E7: Introduction to Computer Programming for Scientists and Engineers
University of California at Berkeley, Spring 2017
Instructor: Lucas A. J. Bastien

Lab Assignment 12: Ordinary Differential Equations

Version: release

Due date: Friday April 21th 2017 at 12 pm (noon).

General instructions, guidelines, and comments:

- For each question, you will have to write and submit one or more Matlab functions. We provide a number of test cases that you can use to test your function. The fact that your function works for all test cases provided does not guarantee that it will work for all possible test cases relevant to the question. It is your responsibility to test your function thoroughly, to ensure that it will also work in situations not covered by the test cases provided. During the grading process, your function will be evaluated on a number of test cases, some of which are provided here, some of which are not.
- Submit on bCourses one m-file for each function that you have to write. The name of each file must be the name of the corresponding function, with the suffix `.m` appended to it. For example, if the name of the function is `my_function`, the name of the file that you have to submit is `my_function.m`. **Carefully check the name of each file that you submit.** Do not submit any zip file. If you re-submit a file that you have already submitted, bCourses may rename the file by adding a number to the file's name (*e.g.,* rename `my_function.m` into `my_function-01.m`). This behavior is okay and should be handled seamlessly by our grading system. Do not rename the file yourself as a response to this behavior.
- A number of optional Matlab toolboxes can be installed alongside Matlab to give it more functionality. All the functions that you have to write to complete this assignment can, however, be implemented without the use of any optional Matlab toolboxes. We encourage you to not use optional toolboxes to complete this assignment. All functions of the Matlab base installation will be available to our grading system, but functions from optional toolboxes may not. If one of your function uses a function that is not available to our grading system, you will loose all points allocated to the corresponding part of this assignment. To guarantee that you are not using a Matlab function from an optional toolbox that is not available to our grading system, use one or both of the following methods:
  - ◇ Only use functions from the base installation of Matlab.
  - ◇ Make sure that your function works on the computers of the 1109 Etcheverry Hall computer lab. All the functions available on these computers will be available to our grading system.

- For this assignment, the required submissions are:
  - ◇ `my_explicit_euler.m`
  - ◇ `my_midpoint.m`
  - ◇ `my_rk4.m`
  - ◇ `my_ode.m`
  - ◇ `my_water_tank.m`

1. Generic solvers

Consider a real-valued function y, differentiable on $\mathbb{R}$, that is the solution of the following first-order ordinary differential equation:

$$y'(t) = F(t, y) \tag{1}$$

In this question, you will write three different numerical solvers of this first-order ordinary differential equation. These solvers will estimate the value of the function y at different times of the interval $[t_0, t_f]$. We will call t_0 the "initial time" and t_f the "final time".

1.1. Explicit Euler

The explicit Euler method uses the following approximation to estimate the solution function at time step t_{i+1} from the value of the function at time step t_i:

$$y(t_{i+1}) = y(t_i) + F(t_i, y(t_i))(t_{i+1} - t_i) \tag{2}$$

Write a function with the following header:

```
function [t, y] = my_explicit_euler(f, t0, tf, y0, n)
```

where

- `f` is a function handle that represents F in Equation 1. The corresponding function takes two input arguments (scalars of class `double`): the time t and the value of the function y at time t, respectively, and returns one output argument (a scalar of class `double`): the value of $F(t, s)$.
- `t0` and `tf` are scalars of class `double` that represent the initial time t_0 and final time t_f, respectively.
- `y0` is a scalar of class `double` that represents the value of the function y at time $t = t_0$.
- `n` is a scalar of class `double` that represents the number of equally-spaced time steps in the interval $[t_0, t_f]$ (t_0 and t_f included) to use for the explicit Euler method. You can assume that `n > 1`.

- `t` is an $1 \times (n+1)$ array of class `double` that represents, in increasing order, the equally-spaced points in the interval $[t_0, t_f]$ used for the explicit Euler method.
- `y` is $1 \times (n+1)$ array of class `double` that represents the values of the function y at the times `t` (in the same order) as approximated by the explicit Euler method.

Test cases:

```
>> % Test case 1
>> y0 = 10;
>> f = @(t, y) -0.5*y;
>> [t, y] = my_explicit_euler(f, 0, 5, y0, 10)
t =
  Columns 1 through 7
         0    0.5000    1.0000    1.5000    2.0000    2.5000    3.0000
  Columns 8 through 11
    3.5000    4.0000    4.5000    5.0000
y =
  Columns 1 through 7
   10.0000    7.5000    5.6250    4.2188    3.1641    2.3730    1.7798
  Columns 8 through 11
    1.3348    1.0011    0.7508    0.5631

>> % Compare our approximate solution to the analytical solution
>> y0 * exp(-0.5*t)
ans =
  Columns 1 through 7
   10.0000    7.7880    6.0653    4.7237    3.6788    2.8650    2.2313
  Columns 8 through 11
    1.7377    1.3534    1.0540    0.8208

>> % Test case 2
>> y0 = 10;
>> f = @(t, y) -0.05 .* t .* y;
>> [t, y] = my_explicit_euler(f, 1, 5, y0, 20)
t =
  Columns 1 through 7
    1.0000    1.2000    1.4000    1.6000    1.8000    2.0000    2.2000
  Columns 8 through 14
    2.4000    2.6000    2.8000    3.0000    3.2000    3.4000    3.6000
  Columns 15 through 21
    3.8000    4.0000    4.2000    4.4000    4.6000    4.8000    5.0000
y =
  Columns 1 through 7
   10.0000    9.9000    9.7812    9.6443    9.4900    9.3191    9.1328
  Columns 8 through 14
    8.9318    8.7175    8.4908    8.2531    8.0055    7.7493    7.4858
  Columns 15 through 21
    7.2163    6.9421    6.6644    6.3845    6.1036    5.8228    5.5433

>> % Test case 3
>> y0 = 5;
>> f = @(t, y) -0.1 * y.^2;
>> [t, y] = my_explicit_euler(f, 1, 10, y0, 24)
t =
  Columns 1 through 7
    1.0000    1.3750    1.7500    2.1250    2.5000    2.8750    3.2500
  Columns 8 through 14
    3.6250    4.0000    4.3750    4.7500    5.1250    5.5000    5.8750
  Columns 15 through 21
    6.2500    6.6250    7.0000    7.3750    7.7500    8.1250    8.5000
  Columns 22 through 25
    8.8750    9.2500    9.6250   10.0000
y =
  Columns 1 through 7
    5.0000    4.0625    3.4436    2.9989    2.6617    2.3960    2.1807
  Columns 8 through 14
    2.0024    1.8520    1.7234    1.6120    1.5146    1.4286    1.3520
  Columns 15 through 21
    1.2835    1.2217    1.1657    1.1148    1.0682    1.0254    0.9860
  Columns 22 through 25
    0.9495    0.9157    0.8842    0.8549
```

1.2. Midpoint method

The midpoint method uses the slope estimated at the middle of the time step:

$$y(t_{i+1}) = y(t_i) + F(t_i + \Delta t_i/2, y^\star(t_{i+1/2}))\Delta t_i \tag{3}$$

where $\Delta t_i = t_{i+1} - t_i$ and $y^\star(t_{i+1/2})$ is evaluated by taking a "half-step" of the Euler explicit method:

$$y^\star(t_{i+1/2}) = y(t_i) + F(t_i, y(t_i))\Delta t_i/2 \tag{4}$$

Write a function with the following header:

```
function [t, y] = my_midpoint(f, t0, tf, y0, n)
```

This function uses the same input and output arguments as `my_explicit_euler`, except that it uses the midpoint method instead of the explicit Euler method to approximate the solution of the differential equation represented by `f`.

Test cases:

```
>> % Test case 1
>> y0 = 10;
>> f = @(t, y) -0.5*y;
>> [t, y] = my_midpoint(f, 0, 5, y0, 10)
t =
  Columns 1 through 7
```

```
         0    0.5000    1.0000    1.5000    2.0000    2.5000    3.0000
  Columns 8 through 11
    3.5000    4.0000    4.5000    5.0000
y =
  Columns 1 through 7
   10.0000    7.8125    6.1035    4.7684    3.7253    2.9104    2.2737
  Columns 8 through 11
    1.7764    1.3878    1.0842    0.8470

>> % Test case 2
>> y0 = 10;
>> f = @(t, y) -0.05 .* t .* y;
>> [t, y] = my_midpoint(f, 1, 5, y0, 20)
t =
  Columns 1 through 7
    1.0000    1.2000    1.4000    1.6000    1.8000    2.0000    2.2000
  Columns 8 through 14
    2.4000    2.6000    2.8000    3.0000    3.2000    3.4000    3.6000
  Columns 15 through 21
    3.8000    4.0000    4.2000    4.4000    4.6000    4.8000    5.0000
y =
  Columns 1 through 7
   10.0000    9.8905    9.7627    9.6173    9.4551    9.2771    9.0842
  Columns 8 through 14
    8.8776    8.6583    8.4276    8.1866    7.9366    7.6789    7.4147
  Columns 15 through 21
    7.1453    6.8719    6.5958    6.3182    6.0401    5.7627    5.4871

>> % Test case 3
>> y0 = 5;
>> f = @(t, y) -0.1 * y.^2;
>> [t, y] = my_midpoint(f, 1, 10, y0, 24)
t =
  Columns 1 through 7
    1.0000    1.3750    1.7500    2.1250    2.5000    2.8750    3.2500
  Columns 8 through 14
    3.6250    4.0000    4.3750    4.7500    5.1250    5.5000    5.8750
  Columns 15 through 21
    6.2500    6.6250    7.0000    7.3750    7.7500    8.1250    8.5000
  Columns 22 through 25
    8.8750    9.2500    9.6250   10.0000
y =
  Columns 1 through 7
    5.0000    4.2300    3.6613    3.2252    2.8809    2.6024    2.3726
  Columns 8 through 14
    2.1799    2.0159    1.8748    1.7521    1.6445    1.5492    1.4644
  Columns 15 through 21
    1.3883    1.3197    1.2576    1.2011    1.1494    1.1020    1.0583
  Columns 22 through 25
    1.0179    0.9805    0.9458    0.9134
```

1.3. Fourth-order Runge-Kutta

The fourth-order Runge-Kutta method uses the following formula:

$$y(t_{i+1}) = y(t_i) + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)\Delta t_i \tag{5}$$

where $\Delta t_i = t_{i+1} - t_i$ and k_1, k_2, k_3, and k_4 are given by:

$$k_1 = F(t_i, y(t_i)) \tag{6}$$
$$k_2 = F(t_i + \Delta t_i/2, y(t_i) + k_1\Delta t_i/2) \tag{7}$$
$$k_3 = F(t_i + \Delta t_i/2, y(t_i) + k_2\Delta t_i/2) \tag{8}$$
$$k_4 = F(t_i + \Delta t_i, y(t_i) + k_3\Delta t_i) \tag{9}$$

Write a function with the following header:

```
function [t, y] = my_rk4(f, t0, tf, y0, n)
```

This function uses the same input and output arguments as `my_explicit_euler` except that it uses the fourth-order Runge-Kutta method instead of the explicit Euler method to approximate the solution of the differential equation represented by `f`.

Test cases:

```
>> % Test case 1
>> y0 = 10;
>> f = @(t, y) -0.5*y;
>> [t, y] = my_rk4(f, 0, 5, y0, 10)
t =
  Columns 1 through 7
         0    0.5000    1.0000    1.5000    2.0000    2.5000    3.0000
  Columns 8 through 11
    3.5000    4.0000    4.5000    5.0000
y =
  Columns 1 through 7
   10.0000    7.7881    6.0654    4.7238    3.6789    2.8652    2.2314
  Columns 8 through 11
    1.7379    1.3535    1.0541    0.8209

>> % Test case 2
>> y0 = 10;
>> f = @(t, y) -0.05 .* t .* y;
>> [t, y] = my_rk4(f, 1, 5, y0, 20)
t =
  Columns 1 through 7
    1.0000    1.2000    1.4000    1.6000    1.8000    2.0000    2.2000
  Columns 8 through 14
    2.4000    2.6000    2.8000    3.0000    3.2000    3.4000    3.6000
  Columns 15 through 21
    3.8000    4.0000    4.2000    4.4000    4.6000    4.8000    5.0000
y =
  Columns 1 through 7
   10.0000    9.8906    9.7629    9.6175    9.4554    9.2774    9.0846
  Columns 8 through 14
    8.8781    8.6589    8.4282    8.1873    7.9374    7.6797    7.4156
  Columns 15 through 21
    7.1462    6.8729    6.5968    6.3192    6.0411    5.7637    5.4881

>> % Test case 3
>> y0 = 5;
>> f = @(t, y) -0.1 * y.^2;
>> [t, y] = my_rk4(f, 1, 10, y0, 24)
t =
  Columns 1 through 7
    1.0000    1.3750    1.7500    2.1250    2.5000    2.8750    3.2500
  Columns 8 through 14
    3.6250    4.0000    4.3750    4.7500    5.1250    5.5000    5.8750
  Columns 15 through 21
    6.2500    6.6250    7.0000    7.3750    7.7500    8.1250    8.5000
  Columns 22 through 25
    8.8750    9.2500    9.6250   10.0000
y =
  Columns 1 through 7
    5.0000    4.2105    3.6364    3.2000    2.8572    2.5807    2.3530
  Columns 8 through 14
    2.1622    2.0000    1.8605    1.7391    1.6327    1.5385    1.4546
  Columns 15 through 21
    1.3793    1.3115    1.2500    1.1940    1.1429    1.0959    1.0526
  Columns 22 through 25
    1.0127    0.9756    0.9412    0.9091
```

2. Application to solve a specific equation

In this question, you will apply the generic methods for solving ordinary differential equations that you wrote in the previous question to numerically solve the following ordinary differential equation:

$$y' + y = te^{-t} \tag{10}$$

between the initial time $t_0 = 0$ and the final time t_f given the following initial condition:

$$y(t = t_0 = 0) = y_0 \tag{11}$$

The analytical solution to this initial value problem is:

$$y(t) = e^{-t}\left(\frac{t^2}{2} + y_0\right) \tag{12}$$

Write a function with the following header:

```
function [t, y_numerical, y_analytical, rmse] = my_ode(tf, y0, n, method)
```

where:

- `tf` is a scalar of class `double` that represents the final time t_f.
- `n` is a scalar of class `double` that represents the number of equally-spaced time steps to use when numerically solving the ordinary differential equation on the interval $[t_0, t_f]$ (t_0 and t_f included). You can assume that `n > 1`.
- `method` is a row vector of class `char` (*i.e.* a character string) that can take one of the following four values:
  - ◇ `'euler'`: in this case, use the explicit Euler method.
  - ◇ `'midpoint'`: in this case, use the midpoint method.
  - ◇ `'rk4'`: in this case, use the fourth-order Runge-Kutta method.
  - ◇ `'ode45'`: in this case, use Matlab's built-in function `ode45`, forcing this function to calculate the numerical solution at times `t` as defined below, as opposed to letting this function determine the sizes of the time steps to take.
- `t` is an $1 \times (n+1)$ array of class `double` that represents, in increasing order, the equally-spaced points in the interval $[t_0, t_f]$ used when numerically solving the ordinary differential equation.
- `y_numerical` is $1 \times (n+1)$ array of class `double` that represents the values of the function y at times `t` (in the same order) as approximated by the numerical solver.
- `y_analytical` is $1 \times (n+1)$ array of class `double` that represents the values of the function y at times t (in the same order) as calculated using the analytical solution.
- `rmse` is a scalar of class `double` that represents the root mean square error between the numerical solution and the analytical solution. The root mean square error is given by:

$$\sqrt{\frac{1}{n+1}\sum_{i=1}^{n+1}(y_{\text{numerical}}(i) - y_{\text{analytical}}(i))^2} \tag{13}$$

Test cases:

```
>> [t, y_numerical, y_analytical, rmse] = my_ode(3, 2, 10, 'euler')
t =
  Columns 1 through 7
         0    0.3000    0.6000    0.9000    1.2000    1.5000    1.8000
  Columns 8 through 11
    2.1000    2.4000    2.7000    3.0000
y_numerical =
  Columns 1 through 7
    2.0000    1.4000    1.0467    0.8315    0.6918    0.5927    0.5153
  Columns 8 through 11
    0.4500    0.3921    0.3398    0.2923
y_analytical =
  Columns 1 through 7
    2.0000    1.5150    1.1964    0.9778    0.8192    0.6973    0.5984
  Columns 8 through 11
    0.5149    0.4427    0.3794    0.3236
rmse =
    0.0956

>> [t, y_numerical, y_analytical, rmse] = my_ode(3, 2, 10, 'midpoint')
t =
  Columns 1 through 7
         0    0.3000    0.6000    0.9000    1.2000    1.5000    1.8000
  Columns 8 through 11
    2.1000    2.4000    2.7000    3.0000
y_numerical =
  Columns 1 through 7
    2.0000    1.5287    1.2150    0.9966    0.8363    0.7117    0.6102
  Columns 8 through 11
    0.5245    0.4503    0.3854    0.3284
y_analytical =
  Columns 1 through 7
    2.0000    1.5150    1.1964    0.9778    0.8192    0.6973    0.5984
  Columns 8 through 11
    0.5149    0.4427    0.3794    0.3236
rmse =
    0.0126

>> [t, y_numerical, y_analytical, rmse] = my_ode(3, 2, 10, 'rk4')
t =
  Columns 1 through 7
         0    0.3000    0.6000    0.9000    1.2000    1.5000    1.8000
  Columns 8 through 11
    2.1000    2.4000    2.7000    3.0000
y_numerical =
  Columns 1 through 7
    2.0000    1.5150    1.1965    0.9779    0.8193    0.6973    0.5984
  Columns 8 through 11
    0.5150    0.4427    0.3794    0.3236
y_analytical =
  Columns 1 through 7
    2.0000    1.5150    1.1964    0.9778    0.8192    0.6973    0.5984
  Columns 8 through 11
    0.5149    0.4427    0.3794    0.3236
rmse =
   4.8180e-05
```

```
>> [t, y_numerical, y_analytical, rmse] = my_ode(3, 2, 10, 'ode45')
t =
  Columns 1 through 7
         0    0.3000    0.6000    0.9000    1.2000    1.5000    1.8000
  Columns 8 through 11
    2.1000    2.4000    2.7000    3.0000
y_numerical =
  Columns 1 through 7
    2.0000    1.5150    1.1964    0.9778    0.8192    0.6973    0.5984
  Columns 8 through 11
    0.5149    0.4427    0.3794    0.3236
y_analytical =
  Columns 1 through 7
    2.0000    1.5150    1.1964    0.9778    0.8192    0.6973    0.5984
  Columns 8 through 11
    0.5149    0.4427    0.3794    0.3236
rmse =
   1.4376e-06

>> [t, y_numerical, y_analytical, rmse] = my_ode(3, 2, 20, 'rk4')
t =
  Columns 1 through 7
         0    0.1500    0.3000    0.4500    0.6000    0.7500    0.9000
  Columns 8 through 14
    1.0500    1.2000    1.3500    1.5000    1.6500    1.8000    1.9500
  Columns 15 through 21
    2.1000    2.2500    2.4000    2.5500    2.7000    2.8500    3.0000
y_numerical =
  Columns 1 through 7
    2.0000    1.7311    1.5150    1.3398    1.1964    1.0776    0.9778
  Columns 8 through 14
    0.8928    0.8193    0.7547    0.6973    0.6455    0.5984    0.5550
  Columns 15 through 21
    0.5149    0.4776    0.4427    0.4100    0.3794    0.3506    0.3236
y_analytical =
  Columns 1 through 7
    2.0000    1.7311    1.5150    1.3398    1.1964    1.0776    0.9778
  Columns 8 through 14
    0.8928    0.8192    0.7547    0.6973    0.6455    0.5984    0.5550
  Columns 15 through 21
    0.5149    0.4776    0.4427    0.4100    0.3794    0.3506    0.3236
rmse =
   2.6406e-06

>> [t, y_numerical, y_analytical, rmse] = my_ode(1, 4, 15, 'ode45')
t =
  Columns 1 through 7
         0    0.0667    0.1333    0.2000    0.2667    0.3333    0.4000
  Columns 8 through 14
    0.4667    0.5333    0.6000    0.6667    0.7333    0.8000    0.8667
  Columns 15 through 16
```

```
    0.9333    1.0000
y_numerical =
  Columns 1 through 7
    4.0000    3.7441    3.5085    3.2913    3.0909    2.9059    2.7349
  Columns 8 through 14
    2.5766    2.4300    2.2940    2.1678    2.0504    1.9411    1.8393
  Columns 15 through 16
    1.7442    1.6555
y_analytical =
  Columns 1 through 7
    4.0000    3.7441    3.5085    3.2913    3.0909    2.9059    2.7349
  Columns 8 through 14
    2.5766    2.4300    2.2940    2.1678    2.0504    1.9411    1.8393
  Columns 15 through 16
    1.7442    1.6555
rmse =
   9.0640e-09
```

3. Elevated water tank

A model for an elevated water tank can be simplified into the model shown in Figure 1.

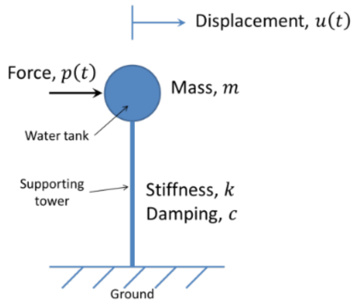

Figure 1: Structural model for an elevated water tank.

The mass of the supporting tower is assumed to be negligible compared to the mass m of the water tank. If a horizontal force of magnitude p is applied to the tank, the tank will move horizontally. We assume that the horizontal displacement u of the tank is small enough so that the vertical displacement of the tank is negligible. The tower supporting the tank is characterized by its stiffness k and its damping coefficient c. In this question, the horizontal displacement of the tank is modeled by the following second-order differential equation:

$$mu''(t) + cu'(t) + ku(t) = p(t) \tag{14}$$

Note that $u'(t)$ is the horizontal velocity of the tank as a function of time t, and $u''(t)$ is the horizontal acceleration of the tank as a function of time t.

In this question, you will write a function that approximates the horizontal displacement $u(t)$ of the tank as a function of time given the horizontal position $u_0 = u(t = 0)$ of the tank and the horizontal force $p(t)$ applied to the tank as a function of time. More precisely, write a function with the following header:

```
function [t, response] = my_water_tank(m, c, k, p, dt, initial_state)
```

where:

- `m`, `c`, and `k` are scalars of class `double` that represent m, c, and k in Equation 14, respectively.
- `p` is a $1 \times n$ array of class `double` that represents the horizontal force applied to the water tank at each time at which your function will calculate a numerical approximate solution to Equation 14. `p(i)` represents $p((i-1)\Delta t)$.
- `dt` is a scalar of class `double` that represents the time step Δt that your numerical solver will use.
- `initial_state` is a 2×1 array of class `double` where the first element represents the initial horizontal displacement u_0 of the tank, and where the second element represents the initial horizontal velocity $u'(t = 0)$ of the tank.
- `t` is a $1 \times n$ array of class `double` that represents, in increasing order, the equally-spaced points at which your function will calculate an approximate solution to Equation 14.
- `response` is a $2 \times n$ array of class `double` where the first row represents the values of the horizontal displacement u of the tank at each time step, and where the second row represents the values of the horizontal velocity u' of the tank at each time step. These values should be calculated by applying **Heun's method** to Equation 14 with time step Δt.

The numerical methods to solve ordinary differential equations that we learned only apply to first-order ordinary differential equations. However, it is possible to rewrite Equation 14 as a system of first-order ordinary differential equations by defining two new functions:

$$z_1 = u \tag{15}$$

$$z_2 = u' \tag{16}$$

We then have:

$$z_1(t)' = z_2(t) \tag{17}$$

$$z_2(t)' = -\frac{c}{m} z_2(t) - \frac{k}{m} z_1(t) + \frac{1}{m} p(t) \tag{18}$$

We can rewrite this system in the following form:

$$z' = G(t, z) \tag{19}$$

with:

$$z = \begin{bmatrix} z_1 \\ z_2 \end{bmatrix} \tag{20}$$

and

$$G(t, z) = \begin{bmatrix} z_2(t) \\ -\frac{c}{m} z_2(t) - \frac{k}{m} z_1(t) + \frac{1}{m} p(t) \end{bmatrix} \tag{21}$$

Heun's method can then be applied to Equation 19.

Assume that the units of all the input arguments to your function are such that you do not have to do any unit conversion in this question.

Test cases (see Figures 2 through 7):

```
% Load the data
% Units are such that u is in inches,
% u' is in inches per seconds, p is in kips,
% and t is in seconds
load Lab12_WaterTank.mat

% Run all the test cases and plot the results
for test_case = 1:6

    m = Lab12_WaterTank(test_case).m;
    c = Lab12_WaterTank(test_case).c;
    k = Lab12_WaterTank(test_case).k;
    p = Lab12_WaterTank(test_case).p
    dt = Lab12_WaterTank(test_case).dt;
    initial_state = Lab12_WaterTank(test_case).u0;

    [t, response] = my_water_tank(m, c, k, p, dt, initial_state);

    % Plot the results
    fig = figure();
    hold on
    subplot(2, 1, 1)
    plot(t, response(1,:))
    xlabel('Time (s)')
    ylabel('Relative displacement (in)')
    xlim([min(t), max(t)])
    title('Relative displacement versus time')

    subplot(2, 2, 3)
    plot(t, response(2,:))
    xlim([min(t), max(t)])
    xlabel('Time (s)')
    ylabel('Velocity (in/s)')
    title('Velocity versus time')

    subplot(2, 2, 4)
    plot(t, p)
    xlim([min(t), max(t)])
    xlabel('Time (s)')
    ylabel('Force (kips)')
    title('Force versus time')

end

% Show some of the values of the test case number 6
t(1:10)
ans =
  Columns 1 through 7
         0    0.0100    0.0200    0.0300    0.0400    0.0500    0.0600
  Columns 8 through 10
    0.0700    0.0800    0.0900
response(1:10)
ans =
  Columns 1 through 7
         0         0   -0.0000   -0.0099   -0.0002   -0.0197   -0.0004
  Columns 8 through 10
   -0.0294   -0.0008   -0.0389
t(end-10:end)
ans =
  Columns 1 through 7
   29.8900   29.9000   29.9100   29.9200   29.9300   29.9400   29.9500
  Columns 8 through 11
   29.9600   29.9700   29.9800   29.9900
response(:,end-10:end)
ans =
  Columns 1 through 7
    0.1421    0.2658    0.3891    0.5117    0.6336    0.7545    0.8745
   12.3975   12.3515   12.2933   12.2232   12.1411   12.0473   11.9417
  Columns 8 through 11
    0.9934    1.1110    1.2273    1.3421
   11.8246   11.6961   11.5563   11.4055
```

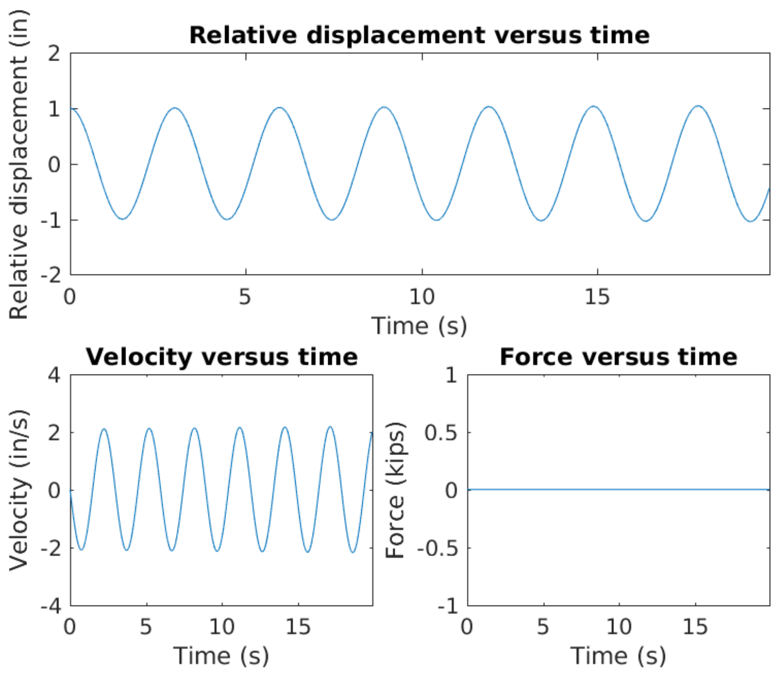

Figure 2: Test case 1: Undamped free vibration

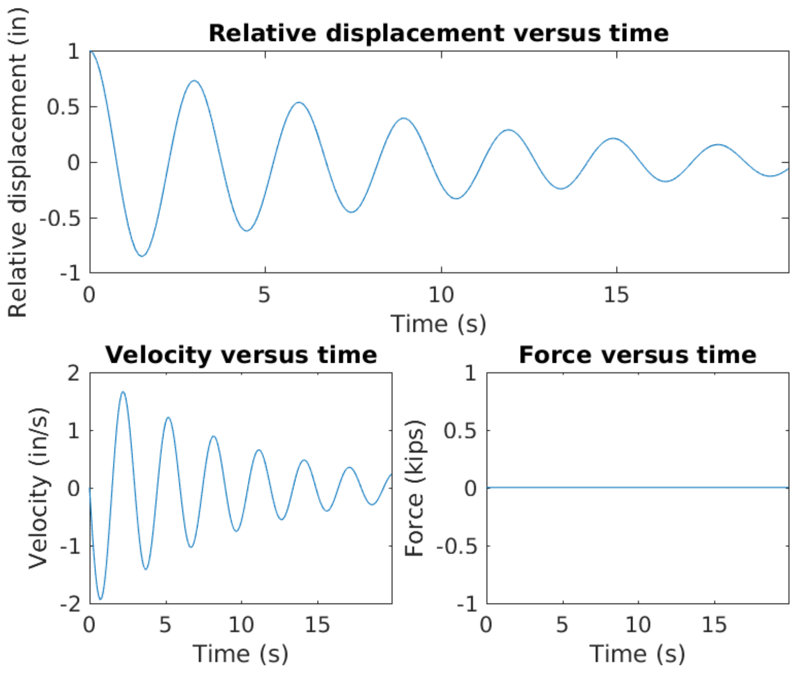

Figure 3: Test case 2: Zero applied force with initial displacement

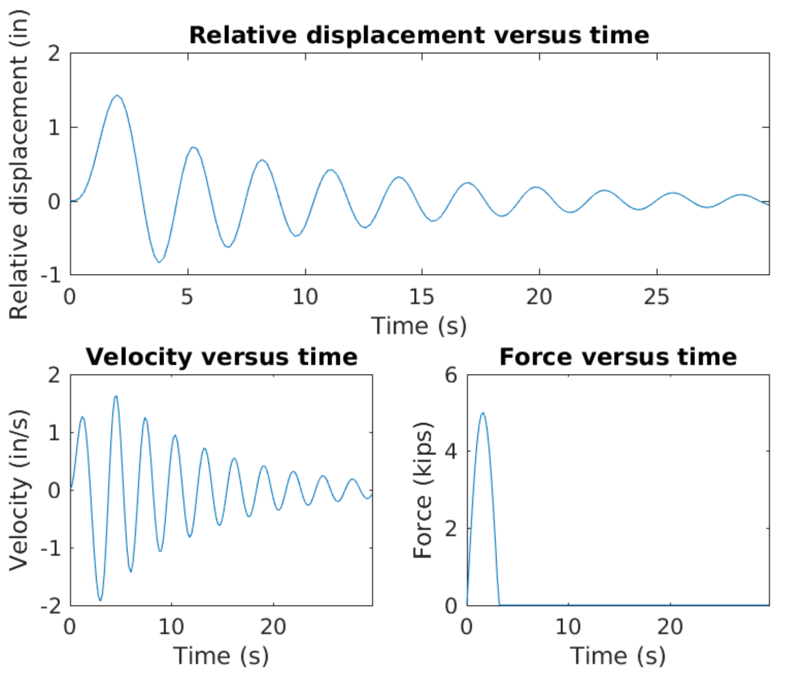

Figure 4: Test case 3: Sine wave with zero initial conditions

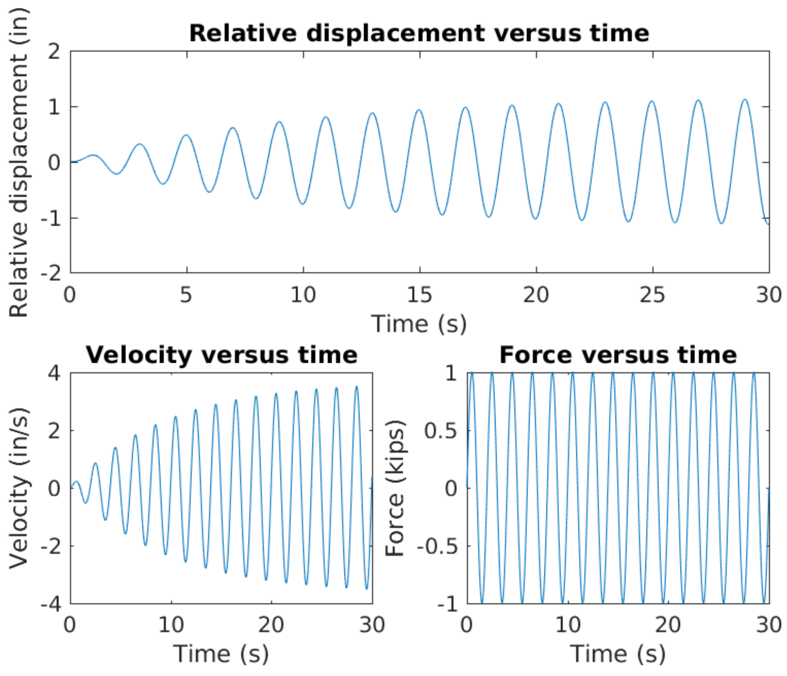

Figure 5: Test case 4: Resonant sine wave

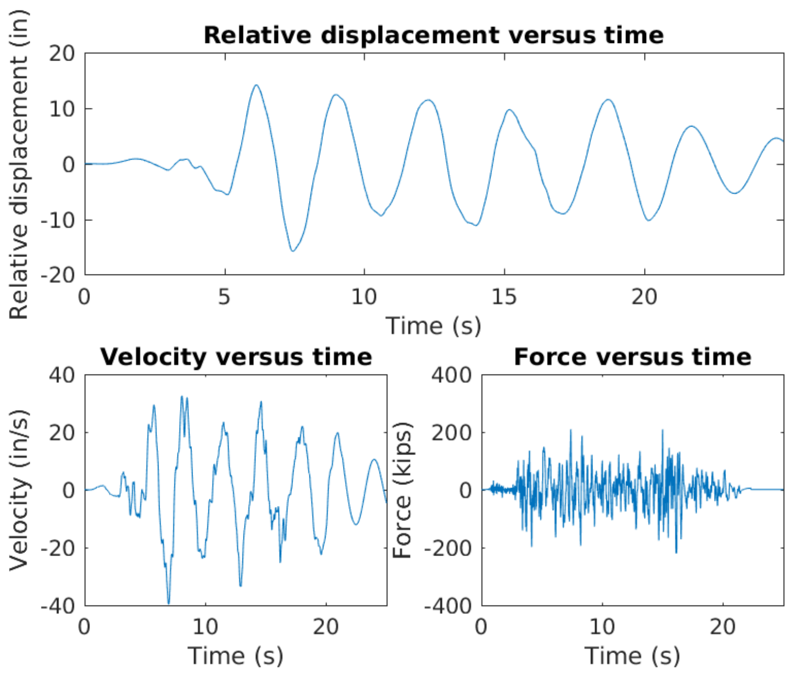

Figure 6: Test case 5: Earthquake RSN725_SUPER.B_B-POE360.AT2, data adapted from ground motion data by the Pacific Earthquake Engineering Research Center (PEER, http://ngawest2.berkeley.edu/).

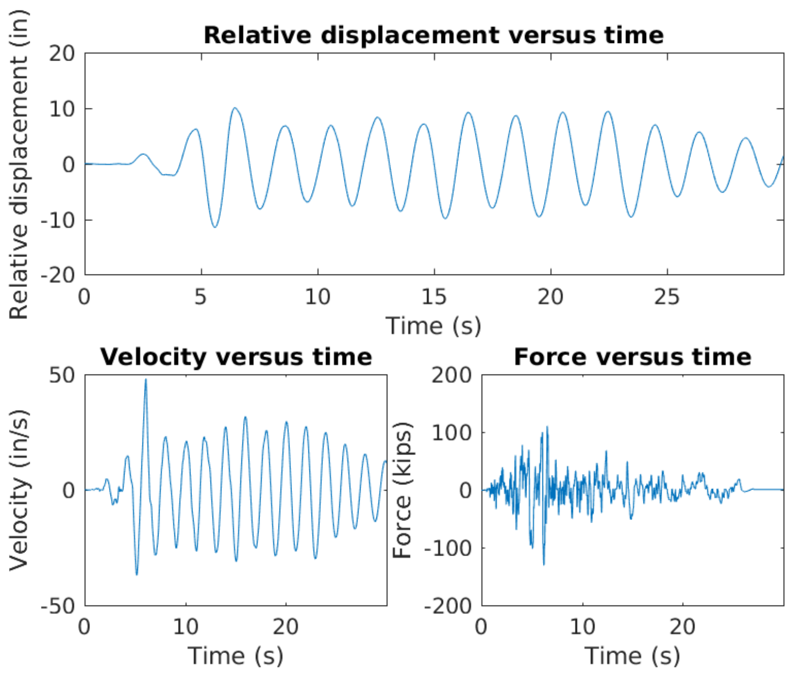

Figure 7: Test case 6: Earthquake RSN266_VICT_CHI102.AT2, data adapted from ground motion data by the Pacific Earthquake Engineering Research Center (PEER, http://ngawest2.berkeley.edu/).