MATH 3315/CSE 3365 : FINAL EXAM (DUE DECEMBER 18, 2024, 11:59PM)

COURSE INSTRUCTOR: JIMMIE ADRIAZOLA

Problem 1: (40 points). For this problem, you will design a variable step-size, fourth order explicit time stepping scheme for solving initial value problems. A sophisticated way to choose the appropriate step size follows from knowledge of the order of the ODE solver. Assume that the solver has order p, so that the local truncation error $e_n = O\left(h^{p+1}\right)$. Let ε be the relative error tolerance allowed by the user for each step. That means the goal is to ensure $e_n/\left|y_n\right| < \varepsilon$.

If the goal $e_n/\left|y_n\right| < \varepsilon$ is met, then the step is accepted and a new step size for the next step is needed. Perhaps this step size is small and we can get away with taking a bigger step. To figure out if we can be more lenient with the stepsize, assume that

$$e_n = ch_n^{p+1} + O\left(h^{p+2}\right)$$

for some constant c, the step size h that best meets the tolerance satisfies

$$\varepsilon\left|y_n\right| = ch^{p+1}.$$

Let's assume we have some estimate for e_n. We've already computed what h_n and y_n are and the tolerance ε is user-specified. This leaves us with two equations, consistent at $\mathcal{O}(h^{p+1})$, for the two unknowns h and c. Solving for h and calling it h_n^* gives us

$$h_n^* = \left(\frac{\varepsilon\left|y_n\right|}{e_n}\right)^{\frac{1}{p+1}} h_n$$

Thus, the next step size will be set to $h_{n+1} = h_n^*$. If the goal $e_n/\left|y_n\right| < \varepsilon$ is not met by the relative error, then the step size is simply cut in half. This is not the most widely accepted practice, but it will suffice for this assignment.

The method just described depends heavily on some way to estimate the error of the current step of the ODE solver $e_n = \left|y_{n+1} - y_{n+1}^{\text{true}}\right|$. An important constraint is to gain the estimate without requiring a large amount of extra computation. The most widely used way for obtaining such an error estimate is to run a higher order ODE solver in parallel with the ODE solver of interest. The higher order method's estimate for y_{n+1}, call it z_{n+1}, will be significantly more accurate than the original y_{n+1}, so that the difference

$$e_{n+1} \approx \left|z_{n+1} - y_{n+1}\right|$$

is used as an absolute error estimate for the current step from t_n to t_{n+1}. In this case, the stepsize we use as an update is given by

$$h_n^* = \left(\frac{\varepsilon}{\left|z_{n+1} - y_{n+1}\right|}\right)^{\frac{1}{p+1}} h_n$$

We will use a fourth-order/fifth-order embedded pair method that uses the following coefficients:

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{1}{4} & 0 & 0 & 0 & 0 & 0 \\ \frac{3}{32} & \frac{9}{32} & 0 & 0 & 0 & 0 \\ \frac{1932}{2197} & -\frac{7200}{2197} & \frac{7296}{2197} & 0 & 0 & 0 \\ \frac{439}{216} & -8 & \frac{3680}{513} & -\frac{845}{4104} & 0 & 0 \\ -\frac{8}{27} & 2 & -\frac{3544}{2565} & \frac{1859}{4104} & -\frac{11}{40} & 0 \end{bmatrix}$$

$$B = \left[\tfrac{25}{216}, 0, \tfrac{1408}{2565}, \tfrac{2197}{4104}, -\tfrac{1}{5}, 0\right]$$

$$C = \left[\tfrac{16}{135}, 0, \tfrac{6656}{12825}, \tfrac{28561}{56430}, -\tfrac{9}{50}, \tfrac{2}{55}\right]$$

$$D = \left[0, \tfrac{1}{4}, \tfrac{3}{8}, \tfrac{12}{13}, 1, \tfrac{1}{2}\right]$$

The stages are computed as:

$$k_i = f\left(t_n + D_i h, y_n + h\sum_{j=1}^{i-1} a_{ij}k_j\right)$$

The fourth- and fifth-order solutions are:

$$y_{n+1} = y_n + h\sum_{i=1}^{6} B_i k_i, \quad z_{n+1} = y_n + h\sum_{i=1}^{6} C_i k_i$$

If the error exceeds the tolerance, the step size is reduced; otherwise, the step is accepted, and the step size may be increased. To ensure the stepsize doesn't get too coarse or too refined, we set a maximum/minimum stepsize $h_{\max}$ and $h_{\min}$. To summarize the steps you need to take to build this variable step size method:

1) Set an initial stepsize h_0, the initial condition y_0, and the final time T.
2) Compute y_1 via the fourth-order method.
3) Compute z_1 via the fifth-order method.
4) Compute the absolute error $e_1 = \max|z_1 - y_1|$. The maximum is taken over the components of the vector $|z_1 - y_1|$ whose size depends on the number of ODEs in your system.
5) If the error $e_1 > \varepsilon$ and $h_0 > h_{\min}$, set $h_0 = \max(h_{\min}, h_0/2)$ and go back to step 2. Else, keep the stepsize h_0 as is and proceed.
6) Accept y_1 as the numerical approximation of the IVP and update the time variable via $t_1 = t_0 + h_0$.
7) If $e_1 < \varepsilon/10$, then the current stepsize is too small. Set $h_1 = \min(h_{\max}, \alpha h_0^*)$, where α is a safety factor. Otherwise, set $h_1 = h_0$.
8) Repeat from step 2 and iterate through until $t_n > T$.

For the following case studies, I suggest you use a tolerance of $\varepsilon = 10^{-6}$, $h_{\min} = 10^{-6}$, $h_{\max} = (T - t_0)/2$, and safety factor $\alpha = 0.8$.

Case Study 1: The Hodgkin-Huxley Model

The Hodgkin-Huxley model describes the electrical behavior of a neuron. The equations are

$$\begin{aligned} C_m \frac{dV}{dt} &= -I_{Na} - I_K - I_L + I_{ext}, \\ \frac{dm}{dt} &= \alpha_m(V)(1-m) - \beta_m(V)m, \\ \frac{dh}{dt} &= \alpha_h(V)(1-h) - \beta_h(V)h, \\ \frac{dn}{dt} &= \alpha_n(V)(1-n) - \beta_n(V)n \end{aligned}$$

The ionic currents are given by

$$I_{Na} = G_{Na} m^3 h (V - E_{Na}), \quad I_K = G_K n^4 (V - E_K), \quad I_L = G_L (V - E_L),$$

while rate functions are given by

$$\alpha_m(V) = \frac{0.1(V+40)}{1-\exp(-(V+40)/10)}, \quad \beta_m(V) = 4\exp(-(V+65)/18)$$

$$\alpha_h(V) = 0.07\exp(-(V+65)/20), \quad \beta_h(V) = \frac{1}{1+\exp(-(V+35)/10)}$$

$$\alpha_n(V) = \frac{0.01(V+55)}{1-\exp(-(V+55)/10)}, \quad \beta_n(V) = 0.125\exp(-(V+65)/80)$$

As can be seen, this is a highly nonlinear set of equations.

Let's simulate the dynamics of this system of ODEs using the embedded order 4/5 method described above. Let's set the initial conditions to

$$V = -65, \quad m = 0.05, \quad h = 0.6, \quad n = 0.32.$$

Let's also set the external current to

$$I_{ext} = 10 \ \mu A/\text{cm}^2,$$

and the constants $C_m = 1.0, G_{Na} = 120.0, G_K = 36.0, G_L = 0.3, E_K = -77.0, E_L = -54.387$. Using this setup, you should observe what's being shown in Figure 1. You may use the code at the end of this document to reproduce these neuron spiking dynamics.

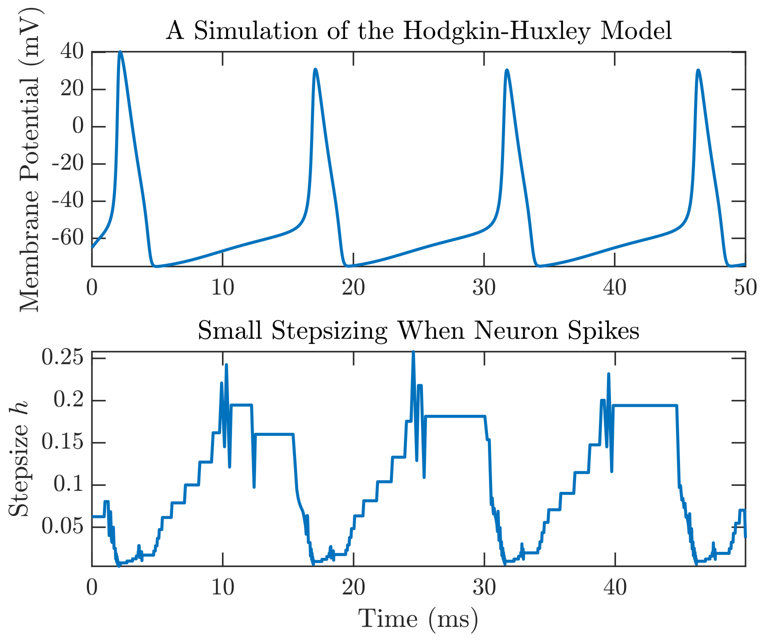

FIGURE 1.

CASE STUDY 2: THE LORENZ SYSTEM

This model is a famous system of equations in chaos theory. The basic idea is that it is a simplified model of Rayleigh-Benard convection in a vertically oriented circular tube. The simplified system of equations used to model the convection in the tube are given by

$$\frac{\mathrm{d}x}{\mathrm{d}t} = \sigma(y - x)$$
$$\frac{\mathrm{d}y}{\mathrm{d}t} = x(\rho - z) - y$$
$$\frac{\mathrm{d}z}{\mathrm{d}t} = xy - \beta z$$

The physical meaning of the variables and parameters are unimportant for this assignment, but is something you should seek out at some point in the future.

Using an initial condition of $x_0 = 0,\ y_0 = 1,\ z_0 = 20$ and parameters $\sigma = 10,\ \rho = 28,\ \beta = 8/3$, we see, in Figure 2, that the numerics generate an odd geometric shape called a strange attractor. Show that you are able to reproduce the strange attractor in Figure 2. You may use the code at the end of this document, too.

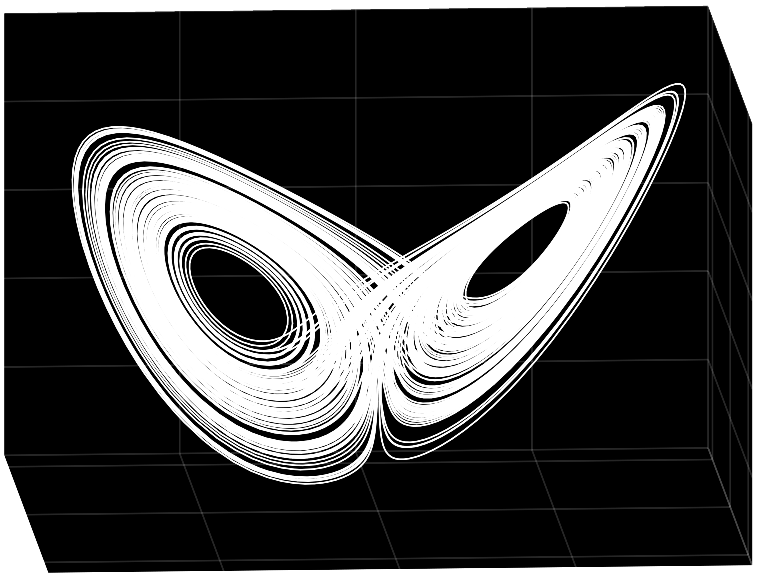

FIGURE 2. A Strange Attractor, indeed.

Problem 2: (30 points). Use the method of finite differences to discretize the nonlinear boundary value problem

$$\begin{cases} y'' = y' + \cos y, \\ y(0) = 0, \\ y(\pi) = 1. \end{cases}$$

To solve the resulting finite-dimensional system of nonlinear equations, use Newton's method. To invert the Jacobian, use MATLAB's backslash command. The numerical solution I found is displayed in Figure 3.

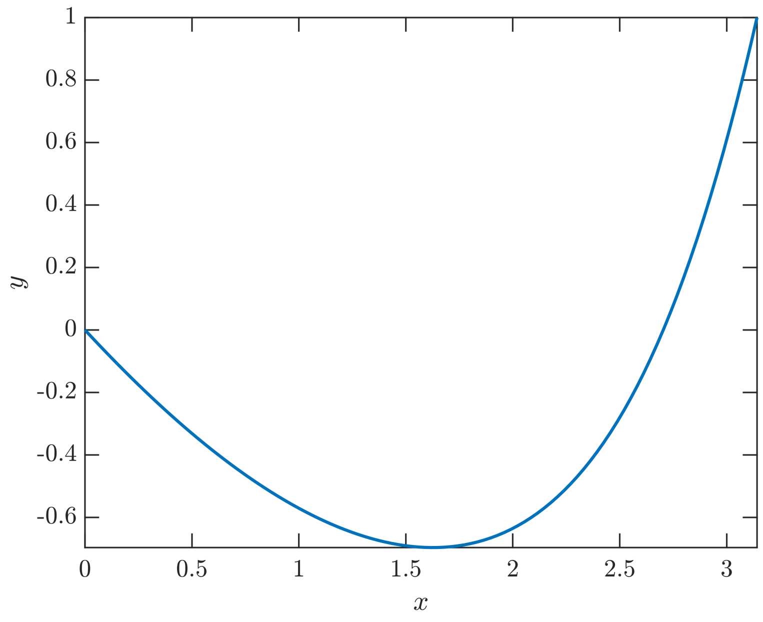

FIGURE 3. The numerical solution of the BVP, described in Problem 2, via a Finite-Differece Newton method with 200 spatial points.

Problem 3: (15 points). Use the following code to generate 200 random data points

```
n=200;
x=linspace(-2,2,n);
y=(x-2).*(x+1).*(x-1).*(x+2).*x;
y=y+4*rand(1,n);
```

These points clearly come from a fifth-order polynomial corrupted by noise. Let's recover the underlying fifth order polynomial using the method of linear least squares. We will guess that we can model the data using a monomial basis, that is,

$$y^{(N)}_{\text{model}} = \sum_{k=0}^{N} c_k x^k$$

Solve the normal equations to find the coefficients c_k for a given N. Use $N = 4,\ 5,\ 6,$ and 40. To invert matrices, use MATLAB's backslash command. After solving the normal equations, you should see something close to what is being displayed in Figure 4.

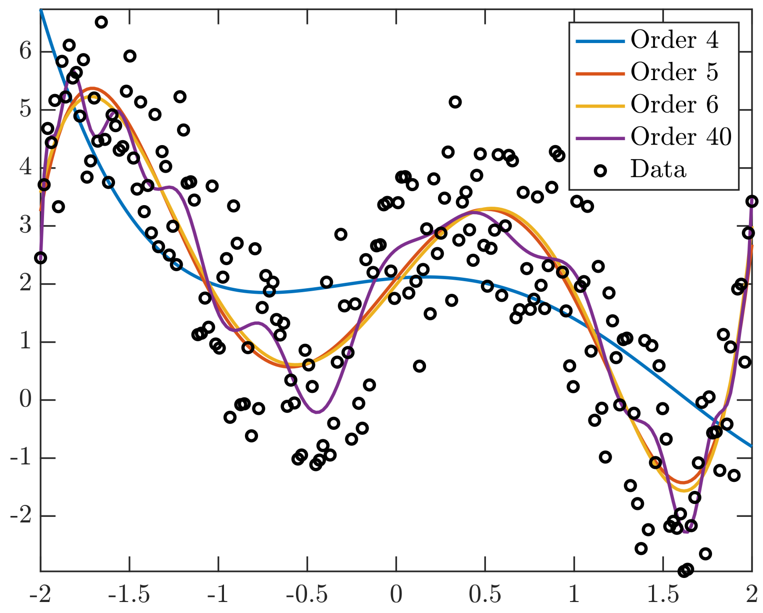

FIGURE 4. A solution of the Linear Least Squares Problem described in Problem 3. You should observe in your numerics that the 5th order model is the most parsimonious one that captures the behavior of the data (as expected).

Problem 4: (15 points). Use the method of gradient descent

$$z_{k+1} = z_k - \alpha \nabla_z J|_{z=z_k}$$

to solve the problem

$$\min_{(x,y)\in\mathbb{R}^2} J(x,y) = \min_{(x,y)\in\mathbb{R}^2} (1-x^2)^2 + (1-y^2)^2$$

This problem is nonconvex (it has four different local minima). Use random initial guesses and a fixed learning rate of $\alpha = 0.05$ to find all four minima. You may use the following code to interpret your results:

```
fprintf('The optimal point found is (%f,%f).\n',xvec(end),yvec(end))
fprintf('This run took %i iterations.\n',k)

% visualize the result
xx=linspace(-1.5,1.5,100);
[X,Y]=meshgrid(xx);
JJ=J(X,Y);

figure
contour(X,Y,JJ,100), shading interp, colorbar, axis equal
hold on, plot(xvec,yvec,'<-')
xlabel('$x$'), ylabel('$y$')
```

In the code, J is an anonymous function modeling the objective. Figure 5 shows that I was able to find one local minimum.

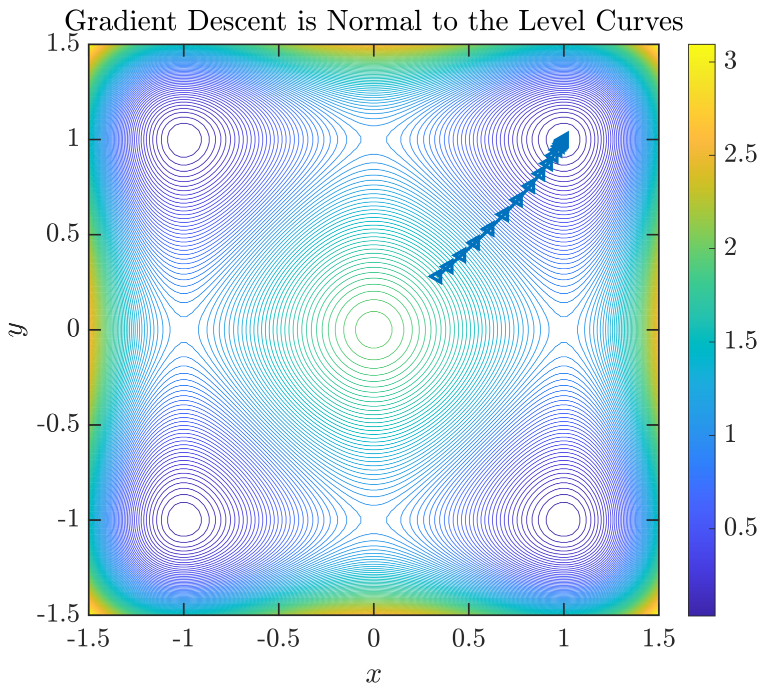

FIGURE 5. Discovery of one local minimum of the objective function $J = (1 - x^2)^2 + (1 - y^2)^2$.

Hodgkin-Huxley Code

```
%% Main Script
% Time Domain
tspan = [0, 50];

% Initial conditions [V, m, h, n]
y0 = [-65; 0.05; 0.6; 0.32];

% tolerance
tol = 1e-6;

% solve the problem using the method from Problem 1
[t, y] = ODESolver(@hodgkin_huxley, tspan, y0, tol);

% Plot results
subplot(2,1,1)
plot(t, y(:, 1))
xlabel('Time (ms)'), ylabel('Membrane Potential (mV)')
title('A Simulation of the Hodgkin-Huxley Model')
axis tight
subplot(2,1,2)
plot(t(1:end-1),diff(t))
xlabel('$t$'), ylabel('Stepsize $h$')
axis tight

%% This function builds the right-hand side
function dydt = hodgkin_huxley(t, y)
% HODGKIN_HUXLEY Implements the Hodgkin-Huxley model equations.
%
% INPUT:
%   t - Time (not used in the equations explicitly)
%   y - State vector [V, m, h, n]
%
% OUTPUT:
%   dydt - Time derivative of the state vector

% Constants
C_m = 1.0;  % Membrane capacitance (µF/cm^2)
G_Na = 120.0; % Maximum sodium conductance (mS/cm^2)
G_K = 36.0;  % Maximum potassium conductance (mS/cm^2)
G_L = 0.3;   % Leak conductance (mS/cm^2)
E_Na = 50.0; % Sodium reversal potential (mV)
E_K = -77.0; % Potassium reversal potential (mV)
E_L = -54.387; % Leak reversal potential (mV)
I_ext = 10.0; % External current (µA/cm^2)
```

```
% State variables
V = y(1); m = y(2); h = y(3); n = y(4);

% Ionic currents
I_Na = G_Na * m^3 * h * (V - E_Na);
I_K = G_K * n^4 * (V - E_K);
I_L = G_L * (V - E_L);

% Voltage equation
dVdt = (I_ext - I_Na - I_K - I_L) / C_m;

% Gating variable equations
alpha_m = 0.1 * (V + 40) / (1 - exp(-(V + 40) / 10));
beta_m = 4 * exp(-(V + 65) / 18);
dmdt = alpha_m * (1 - m) - beta_m * m;

alpha_h = 0.07 * exp(-(V + 65) / 20);
beta_h = 1 / (1 + exp(-(V + 35) / 10));
dhdt = alpha_h * (1 - h) - beta_h * h;

alpha_n = 0.01 * (V + 55) / (1 - exp(-(V + 55) / 10));
beta_n = 0.125 * exp(-(V + 65) / 80);
dndt = alpha_n * (1 - n) - beta_n * n;

% Return derivatives
dydt = [dVdt; dmdt; dhdt; dndt];
end
```

Lorenz Model Code

```
%% Main Script
% chaotic parameters
params=[10;28;8/3];

% initial condition
x0=[0; 1; 20];

% time discretization
tspan = [0 200];

% tolerance
tol = 1e-6;

% solve the Lorenz model
[t,x]=ODESolver(@(t,x)lorenz(t,x,params), tspan, x0, tol);

% visualize
figure
plot3(x(:,1),x(:,2),x(:,3),'w','LineWidth',1);
set(gca,'color','k','xcolor','w','ycolor','w','zcolor','w')
set(gcf,'color','k')
grid on
xlabel('$x$'),ylabel('$y$'),zlabel('$z$')
%% This is the function that builds the RHS for the ODE system
function dx=lorenz(t,x,p)

dx=[ p(1)*(x(2)-x(1));
     x(1)*(p(2)-x(3))-x(2);
     x(1)*x(2)-p(3)*x(3);
     ];
end
```