COSC2500 Assignment 1 2020

Tristan Batchler
SN: 44055189
24 Aug 2020

R1.1

Question

Provide a simple approximation to the below xy data.

`r1.1.m`

```
x = [0.0000 0.1341 0.2693 0.4034 0.5386 0.6727 ...
     0.8079 0.9421 1.0767 1.2114 1.3460 1.4801 ...
     1.6153 1.7495 1.8847 2.0199 2.1540 2.2886 ...
     2.4233 2.5579 2.6921 2.8273 2.9614 3.0966 ...
     3.2307 3.3659 3.5000];
y = [0.0000 0.0310 0.1588 0.3767 0.6452 0.8780 ...
     0.9719 1.0000 0.9918 0.9329 0.8198 0.7707 ...
     0.8024 0.7674 0.6876 0.5937 0.5778 0.4755 ...
     0.3990 0.3733 0.2870 0.2156 0.2239 0.1314 ...
     0.1180 0.0707 0.0259];
```

Answer

The first 9 points look to be a higher-order polynomial, but we will represent it as three piecewise cubics. The remaining points appear to be a straight line.

We are using piecewise cubics because the spline function in Matlab produces these when fitting points. The advantage of using splines is that one can control the slope of either endpoint which will allow a continuous curve into the straight line.

`r1.1.m`

```
% Split the xy data into two parts
splitpoint = 9;

x2 = x(splitpoint: numel(x));
y2 = y(splitpoint: numel(y));

% x2y2 can be approximated with a line
l = polyfit(x2, y2, 1);
x2fit = [x2(1), x2(numel(x2))];
y2fit = polyval(l, x2fit);

% Save the left endpoint of the line as well as the slope so that it
```

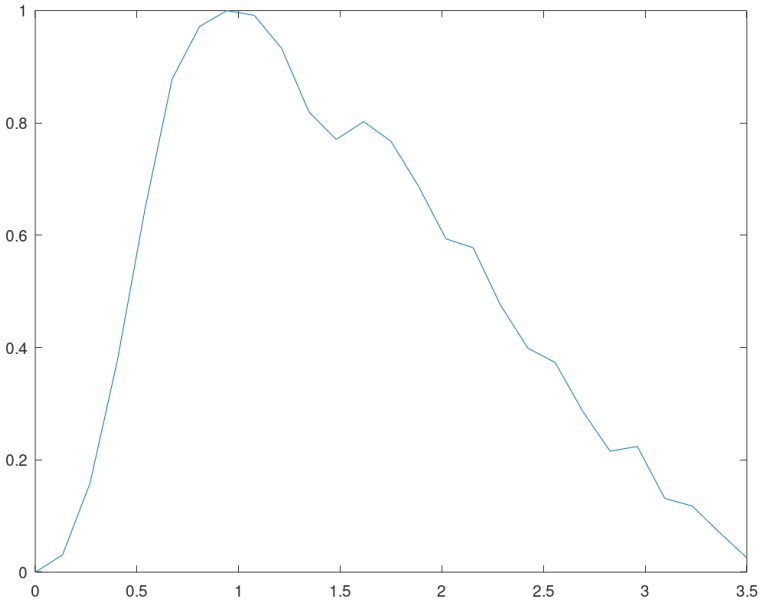


Figure 1: R1.1 original plot

```
% can connect to the cubic spline coming into it
slope = l(1);
connectx = x2fit(1);
connecty = y2fit(1);

% x1y1 can be approximated with three cublic splines
% That way we can control the last endpoint to be the slope into the
% x2y2 linear approximation for continuitity
x1 = [x(1) x(3) x(6) connectx];
y1 = [y(1) y(3) y(6) connecty];

cs = spline(x1, [0, y1, slope]);
x1fit = linspace(x1(1), x1(numel(x1)), 1000);
y1fit = ppval(cs, x1fit);

% Plot the results
figure;
p_orig = plot(x, y, 'r.');
hold on;
p_spline = plot(x1fit, y1fit, 'g');
hold on;
p_line = plot(x2fit, y2fit, 'b');
legend([p_orig, p_spline, p_line], {
    "original data samples",
    [num2str(cs.pieces) " cubic splines"],
    [num2str(slope) "x + " num2str(l(2))]
});
```

The aim was to keep the number of cubic splines low as to keep the approximation simple, however, using just one or two did not provide a good enough approximation. This is why three was used.

It is clear that the straight line is a good fit for all of the points past the 9th.

Because the slope of the endpoint of the last spline was able to match the slope of the straight line, we have a continuous approximation of the sample data.

R1.2

Question

- For the polynomial $f(x) = x^4 - 2x^3$, calculate the derivative and the 2nd derivative analytically.
- Choose different step sizes h and plot a log-log graph of error vs step size to compare the numerical derivatives found using the forward, backward, and central difference formulas with the actual derivative.
  - How accurate are the numerical derivatives at x = 0, 1, 1.5, and 2?

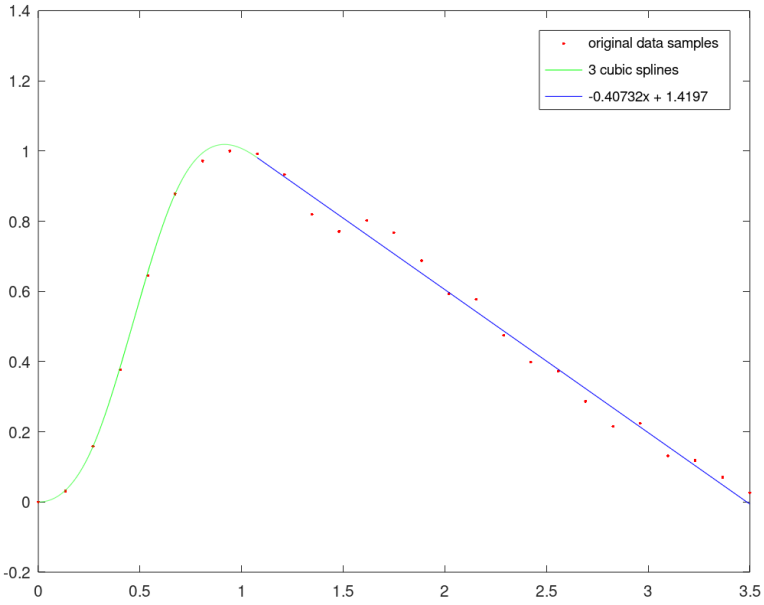


Figure 2: R1.1 final plot

  - At what step sizes do you obtain the minimum errors, and what are the minimum errors?
  - Do you expect that this would be a problem?
  - What if you were using single-precision floating point?
- Add some additional error to your polynomial.
  - How does the error in the numerical derivative vary with step size h?
  - Try changing the size of the error.

Answer

- Analytically, $f'(x) = 4x^3 - 6x^2$ and $f''(x) = 12x^2 - 12x$.
- `squarediff.m`

```
% Function to calculate squared difference sum of two vectors
function sd = squarediff(y1, y2);
  sd = 0;
  nvals = numel(y1);
  if nvals != numel(y2);
    error("y1 and y2 not same size");
  end;
  for n = 1: nvals;
    sd += (y1(n) - y2(n))^2;
  end;
end;
```

`r1.2a.m`

```
x = linspace(-10, 10, 1000);
f = @(x) x.^4 - 2.*x.^3;                                % Our f(x)
dfdx = @(x) 4.*x.^3 - 6.*x.^2;                  % Exact derivative
dfdxfwd = @(x, h) (f(x + h) - f(x)) / h;       % Forwards approx
dfdxbwd = @(x, h) (f(x) - f(x - h)) / h;              % Backwards
dfdxcnt = @(x, h) (f(x + h) - f(x - h)) / (2 * h);     % Central

num_steps = 1000;

% Set up the errors
difffwd = zeros(1, num_steps);
diffbwd = zeros(1, num_steps);
diffcnt = zeros(1, num_steps);

% Fill in the errors
hsteps = logspace(0, -20, num_steps);
for n = 1: num_steps;
  h = hsteps(n);
  y = dfdx(x);
  yfwd = dfdxfwd(x, h);
```

```
  ybwd = dfdxbwd(x, h);
  ycnt = dfdxcnt(x, h);
  difffwd(n) = squarediff(y, yfwd);
  diffbwd(n) = squarediff(y, ybwd);
  diffcnt(n) = squarediff(y, ycnt);
end;

% Plot the errors against step size
figure;
pfwd = loglog(hsteps, difffwd, 'r');
hold on;
pbwd = loglog(hsteps, diffbwd, 'g');
hold on;
pcnt = loglog(hsteps, diffcnt, 'b');
xlabel('h');
legend([pfwd, pbwd, pcnt], {
  "forwards error",
  "backwards error",
  "central error"
});
```

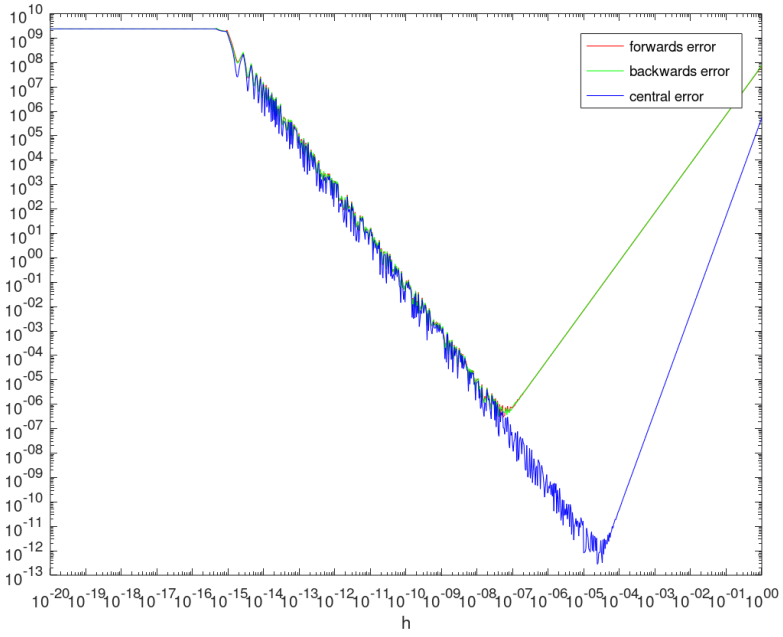


Figure 3: R1.2a plot

– We can repeat the above process for the derivative at the specific points required rather than over the arbitrary range of $[-10, 10]$.

r1.2a-all.m

```
f = @(x) x.^4 - 2.*x.^3;                                % Our f(x)
dfdx = @(x) 4.*x.^3 - 6.*x.^2;                  % Exact derivative
dfdxfwd = @(x, h) (f(x + h) - f(x)) / h;         % Forwards approx
dfdxbwd = @(x, h) (f(x) - f(x - h)) / h;               % Backwards
dfdxcnt = @(x, h) (f(x + h) - f(x - h)) / (2 * h);      % Central

xs = [0, 1, 1.5, 2];
for n = 1: numel(xs);
  thisx = xs(n);
  num_steps = 1000;

  % Set up the errors
  difffwd = zeros(1, num_steps);
  diffbwd = zeros(1, num_steps);
  diffcnt = zeros(1, num_steps);

  % Fill in the errors
  hsteps = logspace(0, -20, num_steps);
  for m = 1: num_steps;
    h = hsteps(m);
    y = dfdx(thisx);
    yfwd = dfdxfwd(thisx, h);
    ybwd = dfdxbwd(thisx, h);
    ycnt = dfdxcnt(thisx, h);
    difffwd(m) = squarediff(y, yfwd);
    diffbwd(m) = squarediff(y, ybwd);
    diffcnt(m) = squarediff(y, ycnt);
  end;

  warning ("off", "Octave:negative-data-log-axis");
  figure;
  pfwd = loglog(hsteps, difffwd, 'r');
  hold on;
  pbwd = loglog(hsteps, diffbwd, 'g');
  hold on;
  pcnt = loglog(hsteps, diffcnt, 'b');
  title(['x = ', num2str(thisx)]);
  xlabel('step size h');
  legend([pfwd, pbwd, pcnt], {
    "forwards error",
    "backwards error",
    "central error"
```

```
  });
end;
```

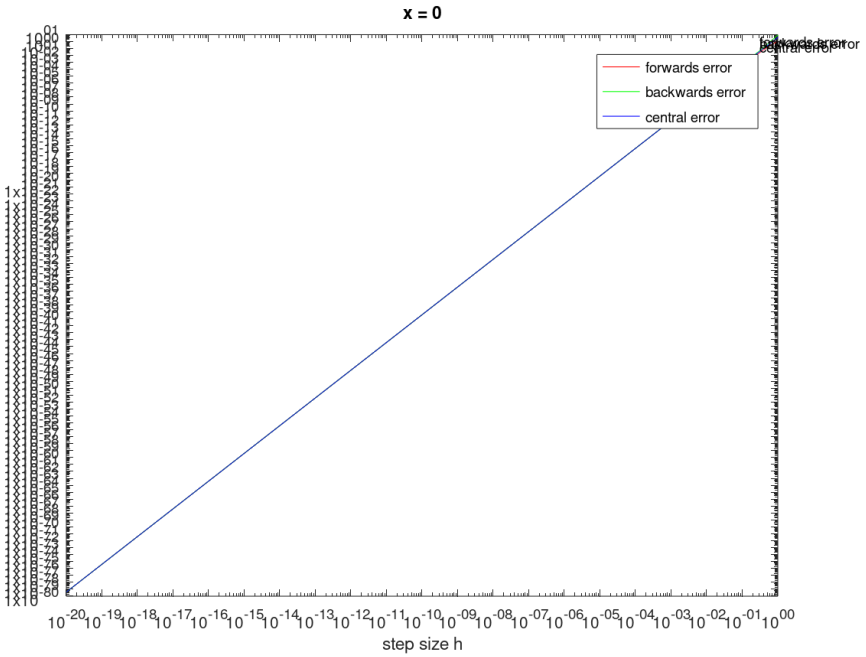


Figure 4: R1.2a x=0 plot

∗ **Discussion**: when $x := 0$, the squared error of the forwards, backwards, and central derivative appear equal except when the step size is greater than about 10^{-1}. It is also worthy to note that the error in all of these derivative approximations grow at the same rate as the step size increases. Because they appear as straight lines in the log-log plot, each error must be roughly of the form $E = cx^m$ for some $c \in \mathbb{R}$ and $m \in \mathbb{N}$.

We can explain the shape of the **central** derivative squared error plot if we consider equation 1.13 from the course notes. It states that the expected error for central differentiation approximations is

$$E_{\text{cnt}}(x) \approx \frac{h^2}{6} f'''(x).$$

Assuming this error is the sum of absolute values of differences, we need to square these expectations because we are using squared error.

Evaluated at $x := 0$, this becomes

$$E^2_{\text{cnt}}(0) \approx \left(\frac{h^2}{6}\left(24(0) - 12\right)\right)^2 = 4h^4.$$

A log-log plot of $y = 4x^4$ shows exactly the shape we see above.

```
>> hsteps = logspace(0, -20, 1000);
>> err = 4 .* hsteps .^ 4;
>> loglog(hsteps, err);
```

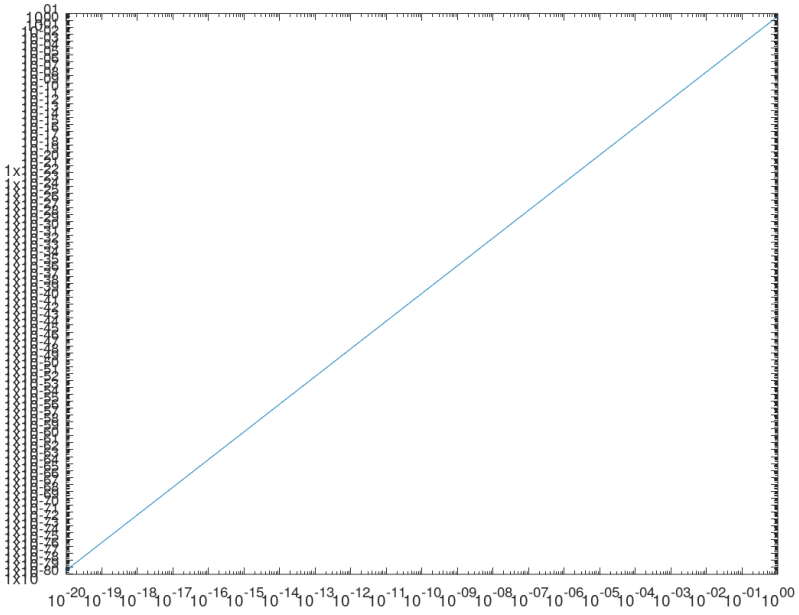


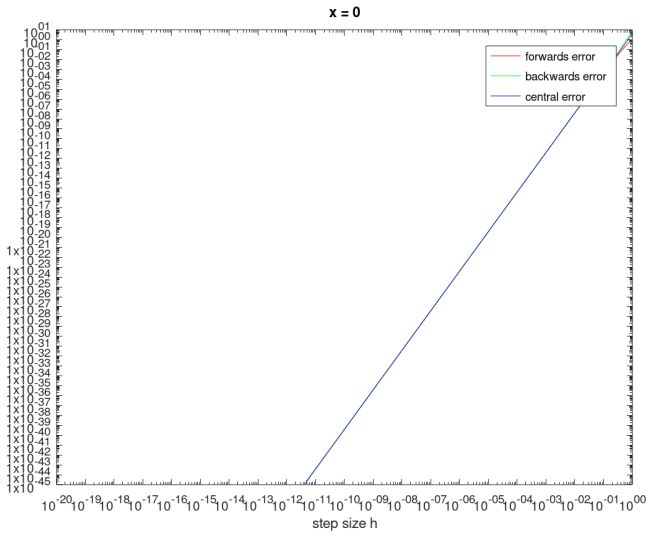
Figure 5: R1.2a x=0 plot central shape confirmation

It is not known why the squared **forwards and backwards** error follows the central error almost exactly considering they should remain at 0 according to equations 1.9 and 1.11 from the course notes.

$$E^2_{\text{fwd}}(0) \approx \left(\frac{h}{2}\left(12(0)^2 - 12(0)\right)\right)^2 = 0$$

$$E^2_{\text{bwd}}(0) \approx \left(-\frac{h}{2}\left(12(0)^2 - 12(0)\right)\right)^2 = 0$$

* **Accuracy**: The accuracy looks extremely good as approximations appear to approach zero error as smaller step sizes are used. It is unknown why the plot isn't "V-shaped" as the others are attributed to very small (even smaller than machine epsilon) values of h being used in calculations. This plot is a straight line which seems to suggest very small values of h aren't making the error larger. I imagine Matlab is doing some optimisations to the approximation code which result in h being cancelled out.
* **Best step size & minimum errors**: The smaller the better in this case although it is very confusing why that is the case. It is impractical to use values of h smaller than ϵ so the fact that the plot is showing smaller errors even for $h < \epsilon$ means to me the plot is not accurate and h isn't being used at all in the calculations.
* **Is this problematic**: This plot in particular seems problematic as it can't be representative of how error changes with step size.
* **With single precision**: Running the same code as above but replacing the definition of `xs` with `xs = [single(0)];`:



It appears the error is the same when calculations are done in single-precision. The reason the curve looks steeper appears to be that Matlab refuses to drawn values less than about 10^{-45} seemingly because it thinks it is just zero. On the other hand, in double-precision mode, Matlab is capable of drawing the curve as low as 10^{-80}. This doesn't mean the errors are lesser in double-precision mode in this instance though.

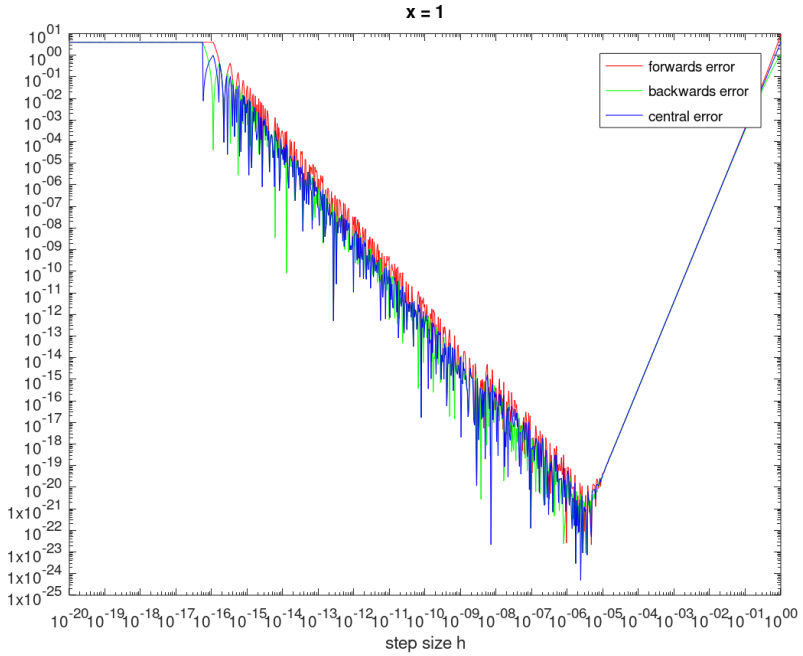


Figure 6: R1.2a x=1 plot

* **Discussion**: when $x := 1$, we notice a "V-shaped" plot for each of forwards, backwards, and central approximation errors. It was expected that the central error would be less than the other two but this is not the case.

  This is further confused by equations 1.9, 1.11, and 1.13 from the course notes.

$$E_{\text{fwd}}(1) \approx \frac{h}{2}\left(12(1)^2 - 12(1)\right) = 0$$

$$E_{\text{bwd}}(1) \approx -\frac{h}{2}\left(12(1)^2 - 12(1)\right) = 0$$

$$E_{\text{cnt}}(1) \approx \frac{h^2}{6}\left(24(1) - 12\right) = 2h^2.$$

  From these questions, we should expect no error in the forwards and backwards approximation, and the central approximation error should be $2h^2$ (meaning $4h^4$ squared error) which is not the same slope we see in the plot.

  Of course the error for small values of h are always expected presumably due to rounding, but the errors for $h \gtrapprox 10^{-6}$ are not what we expect.

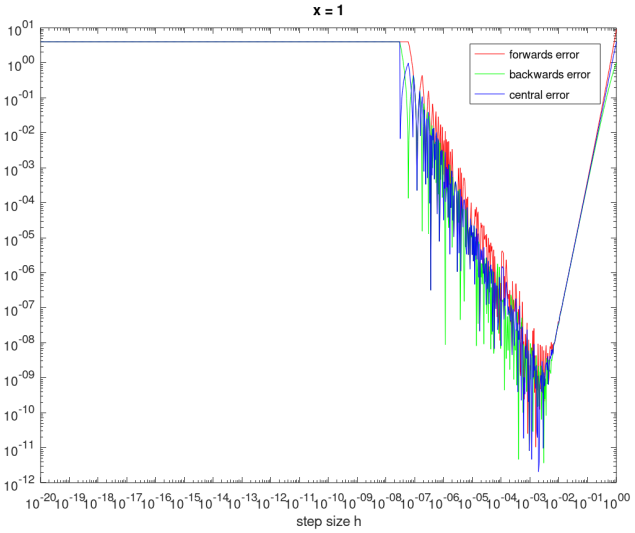
  We can also see the machine epsilon taking effect for step sizes of $h \lessapprox 2 \times 10^{-16}$ as all errors flatline at a value of 4. It is surprising the interpreter didn't just throw a "divide by zero" exception since values of $h < \epsilon$ should be considered 0, and we are always dividing by h in these approximations.

* **Accuracy**: The squared error can get sufficiently close to zero (discussed more below).

* **Best step size & minimum errors**: The minimum error for **all** approximations appears to be achieved for a step size of $h \approx 10^{-6}$. The minimum squared error is around the order of 10^{-22}. Using step sizes smaller than 10^{-7} will result in larger errors presumably due to round-off.

* **Is this problematic**: The observed behaviour is not problematic. It is strange that central approximation does not perform any better than the others in this case but that does not appear to be a problem. Similarly, the errors observed in the plot do not match what we would expect from equations 1.9, 1.11, and 1.13 but again this does not indicate a problematic plot. This could just be an edge case in the expectation models.

* **With single precision**: Running the same code as above but replacing the definition of `xs` with `xs = [single(1)];`:



Things to note:

- The "flat line" at 4 is much longer because single precision can't differentiate between small values and 0 as well.
- The minimum error is only of the order of about 10^{-10} which is much less accurate than the 10^{-22} obtained with double precision.
- The minimum error is attained for a step size of about 10^{-3} which is much larger than the best step size of about 10^{-6} with double precision.

∗ **Discussion**: when $x := 1.5$, we note that, analytically, $f'(x) = 0$.

This must mean

$$\lim_{h \to 0} \frac{f(1.5 + h) - f(h)}{h} = 0$$

Numerically, by looking at the plot, we can see there is very little roundoff error for very small values of h. I believe this is because roundoff error in the numerator of the approximation makes the entire fraction 0 and this just so happens to be the correct derivative, hence no error. Notice this isn't happening all of the time for very small h but certainly most of the time.

We observe the usual behaviour for larger values of h, which is the same behaviour as in the case when $x := 2$.

∗ **Accuracy**: The accuracy is very good, but I believe this is coincidental just due to the fact that the exact derivative is 0.

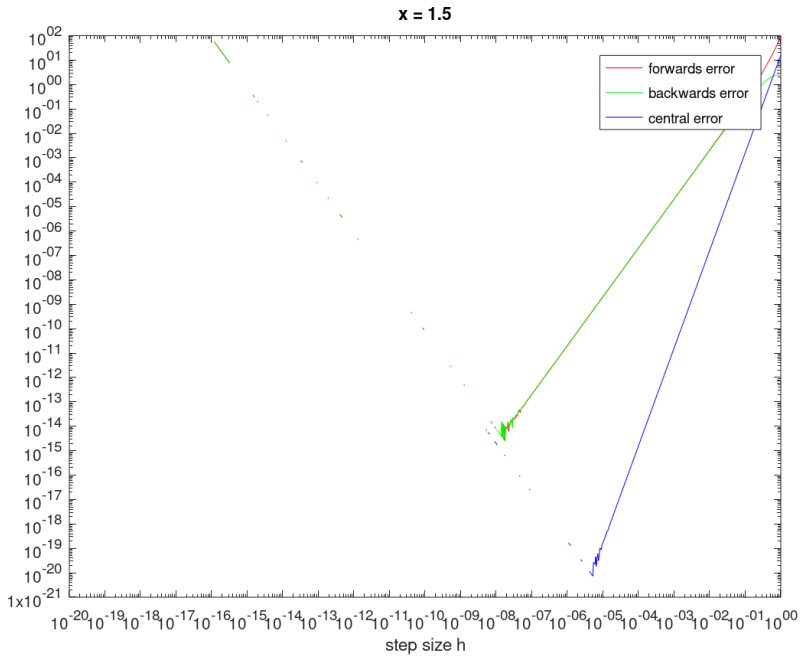
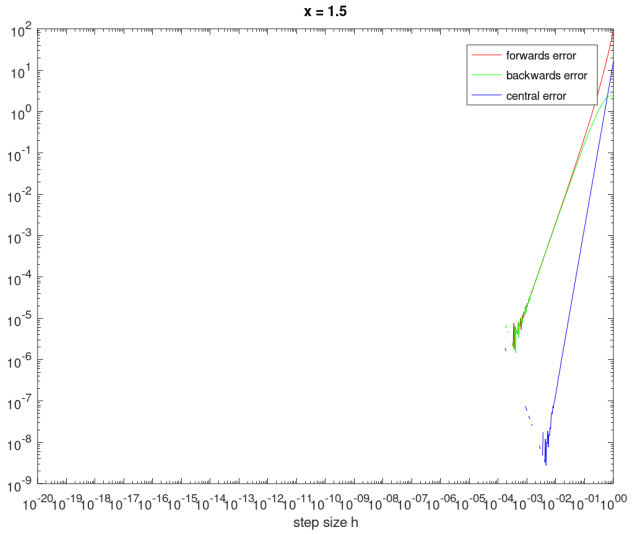


Figure 7: R1.2a x=1.5 plot

* **Best step size & minimum errors**: Very small step sizes attain an error of 0 but this is not reliable as it is abusing roundoff error in the approximation. The ideal step size in this instance could be around 10^{-5} using the central approximation.
* **Is this problematic**: It is problematic that the results are deceiving.
* **With single precision**: Running the same code as above but replacing the definition of `xs` with `xs = [single(1.5)];`:



This is pretty much expected. Note the roundoff error is much more consistent here and isn't non-zero at all for $h \lessapprox 10^{-5}$. This should be because roundoff error is much more common in single precision.

---

* **Discussion**: when $x := 2$, the numerical results as shown in the plot are exactly as expected. This discussion won't point out anything noteworthy if it's already been mentioned above.
* **Accuracy**: Accuracy is sensible. As h gets smaller, so does the error as the approximation is getting more fine-grain. However, after a certain point, more roundoff error is introduced for very small values of h and the error begins to climb again. Once $h < \epsilon$, we see the error maintain at 64.
* **Best step size & minimum errors**: The best step size, as with other plots, appears to be around 10^{-5} using central approximation. This produces a minimum squared error of around 10^{-23}.

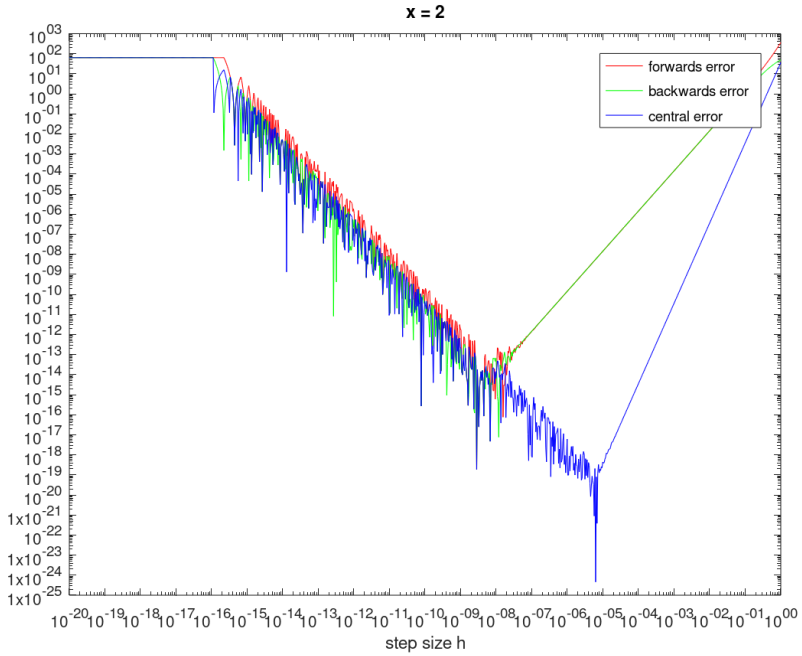
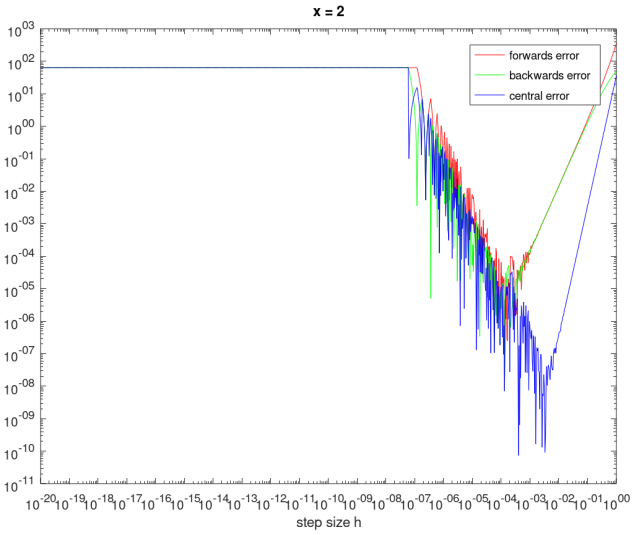


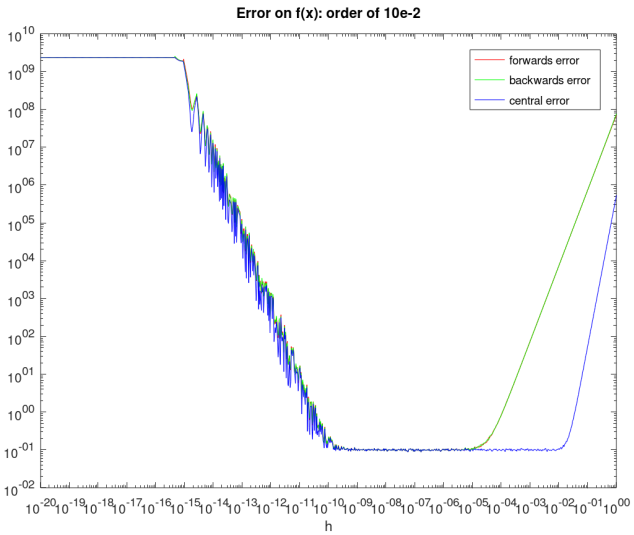
Figure 8: R1.2a x=2 plot

* **Is this problematic**: No.
* **With single precision**: Running the same code as above but replacing the definition of `xs` with `xs = [single(2)];`:

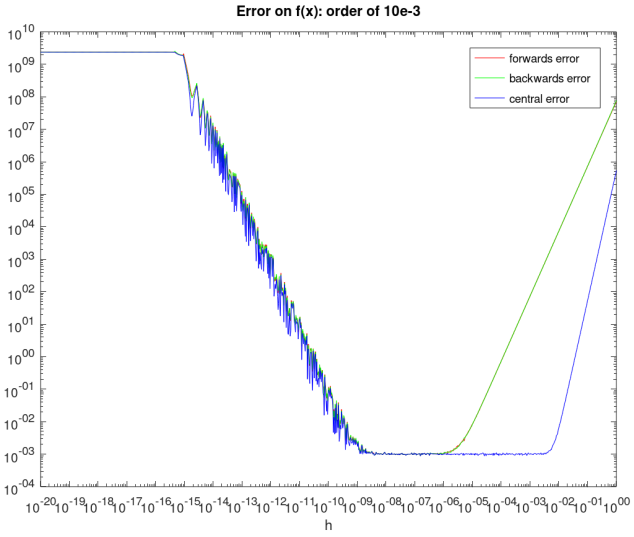


– We introduce additional error to the polynomial y we are comparing the others to. Here are the results.

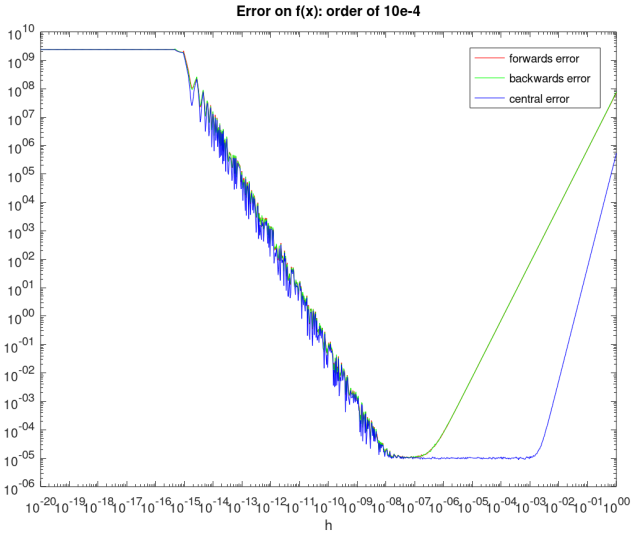
* `y = dfdx(thisx) + 10e-2 * randn(size(thisx));`



* `y = dfdx(thisx) + 10e-3 * randn(size(thisx));`



* `y = dfdx(thisx) + 10e-4 * randn(size(thisx));`



Note that the random noise introduced makes it impossible for the error to get below a certain threshold. So the "V-shape" turns into more of a "U-shape" because it clips the lower end of the error off.

The order of the noise introduced determines how much is clipped off, with smaller noise clipping less.

R1.3

Question

Compare the trapezoid rule and Simpson's rule for integrating $cos(x)$ from $x = 0$ to $x = \pi/2$ and $x = \pi$, and $sin(x^2) + 1$ from $x = 0$ to $x = 10$. * Plot a log-log graph to compare the accuracy of the trapezoid rule and Simpson's rule for different step sizes. * Compare the accuracy of an adaptive step size integrator (e.g., `integral()` or `quad()` in Matlab). * How does additional error affect integration? Try it and see, using the sam e random error term you used before. * How much does round-off error affect integration? * Since a small value of h means that we need many points, we also increase our requirements in time and memory. Do the time and/or memory required become excessive before we run into any problems with round-off error?

Answer

- Below is a Matlab code snippet which takes in a function handle f, start & end values x_0, x_1 to integrate over, the exact integral (known analytically) and a vector containing the numbers of steps to plot for.

  It plots (log-log) the errors of the trapezoidal vs. Simpson approximations of

  $$\int_{x_0}^{x_1} f(x)dx$$

  against the varying step sizes determined from the provided vector of numbers of steps.

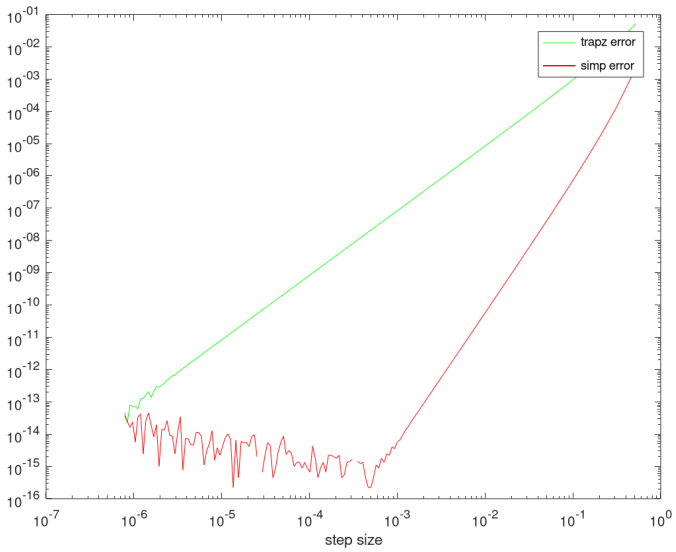
  `plotinterrors.m`

```
function plotinterrors(start, stop, func, exact, nstepsvector);
  N = numel(nstepsvector);
  traperrors = zeros(N);
  simperrors = zeros(N);

  for n = 1: N;
      nsteps = nstepsvector(n);
      x = linspace(start, stop, nsteps);
      y = func(x);
      trap = trapz(x, y);
      simp = SIMP42(y, x(2) - x(1));

      traperrors(n) = abs(exact - trap);
      simperrors(n) = abs(exact - simp);
  end;

  warning ("off", "Octave:negative-data-log-axis");
  stepsizesvector = (stop - start) ./ nstepsvector;
```

```
  figure;
  ptraperrs = loglog(stepsizesvector, traperrors, 'g');
  hold on;
  psimperrs = loglog(stepsizesvector, simperrors, 'r');
  xlabel('step size');
  legend([ptraperrs, psimperrs], {
      "trapz error",
      "simp error"
  });
end;
```

So to compare the items above, we run the following. We provide `2*round(logspace(0, 6, 200))+1);` as the number vector containing numbers of points to get a vector of 200 logarithmically spaced odd integers from 3 to 2,000,001 (taken from note [@33 on Piazza](https://piazza.com/class/kd9obdfovp31wi?cid=33)). The reason for making them odd is for compatibility with `SIMP42`.

- `plotinterrors(0, pi/2, @(x) cos(x), 1, 2*round(logspace(0, 6, 200))+1);`
  $\left(\int_0^{\pi/2} \cos(x)dx = \sin(\pi/2) - \sin(0) = 1 - 0 = 1\right)$



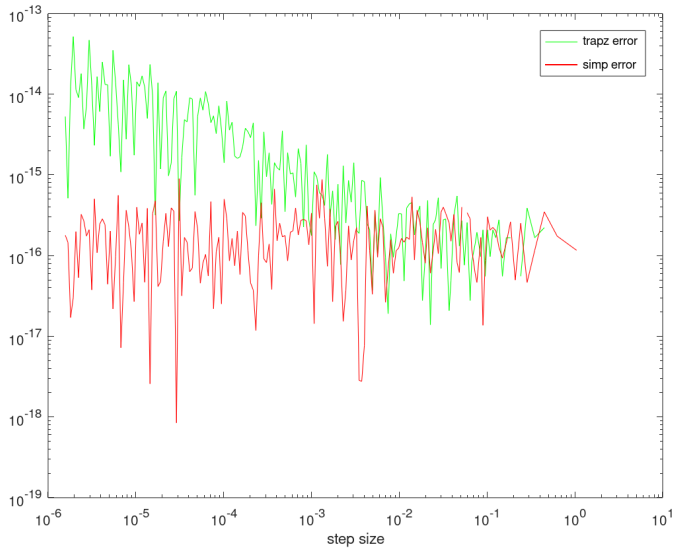
Here it wee `SIMP42` is a better approximation for the larger step sizes, but both approximators converge to the same error size for small enough step size (i.e. error of about 10^{-13} at step size of about 10^{-6}).

`SIMP42` reaches the minimum error much quicker than `trapz`, but oddly the error seems to increase slowly as step size gets smaller. I am unsure if this can be attributed to rounding error.

Also noticible in the plot are some gaps in the `SIMP42` error. This is when the error is zero but can't be drawn on a log-log plot.

I am not sure why the behaviour is so erratic around the minimum error for the approximations. It seems likely this can be attributed to the error being so close to the machine epsilon $\left(\approx 2.2 \times 10^{-16}\right)$. I believe the error cannot be measured once it goes below ϵ which is why it appears to hover around the bottom like that no matter how small the step size gets.

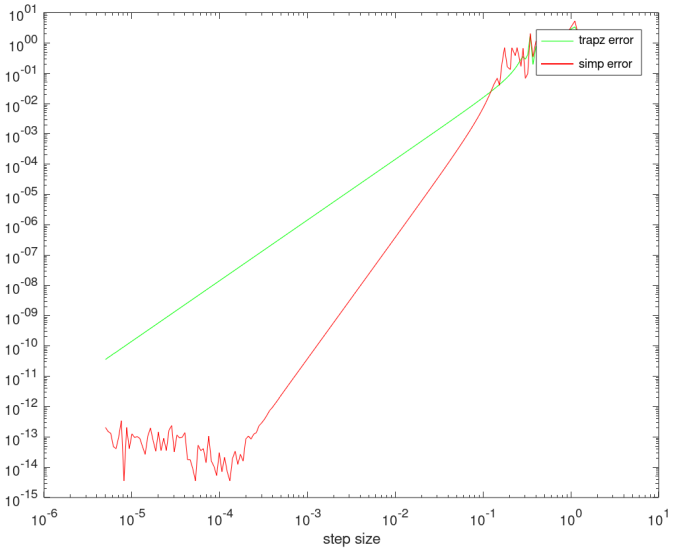
– `plotinterrors(0, pi, @(x) cos(x), 0, 2*round(logspace(0, 6, 200))+1);`
$\left(\int_0^{\pi} \cos(x)dx = \sin(\pi) - \sin(0) = 0 - 0 = 0\right)$



Here the error gets worse for smaller step sizes, but only for `trapz`. `SIMP42` appears to exhibit consistent behaviour as the step size changes.

Here, again `SIMP42` outperforms `trapz`.

– `plotinterrors(0, 10, @(x) sin(x.^2) + 1, 10.58367089992962334, 2*round(logspace(0, 6, 200))+1);`
$\left(\int_0^{10} \sin(x^2) + 1dx \approx 10.58367089992962334 \text{ from assignment spec}\right)$



Similarly to the first plot, `SIMP42` performs best. Interesting to note is the "jaggedness" on either side of the smooth descent in error for `SIMP42`. I believe the top jagged part is because large step sizes are a lot more volatile to overshooting, undershooting, or being accurate as an approximation. The bottom left jagged part, again, might be a failure to correctly obtain the error since it is so close to the machine epsilon.

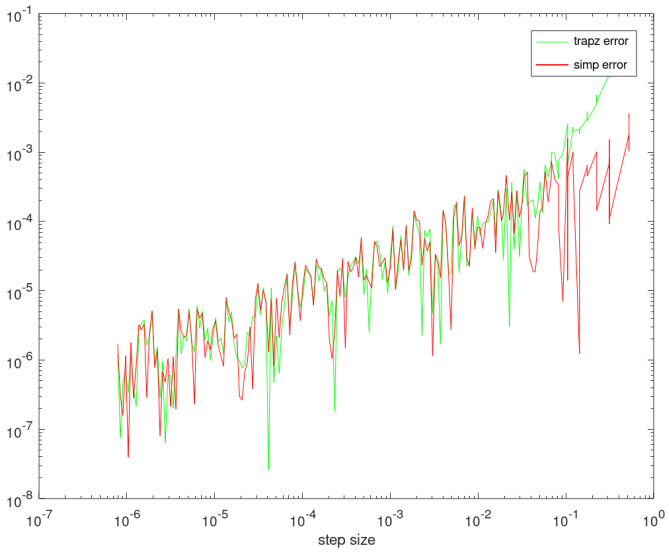
- We find the errors using adaptive-step `integral`:

```
>> e1 = abs(1 - integral(@(x) cos(x), 0, pi/2))
e1 = 0
>> e2 = abs(0 - integral(@(x) cos(x), 0, pi))
e2 =    3.9740e-17
>> e3 = abs(10.58367089992962334 - integral(@(x) sin(x.^2)+1, 0, 10))
e3 =    1.7764e-15
```

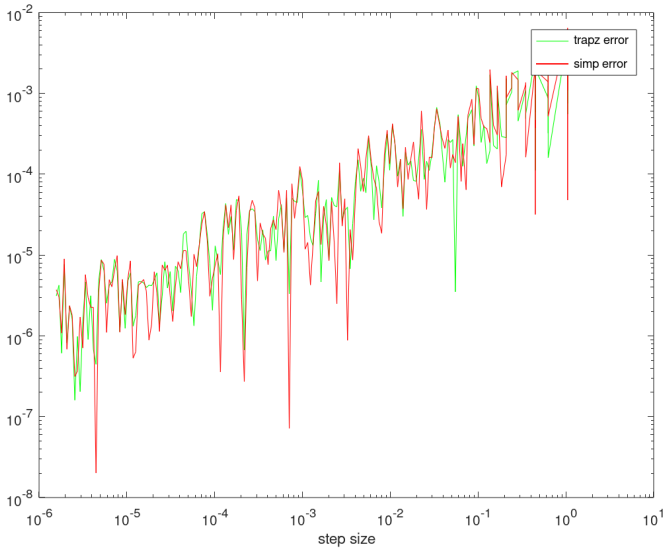
  - $\int_0^{\pi/2} \cos(x)dx$ was calculated with no error, whereas `SIMP42` was only able to get down to between about 10^{-15} to 10^{-16} error.
  - $\int_0^{\pi} \cos(x)dx$ was calculated within 3.97×10^{-17} of the exact answer. It is strange that this is distinguished from zero considering it is less than machine epsilon. This looks to be about the same error obtained by the fixed-step approximators.
  - $\int_0^{10} \sin(x^2) + 1dx$ was calculated within 1.77×10^{-15} of the exact answer. This appears to be about what was obtained with `SIMP42` for step size below about 10^{-4}.

- We will introduce random error to each of the plots from R1.3(a) and compare to their non-noisy counterparts.

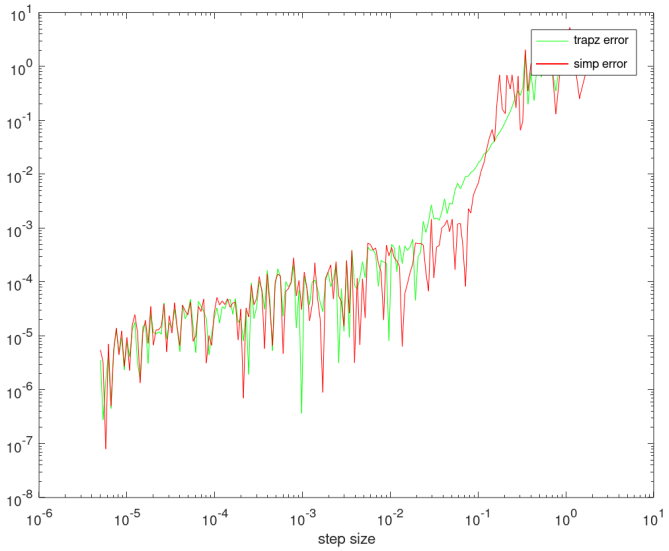
– `plotinterrors(0, pi/2, @(x) cos(x) + 0.001 * randn(size(x)), 1, 2*round(logspace(0, 6, 200))+1);`



– `plotinterrors(0, pi, @(x) cos(x) + 0.001 * randn(size(x)), 1, 2*round(logspace(0, 6, 200))+1);`



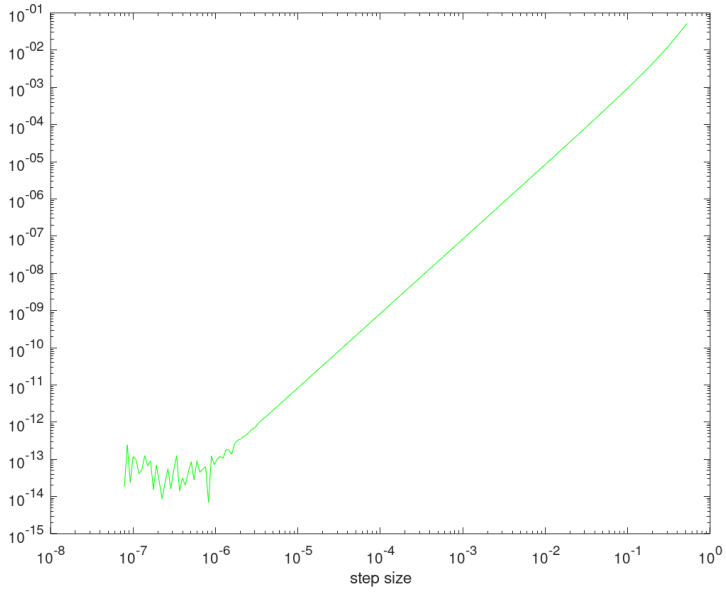
– `plotinterrors(0, 10, @(x) sin(x.^2) + 1 + 0.001 * randn(size(x)), 10.58367089992962334, 2*round(logspace(0, 6, 200))+1);`



We see that each plot is generally a lot more rough and the error doesn't dip down nearly as much as it did without the noise.

- I tried approximating $\int_0^{\pi/2} \cos(x)dx$ with `trapz` for 20,000,001 steps. This brought the step size down to around 10^{-7} but took a very long time to produce the plot on my machine.

  It appears it wasn't worth the time spent over the previous number of steps used because the accuracy hovered around the same minimum value for step sizes below about 10^{-6}.



I would say using a step size of 10^{-6} gives very good accuracy and is not terrible in speed. Going beyond that is not worth it due to additional error.

R1.4

Question

Integrate $\exp(-x)$ from $x = 0$ to $x = \infty$, by numerically integrating from $x = 0$ to increasing values of x. Plot a graph of the error vs the value of x you integrate to. Compare results for * a fixed step size and * a fixed number of steps.

Answer

Note that $\int_0^\infty \exp(-x) = 1$. We use this to calculate the error for each increasing final x to integrate to.

- `r1.4-fixed-step-size.m`

```
step_size = 10e-5;
xfinals = [];
errors = [];

exact = 1;
xfinal = 0;
thiserror = inf;
while thiserror > 10e-9;
```

```
    xfinal += 0.05;
    x = 0: step_size: xfinal;
    y = exp(-x);
    A = trapz(x, y);
    thiserror = abs(exact - A);
    xfinals = [xfinals; xfinal];
    errors = [errors; thiserror];
end;

figure;
semilogy(xfinals, errors);
xlabel("Final x");
ylabel("Error");
title(["Fixed step size: ", num2str(step_size)])
```

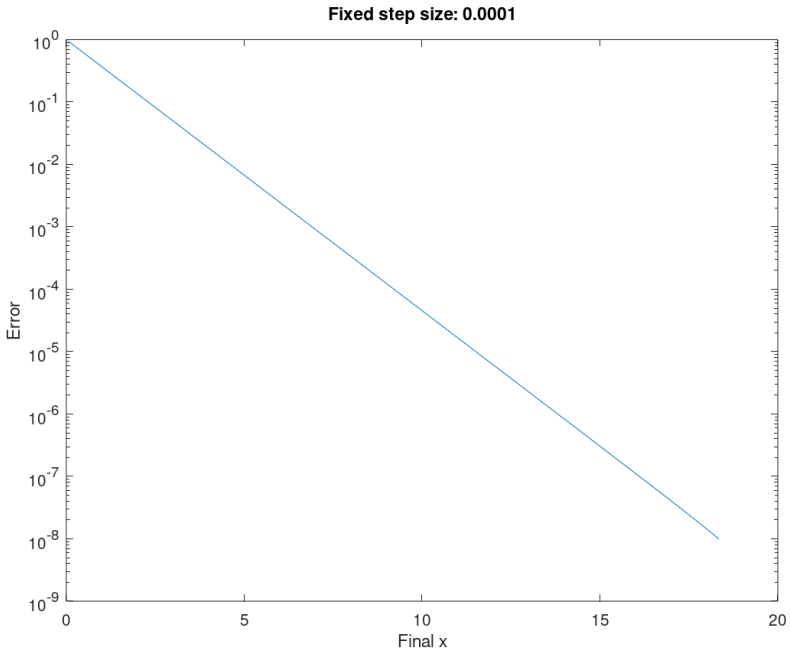


Figure 9: R1.4 fixed step size

With a fixed step size of 0.0001, an error of $\approx 9.9 \times 10^{-9}$ was reached after integrating to $x = 18.35$. It took 367 iterations and 1.76 seconds (on my machine to do so).

- `r1.4-fixed-num-steps.m`

```
num_steps = 10000
```

```
xfinals = [];
errors = [];

exact = 1;
xfinal = 0;
thiserror = inf;
while thiserror > 10e-9;
    xfinal += 0.05;
    x = linspace(0, xfinal, num_steps);
    y = exp(-x);
    A = trapz(x, y);
    thiserror = abs(exact - A);
    xfinals = [xfinals; xfinal];
    errors = [errors; thiserror];
end;

figure;
semilogy(xfinals, errors);
xlabel("Final x");
ylabel("Error");
title(["Fixed number of steps: ", num2str(num_steps)])
```

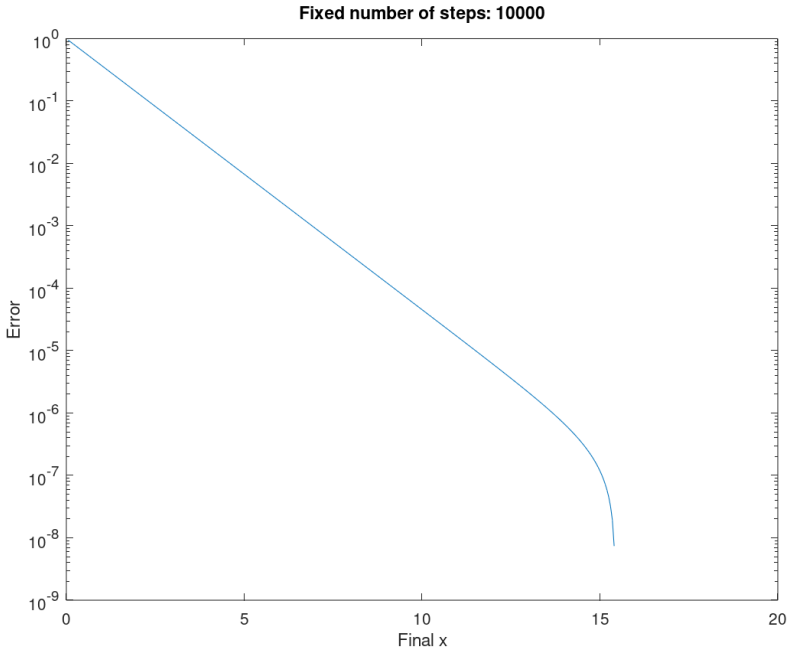


Figure 10: R1.4 fixed step size

With a fixed number of steps of $10,000$, an error of $\approx 7.4 \times 10^{-9}$ was reached after integrating to $x = 15.4$. It took 308 iterations and 0.56 seconds (on my machine) to do so.

Comparing the two methods, it looks like the fixed number of steps approach reached the minimum error threshold quicker, as seen by the steepened drop in error after $x \approx 15$. The fixed step size approach, on the other hand, stayed logarithmically constant in the error decline until the minimum error threshold was reached.

This also meant the fixed number of steps approach took less compute time.

A1.2

Question

While Matlab is an interpreted language, and therefore loops can be slow, operations on matrices and vectors can be much faster. Matlab code is often written to make use of vector operations instead of loops ("vectorisation"); this can lead to code that is fast, but requires much more memory. Compare the memory and time requirements for numerical integration in * Matlab using vector operations, * Matlab using loops, and * C using loops.

Answer

In each case, we calculate

$$\int_0^2 \sin\left(2x^3\right) dx$$

using Simpson's method (code used or modified from `SIMP42`).

- `a1.2-vectors.m`

```
% Calculate definite integratal of sin(2x^3) from 0 to 2 using
% Simpson's method
tic
x = linspace(0, 2, 500000);
% 500001 double-precision complex numbers (128 bits each): 8 MB.
y = sin(2.*x.^3);
% Another 8 MB - total: 16 MB.
A = trapz(x, y);
% Another 128 bits for A - insignificant.
toc
printf("Definite integral of sin(2x^3) from 0 to 2: %d", A);
% Elapsed time is 0.0338781 seconds.
% Total memory usage: roughly 16 MB.
```

Output:

```
Elapsed time is 0.0338781 seconds.
Definite integral of sin(2x^3) from 0 to 2: 0.394606
```

As counted in the comments, the memory consumption is roughly 16 MB.
The method of counting was used from [Piazza @39](https://piazza.com/class/kd9obdfovp31wi?cid=39).

Again as mentioned in the comments, the time spent was only about 0.03 seconds on my machine.

- `a1.2-for-loop.m`

```
% Calculate definite integratal of sin(2x^3) from 0 to 2 using
% Simpson's method
tic
N = 500001;
h = 2 / N;

Y_mid = 0;
Y_end = 0;
for n = 1: N;
  x = n * h;
  y = sin(2 * x^3);
  if mod(n, 2) == 0;
    Y_mid += 4 * y;
    if n >= 3;
      Y_end += 2 * y;
    end;
  end;
end;

A = h/3*(Y_mid + Y_end + sin(2 * 1^3)+ sin(2 * N^3));
toc
printf("Definite integral of sin(2x^3) from 0 to 2: %d", A);
% Elapsed time is 7.15432 seconds.

% Total memory usage: there are 7 distinct double-precision complex
% variables at 128 bits each - therefore 112 bytes. Insignificant.
```

Output:

```
Elapsed time is 7.15432 seconds.
Definite integral of sin(2x^3) from 0 to 2: 0.394606
```

The loop code was inspired by the provided `SIMP42.m` file. Instead of storing as everything, we just do the summation on the fly, using only about 112 bytes but taking over 7 seconds on my machine.

- `a1.2-for-loop.c`

```
#include <stdio.h>
```

```c
#include <math.h>
#include <time.h>

int main() {
  clock_t begin = clock();

  unsigned int N = 500001;
  double h = 2.0 / N;

  double Y_mid = 0.0;
  double Y_end = 0.0;

  for (unsigned int n = 0; n < N; n++) {
      double x = n * h;
      double y = sin(2 * pow(x, 3));

      if (n % 2 == 0) {
          Y_mid += 4.0 * y;
          if (n >= 3) {
              Y_end += 2.0 * y;
          }
      }
  }

  double A = h / 3.0 * (
      Y_mid + Y_end + sin(2 * pow(1, 3)) + sin(2 * pow(N, 3))
  );

  // Timing code copied from
  // stackoverflow.com/questions/5248915/execution-time-of-c-program
  clock_t end = clock();
  double time_spent = (double)(end - begin) / CLOCKS_PER_SEC;

  int memory_used = (
      2 * sizeof(clock_t) +
      2 * sizeof(unsigned int) +
      7 * sizeof(double) +
      sizeof(int)   // For this variable itself
  );
  printf("A = %g. "
      "Elapsed time is %g seconds. "
      "Memory used: %d bytes.\n", A, time_spent, memory_used);

  return 0;
}
```

Output:

```
A = 0.394606. Elapsed time is 0.031 seconds. Memory used: 76 bytes.
```

This code was basically just a syntactical translation of `a1.2-for-loop.m` in C. It runs significantly faster and even uses less memory because we have the option to specify sensible types for variables that don't need to store very precise / large numbers.

Method	Time (seconds)	Space (bytes)
Matlab vectors	0.03	1.6×10^7
Matlab loop	7.15	112
C loop	0.03	76

Clearly, the C method is the winner here considering they all produced the same answer for the integral approximation.