1 Choosing axis scaling

I've noticed that on the homeworks, some people are having trouble choosing appropriate scaling for the axes on their plots. Here are some examples that were chosen to help you decide when to use linear, loglog, or semilog scaling, avoid common mistakes and misconceptions, and design readable plots.

1.1 Polynomial approximations of e^x near zero

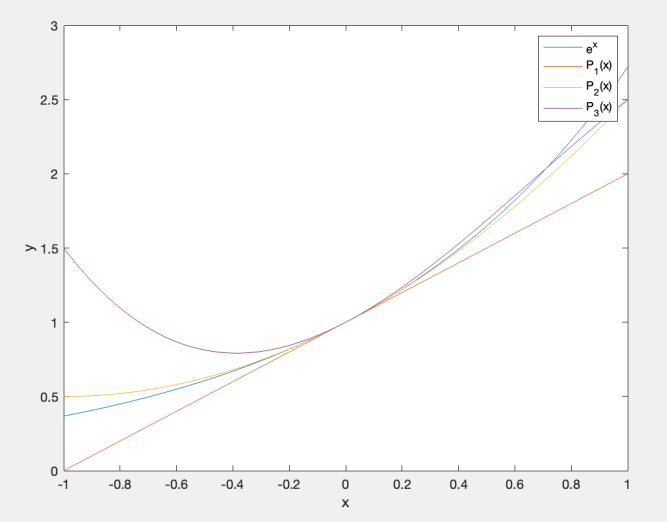
Consider the following three functions:

$$P_1(x) = 1 + x$$

$$P_2(x) = 1 + x + \frac{x^2}{2}$$

$$P_3(x) = 1 + x + x^2 - \frac{x^3}{2}$$

Here's what we see if we plot them together with the function $f(x) = e^x$:

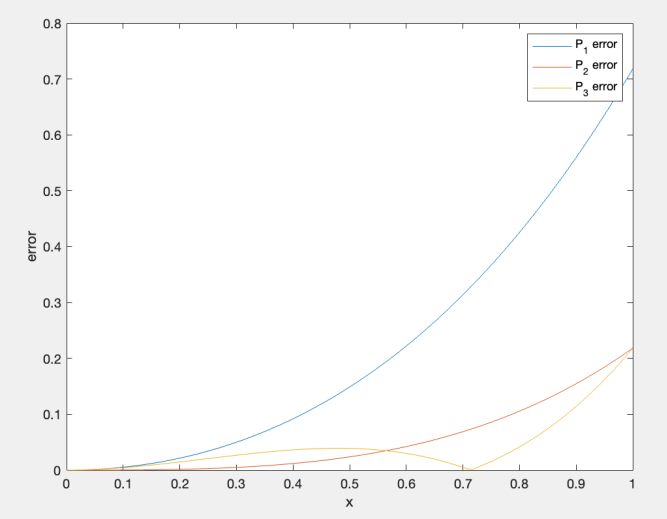


Notice that all three of these polynomials seem to "hug" the graph of $y = e^x$ very closely for small values of x close to $x = 0$, though the graphs diverge as x moves away from zero. We might ask the question: which of these functions approximates $y = e^x$ most closely for very

small values of x? To begin to answer this question, we might start by plotting the difference between each polynomial and the value of the exponential function at a sequence of x values, and trying to observe which one has an error value that approaches zero most quickly. We can plot the errors using the following code:

```
P1 = @(x) 1 + x;
P2 = @(x) 1 + x + x.^2/2;
P3 = @(x) 1 + x + x.^2 - x.^3/2;
x = linspace(0, 1, 100);
figure(1);
plot(x, abs(exp(x)-P1(x))); hold on
plot(x, abs(exp(x)-P2(x)));
plot(x, abs(exp(x)-P3(x)));
xlabel('x');
ylabel('error');
legend("P_1 error", "P_2 error", "P_3 error");
```

This gives us the following plot of the errors:

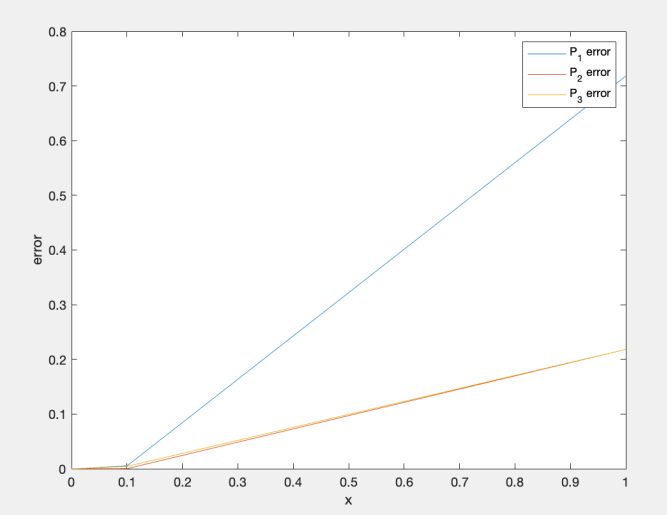


From this graph, it seems more clear that P_2 is a better approximation than P_1 and P_3, but it's kind of hard to see how all three errors behave really close to $x = 0$. If we want to see the behavior of the error very close to zero, rather than plotting at a grid of evenly spaced points `linspace(0,1,100)`, we might instead plot at a sequence of points whose magnitude decreases *exponentially*. For instance, we could plot the error at $x = 10^0, 10^{-1}, 10^{-2}, \cdots$ using the following code:

```
P1 = @(x) 1 + x;
P2 = @(x) 1 + x + x.^2/2;
P3 = @(x) 1 + x + x.^2 - x.^3/2;
x = 10.^-(0:10);
```

```
figure(1);
plot(x, abs(exp(x)-P1(x))); hold on
plot(x, abs(exp(x)-P2(x)));
plot(x, abs(exp(x)-P3(x)));
xlabel('x');
ylabel('error');
legend("P_1 error", "P_2 error", "P_3 error");
```

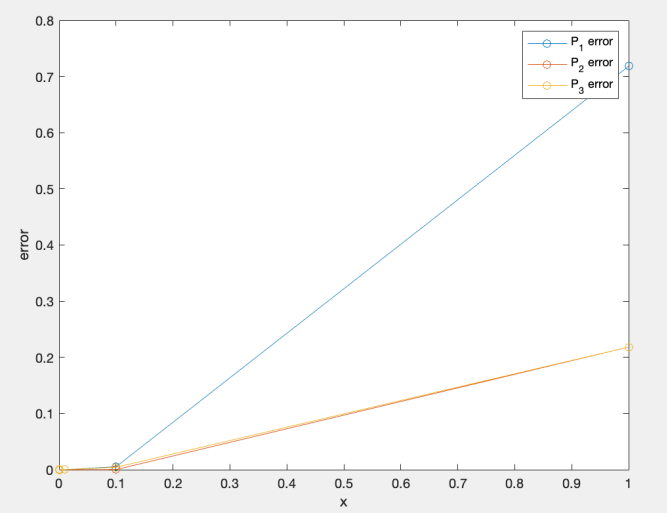
This gives the following plot:



This plot looks kind of funny. These curves *almost* look like straight lines, so we might be tempted to say that the error shrinks linearly as x approaches zero. But this would be mistaken! This graph is very misleading, because most of the data points have very small x-values, meaning that they are all clustered near zero. When MATLAB plots a curve through a collection of points, it draws *the line segment* joining each pair of consecutive points - so the part of this graph between $x = 0.1$ and $x = 1$ that looks like a straight line doesn't reflect how the error behaves on that interval, it just behaves that way because MATLAB is plotting a line segments between two points with $x = 0.1$ and $x = 1$ for lack of any data in between those two points! This phenomenon becomes more obvious if we ask MATLAB to actually display the data points on the graph, by modifying the `plot` lines as follows:

```
plot(x, abs(exp(x)-P1(x)), '-o'); hold on
plot(x, abs(exp(x)-P2(x)), '-o');
plot(x, abs(exp(x)-P3(x)), '-o');
```

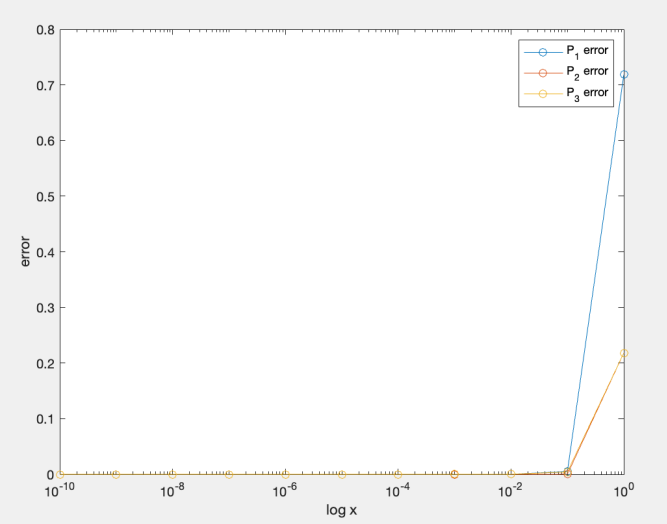
This gives the following graph:



Notice how there are no data points at all between $x = 0.1$ and $x = 1$, and several very close to $x = 0$? It doesn't make for a very good graph if we let just of the data points determine 90 percent of the graph, while all the rest are squished into a tiny little interval near zero. This is why it would be a good idea to use *logarithmic scaling* on the x axis, since our chosen x-values are decreasing exponentially. We can use the following code to do a semilogx plot which uses a logarithmic scale for the x-axis only:

```
P1 = @(x) 1 + x;
P2 = @(x) 1 + x + x.^2/2;
P3 = @(x) 1 + x + x.^2 - x.^3/2;
x = 10.^-(0:10);
figure(1);
semilogx(x, abs(exp(x)-P1(x)), 'o-'); hold on
semilogx(x, abs(exp(x)-P2(x)), 'o-');
semilogx(x, abs(exp(x)-P3(x)), 'o-');
xlabel('log x');
ylabel('error');
legend("P_1 error", "P_2 error", "P_3 error");
```

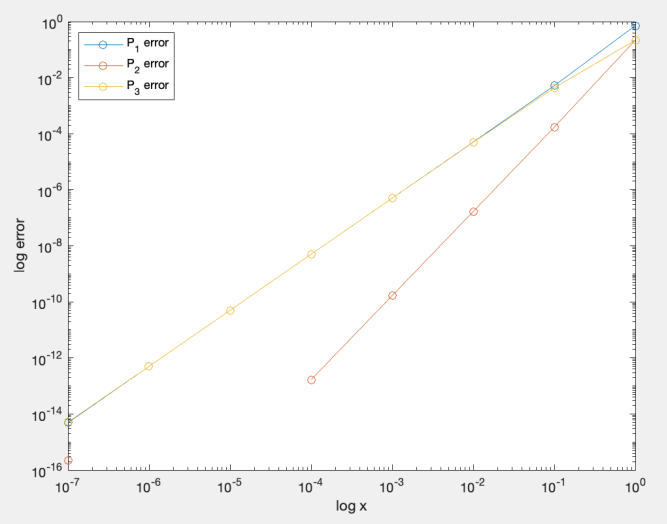
This gives the following graph:



Now we can see that the data points are pretty evenly spaced along the x-axis, but it's very difficult to see the differences between the error values as x gets smaller, because they're all very close to zero. In fact, it seems like they might be getting *exponentially* smaller as x shrinks exponentially. This means that rather than using a semilogx plot, we should probably use a *loglog* plot, so that both the x-values and the error values are plotted on a logarithmic scale. We can use this code:

```
P1 = @(x) 1 + x;
P2 = @(x) 1 + x + x.^2/2;
P3 = @(x) 1 + x + x.^2 - x.^3/2;
x = 10.^-(0:10);
figure(1);
loglog(x, abs(exp(x)-P1(x)), 'o-'); hold on
loglog(x, abs(exp(x)-P2(x)), 'o-');
loglog(x, abs(exp(x)-P3(x)), 'o-');
xlabel('log x');
ylabel('log error');
legend("P_1 error", "P_2 error", "P_3 error", location='northwest');
```

This gives us the following plot:



This plot is much more readable! It shows us that P_1 and P_3 have about the same error at $x \to 0$, whereas the error in P_2 decreases significantly faster. In fact, the error for P_2 grows so small for small values of x that it is rounded to zero during the calculation, which is why several points are absent from the graph of P_2 (because the log of zero is undefined).

In fact, using a loglog plot allows us to quantify just how much faster the error for P_2 approaches zero compared to the other two polynomials. By inspecting the graph, we can see that $\log(|P_2 - e^x|)$ is approximately a linear function of $\log(x)$ with a slope of 2, whereas $\log(|P_1 - e^x|)$ and $\log(|P_3 - e^x|)$ are approximately linear with a slope of 1. In other words,

$$\log(|P_1(x) - e^x|) \approx x + a$$
$$\log(|P_2(x) - e^x|) \approx 2x + b$$
$$\log(|P_3(x) - e^x|) \approx 2x + c$$

for some constants a, b, c. Exponentiating both sides of these equations yields the following:

$$|P_1(x) - e^x| \approx Ax$$
$$|P_2(x) - e^x| \approx Bx^2$$
$$|P_3(x) - e^x| \approx Cx$$

where $A = 10^a$, $B = 10^b$, and $C = 10^c$. Using big-O notation, we can express this as follows:

$$|P_1(x) - e^x| = O(x)$$
$$|P_2(x) - e^x| = O(x^2)$$
$$|P_3(x) - e^x| = O(x)$$

In other words, the error for the second polynomial shrinks *quadratically* as x approaches zero, but the error for the other two polynomials only shrinks *linearly*.

We've seen above how a loglog plot can be useful. In the following two shorter examples, we'll see where semilogy and semilogx plots can come in handy.

1.2 Partial sums of a geometric series

You might have seen the following infinite summation identity before:

$$\sum_{k=1}^{\infty} \frac{1}{2^k} = \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots = 1$$

What this equation actually means is that as we successively add together more and more of the first n terms of the series, the resulting value will get arbitrarily close to 1. For instance,

$$\frac{1}{2} + \frac{1}{4} = 0.75$$

$$\frac{1}{2} + \frac{1}{4} + \frac{1}{8} = 0.875$$

$$\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} = 0.9375$$

$$\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \frac{1}{32} = 0.9688$$

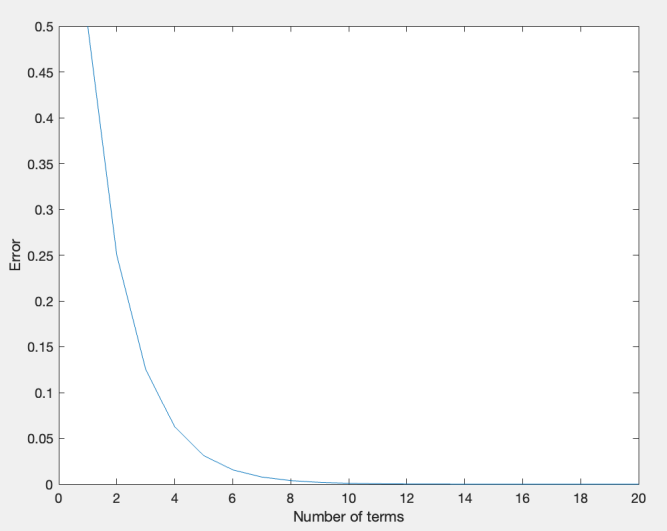
If we add up the first 20 terms, we will get the following:

$$\frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{2^{20}} \approx 0.999999046325684$$

We might be curious about exactly how fast this sequence of *partial sums* approaches the value 1. We can do this by plotting the difference between the sum of the first n terms and the exact value of 1 as n increases, using the following MATLAB code:

```
n = 1:20;
terms = 2.^-n;
sums = cumsum(terms);
plot(n, abs(1-sums))
xlabel("Number of terms")
ylabel("Error")
```

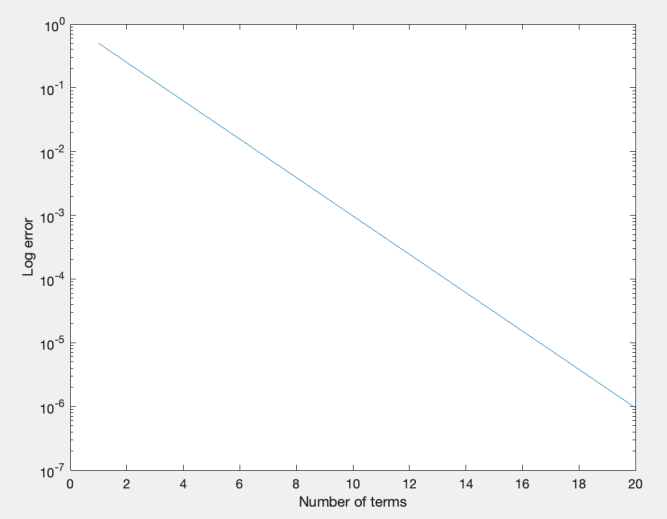
This gives the following plot:



We can see that the error seems to be decreasing exponentially. In fact, if you look closely, you might notice that it is being halved each time we add a new term. But for larger values of n, the error is so small that it's hard to see if the error is changing at all from one partial sum to the next, making this plot a poor representation of the data. Since the error data seems to be decreasing exponentially as the x-values increase linearly, we should use a semilogy plot to represent the data:

```
n = 1:20;
terms = 2.^-n;
sums = cumsum(terms);
semilogy(n, abs(1-sums))
xlabel("Number of terms")
ylabel("Log error")
```

which gives the following plot:



Notice that this graph looks like a straight line, which confirms that the error values are indeed decreasing exponentially. As a matter of fact, if you calculate the slope of this line, you might find that its slope is close to $-0.301\cdots = -\log_{10}(2)$. This means that

$$\log(|S_n - 1|) \approx \log(2) \cdot n + a$$

for some constant a. If we exponentiate both sides of this equation, we find that

$$|S_n - 1| \approx A \cdot 10^{-\log(2)\cdot n} = \frac{A}{2^n}$$

where S_n is the sum of the first n terms of the series and $A = 10^a$. As a matter of fact, it can be proven mathematically that the error is always *exactly* equal to $1/2^n$, which confirms this experimental observation (the value of A would then be equal to 1). You can try to prove this as an exercise, if you want.

1.3 Growth of the harmonic numbers

Let's look at a different kind of sum. Consider the *harmonic numbers* H_n, which are defined as follows:

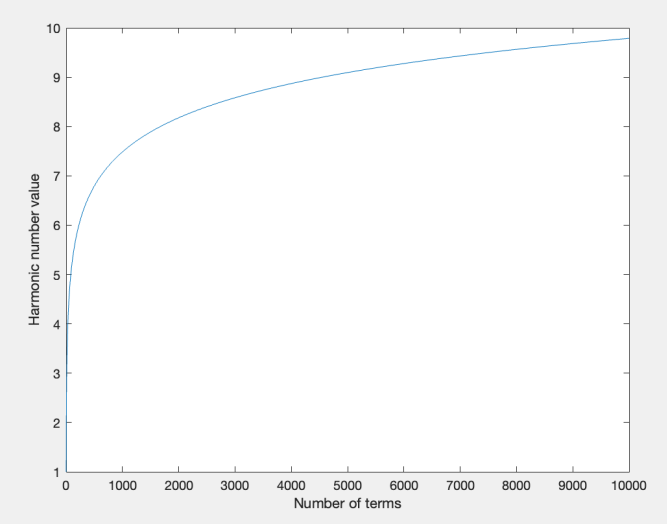
$$H_n = \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$$

Let's think about how these sums increase as a function of n. Unlike the sums of the reciprocals of the powers of 2, considered in the previous example, the values of the sums H_n *do not* approach any finite value, but rather grow arbitrarily large. This might be surprising, since the value that we are adding to the sum gets smaller and smaller each time! But we can observe this phenomenon using a numerical experiment in MATLAB. (Proving this fact mathematically is another thing entirely! If you're curious, maybe you can try it yourself.)

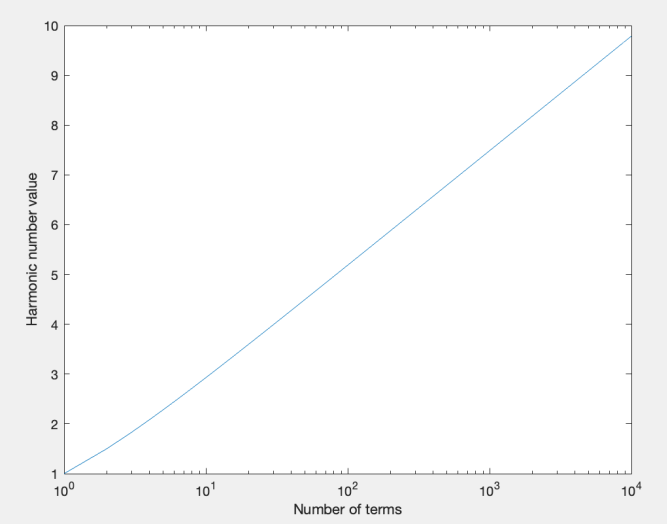
The following code can be used to plot the values of H_n for $n = 1, 2, \cdots, 10000$:

```
n = 1:10000;
terms = 1./n;
sums = cumsum(terms);
plot(n, sums);
```

which gives us this plot:



Sure enough, it looks like the harmonic numbers continue to increase significantly all the way up to $n = 10,000$, although the rate at which they grow slows down quite a bit after the first 1000 or so terms. However, this plot isn't *entirely* convincing. After all, maybe this sum does in fact converge to some value larger than 9, but just takes an extraordinarily long time to do so, with the error shrinking very slowly. If you inspect the value of H_n closely, you might notice that doubling the number of terms increases the harmonic numbers by about a constant amount. If this were always true, then of course the values of H_n would grow to infinity - to find an arbitrarily large value of H_n, we would just have to double the number of terms enough times. To observe this phenomenon graphically, we can make a *semilogx* plot by changing `plot` to `semilogx` in the above code:



Check it out! This graph looks like a straight line, indicating that H_n is approximately $C \cdot \log(n) + D$ for some constants C and D, for large values of n. This is a bit more convincing, since this curve is a very straight line when plotted using `loglog`, and we know that the logarithm function grows unboundedly large (albeit very slowly).

1.4 Summary

Here's a list of key takeaways:

- Try to pick your axis scaling so that all of the data points aren't clumped together in one place, so that it's easy to see the trend.
- If data values grow very close to zero but are plotted linearly, it can be difficult to see exactly how quickly they approach zero, since it's hard to see the difference between, say, 0.01 and 0.001 on a linear scale.
- If two data points are very far apart, MATLAB will plot a long line segment between them, which might misleadingly suggest that the trend is linear when it really isn't.
- In general, `loglog` is useful when the dependent variable is approximately a polynomial/power function of the independent variable, `semilogy` is useful when the dependent variable is approximately an exponential function of the independent variable, and `semilogx` is useful when the dependent variable is approximately a logarithmic function of the independent variable.
- You might have to experiment with a couple different plots before finally choosing the most appropriate scaling.