Summer Course

Numerical Methods for Embedded Optimization and Optimal Control

— Exercises —

Moritz Diehl, Daniel Axehill and Lars Eriksson

Linköping University

June 2011

Introduction

This collection of exercises is intended to give a quick overview of some important concepts in numerical optimal control and is a part of the summer course Numerical Methods for Embedded Optimization and Optimal Control at Linköping University 2011. It is divided into seven parts, Part A – Part G. The main purpose is to show how optimal control problems can be solved numerically and illustrate some fundamental ideas that can be employed for that purpose.

Robin Vujanic and Alexander Domahidi at ETH Zürich are acknowledged for help when creating the exercises when the course was given at ETH Zürich during spring 2011.

A Nonlinear Programming and Single Shooting

The first part has as its aim the formulation and numerical solution of a simple nonlinear programming problem followed by the solution of a simple optimal control problem. The solution algorithm is provided by MATLAB.

1. Log in and start MATLAB and open an editor of your choice. Use `help fmincon` to learn what is the syntax of a call of `fmincon`.

2. Now consider, first just on paper, the following nonlinear programming problem (NLP):

$$\min_{x\in\mathbb{R}^2} x_2 \quad \text{subject to} \quad \begin{cases} x_1^2 + 4x_2^2 & \leq & 4 \\ x_1 & \geq & -2 \\ x_1 & = & 1 \end{cases}$$

   (a) How many degrees of freedom, how many equality, and how many inequality constraints does this problem have?

   (b) Sketch the feasible set Ω of this problem. What is the optimal solution?

   (c) Bring this problem into the NLP standard form

$$\min_{x\in\mathbb{R}^n} f(x) \quad \text{subject to} \quad \begin{cases} g(x) & = & 0 \\ h(x) & \geq & 0 \end{cases}$$

   by defining the dimension n and the functions f, g, h along with their dimensions appropriately.

3. Now formulate three MATLAB functions f, g, h for the above NLP, choose an initial guess for x, and solve the problem using `fmincon`. Check that the output corresponds to what you expected.

4. Now we want to solve the first optimal control problem in this course. The aim is to bring a harmonic oscillator to rest with minimal control effort. For this aim we apply Euler integration to the continuous time model and get the following linear discrete time dynamic system:

$$\begin{bmatrix} p_{k+1} \\ v_{k+1} \end{bmatrix} = \begin{bmatrix} p_k \\ v_k \end{bmatrix} + \Delta t \left(\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \begin{bmatrix} p_k \\ v_k \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u_k \right), \quad k = 1, \dots, N-1 \tag{1}$$

   Choose the fixed values $p_1 = 10, v_1 = 0, \Delta t = 0.2, N = 51$, and denote for simplicity from now on $x_k = (p_k, v_k)^T$. Now write a MATLAB routine `[xN]=oscisim(U)` that computes x_N as a function of the control inputs $U = (u_1, \dots, u_{N-1})^T$. Mathematically, we will denote this function by $f_{\text{oscisim}} : \mathbb{R}^{N-1} \to \mathbb{R}^2$.

5. To verify that your routine does what you want, plot the simulated positions $p_1, \ldots, p_N$ within this routine for the input $U = 0$.

6. Now we want to solve the optimal control problem

$$\min_{U \in \mathbb{R}^{N-1}} \|U\|_2^2 \quad \text{subject to} \qquad f_{\text{oscisim}}(U) = 0$$

   Formulate and solve this problem with `fmincon`. Plot the solution vector U as well as the trajectory of the positions in the solution.

7. Now add inequalities to the problem, limiting the inputs u_k in amplitude by an upper bound $|u_k| \leq u_{\max}, k = 1, \ldots, N-1$. This adds $2(N-1)$ inequalities to your problem. Which?

8. Formulate the problem with inequalities in `fmincon`. Experiment with different values of $u_{\max}$, starting with big ones and making it smaller. If it is very big, the solution will not be changed at all. At which critical value of $u_{\max}$ does the solution start to change? If it is too small, the problem will become infeasible. At which critical value of $u_{\max}$ does this happen?

9. Both of the above problems are convex, i.e., each local minimum is also a global minimum. Note that the equality constraint of the optimal control problems is just a linear function at the moment. Now, try to make this constraint nonlinear and thus make the problem nonconvex. One way is to add a small nonlinearity into the dynamic system (1) by making the spring nonlinear, i.e., replacing the term -1 in the lower left corner of the system matrix by $-(1 + \mu p_k^2)$ with a small μ, and solving the problem again. At which value of μ does the solver `fmincon` need twice as many iterations as before?

B Runge-Kutta Integrator and Gauss-Newton Algorithm

In this part we compare two methods for numerical integration of ordinary differential equations (ODE) and program our first own optimization code, a Gauss-Newton algorithm.

1. ODE Simulation: throughout this part, we regard again the controlled harmonic oscillator from Task A.4 described by

$$\frac{d}{dt}\begin{bmatrix} p(t) \\ v(t) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \begin{bmatrix} p(t) \\ v(t) \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u(t), \quad t \in [0, T]. \tag{2}$$

   We abbreviate this ODE as $\dot{x} = f(x, u)$ with $x = (p, v)^T$. Let us choose again the fixed initial value $x_0 = (10, 0)^T$ and $T = 10$.

   In the next questions we are interested in comparing the simulation results for $u(t) = 0$ that are obtained by two different integration schemes, namely the Euler integrator from Task A.4, and a Runge-Kutta integrator. We regard in particular the value $p(10)$, and as the ODE is explicitly solvable, we know it exactly, which is useful for comparisons. What is the analytical expression for $p(10)$?

2. First run again your explicit Euler method from Task A.4, e.g., again with $N = 50$ integrator steps, i.e. with a stepsize of $\Delta t = 10/50 = 0.2$. The central line in the Euler code reads

$$x_{k+1} \quad = \quad x_k + \Delta t \cdot f(x_k, u_k) \tag{3}$$

   Plot your trajectories $\{(t_k, x_k)\}_1^{N+1}$ for $u_k = 0$.

3. Now exchange in your Euler simulation code the line that generates the step (3) by the following five lines:

$$\begin{aligned} k_1 &= f(x_k, u_k) & (4)\\ k_2 &= f(x_k + \frac{1}{2}\Delta t \cdot k_1, u_k) & (5)\\ k_3 &= f(x_k + \frac{1}{2}\Delta t \cdot k_2, u_k) & (6)\\ k_4 &= f(x_k + \Delta t \cdot k_3, u_k) & (7)\\ x_{k+1} &= x_k + \Delta t \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) & (8) \end{aligned}$$

This is the classical Runge Kutta method of order four (RK4). Note that each integrator step is four times as expensive as an Euler step. What is the advantage of this extra effort? To get an idea, plot your trajectories $\{(t_k, x_k)\}_1^{N+1}$ for the same number N of integrator steps.

4. To make the comparison of Euler and RK4 quantitative, regard the different approximations of $p(10)$ that you obtain for different stepsizes, e.g., $\Delta t = 10^{-k}$ with $k = 0, \dots, 5$. We call these approximations $\tilde{p}(10; \Delta t)$. Compute the errors $|p(10) - \tilde{p}(10; \Delta t)|$ and plot them doubly logarithmic, i.e., plot $\log_{10}(|p(10) - \tilde{p}(10; \Delta t)|)$ on the y-axis and $\log_{10} \Delta t$ on the x-axis, for each of the integrators. You should see a line for each integrator. Can you explain the different slopes?

5. We consider again the optimal control problem from Task A.4. We had previously used the Euler integrator, but here we use our new RK4 integrator because it is more accurate. We do again $N = 50$ integrator steps to obtain the terminal state as a function of the controls $u_0, \dots, u_{N-1}$, and denote the function here by $g_{\text{sim}} : \mathbb{R}^N \to \mathbb{R}^2$. The nonlinear program we want to solve is again

$$\min_{U \in \mathbb{R}^N} \|U\|_2^2 \quad \text{subject to} \quad g_{\text{sim}}(U) = 0 \tag{9}$$

In this task, we do not solve this problem with `fmincon`, but we write our first Newton-type optimization method. To prepare for the next step, make sure you have the routine g_{sim} as a function of the 50 controls with the two terminal states as outputs.

6. Motivating background information (you might skip to the next question): The necessary optimality conditions (KKT conditions) for the above problem are

$$\begin{aligned} 2U^* + \frac{\partial g_{\text{sim}}}{\partial U}(U^*)^T \lambda^* &= 0 & (10)\\ g_{\text{sim}}(U^*) &= 0. & (11) \end{aligned}$$

Let us introduce a shorthand for the Jacobian matrix

$$J_{\text{sim}}(U) := \frac{\partial g_{\text{sim}}}{\partial U}(U)$$

By linearization of the constraint at some given iterate (U_k, λ_k) and neglecting its second order derivatives, we get the following (Gauss-Newton) approximation of the KKT conditions:

$$\begin{bmatrix} 2U_k \\ g_{\text{sim}}(U_k) \end{bmatrix} + \begin{bmatrix} 2\mathbb{I} & J_{\text{sim}}(U_K)^T \\ J_{\text{sim}}(U_k) & 0 \end{bmatrix} \begin{bmatrix} U_{k+1} - U_k \\ \lambda_{k+1} \end{bmatrix} = 0$$

This system can easily be solved by a linear solve in order to obtain a new iterate U_{k+1}. But in order to do this, we first need to compute the Jacobian $J_{\text{sim}}(U)$.

7. Implement a routine that uses finite differences, i.e., calls the function g_{sim} $(N+1)$ times, once at the nominal value and then with each component slightly perturbed by, e.g., $\delta = 10^{-4}$ in the direction of each unit vector e_k, so that we get the approximations

$$\frac{\partial g_{\text{sim}}}{\partial u_k}(U) \approx \frac{g_{\text{sim}}(U + \delta e_k) - g_{\text{sim}}(U)}{\delta}.$$

We denote the resulting function that gives the full Jacobian matrix of g_{sim} by $J_{\text{sim}} : \mathbb{R}^N \rightarrow \mathbb{R}^{2\times N}$.

8. Now, we implement the Gauss-Newton scheme from above, but as we are not interested in the multipliers we just implement it as follows:

$$U_{k+1} = U_k - \begin{bmatrix} \mathbb{I} & 0 \end{bmatrix} \begin{bmatrix} 2\mathbb{I} & J_{\text{sim}}(U_k)^T \\ J_{\text{sim}}(U_k) & 0 \end{bmatrix}^{-1} \begin{bmatrix} 2U_k \\ g_{\text{sim}}(U_k) \end{bmatrix}$$

Choose an initial guess for the controls, e.g., $U = 0$, and start your iteration and stop when $\|U_{k+1} - U_k\|$ is very small. How many iterations do you need to converge? Do you have an idea why?

C Gauss-Newton vs. BFGS, and Reverse Differentiation

In this part we use again our Runge-Kutta simulator within a sequential approach. We make our model a bit nonlinear. Second, we compare the Gauss-Newton algorithm with a BFGS algorithm. Third we write a new routine for the Jacobian calculation based on the reverse mode of algorithmic differentiation.

1. Throughout this part, we make our controlled oscillator slightly nonlinear by making it a pendulum and setting

$$\frac{d}{dt}\begin{bmatrix} p(t) \\ v(t) \end{bmatrix} = \begin{bmatrix} v(t) \\ -C\sin(p(t)/C) \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u(t), \quad t \in [0,T]. \tag{12}$$

with $C := 180/\pi/4$. We again abbreviate the ODE as $\dot{x} = f(x,u)$ with $x = (p,v)^T$, and choose again the fixed initial value $x_0 = (10,0)^T$ and $T = 10$. Note that p now measures the deviation from the equilibrium state in multiples of 4 degrees (i.e., we start with 40 degrees).

We consider again the optimal control problem from the previous parts

$$\min_{U\in\mathbb{R}^N} \|U\|_2^2 \quad \text{subject to} \qquad g_{\text{sim}}(U) = 0 \tag{13}$$

and we use again RK4 and do again $N = 50$ integrator steps to obtain the terminal state x_N as a function of the controls $u_0, \dots, u_{N-1}$.

Run again your Gauss-Newton scheme from the previous part, i.e., use in each iteration finite differences to compute the Jacobian matrix

$$J_{\text{sim}}(U) := \frac{\partial g_{\text{sim}}}{\partial U}(U)$$

and iterate

$$U_{k+1} = U_k - \begin{bmatrix} \mathbb{I} & 0 \end{bmatrix} \begin{bmatrix} 2\mathbb{I} & J_{\text{sim}}(U_k)^T \\ J_{\text{sim}}(U_k) & 0 \end{bmatrix}^{-1} \begin{bmatrix} 2U_k \\ g_{\text{sim}}(U_k) \end{bmatrix}$$

How many iterations do you need now, with the nonlinear oscillator? Plot the vector U_k and the resulting trajectory of p in each Gauss-Newton iteration so that you can observe the Gauss-Newton algorithm at work.

2. Modify your Gauss-Newton scheme so that you also obtain the multiplier vectors, i.e., iterate with $B_k = 2\mathbb{I}$ as follows:

$$\begin{bmatrix} U_{k+1} \\ \lambda_{k+1} \end{bmatrix} = \begin{bmatrix} U_k \\ 0 \end{bmatrix} - \begin{bmatrix} B_k & J_{\text{sim}}(U_k)^T \\ J_{\text{sim}}(U_k) & 0 \end{bmatrix}^{-1} \begin{bmatrix} 2U_k \\ g_{\text{sim}}(U_k) \end{bmatrix}$$

   Choose as your stopping criterion now that the norm of the residual

$$\text{KKTRES}_k := \left\| \begin{bmatrix} \nabla_U \mathbb{L}(U_k, \lambda_k) \\ g_{\text{sim}}(U_k) \end{bmatrix} \right\| = \left\| \begin{bmatrix} 2U_k + J_{\text{sim}}(U_k)^T \lambda_k \\ g_{\text{sim}}(U_k) \end{bmatrix} \right\|$$

   shall be smaller than a given tolerance, $\text{TOL} = 10^{-4}$. Store the values KKTRES_k and plot their logarithms against the iteration number k. What do you see?

3. BFGS method: now use a different Hessian approximation, namely the BFGS update, i.e., start with a unit Hessian, $B_0 = \mathbb{I}$ and then update the Hessian according to

$$B_{k+1} := B_k - \frac{B_k s_k s_k^T B_k}{s_k^T B_k s_k} + \frac{y_k y_k^T}{s_k^T y_k}.$$

   with $s_k := U_{k+1} - U_k$ and $y_k := \nabla_U \mathbb{L}(U_{k+1}, \lambda_{k+1}) - \nabla_U \mathbb{L}(U_k, \lambda_{k+1})$. Devise your BFGS algorithm so that you need to evaluate the expensive Jacobian $J_{\text{sim}}(U_k)$ only once per BFGS iteration. Tipp: remember the old Jacobian $J_{\text{sim}}(U_k)$, then evaluate the new one $J_{\text{sim}}(U_{k+1})$, and only then compute B_{k+1}.

4. Observe the BFGS iterations and consider the logarithmic plot of KKTRES_k. How many iterations do you need now? Can you explain the form of the plot? What happens if you make your initial Hessian guess B_0 equal to the Gauss-Newton Hessian, i.e., $B_0 = 2\mathbb{I}$?

5. For a start, save your old routine for $J_{\text{sim}}(U)$ in a separate folder to be able to compare the results of your new routine with it later.

6. Then, note that the RK4 integrator step can be summarized in a function Φ so that the last state x_N, i.e., the output of the function $g_{\text{sim}}(U)$, is obtained by the recursion

$$x_{k+1} = \Phi(x_k, u_k), \quad k = 0, \ldots, N-1.$$

   Along the simulated trajectory $\{(x_k, u_k)\}_{k=0}^{N-1}$, this system can be linearized as

$$\delta x_{k+1} = A_k \delta x_k + B_k \delta u_k, \quad k = 0, \ldots, N-1.$$

   where the matrices

$$A_k := \frac{\partial \Phi}{\partial x}(x_k, u_k) \quad \text{and} \quad B_k := \frac{\partial \Phi}{\partial u}(x_k, u_k).$$

   can be computed by finite differences. Note that we use the symbol B_k here for coherence with the notation of linear system theory, but that this symbol B_k here has nothing to do with the Hessian matrix B_k used in the other questions.

   More specifically, modify your integrator such that

   - Your RK4 step is encapsulated in a single function `[xnew]=RK4step(x,u)`.
   - You also write a function `[xnew,A,B]=RK4stepJac(x,u)` using finite differences with a step size of $\delta = 10^{-4}$.

- Your integrator stores and outputs both the trajectory of states $\{x_k\}_{k=0}^{N-1}$ and the trajectory of matrices $\{(A_k, B_k)\}_{k=0}^{N-1}$. Use three dimensional tensors like `Atraj(i,j,k)`.

   The interface of the whole routine could be `[x,Atraj,Btraj]=forwardsweep(U)`.

7. Now, using the matrices A_k, B_k, we want to compute $J_{\text{sim}}(U)$, i.e., write a routine with the interface `[Jsim]=backwardsweep(Atraj,Btraj)`. For this aim we observe that

$$\frac{\partial g_{\text{sim}}}{\partial u_k}(U) = \underbrace{(A_{N-1}A_{N-2}\cdots A_{k+1})}_{=:G_{k+1}} B_k$$

   In order to compute all derivatives $\frac{\partial g_{\text{sim}}}{\partial u_k}(U)$ in an efficient way, we compute the matrices $G_{k+1} = (A_{N-1}A_{N-2}\cdots A_{k+1})$ in reverse order, i.e., we start with $k = N-1$ and then go down to $k = 0$. We start by $G_N := \mathbb{I}$ and then compute

$$G_k := G_{k+1}A_k, \quad k = N-1, \ldots, 0$$

8. Combining the forward and the backward sweep from the previous two questions, and write a new function for $J_{\text{sim}}(U)$. It is efficient to combine it with the computation of $g_{\text{sim}}(U)$, i.e., to use an interface like `[gsim,Jsim]=gsimJac(U)`. Compare the result with the numerical Jacobian calculation from before by taking `norm(Jsimold-Jsimnew)`.

9. How do the computation times of the old and the new Jacobian routine scale with N? This question can be answered without any numerical experiments, just by thinking.

10. Now run your Gauss-Newton algorithm again and verify that it gives the same solution and same number of iterations as before.

D Simultaneous Optimal Control with Gauss-Newton

In this part we use the simultaneous approach with SQP and a Gauss-Newton Hessian to solve the optimal control problem.

1. Throughout this part, we consider again the discrete time system

$$x_{k+1} = \Phi(x_k, u_k)$$

   that is generated by one RK4 step applied to the controlled nonlinear pendulum. Its state is $x = (p, v)^T$ and the ODE $\dot{x} = f(x, u)$ is with $C := 180/\pi/10$ given as

$$f(x,u) = \begin{bmatrix} v(t) \\ -C\sin(p(t)/C) \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u(t). \tag{14}$$

   We use again a time step $\Delta t = 0.2$ and recall that one RK4 step is given by

$$\begin{aligned} k_1 &= f(x_k, u_k) && (15)\\ k_2 &= f(x_k + \frac{1}{2}\Delta t\ \cdot k_1, u_k) && (16)\\ k_3 &= f(x_k + \frac{1}{2}\Delta t\ \cdot k_2, u_k) && (17)\\ k_4 &= f(x_k + \Delta t\ \cdot k_3, u_k) && (18)\\ x_{k+1} &= x_k + \Delta t\,\frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) && (19) \end{aligned}$$

   Write the function $\Phi(x_k, u_k)$ as a MATLAB code encapsulated in a single function `[xnew]=RK4step(x,u)`.

2. Now we choose again the initial value $\bar{x}_0 = (10, 0)^T$ and $N = 50$ time steps. We also define bounds on the sizes of p, v, and u, namely $p_{\max} = 10$, $v_{\max} = 10$, i.e., $x_{\max} = (p_{\max}, v_{\max})^T$, and $u_{\max} = 3$. The optimization problem we want to solve is given by

$$\underset{x_0, u_0, x_1, \ldots, u_{N-1}, x_N}{\text{minimize}} \quad \sum_{k=0}^{N-1} \|u_k\|_2^2 \tag{20a}$$

$$\begin{aligned}
\text{subject to} \qquad \bar{x}_0 - x_0 &= 0, && && \text{(20b)}\\
\Phi(x_k, u_k) - x_{k+1} &= 0, && \text{for} \quad k = 0, \ldots, N-1, && \text{(20c)}\\
x_N &= 0, && && \text{(20d)}\\
-x_{\max} \leq x_k &\leq x_{\max}, && \text{for} \quad k = 0, \ldots, N-1, && \text{(20e)}\\
-u_{\max} \leq u_k &\leq u_{\max}, && \text{for} \quad k = 0, \ldots, N-1. && \text{(20f)}
\end{aligned}$$

Summarizing the variables of this problem in a vector $w = (x_0, u_0, \ldots, u_{N-1}, x_N) \in \mathbb{R}^n$ of dimension $n = 152$, formulate the nonlinear function $G(w)$, Hessian matrix H, and bounds $w_{\max}$ such that the above problem is an NLP of the following form:

$$\underset{w \in \mathbb{R}^{152}}{\text{minimize}} \quad w^T H w \tag{21a}$$

$$\begin{aligned}
\text{subject to} \quad G(w) &= 0, && \text{(21b)}\\
-w_{\max} \leq w &\leq w_{\max}. && \text{(21c)}
\end{aligned}$$

Define what H_x and H_u need to be in the Hessian

$$H = \begin{bmatrix} H_x & & & \\ & H_u & & \\ & & \ddots & \\ & & & H_x \end{bmatrix}.$$

For G, use the same ordering as in the OCP above:

$$G(w) = \begin{bmatrix} \\ \\ \vdots \\ \\ \\ \end{bmatrix} \quad w_{\max} = \begin{bmatrix} \\ \\ \\ \\ \end{bmatrix}$$

Construct the matrix H and vector $w_{\max}$ in MATLAB, and write a MATLAB function `[G]=Gfunc(w)`.

3. Check if your function $G(w)$ does what you want by writing a forward simulation function `[w]=simulate(x0,U)` that simulates, for a given initial value x_0 and control profile $U = (u_0, \ldots, u_{N-1})$, the whole trajectory $x_1, \ldots, x_N$ and constructs from this the full vector $w = (x_0, u_0, x_1, \ldots, x_N)$. If you generate for any x_0 and U a vector w and then you call your function $G(w)$ with this input, nearly all your residuals should be zero. Which components will not be zero?

   As a test, simulate, e.g., with $x_0 = (5, 0)$ and $u_k = 1$, $k = 0, \ldots, N-1$ in order to generate w, and then call $G(w)$, to test that your function G is correct.

4. The SQP with Gauss-Newton Hessian (also called *constrained Gauss-Newton method*) solves a linearized version of this problem in each iteration. More specific, if the current iterate is $\bar{w}$, the next iterate is the solution of the following QP:

$$\begin{aligned} \underset{w \in \mathbb{R}^{152}}{\text{minimize}} \quad & w^T H w && (22a) \\ \text{subject to} \quad G(\bar{w}) + J_G(\bar{w})(w - \bar{w}) &= 0, && (22b) \\ -w_{\max} \leq w &\leq w_{\max}. && (22c) \end{aligned}$$

Important for implementing the Gauss-Newton method is the computation of the Jacobian $J_G(w) = \frac{\partial G}{\partial w}(w)$, which is block sparse with as blocks either (negative) unit matrices or the partial derivatives $A_k = \frac{\partial \Phi}{\partial x}(x_k, u_k)$ and $B_k = \frac{\partial \Phi}{\partial u}(x_k, u_k)$. Fill in the corresponding blocks in the following matrix

$$J_G(w) = \begin{bmatrix} & & & \\ & \ddots \quad \ddots \quad \ddots & & \\ & & & \end{bmatrix}$$

5. In this part we compute the Jacobian $J_G(w)$ just by finite differences, perturbing all 152 directions one after the other. This needs in total 153 calls of G. Give your routine, e.g., the name `[G,J]=GfuncJacSlow(w)`. Compute J_G for a given w (e.g., the one from above) and look at the structure this matrix, e.g., using the command `spy(J)`.

6. Now learn how to use the MATLAB QP solver `quadprog` by calling `help quadprog`.

7. Write a function `[wplus]=GNStep(w)` that performs one SQP-Gauss-Newton step by first calling `[G,J]=GfuncJac(w)` and then solving the resulting QP (22) using `quadprog`. Note that the QP is a very sparse QP but that this sparsity is not exploited by `quadprog`. This is a limitation of `quadprog` rather than the method in itself.

8. Write a loop around your function `GNStep`, initialize the GN procedure at at $w = 0$, and stop the iterations when $\|w_{k+1} - w_k\|$ gets smaller than 10^{-4}. Plot the iterates as well as the vector G during the iterations. How many iterations do you need?

E Pontryagin and the Indirect Approach

In this part we use the indirect approach to solve the continuous time optimal control problem. We first employ Pontryagin's Maximum Principle to derive the Euler-Lagrange differential equation in states and adjoints, using only pen and paper. Second, we numerically solve the resulting boundary value problem (BVP) using single shooting and a Newton method. The task is to find only one vector, λ_0. Once it is found, we can generate the complete optimal solution just by a forward ODE simulation.

1. In this part, we regard again the continuous time optimal control problem

$$\begin{aligned}
\underset{x(\cdot),\,u(\cdot)}{\text{minimize}} \quad & \int_0^T u(t)^2 \quad dt && \text{(23a)}\\
\text{subject to} \quad & x(0) \;=\; \bar{x}_0, && \text{(23b)}\\
& \dot{x}(t) \;=\; f(x(t),u(t)), \quad \text{for} \quad t \in [0,T] && \text{(23c)}\\
& x(T) \;=\; 0, && \text{(23d)}\\
& -u_{\max} \;\le\; u(t) \;\le\; u_{\max}, \quad \text{for} \quad t \in [0,T] && \text{(23e)}
\end{aligned}$$

The state is $x = (x_1, x_2)^T$ and the ODE $\dot{x} = f(x,u)$ is with $C := 180/\pi$ given as

$$f(x,u) = \begin{bmatrix} x_2(t) \\ -C\sin(x_1(t)/C) + u(t) \end{bmatrix}. \tag{24}$$

We choose again the initial value $\bar{x}_0 = (10,0)^T$ and $T = 10$. We will first leave away the control bound (23e), but at the end we will add it again.

We remember that the Hamiltonian function for a general OCP with integral cost $L(x,u)$ is defined to be $H(x,\lambda,u) = L(x,u) + \lambda^T f(x,u)$. In our case, we have $L(x,u) = u^2$.

Write down explicitly the the Hamiltonian function of the above optimal control problem as a function of the five variables $(x_1, x_2, \lambda_1, \lambda_2, u)$.

2. Next, let us recall that the indirect approach eliminates the controls to obtain an explicit function $u^*(x,\lambda)$ that minimizes the Hamiltonian for given (x,λ), i.e., $u^*(x,\lambda) = \arg\min_u H(x,\lambda,u)$. This optimal u^* can be computed by setting the gradient of the Hamiltonian w.r.t. u to zero, i.e. it must hold $\frac{\partial H}{\partial u}(x,\lambda,u^*) = 0$. First derive $\frac{\partial H}{\partial u}(x,\lambda,u)$. Thereafter, invert this relation so that you obtain the explicit expression for the controls $u^*(x,\lambda)$.

3. We will also need the derivatives w.r.t. x. Hence, also calculate $\frac{\partial H}{\partial x_1}(x,\lambda,u)$ and $\frac{\partial H}{\partial x_2}(x,\lambda,u)$.

4. Now we recall that the indirect approach formulates the Euler-Lagrange differential equations for the states and adjoints together. They are given by $\dot{x} = f(x, u^*(x,\lambda))$ and by $\dot{\lambda} = -\nabla_x H(x,\lambda,u^*(x,\lambda))$. For notational convenience, we define the vector $y = (x,\lambda)$ so that the Euler-Lagrange equation can be briefly written as the ODE $\dot{y} = \tilde{f}(y)$.

Collect all data from above to explicitly define the ODE right hand side $\tilde{f}(y)$ as a function of the four components of the vector $y = (x_1, x_2, \lambda_1, \lambda_2)$.

5. The boundary value problem (BVP) that we now have to solve is given by

$$\begin{aligned}
x(0) &= \bar{x}_0, && (25)\\
x(T) &= 0, && (26)\\
\dot{y}(t) &= \tilde{f}(y(t)), \quad t \in [0,T]. && (27)
\end{aligned}$$

We will solve it by single shooting and a Newton procedure. The first step is to write an ODE simulator that for a given initial value $y_0 = (x_0, \lambda_0)$ simulates the ODE on the whole time horizon, i.e., that solves the initial value problem

$$\begin{aligned}
y(0) &= y_0, && (28)\\
\dot{y}(t) &= \tilde{f}(y(t)), \quad t \in [0,T]. && (29)
\end{aligned}$$

Let us call the resulting trajectory $y(t; y_0)$, $t \in [0, T]$, and denote its terminal value by $y(T; y_0)$. We can numerically generate this terminal value adapting our RK4 integrator with step size $\Delta t = 0.2$ and $N = 50$ time steps. Write a simulation routine that computes for given y_0 the value $y_N = y(T; y_0)$. Recall that one RK4 step is given by

$$
\begin{aligned}
k_1 &= \tilde{f}(y_k) && (30)\\
k_2 &= \tilde{f}(y_k + \frac{1}{2}\Delta t \cdot k_1) && (31)\\
k_3 &= \tilde{f}(y_k + \frac{1}{2}\Delta t \cdot k_2) && (32)\\
k_4 &= \tilde{f}(y_k + \Delta t \cdot k_3) && (33)\\
y_{k+1} &= y_k + \Delta t \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) && (34)
\end{aligned}
$$

6. Regard the initial value $y_0 = (x_0, \lambda_0)$. As the initial value for the states, x_0, is fixed to $\bar{x}_0$, we only need to find the right initial value for the adjoints, λ_0, i.e. we will fix $x_0 = \bar{x}_0$ and only keep $\lambda_0 \in \mathbb{R}^2$ as an unknown input to our simulator. Also, we have only to meet a terminal condition on $x(T)$, namely $x(T) = 0$, while $\lambda(T)$ is free. Thus, we are only interested in the map from λ_0 to $x(T)$, which we denote by $F(\lambda_0)$. Note that $F : \mathbb{R}^2 \to \mathbb{R}^2$. Using your simulator, write a MATLAB function `[x_end]=F(lambda_start)`.

7. Add to your function functionality for plotting the trajectories of $x_1, x_2, \lambda_1, \lambda_2$, e.g., by extending the output to `[x_end,ytraj]=F(lambda_start)`.

   For $\lambda_0 = 0$, call $F(\lambda_0)$ and plot the states and adjoints of your system. In this scenario, what is the numerical value of the final state $x(T) = F(0)$?

8. The solution of the BVP is found if we have found λ_0^* such that $F(\lambda_0^*) = 0$. This system can be solved by Newton's method, that iterates, starting with some guess $\lambda_0^{[0]}$ (e.g. zero).

$$\lambda_0^{[k+1]} = \lambda_0^{[k]} - \left(\frac{\partial F}{\partial \lambda_0}(\lambda_0^{[k]})\right)^{-1} F(\lambda_0^{[k]})$$

   First write a routine that computes the Jacobian $J_F(\lambda_0) = \frac{\partial F}{\partial \lambda_0}(\lambda_0)$ by finite differences using a perturbation $\delta = 10^{-4}$. Then implement a (full-step) Newton method that stops when $\|F(\lambda_0^{[k]})\| \leq \text{TOL}$ with $\text{TOL} = 10^{-3}$.

9. For your obtained solution, plot the resulting state trajectories and verify by inspection that $x(T) = 0$.

10. Using your function $u^*(x, \lambda)$ from Task E.2, also plot the corresponding control trajectories $u(t)$.

11. You can now add the control bounds (23e) with $u_{\max} = 3$. The only part in your whole algorithm that you need to change is the expression for $u^*(x, \lambda)$. The new constrained function

$$u^*_{\text{con}}(x, \lambda) = \arg\min_u H(x, \lambda, u) \quad \text{s.t.} \quad -u_{\max} \leq u \leq u_{\max},$$

   is simply given by the "clipped" or "saturated" version of your old unconstrained function $u^*_{\text{unc}}(x, \lambda)$, namely by

$$u^*_{\text{con}}(x, \lambda) = \max\{-u_{\max}, \min\{u_{\max}, u^*_{\text{unc}}(x, \lambda)\}\}$$

Modify your differential equation $\tilde{f}$ by using this new expression for u^* and run your algorithm again. We remark that strictly speaking, the ODE right hand side is no longer differentiable so that the use of a RK4 is questionable as well as the computation of the Jacobian of F, but we cross our fingers and are happy that it works. For initialization of the Newton procedure, choose the multiplier λ_0^* from the unconstrained solution.

In the solution, plot again the resulting trajectories for states, adjoints, and for $u(t)$, using of course your new function. Compare with the solution you obtained previously with the Gauss-Newton approach in Part D.

F Real-Time Iterations and Model Predictive Control

This part builds exactly on the simultaneous Gauss-Newton algorithm developed in Part D. We modify this algorithm so that it allows us to perform real-time iterations for different values of $\bar{x}_0$. We finally use this algorithm to perform closed-loop NMPC simulations for stabilization of the nonlinear pendulum.

1. We use the same setup as in Task D.2 with $C = 180/\pi/10$ and recall that the parametric optimization problem we want to solve is given by the equation in (20). Note, however, that the dependence on the initial state $\bar{x}_0$ will be more explicitly considered in this task. Summarizing the variables of this problem in a vector $w = (x_0, u_0, \ldots, u_{N-1}, x_N) \in \mathbb{R}^n$ of dimension $n = 152$, we note that the problem can be summarized as a parametric NLP of the form:

$$\text{pNLP}(\bar{x}_0): \quad \underset{w \in \mathbb{R}^{152}}{\text{minimize}} \quad w^T H w \tag{35a}$$

$$\text{subject to}$$

$$G(\bar{x}_0, w) = 0, \tag{35b}$$

$$-w_{\max} \leq w \leq w_{\max}. \tag{35c}$$

   Modify your function G such that it accepts as argument also the parameter $\bar{x}_0$.

2. The Gauss-Newton real-time iteration solves a linearized version of this problem in each iteration, for varying values of $\bar{x}_0$. More specific, if the last iterate was $\bar{w}$, and we want to solve a problem with the parameter $\bar{x}_0'$, the next iterate w' is determined as the solution of the following QP:

$$\text{pQP}(\bar{x}_0', \bar{w}): \quad \underset{w \in \mathbb{R}^{152}}{\text{minimize}} \quad w^T H w \tag{36a}$$

$$\text{subject to}$$

$$G(\bar{x}_0', \bar{w}) + J_G(\bar{w})(w - \bar{w}) = 0, \tag{36b}$$

$$-w_{\max} \leq w \leq w_{\max}. \tag{36c}$$

   Modify your previous function `GNStep` so that it accepts the parameter $\bar{x}_0$ as a second input, i.e., write a function `[wplus]=GNRTIStep(xprime,w)` that performs one SQP-Gauss-Newton real-time iteration by solving $\text{pQP}(\bar{x}_0', w)$.

3. In order to visualize the generalized tangential predictor, fix the variable vector w at zero and call your function `GNRTIStep` with different values for $\bar{x}_0$. Use linear interpolation of 100 points between zero and the value $(10, 0)^T$, i.e., set $\bar{x}_0 = \lambda(10, 0)^T$ for $\lambda \in [0, 1]$. Plot the first control u_0 as a function of λ and keep your plot.

4. In order to compute the exact solution manifold with relatively high accuracy, perform now the same procedure for the same 100 increasing values of λ, but this time perform in each iteration the Gauss-Newton step, i.e., set w to the output of the last real-time iteration. Plot the obtained values for u_0 and compare with the tangential predictor by plotting them in the same plot.

5. In order to see how the real-time iterations work in a more realistic setting, let the values of λ jump faster from 0 to 1, e.g., by doing only 10 steps, and plot the result again into the same plot.

6. Modify the previous algorithm as follows: after each change of λ by 0.1 keep it for 9 iterations constant, before you do the next jump. This will result in about 100 consecutive real-time iterations. Interpret what you see.

7. Now we do the first *closed-loop simulation*: set the value of $\bar{x}_0'$ to $(10, 0)^T$ and initialize w at zero, and perform the first real-time iteration by solving $\mathrm{pQP}(\bar{x}_0', w)$. This iteration yields the new solution guess w' and corresponding control u_0'. Use this control at the "real plant", i.e., generate the next value of $\bar{x}_0$, say $\bar{x}_0''$, by calling the one step simulation function, $\bar{x}_0'' := \Phi(\bar{x}_0', u_0')$. Close the loop by solving now $\mathrm{pQP}(\bar{x}_0'', w')$, etc., and perform 100 iterations. For better observation, plot after each real-time iteration the control and state variables on the whole prediction horizon. (It is interesting to note that the state trajectory is not necessarily feasible).

   Also observe what happens with the states $\bar{x}_0$ during the scenario, and plot them in another plot the sequence against the time index. Do they converge, and if yes, to what value?

G Optimal Control using ACADO

In this part, we will consider two problems: energy optimal control of a cart and time optimal control of a crane.

Some prerequisites before we start is to make sure that the package is ACADO package is setup and that you have tested the installation. Some instructions are given below.

- Follow the instructions on the course page to download your copy of `acadoTemplate.zip`.
- Extract the archive.
- Edit the `Makefile` so that `LOCAL_PATH_PREFIX` points to the place where ACADO is installed on your system.
- Test the installation by, at the prompt, executing `make rocket` and then `./rocket` to make sure that the installation and `Makefile` are correctly configured.
- In addition, it might help to briefly browse through the ACADO online tutorials at `www.acadotoolkit.org` to look up the notation. The `rocket` example is included in the ACADO toolkit as a standard example.

G.1 Energy Optimal Control of a Cart using ACADO

In this part we will consider the Bryson-Denham problem (see, e.g., Bryson and Ho (1975), Chapter 11.3). It is a minimum energy problem of a friction free cart with unit mass. A sketch of the problem is shown in Figure 1. The problem is stated as follows: At $t = 0$ the cart is at $x(0) = 0$

with $v(0) = 1$ and travels towards a wall (positioned at $s = l$). At $t = 1$ the cart should be back at $x(1) = 0$ with velocity $v(1) = -1$ without hitting the wall, i.e., $s \leq l$, and the goal is minimize the control energy (i.e., $\frac{1}{2}\int_0^1 (u(t))^2 dt$). Since the cart is friction free and has unit mass the differential equation for the position is a double integrator $\ddot{x}(t) = u(t)$.

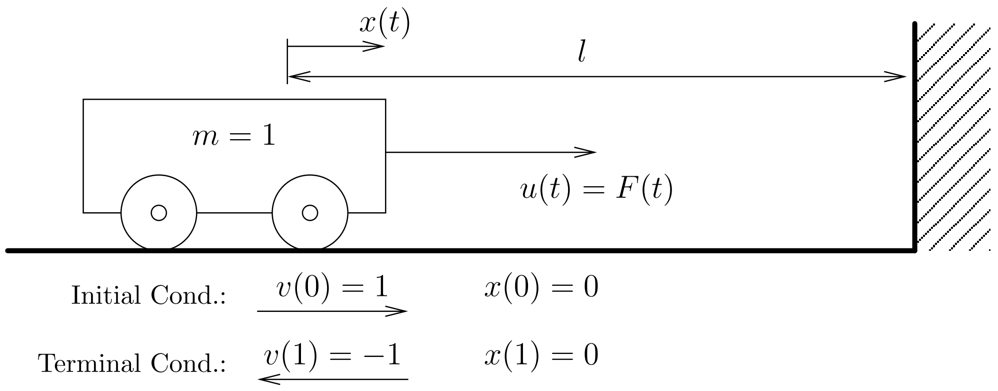

Figure 1: Sketch of the problem, showing the cart with initial conditions, final conditions, and constraint.

1. Now we'll start looking at the optimal control problem. What is the mathematical definition of the objective function?
2. Write the dynamic equations on standard ODE form $\dot{x} = f(x, u)$.
3. What are the constraints for the problem?
4. Summarize once again the objective and the constraints and write them down in the standard formulation of optimal control problems. What type of problem is it: Mayer, Lagrange, or Bolza?
5. In order to discretize the control input, we choose a piecewise constant control discretization with 40 pieces. How many degrees of freedom does the above optimal control problem have? How many constraints do you have?
6. Next implement and solve the optimal control problem using ACADO with $l = 1$. It might help to briefly browse through the ACADO online tutorials at `www.acadotoolkit.org` to look up the notation. A starting point is available in the file `BrysonDenham.cpp` (included in `acadoTemplate.zip`).

   When the problem is implemented you can run it by executing `make BrysonDenham` and `./BrysonDenham` at the command prompt.
7. Depending on l the solutions will show three different types of characteristics, for $l \geq a$ the solution is the same as the unconstrained one. Play with l and see how the solution changes character. For what values of l does the solution switch character? (You can for example look at the solution for $l = 1$ and get an estimate of what a is. Then there is another value where the character switches again.)

G.2 Time Optimal Control of a Crane using ACADO

In this part, we will consider a crane model:

$$\begin{aligned} \dot{x}(t) &= v(t) & (37) \\ \dot{v}(t) &= a_x(t) & (38) \\ \ddot{\phi}(t) &= -\frac{a_x(t)}{L}\cos(\phi(t)) - \frac{g}{L}\sin(\phi(t)) - \frac{b}{mL^2}\dot{\phi}(t)\ . & (39) \end{aligned}$$

Use the parameters $m = 1\,\mathrm{kg}$, $L = 1\,\mathrm{m}$, $b = 0.2\,\mathrm{J\,s}$ as well as $g = 9.81\,\frac{\mathrm{m}}{\mathrm{s}^2}$.

Our aim is to move the crane from one to another point as fast as possible. However, there are restrictions: first the control input $a_x(t)$ should during the maneuver be between $-5\,\frac{\mathrm{m}}{\mathrm{s}^2}$ and $5\,\frac{\mathrm{m}}{\mathrm{s}^2}$. Moreover, the crane should be started at $\phi(0) = 0$ and $x(0) = 0\,\mathrm{m}$ being at rest. The target position is $x(T) = 10\,\mathrm{m}$ and $\phi(T) = 0$ stopping again at rest. Here, T is the end time.

1. As specified we would like to move the crane as fast as possible between the specified points. What is the mathematical definition of the objective function?

2. How can the restrictions from the introduction be formulated in form of constraints?

3. Summarize once again the objective and the constraints and write them down in the standard formulation of optimal control problems.

4. In order to discretize the control input, we choose a piecewise constant control discretization with 40 pieces. How many degrees of freedom does the above optimal control problem have? How many constraints do you have?

5. Solve the above problem optimal control problem in ACADO. You can use the code template below to learn the syntax (also included in `timeOptimalControl.cpp` in `acadoTemplate.zip`).

6. What is the result for the minimum time? Can you interpret the result for the optimal control input?

7. By default ACADO uses an SQP algorithm with BFGS updates. Use the ACADO command `acadoGetTime()` to measure how long time the optimization takes? In the next step we would like to use an exact Hessian. For this aim we set the option

   ```
   algorithm.set( HESSIAN_APPROXIMATION, EXACT_HESSIAN );
   ```

   How does the behavior of the algorithm change and how long does it take?

8. Look up the option settings on the ACADO webpage and try how fast you can make the algorithm by adjusting, e.g., the KKT tolerance, integrator tolerances or by providing a better initial guess for the optimal solution.

```
#include <acado_optimal_control.hpp>
#include <gnuplot/acado2gnuplot.hpp>


int main( ){

    USING_NAMESPACE_ACADO;

    // VARIABLES:
    // ---------------------------
    DifferentialState          x;    // Position of the trolley
    ....

    Parameter                  T;    // the end time
    Control                   ax;    // trolley accelaration


    // DIFFERENTIAL EQUATION:
    // ---------------------------
    DifferentialEquation f( t_start , T );

    f << ...   // The model equations


    // DEFINE AN OPTIMAL CONTROL PROBLEM:
    // --------------------------------------

    const int numberOfIntervals = 40;

    OCP ocp;
    ocp.minimizeMayerTerm( T );
    ocp.subjectTo( f, numberOfIntervals );

    ocp.subjectTo( AT_START, x == 0.0  );
    ocp.subjectTo( AT_END  , x == 10.0 );

    ocp.subjectTo(  0.0 <= T  <= 10.0  );

    ... // implement the other constraints.


    // DEFINE A PLOT WINDOW:
    // ---------------------
    GnuplotWindow window;
        window.addSubplot( x    ,"POSITION OF THE TROLLEY:  x"    );
        window.addSubplot( v    ,"VELOCITY OF THE TROLLEY:  v"    );
        window.addSubplot( phi  ,"EXCITATION ANGLE        :  phi"  );
        window.addSubplot( dphi ,"ANGLUAR VELOCITY        :  dphi" );
        window.addSubplot( ax   ,"THE CONTROL INPUT       :  aw"   );


    // DEFINE AN OPTIMIZATION ALGORITHM AND SOLVE THE OCP:
    // ----------------------------------------------------
    OptimizationAlgorithm algorithm(ocp);
    algorithm << window;
    algorithm.solve();

    return 0;
}
```