Problem Set 2
EPS 528, Science of Complex Systems

Jonas Katona

November 16, 2022

Problem 1.

(a) *Solution.* Before moving on, we can solve the ODE $du/dt = au$ exactly by guessing a solution of the form $u(t) = Ce^{\lambda t}$

$$\Rightarrow \frac{d}{dt}\left[Ce^{\lambda t}\right] = aCe^{\lambda t} \Rightarrow \lambda Ce^{\lambda t} = aCe^{\lambda t} \xRightarrow{e^{\lambda t} \neq 0 \ \forall t \in \mathbb{R}} \lambda = a.$$

Hence, $u(t) = Ce^{at}$ is the general solution to $du/dt = au$, and matching with the initial condition $u(0) = 10$, we find that $C = 10 \Rightarrow u(t) = 10e^{at}$. Furthermore, if we wanted to solve the IVP $du/dt = au$, $u(0) = 10$ numerically via the forward Euler method, our formulation would look like

$$u_{n+1} = u_n + au_n\Delta t \Rightarrow u_{n+1} = (1 + a\Delta t)\, u_n \tag{1}$$

with initial condition $u_0 = 10$. The actual solution $u(t)$, as we derived, is stable, because $a < 0 \Rightarrow 10e^{at} \xRightarrow{t\to\infty} 0$. For (1) to also yield a solution which is also stable — and in particular, A-stable — we require $|u_{n+1} - u_n| \to 0$, which necessitates that

$$|1 + a\Delta t| \leq 1 \Rightarrow -1 \leq 1 + a\Delta t \leq 1 \Rightarrow -2 \leq a\Delta t \leq 0. \tag{2}$$

$a < 0 \Rightarrow a\Delta t > 0$ since we want to integrate our system forward in time. Hence, (2) implies that the only necessary stability requirement is

$$a\Delta t \geq -2 \Rightarrow \Delta t \leq -\frac{2}{a}, \tag{3}$$

whence we should let $\Delta t_m = -2/a$.

Finally, in our case, since $a = -5$, (1)-(3) (and before) imply that our exact solution is $\boxed{u(t) = 10e^{-5t}}$ with numerical formulation $\boxed{u_{n+1} = (1 - 5\Delta t)\, u_n, u_0 = 10}$ and stability requirement $\Delta t \leq -2/a = 2/5 = 0.4$, such that $\boxed{\Delta t_m = 2/5 = 0.4}$. With these results, I wrote the following code in MATLAB that implements the numerical scheme and compares it to the exact solution:

```
u0 = 10;
a = -5;
exact = @(t) u0 * exp(a * t); % exact solution

% set initial conditions and step sizes
```

```
dtm = -2 / a;
dt = dtm / 2;
tf = 4 / dt;
tq = dt * (0 : tf);
uq = zeros(1, length(tq));
uq(1) = u0;

% run forward Euler time stepping
factor = 1 + dt * a;
for i = 2 : length(tq)
    uq(i) = factor * uq(i - 1);
end

% plot exact vs. numerical results
figure(1)
plot(tq, exact(tq), 'o-', tq, uq, 'o-')
title('Problem 1(a) solutions', 'fontsize', 16)
legend('Exact solution', ['Numerical solution ',...
    '($\Delta t=\Delta t_{m} / 2)$'], 'interpreter', 'latex')
xlabel('t', 'fontsize', 16)
ylabel('u(t)', 'fontsize', 16)

% plot errors (both in regular and semilog plots)
figure(2)
tiledlayout(2, 1)
nexttile
plot(tq, abs(uq - exact(tq)))
title('Problem 1(a) errors', 'fontsize', 16)
legend('Absolute error', 'interpreter', 'latex')
xlabel('t', 'interpreter', 'latex', 'fontsize', 16)
ylabel('$\left|u(t)-u_{h}(t)\right|$', 'interpreter',...
    'latex', 'fontsize', 16)

nexttile
semilogy(tq, abs(uq - exact(tq)), 'o-')
legend('Logarithm of the absolute error', 'interpreter', 'latex')
xlabel('t', 'interpreter', 'latex', 'fontsize', 16)
ylabel('$\log\left|u(t)-u_{h}(t)\right|$', 'interpreter',...
    'latex', 'fontsize', 16)
```

Even though my comments are probably enough suffice to understand the code, I will explain how the code works briefly. At first, we set our initial conditions `u0` and `a` at the top of the code, which allows us to produce a function handle `exact` for the exact solution to our IVP. Next, we initialize the numerical scheme by setting the timestep, `dt`; vector of equally-spaced sample times for our solution, `tq`, with spacing `dt`=$t_m/2$; and the vector of values for our numerical solution, `uq`. Finally, we step the numerical solution forward in time to generate all the values in `uq` and use this to produce three plots: (1) a plot comparing the exact solution with the numerical solution at $\Delta t = \Delta t_m/2$, (2) a plot of the absolute error $|u(t) - u_h(t)|$ vs. the times in `tq`; and finally, (3) a semilog version of (2), where the y-axis is logarithmic. These

plots are attached in Figure 1.

Between the exact and numerical results, we see that, despite the fact that the numerical solution was stable as desired, unfortunately, it decreased way too quickly and failed to capture the "smoother" curve produced as the exact solution, $u(t) = 10e^{-5t}$, decreased to zero. Hence, we can clearly see that, an A-stable need not be at all accurate, and in the case of nonlinear ODEs where the linearization of the numerical scheme does not capture the nonlinear aspects of stability, we also know that an accurate scheme need not be A-stable either. A balance between these two features, stability and accuracy, is necessary to yield a reasonable and useful numerical solution to any ODE, and in this case, the accuracy in our numerical solution been compromised.

Furthermore, note that the semilog plot yields a curiously linear plot. In fact, the slope of this line, as we can check via linear regression, is almost exactly -5, such that $\log|u(t) - u_h(t)| \approx -5t + a \Rightarrow |u(t) - u_h(t)| \approx Ce^{-5t}$ for some constant C, and since $u_h(t) \leq u(t)$ for all $t \in [0, 4]$, this really means that $u(t) \approx u_h(t) + Ce^{-5t}$. But why should we expect this to be the case? Well, intuitively, $u_h(t)$ decreases so quickly to zero that $u(t)$ dominates the error, and since $u(t) \sim e^{-5t}$, we have the result that $|u(t) - u_h(t)| \approx Ce^{-5t}$. □

(b) *Solution.* The code I used for this problem is attached below:

```
u0 = 10;
a = -5;
exact = @(t) u0 * exp(a * t); % exact solution

% set initial conditions and step sizes
dtm = -2 / a;
% dtq = dtm .* (2 .^ -(2 : 4));
dtq = dtm .* (2 .^ -(1 : 20));
rmserror = zeros(1, length(dtq));

% run through forward each step size in dtq
for k = 1 : length(dtq)
    % initialize numerical scheme for each case
    dt = dtq(k);
    tf = 4 / dt;
    tq = dt * (0 : tf);
    u = u0;
    num = 0; % numerator of RMSE
    den = u0 ^ 2; % denominator of RMSE

    % run forward Euler time stepping
    factor = 1 + dt * a;
    for i = 2 : length(tq)
        u = factor * u;
        ut = exact(tq(i));
        % update RMSE
        num = num + (u - ut) ^ 2;
        den = den + ut ^ 2;
    end
    rmserror(k) = sqrt(num / den);
```

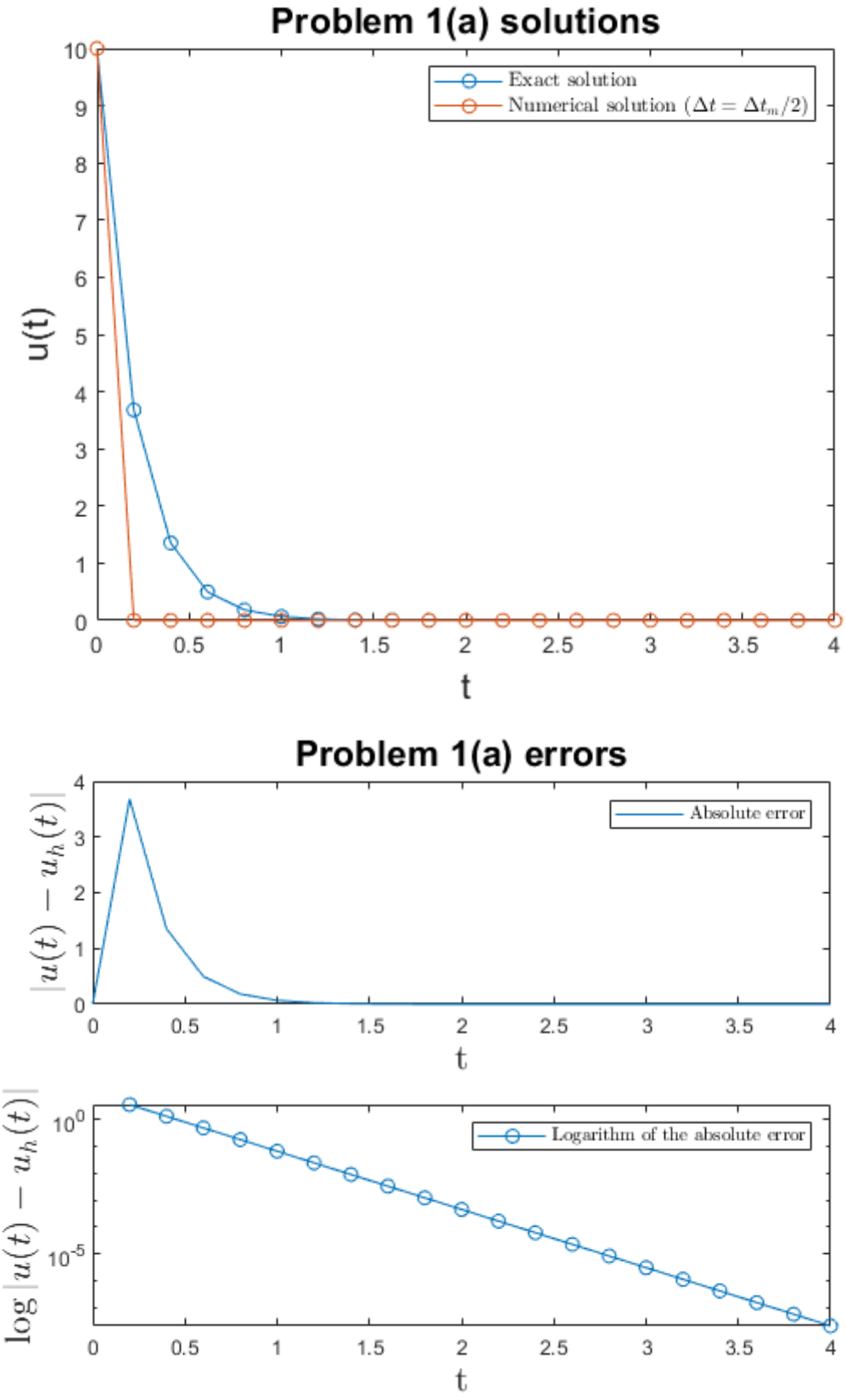


Figure 1: The three plots for Problem 1(a).

```
end

% slope of loglog plot (error convergence rate)
m = sum((log(rmserror) - mean(log(rmserror))) .* (log(dtq) -...
    mean(log(dtq)))) / sum((log(dtq) - mean(log(dtq))) .^ 2);
disp(['Convergence rate for error: ', num2str(m), '.'])

% plot results
figure(1)
loglog(dtq, rmserror, 'o-')
% title('Problem 1(b) results (as given)', 'fontsize', 16)
title('Problem 1(b) results (with more step sizes)',...
    'fontsize', 16)
legend(['Slope: ', num2str(m)])
xlabel('$\log \Delta t$', 'interpreter', 'latex',...
    'fontsize', 16)
ylabel(['$\log\left(\frac{\sum_{i}{\left(u_{i}-u\left(t_{i}',...
    '\right)\right)^{2}}}{\sum_{i}{u\left(t_{i}\right)^2}}',...
    '\right)^{1/2}$'], 'interpreter', 'latex', 'fontsize', 16)
```

The code above works in a very similar way as the one used in Problem 1(a). However, in this case, we make a vector of query step sizes, `dtq`, and run the forward Euler scheme from $t = 0$ to $t = 4$ at each of the desired step sizes. While doing this, to save memory and computational time, we save neither the exact nor numerical solutions in each case. Instead, we update the numerical solution, exact solution, and the sums at the top and bottom of the expression for the RMSE without saving anything at a given timestep, and then only save the final RMSE value after the conclusion of timestepping for each case. This is only possible because the RMSE error terms on the top and bottom are each calculated via a sum involving terms for the exact and numerical solutions at each timestep, and hence, we can just update the sums after each timestep.

Furthermore, in our code, we calculate and return the slope of the loglog plot of the error, because I claim that this should equal the convergence rate of the error between the numerical and exact solutions. Why? A numerical scheme with order p has a leading truncation error which should be of the form $C\left(\Delta t\right)^p$ for some constant C, i.e., $|u(t) - u_h(t)| \approx C\left(\Delta t\right)^p$, where u_h is our numerical solution evaluated at some time $t \in \Delta t\mathbb{Z}$.[1] Then, taking the logarithm of both sides, we find that $\log|u(t) - u_h(t)| \approx \log\left[C\left(\Delta t\right)^p\right] = \log C + p\log\left(\Delta t\right)$. Hence, the relationship between $\log|u(t) - u_h(t)|$ and $\log\left(\Delta t\right)$ is approximately linear with slope p. Furthermore, the loglog plot between $|u(t) - u_h(t)|$ and Δt will show a line, and if we take some kind of universal error such as the RMS relative error as defined in the problem, the error at each timestep will obey this scaling relationship, such that

$$|u\left(t_i\right) - u_i| \approx C_i\left(\Delta t\right)^p \Rightarrow \sum_i \left(u\left(t_i\right) - u_i\right)^2 \approx \sum_i \left(C_i\left(\Delta t\right)^p\right)^2 = \left(\Delta t\right)^{2p} \sum_i C_i^2$$

$$\Rightarrow \mathrm{E} = \left(\frac{\sum_i \left(u\left(t_i\right) - u_i\right)^2}{\sum_i u\left(t_i\right)^2}\right)^{1/2} \approx \left(\frac{\left(\Delta t\right)^{2p} \sum_i C_i^2}{\sum_i u\left(t_i\right)^2}\right)^{1/2} = \left(\Delta t\right)^p \left(\frac{\sum_i C_i^2}{\sum_i u\left(t_i\right)^2}\right)^{1/2}. \quad (4)$$

[1] All higher-order terms will be small when $\Delta t \ll 1$, which is usually be the case if we want a reasonably

Thus, (4) demonstrates that the RMS relative error should also be proportional to $(\Delta t)^p$, and hence, by the same reasoning in the paragraph above, the loglog plot of E vs. Δt should also yield a line with slope of p. To calculate this slope, one can use least-squares linear regression across the errors sampled at each timestep used, i.e., if we sample the error at k different timesteps, then

$$p \approx \frac{\sum_k \left(\log E_k - \overline{\log E}\right)\left(\log \Delta t_k - \overline{\log \Delta t}\right)}{\sum_k \left(\log \Delta t_k - \overline{\log \Delta t}\right)^2},$$

where $\overline{x}$ is understood to mean the average over $x_1, \ldots, x_k$.

In summary, by all of our work above, we produce the plots found in Figure 2. Regardless of how many different timesteps we sample, we find a best-fit slope (in the least-squares sense) very close to $p = 1$ ($p \approx 0.98066$ and $p \approx 0.9987$, respectively), but not exactly. Note that we should not expect p to be precisely one, because the order of a numerical scheme is determined by the leading truncation error, and higher-order terms, while small, could still affect the solution in rather complicated ways, considering that the magnitude of these terms depends not only off of Δt, but also the higher-order derivatives of the function used to define our original ODE.

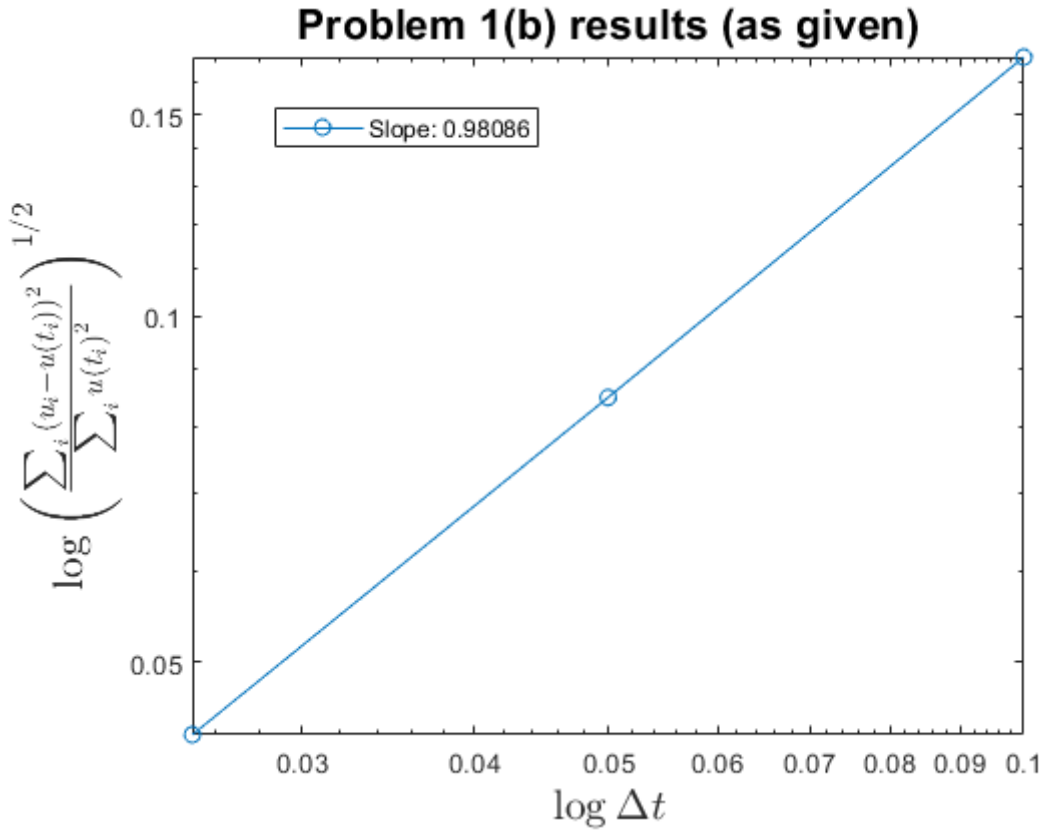
□

(c) *Solution.* We can solve the IVP $du/dt = au$, $u(0) = 10$, $a = -5$ with RK4, i.e., "the Runge-Kutta method," as follows:

$$k_{1,n} = au_n, \quad k_{2,n} = a\left(u_n + \Delta t \frac{k_{1,n}}{2}\right), \quad k_{3,n} = a\left(u_n + \Delta t \frac{k_{2,n}}{2}\right),$$
$$k_{4,n} = a\left(u_n + k_{3,n}\Delta t\right), \quad u_{n+1} = u_n + \frac{\Delta t}{6}\left(k_{1,n} + 2k_{2,n} + 2k_{3,n} + k_{4,n}\right). \tag{5}$$

To code up the numerical scheme described in (5), we can use the same code as we used in part (b), except with the forward Euler part of the scheme replaced with the RK4 scheme described in (5). This is presented below:

```
u0 = 10;
a = -5;
exact = @(t) u0 * exp(a * t); % exact solution

% set initial conditions and step sizes
dtm = -2 / a;
% dtq = dtm .* (2 .^ -(2 : 4));
dtq = dtm .* (2 .^ -(0 : 0.5 : 10));
rmserror = zeros(1, length(dtq));

% run through forward each step size in dtq
for k = 1 : length(dtq)
    % initialize numerical scheme for each case
    dt = dtq(k);
    tf = 4 / dt;
    tq = dt * (0 : tf);
    u = u0;
```



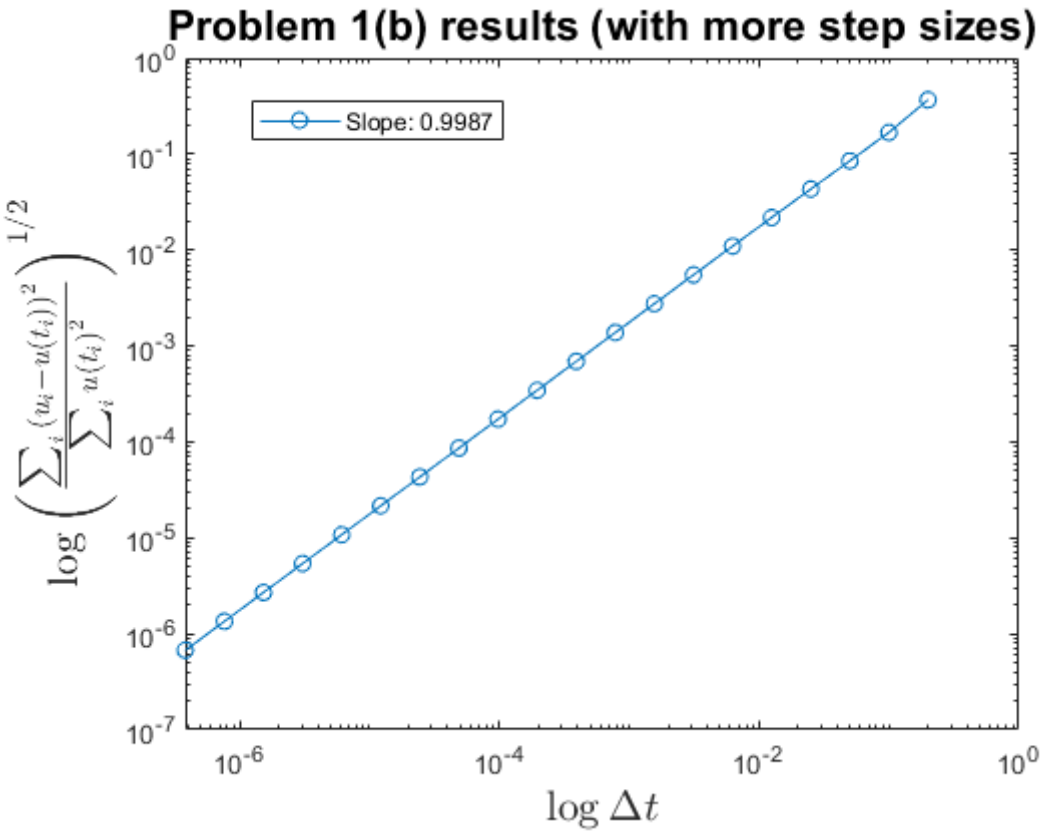


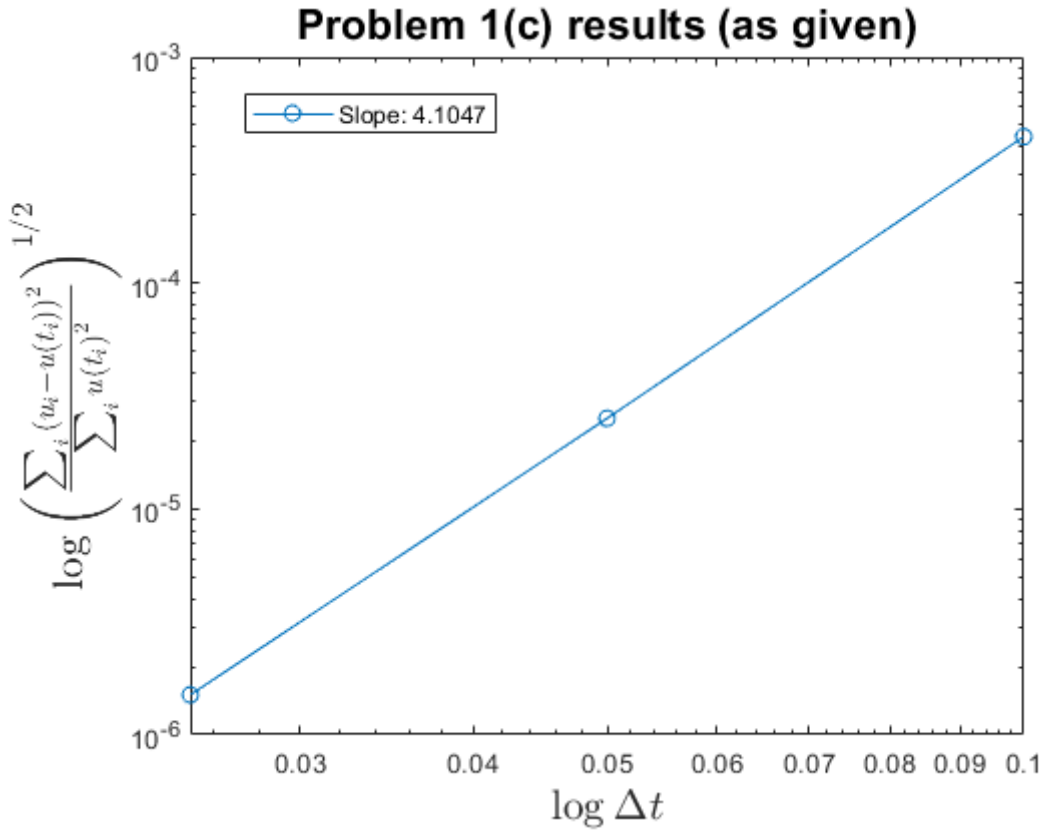
Figure 2: Two plots for Problem 1(b). In the original problem, we were asked to generate our error convergence plot using only three step sizes: $\Delta t_m/4$, $\Delta t_m/8$, and $\Delta t_m/16$. To find a more accurate estimate for p, I also produced a second convergence plot which samples $\Delta t = \Delta t_m/2, \Delta t_m/2^2, \ldots, \Delta t_m/2^{20}$.

```
        num = 0; % numerator of RMSE
        den = u0 ^ 2; % denominator of RMSE

        % run RK4
        for i = 2 : length(tq)
            k1 = a * u;
            k2 = a * (u + 0.5 * dt * k1);
            k3 = a * (u + 0.5 * dt * k2);
            k4 = a * (u + dt * k3);
            u = u + (dt / 6) * (k1 + 2 * k2 + 2 * k3 + k4);
            ut = exact(tq(i));
            % update RMSE
            num = num + (u - ut) ^ 2;
            den = den + ut ^ 2;
        end
        rmserror(k) = sqrt(num / den);
    end

    % slope of loglog plot (error convergence rate)
    m = sum((log(rmserror) - mean(log(rmserror))) .* (log(dtq) -...
        mean(log(dtq)))) / sum((log(dtq) - mean(log(dtq))) .^ 2);
    disp(['Convergence rate for error: ', num2str(m), '.'])

    % plot results
    figure(1)
    loglog(dtq, rmserror, 'o-')
    % title('Problem 1(c) results (as given)', 'fontsize', 16)
    title('Problem 1(c) results (with more step sizes)',...
        'fontsize', 16)
    legend(['Slope: ', num2str(m)])
    xlabel('$\log \Delta t$', 'interpreter', 'latex',...
        'fontsize', 16)
    ylabel(['$\log\left(\frac{\sum_{i}{\left(u_{i}-u\left(t_{i}',...
        '\right)\right)^{2}}}{\sum_{i}{u\left(t_{i}\right)^2}}',...
        '\right)^{1/2}$'], 'interpreter', 'latex', 'fontsize', 16)
```

The results of the code immediately above is found in Figures 3 and 4.

As in part (b), by considering the slope of the loglog plot of E vs. Δt, we can compute the order of convergence, which in two of the cases we tried was very close to 4 ($p \approx 4.1047$ and $p \approx 4.0789$, respectively). In the last case, the only reason why we did not get an order of convergence of 4 was because our solution converged so quickly that the error fell below or near machine tolerance ($\epsilon = 2.2204 \times 10^{-16}$), and in that case, the computation of E — in particular, the difference between the exact and numerical solutions at each timestep — would be ridden with floating point and precision errors. Certainly, this explains very well the trend in the loglog plot: The solution looks linear at first for larger values of Δt, but as Δt becomes small enough, this linear trend is broken once E nears a small enough magnitude. □

convergent solution.



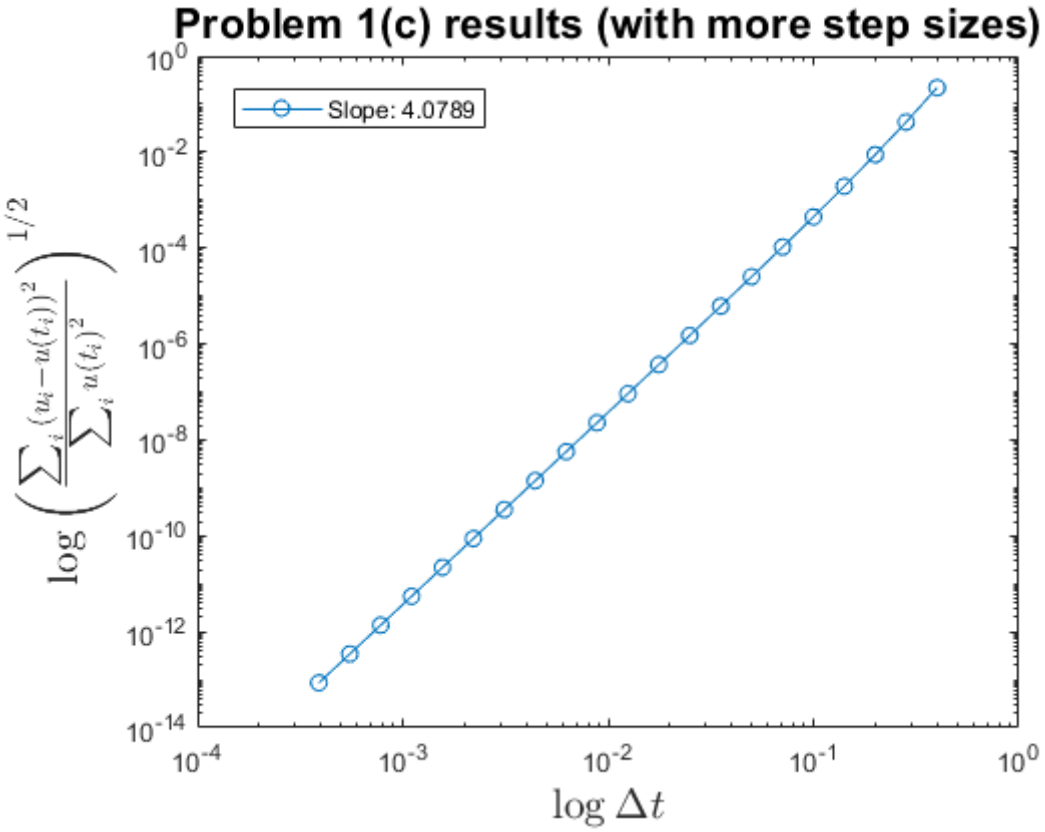


Figure 3: The first two plots for Problem 1(c). In the original problem, we were asked to generate our error convergence plot using only three step sizes: $\Delta t_m/4$, $\Delta t_m/8$, and $\Delta t_m/16$. To find a more accurate estimate for p, I produced two more convergence plots, the first of which samples $\Delta t = \Delta t_m/2^0, \Delta t_m/2^{0.5}, \ldots, \Delta t_m/2^{10}$. The third one is found in Figure 4.

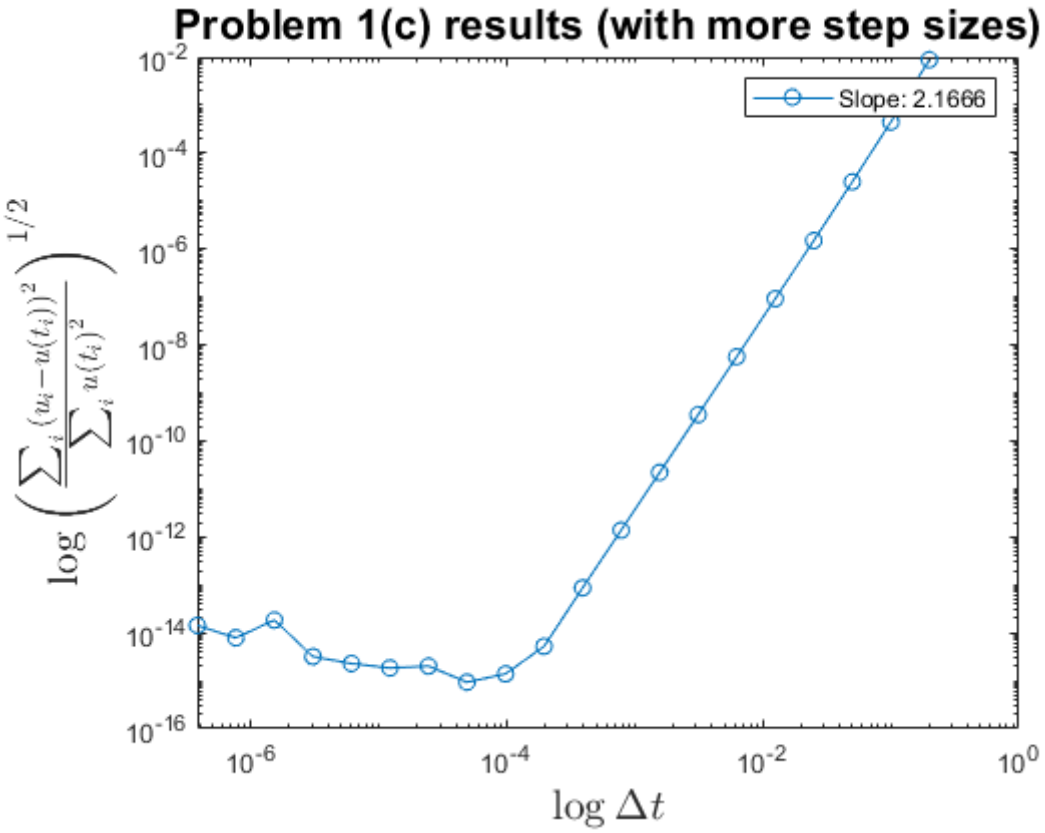


Figure 4: The third plot for Problem 1(c) which uses $\Delta t = \Delta t_m/2^0, \Delta t_m/2^{0.5}, \ldots, \Delta t_m/2^{20}$. We provide a very likely explanation for the deviation of the linear trend in the third plot in the main text.

Problem 2.

Solution. If Method A is 1st order, then we would expect the relative error in Method A, E_A, to scale with Δt, i.e., $E_A \sim \Delta t \Rightarrow E_A \approx A\Delta t$, for constant A and some Δt small enough such that all higher-order terms in the leading truncation error for the scheme are negligible. Similarly, $E_B \sim (\Delta t)^4 \Rightarrow E_B \approx B\,(\Delta t)^4$ for some constant B.

We also want to consider the computational time that each timestep takes,[2] assuming that the change in computational cost for other things that might vary with Δt, e.g., MATLAB's internal corrections for floating point error, matrix solvers, etc., are negligible. Then, the number of timesteps is directly proportional to the number of function evaluations for either scheme, since for most standard numerical schemes, the number of function evaluations at each timestep does not change with the timestep, or if they do, these changes do not depend directly on the number of timesteps. Hence, we can assume that each method has linear time complexity $\mathcal{O}(n)$ with respect to the timestep, such that the times for each scheme to complete n timesteps are $T_A \approx A'n = A'\,(t/\Delta t)$ and $T_B \approx B'n = B'\,(t/\Delta t)$, respectively, for constants A' and B' to be determined.

From here, we need to solve for constants A, B, A', and B'. We already know that $\Delta t = 0.1$, $t = 100$, $E_A = 0.02$, $E_B = 0.005$, $T_A = 5$ s, and $T_B = 2\,(5)$ s $= 10$ s. Hence,

$$E_A \approx A\Delta t \Rightarrow 0.02 \approx A(0.1) \Rightarrow A \approx \frac{0.02}{0.1} = 0.2, \tag{6}$$

$$E_B \approx B\,(\Delta t)^4 \Rightarrow 0.005 \approx B\,(0.1)^4 \Rightarrow B \approx \frac{0.005}{0.0001} = 50, \tag{7}$$

$$T_A \approx A'\left(\frac{t}{\Delta t}\right) \Rightarrow 5 \text{ s} \approx A'\left(\frac{100}{0.1}\right) \Rightarrow A' \approx 5\frac{0.1}{100} \text{ s} = 0.005 \text{ s}, \tag{8}$$

$$T_B \approx B'\left(\frac{t}{\Delta t}\right) \Rightarrow 10 \text{ s} \approx B'\left(\frac{100}{0.1}\right) \Rightarrow B' \approx 10\frac{0.1}{100} \text{ s} = 0.01 \text{ s}. \tag{9}$$

Using (6)-(9), we can now calculate T_A and T_B, noting that, for the desired case in the problem, $E_A = E_B = 0.0001$. However, since we are now setting the accuracies equal for each scheme, we should also expect different stepsizes for each scheme, such that Method A has stepsize Δt_A and Method B Δt_B.

$$E_A = E_B = 0.0001 \Rightarrow A\Delta t_A \approx B\,(\Delta t_B)^4 \approx 0.0001 \Rightarrow 0.2\Delta t_A \approx 50\,(\Delta t_B)^4 \approx 0.0001 \Rightarrow$$
$$\Delta t_A \approx \frac{0.0001}{0.2} = 0.0005, \quad (\Delta t_B)^4 \approx \frac{0.0001}{50} = 2 \times 10^{-6} \Rightarrow \Delta t_B \approx 0.037606. \tag{10}$$

Then, using (10), we can calculate T_A and T_B as such:

$$T_A \approx A'\left(\frac{t}{\Delta t_A}\right) = 0.005\left(\frac{10^5}{0.0005}\right) \text{ s} = 10^6 \text{ s} \approx 11.574 \text{ days}, \tag{11}$$

$$T_B \approx B'\left(\frac{t}{\Delta t_B}\right) \approx 0.01\left(\frac{10^5}{0.037606}\right) \text{ s} = 26591.5 \text{ s} \approx 7.3865 \text{ hours}. \tag{12}$$

Therefore, by (11) and (12), Method A should take $\boxed{T_A \approx 11.574 \text{ days}}$ to integrate the system and solve the IVP, while Method B should take $\boxed{T_B \approx 7.3865 \text{ hours}}$ for the same accuracy

[2]The differences in computational cost between each scheme could easily be nontrivial. For instance, it is generally true that a higher-order scheme will require more function evaluations at each timestep than a lower-order one, and these are bound to increase the computational complexity in some way which could be

of 0.01%, which is a drastic difference in computation time and highlights the advantages of using higher-order schemes whenever each function evaluation is not overly expensive. In other words, if you needed to integrate this ODE for your next weekly problem set, Method B can do it in a little less than a third of a day, which gives you ample time to analyze your results, while for Method A, the integration will not have finished in time for the due date, unfortunately. □

Problem 3.

(a) *Solution.* As is usual for higher-dimensional systems, we start by transforming the system to a second-order system of first-order ODEs via the variables $x_1 = x \Rightarrow \dot{x}_1 = \dot{x}$, $x_2 = \dot{x}_1 = \dot{x} \Rightarrow \dot{x}_2 = \ddot{x} = -dV(x)/dx = x - x^3$, which gives us that

$$\begin{pmatrix} \dot{x}_1 \\ \dot{x}_2 \end{pmatrix} = \begin{pmatrix} x_2 \\ -\frac{dV}{dx}\big|_{x=x_1} \end{pmatrix} = \begin{pmatrix} x_2 \\ x_1 - x_1^3 \end{pmatrix}. \tag{13}$$

To perform linear stability analysis on (13), we need to 1) find the fixed points for (13) and 2) calculate the eigenvalues and eigenvectors of the Jacobian for (13) evaluated at these fixed points. The fixed points occur when *both* the "position" x_1 and "velocity" x_2 are simultaneously zero, and this occurs when $x_2^* = 0$ and $x_1^* - (x_1^*)^3 = 0 \Rightarrow x_1^* = 0, \pm 1$, giving us three fixed points. Furthermore, the Jacobian for (13) is

$$J(x_1, x_2) = \begin{pmatrix} \frac{\partial \dot{x}_1}{\partial x_1} & \frac{\partial \dot{x}_1}{\partial x_2} \\ \frac{\partial \dot{x}_2}{\partial x_1} & \frac{\partial \dot{x}_2}{\partial x_2} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 - 3x_1^2 & 0 \end{pmatrix},$$

which allows us to compute the eigenvalues and eigenvectors at each (x_1^*, x_2^*). We do this below, noting that an eigenvalue λ and its associated eigenvector $\mathbf{v}$ should satisfy the equation $J(x_1^*, x_2^*)\,\mathbf{v} = \lambda \mathbf{v} \Rightarrow (J(x_1^*, x_2^*) - \lambda I)\,\mathbf{v} = 0 \Rightarrow \det(J(x_1^*, x_2^*) - \lambda I) = 0$ in this case:

- *Fixed point 1:* $(x_1^*, x_2^*) = (-1, 0)$. In this case, we see that

$$\begin{aligned} J(x_1^*, x_2^*) = J(-1, 0) &= \begin{pmatrix} 0 & 1 \\ 1 - 3(-1)^2 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ -2 & 0 \end{pmatrix} \\ \Rightarrow J(x_1^*, x_2^*) - \lambda I &= \begin{pmatrix} -\lambda & 1 \\ -2 & -\lambda \end{pmatrix} \Rightarrow \det(J(x_1^*, x_2^*) - \lambda I) = \lambda^2 + 2 = 0 \\ &\Rightarrow \boxed{\lambda = \pm i\sqrt{2}.} \end{aligned} \tag{14}$$

  Hence, by (14), since $J(-1, 0)$ has a complex conjugate pair of imaginary eigenvalues, the fixed point at $(-1, 0)$ is a center, and for trajectories close enough to $(-1, 0)$, the system will exhibit oscillations with frequency $\sqrt{2}$. Furthermore, both eigenvectors will be complex since the eigenvalues are complex.

- *Fixed point 2:* $(x_1^*, x_2^*) = (0, 0)$. When evaluated at the origin, the Jacobian is

$$\begin{aligned} J(x_1^*, x_2^*) = J(0, 0) &= \begin{pmatrix} 0 & 1 \\ 1 - 3(0)^2 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \\ (x_1^*, x_2^*) - \lambda I &= \begin{pmatrix} -\lambda & 1 \\ 1 & -\lambda \end{pmatrix} \Rightarrow \det(J(x_1^*, x_2^*) - \lambda I) = \lambda^2 - 1 = 0 \\ &\Rightarrow \boxed{\lambda = \pm 1.} \end{aligned} \tag{15}$$

Hence, by (15), since $J(0,0)$ has two real eigenvalues with opposite signs, we conclude that $(0,0)$ is a saddle. Furthermore,

$$((x_1^*, x_2^*) - \pm I)\,\mathbf{v} = \begin{pmatrix} \mp 1 & 1 \\ 1 & \mp 1 \end{pmatrix} \mathbf{v} = 0 \Rightarrow \mathbf{v} = \begin{pmatrix} \pm 1 \\ 1 \end{pmatrix}. \tag{16}$$

By (16), we observe that $\mathbf{v} =< -1, 1 >$ corresponds to $\lambda = -1$ and $\mathbf{v} =< 1, 1 >$ to $\lambda = 1$. Thus, $< -1, 1 >$ is the stable eigendirection while $< 1, 1 >$ is the unstable eigendirection.

- *Fixed point 3:* $(x_1^*, x_2^*) = (1, 0)$. Since $J(-1, 0) = J(1, 0)$, the fixed point at $(1, 0)$ must exhibit the same stability as the one at $(-1, 0)$, as the Jacobians are the same at both fixed points, such that they have the same eigenvalues and eigenvectors. Hence, $(1, 0)$ is also a center with eigenvalues $\pm i\sqrt{2}$.

□

(b) *Solution.* I claim that, yes, the total energy $H = \frac{1}{2}(\dot{x})^2 + V(x) = \frac{1}{2}x_2^2 + V(x_1)$ is conserved for this system. In fact, (1) if we plotted H at fixed energy levels (as constrained by initial conditions), we could reproduce *any* phase space trajectory and (2) our system is actually Hamiltonian if we define H as our Hamiltonian. To show this, we take the total time derivative of H via chain rule:

$$\frac{d}{dt}H(x_1, x_2) = \dot{x}_1 \frac{\partial H}{\partial x_1} + \dot{x}_2 \frac{\partial H}{\partial x_2} = \dot{x}_1 \frac{\partial}{\partial x_1}\left[\frac{1}{2}x_2^2 + V(x_1)\right] + \dot{x}_2 \frac{\partial}{\partial x_2}\left[\frac{1}{2}x_2^2 + V(x_1)\right] =$$
$$\dot{x}_1 \frac{\partial V(x_1)}{\partial x_1} + \dot{x}_2 \frac{\partial}{\partial x_2}\left(\frac{1}{2}x_2^2\right) = x_2 \frac{\partial V(x_1)}{\partial x_1} + \left(-\frac{dV(x_1)}{dx_1}\right) x_2 = 0. \tag{17}$$

Thus, by (17), we conclude that the total energy H is conserved along trajectories, i.e., $\boxed{dH/dt = 0}$. □

(c) *Solution.* To plot the phase portrait of the system, we use the function

```
function [ sol ] = problem3c1(x10, x20, t0, tf)
tspan = [t0 tf]; % set initial and end times
u0 = [x10 x20]; % set initial values for x_{1} and x_{2}
opts = odeset('RelTol', 1e-12); % set error tolerance
func = @(t, x) [x(2) ; x(1) * (1 - x(1) ^ 2)]; % function handle
sol = ode45(func, tspan, u0, opts); % use ode45 to solve ODE
```

in the following script:

```
fps = [-1 0 1; 0 0 0]; % fixed points
spacing = 0.2; % spacing for vector plot
[X1, X2] = meshgrid(-2: spacing: 2, -2: spacing: 2);
% vector positions
U = X2; % \dot{x_{1}}
V = X1 .* (1 - X1 .^ 2); % \dot{x_{2}}

x10q = [-0.8 : 0.2 : 0.8 1.5 : 0.1 : 2]; % sample trajectories
% we take x_{2}(0)=0 for all x_{2} initial conditions.
```

poorly understood a priori. Also, schemes such as the implicit Runge-Kutta schemes often involve nonlinear

```
t0 = 0; % initial time
tf = 200; % final time

figure(1)
quiver(X1, X2, U, V, 'b') % plot flow field
title('Phase portrait for the Duffing oscillator',...
    'fontsize', 16)
xlabel('$x_{1}=x$', 'interpreter', 'latex',...
    'fontsize', 16)
ylabel('$x_{2}=\dot{x}$', 'interpreter', 'latex',...
    'fontsize', 16)
xlim([-2.1 2.1])
ylim([-2.2 2.2])
hold on
% plot trajectories
for i = 1 : length(x10q)
    sol = problem3c1(x10q(i), 0, t0, tf);
    u = sol.y;
    plot(u(1, :), u(2, :), 'k-')
end
% plot eigenvectors
annotation('line', [0.46 0.5] + 0.02,...
    [0.54 0.5] + 0.02, 'color', 'r')
annotation('line', [0.54 0.5] + 0.02,...
    [0.46 0.5] + 0.02, 'color', 'r')
annotation('arrow', [0.47 0.471] + 0.02,...
    [0.53 0.529] + 0.02, 'color', 'r')
annotation('arrow', [0.53 0.529] + 0.02,...
    [0.47 0.471] + 0.02, 'color', 'r')
annotation('arrow', [0.5 0.55] + 0.02,...
    [0.5 0.55] + 0.02, 'color', 'r')
annotation('arrow', [0.5 0.45] + 0.02,...
    [0.5 0.45] + 0.02, 'color', 'r')
% plot fixed points
scatter(fps(1, :), fps(2, :), 'filled', 'g')
hold off
```

`problem3c1` takes inputs `x10` and `x20` for $x_1(t_0)$ and $x_2(t_f)$, respectively, and integrates the ODEs described in (13) using `ode45` with a relative error tolerance at each timestep of 10^{-12}.[3] Then, using `problem3c1`, we can construct the second script which is pretty self-explanatory. In particular, all of our initial conditions are of the form $(x, 0)$ for some $x \in [-0.8, 2]$; this suffices because our trajectories are closed and almost always cross the x_1-axis. We did not plot the edge cases which lie on the separatrix, for which case we have two homoclinic orbits connecting back to the fixed point at the origin;

solvers which could increase the computational time in a very complicated and nontrivial fashion.

[3] As an adaptive ODE solver, `ode45` is a predictor-corrector method based off of a 4th-order explicit Runge-Kutta method and an embedded (i.e., using the previous function evaluations for the 4th-order scheme) 5th-order explicit Runge-Kutta method to approximate the local truncation error. Using this approximation, `ode45` chooses an optimal timestep which stays below the prescribed error tolerance, such that the timesteps are not so small that we lose computational efficiency but small enough that this error tolerance is met.

these correspond to when $H(x_1(t), x_2(t)) = 0 \Rightarrow x_2 = \pm |x_1| \sqrt{2 - x_1^2}/\sqrt{2}$. The results for both of these codes can be found in Figure 5.

Since H as defined in part (b) is conserved along trajectories of the system, we should expect any numerical trajectory to lie arbitrarily close (as $\Delta t \to 0$) to some level curve of H. In fact, we can determine these trajectories explicitly since $H(x_1(t), x_2(t)) = H(x_1(0), x_2(0))$ for all $t \in \mathbb{R}$, and in fact, this relationship allows us to solve for x_2 as a function of x_1 and the initial conditions. Hence, also in Figure 5, we compare our numerical results with the exact trajectories as plotted in Desmos. □

(d) *Solution.* Looking at Figure 5, we see that our system consists of two "bounded states" at $x = \pm 1$. In other words, even though all trajectories are closed, for initial conditions (i.e., "energies") close enough to $(\pm 1, 0)$, all trajectories stay in the right or left half-plane, respectively. However, once we leave a certain region, i.e., if our system passes a certain energy, the system exhibits orbits that go between both states. This boundary region is called the separatrix for the system and consists of two homoclinic orbits that go back to the origin.

Physically, we see that V defines a prototypical double-well potential, and the ODE system we just analyzed is often called a Duffing oscillator, as I have aptly titled the plot in Figure 5. This system appears as a higher-order correction to the physical pendulum and other systems that exhibit nonlinear oscillations, as well as provides an exact model (usually after the scaling of certain terms) for physical scenarios in which there are two local energy minima. For instance, as suggested with the marble example, suppose that we have a marble free to roll between two valleys separated by a mound in the middle. If the marble does not have enough energy at the start, e.g., if it starts too low or close to the bottom of one valley with little velocity, the marble will only be able to move back and forth within the valley it started nearest to. However, if the marble starts off with a lot of energy, e.g., it starts very high or with a high velocity, the marble will be able to overcome the mound in the middle and roll freely between the two different valleys. □

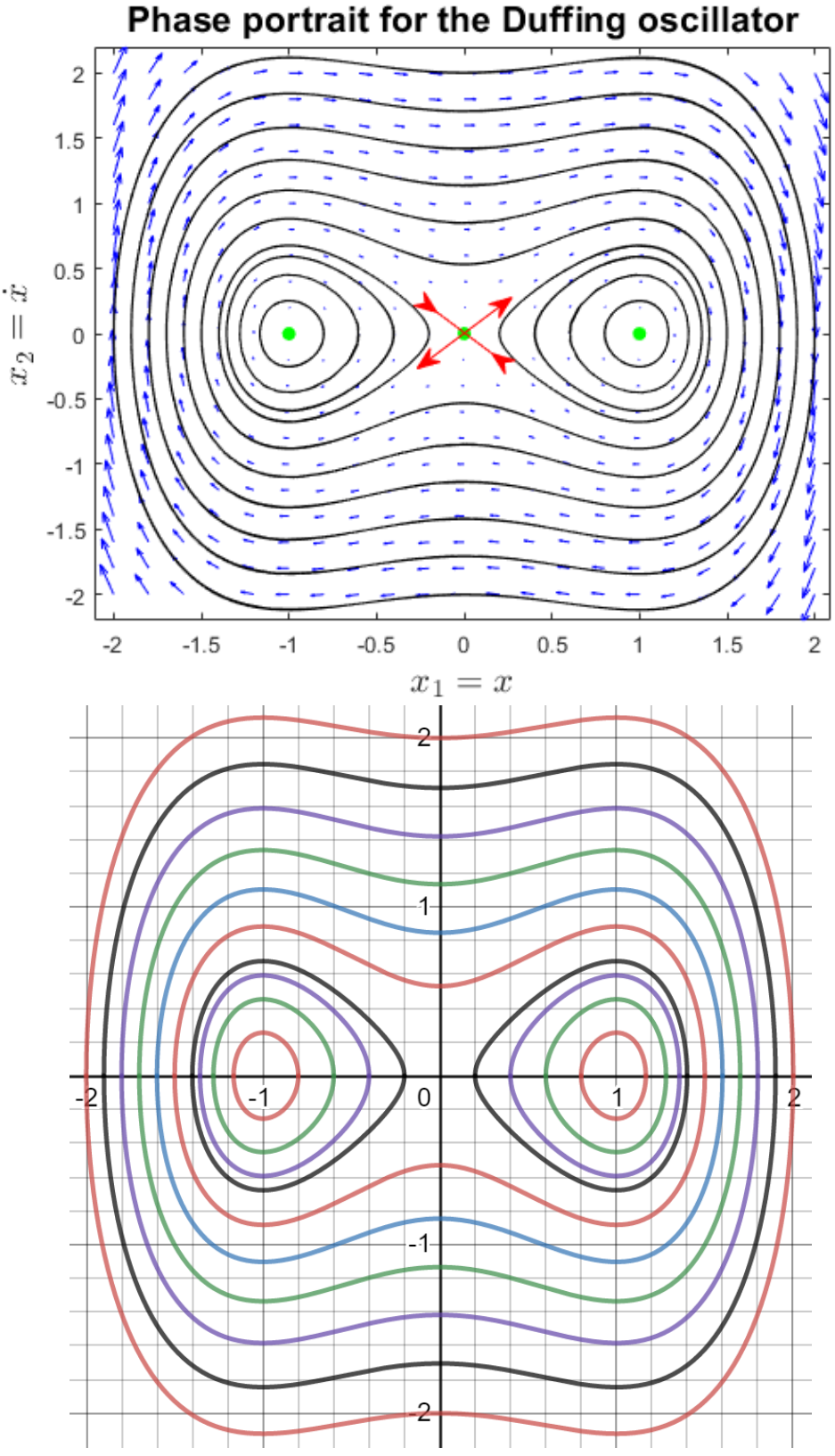


Figure 5: The top plot is my numerical phase portrait produced in MATLAB, and the bottom shows the exact phase trajectories plotted in Desmos. In the top, the green points correspond to fixed points, the black curves are representative trajectories, the blue arrows produce the flow field, and the red arrows are the eigendirections, for which their stabilities are shown via the arrow directions. For both plots, we see that the trajectories graphically appear to match exactly.