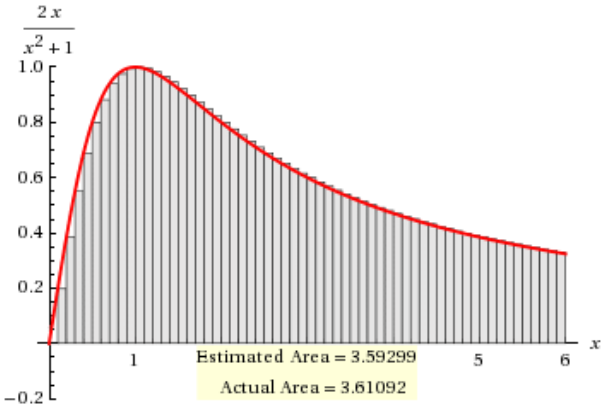
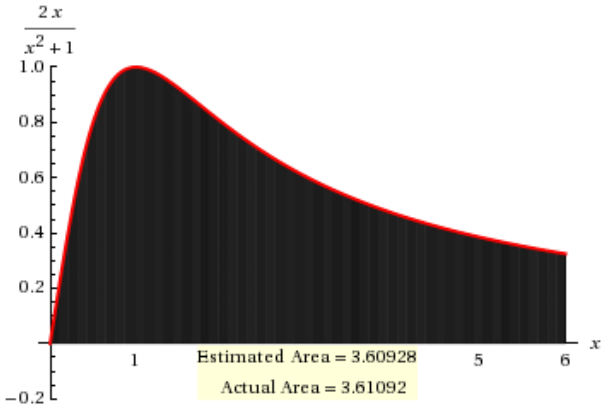
MAD 4401 Numerical Analysis: Project 2
Leeson Chen (UFID: 7907-2212)
December 2 2018

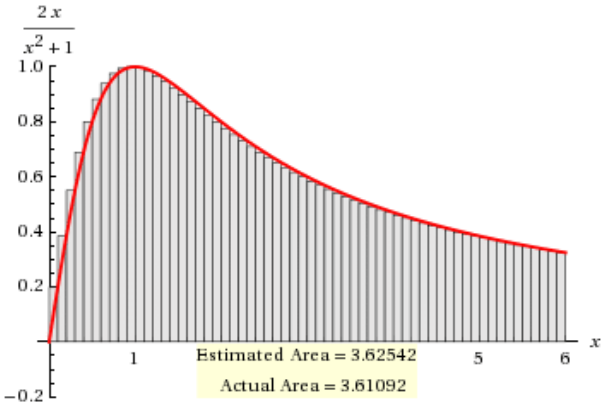
[Numerical Integration] The goal is to find the value of $f(x) = 2x / (1+x^2)$ integrated from 0 to 6. We analyze h = .1, .01, .001, and so on until a satisfactory answer is reached. It is important to note that the true value of this integrated function is Area A = ln(37) = 3.61092… (approx.).

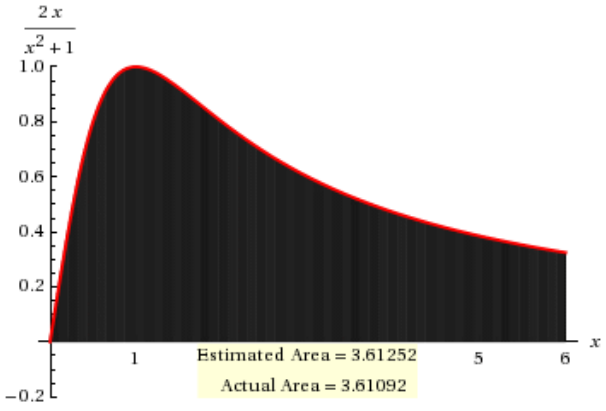
1. **[Riemann Sums]** For a Left Riemann sum with intervals of h = .1 (60 rectangles), we calculate the area to be A = 3.59299, with the graph in the following left figure. At h = .01 (600 rectangles), the area is calculated to be A = 3.60928, which is very close to the actual value.



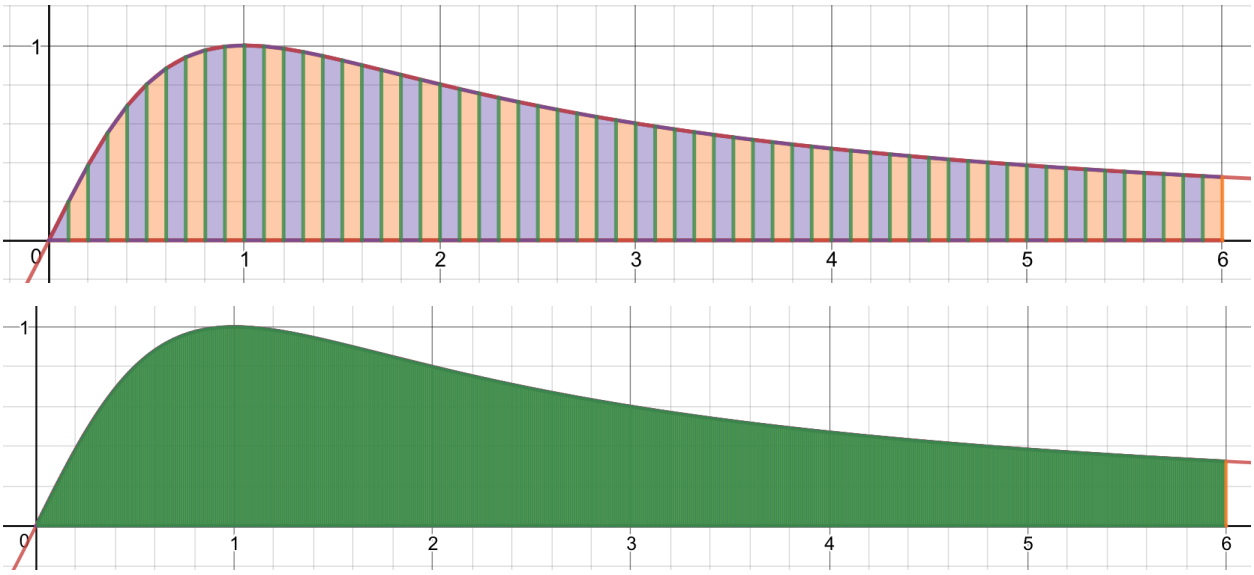


With a Right Riemann sum, intervals of h = .1 (60 rectangles) gives an area A = 3.62542 (left figure below). At h = .01 (600 rectangles), we calculate a more accurate area of A = 3.61252 (right figure below).

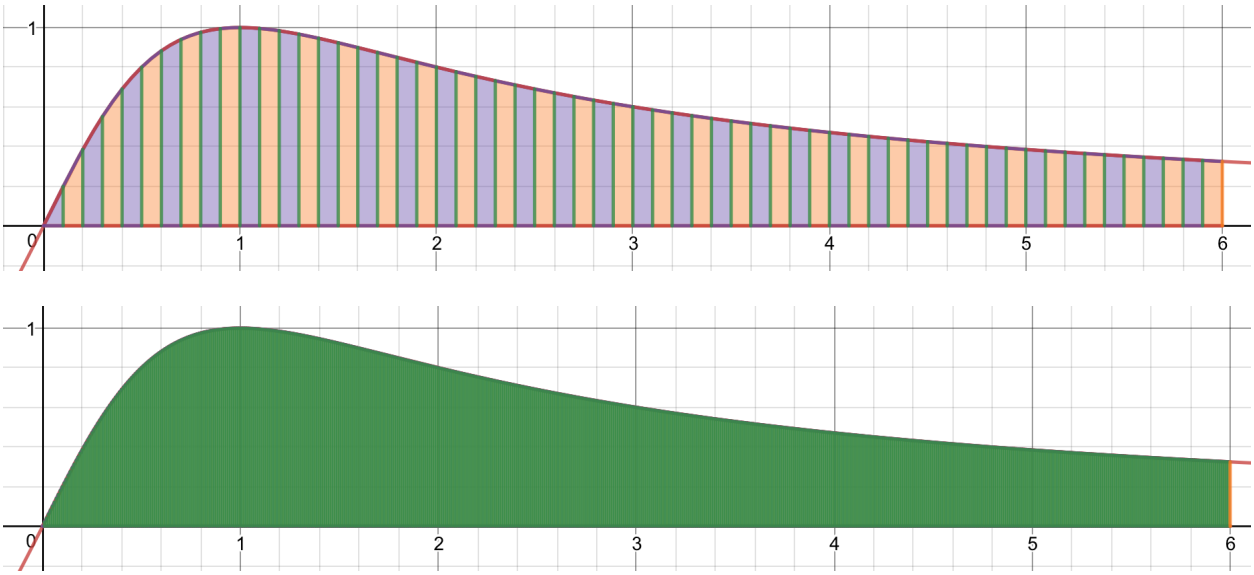




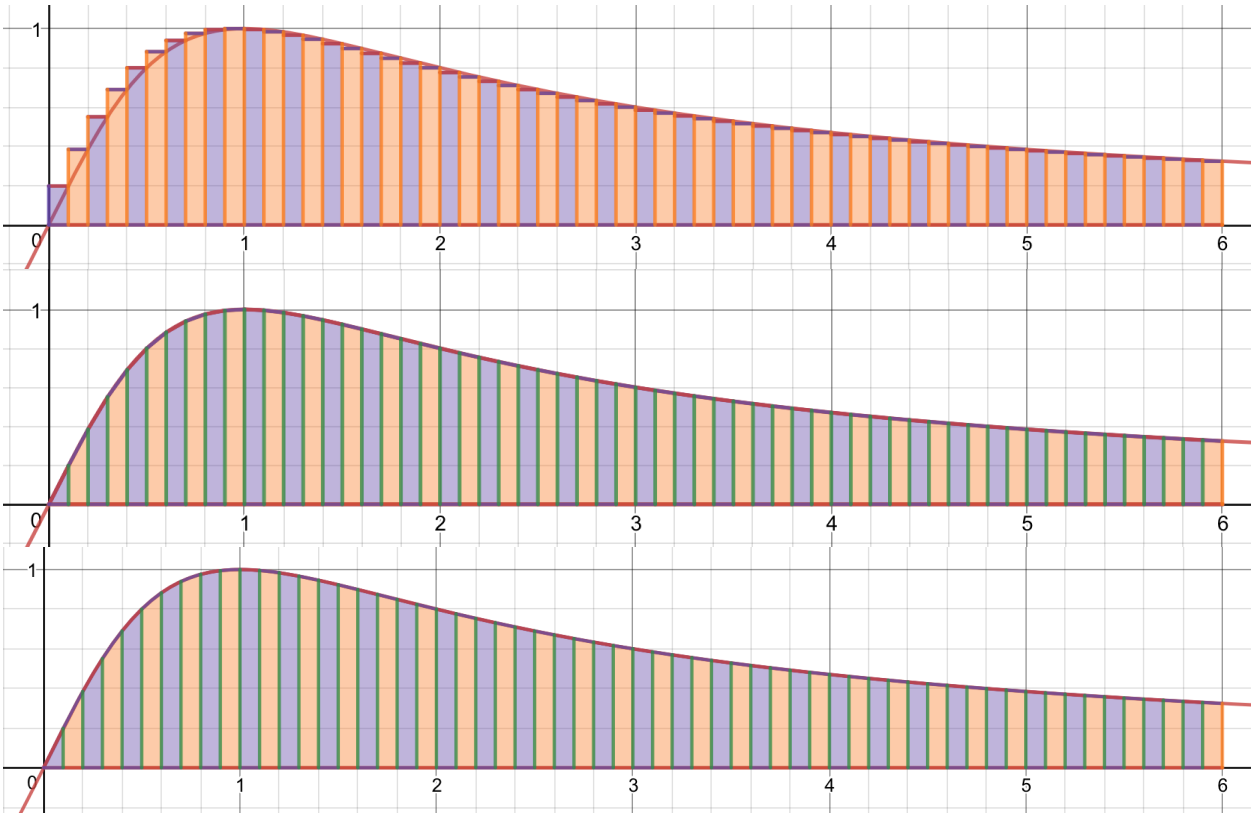
2. **[Trapezoid Rule]** The Trapezoid Rule, when evaluated at h = .1 (60 trapezoids), we calculate the area to be A = 3.60921 (figure below, top). At h = .01 (600 trapezoids), the calculated area becomes A = 3.61090 (figure below, bottom. The individual trapezoids are too small to see). This is close enough to the true A value.



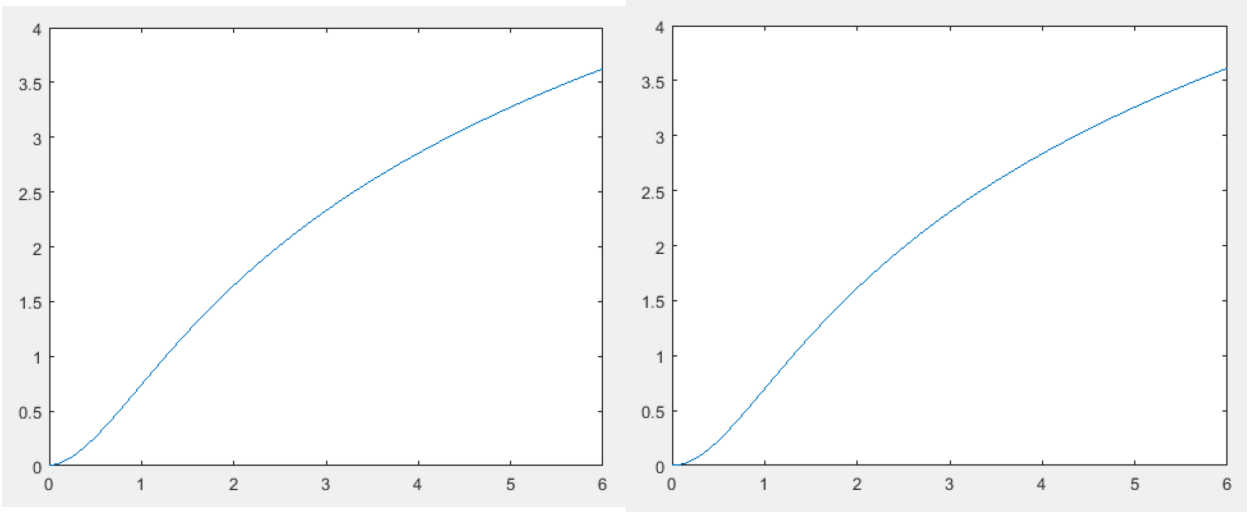
3. **[Simpson's Rule]** Simpson's Rule, when evaluated at h = .1 (60 intervals), gives a calculation of A = 3.61092 (figure below, top). This is very accurate to the true value of A. At h = .01 (600 intervals), we get a calculation of A = 3.61092, the same value to 5 significant digits (figure below, bottom). Both of these estimations are the same as the true value of A at 5 significant digits. Therefore, we have demonstrated that Simpson's rule truly is better at estimating a function than both the Riemann Sum and the Trapezoid Rule.



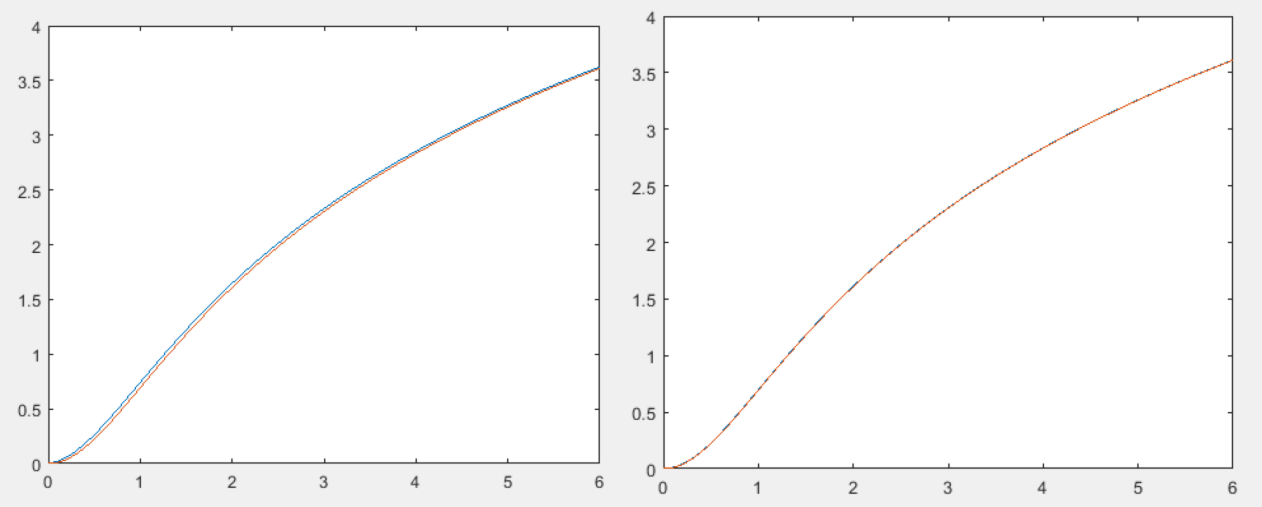
4. **[Comparison]** As we can see from the calculations, Simpson's Rule is the fastest way to approximate the area under a curve, because it gets the exact calculation correct to at least 5 significant digits after only one iteration of h = .1 (60 intervals). The second fastest method is with the Trapezoid Rule, and in last place the Riemann Sums. Below are the three graphs, for visual inspection of their comparison, in the order of Riemann, Trapezoid, Simpson, at h = .1.



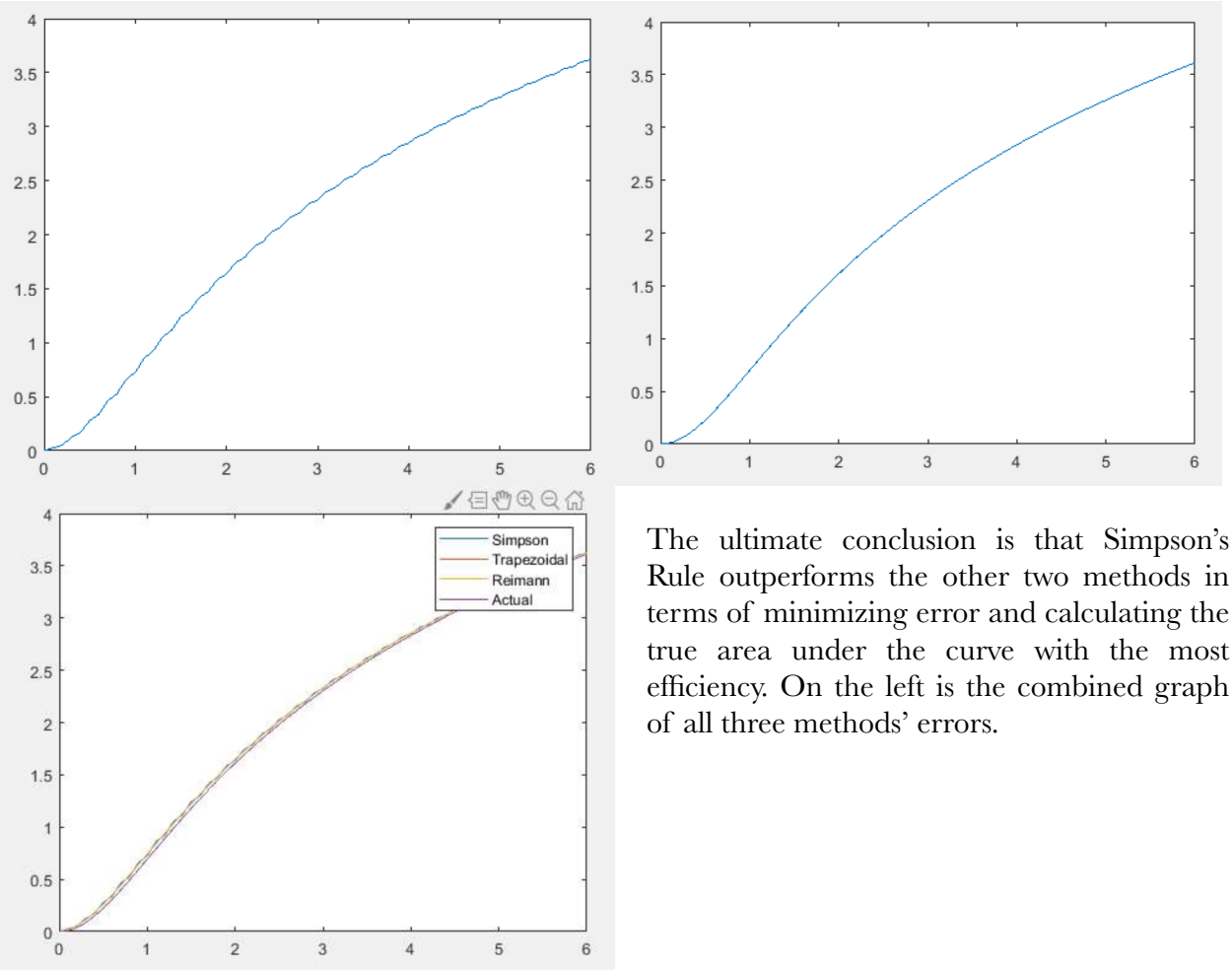
Following, here are graphs of the errors of each method. First is the graph of the error for Riemann Sums, at h = .1 on the left and then .01 on the right.



Next are graphs of the Trapezoid Rule's error, at the same intervals of h = .1 on the left and then .01 on the right.



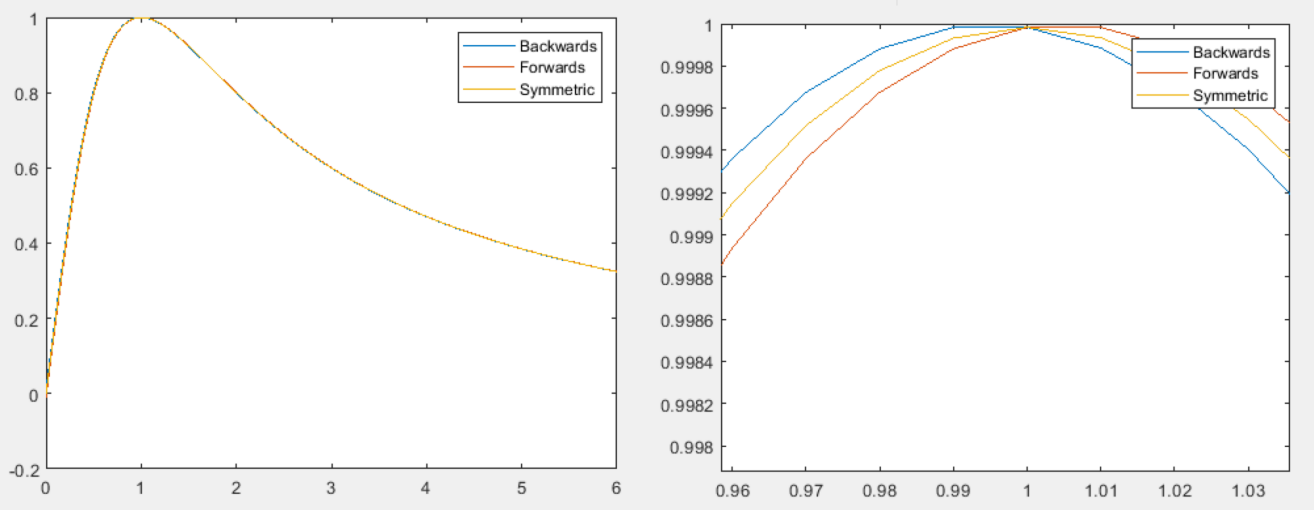
Finally, two graphs of the error of Simpson's Rule, again with h = .1 on the left and then .01 on the right.

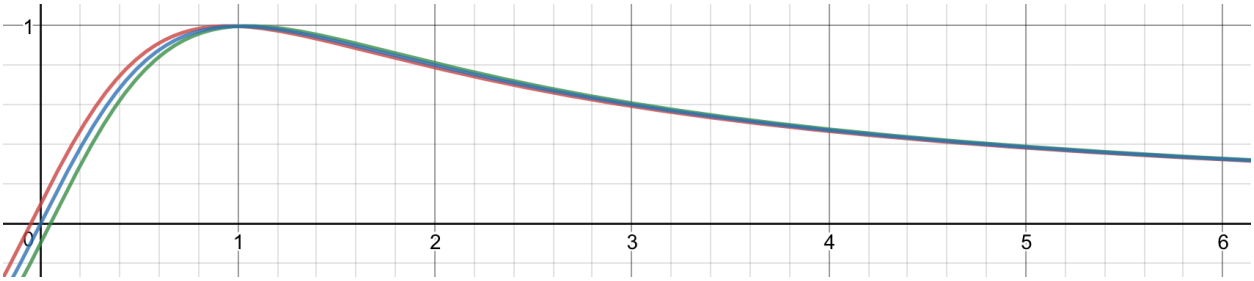


The ultimate conclusion is that Simpson's Rule outperforms the other two methods in terms of minimizing error and calculating the true area under the curve with the most efficiency. On the left is the combined graph of all three methods' errors.

[Numerical Differentiation] Differentiate the function $f(x) = \ln(1+x^2)$ on the interval [0, 6] for h = .1, .01. The true value of this integrated function is A = 12.47680…

5. **[Backwards, Forwards, and Symmetric]** As demonstrated by the following graphs, the equations for backwards, forwards, and symmetric difference perform as expected. Visible in the zoomed-in look at the graph below, the backwards difference lags slightly behind, the forwards difference is slightly forward, and symmetric difference is exactly in the middle of both lines. Because this graph is a visual example of the equations' errors, we have to use a different calculator to get the actual calculation of the function.





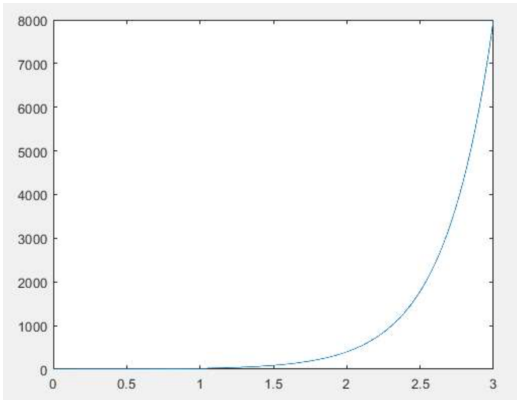
The above Desmos graph more clearly illustrates the curve, with backwards, symmetric, and forwards difference being the first, second, and third lines respectively. The area under those curves is 3.62373, 3.60751, and 3.59129 respectively. The symmetric difference is still the midpoint value of the backwards and forwards values. However, due to the wording of the question we need not make a comparison between the area under these difference formulas, which approximate f'(x), and the area under f(x) (which is roughly 12, very different from the area under the difference curves).

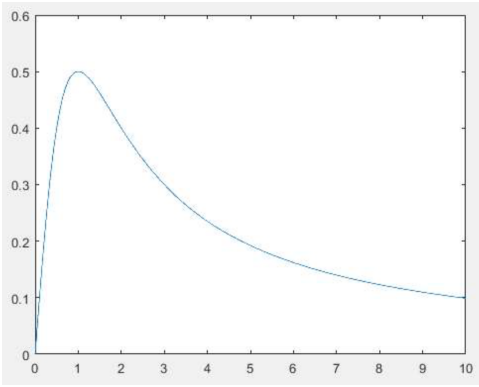
[Numerical ODE Solving Routines] For the following initial value ODE problems:

1) $y' = 3y$, $y(0) = 1$ on $[0, 3]$
2) $y' = 1 / (1+x^2) - 2y^2$, $y(0) = 0$ with solution $y(x) = x / (1+x^2)$ on $[0, 10]$

with $h = .1, .01, .001$, utilize the following methods and measure accuracy with maximum error.

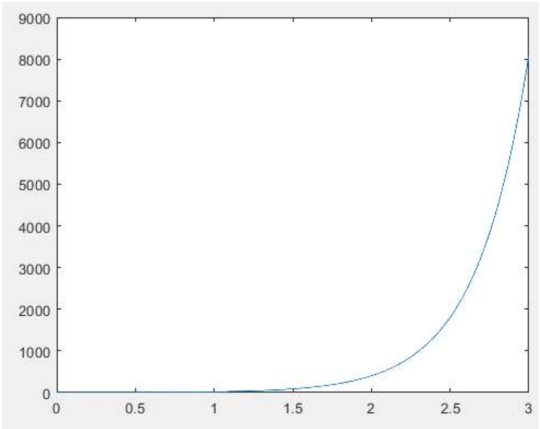
6. **[Euler's Method]** For the first equation, plotting the results in MatLAB results in the following graph of the errors. The command window also displays the value of the maximum error, which during each iteration approaches the value of 8, starting at 0 until quitting at 7.9946. This value is the resultant vector, which is plotted in the graph to the right. We can see that the error increases drastically at an exponential rate.

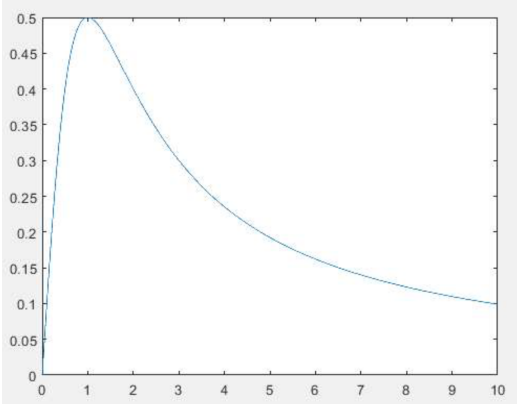




To the left is Euler's method again, plotted for the second equation. In this instance the command window outputs a vector which, during each iteration, begins at 0, and then rapidly increases then decreases until asymptoting to .0990.

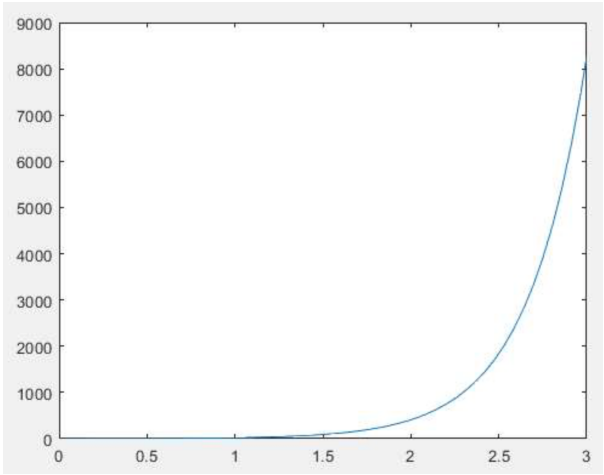
7. **[Midpoint Method]** For the first equation, plotting the results in MatLAB results in the following graph of the errors. The command window also displays the value of the maximum error, which during each iteration approaches the value of 9, starting at 0 until quitting at 8.1030. This value is the resultant vector, which is plotted in the graph to the right. We can see that the error increases drastically at an exponential rate.

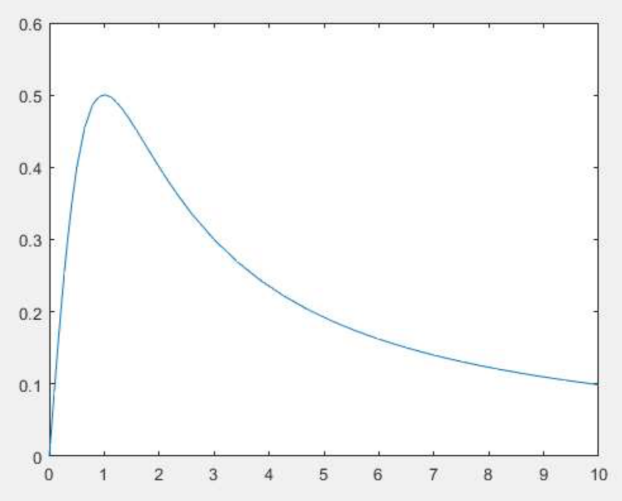




To the left is the Midpoint method again, plotted for the second equation. In this instance the command window outputs a vector which, during each iteration, begins at 0, and then rapidly increases then decreases until asymptoting to .0990, just the same as Euler's method.

8. **[Trapezoid Method]** For the first equation, plotting the results in MatLAB results in the following graph of the errors. The command window also displays the value of the maximum error, which during each iteration approaches the value of 8, starting at 0 until quitting at 8.2672. This value is the resultant vector, which is plotted in the graph to the right. We can see that the error increases drastically at an exponential rate.





To the left is the Trapezoid method again, plotted for the second equation. In this instance the command window outputs a vector which, during each iteration, begins at 0, and then rapidly increases then decreases until asymptoting to .0990, just as the previous two methods.

9. **[Comparison]** As we can see from the command window in the first problem, each "step" of the Trapezoid method increases the current calculation by about .025. While not much, this is actually much faster than the previous two methods, all of which would only increase the current calculation by about .005 for Midpoint and an even lower .0001 for Euler's. This means that of the three methods, Trapezoid is clearly the fastest method of reducing error, followed by the Midpoint method and then finally Euler's method as the slowest.

However, an important distinction is that the Trapezoid method has potential error greater than the Midpoint method. This means that overall, the Midpoint method may be the best method to use in most cases, as it has the greatest accuracy over the other two.

[Code] Here are the source files for all the code I wrote to create this report. Most of the graphs were generated using the online graphing tool desmos.com.

[MatLAB Code: Riemann Sums]

```
function riemann(h)
x = zeros((6/h), 1);
y = zeros((6/h), 1);
f = @(x) 2.*x/(1+x.^2);
i = 0;
xeq = 1;
yeq = 1;
value = 0;
while(i <= 6)
    value = value + h*f(i);
    x(xeq) = i;
    y(yeq) = value;
    i = i+h;
    xeq = xeq + 1;
    yeq = yeq + 1;
end
plot(x,y);
```

[MatLab Code: Trapezoid Rule]

```
function trapezoidRule(h)
x = zeros((6/h), 1);
y = zeros((6/h), 1);
f = @(x) 2.*x/(1+x.^2);
i = 0;
xeq = 1;
yeq = 1;
value = 0;
while(i <= 6)
    if i == 0
       value = value + (1/2)*h*f(i);
    elseif i == 6
       value = value + (1/2)*h*f(i);
    else
       value = value + h*f(i);
    end
    x(xeq) = i;
    y(yeq) = value;
    i = i+h;
    xeq = xeq + 1;
    yeq = yeq + 1;
end
p = [0:.001:6];
q = log(1+p.^2);
plot(x,y,p,q);
```

[MatLab Code: Simpson's Rule]

```
function simpsonsRule(h)
x = zeros((6/h), 1);
y = zeros((6/h), 1);
f = @(x) 2.*x/(1+x.^2);
i = 0;
xeq = 1;
yeq = 1;
value = 0;
count = 0;
while(i <= 6)

    if i == 0
        value = value + h*f(i);
    elseif i == 6
        value = value + h*f(i);
    elseif count == 1
        value = value + 4*h*f(i);
    elseif count == 2
        count = 0;
        value = value + 2*h*f(i);
    end
    x(xeq) = i;
    y(yeq) = value;
    i = i+h;
    xeq = xeq + 1;
    yeq = yeq + 1;
    count = count +1;
end
y = 0.33333 * y;
plot(x,y);
```

[MatLab Code: Backwards, Forwards, & Symmetric Difference]

```
function bfsDifference()

f = @(x) log(1+x.^2);
x = zeros((6/.01), 1);
y = zeros((6/.01), 1);
xeq = 1;
yeq = 1;
%backwards
i = 0;
while i <= 6
   ii = i +.01;
   x(xeq) = i;
   y(yeq) = (f(ii)-f(i))/.01;
   i = i + .01;
   xeq = xeq + 1;
   yeq = yeq + 1;
end

%forwards
xx = zeros((6/.01), 1);
yy = zeros((6/.01), 1);
xeq = 1;
yeq = 1;
i = 0;
while i <= 6
   ii = i -.01;
   xx(xeq) = i;
   yy(yeq) = (f(i)-f(ii))/.01;
   i = i + .01;
   xeq = xeq + 1;
   yeq = yeq + 1;
end

%symmetric
xxx = xx;
yyy = (yy+y)/2;
plot(x,y,xx,yy,xxx,yyy);
legend('Backwards','Forwards','Symmetric');
```

[MatLab Code: Euler's Method]

```
function eulers1(F,t0,h,tfinal,y0)

y = y0;
yout = y;
for t = t0 : h : tfinal-h
    s = F(t,y);
    y = y + h*s;
    yout = [yout; y];
end
x = [t0:h:tfinal];
plot(x,yout);
f = @(t,y) exp(3*t);
```

[MatLab Code: Midpoint Method]

```
function midpointMethod(F,t0,h,tfinal,y0)
y = y0;
yout = y;
for t = t0 : h : tfinal-h
    s1 = F(t,y);
    s2 = F(t+h/2, y+h*s1/s2);
    y = y + h*s2;
    yout = [yout; y];
end
disp(yout)
x = [t0:h:tfinal];
plot(x,yout);
```

[MatLab Code: Trapezoid Method]

```
function trapezoidMethod(F,t0,h,tfinal,y0)
tspan = [t0 tfinal];
[t,y] = ode23t(F, tspan, y0)
plot(t,y)
```