Assignment 9 :

By : Abrar Imon

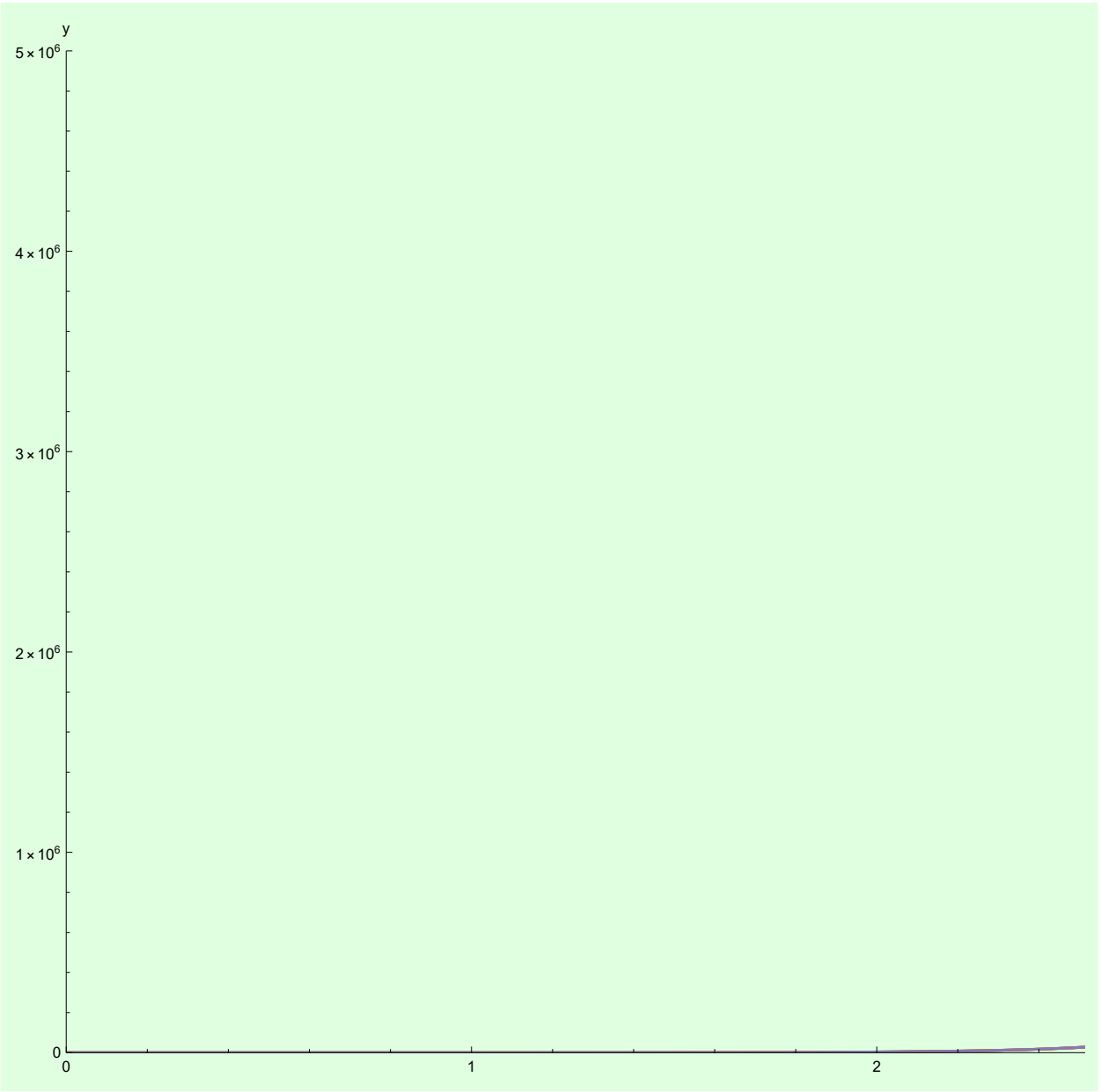
In[ ]:=
```
(*This cell sets up the differential equation and the analytical solution to it.*)
yPrime[t_, y_] := 1 - t + 4 y;
y[t_] := N[t / 4 - 3 / 16 + (19 / 16) E^(4 t), 32];
```

In[ ]:=
```
(*This cell creates a function for the Runge-Kutta method.*)
rungeKutta[diffEq_, initialT_, initialY_, h_, finalT_] := (
  t = initialT;
  w = initialY;
  totalSteps = (finalT - initialT) / h; For[i = 0, i < totalSteps, i++,
   w = N[w + (h / 6) * (diffEq[t, w] + 2 * diffEq[t + h / 2, w + h / 2 * diffEq[t, w]] +
         2 * diffEq[t + h / 2, w + h / 2 * diffEq[t + h / 2, w + h / 2 * diffEq[t, w]]] + diffEq[t + h,
          w + h * diffEq[t + h / 2, w + h / 2 * diffEq[t + h / 2, w + h / 2 * diffEq[t, w]]]]), 32];
   t = t + h];
  Return[N[w, 6]]
 )
```

In[ ]:=
```
(*This cell makes lists with the rungeKutta function with different step sizes. The
 list will be in a form that we can easily plot, using ListLinePlot[].*)
rungeKuttaSizePoint05 = Table[{t, rungeKutta[yPrime, 0, 1, .05, t]}, {t, 0, 4, 0.05}];
rungeKuttaSizePoint025 = Table[{t, rungeKutta[yPrime, 0, 1, .025, t]}, {t, 0, 4, 0.025}];
rungeKuttaSizePoint01 = Table[{t, rungeKutta[yPrime, 0, 1, .01, t]}, {t, 0, 4, 0.01}];
rungeKuttaSizePoint001 = Table[{t, rungeKutta[yPrime, 0, 1, .001, t]}, {t, 0, 4, 0.001}];
exactValues = Table[{t, y[t]}, {t, 0, 4, 0.0001}];
```

In[ ]:=
```
(*This cell plots the exact solution and also plots
 our rungeKutta estimate with the different step sizes.*)
ListLinePlot[{rungeKuttaSizePoint05, rungeKuttaSizePoint025, rungeKuttaSizePoint01,
  rungeKuttaSizePoint001, exactValues}, PlotRange → {{0, 4}, {0, 5 * 10^6}},
 PlotLegends → {"h = 0.05", "h = 0.025", "h = 0.01", "h = 0.001", "Exact Solution"},
 ImageSize → 1000, AxesLabel → {"t", "y"}]
```

Out[ ]=



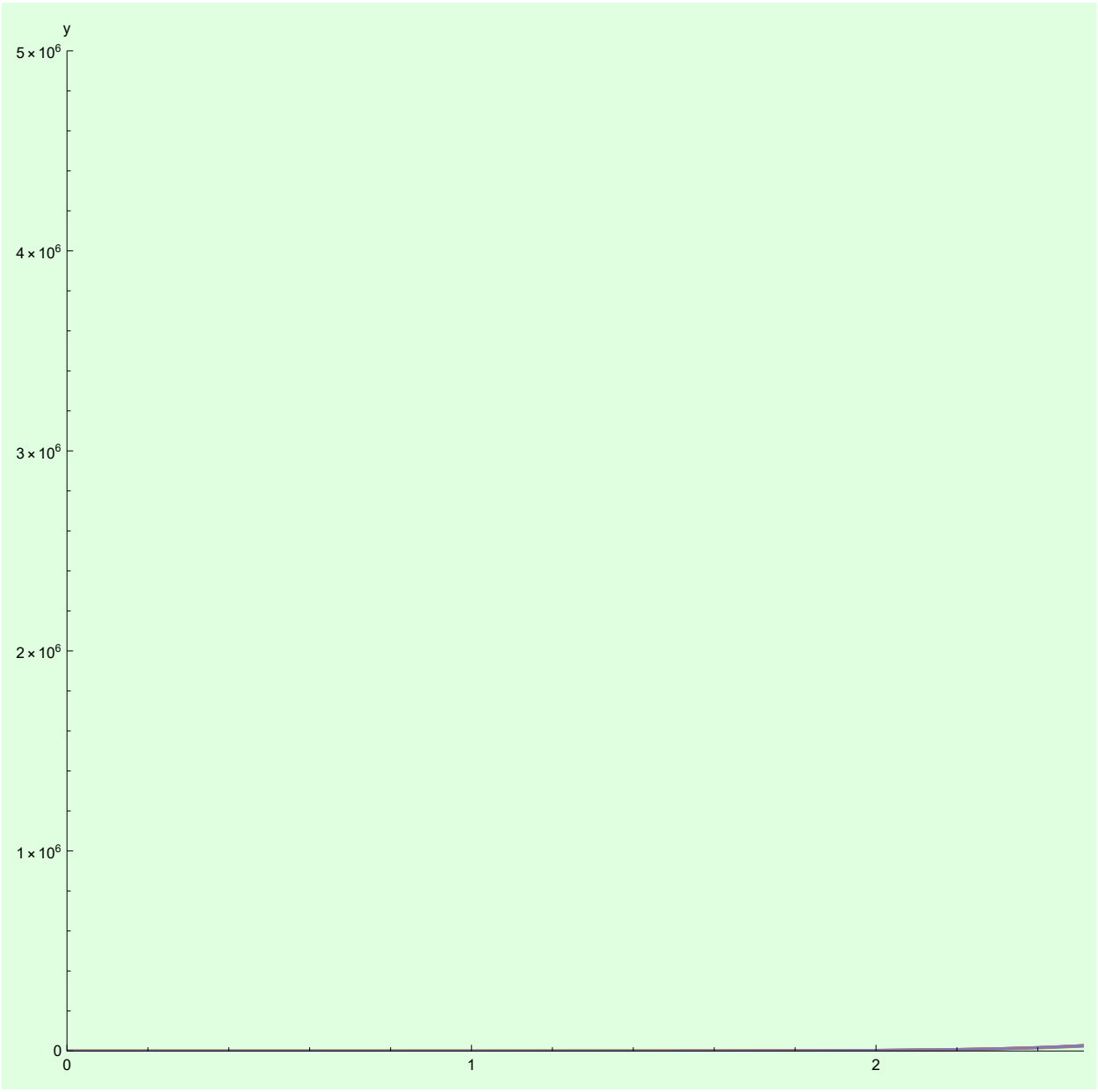
The error of the Runge-Kutta estimation is so low that we cannot see it in this plot. I investigated this by zooming in and noticed that at the right half of the plot, the error was highest for h = 0.05 and the error decreased as the value for h decreased. Later in this assignment, we will have a table that better shows the error for this function. I zoomed out of graph again, to make it more presentable.

In[ ]:=
```
(*This cell creates a function for the second order Adams-
 Bashforth method. This function includes an "If" function so that it
  uses the Euler's method for the first step. This function also uses
  a list called "lis" to refer to an estimate from two steps ago.*)
secondBash[diffEq_, initialT_, initialY_, h_, finalT_] := (
  t = initialT;
  w = initialY;
  lis = {w};
  totalSteps = (finalT - initialT) / h; For[i = 0, i < totalSteps, i++,
   If[i > 0, w = N[w + (h / 2) * (3 * diffEq[t, w] - diffEq[t - h, lis[[2]]]), 32];
    lis = Join[{w}, lis], w = N[w + h * diffEq[t, w], 32];
    lis = Join[{w}, lis]];
   t = t + h];
  Return[N[w, 6]]
 )
```

In[ ]:=
```
(*This cell makes lists with the secondBash function with different step sizes. The
 list will be in a form that we can easily plot, using ListLinePlot[].*)
secondBashSizePoint05 = Table[{t, secondBash[yPrime, 0, 1, .05, t]}, {t, 0, 4, 0.05}];
secondBashSizePoint025 = Table[{t, secondBash[yPrime, 0, 1, .025, t]}, {t, 0, 4, 0.025}];
secondBashSizePoint01 = Table[{t, secondBash[yPrime, 0, 1, .01, t]}, {t, 0, 4, 0.01}];
secondBashSizePoint001 = Table[{t, secondBash[yPrime, 0, 1, .001, t]}, {t, 0, 4, 0.001}];
```

In[ ]:=
```
(*This cell plots the exact solution and also plots
 our secondBash estimate with the different step sizes.*)
ListLinePlot[{secondBashSizePoint05, secondBashSizePoint025, secondBashSizePoint01,
  secondBashSizePoint001, exactValues}, PlotRange → {{0, 4}, {0, 5 * 10^6}},
 PlotLegends → {"h = 0.05", "h = 0.025", "h = 0.01", "h = 0.001", "Exact Solution"},
 ImageSize → 1000, AxesLabel → {"t", "y"}]
```

Out[ ]=

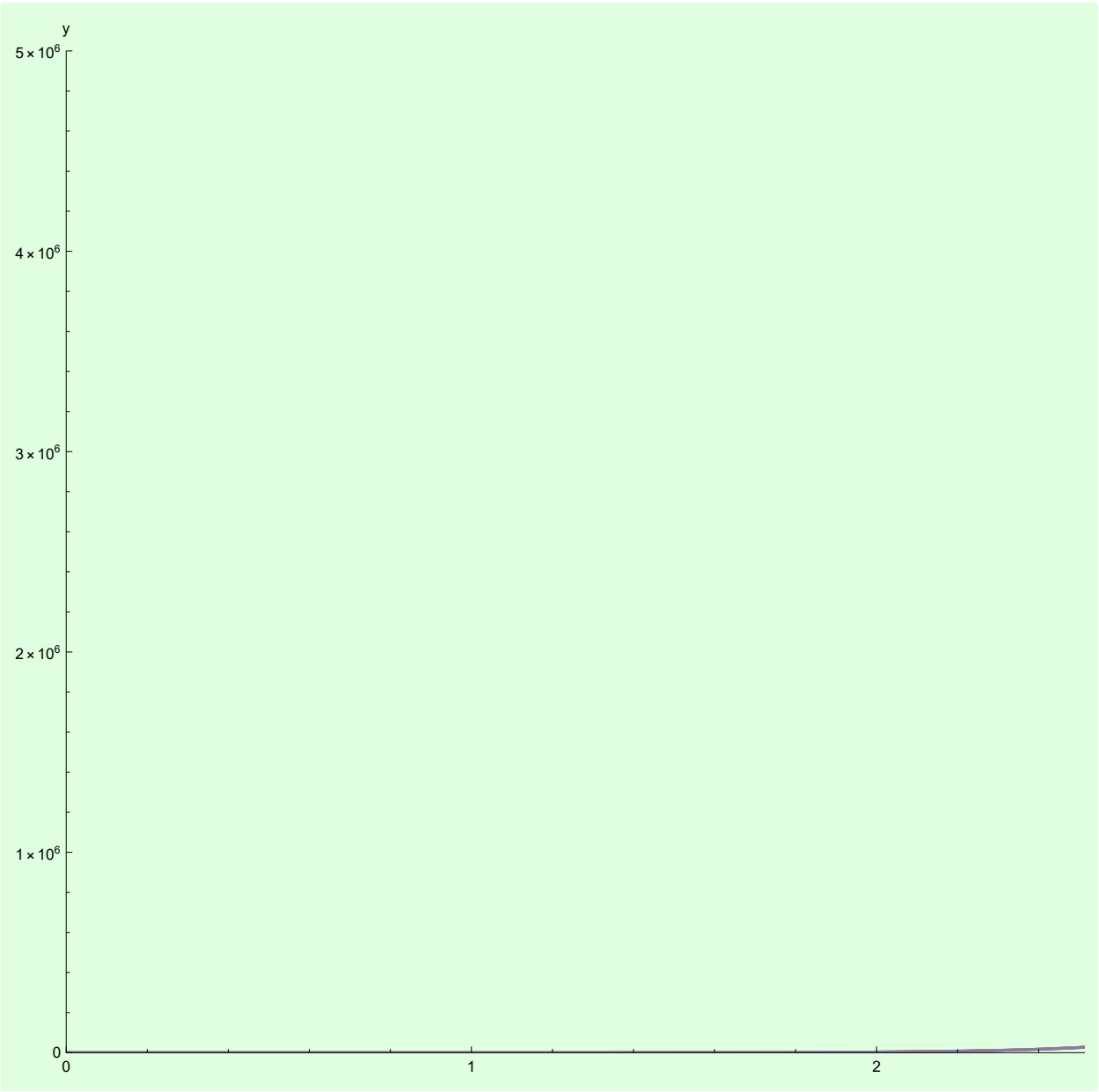


From zooming into this plot, I observed that the second order Adam's Bashforth for the given differential equation is least accurate for h = 0.05. And it gets more accurate as h decreases. Out of the h values I tried, h = 0.001 seems to be the most accurate.

In[ ]:=
```
(*This cell creates a function for the fourth order Adam's Bashforth
 method. For the first three steps, it uses the Euler's method.*)
fourthBash[diffEq_, initialT_, initialY_, h_, finalT_] := (
  t = initialT;
  w = initialY;
  lis = {w};
  totalSteps = (finalT - initialT) / h; For[i = 0, i < totalSteps, i++,
   If[i > 2, w = N[w + h / 24 * (55 * diffEq[t, w] - 59 * diffEq[t - h, lis[[2]]] +
          37 * diffEq[t - 2 * h, lis[[3]]] - 9 * diffEq[t - 3 * h, lis[[4]]]), 32];
    lis = Join[{w}, lis], w = N[w + h * diffEq[t, w], 32];
    lis = Join[{w}, lis]];
   t = t + h];
  Return[N[w, 6]]
 )
```

In[ ]:=
```
(*This cell makes lists with the fourthBash function with different step sizes. The
 list will be in a form that we can easily plot, using ListLinePlot[].*)
fourthBashSizePoint05 = Table[{t, fourthBash[yPrime, 0, 1, .05, t]}, {t, 0, 4, 0.05}];
fourthBashSizePoint025 = Table[{t, fourthBash[yPrime, 0, 1, .025, t]}, {t, 0, 4, 0.025}];
fourthBashSizePoint01 = Table[{t, fourthBash[yPrime, 0, 1, .01, t]}, {t, 0, 4, 0.01}];
fourthBashSizePoint001 = Table[{t, fourthBash[yPrime, 0, 1, .001, t]}, {t, 0, 4, 0.001}];
```

In[ ]:=
```
(*This cell plots the exact solution and also plots
 our fourthBash estimate with the different step sizes.*)
ListLinePlot[{fourthBashSizePoint05, fourthBashSizePoint025, fourthBashSizePoint01,
  fourthBashSizePoint001, exactValues}, PlotRange → {{0, 4}, {0, 5 * 10^6}},
 PlotLegends → {"h = 0.05", "h = 0.025", "h = 0.01", "h = 0.001", "Exact Solution"},
 ImageSize → 1000, AxesLabel → {"t", "y"}]
```



Similar to the second order Adam's Bashforth, the fourth order seems to be least accurate for h = 0.05 and gets progressively more accurate as h increases to 0.001. Compared to the second order Adam's Bashforth, this function seems to be more accurate most of the time. For the first few steps, the fourth order version is probably less accurate than the second order Adam's Bashforth, because the fourth ordered version uses the Euler's method a bit more than the second order Adam's Bash-

forth does.

In[ ]:=
```
(*This cell will create a grid that displays the max
 error for each of the methods at the different h values.*)
Grid[Join[{{"Step size (h)", "Max Error Runge-Kutta",
    "Max Error Second Order AB", "Max Error fourth Order AB"}},
  Table[{h, Max[Table[N[(Abs[y[t] - rungeKutta[yPrime, 0, 1, h, t]]), 6], {t, 0, 4, h}]],
    Max[Table[N[(Abs[y[t] - secondBash[yPrime, 0, 1, h, t]]), 6], {t, 0, 4, h}]],
    Max[Table[N[(Abs[y[t] - fourthBash[yPrime, 0, 1, h, t]]), 6], {t, 0, 4, h}]]},
   {h, {0.05, 0.025, 0.01, 0.001}}]], Frame → All]
```

Out[ ]=

Step size (h)	Max Error Runge-Kutta	Max Error Second Order AB	Max Error fourth Order AB
0.05	1906.15	2.30315×10^6	612 614.
0.025	129.459	681 957.	152 787.
0.01	3.48381	117 194.	24 809.4
0.001	0.000358988	1206.87	252.621

Every single one of these max errors occurred at t = 4, which makes sense since that is the furthest t value from the initial t value we feed into our functions. The fourth order Adams Bashforth has a lower max error than the second order Adams Bashforth's max error at all of the h values, that were tested. Of the three methods, Runge-Kutta has the smallest max error at all h values except for h = 0.05.