HW5
CS 6210

Spencer Peterson

12/16/2021

1 Simpson's Rule

a. Implementation

My implementation of Simpson's rule is shown below.

```
function Area = simpsons_rule(F, a, b, N)
% Implementation of simpsons rule. F is a function handle, a and b are the
% starting and end points. N is the number of points to evaluate on.
    width = (b-a)/N;

    points = width:width:b;

    Area = sum(F(points) .* width);

end
```

b. First Function

The results for $f(x) = x^p$ are shown in the table below.

p	Analytical Solution	N=17	N=33	N=65	N=129	N=257	N=513
2	1/3=0.33333	0.3633	0.3486	0.3411	0.3372	0.3353	0.3343
3	1/4=0.25	0.2803	0.2654	0.2578	0.2539	0.2519	0.2510
4	1/5=0.2	0.2306	0.2155	0.2078	0.2039	0.2020	0.2010
5	1/6=0.16667	0.1975	0.1822	0.1745	0.1706	0.1686	0.1676
6	1/7=0.14286	0.1740	0.1585	0.1507	0.1468	0.1448	0.1438
8	1/9=0.11111	0.1428	0.1269	0.1190	0.1150	0.1131	0.1121

My implementation of Simpsons' rule doesn't start to converge until N is fairly large. Even then the error is still in the range of 1e-3.

c. Second Function

I also tested it with the function $\int_0^{2\pi} 1 + \sin x \cdot \cos \frac{2x}{3} \cdot \sin 4x \; dx$ The values start to converge once N is greater than 129.

N	17	33	65	129	257	513
Area	6.30473283	6.30514394	6.30516885	6.30517044	6.30517054	6.30517055

2 QuadTX

The results for the QuadTX expirement are shown below. Overall increasing the tolerance by a factor of 1/10 tended to make the number of function calls grow by a factor of 1.3 and the time either by a similar but less precise amount.

Tolerance	1e-07	1e-08	1e-09	1e-10	1e-11	1e-12	1e-13	1e-14
Function Calls	813	1365	2089	3177	5013	7841	12241	19545
Time	3.82 ms	3.22 ms	4.75 ms	9.08 ms	19.93 ms	17.65 ms	37.86 ms	46.89 ms

3 Comparison of Methods

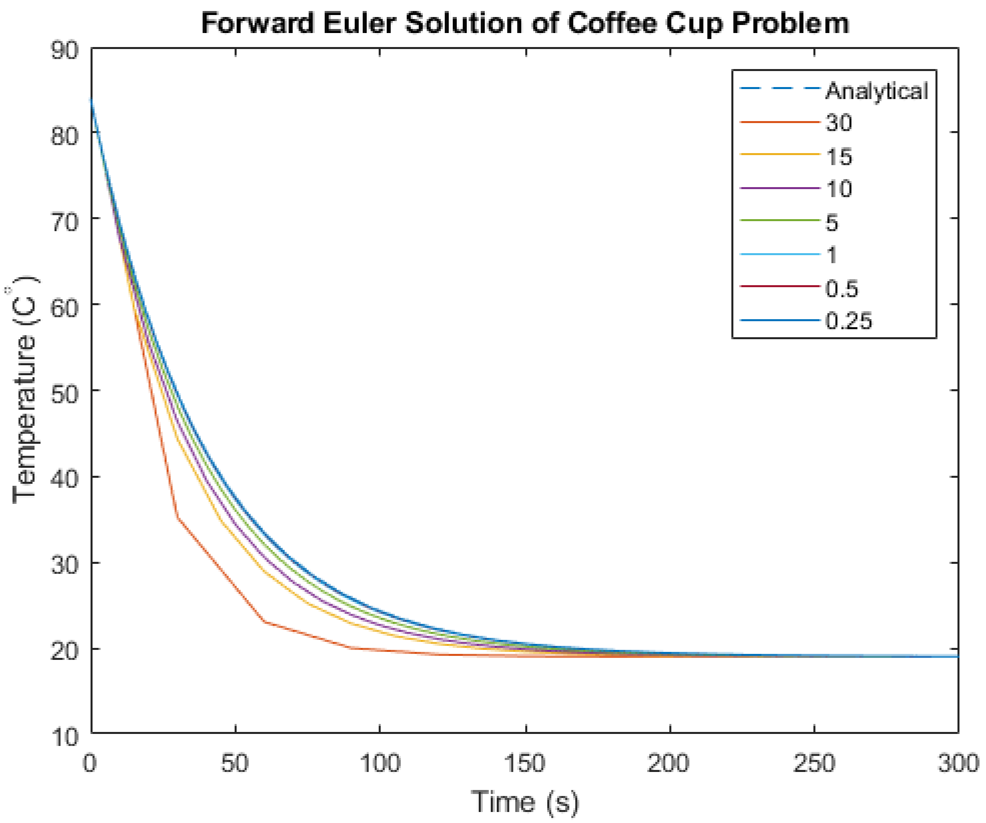
4 Coffee Cup

a. Analytical Solution

$$\begin{aligned}
\frac{dT_c}{dt} &= -r(T_c - T_s) \\
T_c &= Ae^{-rt} + T_s \\
T_c(0) &= 84 = A + 19 \\
A &= 65 \\
T_c &= 65e^{-0.025 \cdot t} + 19
\end{aligned}$$

b. Forward Euler

The plot of the Forward Euler method results are shown below:



The error after the first step is shown in the table below:

h	30	15	10	5	1	0.5	0.25
Error	1.445e+01	4.049e+00	1.872e+00	4.873e-01	2.014e-02	5.057e-03	1.267e-03

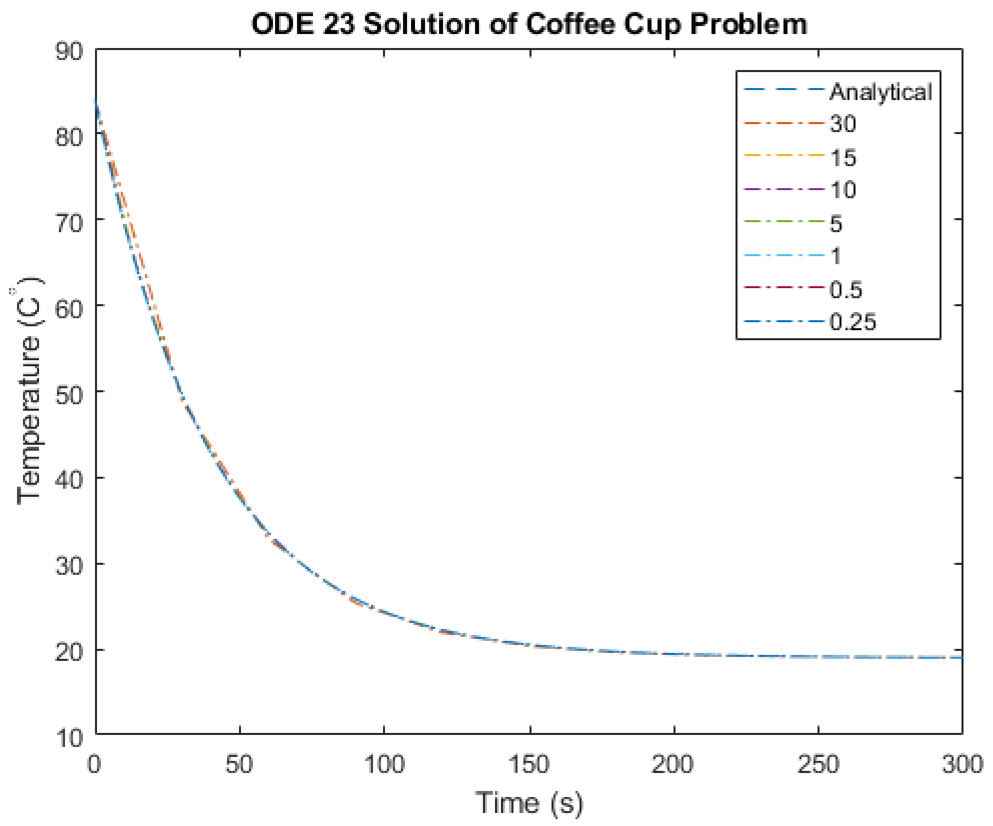
The error seems to be linearly related to the step size h.

5 ODE23 Implementation

Below is my implementation of the ODE23 method.

```
function [t, y, error] = MyODE23(f, t0, tEnd, y0, h)
    % Implementation of ODE23 Method for problem 5 of homework 5 CS 6210.

    curr_y = y0;
    curr_t = t0;
    ys = [curr_y];
    times = [curr_t];
    errors = [];

    while curr_t < tEnd
        S1 = f(curr_t, curr_y);

        S2 = f(curr_t + h/2, curr_y + h/2 * S1);

        S3 = f(curr_t + 3*h/4, curr_y + 3 * h/ 4 * S2);

        next_t = curr_t + h;

        next_y = curr_y + h/9 * (2 * S1 + 3 * S2 + 4 * S3);


        S4 = f(next_t, next_y);
        curr_error = h/72 * (-5 * S1 + 6 * S2 + 8*S3 - 9 * S4);

        times = [times next_t];
        ys = [ys next_y];

        errors = [errors curr_error];
        curr_t = next_t;
        curr_y = next_y;

    end
    t = times;
    y = ys;
    error = errors;
end
```

6 ODE23 Plot



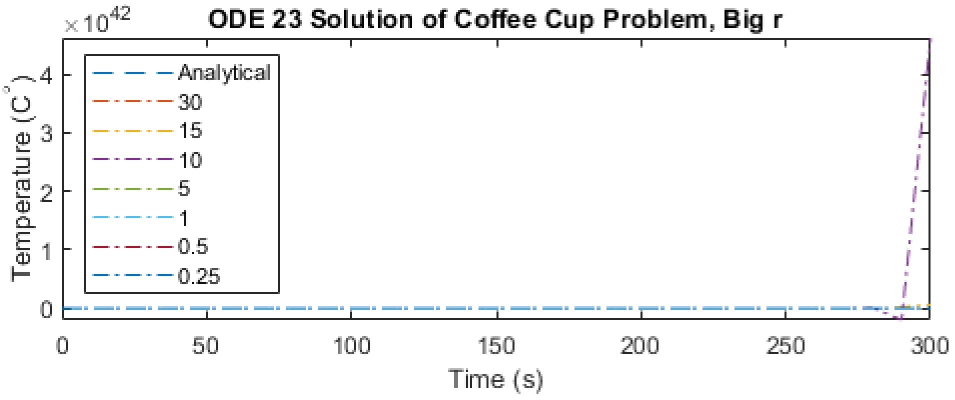
The predicted and actual errors for the first time step are shown below:

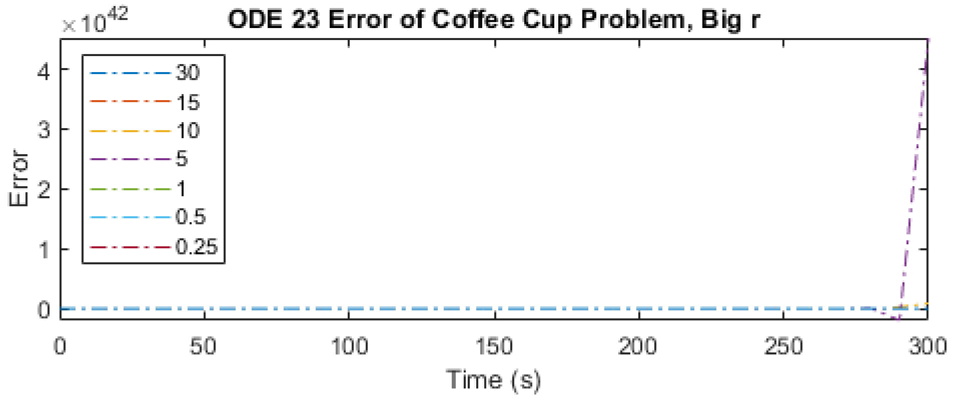
h	30	15	10	5	1	0.5	0.25
Predicted Error	6.583e-02	3.064e-02	1.236e-02	2.042e-03	2.012e-05	2.579e-06	3.265e-07
Actual Error	7.429e-01	4.978e-02	1.007e-02	6.450e-04	1.053e-06	6.596e-08	4.127e-09

The actual error is larger than predicted for a large h and smaller than predicted for a small h.

7 Larger r

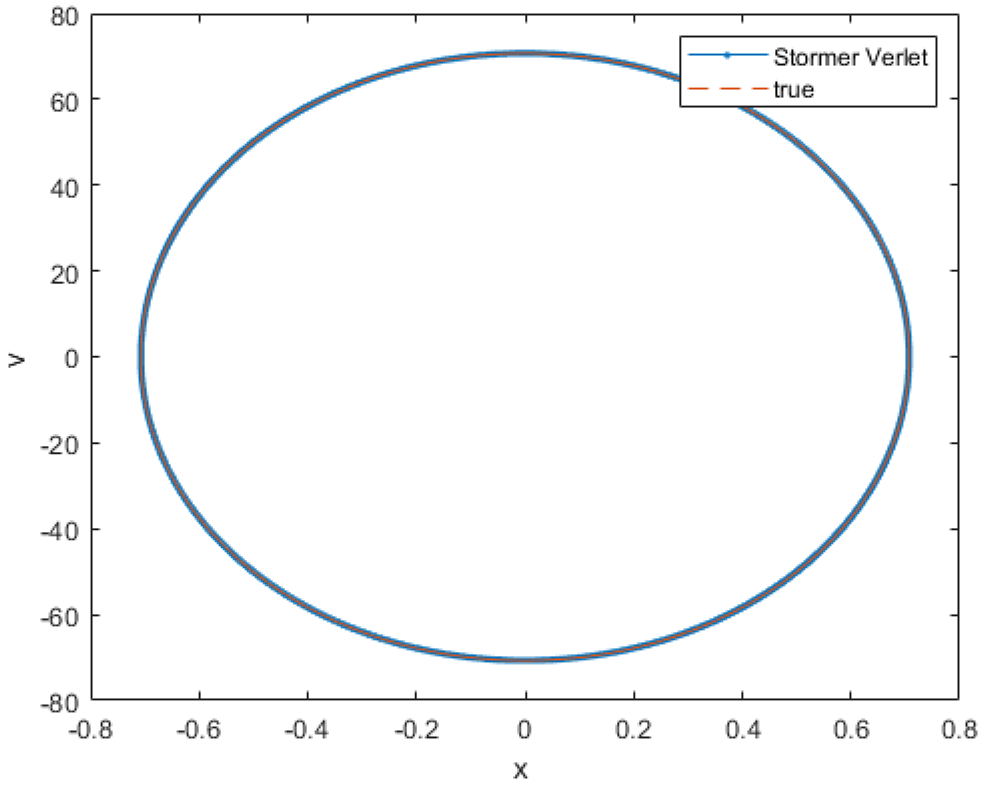
The method does blow up for certain values of h. The error estimator also blows up when a solution blows up.





8 Stormer Verlet Method

The results of the Stormer Verlet method are shown below. I used a time step of 1e-3 seconds for all trials.



I measured the error in the Hamiltonian by finding the infinity norm of the difference of the analytical Hamiltonian and the calculated Hamiltonian. The result was 6.250. This is within the same order of magnitude that we got in class.

9 Trapezoidal and Backward Euler Methds

I implemented the Reverse Euler as well using an inversion of a 2x2 matrix. The derivation of this matrix is shown below:

$$x(t+1) = x(t) + dt * v(t)$$
$$v(t+1) = v(t) - dt\omega^2 x(t)$$

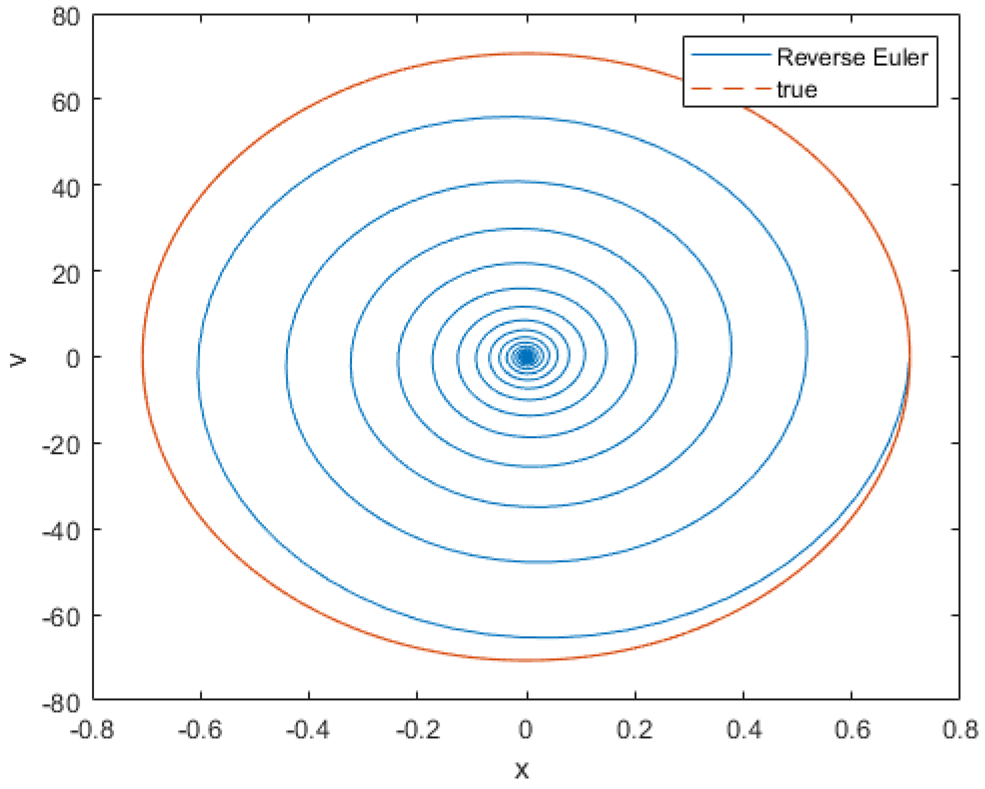
$$x(t+1) - dt * v(t) = x(t)$$
$$dt\omega^2 x(t) + v(t+1) = v(t)$$

$$\begin{bmatrix} 1 & -dt \\ dt\omega^2 & 1 \end{bmatrix} \begin{bmatrix} x(t+1) \\ v(t+1) \end{bmatrix} = \begin{bmatrix} x(t) \\ v(t) \end{bmatrix}$$
$$A \begin{bmatrix} x(t+1) \\ v(t+1) \end{bmatrix} = \begin{bmatrix} x(t) \\ v(t) \end{bmatrix}$$
$$\begin{bmatrix} x(t+1) \\ v(t+1) \end{bmatrix} = A^{-1} \begin{bmatrix} x(t) \\ v(t) \end{bmatrix}$$

The code was implemented as follows. Please see code for the full context as this is just the matrix math as it relates to what was determined above.

```
A = [1 -dt; dt*omega^2 1]; % Matrix to be inverted for implicit Euler's
Ainv = inv(A);

temp = Ainv * [x_re(step); v_re(step)];
x_re(step+1) = temp(1);
v_re(step+1) = temp(2);
```

The results are shown below:



The Hamiltonian is not preserved and using the same method as above for the Stormer Verlet method, the erorr was 2.5e3.
Additionally, I implemented the Trapezoidal method. The derivation for the matrices used is shown below:

$$\begin{aligned} x(t+1) &= x(t) + \frac{dt}{2}(v(t) + v(t+1)) \\ v(t+1) &= v(t) + \frac{dt}{2}(-\omega^2 x(t) - \omega^2 x(t+1)) \end{aligned}$$

$$\begin{aligned} x(t+1) - \frac{dt}{2}v(t+1) &= x(t) + \frac{dt}{2}v(t) \\ \frac{dt}{2}\omega^2 x(t+1) + v(t+1) &= -\frac{dt}{2}\omega^2 x(t) + v(t) \end{aligned}$$

$$\begin{bmatrix} 1 & -\frac{dt}{2} \\ \frac{dt}{2}\omega^2 & 1 \end{bmatrix} \begin{bmatrix} x(t+1) \\ v(t+1) \end{bmatrix} = \begin{bmatrix} 1 & \frac{dt}{2} \\ -\frac{dt}{2}\omega^2 & 1 \end{bmatrix} \begin{bmatrix} x(t) \\ v(t) \end{bmatrix}$$

$$B \begin{bmatrix} x(t+1) \\ v(t+1) \end{bmatrix} = C \begin{bmatrix} x(t) \\ v(t) \end{bmatrix}$$

$$\begin{bmatrix} x(t+1) \\ v(t+1) \end{bmatrix} = B^{-1}C \begin{bmatrix} x(t) \\ v(t) \end{bmatrix}$$

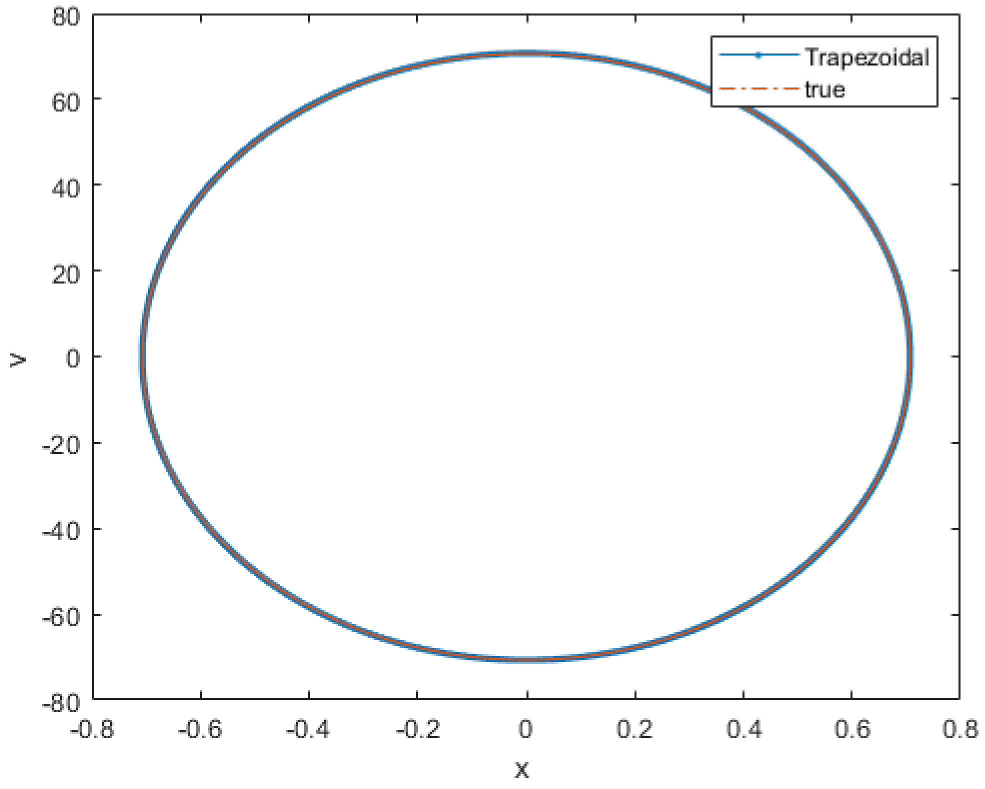
The matrices B and C were calculated as follows for the implementation of this method.

```
Binv = inv([1 -1/2*dt; 1/2*dt*omega^2 1]);
C = [1 1/2*dt; -1/2*dt*omega^2 1];
```

And they were used to calculate the next step as follows:

```
temp = Binv * C * [x_tr(step); v_tr(step)];
x_tr(step+1) = temp(1);
v_tr(step+1) = temp(2);
```

The results for this method are shown below:



The error for the Hamiltonian was calculated as 1.55e-9. This is much smaller than any of the other methods we've tried in class.