)}(t_0) &= \mathbf{0}, \\ \mathbf{v}^{(p)}(t_f) - \mathbf{v}^{(p+1)}(t_0) &= \mathbf{0}, \qquad (p = 1, \ldots, 3) \\ m^{(p)}(t_f) - m_{dry}^{(p)} - m^{(p+1)}(t_0) &= 0 \end{aligned} \tag{185}$$

where the superscript (p) represents the phase number.

The C++ code that solves this problem is shown below.

```
//////////////////////////////////////////////////////////////////////////
//////////////////           launch.cxx          ///////////////////////
//////////////////////////////////////////////////////////////////////////
////////////////          PSOPT  Example           ////////////////////
//////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////
```

```
//////// Title:         Multiphase vehicle launch         ////////////////
//////// Last modified: 05 January 2009                   ////////////////
//////// Reference:     GPOPS Manual                      ////////////////
//////// (See PSOPT handbook for full reference)           ////////////////
//////////////////////////////////////////////////////////////////////////
////////     Copyright (c) Victor M. Becerra, 2009        ////////////////
//////////////////////////////////////////////////////////////////////////
//////// This is part of the PSOPT software library, which ////////////////
//////// is distributed under the terms of the GNU Lesser ////////////////
//////// General Public License (LGPL)                    ////////////////
//////////////////////////////////////////////////////////////////////////

#include "psopt.h"

using namespace PSOPT;


//////////////////////////////////////////////////////////////////////////
///////////////////  Declare auxiliary functions  ////////////////////////
//////////////////////////////////////////////////////////////////////////

void oe2rv(MatrixXd& oe, double mu, MatrixXd* ri, MatrixXd* vi);

void rv2oe(adouble* rv, adouble* vv, double mu, adouble* oe);


//////////////////////////////////////////////////////////////////////////
/////////  Declare an auxiliary structure to hold local constants  ///////
//////////////////////////////////////////////////////////////////////////

struct Constants {
  MatrixXd* omega_matrix;
  double mu;
  double cd;
  double sa;
  double rho0;
  double H;
  double Re;
  double g0;
  double thrust_srb;
  double thrust_first;
  double thrust_second;
  double ISP_srb;
  double ISP_first;
  double ISP_second;
};

typedef struct Constants Constants_;


//////////////////////////////////////////////////////////////////////////
///////////////////  Define the end point (Mayer) cost function //////////
//////////////////////////////////////////////////////////////////////////

adouble endpoint_cost(adouble* initial_states, adouble* final_states,
                      adouble* parameters,adouble& t0, adouble& tf,
                      adouble* xad, int iphase, Workspace* workspace)
{
    adouble retval;
    adouble mass_tf = final_states[6];

    if (iphase < 4)
      retval = 0.0;

    if (iphase== 4)
      retval = -mass_tf;

    return retval;
}

//////////////////////////////////////////////////////////////////////////
///////////////////  Define the integrand (Lagrange) cost function  //////
//////////////////////////////////////////////////////////////////////////

adouble integrand_cost(adouble* states, adouble* controls, adouble* parameters,
                     adouble& time, adouble* xad, int iphase, Workspace* workspace)

{
    return  0.0;
}
```

```
/////////////////////////////////////////////////////////////////////////////
///////////////////  Define the DAE's ////////////////////////////////////////
/////////////////////////////////////////////////////////////////////////////

void dae(adouble* derivatives, adouble* path, adouble* states,
         adouble* controls, adouble* parameters, adouble& time,
         adouble* xad, int iphase, Workspace* workspace)
{

    int j;

    Constants_& CONSTANTS = *( (Constants_ *) workspace->problem->user_data );

    adouble* x = states;
    adouble* u = controls;

    adouble  r[3]; r[0]=x[0]; r[1]=x[1]; r[2]=x[2];

    adouble  v[3]; v[0]=x[3]; v[1]=x[4]; v[2]=x[5];

    adouble  m = x[6];

    double T_first, T_second, T_srb, T_tot, m1dot, m2dot, mdot;

    adouble rad = sqrt( dot( r, r, 3)  );

    MatrixXd& omega_matrix = *CONSTANTS.omega_matrix;

    adouble vrel[3];
    for (j=0;j<3;j++)
      vrel[j] =  v[j] - omega_matrix(j,0)*r[0] -omega_matrix(j,1)*r[1] - omega_matrix(j,2)*r[2];


    adouble speedrel = sqrt( dot(vrel,vrel,3) );
    adouble altitude = rad-CONSTANTS.Re;

    adouble rho = CONSTANTS.rho0*exp(-altitude/CONSTANTS.H);
    double   a1 = CONSTANTS.rho0*CONSTANTS.sa*CONSTANTS.cd;
    adouble  a2 = a1*exp(-altitude/CONSTANTS.H);
    adouble  bc = (rho/(2*m))*CONSTANTS.sa*CONSTANTS.cd;



    adouble bcspeed    = bc*speedrel;

    adouble Drag[3];
    for(j=0;j<3;j++) Drag[j] =  - (vrel[j]*bcspeed);



    adouble muoverradcubed = (CONSTANTS.mu)/(pow(rad,3));
    adouble grav[3];
    for(j=0;j<3;j++) grav[j] = -muoverradcubed*r[j];


   if (iphase==1) {
     T_srb = 6*CONSTANTS.thrust_srb;
     T_first = CONSTANTS.thrust_first;
     T_tot = T_srb+T_first;
     m1dot = -T_srb/(CONSTANTS.g0*CONSTANTS.ISP_srb);
     m2dot = -T_first/(CONSTANTS.g0*CONSTANTS.ISP_first);
     mdot = m1dot+m2dot;
   }
   else if (iphase==2) {
     T_srb = 3*CONSTANTS.thrust_srb;
     T_first = CONSTANTS.thrust_first;
     T_tot = T_srb+T_first;
     m1dot = -T_srb/(CONSTANTS.g0*CONSTANTS.ISP_srb);
     m2dot = -T_first/(CONSTANTS.g0*CONSTANTS.ISP_first);
     mdot = m1dot+m2dot;
   }
   else if (iphase==3) {
     T_first = CONSTANTS.thrust_first;
     T_tot = T_first;
     mdot = -T_first/(CONSTANTS.g0*CONSTANTS.ISP_first);
   }
   else if (iphase==4) {
     T_second = CONSTANTS.thrust_second;
     T_tot = T_second;
     mdot = -T_second/(CONSTANTS.g0*CONSTANTS.ISP_second);
   }
```

```
   adouble Toverm = T_tot/m;

   adouble thrust[3];

   for(j=0;j<3;j++) thrust[j] = Toverm*u[j];

   adouble rdot[3];
   for(j=0;j<3;j++) rdot[j] = v[j];

   adouble vdot[3];
   for(j=0;j<3;j++)  vdot[j] = thrust[j]+Drag[j]+grav[j];


   derivatives[0] = rdot[0];
   derivatives[1] = rdot[1];
   derivatives[2] = rdot[2];
   derivatives[3] = vdot[0];
   derivatives[4] = vdot[1];
   derivatives[5] = vdot[2];
   derivatives[6] = mdot;

   path[0] = dot( controls, controls, 3);

}

////////////////////////////////////////////////////////////////////////////
///////////////////  Define the events function ////////////////////////////
////////////////////////////////////////////////////////////////////////////

void events(adouble* e, adouble* initial_states, adouble* final_states,
            adouble* parameters,adouble& t0, adouble& tf, adouble* xad,
            int iphase, Workspace* workspace)
{


   Constants_& CONSTANTS = *( (Constants_ *) workspace->problem->user_data );

   adouble rv[3]; rv[0]=final_states[0]; rv[1]=final_states[1]; rv[2]=final_states[2];
   adouble vv[3]; vv[0]=final_states[3]; vv[1]=final_states[4]; vv[2]=final_states[5];

   adouble oe[6];

   int j;

   if(iphase==1) {
       // These events are related to the initial state conditions in phase 1
       for(j=0;j<7;j++) e[j] = initial_states[j];
   }

   if (iphase==4) {
     // These events are related to the final states in phase 4
     rv2oe( rv, vv, CONSTANTS.mu, oe );
     for(j=0;j<5;j++) e[j]=oe[j];

   }



}


///////////////////////////////////////////////////////////////////////////
///////////////////  Define the phase linkages function ///////////////////
///////////////////////////////////////////////////////////////////////////

void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{

    double m_tot_first   = 104380.0;
    double m_prop_first  = 95550.0;
    double m_dry_first   = m_tot_first-m_prop_first;
    double m_tot_srb     = 19290.0;
    double m_prop_srb    = 17010.0;
    double m_dry_srb     = m_tot_srb-m_prop_srb;

    int index=0;

    auto_link(linkages, &index, xad, 1, 2, workspace );
    linkages[index-2]-= 6*m_dry_srb;
    auto_link(linkages, &index, xad, 2, 3, workspace );
    linkages[index-2]-= 3*m_dry_srb;
    auto_link(linkages, &index, xad, 3, 4, workspace );
```

```
    linkages[index-2]-= m_dry_first;

}



////////////////////////////////////////////////////////////////////////////
///////////////////  Define the main routine ///////////////////////////////
////////////////////////////////////////////////////////////////////////////

int main(void)
{
////////////////////////////////////////////////////////////////////////////
///////////////////  Declare key structures ////////////////////////////////
////////////////////////////////////////////////////////////////////////////

    Alg  algorithm;
    Sol  solution;
    Prob problem;

////////////////////////////////////////////////////////////////////////////
///////////////////  Register problem name  ////////////////////////////////
////////////////////////////////////////////////////////////////////////////

    problem.name         = "Multiphase vehicle launch";
    problem.outfilename                 = "launch.txt";

////////////////////////////////////////////////////////////////////////////
///////////////////  Declare an instance of Constants structure /////////////
////////////////////////////////////////////////////////////////////////////


    Constants_ CONSTANTS;

    problem.user_data = (void*) &CONSTANTS;



////////////////////////////////////////////////////////////////////////////
////////////  Define problem level constants & do level 1 setup ////////////
////////////////////////////////////////////////////////////////////////////

    problem.nphases    = 4;
    problem.nlinkages                  = 24;

    psopt_level1_setup(problem);

/////////////////////////////////////////////////////////////////////////////
/////////   Define phase related information & do level 2 setup  ////////////
/////////////////////////////////////////////////////////////////////////////

    problem.phases(1).nstates   = 7;
    problem.phases(1).ncontrols = 3;
    problem.phases(1).nevents   = 7;
    problem.phases(1).npath     = 1;

    problem.phases(2).nstates   = 7;
    problem.phases(2).ncontrols = 3;
    problem.phases(2).nevents   = 0;
    problem.phases(2).npath     = 1;


    problem.phases(3).nstates   = 7;
    problem.phases(3).ncontrols = 3;
    problem.phases(3).nevents   = 0;
    problem.phases(3).npath     = 1;

    problem.phases(4).nstates   = 7;
    problem.phases(4).ncontrols = 3;
    problem.phases(4).nevents   = 5;
    problem.phases(4).npath     = 1;

    problem.phases(1).nodes     = (RowVectorXi(2) << 15, 18).finished();
    problem.phases(2).nodes     = (RowVectorXi(2) << 15, 18).finished();
    problem.phases(3).nodes     = (RowVectorXi(2) << 15, 18).finished();
    problem.phases(4).nodes     = (RowVectorXi(2) << 20, 25).finished();





    psopt_level2_setup(problem, algorithm);
```

```
////////////////////////////////////////////////////////////////////////////
///////////////////  Declare MatrixXd objects to store results /////////////
////////////////////////////////////////////////////////////////////////////

    MatrixXd x, u, t, H;

////////////////////////////////////////////////////////////////////////////
///////////////////  Initialize CONSTANTS and //////////////////////////////
///////////////////  declare local variables  //////////////////////////////
////////////////////////////////////////////////////////////////////////////


    double omega            = 7.29211585e-5; // Earth rotation rate (rad/s)
    MatrixXd omega_matrix(3,3);

    omega_matrix(0,0) = 0.0;   omega_matrix(0,1) = -omega;   omega_matrix(0,2)=0.0;
    omega_matrix(1,0) = omega; omega_matrix(1,1) = 0.0;      omega_matrix(1,2)=0.0;
    omega_matrix(2,0) = 0.0;   omega_matrix(2,1) = 0.0;      omega_matrix(2,2)=0.0;

    CONSTANTS.omega_matrix = &omega_matrix; // Rotation rate matrix (rad/s)
    CONSTANTS.mu = 3.986012e14;       // Gravitational parameter (m^3/s^2)
    CONSTANTS.cd = 0.5;               // Drag coefficient
    CONSTANTS.sa = 4*pi;              // Surface area (m^2)
    CONSTANTS.rho0 = 1.225;           // sea level gravity (kg/m^3)
    CONSTANTS.H = 7200.0;             // Density scale height (m)
    CONSTANTS.Re = 6378145.0;         // Radius of earth (m)
    CONSTANTS.g0 = 9.80665;           // sea level gravity (m/s^2)

    double lat0 = 28.5*pi/180.0;              // Geocentric Latitude of Cape Canaveral
    double x0 = CONSTANTS.Re*cos(lat0);      // x component of initial position
    double z0 = CONSTANTS.Re*sin(lat0);      // z component of initial position
    double y0 = 0;
    MatrixXd r0(3,1); r0 <<  x0, y0, z0;
    MatrixXd v0 = omega_matrix*r0;

    double bt_srb = 75.2;
    double bt_first = 261.0;
    double bt_second = 700.0;

    double t0 = 0;
    double t1 = 75.2;
    double t2 = 150.4;
    double t3 = 261.0;
    double t4 = 961.0;

    double m_tot_srb     = 19290.0;
    double m_prop_srb    = 17010.0;
    double m_dry_srb     = m_tot_srb-m_prop_srb;
    double m_tot_first   = 104380.0;
    double m_prop_first  = 95550.0;
    double m_dry_first   = m_tot_first-m_prop_first;
    double m_tot_second  = 19300.0;
    double m_prop_second = 16820.0;
    double m_dry_second  = m_tot_second-m_prop_second;
    double m_payload     = 4164.0;
    double thrust_srb    = 628500.0;
    double thrust_first  = 1083100.0;
    double thrust_second = 110094.0;
    double mdot_srb      = m_prop_srb/bt_srb;
    double ISP_srb       = thrust_srb/(CONSTANTS.g0*mdot_srb);
    double mdot_first    = m_prop_first/bt_first;
    double ISP_first     = thrust_first/(CONSTANTS.g0*mdot_first);
    double mdot_second   = m_prop_second/bt_second;
    double ISP_second     = thrust_second/(CONSTANTS.g0*mdot_second);

    double af = 24361140.0;
    double ef = 0.7308;
    double incf = 28.5*pi/180.0;
    double Omf = 269.8*pi/180.0;
    double omf = 130.5*pi/180.0;
    double nuguess = 0;
    double cosincf = cos(incf);
    double cosOmf = cos(Omf);
    double cosomf = cos(omf);
    MatrixXd oe(6,1); oe << af, ef, incf, Omf, omf, nuguess;

    MatrixXd rout(3,1);
    MatrixXd vout(3,1);

    oe2rv(oe,CONSTANTS.mu, &rout, &vout);

    rout= rout.transpose().eval();
```

```
    vout= vout.transpose().eval();

    double m10 = m_payload+m_tot_second+m_tot_first+9*m_tot_srb;
    double m1f = m10-(6*mdot_srb+mdot_first)*t1;
    double m20 = m1f-6*m_dry_srb;
    double m2f = m20-(3*mdot_srb+mdot_first)*(t2-t1);
    double m30 = m2f-3*m_dry_srb;
    double m3f = m30-mdot_first*(t3-t2);
    double m40 = m3f-m_dry_first;
    double m4f = m_payload;

    CONSTANTS.thrust_srb    = thrust_srb;
    CONSTANTS.thrust_first  = thrust_first;
    CONSTANTS.thrust_second = thrust_second;
    CONSTANTS.ISP_srb       = ISP_srb;
    CONSTANTS.ISP_first     = ISP_first;
    CONSTANTS.ISP_second    = ISP_second;


    double rmin = -2*CONSTANTS.Re;
    double rmax = -rmin;
    double vmin = -10000.0;
    double vmax = -vmin;

////////////////////////////////////////////////////////////////////////////
///////////////////  Enter problem bounds information //////////////////////
////////////////////////////////////////////////////////////////////////////


    int iphase;


    // Phase 1 bounds

    iphase =  1;


    problem.phases(iphase).bounds.lower.states   << rmin, rmin, rmin, vmin, vmin, vmin, m1f;
    problem.phases(iphase).bounds.upper.states   << rmax, rmax, rmax, vmax, vmax, vmax, m10;

    problem.phases(iphase).bounds.lower.controls << -1.0, -1.0, -1.0;
    problem.phases(iphase).bounds.upper.controls <<  1.0,  1.0,  1.0;

    problem.phases(iphase).bounds.lower.path     <<  1.0;
    problem.phases(iphase).bounds.upper.path     <<  1.0;


    // The following bounds fix the initial state conditions in phase 0.

    problem.phases(iphase).bounds.lower.events << r0(0), r0(1), r0(2), v0(0), v0(1), v0(2), m10;
    problem.phases(iphase).bounds.upper.events << r0(0), r0(1), r0(2), v0(0), v0(1), v0(2), m10;

    problem.phases(iphase).bounds.lower.StartTime    = 0.0;
    problem.phases(iphase).bounds.upper.StartTime    = 0.0;

    problem.phases(iphase).bounds.lower.EndTime    = 75.2;
    problem.phases(iphase).bounds.upper.EndTime    = 75.2;


    // Phase 2 bounds

    iphase =  2;

    problem.phases(iphase).bounds.lower.states   << rmin, rmin, rmin, vmin, vmin, vmin, m2f;
    problem.phases(iphase).bounds.upper.states   << rmax, rmax, rmax, vmax, vmax, vmax, m20;


    problem.phases(iphase).bounds.lower.controls << -1.0, -1.0, -1.0;
    problem.phases(iphase).bounds.upper.controls <<  1.0,  1.0,  1.0;

    problem.phases(iphase).bounds.lower.path     <<  1.0;
    problem.phases(iphase).bounds.upper.path     <<  1.0;

    problem.phases(iphase).bounds.lower.StartTime    = 75.2;
    problem.phases(iphase).bounds.upper.StartTime    = 75.2;

    problem.phases(iphase).bounds.lower.EndTime    = 150.4;
    problem.phases(iphase).bounds.upper.EndTime    = 150.4;

    // Phase 3 bounds

    iphase =  3;
```

```
    problem.phases(iphase).bounds.lower.states   << rmin, rmin, rmin, vmin, vmin, vmin, m3f;
    problem.phases(iphase).bounds.upper.states   << rmax, rmax, rmax, vmax, vmax, vmax, m30;

    problem.phases(iphase).bounds.lower.controls << -1.0, -1.0, -1.0;
    problem.phases(iphase).bounds.upper.controls <<  1.0,  1.0,  1.0;

    problem.phases(iphase).bounds.lower.path     <<  1.0;
    problem.phases(iphase).bounds.upper.path     <<  1.0;

    problem.phases(iphase).bounds.lower.StartTime  = 150.4;
    problem.phases(iphase).bounds.upper.StartTime  = 150.4;

    problem.phases(iphase).bounds.lower.EndTime    = 261.0;
    problem.phases(iphase).bounds.upper.EndTime    = 261.0;


    // Phase 4 bounds

    iphase =  4;


    problem.phases(iphase).bounds.lower.states   << rmin, rmin, rmin, vmin, vmin, vmin, m4f;
    problem.phases(iphase).bounds.upper.states   << rmax, rmax, rmax, vmax, vmax, vmax, m40;


    problem.phases(iphase).bounds.lower.controls << -1.0, -1.0, -1.0;
    problem.phases(iphase).bounds.upper.controls <<  1.0,  1.0,  1.0;

    problem.phases(iphase).bounds.lower.path     <<  1.0;
    problem.phases(iphase).bounds.upper.path     <<  1.0;

    problem.phases(iphase).bounds.lower.events   << af, ef, incf, Omf, omf;
    problem.phases(iphase).bounds.upper.events   << af, ef, incf, Omf, omf;

    problem.phases(iphase).bounds.lower.StartTime    = 261.0;
    problem.phases(iphase).bounds.upper.StartTime    = 261.0;

    problem.phases(iphase).bounds.lower.EndTime    = 261.0;
    problem.phases(iphase).bounds.upper.EndTime    = 961.0;


////////////////////////////////////////////////////////////////////////////
//////////////////  Define & register initial guess ///////////////////////
////////////////////////////////////////////////////////////////////////////

    iphase = 1;

    problem.phases(iphase).guess.states = zeros(7,5);

    problem.phases(iphase).guess.states.row(0) = linspace( r0(0), r0(0), 5);
    problem.phases(iphase).guess.states.row(1) = linspace( r0(1), r0(1), 5);
    problem.phases(iphase).guess.states.row(2) = linspace( r0(2), r0(2), 5);
    problem.phases(iphase).guess.states.row(3) = linspace( v0(0), v0(0), 5);
    problem.phases(iphase).guess.states.row(4) = linspace( v0(1), v0(1), 5);
    problem.phases(iphase).guess.states.row(5) = linspace( v0(2), v0(2), 5);
    problem.phases(iphase).guess.states.row(6) = linspace( m10  , m1f  , 5);

    problem.phases(iphase).guess.controls = zeros(3,5);

    problem.phases(iphase).guess.controls.row(0) = ones( 1, 5);
    problem.phases(iphase).guess.controls.row(1) = zeros(1, 5);
    problem.phases(iphase).guess.controls.row(2) = zeros(1, 5);


    problem.phases(iphase).guess.time = linspace(t0,t1, 5);


    iphase = 2;

    problem.phases(iphase).guess.states = zeros(7,5);

    problem.phases(iphase).guess.states.row(0) = linspace( r0(0), r0(0), 5);
    problem.phases(iphase).guess.states.row(1) = linspace( r0(1), r0(1), 5);
    problem.phases(iphase).guess.states.row(2) = linspace( r0(2), r0(2), 5);
    problem.phases(iphase).guess.states.row(3) = linspace( v0(0), v0(0), 5);
    problem.phases(iphase).guess.states.row(4) = linspace( v0(1), v0(1), 5);
    problem.phases(iphase).guess.states.row(5) = linspace( v0(2), v0(2), 5);
    problem.phases(iphase).guess.states.row(6) = linspace( m20  , m2f  , 5);

    problem.phases(iphase).guess.controls = zeros(3,5);
```

```
    problem.phases(iphase).guess.controls.row(0) = ones( 1, 5);
    problem.phases(iphase).guess.controls.row(1) = zeros(1, 5);
    problem.phases(iphase).guess.controls.row(2) = zeros(1, 5);


    problem.phases(iphase).guess.time = linspace(t1,t2, 5);

    iphase = 3;

    problem.phases(iphase).guess.states = zeros(7,5);

    problem.phases(iphase).guess.states.row(0) = linspace( r0(0), r0(0), 5);
    problem.phases(iphase).guess.states.row(1) = linspace( r0(1), r0(1), 5);
    problem.phases(iphase).guess.states.row(2) = linspace( r0(2), r0(2), 5);
    problem.phases(iphase).guess.states.row(3) = linspace( v0(0), v0(0), 5);
    problem.phases(iphase).guess.states.row(4) = linspace( v0(1), v0(1), 5);
    problem.phases(iphase).guess.states.row(5) = linspace( v0(2), v0(2), 5);
    problem.phases(iphase).guess.states.row(6) = linspace( m30  , m3f  , 5);

    problem.phases(iphase).guess.controls = zeros(3,5);

    problem.phases(iphase).guess.controls.row(0) = ones( 1, 5);
    problem.phases(iphase).guess.controls.row(1) = zeros(1, 5);
    problem.phases(iphase).guess.controls.row(2) = zeros(1, 5);


    problem.phases(iphase).guess.time = linspace(t2,t3, 5);

    iphase = 4;

    problem.phases(iphase).guess.states = zeros(7,5);

    problem.phases(iphase).guess.states.row(0) = linspace( rout(0), rout(0), 5);
    problem.phases(iphase).guess.states.row(1) = linspace( rout(1), rout(1), 5);
    problem.phases(iphase).guess.states.row(2) = linspace( rout(2), rout(2), 5);
    problem.phases(iphase).guess.states.row(3) = linspace( vout(0), vout(0), 5);
    problem.phases(iphase).guess.states.row(4) = linspace( vout(1), vout(1), 5);
    problem.phases(iphase).guess.states.row(5) = linspace( vout(2), vout(2), 5);
    problem.phases(iphase).guess.states.row(6) = linspace( m40  , m4f  , 5);

    problem.phases(iphase).guess.controls = zeros(3,5);

    problem.phases(iphase).guess.controls.row(0) = ones( 1, 5);
    problem.phases(iphase).guess.controls.row(1) = zeros(1, 5);
    problem.phases(iphase).guess.controls.row(2) = zeros(1, 5);

    problem.phases(iphase).guess.time = linspace(t3,t4, 5);


////////////////////////////////////////////////////////////////////////////
///////////////////  Register problem functions  ///////////////////////////
////////////////////////////////////////////////////////////////////////////


    problem.integrand_cost  = &integrand_cost;
    problem.endpoint_cost  = &endpoint_cost;
    problem.dae  = &dae;
    problem.events  = &events;
    problem.linkages = &linkages;

////////////////////////////////////////////////////////////////////////////
///////////////////  Enter algorithm options  //////////////////////////////
////////////////////////////////////////////////////////////////////////////


    algorithm.nlp_method                  = "IPOPT";
    algorithm.scaling                     = "automatic";
    algorithm.derivatives                 = "automatic";
    algorithm.nlp_iter_max                = 1000;
    algorithm.collocation_method            = "Chebyshev";
//    algorithm.mesh_refinement             = "automatic";
//    algorithm.ode_tolerance           = 1.e-5;

////////////////////////////////////////////////////////////////////////////
///////////////////  Now call PSOPT to solve the problem   /////////////////
////////////////////////////////////////////////////////////////////////////

    psopt(solution, problem, algorithm);

////////////////////////////////////////////////////////////////////////////
//////////  Extract relevant variables from solution structure   //////////
////////////////////////////////////////////////////////////////////////////
```

```
    MatrixXd x_ph1, x_ph2, x_ph3, x_ph4, u_ph1, u_ph2, u_ph3, u_ph4;
    MatrixXd t_ph1, t_ph2, t_ph3, t_ph4;

    x_ph1 = solution.get_states_in_phase(1);
    x_ph2 = solution.get_states_in_phase(2);
    x_ph3 = solution.get_states_in_phase(3);
    x_ph4 = solution.get_states_in_phase(4);

    u_ph1 = solution.get_controls_in_phase(1);
    u_ph2 = solution.get_controls_in_phase(2);
    u_ph3 = solution.get_controls_in_phase(3);
    u_ph4 = solution.get_controls_in_phase(4);

    t_ph1 = solution.get_time_in_phase(1);
    t_ph2 = solution.get_time_in_phase(2);
    t_ph3 = solution.get_time_in_phase(3);
    t_ph4 = solution.get_time_in_phase(4);



    x.resize(7, x_ph1.cols()+ x_ph2.cols()+ x_ph3.cols()+ x_ph4.cols() );
    u.resize(3, u_ph1.cols()+ u_ph2.cols()+ u_ph3.cols()+ u_ph4.cols() );
    t.resize(1, t_ph1.cols()+ t_ph2.cols()+ t_ph3.cols()+ t_ph4.cols() );

    x << x_ph1, x_ph2, x_ph3, x_ph4;
    u << u_ph1, u_ph2, u_ph3, u_ph4;
    t << t_ph1, t_ph2, t_ph3, t_ph4;


////////////////////////////////////////////////////////////////////////////
///////////  Save solution data to files if desired ////////////////////////
////////////////////////////////////////////////////////////////////////////

    Save(x,"x.dat");
    Save(u,"u.dat");
    Save(t,"t.dat");

////////////////////////////////////////////////////////////////////////////
///////////  Plot some results if desired (requires gnuplot) ///////////////
////////////////////////////////////////////////////////////////////////////

    MatrixXd r, v, altitude, speed;

    r = x.block(0,0,3,x.cols());

    v = x.block(3,0,3,x.cols());

    altitude = (sum_columns(elemProduct(r,r)).cwiseSqrt())/1000.0;

    speed = sum_columns(elemProduct(v,v)).cwiseSqrt();

    plot(t,altitude,problem.name, "time (s)", "Altitude (km)");


    plot(t,speed,problem.name, "time (s)", "speed (m/s)");

    plot(t,u,problem.name,"time (s)", "u");


    plot(t,altitude,problem.name, "time (s)", "Altitude (km)", "alt",
                                  "pdf", "launch_altitude.pdf");

    plot(t,speed,problem.name, "time (s)", "speed (m/s)", "speed",
                               "pdf", "launch_speed.pdf");

    plot(t,u,problem.name,"time (s)", "u (dimensionless)", "u1 u2 u3",
                               "pdf", "launch_control.pdf");



}

////////////////////////////////////////////////////////////////////////////
////////////////// Define auxiliary functions //////////////////////////////
////////////////////////////////////////////////////////////////////////////


void rv2oe(adouble* rv, adouble* vv, double mu, adouble* oe)
{
       int j;

       adouble K[3]; K[0] = 0.0; K[1]=0.0; K[2]=1.0;
```

```
        adouble hv[3];
        cross(rv,vv, hv);

        adouble nv[3];
        cross(K, hv, nv);

        adouble n  = sqrt(  dot(nv,nv,3) );

        adouble h2 = dot(hv,hv,3);

        adouble v2 = dot(vv,vv,3);

        adouble r         = sqrt(dot(rv,rv,3));

        adouble ev[3];
        for(j=0;j<3;j++)  ev[j] = 1/mu *( (v2-mu/r)*rv[j] - dot(rv,vv,3)*vv[j] );

        adouble p         = h2/mu;

adouble e  = sqrt(dot(ev,ev,3)); // eccentricity
adouble a  = p/(1-e*e);    // semimajor axis
adouble i  = acos(hv[2]/sqrt(h2)); // inclination



#define USE_SMOOTH_HEAVISIDE
         double a_eps = 0.1;

#ifndef USE_SMOOTH_HEAVISIDE
  adouble Om = acos(nv[0]/n); // RAAN
  if ( nv[1] < -PSOPT_extras::GetEPS() ){ // fix quadrant
  Om = 2*pi-Om;
  }
#endif

#ifdef USE_SMOOTH_HEAVISIDE

         adouble Om =  smooth_heaviside( (nv[1]+PSOPT_extras::GetEPS()), a_eps )*acos(nv[0]/n)
                      +smooth_heaviside( -(nv[1]+PSOPT_extras::GetEPS()), a_eps )*(2*pi-acos(nv[0]/n));
#endif

#ifndef USE_SMOOTH_HEAVISIDE
  adouble om = acos(dot(nv,ev,3)/n/e); // arg of periapsis
  if ( ev[2] < 0 ) { // fix quadrant
  om = 2*pi-om;
  }
#endif

#ifdef USE_SMOOTH_HEAVISIDE
         adouble om =  smooth_heaviside( (ev[2]), a_eps )*acos(dot(nv,ev,3)/n/e)
                      +smooth_heaviside( -(ev[2]), a_eps )*(2*pi-acos(dot(nv,ev,3)/n/e));
#endif

#ifndef USE_SMOOTH_HEAVISIDE
  adouble nu = acos(dot(ev,rv,3)/e/r); // true anomaly
  if ( dot(rv,vv,3) < 0 ) { // fix quadrant
  nu = 2*pi-nu;
  }
#endif

#ifdef USE_SMOOTH_HEAVISIDE
        adouble nu =  smooth_heaviside( dot(rv,vv,3), a_eps )*acos(dot(ev,rv,3)/e/r)
                      +smooth_heaviside( -dot(rv,vv,3), a_eps )*(2*pi-acos(dot(ev,rv,3)/e/r));
#endif

         oe[0] = a;
         oe[1] = e;
         oe[2] = i;
         oe[3] = Om;
         oe[4] = om;
         oe[5] = nu;

         return;

}


void oe2rv(MatrixXd& oe, double mu, MatrixXd* ri, MatrixXd* vi)
{
double a=oe(0), e=oe(1), i=oe(2), Om=oe(3), om=oe(4), nu=oe(5);
double p = a*(1-e*e);
double r = p/(1+e*cos(nu));
```

```
MatrixXd rv(3,1);
        rv(0) = r*cos(nu);
        rv(1) = r*sin(nu);
        rv(2) = 0.0;

MatrixXd vv(3,1);

        vv(0) = -sin(nu);
        vv(1) = e+cos(nu);
        vv(2) = 0.0;
        vv    *= sqrt(mu/p);

double cO = cos(Om),  sO = sin(Om);
double co = cos(om),  so = sin(om);
double ci = cos(i),   si = sin(i);

MatrixXd R(3,3);
   R(0,0)=  cO*co-sO*so*ci;  R(0,1)=  -cO*so-sO*co*ci; R(0,2)=  sO*si;
R(1,0)=  sO*co+cO*so*ci; R(1,1)=  -sO*so+cO*co*ci; R(1,2)=-cO*si;
R(2,0)=   so*si;         R(2,1)=    co*si;         R(2,2)=  ci;

*ri = R*rv;
*vi = R*vv;

        return;
}

////////////////////////////////////////////////////////////////////////////
///////////////////////      END OF FILE     ///////////////////////////////
////////////////////////////////////////////////////////////////////////////
```

The output from $\mathcal{PSOPT}$ is summarised in the box below and shown in Figures 112, 113 and 114, which contain the trajectories of the altitude, speed and the elements of the control vector, respectively.

```
PSOPT results summary
=====================

Problem:  Multiphase vehicle launch
CPU time (seconds): 1.915171e+00
NLP solver used:  IPOPT
PSOPT release number:  5.0.3
Date and time of this run:  Thu Mar  6 16:11:58 2025

Optimal (unscaled) cost function value:  -7.529661e+03
Phase 1 endpoint cost function value: 0.000000e+00
Phase 1 integrated part of the cost:  0.000000e+00
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 7.520000e+01
Phase 1 maximum relative local error: 5.558508e-07
Phase 2 endpoint cost function value: 0.000000e+00
Phase 2 integrated part of the cost:  0.000000e+00
Phase 2 initial time: 7.520000e+01
Phase 2 final time: 1.504000e+02
Phase 2 maximum relative local error: 1.549203e-06
```

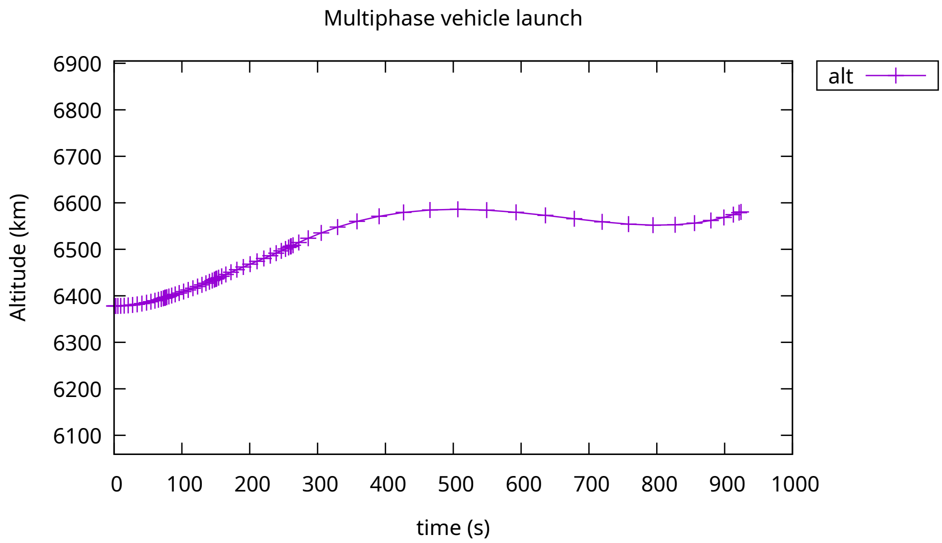

Figure 112: Altitude for the vehicle launch problem

```
Phase 3 endpoint cost function value: 0.000000e+00
Phase 3 integrated part of the cost:  0.000000e+00
Phase 3 initial time: 1.504000e+02
Phase 3 final time: 2.610000e+02
Phase 3 maximum relative local error: 5.739083e-07
Phase 4 endpoint cost function value: -7.529661e+03
Phase 4 integrated part of the cost:  0.000000e+00
Phase 4 initial time: 2.610000e+02
Phase 4 final time: 9.241413e+02
Phase 4 maximum relative local error: 1.377097e-06
NLP solver reports:  The problem has been solved!
```

44 Zero propellant maneouvre of the International Space Station

This problem illustrates the use of $\mathcal{PSOPT}$ for solving an optimal control problem associated with the design of a zero propellant maneouvre for the international space station by means of control moment gyroscopes (CMGs). The example is based on the results presented in the thesis by Bhatt [5] and also reported by Bedrossian and co-workers [1]. The original 90 and 180 degree maneouvres were computed using DIDO, and they were actually implemented on the International Space Station on 5 November 2006 and 2

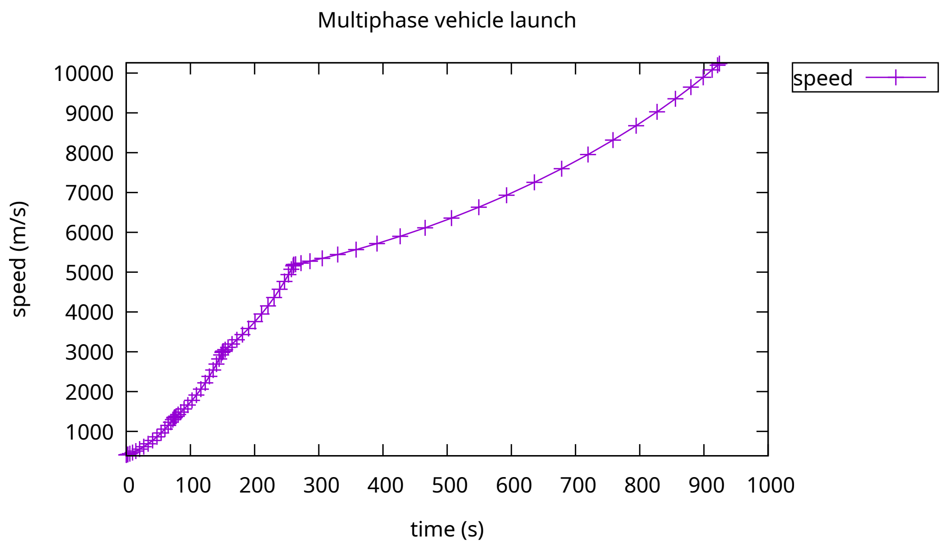

Figure 113: Speed for the vehicle launch problem

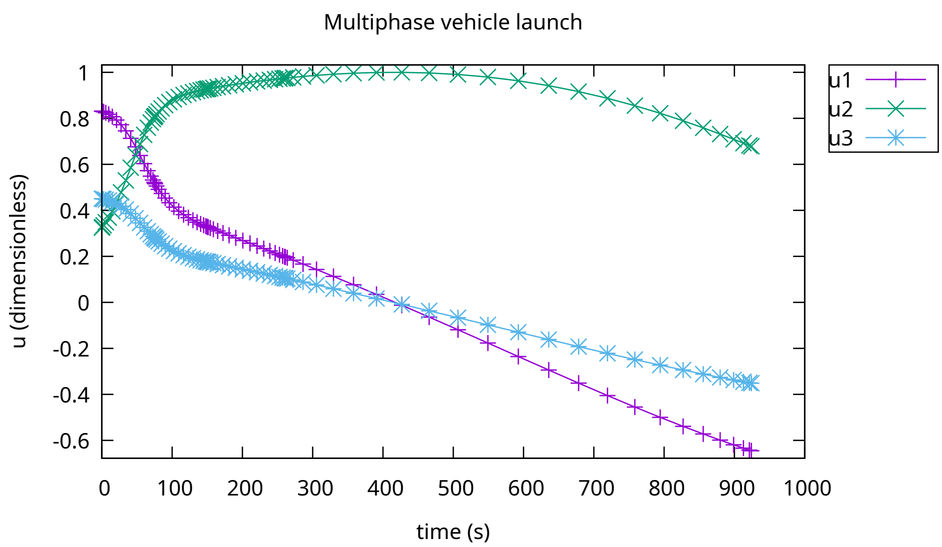

Figure 114: Controls for the vehicle launch problem

January 2007, respectively, resulting in savings for NASA of around US$1.5m in propellant costs. The dynamic model employed here does not account for atmospheric drag as the atmosphere model used in the original study is not available. Otherwise, the equations and parameters are the same as those reported by Bhatt in his thesis. The effects of atmospheric drag are, however, small, and the results obtained are comparable with those given in Bhatt's thesis. The implemented case corresponds with a maneovure lasting 7200 seconds and using 3 CMG's.

The problem is formulated as follows. Find $\mathbf{q_c}(t) = [q_{c,1}(t)\, q_{c,2}(t)\, q_{c,3}(t)\, q_{c,4}]^T$, $t \in [t_0, t_f]$ and the scalar parameter γ to minimise,

$$J = 0.1\gamma + \int_{t_0}^{t_f} ||u(t)||^2 \mathrm{d}t \tag{186}$$

subject to the dynamical equations:

$$\begin{aligned} \dot{\mathbf{q}}(t) &= \frac{1}{2}\mathbf{T}(\mathbf{q})(\omega(t) - \omega_o(\mathbf{q})) \\ \dot{\omega}(t) &= \mathbf{J}^{-1}\left(\tau_d(\mathbf{q}) - \omega(t) \times (\mathbf{J}\omega(t)) - \mathbf{u}(t)\right) \\ \dot{\mathbf{h}}(t) &= \mathbf{u}(t) - \omega(t) \times \mathbf{h}(t) \end{aligned} \tag{187}$$

the path constraints:

$$\begin{aligned} ||\mathbf{q}(t)||_2^2 &= 1 \\ ||\mathbf{q}_c(t)||_2^2 &= 1 \\ ||\mathbf{h}(t)||_2^2 &\le \gamma \\ ||\dot{\mathbf{h}}(t)||_2^2 &= \dot{h}_{\max}^2 \end{aligned} \tag{188}$$

the parameter bounds

$$0 \le \gamma \le h_{\max}^2 \tag{189}$$

and the boundary conditions:

$$\begin{array}{lll} \mathbf{q}(t_0) = \bar{\mathbf{q}}_0 & \omega(t_0) = \omega_o(\bar{\mathbf{q}}_0) & \mathbf{h}(t_0) = \bar{\mathbf{h}}_0 \\ \mathbf{q}(t_f) = \bar{\mathbf{q}}_\mathbf{f} & \omega(t_f) = \omega_o(\bar{\mathbf{q}}_f) & \mathbf{h}(t_f) = \bar{\mathbf{h}}_f \end{array} \tag{190}$$

where $\mathbf{J}$ is a 3×3 inertia matrix, $\mathbf{q} = [q_1, q_2, q_3, q_4]^T$ is the quarternion vector, ω is the spacecraft angular rate relative to an inertial reference frame and expressed in the body frame, $\mathbf{h}$ is the momentum, $\mathbf{T}(\mathbf{q})$ is given by:

$$\mathbf{T}(\mathbf{q}) = \begin{bmatrix} -q_2 & -q_3 & -q_4 \\ q_1 & -q_4 & q_3 \\ q_4 & q_1 & -q_2 \\ -q_3 & q_2 & q_1 \end{bmatrix} \tag{191}$$

$\mathbf{u}$ is the control force, which is given by:

$$\mathbf{u}(t) = \mathbf{J}\left(K_P \tilde{\varepsilon}(q, q_c) + K_D \tilde{\omega}(\omega, q_c)\right) \tag{192}$$

where

$$\begin{aligned} \tilde{\varepsilon}(\mathbf{q}, \mathbf{q}_c) &= 2\mathbf{T}(\mathbf{q}_c)^T\mathbf{q} \\ \tilde{\omega}(\omega, \omega_c) &= \omega - \omega_c \end{aligned} \tag{193}$$

ω_o is given by:

$$\omega_o(\mathbf{q}) = n\mathbf{C}_2(\mathbf{q}) \tag{194}$$

where n is the orbital rotation rate, $\mathbf{C}_j$ is the j column of the rotation matrix:

$$\mathbf{C}(\mathbf{q}) = \begin{bmatrix} 1-2(q_3^2+q_4^2) & 2(q_2q_3+q_1q_4) & 2(q_2q_4-q_1q_3) \\ 2(q_2q_3-q_1q_4) & 1-2(q_2^2+q_4^2) & 2(q_3q_4+q_1q_2) \\ 2(q_2q_4+q_1q_3) & 2(q_3q_4-q_1q_2) & 1-2(q_2^2+q_3^2) \end{bmatrix} \tag{195}$$

τ_d is the disturbance torque, which in this case only incorporates the gravity gradient torque τ_{gg} (the disturbance torque also incorporates the atmospheric drag torque in the original study):

$$\tau_d = \tau_{gg} = 3n^2\mathbf{C}_3(\mathbf{q}) \times (\mathbf{J}\mathbf{C}_3(\mathbf{q})) \tag{196}$$

The constant parameter values used were: $n = 1.1461 \times 10^{-3}$ rad/s, $h_{\max} = 3 \times 3600.0$ ft-lbf-sec, $\dot{h}_{\max} = 200.0$ ft-lbf, $t_0 = 0$ s, $t_f = 7200$ s, and

$$\mathbf{J} = \begin{bmatrix} 18836544.0 & 3666370.0 & 2965301.0 \\ 3666370.0 & 27984088.0 & -1129004.0 \\ 2965301.0 & -1129004.0 & 39442649.0 \end{bmatrix} \text{slug} - \text{ft}^2 \tag{197}$$

The C++ code that solves this problem is shown below.

```
//////////////////////////////////////////////////////////////////////////
////////////////             zpm.cxx              //////////////////////
//////////////////////////////////////////////////////////////////////////
////////////////           PSOPT Example          //////////////////////
//////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////
//////// Title:  Zero propellant maneouvre problem  ///////////////
//////// Last modified: 09 November 2009            ////////////////
//////// Reference:     S.A. Bhatt (2007), Masters Thesis,///////////////
////////                Rice University, Houston TX      ////////////////
//////////////////////////////////////////////////////////////////////////
////////     Copyright (c) Victor M. Becerra, 2009       ////////////////
//////////////////////////////////////////////////////////////////////////
//////// This is part of the PSOPT software library, which ///////////////
//////// is distributed under the terms of the GNU Lesser ///////////////
//////// General Public License (LGPL)                   ////////////////
//////////////////////////////////////////////////////////////////////////

#include "psopt.h"

using namespace PSOPT;


// Set CASE below to 1: 6000 s maneouvre, or to 2: 7200 maneouvre
#define CASE 2



struct Constants {
MatrixXd J;
double n;
double Kp;
double Kd;
double hmax;
};
```

```
typedef struct Constants Constants_;

static Constants_ CONSTANTS;


void Tfun( adouble T[][3], adouble *q )
{

adouble q1 = q[0];
adouble q2 = q[1];
adouble q3 = q[2];
adouble q4 = q[3];

T[0][0] = -q2 ; T[0][1] = -q3; T[0][2] = -q4;

T[1][0] =  q1 ; T[1][1] = -q4; T[1][2] =  q3;

T[2][0] =  q4 ; T[2][1] =  q1; T[2][2] = -q2;

T[3][0] = -q3;  T[3][1] =  q2; T[3][2] =  q1;

}


void compute_omega0(adouble* omega0, adouble* q)
{
// This function computes the angular speed in the rotating LVLH reference frame
int i;
double n = CONSTANTS.n;
adouble C2[3];

   adouble q1 = q[0];
   adouble q2 = q[1];
   adouble q3 = q[2];
   adouble q4 = q[3];

C2[ 0 ] = 2*(q2*q3 + q1*q4);
C2[ 1 ] = 1.0-2.0*(q2*q2+q4*q4);
C2[ 2 ] = 2*(q3*q4-q1*q2);

for (i=0;i<3;i++)   omega0[i] = -n*C2[i];

}

void compute_control_torque(adouble* u, adouble* q, adouble* qc, adouble* omega )
{
// This function computes the control torque
//

double Kp = CONSTANTS.Kp; // Proportional gain
double Kd = CONSTANTS.Kd; // Derivative gain
double n  = CONSTANTS.n;  // Orbital rotation rate [rad/s]


int i, j;

MatrixXd& J = CONSTANTS.J;


adouble T[4][3];

Tfun( T, q );

adouble Tc[4][3];

Tfun( Tc, qc );

adouble epsilon_tilde[3];

for(i=0;i<3;i++) {
epsilon_tilde[i] = 0.0;
for(j=0;j<4;j++) {
epsilon_tilde[i] += 2*Tc[j][i]*q[j];
}
}

adouble omega_c[3];

compute_omega0( omega_c, qc );

adouble omega_tilde[3];
```

```
for(i=0;i<3;i++) {
omega_tilde[i] = omega[i]-omega_c[i];
}

adouble uaux[3];

for(i=0;i<3;i++) {
uaux[i]= Kp*epsilon_tilde[i]+Kd*omega_tilde[i];
}

product_ad( J, uaux, 3, u );




}

void quarternion2Euler( MatrixXd& phi, MatrixXd& theta, MatrixXd& psi, MatrixXd& q)
{
// This function finds the Euler angles given the quarternion vector
//
long N = q.cols();
MatrixXd q0; q0 = q.row(0);
  MatrixXd q1; q1 = q.row(1);
  MatrixXd q2; q2 = q.row(2);
MatrixXd q3; q3 = q.row(3);

phi.resize(1,N);
theta.resize(1,N);
psi.resize(1,N);

    for(int i=0;i<N;i++) {
          phi(i)=atan2( 2*(q0(i)*q1(i) + q2(i)*q3(i)), 1.0-2.0*(q1(i)*q1(i)+q2(i)*q2(i)) );
            theta(i)=asin( 2*(q0(i)*q2(i)-q3(i)*q1(i)) );
            psi(i) = atan2( 2*(q0(i)*q3(i)+q1(i)*q2(i)), 1.0-2.0*(q2(i)*q2(i)+q3(i)*q3(i)) );
   }
}



void compute_aerodynamic_torque(adouble* tau_aero, adouble& time )
{
// This function approximates the aerodynamic torque by using the model and
// parameters given in the following reference:
// A. Chun Lee (2003) "Robust Momemtum Manager Controller for Space Station Applications".
// Master of Arts Thesis, Rice University.
//

double alpha1[3] = {1.0, 4.0, 1.0};
   double alpha2[3] = {1.0, 2.0, 1.0};
   double alpha3[3] = {0.5, 0.5, 0.5};
adouble t = time;

double phi1 = 0.0;
double phi2 = 0.0;

double n = CONSTANTS.n;

  for(int i=0;i<3;i++) {
                // Aerodynamic torque in [lb-ft]
tau_aero[i] = alpha1[i] + alpha2[i]*sin( n*t + phi1 ) + alpha3[i]*sin( 2*n*t + phi2);
}
}


///////////////////////////////////////////////////////////////////////////
//////////////////  Define the end point (Mayer) cost function //////////
///////////////////////////////////////////////////////////////////////////


adouble endpoint_cost(adouble* initial_states, adouble* final_states,
                      adouble* parameters,adouble& t0, adouble& tf,
                      adouble* xad, int iphase, Workspace* workspace)
{

double end_point_weight = 0.1;

   adouble gamma     = parameters[ 0 ];

return (end_point_weight*gamma);

}
```

```
////////////////////////////////////////////////////////////////////////////
///////////////////  Define the integrand (Lagrange) cost function  //////
////////////////////////////////////////////////////////////////////////////

adouble integrand_cost(adouble* states, adouble* controls, adouble* parameters,
                     adouble& time, adouble* xad, int iphase, Workspace* workspace)
{
double running_cost_weight = 1.0;

adouble q[4]; // quarternion vector
adouble u[3]; // control torque

q[0] = states[ 0 ];
q[1] = states[ 1 ];
q[2] = states[ 2 ];
q[3] = states[ 3 ];

adouble omega[3]; // angular rate vector

omega[0] = states[ 4 ];
omega[1] = states[ 5 ];
omega[2] = states[ 6 ];


adouble qc[4]; // control vector

qc[0] = controls[ 0 ];
qc[1] = controls[ 1 ];
qc[2] = controls[ 2 ];
qc[3] = controls[ 3 ];

compute_control_torque(u,q,qc,omega);

return running_cost_weight*dot(u,u,3);

}

////////////////////////////////////////////////////////////////////////////
///////////////////  Define the DAE's /////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
void dae(adouble* derivatives, adouble* path, adouble* states,
         adouble* controls, adouble* parameters, adouble& time,
         adouble* xad, int iphase, Workspace* workspace)
{

int i,j;


double n = CONSTANTS.n; // Orbital rotation rate [rad/s]

adouble q[4]; // quarternion vector

q[0] = states[ 0 ];
q[1] = states[ 1 ];
q[2] = states[ 2 ];
q[3] = states[ 3 ];

adouble omega[3]; // angular rate vector

omega[0] = states[ 4 ];
omega[1] = states[ 5 ];
omega[2] = states[ 6 ];

adouble h[3]; // momentum vector

h[0] = states[ 7 ];
h[1] = states[ 8 ];
h[2] = states[ 9 ];

adouble qc[4]; // control vector

qc[0] = controls[ 0 ];
qc[1] = controls[ 1 ];
qc[2] = controls[ 2 ];
qc[3] = controls[ 3 ];

adouble C2[3], C3[3];

adouble u[3];

adouble gamma;
```

```
gamma = parameters[ 0 ];


// Inertia matrix in slug-ft^2

MatrixXd& J = CONSTANTS.J;

MatrixXd Jinv; Jinv = J.inverse();

adouble q1 = q[0];
adouble q2 = q[1];
adouble q3 = q[2];
adouble q4 = q[3];

C3[ 0 ] = 2*(q2*q4 - q1*q3);
C3[ 1 ] = 2*(q3*q4 + q1*q2);
C3[ 2 ] = 1.0-2.0*(q2*q2 + q3*q3);



adouble T[4][3];

Tfun( T, q );

adouble qdot[4];

adouble omega0[3];

compute_omega0( omega0, q);

// Quarternion attitude kinematics

for(j=0;j<4;j++) {

qdot[j]=0;

for(i=0;i<3;i++) {

   qdot[j] += 0.5*T[j][i]*(omega[i]-omega0[i]);

}

}


adouble Jomega[3];

product_ad( J, omega, 3, Jomega );

adouble omegaCrossJomega[3];

cross(omega,Jomega, omegaCrossJomega);

adouble F[3];

// Compute the torque disturbances:

adouble tau_grav[3], tau_aero[3];

adouble v1[3];

for(i=0;i<3;i++) {
  v1[i] = 3*pow(n,2)*C3[i];
}

adouble JC3[3];

product_ad( J, C3, 3, JC3 );

//gravity gradient torque

cross( v1, JC3, tau_grav );

//Aerodynamic torque
compute_aerodynamic_torque(tau_aero, time );

for(i=0;i<3;i++) {
  // Uncomment this section to ignore the aerodynamic disturbance torque
  tau_aero[i] = 0.0;
}
adouble tau_d[3];
```

```
for (i=0;i<3;i++) {

   tau_d[i] =  tau_grav[i] + tau_aero[i];

}

compute_control_torque(u, q, qc, omega );


for (i=0;i<3;i++) {
   F[i]     =  tau_d[i] - omegaCrossJomega[i] - u[i];
}

adouble omega_dot[3];

// Rotational dynamics

product_ad( Jinv, F, 3, omega_dot );

adouble OmegaCrossH[3];

cross( omega, h , OmegaCrossH );

adouble hdot[3];


//Momemtum derivative

for(i=0; i<3; i++) {
   hdot[i] = u[i] - OmegaCrossH[i];
}



derivatives[0] =       qdot[ 0 ];
derivatives[1] =       qdot[ 1 ];
derivatives[2] =       qdot[ 2 ];
derivatives[3] =       qdot[ 3 ];
derivatives[4] =       omega_dot[ 0 ];
derivatives[5] =       omega_dot[ 1 ];
derivatives[6] =       omega_dot[ 2 ];
derivatives[7] =       hdot[ 0 ];
derivatives[8] =       hdot[ 1 ];
derivatives[9]=        hdot[ 2 ];


path[ 0 ] =   dot( q,  q,  4);
path[ 1 ] =   dot( qc, qc, 4);
path[ 2 ] =   dot( h, h, 3 ) - gamma;   //  <= 0
path[ 3 ] =   dot( hdot, hdot,3); // <= hdotmax^2,

}







////////////////////////////////////////////////////////////////////////////
//////////////////  Define the events function ////////////////////////////
////////////////////////////////////////////////////////////////////////////

void events(adouble* e, adouble* initial_states, adouble* final_states,
            adouble* parameters,adouble& t0, adouble& tf, adouble* xad,
            int iphase, Workspace* workspace)

{

adouble q1_i        = initial_states[0];
adouble q2_i        = initial_states[1];
adouble q3_i        = initial_states[2];
adouble q4_i        = initial_states[3];
adouble omega1_i    = initial_states[4];
adouble omega2_i    = initial_states[5];
adouble omega3_i    = initial_states[6];
adouble h1_i        = initial_states[7];
adouble h2_i        = initial_states[8];
adouble h3_i        = initial_states[9];

adouble q1_f        = final_states[0];
adouble q2_f        = final_states[1];
```

```
adouble q3_f        = final_states[2];
adouble q4_f        = final_states[3];
adouble omega1_f    = final_states[4];
adouble omega2_f    = final_states[5];
adouble omega3_f    = final_states[6];
adouble h1_f        = final_states[7];
adouble h2_f        = final_states[8];
adouble h3_f        = final_states[9];

// Initial conditions

e[ 0 ] = q1_i;
e[ 1 ] = q2_i;
e[ 2 ] = q3_i;
e[ 3 ] = q4_i;
e[ 4 ] = omega1_i;
e[ 5 ] = omega2_i;
e[ 6 ] = omega3_i;
e[ 7 ] = h1_i;
e[ 8 ] = h2_i;
e[ 9 ] = h3_i;

// Final conditions

 e[ 10 ] = q1_f;
 e[ 11 ] = q2_f;
 e[ 12 ] = q3_f;
 e[ 13 ] = q4_f;
 e[ 14 ] = omega1_f;
 e[ 15 ] = omega2_f;
 e[ 16 ] = omega3_f;
 e[ 17 ] = h1_f;
 e[ 18 ] = h2_f;
 e[ 19 ] = h3_f;


}



///////////////////////////////////////////////////////////////////////////
//////////////////  Define the phase linkages function ///////////////////
///////////////////////////////////////////////////////////////////////////

void linkages( adouble* linkages, adouble* xad, Workspace* workspace)
{
    // Single phase
}



////////////////////////////////////////////////////////////////////////////
//////////////////  Define the main routine ///////////////////////////////
////////////////////////////////////////////////////////////////////////////

int main(void)
{

////////////////////////////////////////////////////////////////////////////
//////////////////  Declare key structures ////////////////////////////////
////////////////////////////////////////////////////////////////////////////

    Alg  algorithm;
    Sol  solution;
    Prob problem;



   CONSTANTS.Kp = 0.000128; // Proportional gain
   CONSTANTS.Kd = 0.015846; // Derivative gain

   double hmax; // maximum momentum magnitude  in [ft-lbf-sec]


   if (CASE==1) {
CONSTANTS.n = 1.1461E-3; // Orbital rotation rate [rad/s]
hmax = 4*3600.0; // 4 CMG's
   }
   else if (CASE==2) {
CONSTANTS.n = 1.1475E-3;
hmax = 3*3600.0; // 3 CMG's
   }
```

```
CONSTANTS.hmax = hmax;

   MatrixXd& J = CONSTANTS.J;

   J.resize(3,3);

   // Inertia matrix in slug-ft^2

   if (CASE==1) {
J(0,0) = 17834580.0 ; J(0,1)= 2787992.0; J(0,2)= 2873636.0;
J(1,0) = 2787992.0  ; J(1,1)= 2773815.0; J(1,2)= -863810.0;
J(2,0) = 28736361.0 ; J(2,1)= -863810.0; J(2,2)=  38030467.0;
   }

   else if (CASE==2) {
J(0,0) = 18836544.0 ; J(0,1)= 3666370.0;  J(0,2)= 2965301.0;
J(1,0) = 3666370.0  ; J(1,1)= 27984088.0; J(1,2)= -1129004.0;
J(2,0) = 2965301.0  ; J(2,1)= -1129004.0; J(2,2)=  39442649.0;
   }



////////////////////////////////////////////////////////////////////////////
///////////////////  Register problem name  ////////////////////////////////
////////////////////////////////////////////////////////////////////////////

    problem.name         = "Zero Propellant Maneouvre of the ISS";
    problem.outfilename                 = "zpm.txt";

////////////////////////////////////////////////////////////////////////////
////////////  Define problem level constants & do level 1 setup ////////////
////////////////////////////////////////////////////////////////////////////

    problem.nphases    = 1;
    problem.nlinkages                    = 0;

    psopt_level1_setup(problem);

/////////////////////////////////////////////////////////////////////////////
/////////   Define phase related information & do level 2 setup  ////////////
/////////////////////////////////////////////////////////////////////////////


    problem.phases(1).nstates    = 10;
    problem.phases(1).ncontrols  = 4;
    problem.phases(1).nevents             = 20;
    problem.phases(1).npath      = 4;
    problem.phases(1).nodes               = (RowVectorXi(5) << 20, 30, 40, 50, 60).finished();   // <<  20, 30, 40, 50, 60;


    problem.phases(1).nparameters         = 1;


    psopt_level2_setup(problem, algorithm);

////////////////////////////////////////////////////////////////////////////
///////////////////  Enter problem bounds information //////////////////////
////////////////////////////////////////////////////////////////////////////

   // Control bounds

    problem.phases(1).bounds.lower.controls(0) =  -1.0;
    problem.phases(1).bounds.lower.controls(1) =  -1.0;
    problem.phases(1).bounds.lower.controls(2) =  -1.0;
    problem.phases(1).bounds.lower.controls(3) =  -1.0;


    problem.phases(1).bounds.upper.controls(0) =   1.0;
    problem.phases(1).bounds.upper.controls(1) =   1.0;
    problem.phases(1).bounds.upper.controls(2) =   1.0;
    problem.phases(1).bounds.upper.controls(3) =   1.0;

    // state bounds

    problem.phases(1).bounds.lower.states(0) =  -1.0;
    problem.phases(1).bounds.lower.states(1) =  -0.2;
    problem.phases(1).bounds.lower.states(2) =  -0.2;
    problem.phases(1).bounds.lower.states(3) =  -1.0;
    problem.phases(1).bounds.lower.states(4) =  -1.E-2;
    problem.phases(1).bounds.lower.states(5) =  -1.E-2;
    problem.phases(1).bounds.lower.states(6) =  -1.E-2;
    problem.phases(1).bounds.lower.states(7) =  -8000.0;
```

```
    problem.phases(1).bounds.lower.states(8) =  -8000.0;
    problem.phases(1).bounds.lower.states(9) =  -8000.0;

    problem.phases(1).bounds.upper.states(0) =  1.0;
    problem.phases(1).bounds.upper.states(1) =  0.2;
    problem.phases(1).bounds.upper.states(2) =  0.2;
    problem.phases(1).bounds.upper.states(3) =  1.0;
    problem.phases(1).bounds.upper.states(4) =  1.E-2;
    problem.phases(1).bounds.upper.states(5) =  1.E-2;
    problem.phases(1).bounds.upper.states(6) =  1.E-2;
    problem.phases(1).bounds.upper.states(7) =  8000.0;
    problem.phases(1).bounds.upper.states(8) =  8000.0;
    problem.phases(1).bounds.upper.states(9) =  8000.0;

   // Parameter bound


    problem.phases(1).bounds.lower.parameters(0) =  0.0;
    problem.phases(1).bounds.upper.parameters(0) =  hmax*hmax;


    // Event bounds

    // Initial / Final  condition values:
    MatrixXd q_i(4,1),  q_f(4,1), omega_i(3,1), omega_f(3,1), h_i(3,1), h_f(3,1);

    adouble q_ad[4], omega_ad[3];

// Initial conditions
    if (CASE==1) {
     q_i(0) =  0.98966;
     q_i(1) =  0.02690;
     q_i(2) = -0.08246;
     q_i(3) =  0.11425;

        q_ad[ 0 ]=q_i(0);
        q_ad[ 1 ]=q_i(1);
        q_ad[ 2 ]=q_i(2);
        q_ad[ 3 ]=q_i(3);
        compute_omega0( omega_ad, q_ad);
        omega_i(0) = omega_ad[ 0 ].value();
        omega_i(1) = omega_ad[ 1 ].value();
        omega_i(2) = omega_ad[ 2 ].value();

 //    omega_i(1) = -2.5410E-4;
 //    omega_i(2) = -1.1145E-3;
 //    omega_i(3) =  8.2609E-5;

     h_i(0) =  -496.0;
     h_i(1) =  -175.0;
     h_i(2) = -3892.0;
    }

    else if (CASE==2) {
   q_i(0) =  0.98996;
     q_i(1) =  0.02650;
     q_i(2) = -0.07891;
     q_i(3) =  0.11422;
        q_ad[ 0 ]=q_i(0);
        q_ad[ 1 ]=q_i(1);
        q_ad[ 2 ]=q_i(2);
        q_ad[ 3 ]=q_i(3);
        compute_omega0( omega_ad, q_ad);
        omega_i(0) = omega_ad[ 0 ].value();
        omega_i(1) = omega_ad[ 1 ].value();
        omega_i(2) = omega_ad[ 2 ].value();

     h_i(0) =  1000.0;
     h_i(1) =  -500.0;
     h_i(2) = -4200.0;
    }

    // Final conditions
    q_f(0) =  0.70531;
    q_f(1) = -0.06201;
    q_f(2) = -0.03518;
    q_f(3) = -0.70531;

    q_ad[ 0 ]=q_f(0);
    q_ad[ 1 ]=q_f(1);
    q_ad[ 2 ]=q_f(2);
    q_ad[ 3 ]=q_f(3);
```

```
    compute_omega0( omega_ad, q_ad);
    omega_f(0) = omega_ad[ 0 ].value();
    omega_f(1) = omega_ad[ 1 ].value();
    omega_f(2) = omega_ad[ 2 ].value();

//    omega_f(1) = 1.1353E-3;
//    omega_f(2) = 3.0062E-6;
//    omega_f(3) = -1.5713E-4;

    h_f(0) = -9.0;
    h_f(1) = -3557.0;
    h_f(2) =  -135.0;

    double DQ = 0.0001;
    double DWF = 0.0;
    double DHF = 0.0;

    problem.phases(1).bounds.lower.events(0)  = q_i(0)-DQ;
    problem.phases(1).bounds.lower.events(1)  = q_i(1)-DQ;
    problem.phases(1).bounds.lower.events(2)  = q_i(2)-DQ;
    problem.phases(1).bounds.lower.events(3)  = q_i(3)-DQ;
    problem.phases(1).bounds.lower.events(4)  = omega_i(0);
    problem.phases(1).bounds.lower.events(5)  = omega_i(1);
    problem.phases(1).bounds.lower.events(6)  = omega_i(2);
    problem.phases(1).bounds.lower.events(7)  = h_i(0);
    problem.phases(1).bounds.lower.events(8)  = h_i(1);
    problem.phases(1).bounds.lower.events(9)  = h_i(2);
    problem.phases(1).bounds.lower.events(10) = q_f(0)-DQ;
    problem.phases(1).bounds.lower.events(11) = q_f(1)-DQ;
    problem.phases(1).bounds.lower.events(12) = q_f(2)-DQ;
    problem.phases(1).bounds.lower.events(13) = q_f(3)-DQ;
    problem.phases(1).bounds.lower.events(14) = omega_f(0)-DWF;
    problem.phases(1).bounds.lower.events(15) = omega_f(1)-DWF;
    problem.phases(1).bounds.lower.events(16) = omega_f(2)-DWF;
    problem.phases(1).bounds.lower.events(17) = h_f(0)-DHF;
    problem.phases(1).bounds.lower.events(18) = h_f(1)-DHF;
    problem.phases(1).bounds.lower.events(19) = h_f(2)-DHF;


    problem.phases(1).bounds.upper.events(0)  = q_i(0)+DQ;
    problem.phases(1).bounds.upper.events(1)  = q_i(1)+DQ;
    problem.phases(1).bounds.upper.events(2)  = q_i(2)+DQ;
    problem.phases(1).bounds.upper.events(3)  = q_i(3)+DQ;
    problem.phases(1).bounds.upper.events(4)  = omega_i(0);
    problem.phases(1).bounds.upper.events(5)  = omega_i(1);
    problem.phases(1).bounds.upper.events(6)  = omega_i(2);
    problem.phases(1).bounds.upper.events(7)  = h_i(0);
    problem.phases(1).bounds.upper.events(8)  = h_i(1);
    problem.phases(1).bounds.upper.events(9)  = h_i(2);
    problem.phases(1).bounds.upper.events(10) = q_f(0)+DQ;
    problem.phases(1).bounds.upper.events(11) = q_f(1)+DQ;
    problem.phases(1).bounds.upper.events(12) = q_f(2)+DQ;
    problem.phases(1).bounds.upper.events(13) = q_f(3)+DQ;
    problem.phases(1).bounds.upper.events(14) = omega_f(0)+DWF;
    problem.phases(1).bounds.upper.events(15) = omega_f(1)+DWF;
    problem.phases(1).bounds.upper.events(16) = omega_f(2)+DWF;
    problem.phases(1).bounds.upper.events(17) = h_f(0)+DHF;
    problem.phases(1).bounds.upper.events(18) = h_f(1)+DHF;
    problem.phases(1).bounds.upper.events(19) = h_f(2)+DHF;

// Path bounds

    double hdotmax = 200.0; // [ ft-lbf ]

    double EQ_TOL = 0.0002;

    problem.phases(1).bounds.lower.path(0) = 1.0-EQ_TOL;
    problem.phases(1).bounds.upper.path(0) = 1.0+EQ_TOL;

    problem.phases(1).bounds.lower.path(1) = 1.0-EQ_TOL;
    problem.phases(1).bounds.upper.path(1) = 1.0+EQ_TOL;


    problem.phases(1).bounds.lower.path(2) = -hmax*hmax;
    problem.phases(1).bounds.upper.path(2) =  0.0;


    problem.phases(1).bounds.lower.path(3) = 0.0;
    problem.phases(1).bounds.upper.path(3) = hdotmax*hdotmax;

    // Time bounds
```

```
    double TFINAL;

    if (CASE==1) {
TFINAL = 6000.0;
    }
    else {
TFINAL = 7200.0;
    }

    problem.phases(1).bounds.lower.StartTime    = 0.0;
    problem.phases(1).bounds.upper.StartTime    = 0.0;

    problem.phases(1).bounds.lower.EndTime      = TFINAL;
    problem.phases(1).bounds.upper.EndTime      = TFINAL;

////////////////////////////////////////////////////////////////////////////
///////////////////  Register problem functions  ///////////////////////////
////////////////////////////////////////////////////////////////////////////


    problem.integrand_cost      = &integrand_cost;
    problem.endpoint_cost       = &endpoint_cost;
    problem.dae                = &dae;
    problem.events          = &events;
    problem.linkages         = &linkages;

////////////////////////////////////////////////////////////////////////////
///////////////////  Define & register initial guess ///////////////////////
////////////////////////////////////////////////////////////////////////////

   MatrixXd time_guess;      time_guess      = linspace(0.0, TFINAL, 50 );
   MatrixXd state_guess;     state_guess     = zeros(10,50);
   MatrixXd control_guess;   control_guess   = zeros(4,50);
   MatrixXd parameter_guess; parameter_guess = hmax*hmax*ones(1,1);

   control_guess.row(0) = linspace( q_i(0), q_i(0), 50 );
   control_guess.row(1) = linspace( q_i(1), q_i(1), 50 );
   control_guess.row(2) = linspace( q_i(2), q_i(2), 50 );
   control_guess.row(3) = linspace( q_i(3), q_i(3), 50 );

   state_guess.row(0) = linspace( q_i(0), q_i(0), 50);
   state_guess.row(1) = linspace( q_i(1), q_i(1), 50);
   state_guess.row(2) = linspace( q_i(2), q_i(2), 50);
   state_guess.row(3) = linspace( q_i(3), q_i(3), 50);

   state_guess.row(4) = linspace( omega_i(0), omega_f(0), 50);
   state_guess.row(5) = linspace( omega_i(1), omega_f(1), 50);
   state_guess.row(6) = linspace( omega_i(2), omega_f(2), 50);

   state_guess.row(7) = linspace( h_i(0), h_f(0), 50);
   state_guess.row(8) = linspace( h_i(1), h_f(1), 50);
   state_guess.row(9) = linspace( h_i(2), h_f(2), 50);



   problem.phases(1).guess.controls    = control_guess;
   problem.phases(1).guess.states      = state_guess;
   problem.phases(1).guess.time        = time_guess;
   problem.phases(1).guess.parameters  = parameter_guess;



////////////////////////////////////////////////////////////////////////////
///////////////////  Enter algorithm options  //////////////////////////////
////////////////////////////////////////////////////////////////////////////


    algorithm.nlp_iter_max                = 1000;
    algorithm.nlp_tolerance               = 1.e-5;
    algorithm.nlp_method                  = "IPOPT";
    algorithm.scaling                     = "automatic";
    algorithm.derivatives                 = "automatic";
    algorithm.defect_scaling              = "jacobian-based";
    algorithm.jac_sparsity_ratio          = 0.104;




////////////////////////////////////////////////////////////////////////////
///////////////////  Now call PSOPT to solve the problem   /////////////////
////////////////////////////////////////////////////////////////////////////
```

```
    psopt(solution, problem, algorithm);

////////////////////////////////////////////////////////////////////////////
///////////  Extract relevant variables from solution structure   //////////
////////////////////////////////////////////////////////////////////////////


    MatrixXd states, controls, t;

    states      = solution.get_states_in_phase(1);
    controls    = solution.get_controls_in_phase(1);
    t           = solution.get_time_in_phase(1);


////////////////////////////////////////////////////////////////////////////
///////////  Save solution data to files if desired ////////////////////////
////////////////////////////////////////////////////////////////////////////

    Save(states,"states.dat");
    Save(controls,"controls.dat");
    Save(t,"t.dat");

    MatrixXd omega, h, q, phi, theta, psi, qc, euler_angles;

    q      = states.block(0,0,4,length(t));
    omega  = states.block(4,0,3,length(t));
    h      = states.block(7,0,3,length(t));
    qc     = controls;

    quarternion2Euler(phi, theta, psi,  q);


    euler_angles.resize(3,length(t));

    euler_angles << phi ,
                    theta ,
                    psi;

    adouble  qc_ad[4], u_ad[3];

    MatrixXd u(3,length(t));
    MatrixXd hnorm(1,length(t));
    MatrixXd hi;
    MatrixXd hm = hmax*ones(1,length(t));


    int i,j;

  for (i=0; i< length(t); i++ ) {
for(j=0;j<3;j++) {
   omega_ad[j] = omega(j,i);
        }
for(j=0;j<4;j++) {
   q_ad[j] = q(j,i);
           qc_ad[j]= qc(j,i);
        }

compute_control_torque(u_ad, q_ad, qc_ad, omega_ad );

for(j=0;j<3;j++) {
   u(j,i) = u_ad[j].value();
   }

   hi = h.col(i);

hnorm(0,i) = hi.norm();

    }

    omega = omega*(180.0/pi)*1000; // convert to mdeg/s

    phi = phi*180.0/pi; theta=theta*180.0/pi; psi=psi*180.0/pi;


    Save(u,"u.dat");

    Save(euler_angles,"euler_angles.dat");



///////////////////////////////////////////////////////////////////////////
///////////  Plot some results if desired (requires gnuplot) //////////////
```

```
////////////////////////////////////////////////////////////////////////////


    plot(t,q,problem.name+" quarternion elements: q",  "time (s)", "q", "q");

    plot(t,qc,problem.name+" Control variables: qc",  "time (s)", "qc", "qc");

    plot(t,phi,problem.name+" Euler angles: phi",     "time (s)", "angles (deg)", "phi");

    plot(t,theta,problem.name+" Euler angles: theta", "time (s)", "angles (deg)", "theta");

    plot(t,psi,problem.name+" Euler angle: psi",       "time (s)", "psi (deg)", "psi");

    plot(t,omega.row(0),problem.name+": omega 1","time (s)", "omega1", "omega1");

    plot(t,omega.row(1),problem.name+": omega 2","time (s)", "omega2", "omega2");

    plot(t,omega.row(2),problem.name+": omega 3","time (s)", "omega3", "omega3");

    plot(t,h.row(0),problem.name+": momentum 1","time (s)", "h1", "h1");

    plot(t,h.row(1),problem.name+": momentum 2","time (s)", "h2", "h2");

    plot(t,h.row(2),problem.name+": momentum 3","time (s)", "h3", "h3");

    plot(t,u.row(0),problem.name+": control torque 1","time (s)", "u1", "u1");

    plot(t,u.row(1),problem.name+": control torque 2","time (s)", "u2", "u2");

    plot(t,u.row(2),problem.name+": control torque 3","time (s)", "u3", "u3");

    plot(t,hnorm,t,hm,problem.name+": momentum norm", "time (s)", "h", "h hmax");


    plot(t,phi,problem.name+" Euler angles: phi",     "time (s)", "angles (deg)", "phi",
       "pdf", "zpm_phi.pdf" );

    plot(t,theta,problem.name+" Euler angles: theta", "time (s)", "angles (deg)", "theta",
       "pdf", "zpm_theta.pdf");

    plot(t,psi,problem.name+" Euler angle: psi",       "time (s)", "psi (deg)", "psi",
       "pdf", "zpm_psi.pdf");

    plot(t,omega.row(0),problem.name+": omega 1","time (s)", "omega1", "omega1",
       "pdf", "zpm_omega1.pdf");

    plot(t,omega.row(1),problem.name+": omega 2","time (s)", "omega2", "omega2",
       "pdf", "zpm_omega2.pdf");

    plot(t,omega.row(2),problem.name+": omega 3","time (s)", "omega3", "omega3",
       "pdf", "zpm_omega3.pdf");

    plot(t,h.row(0),problem.name+": momentum 1","time (s)", "h1", "h1",
              "pdf", "zpm_h1.pdf");

    plot(t,h.row(1),problem.name+": momentum 2","time (s)", "h2", "h2",
      "pdf", "zpm_h2.pdf");

    plot(t,h.row(2),problem.name+": momentum 3","time (s)", "h3", "h3",
      "pdf", "zpm_h3.pdf");

    plot(t,u.row(0),problem.name+": control torque 1","time (s)", "u1", "u1",
      "pdf", "zpm_u1.pdf");

    plot(t,u.row(1),problem.name+": control torque 2","time (s)", "u2", "u2",
      "pdf", "zpm_u2.pdf");

    plot(t,u.row(2),problem.name+": control torque 3","time (s)", "u3", "u3",
      "pdf", "zpm_u3.pdf");

    plot(t,hnorm,t,hm,problem.name+": momentum norm", "time (s)", "h (ft-lbf-sec)", "h hmax",
      "pdf", "zpm_hnorm.pdf");

    plot(t,u,problem.name+": control torques","time (s)", "u (ft-lbf)", "u1 u2 u3",
      "pdf", "zpm_controls.pdf");

}

////////////////////////////////////////////////////////////////////////////
///////////////////////      END OF FILE     ///////////////////////////////
////////////////////////////////////////////////////////////////////////////
```

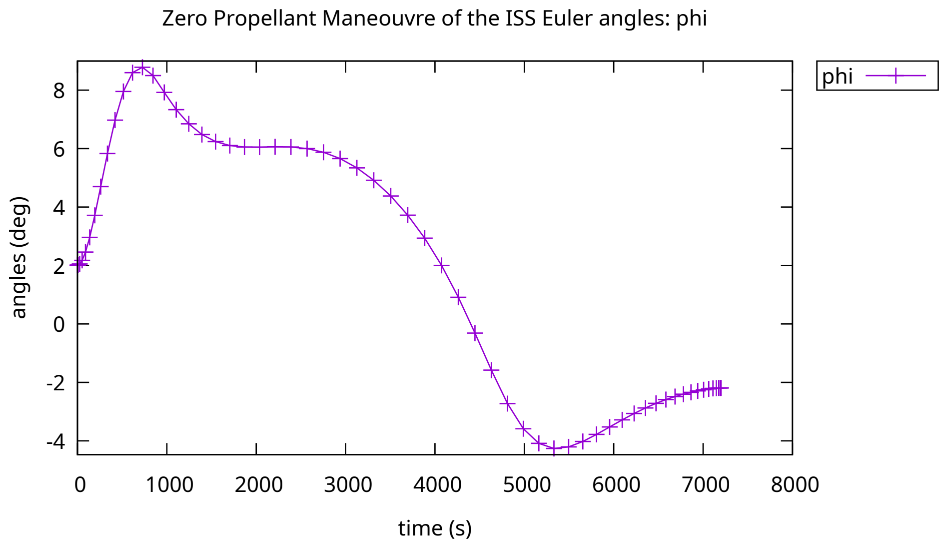

Figure 115: Euler angle ϕ (roll)

The output from $\mathcal{PSOPT}$ is summarised in the box below and shown in Figures 115 to 127..

```
PSOPT results summary
=====================

Problem:  Zero Propellant Maneouvre of the ISS
CPU time (seconds): 2.926508e+01
NLP solver used:  IPOPT
PSOPT release number:  5.0.3
Date and time of this run:  Thu Mar  6 17:06:00 2025

Optimal (unscaled) cost function value:  6.680107e+06
Phase 1 endpoint cost function value: 5.240045e+06
Phase 1 integrated part of the cost:  1.440062e+06
Phase 1 initial time: 0.000000e+00
Phase 1 final time: 7.200000e+03
Phase 1 maximum relative local error: 1.012118e-03
NLP solver reports:  The problem has been solved!
```

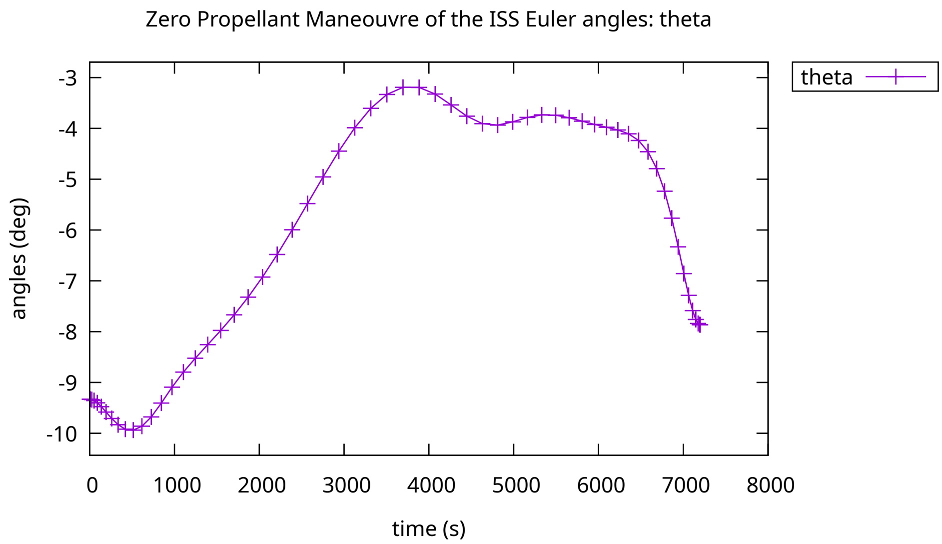

Figure 116: Euler angle θ (pitch)

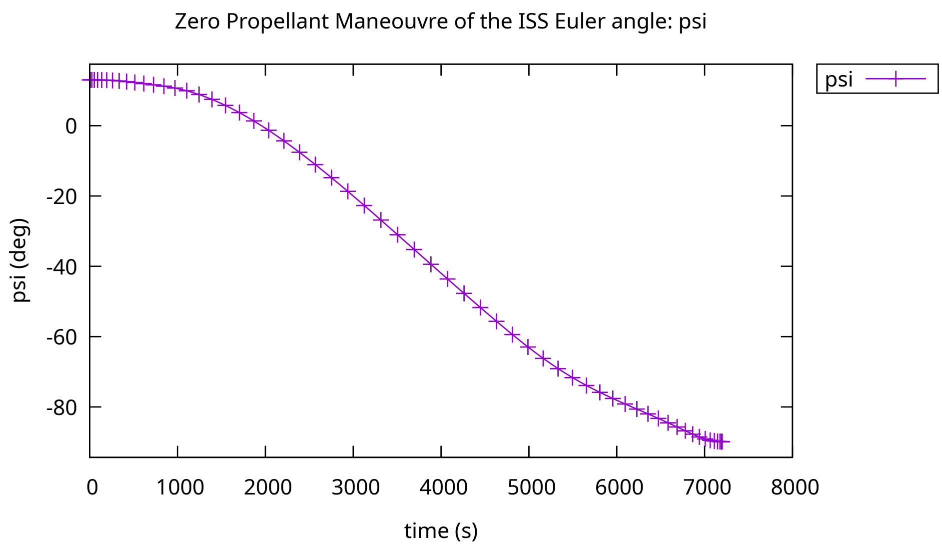

Figure 117: Euler angle ψ (yaw)

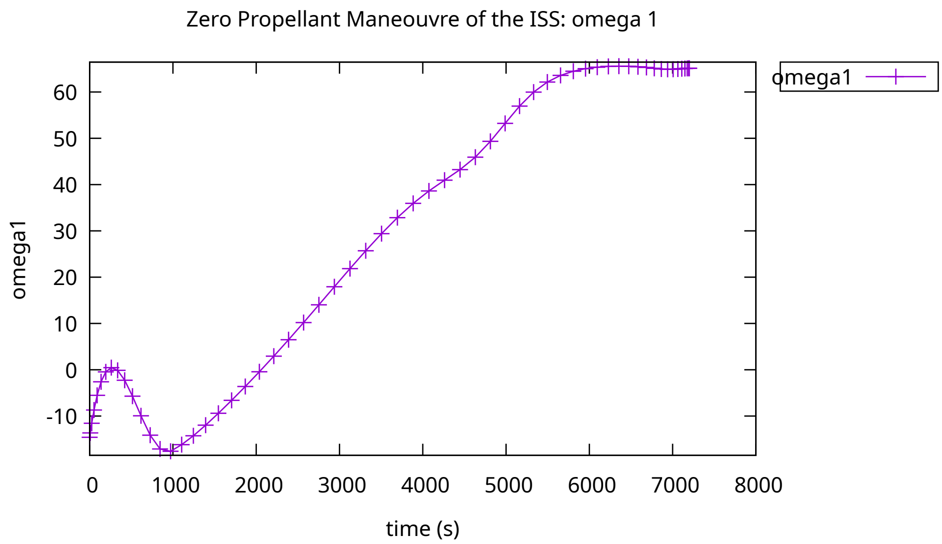

Figure 118: Angular speed ω_1 (roll)

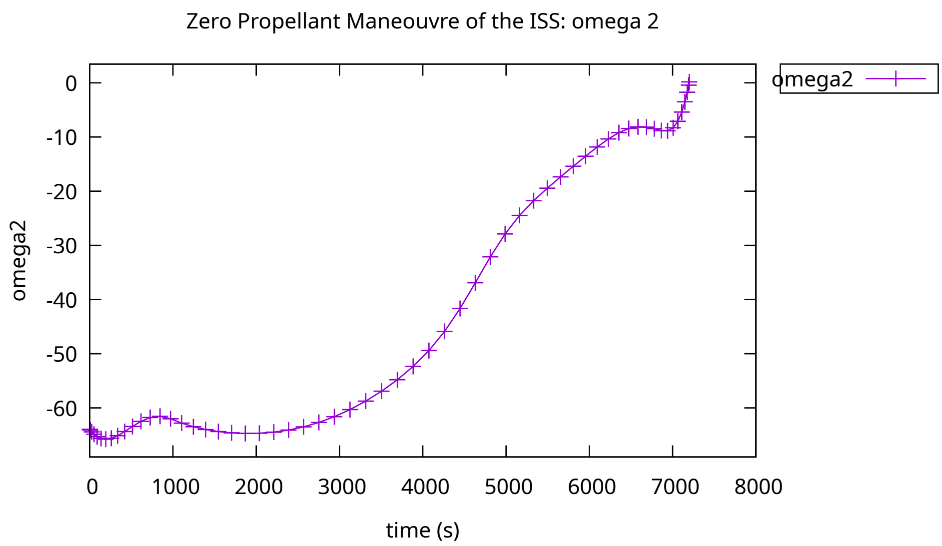

Figure 119: Angular speed ω_2 (pitch)

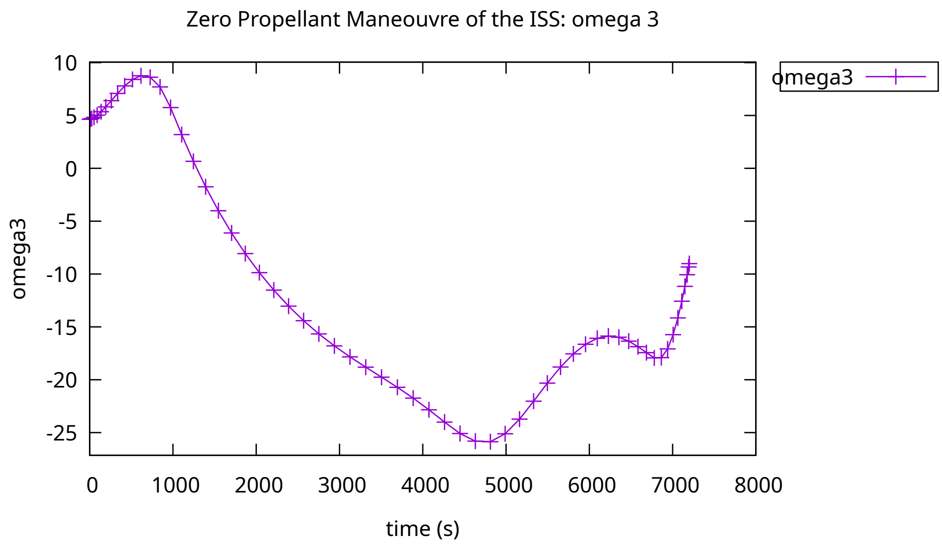

Figure 120: Angular speed ω_3 (yaw)

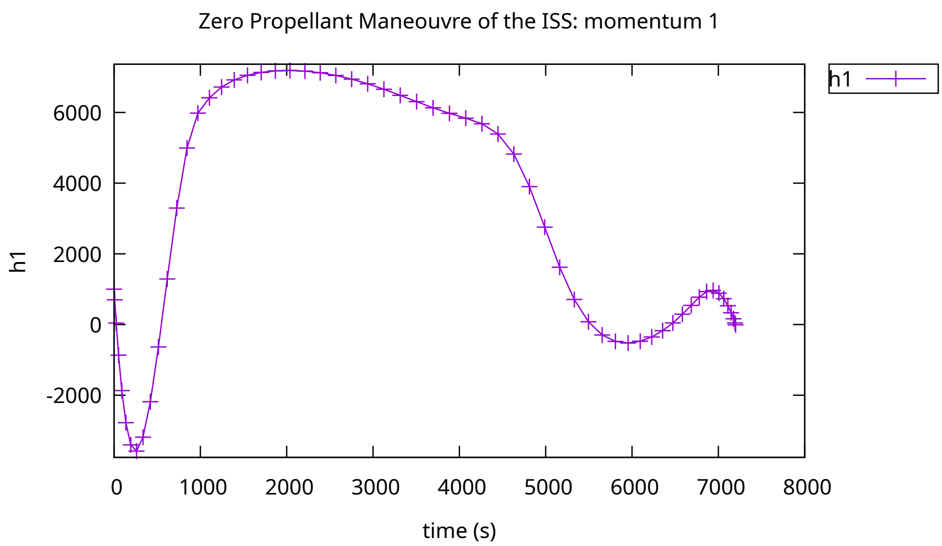

Figure 121: Momentum h_1 (roll)

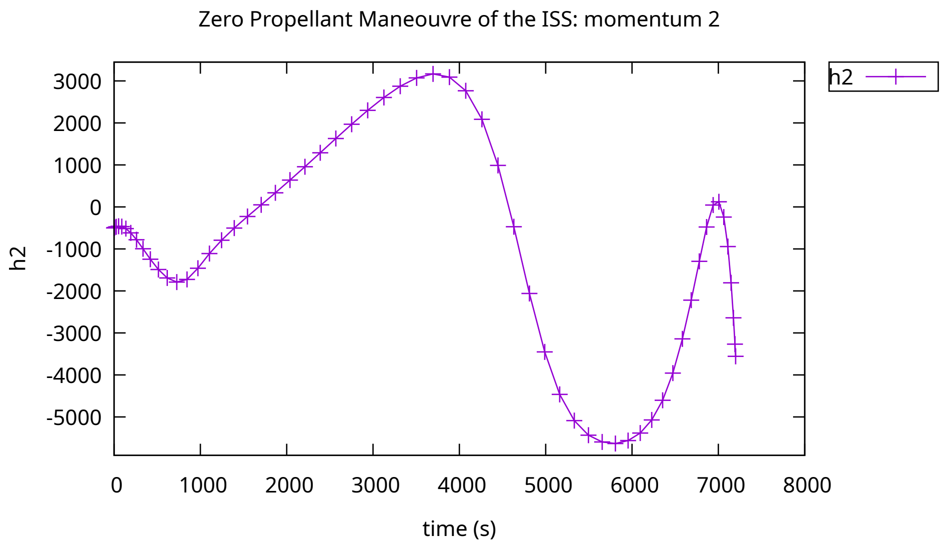

Figure 122: Momentum h_2 (pitch)

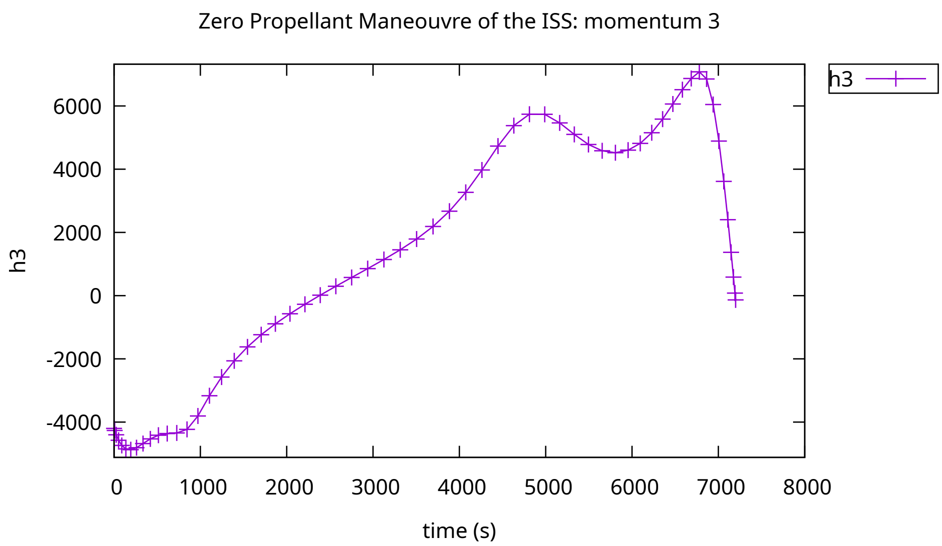

Figure 123: Momentum h_3 (yaw)

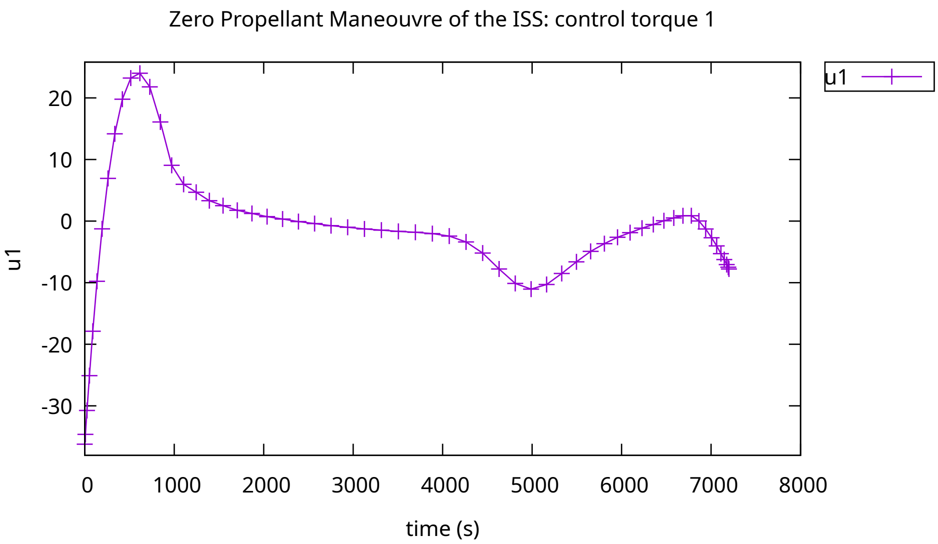

Figure 124: Control torque u_1 (roll)

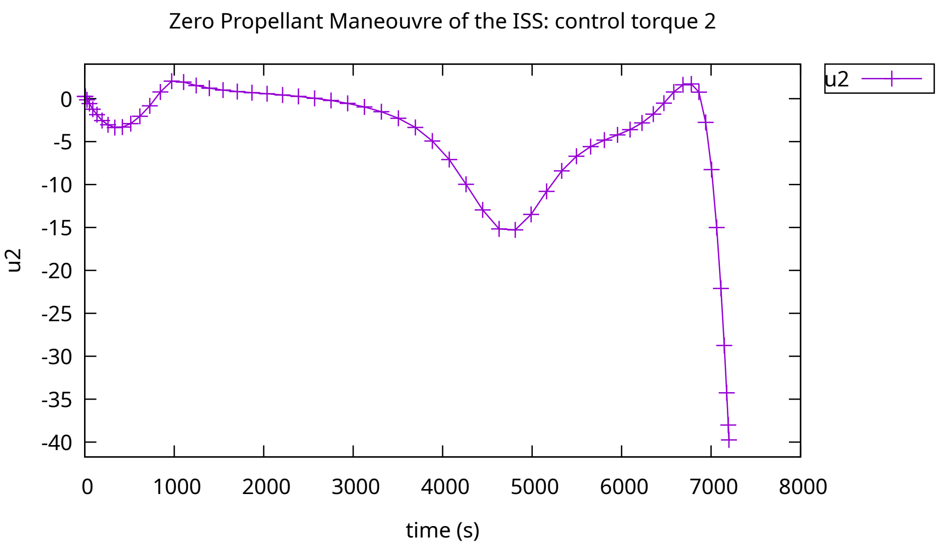

Figure 125: Control torque u_2 (pitch)

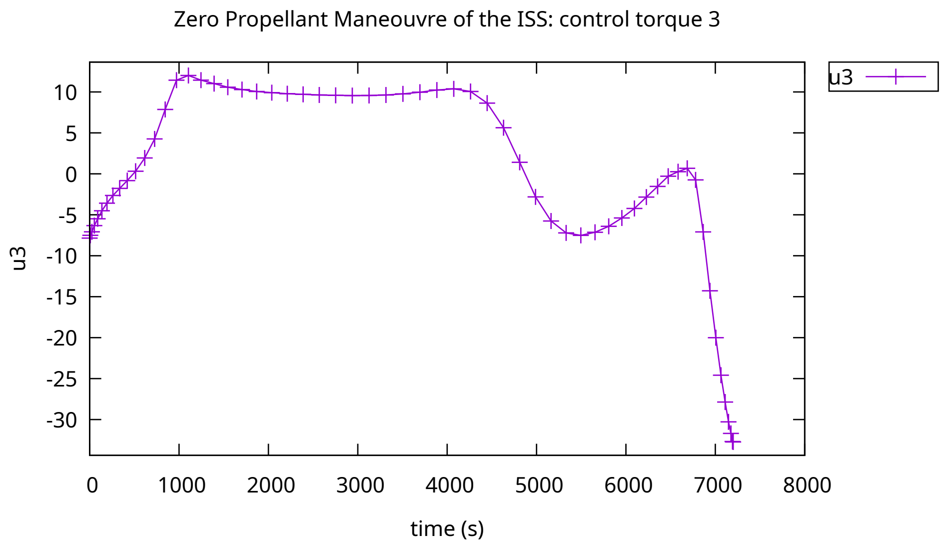

Figure 126: Control torque u_3 (yaw)

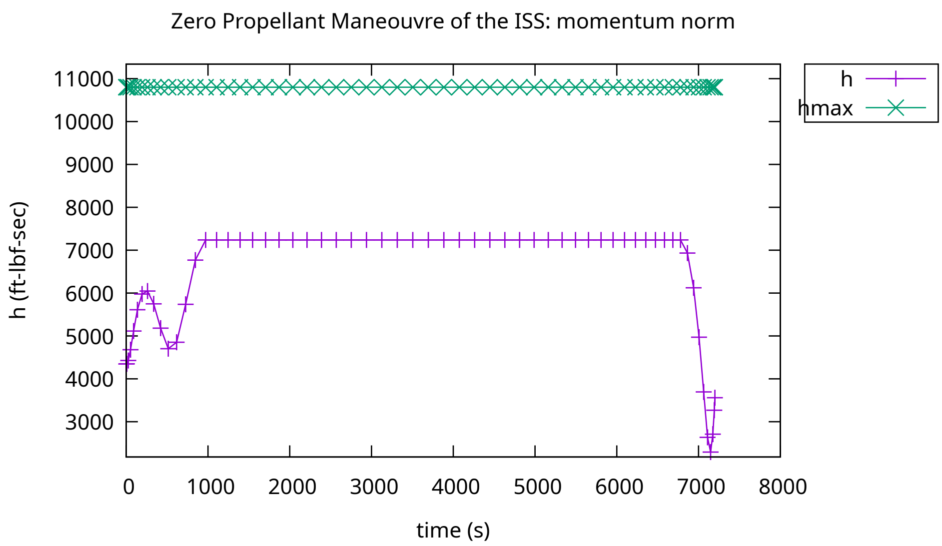

Figure 127: Momentum norm $||\mathbf{h}(t)||$

References

[1] N.S. Bedrossian, S. Bhatt, W. Kang, and I.M. Ross. Zero Propellant Maneuver Guidance. *IEEE Control Systems Magazine*, 29:53–73, 2009.

[2] D. A. Benson. *A Gauss Pseudospectral Transcription for Optimal Control*. PhD thesis, MIT, Department of Aeronautics and Astronautics, Cambridge, Mass., 2004.

[3] J. T. Betts. *Practical Methods for Optimal Control Using Nonlinear Programming*. SIAM, 2001.

[4] J. T. Betts. *Practical Methods for Optimal Control and Estimation Using Nonlinear Programming*. SIAM, 2010.

[5] S.A. Bhatt. Optimal reorientation of spacecraft using only control moment gyroscopes. Master's thesis, Rice University, Houston, Texas, 2007.

[6] A.E. Bryson. *Dynamic Optimization*. Addison-Wesley, 1999.

[7] A.E. Bryson, M.N. Desai, and W.C. Hoffman. Energy-State Approximation in Performance Optimization of Supersonic Aircraft. *Journal of Aircraft*, 6:481–488, 1969.

[8] A.E. Bryson and Yu-Chi Ho. *Applied Optimal Control*. Halsted Press, 1975.

[9] E. D. Dolan and J. J. More. *Benchmarking optimization software with COPS 3.0*. Argonne National Laboratory, 9700 South Cass Avenue, Argonne, Illinois 60439, 2004.

[10] J. Franke and M. Otter. The manutec r3 benchmark models for the dynamic simulation of robots. Technical report, Institute for Robotics and System Dynamics, DLR Oberpfaffenhofen, 1993.

[11] Q. Gong, F. Fahroo, and I.M. Ross. Spectral algorithms for pseudospectral methods in optimal control. *Journal of Guidance Control and Dynamics*, 31:460–471, 2008.

[12] L.S. Jennings, M.E. Fisher, K.L. Teo, and C.J. Goh. *MISER3 Optimal Control Software Version 2.0 Theory and User Manual*. Department of Mathematics, The University of Western Australia, 2002.

[13] Z. Li, M. R. Osborne, and T. Prvan. Parameter estimation of ordinary differential equations. *IMA Journal of Numerical Analysis*, 25:264–285, 2005.

[14] R. Luus. *Iterative Dynamic Programming*. Chapman and Hall / CRC, 2002.

[15] M. Otter and S. Turk. The dfvrl models 1 and 2 of the manutec r3 robot. Technical report, Institute for Robotics and System Dynamics, DLR Oberpfaffenhofen, Germany, 1987.

[16] A.V. Rao, D. Benson, G. Huntington, and C. Francolin. *User's manual for GPOPS version 1.3: A Matlab package for dynamic optimization using the Gauss pseudospectral method*. 2008.

[17] A.V. Rao and K.D. Mease. Eigenvector Approximate Dichotomic Basis Method for Solving Hyper- sensitive Optimal Control Problems. *Optimal Control Applications and Methods*, 21:1–19, 2000.

[18] P. E. Rutquist and M. M. Edvall. *PROPT Matlab Optimal Control Software.* TOMLAB Optimization, 2009.

[19] K. Schittkowski. *Numerical Data Fitting in Dynamical Systems.* Kluwer Academic Publishers, Dordrecht, The Netherlands, 2002.

[20] O. Von Stryk. *User's guide for DIRCOL (Version 2.1): A direct collocation method for the numerical solution of optimal control problems.* Technical Report, Technische Universitat Munchen, 1999.

[21] S. Subchan and R. Zbikowski. *Computational optimal control: tools and practice.* Wiley, 2009.

[22] K.L. Teo, C.J. Goh, and K.H. Wong. *A Unified Computational Approach to Optimal Control Problems.* Wiley, New York, 1991.

[23] D.A. Vallado. *Fundamentals of Astrodynamics and Applications.* Kluwer Academic Publishers, Dordrecht, The Netherlands, 2001.