ECE-597: Optimal Control
Homework #5

Due: October 10, 2007

Read the Appendix before attempting problems 1-4.

1. In this problem we will try some things with **Example A**.

a) Run the program **fop0_example_a.m** for **Example A**, using the initial estimate of u given in the program. Use $N = 10$ and mxit = 10. Turn in your plot.

b) Comment out the initial estimate, and run the program again for $N = 20$ by interpolating the solution it ended with on part (a)

c) Use this initial (unconverged but heading towards a solution) u and run **fminunc_example_a.m** for N = 20 time samples.

d) After each run of **fminunc_example_a.m**, double the number of sample points until you have a solution for at least $N = 200$. Turn in your plot.

e) Run **fop0_example_a.m** until it converges, and compare with the results from **fminunc_example_a.m**. You should not use the same $u's$ that **fminunc_example_a.m** used. Start with a new initial guess. Your answers should be slightly different. Turn in your plot.

2. Rerun the system in **Example A** using initial values of $x(0) = 2$, $y(0) = -3$ using both **fop0_example_a.m** and **fminunc_example_a.m**.

a) Run **fop0_example_a.m** using $N = 20$ and mxit = 10. Use the default initial estimate. Turn in your plot.

b) Using the initial guess from part **a**, run **fminunc_example_a.m** for $N = 20$, then $N = 100$, then $N = 200$, interpolating you initial as you refine. Turn in your plot.

c) Rerun part **a** (to get the same initial values), and then run **fminunc_example_a.m** for $N = 200$. Does the routine find a solution? Turn in your plot.

d) Find the solution using **fop0_example_a.m**. You should not use the same $u's$ that **fminunc_example_a.m** used. Start with a new initial guess. Your answers should be slightly different. Turn in your plot.

3. Using initially **fop0_example_b.m** to find a good initial guess, and then **fminunc_example_b.m**, modify Q_f, R, and Q so that x_1 is within ± 0.1 of 1 for at least half a second. You need to use $N \geq 100$. Turn in your plot.

4. In this problem, we will first derive a somewhat complicated analytical solution, and then simulate the system.

Consider the problem

$$\begin{aligned} \text{minimize } J &= \frac{1}{2}(x(t_f)-d)^2 + \frac{1}{2}\int_{t_0}^{t_f} u(t)^2 dt \\ \text{subject to} &\quad \dot{x}(t) = ax(t) + bu(t) \end{aligned}$$

where $x(t_0) = x_0$, t_0, t_f, and the desired final point, d are given.

a) Show that the continuous time Euler Lagrange equations are

$$\begin{aligned} \dot{\lambda} &= -a\lambda \\ \lambda(t_f) &= x(t_f) - d \\ H_u &= u + \lambda b = 0 \end{aligned}$$

b) Show that we can use these equations to write

$$u = \gamma e^{-at}$$

where

$$\gamma = b(d - x(t_f))e^{at_f}$$

c) Show that the constraint equation becomes

$$\dot{x} - ax = b\gamma e^{-at}$$

and that we can rewrite this as

$$\frac{d}{dt}\left(e^{-at}x\right) = b\gamma e^{-2at}$$

d) Integrating the above equation from t_0 to t, show that we get

$$x(t) = x(t_0)e^{a(t-t_0)} + \frac{b\gamma}{2a}(e^{a(t-2t_0)} - e^{-at})$$

e) Show that the above equation can be written

$$x(t) = x(t_0)e^{a(t-t_0)} + \frac{b\gamma}{a}e^{-at_0}\sinh(a(t-t_0))$$

f) Evaluating the above expression at the final time t_f show that we get

$$x(t_f) = \frac{x(t_0)e^{a\delta} + \frac{b^2 d}{2a}(e^{2a\delta} - 1)}{1 + \frac{b^2}{2a}(e^{2a\delta} - 1)}$$

where $\delta = t_f - t_0$. At this point, we have an analytical expression for the final value of $x(t)$ that only depends on known quantities.

g) Using the above expression, show that we can write

$$\gamma = be^{at_f}\left[\frac{d - x(t_0)e^{a\delta}}{1 + \frac{b^2}{2a}(e^{2a\delta} - 1)}\right]$$

h) Now we want to simulate the problem using the routines **fop0.m** and **fminunc**. We will examine the specific problem with $a = -0.8$, $b = 0.2$, $x_0 = 1, t_0 = 0, t_f = 2$ and $d = -4.0$ Plot the analytical and estimated values of the control signals (u) and the states (x). You must estimate these signals using both **fop0.m** and **fminunc**. You may use you final values from **fop0.m** as the initial values for **fminunc**, but I want to be sure you can code both of these. Show that your estimates agree with the analytical values. *Turn in your plots and you code.*

5. In this problem we will derive the continuous time version of the Linear Quadratic tracker problem, then learn to use Matlab to determine the solution by solving the differential equations. The solution technique will be very similar to what you did last week for the discrete-time case. For this problem we want to determine the continuous time control signal $u(t)$ to minimize the cost

$$J = \frac{1}{2}\left[y(t_f) - r(t_f)\right]^T Q_f \left[y(t_f) - r(t_f)\right] + \frac{1}{2}\int_o^{t_f} \left\{\left[y(t) - r(t)\right]^t Q \left[y(t) - r(t)\right] + u(t)^T R u(t)\right\} dt$$

subject to the constraints

$$\begin{aligned} \dot{x}(t) &= Ax(t) + Bu(t) \\ y(t) &= Cx(t) \\ x(0) &= x_0 \end{aligned}$$

We also assume all matrices are symmetric and invertible.

a) Determine the Hamiltonian and $\phi(x(t_f))$, then show that the Euler-Lagrange equations lead to

$$\begin{aligned} -\dot{\lambda}(t) &= C^T Q C x(t) - C^T Q r(t) + A^T \lambda(t) \\ u(t) &= -R^{-1} B^T \lambda(t) \end{aligned}$$

b) Now assume $\lambda(t) = S(t)x(t) - v(t)$, so that $\dot{\lambda}(t) = \dot{S}(t)x(t) + S(t)\dot{x}(t) - \dot{v}(t)$. Show that by substitution we can eliminate $\lambda(t)$ and $\dot{x}(t)$, and end up with the equation (ignoring all the functions of t)

$$\left\{-\dot{S} - SA - A^T S + SBR^{-1}B^T S - C^T Q C\right\} x + \left\{-SBR^{-1}B^T v + C^T Q r + A^T v + \dot{v}\right\} = 0$$

Since this equation must be true for all x, we have

$$\begin{aligned}\dot{S}(t) &= -S(t)A - A^T S(t) + S(t)BR^{-1}B^T S(t) - C^T QC \\ \dot{v}(t) &= -\left[A^T - S(t)BR^{-1}B^T\right] v(t) - C^T Qr(t)\end{aligned}$$

The equation for $S(t)$ is called the *Riccati* equation, and is very important in optimal control and anytime you use a Linear Quadratic anything.

c) Show that the boundary conditions for $S(t)$ and $v(t)$ are given by

$$\begin{aligned}S(t_f) &= C^T QC \\ v(t_f) &= C^T Q_f r(t_f)\end{aligned}$$

d) Except for really weenie problems, we can't solve the Riccati equation analytically. Hence we must resort to using a differential equation solver. However, most differential equation solvers want to start with an initial condition and march forward in time. We have no initial condition but a condition at the end point for both of our differential equations. In order to use Matlab's differential equation solvers, we will make a change of variable so we have initial conditions, and then "time-reverse" the solutions to get what we want (this will be done for you in the Matlab code). Define

$$\begin{aligned}Z(t_f - t) &= Z(\tau) = S(t) \\ W(t_f - t) &= W(\tau) = v(t)\end{aligned}$$

and show our differential equations for $S(t)$ and $v(t)$ can be transformed to

$$\begin{aligned}\frac{dZ(\tau)}{d\tau} &= Z(\tau)A + A^T Z(\tau) - Z(\tau)BR^{-1}B^T Z(\tau) + C^T QC \\ Z(0) &= C^T QC \\ \frac{dW(\tau)}{d\tau} &= \left[A^T - S(t_f - \tau)BR^{-1}B^T\right] W(\tau) + C^T Qr(t_f - \tau) \\ W(0) &= C^T Q_f r(t_f)\end{aligned}$$

We can now use Matlab's differential equation solvers to march forward in time. The code **continuous_tracker.m** implements a solution to the continuous-time linear tracker problem, including numerically solving the differential equations for $S(t)$ and $v(t)$. You should try and understand what is going on in this program in case it shows up on your next exam....

e) The program is set up to control the position of an ECP one-degree of freedom torsional system. Choose the gains (Q, S, and Q_f) so that

i) the system tracks a ramp ($r(t) = 0.7t$) as well as possible from 0 to 2 seconds.

ii) the system matches the ramp at 2 seconds, but the "minimum" energy is used

iii) the system tracks a cosine $r(t) = \frac{\pi}{30}\sin(2\pi t)$ as well as possible.

iv) modify the program to use the steady state gain and rerun (iii). This is the gain that the Matlab routine LQR generates. Turn in all four (neatly labeled) plots.

Appendix

fop0.m and fminunc

Only very simple problems can be solved analytically. Usually we need to use a numerical procedure. We will do this using the *Mayer formulation* of the problem. We have been using the *Bolza formulation*. Both of these methods are *equivalent*, and are just different methods of looking at the same thing.

In the *Mayer* formulation, the state vector is augmented by one state $q(t)$ that is the integral of L up to the final time:

$$q = \int_{t_0}^{t_f} L[x(t), u(t), t]dt$$

or

$$\dot{q}(t) = L[x(t), u(t), t]$$

The *Bolza* performance index

$$J = \phi[x(t_f)] + \int_{t_0}^{t_f} L\left[x(t), u(t), t\right] dt$$

becomes the *Mayer* performance index

$$J = \phi[x(t_f)] + q(t_f) = \bar{\phi}[\bar{x}(t_f)]$$

where

$$\bar{x} = \left[\begin{array}{c} x \\ q \end{array} \right]$$

We usually drop the bar on x and ϕ and the problem is stated as finding a function $u(t),\ t_0 \leq t \leq t_f$, to minimize (or maximize)

$$J = \phi[x(t_f)]$$

subject to

$$\dot{x}(t) = f[x(t), u(t), t]$$

with $x(0)$, t_0 and t_f specified.

We will be utilizing the routines **fop0.m** and Matlab's routine **fminunc** in this assignment. The routine **fop0.m** solves continuous-time optimization problems of the form: find the input sequence $u(t),\ 0 \leq t \leq t_f$ to minimize

$$J = \phi[x(t_f)] + \int_0^{t_f} L[x(t), u(t), t]dt$$

subject to the constraints

$$\begin{aligned} \dot{x}(t) &= f[x(t), u(t), t] \\ x(0) &= x_0 \ (known) \end{aligned}$$

and t_f are known. Note that the program assumes the initial time t_0 is zero! Matlab's routine **fminunc** can be made to solve the same problem, does not require knowledge of derivatives, and is generally a bit more robust, but we have to do a little work first.
We will first give examples using **fop0.m**, then the same examples using **fminunc**.

Note that we cannot put any *hard* terminal constraints on this problem. That is, we cannot force $x(t_f)$ to be anything in particular. We will get to that next week!

Using **fop0.m**

The program **fop0.m** utilizes the Euler-Lagrange equations in combination with a gradient search algorithm to numerically solve the problem. This algorithm also takes into account the gradients of the functions. It is often easier to make this routine find the minimum than the **fminunc** routine. In order to use the routine **fop0.m**, you need to write a routine that returns one of three things depending on the value of the variable **flg**. Note that **fop0.m** also uses the routines **fop0_f.m** and **fop0_b.m**. The general form of your routine will be as follows:

```
function [f1,f2] = bobs_fop0(u,s,t,flg)
```

Here u is the current input, $u(t)$, and s contains the current state (including the augmented state), $s(t)$, so $\dot{s}(t) = f(s(t), u(t), t)$. t is the current time. Your routine should compute the following:

$$\begin{aligned} &\text{if } \mathbf{flg} = 1 \quad \text{f1} = \dot{s}(t) = f(s(t), u(t), t) \\ &\text{if } \mathbf{flg} = 2 \quad \text{f1} = \bar{\phi}[\bar{x}(t_f)] = \bar{\phi}[s(t_f)], \text{f2} = \bar{\phi}_s[\bar{x}(t_f)] = \bar{\phi}[s(t_f)] \\ &\text{if } \mathbf{flg} = 3 \quad \text{f1} = f_s, \ \text{f2} = f_u \end{aligned}$$

An example of the usage is:

```
[tu,ts,la0] = fop0('bobs_fop0',tu,tf,s0,k,told,tols,mxit)
```

The (input) arguments to **fop0.m** are the following:

- the function you just created (in single quotes).
- tu is an initial guess of times (first column), and control values (subsequent columns) that minimizes J. If there are multiple control signals at a given time, they are all in the same row. Note that these are just the initial time and control values, the times and control values will be modified as the program runs. The initial time should start at zero.

- the initial states, $s0$. Note that you must include and initial guess for the "cumulative" state q also.
- the final time, tf.
- k, the step size parameter, $k > 0$ to minimize. Often you need to play around with this one.
- $told$, the tolerance (a stopping parameter) for ode23 (differential equation solver for Matlab)
- $tols$, the tolerance (a stopping parameter); when $|\Delta u| < tols$ between iterations, the programs stops.
- $mxit$, the maximum number of iterations to try.

fop0.m returns the following:

- tu the optimal input sequence and corresponding times. The first column is the time, the corresponding columns are the control signals. All entries in one row correspond to one time.
- ts the states and corresponding times. The first column is the time, the corresponding columns are the states. All entries in one row correspond to the same time. Note that the times in tu and the times in ts may not be the same, and they may not be evenly spaced.
- $la0$ the Lagrange multipliers

It is usually best to start with a small number of iterations, like 5, and see what happens as you change k. Start with small values of k and gradually increase them. It is also generally better to start with a fairly small number of time steps, and then increase the time steps once you have found an acceptable solution for the reduced number of time steps. It can be difficult to make this program converge, especially if your initial guess is far away from the true solution.

Note!! If you are using the **fop0.m** file, and you use the maximum number of allowed iterations, assume that the solution has NOT converged. You must usually change the value of k and/or increase the number of allowed iterations. Try to make k as large as possible and still have convergence.

Example A From Homework 4, consider the problem

$$\begin{aligned} \text{minimize } J &= \frac{1}{2}(x(t_f)^2 + y(t_f)^2) \\ \text{subject to} \quad & \dot{x}(t) = t + u(t) \\ & \dot{y}(t) = x(t) \end{aligned}$$

where $x(0) = 1$, $y(0) = 1$, $t_f = 0.5$.

Let's define the state variables as

$$s = \begin{bmatrix} x \\ y \end{bmatrix}$$

Note that this problem is already in Mayer form, so $q = 0$. We then have

$$\dot{s}(t) = f(s) = \begin{bmatrix} t+u \\ x \end{bmatrix}$$

and

$$\begin{aligned} \bar{\phi}[\bar{x}(t_f)] &= \bar{\phi}[s(t_f)] = 0.5(x^2 + y^2) \\ \bar{\phi}_s[s] &= \begin{bmatrix} x & y \end{bmatrix} \\ f_s &= \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} \\ f_u &= \begin{bmatrix} 1 \\ 0 \end{bmatrix} \end{aligned}$$

This is implemented in the routine **bobs_fop0_a.m** on the class web site, and it is run using the driver file **fop0_example_a.m**.

Example B Consider the *tracking* problem

$$\begin{aligned} \text{minimize } J &= \frac{1}{2}\left(x_1(t_f) - r(t_f)\right)^2 Q_f + \int_0^{t_f} \left\{ \frac{1}{2}\left(x_1(t) - r(t)\right)^2 Q + \frac{1}{2} R u^2(t) \right\} dt \\ \text{subject to} \quad & \dot{x}(t) = Ax + Bu \end{aligned}$$

where $x(0) = 0$, t_f, Q_f, Q, and R are known, and $r(t)$ is a signal we want to track (follow). For convenience we will assume there are two states.

Let's define the state variables as

$$s = \begin{bmatrix} x_1 \\ x_2 \\ q \end{bmatrix}$$

Where

$$\dot{q}(t) = \frac{1}{2}\left(x_1(t) - r(t)\right)^2 Q + \frac{1}{2} R u^2(t)$$

We have then

$$\dot{s}(t) = f(s) = \begin{bmatrix} Ax + Bu \\ \frac{1}{2}\left(x_1(t) - r(t)\right)^2 Q + \frac{1}{2} R u^2(t) \end{bmatrix}$$

and

$$\begin{aligned}
\bar{\phi}[\bar{x}(t_f)] &= \bar{\phi}[s(t_f)] = \frac{1}{2}Q_f(x_1 - r)^2 + q \\
\bar{\phi}_s[s] &= \begin{bmatrix} Q_f(x_1 - r) & 0 & 1 \end{bmatrix} \\
f_s &= \begin{bmatrix} A & 0 \\ Q(x_1 - r) & 0 & 0 \end{bmatrix} \\
f_u &= \begin{bmatrix} B \\ Ru \end{bmatrix}
\end{aligned}$$

This is implemented in the routine **bobs_fop0_b.m** on the class web site, and it is run using the driver file **fop0_example_b.m**.

Using **fminunc**

Matlab's routine **fminunc** (*find the minimum of an unconstrained multivariable function*) is a routine that finds the minimum of a function $J(u)$, assuming there are no constraints on u. For this routine you do not need to determine the step size, and you do not need to have knowledge of derivatives either. It is a more general routine than **fop0.m**, but that also makes it a little more awkward to use. Since it does not solve the Euler-Lagrange equations or use derivative information, it can be very difficult to find a solution unless you have a good idea of the solution before you begin!

This routine is very well suited to the *Mayer* form of the problem, however. We need to be able to write our function

$$J(x) = \phi[x(t_f)] + \int_0^{t_f} L\left[x(t), u(t), t\right] dt$$

as

$$J(\bar{x}) = \phi[x(t_f)] + q(t_f) = \bar{\phi}[\bar{x}(t_f)] = \bar{\phi}[s]$$

where s is governed by the dynamic equations

$$\dot{s}(t) = f(s(t), u(t), t)$$

In order to determine the final values, we will need to integrate the equations forward in time using a numerical integration routine.

Example A (Again) Consider the problem

$$\begin{aligned}
\text{minimize } J &= \frac{1}{2}(x(t_f)^2 + y(t_f)^2) \\
\text{subject to} \quad & \dot{x}(t) = t + u(t) \\
& \dot{y}(t) = x(t)
\end{aligned}$$

where $x(0) = 1$, $y(0) = 1$, $t_f = 0.5$.

Let's define the state variables as

$$s = \begin{bmatrix} x \\ y \end{bmatrix}$$

Note that this problem is already in Mayer form, so $q = 0$. We then have

$$\dot{s}(t) = f(s) = \begin{bmatrix} t+u \\ x \end{bmatrix}$$

and

$$\bar{\phi}[\bar{x}(t_f)] = \bar{\phi}[s(t_f)] = 0.5(x^2 + y^2)$$

This problem is solved using the code **fminunc_example_a.m** which is available on the class website. This is the basic file that sets everything up, invokes **fminunc** with the function (**fminunc_a**) that will compute $\bar{\phi}$, and then plots the results. The routine **fminunc_a** utilizes one of Matlab's build in functions for integrating a differential equation forward in time (we used **ode45**, there are other choices depending on your differential equation). The function **ode_a** contains the state variable description of the equation we need to solve. Note that the arrays *times* and *ut* are optional arrays we pass to the function. They contain all of the time values and the corresponding control values. When Matlab calls this routine using its solver (**ode45** in this case) it only passes one time value (t) and the corresponding state value (s). In order to determine which value of u to use we need to interpolate. Finally, once the problem has bee solved, the differential equation is solved once more so we can plot the optimal control vector and the corresponding states. **Note:** If **fminunc** claims it has not converged, often you need to change the tolerance, *tolfun*.

Example B (Again) Consider the *tracking* problem

$$\begin{aligned} \text{minimize } J &= \frac{1}{2}\left(x_1(t_f) - r(t_f)\right)^2 Q_f + \int_0^{t_f} \left\{ \frac{1}{2}\left(x_1(t) - r(t)\right)^2 Q + \frac{1}{2} R u^2(t) \right\} dt \\ \text{subject to} \quad & \dot{x}(t) = Ax + Bu \end{aligned}$$

where $x(0) = 0$, t_f, Q_f, Q, and R are known, and $r(t)$ is a signal we want to track (follow). For convenience we will assume there are two states.

Let's define the state variables as

$$s = \begin{bmatrix} x_1 \\ x_2 \\ q \end{bmatrix}$$

Where

$$\dot{q}(t) = \frac{1}{2}\left(x_1(t) - r(t)\right)^2 Q + \frac{1}{2} R u^2(t)$$

We have then

$$\dot{s}(t) \;=\; f(s) = \begin{bmatrix} Ax + Bu \\ \frac{1}{2}\left(x_1(t) - r(t)\right)^2 Q + \frac{1}{2} R u^2(t) \end{bmatrix}$$

and

$$\bar{\phi}[\bar{x}(t_f)] \;=\; \bar{\phi}[s(t_f)] = \frac{1}{2} Q_f (x_1 - r)^2 + q$$

This problem is solved using the code **fminunc_example_b.m** which is available on the class website. The only thing different is that we need to again use interpolation to determine the corresponding value of $r(t)$ in the routine **ode_b**.

Interpolation

For both methods of solving the problem, we utilize a method of interpolation as we refine the number of sample points. The basic idea is as follows:

- Try and initially solve the problem using only a few sample points, say between 10 and 20 for most problems.
- Guess an initial (optimal) control sequence, as well as other parameters.
- After each optimization attempt, the resulting times and control values are saved to a file (*old_ut*).
- Once you have results that seems to be converging, comment out your initial guess and uncomment the part of the code that reads in *old_ut* and uses the time sequence and control sequence your last run ended with.
- Once you have a solution for a few sample points, use the existing solution to interpolate an initial solution for more points. Usually this works fastest if your new number of sample points is limited to twice the original number of sample points.
- Note that if your solution starts to diverge, stop the program before it writes to the file by hitting (control c). If your computer is not hung up this will stop your code.
- Right now the program is set up to show your original and the new interpolated points.
- Both of the programs have been set up to write and interpolate the solutions the same way. However, the **fop0.m** routine determines the number of sample points it will ultimately end up with, while we have set up **fminunc** to use the number of sample points we choose.
- **fop0.m** is a pretty good choice for finding an initial solution estimate, even if if has not completely converged.