# ME 149: Midterm Exam Solution

Optimal Control for Robotics

March 15 — Start: 6:00pm — End: 7:15pm

No calculators, notes, books, or computers allowed. Total time: 75 minutes.

## Student Name: SOLUTION

## How to optimize your score?

- Be neat and well organized
- For longer problems, show intermediate steps and box your answer
- If you need extra space... use the back of the page, the final (blank) page of the exam, or ask for more paper. Clearly indicate where the extra work is.
- Define all variables that you use and state any assumptions that you make.

## Score: _____ / _____

1. _____ / _____
2. _____ / _____
3. _____ / _____
4. _____ / _____
5. _____ / _____
6. _____ / _____
7. _____ / _____
8. _____ / _____
9. _____ / _____

# 1 Newton's Method

A. Suppose that you want to solve the scalar nonlinear equation $f(x) = 0$.
The current estimate of the root is given by $x_k$ and the next estimate of the root is given by $x_{k+1}$. Derive the Newton-Rhapson update that computes $x_{k+1}$ given $x_k$.

| | |
|---|---|
| $f(x) = f(x_0) + \Delta x \cdot f'(x_0) + \mathcal{O}(\Delta x^2)$ | Taylor series approx. for $f(x)$ near $x_0$ |
| $f(x) \approx f(x_0) + \Delta x \cdot f'(x_0)$ | Let high order terms $\mathcal{O}(\Delta x^2) \to 0$ |
| $0 = f(x_0) + \Delta x \cdot f'(x_0)$ | Set linear approximation equal to zero |
| $0 = f(x_k) + (x_{k+1} - x_k) \cdot f'(x_k)$ | Replace $x_0$ with $x_k$ and $\Delta x$ with $x_{k+1} - x_k$ |
| $x_{k+1} = x_k - \dfrac{f(x_k)}{f'(x_k)}$ | Solve for $x_{k+1}$ |

B. What is the difference between the Newton–Rhapson and the secant methods for scalar root finding? In what situation would you prefer one to the other?

*Both methods work by computing the root of a local linear approximation of the nonlinear function. The two differ in the way that this linear approximation is constructed. Newton's method uses the analytic derivative of the function to construct the linear model. The secant method uses the function value at the current and previous point to construct the linear model — it uses a finite difference equation.*

*The Newton–Rhapson method for root finding is prefered to the secant method when the analytic derivative is known: it will converge more rapidly than the secant method. The secant method is used when the analytic derivative is unknown.*

C. Draw a figure that clearly demonstrates a situation in which Newton's method will fail to converge to a root of the function. Why does the Newton–Rhapson method fail in this situation?

*The Newton–Rhapson method for root finding can fail to converge in a variety of situations. One example is shown in Figure 1. Both the top and bottom sub-plots show the method applied to the hyperbolic tangent function. A small change in the initialization makes the difference between convergence (top plot) and divergence (bottom plot). When diverging, each iteration moves the current approximation of the root farther from the true root, until the approximation (in this case) reaches infinity. The method will also fail in situations where there is no root, or where there is a local extremum where the function does not cross the x-axis. In either situation the solver will get stuck in a limit cycle around the extremum. Finally, if the solver every finds a point where the slope of the function is tiny (or zero) then it will extrapolate to near-infinity, which almost always will cause the method to fail. One key insight is that the convergence of the Newton–Rhapson method depends entirely on the initial condition! You can typically find an initial guess that will converge to the root, although it might be very difficult to do so.*

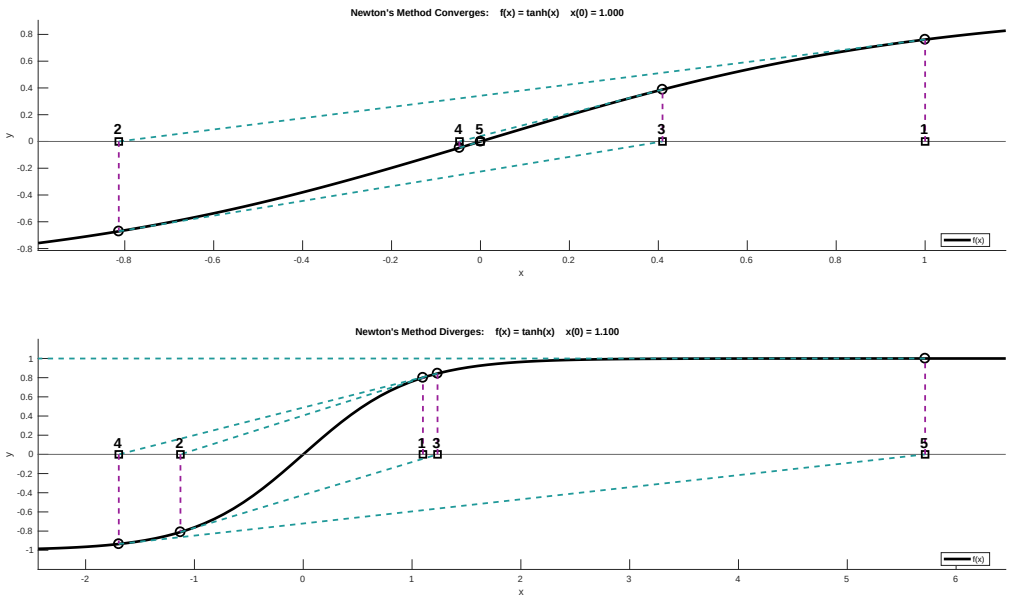


Figure 1: Stability of Newton–Rhapson method for root finding

## 2 Midpoint Method Implementation

Implement the function `simStepMidpoint()` on the following page. This function computes a single simulation step using the midpoint method. You code should be clear, correct, and follow the best practices that have been discussed throughout the course. Use the space below for planning your solution. Write your Matlab code inside of the function template on the following page.

```
function [zNext, nEval] = simStepMidpoint(dynFun, tPrev, tNext, zPrev)
% [zNext, nEval] = simStepMidpoint(dynFun, tPrev, tNext, zPrev)
%
% Compute a single integration step using the midpoint method.
%
% INPUTS:
%    dynFun = a function handle:  dz = dynFun(t, z)
%        IN:  t = [1, nTime] = row vector of time
%        IN:  z = [nState, nTime] = matrix of states at times t
%        OUT: dz = [nState, nTime] = time-derivative of z
%    tPrev = scalar = previous time grid point
%    tNext = scalar = next time grid point
%    zPrev = [nDim, 1] = state at tPrev
%
% OUTPUTS:
%    zNext = [nDim, 1] = state at tNext
%    nEval = scalar = numer of internal calls to dynamics function
%


% time-step for the integration method
h = tNext - tPrev;

% Compute the state estimate at the midpoint, using Euler step
tMid = tPrev + 0.5 * h;
zMid = zPrev + 0.5 * h * dynFun(tPrev, zPrev);

% Evaluate the dynamics at the midpoint state estimate:
dzMid = dynFun(tMid, zMid);

% Compute the state at the next point, using dynamics at the midpoint:
zNext = zPrev + h * dzMid;

% There are always two function evaluations in the midpoint method
nEval = 2;

end
```

## 3 Bisection Search

Use a bisection search to iteratively reduce the interval that is known to bracket the root of the function shown in Figure 2. **Populate the table below**, showing the bracket for the first five iterations and the new point that will be evaluated on that iteration.

- `Iter 0:  bracket:  [-1.000,  1.000]  xNew = 0.0`
- `Iter 1:  bracket:  [-1.000,  0.000]  xNew = -0.5`
- `Iter 2:  bracket:  [-0.500,  0.000]  xNew = -0.25`
- `Iter 3:  bracket:  [-0.500,  -0.250]  xNew = -0.375`
- `Iter 4:  bracket:  [-0.375,  -0.250]  xNew = -0.3125`

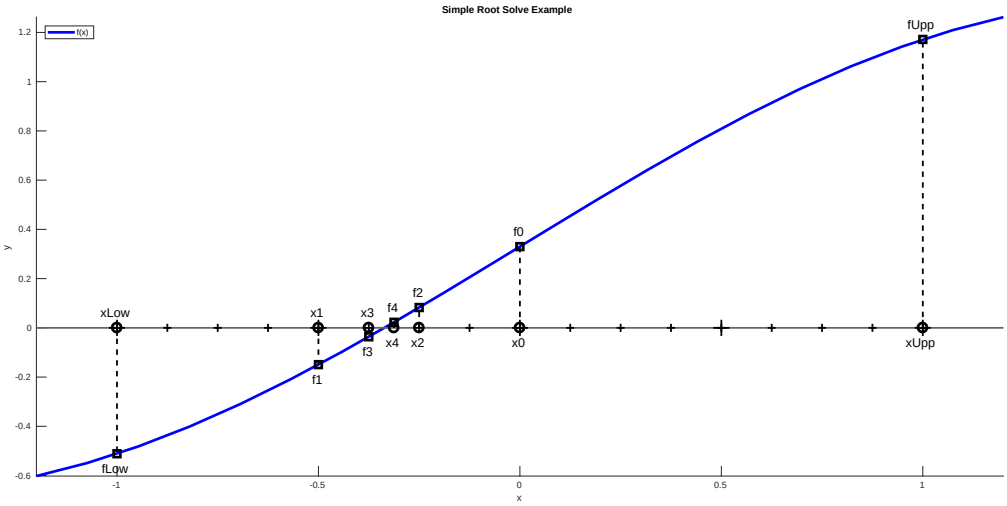


Figure 2: Bisection Search Example Figure

## 4 Scalar Taylor Series

Write the Taylor series approximation of $f(t)$ to second-order around the point $t_0$.

$$f(t) \approx f(t_0) + f'(t_0) \cdot (t - t_0) + f''(t_0) \cdot \tfrac{1}{2}(t - t_0)^2$$

$$f'(t_0) = \frac{d}{dt} f(t)\Big|_{t=t_0} \qquad\qquad f''(t_0) = \frac{d^2}{dt^2} f(t)\Big|_{t=t_0}$$

## 5 Vector Taylor Series

Write the Taylor series approximation of $f(t, \boldsymbol{x}, \boldsymbol{u})$ to first-order about the point: $t_0$, $\boldsymbol{x}_0$, and $\boldsymbol{u}_0$.

$$f(t, \boldsymbol{x}, \boldsymbol{u}) \approx f(t_0, \boldsymbol{x}_0, \boldsymbol{u}_0) + f_t(t_0, \boldsymbol{x}_0, \boldsymbol{u}_0) \cdot (t - t_0) + f_x(t_0, \boldsymbol{x}_0, \boldsymbol{u}_0) \cdot (\boldsymbol{x} - \boldsymbol{x}_0) + f_u(t_0, \boldsymbol{x}_0, \boldsymbol{u}_0) \cdot (\boldsymbol{u} - \boldsymbol{u}_0)$$

$$f_t(t_0, \boldsymbol{x}_0, \boldsymbol{u}_0) = \frac{\delta}{\delta t} f(t, \boldsymbol{x}, \boldsymbol{u})\Big|_{t=t_0, \boldsymbol{x}=\boldsymbol{x}_0, \boldsymbol{u}=\boldsymbol{u}_0}$$

$$f_x(t_0, \boldsymbol{x}_0, \boldsymbol{u}_0) = \frac{\delta}{\delta x} f(t, \boldsymbol{x}, \boldsymbol{u})\Big|_{t=t_0, \boldsymbol{x}=\boldsymbol{x}_0, \boldsymbol{u}=\boldsymbol{u}_0}$$

$$f_u(t_0, \boldsymbol{x}_0, \boldsymbol{u}_0) = \frac{\delta}{\delta u} f(t, \boldsymbol{x}, \boldsymbol{u})\Big|_{t=t_0, \boldsymbol{x}=\boldsymbol{x}_0, \boldsymbol{u}=\boldsymbol{u}_0}$$

# 6 Function Handle Gymnastics

**What will the following script print out?**
For each part A, B, C, show your work and then box what will be printed.

```
%% Set up the workspace
a = 1; b = 2; c = 3; d = 5; e = 7;
trigFun = @(t)( b + sin(a + t) );
funTimes = @(x, y)( d * x + b );
addFun = @(x, z)( [x + c; z - b] );
chainFun = @(y)( sum(addFun(y, e)) );
printFun = @(a, b)( fprintf('%s: %d\n', a, b) );
%% PART A:
A = trigFun(-a);    printFun('A', A);
%% PART B:
B = funTimes(b, d);    printFun('B', B);
%% PART C:
C = chainFun(c);    printFun('C', C);
```

### Part A:

```
A: 2
```

### Part B:

```
B: 12
```

### Part C:

```
C: 11
```

# 7 Matlab Programming Style

The program below simulates a simple pendulum. It works, but is written poorly.
**Clearly identify at least 5 distinct issues with the code.**

```
1  function runSloppySimulation()
2  tLow = -0.15;  % start time
3  tUpp = 2.9234;  % stop time
4  dynFun = @(t, z)( [z(2); -sin(z(1))] );
5  zInit = [-0.2; 0.6];
6  [tGrid, zGrid] = runSimBadly(dynFun, tLow, tUpp, zInit);
7  plot(tGrid, zGrid);
8  end
9
10 function [tGrid, zGrid] = runSimBadly(dynFun, tLow, tUpp, zInit)
11 tGrid = tLow:0.1:tUpp;
12 zGrid = zeros(2)
13 zGrid = zInit;
14 nDim = length(zInit);
15 for i = 2:31
16     dz = dynFun(tGrid(i), zGrid(:, i-1))
17     zGrid(:, i) = zGrid(:, i-1) + 0.1 * dz;
18 end
19 end
```

- **Line 4:** arguments `@(t, z)` should be replaced with `@(~, z)`
- **Line 7:** plotting should include annotations
- **Line 17:** hard-coded time step — should be a parameter
- **Line 11:** use `linspace` instead — as written `tGrid(end)` is not `tUpp`
- **Line 12:** missing semicolon
- **Line 12-13:** as written, line 12 does nothing and line 13 allocates a matrix of the wrong size. These lines should be replaced by a correctly-written memory allocation and then setting the first column to the initial state.
- **Line 14:** `nDim` is unused — this line should be deleted
- **Line 15:** upper loop limit should not be hard-coded
- **Line 16:** missing semicolon
- **Line 17:** hard-coded time step — should be related to tGrid

# 8 Linearized Dynamical System

Given the non-linear system dynamics described below,

$$\dot{\boldsymbol{z}} = \begin{bmatrix} \dot{\alpha} \\ \dot{\gamma} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \beta\gamma + \theta^2 \\ \alpha\,\sigma - \theta \\ \sin(\sigma) + \beta\,\gamma \end{bmatrix} = \boldsymbol{f}(\boldsymbol{z}, \boldsymbol{u}) \qquad \boldsymbol{z} = \begin{bmatrix} \alpha \\ \gamma \\ \theta \end{bmatrix} \qquad \boldsymbol{u} = \begin{bmatrix} \beta \\ \sigma \end{bmatrix}$$

A. **List the state variables:** $\alpha,\ \gamma,\ \theta$

B. **List the control variables:** $\beta,\ \sigma$

C. **What is the difference between a state and a control variable?**
*State variables are those variables that have derivatives in the system dynamics. For example, both $\alpha$ and $\dot{\alpha}$ show up in the dynamics above. Control variables appear algebraically in the system dynamics. For example, $\sigma$ is in the system dynamics, but not $\dot{\sigma}$.*

D. **Compute:** $\dfrac{\delta \boldsymbol{f}}{\delta \boldsymbol{z}}$

$$\frac{\delta \boldsymbol{f}}{\delta \boldsymbol{z}} = \begin{bmatrix} 0 & \beta & 2\theta \\ \sigma & 0 & -1 \\ 0 & \beta & 0 \end{bmatrix}$$

E. **Compute:** $\dfrac{\delta \boldsymbol{f}}{\delta \boldsymbol{u}}$

$$\frac{\delta \boldsymbol{f}}{\delta \boldsymbol{u}} = \begin{bmatrix} \gamma & 0 \\ 0 & \alpha \\ \gamma & \cos\sigma \end{bmatrix}$$

# 9 Trajectory Optimization

Given the following continuous-time trajectory optimization problem.

$$\begin{aligned} &\text{minimize:} && J = \int_0^T g(t,\, \boldsymbol{x},\, \boldsymbol{u})\, dt \\ &\text{subject to:} && \mathbf{0} = \boldsymbol{h}(\boldsymbol{x}(0),\, \boldsymbol{x}(T)) \\ &\text{dynamics:} && \dot{\boldsymbol{x}} = \boldsymbol{f}(t,\, \boldsymbol{x},\, \boldsymbol{u}) \end{aligned}$$

Suppose that you plan to solve the optimization using direct multiple shooting with Euler's method on a uniform grid of $N$ segments with one integration step per segment. Write out the decision variables, objective function, and constraints that will form the resulting non-linear program.

A. **Decision Variables**

$$\boldsymbol{x}_0, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_N, \quad \boldsymbol{u}_0, \boldsymbol{u}_1, \ldots, \boldsymbol{u}_{N-1}$$

B. **Objective Function**

$$J = h \cdot \sum_{k=0}^{N-1} g(t_k,\, \boldsymbol{x}_k,\, \boldsymbol{u}_k)$$

$$h = \frac{T}{N} \qquad\qquad t_k = k \cdot h$$

C. **Boundary Constraints**

$$\mathbf{0} = \boldsymbol{h}(\boldsymbol{x}_0,\, \boldsymbol{x}_N)$$

D. **System Dynamics Constraints**

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + h \cdot \boldsymbol{f}(t_k,\, \boldsymbol{x}_k,\, \boldsymbol{u}_k) \qquad k \in 0 \ldots (N-1)$$