Lesson 0 TEST

This is a test exercise set that will be done *in class* as part of "homework 0". *Do not try to hand this in.* But if you want to get a running start on Matlab, by all means read the first half of lesson 1 and then try to do these exercises all by yourself first! (If you get lost, note that the solution is already posted on the top course web page.)

Note for Octave users: put file setup.tex in the folder with your script files. You can get better looking results with Octave than Matlab. (But, warning!, Octave is buggier.)

1. Use Matlab to evaluate:
   (a) $\frac{2+3}{4}$. Use a code comment line before the Matlab code to say whether you need parentheses or not.
   (b) $\sin(30^\circ)$. Do this both using the Matlab `sind` function and the `sin` function. Show each result *immediately* behind the Matlab command that creates it.

   Bare solution script

2. Store in variable `averageGrade` the value of average exam grade, if the two exam grades were 45 and 90. Print the value in `averageGrade` out separately from storing the number in it.

   To show that indeed nothing is printed in the assignment statement, follow it by a double-precent line immediately followed by some mark-up comment.

   Bare solution script

3. Matlab has a function `sqrt` that returns the *square root* of its argument. But Matlab does not have a function that returns the *square* of its argument. Fix this by defining your own function `sqr` that returns the square of its argument. Test it on the values 3 and 4, and on a variable called `myVar` that contains the value 5.

   Use double percent lines to show each test separately.

   Remember that function `sqr` must be in a function file called `sqr.m`.

   After all works well, add comments to function file `sqr.m` that explain what the function does and what its input argument must be. Then add a test that the command "`help sqr`" does give proper information on how to use the function.

   Bare solution script

Lesson 1 INTRODUCTION

Before doing the below exercises, you should first read and try to understand the relevant parts of posted lesson 1. If you did not, neither the instructor nor any TA will help you with problems until you do.

1. Use Matlab to evaluate:

   (a) $2 + \frac{6}{4}$.

   (b) $\frac{2+6}{4}$.

   (c) 65 F in degrees Centigrade.

   (d) The circumference of a circle of radius 3.

   (e) The number of molecules in a mole, using exponential form.

   (f) $\cos(10^2\pi)$, and explain whether the answer is correct.

   (g) $\cos(10^{20}\pi)$, and explain whether the answer is correct.

   (h) The tan of 45 degrees, and explain whether the answer is correct.

   (i) arccos(-1), and explain whether the answer is correct. If this is unsure, say why.

   (j) Euler's number e, but do not print out the value.

   (k) Only now print out the value of e, twice. Use a "%%" line between the two printouts to show the result of the first print-out command before the second print-out command. State whether the answer is correct.

   Be sure to start the subexercises with "%% a)", "%% b)", ... lines in your script so that they are shown separately and found in the contents of your pdf.

   Bare solution script

2. Define a function `Circle_Area` so that `Circle_Area(r)` returns the area of a circle of radius `r`. Use the function with arguments 1, 2, `r`, and `x`, where r is a variable whose value is 1 and x is a variable whose value is 2. Use "%%" lines to separate the four evaluations from each other. Check in each case that you get the correct area.

   Remember that function `Circle_Area` must be in a function file called `Circle_Area.m`. Use "New," "Function" in Matlab and save appropriately.

   Bare solution script

3. Define a function `Cone_Volume` so that `Cone_Volume(r,h)` returns the volume of a cone whose base is a circle of radius `r` and whose height is `h`. Use the function with arguments (1,3), (3,1), and (2,7). Check that in each case, you get the correct volume. Store the three values in variables V1, V2, and V3 before printing them.

   Bare solution script

   .....

4. Use Matlab to evaluate the following quantities. Use a separate subsection for each subexercise. Also, for each answer, use a double-percent line followed by mark up text to comment on why you think the answer should be like this (or not).

   (a) 1/Inf

   (b) Inf/Inf

   (c) Inf-Inf

   (d) $((10^{10})^{10})^{10}$

   (e) $1 + (3 * (5^{0.5}))$ in the order shown using the minimal number of parentheses needed.

   (f) $((1 + 3) * 5)^{0.5}$ in the order shown using the minimal number of parentheses needed.

   (g) Evaluate Bessel function $J_0(x)$ at $x = 1$. (Hint: Probably you do not know this function. To figure it out, use the `help` command. Note that Matlab uses lowercase for function names. Then try Tab completion. The correct value of $J_0(x)$ at $x = 1$ starts as $0.7\ldots$)

   Bare solution script

5. Assign the values 1, 2, and 3 to variables A, B, and C, respectively. Then move the original value of B to C, of A to B, and of C to A (without using the values explicitly). Generalize the procedure used in the lesson to do this. Use only one additional variable. Use double percent lines and mark-up text as appropriate.

   Bare solution script

6. Reconsider your function `Cone_Volume(r,h)` from exercise 3. Copy it into a function named `ConeVolume(r,h)`, (no underscore), then fix `ConeVolume` so that it works correctly even if radius `r` and height `h` are arrays, call them `rVals` and `hVals`. Try it out with arrays `rVals` = [1,3,2] and `hVals` = [3,1,7] and show that you get the same three volumes in the resulting array `VVals` as in exercise 3. Also try it out with `rVals` = [2,3,4,5,6] and `hVals` = [11,9,7,5,3], each written as concisely as possible (i.e. with the minimum number of characters possible, using the Matlab colon notation). Use double percent lines where appropriate.

   Also solidly comment your function, fully explaining purpose, input arguments and output argument. Demonstrate that

   ```
   help ConeVolume
   ```

   produces full information on your function to anyone who enters that command. As an example, see function `Cone_Volume` in the posted solution of exercise 3.

   Bare solution script

Lesson 2 ZEROS OF FUNCTIONS

Before doing the below exercises, you should first read and try to understand the relevant parts of posted lesson 2. If you did not, neither the instructor nor any TA will help you with problems until you do.

Homework Motivation: Consider a drum whose membrane is flexibly attached to the drum rim. If you hit such a drum in the center of the membrane, the nondimensionalized frequencies (tones) ω that are produced satisfy the equation

$$J_0(\omega) = k\omega J_1(\omega)$$

where J_0 and J_1 are Bessel functions of the first kind and the given constant k is a nondimensionalized flexibility of the membrane attachment.

(No, you do not need to know how to derive the above equation. All you are asked to do is solve it. And no, you do not need to know what Bessel functions are either. The only thing you need to know, from homework 1, is that Matlab can evaluate it for you.)

1. Since you probably do not know how the two Bessel functions in the *motivation* above look, plot both $J_0(\omega)$ and $J_1(\omega)$ in a single graph. In particular, plot the values of these two functions at 201 equally spaced ω plot values between 0 and 3.5π. (See help on the Matlab `linspace` command on a simpler way to create the ω values than by defining an array using START:STEP:END.) Make J_0 blue and J_1 red. Use a suitable title, suitable axes labels, a legend to distinguish J_0 from J_1, and a grid. The horizontal axis length should be 3.5π, starting from 0. The tick marks on the horizontal axis should correspond to whole multiples of 0.5π and labeled as such (instead of with raw numbers). (See "how about the other frequencies?" in the posted lesson2 for how to select tick marks and labels.) Let Matlab decide the size of, and labels on, the vertical axis (specify the vertical limits as '`-Inf Inf`').

   Bare solution script

2. Next create a function `DrumFreqEqError0pt5` that gives the error in the equation in the *motivation* above. To keep it simple, in this function assume that $k = 0.5$. Make sure that this function multiplies correctly even if its input argument is an array. And that the function does not print out any values while it is doing its thing (use appropriate semicolons). The function must be in a separate function file named `DrumFreqEq0pt5.m`. Note that when you save the function file the first time, Matlab will add the `.m` if you do not specify it yourself.

   Plot this function in your homework script file `l2_Zeros_x2.m`. The plot should satisfy the same requirements as the previous one (except for the legend). So use cut and paste where appropriate.

   Function `DrumFreqEq0pt5.m` should be `included` in your homework script so that the grader can find it in the `published` print-out that you hand in. The provided template script `l2_Zeros_x2.m` already has the needed `include` in it. Just don't mess it up. And avoid typos in the function name.

   Bare solution script

3. By looking at your graph, ballpark a very close value `omega1Approx` for the first frequency ω_1. Take the ballpark to be some suitable integer multiple of 0.5π.

   (a) Use this ballpark to let `fzero` find the correct frequency ω_1 (`omega1` in Matlab) to about 16 digits accuracy.

   (b) As a better approach, ballpark a frequency range in which the first frequency, and no other, is located. Take the end points of this range to be integer multiples of π.

   You must now first check that the errors at the end points are of different sign. That is to ensure that a zero crossing must be in your range. To do so, put the chosen frequency range into `DrumFreqEq0pt5.m` and check whether the two numbers are of opposite sign.

   Then let `fzero` again find the root, now by searching in the range.

   (c) Check that you get the same answer as before. In particular, print the difference between the two values of ω_1 and comment whether it is small enough. Note that the range method always works, if used correctly. The initial point method can fail (and readily does so in Octave).

   Bare solution script

4. We no longer want to assume a priori that $k = 0.5$. So, create a function `DrumFreqEqError` with input arguments ω and k. It must return the error in the frequency equation for any ω and k.

   (a) To check, show that for $k = 0.5$ and the same ω range as before, you get the same errors at the end points of the range as with `DrumFreqEqError0pt5`.

   (b) Your function should be very well commented; compare the posted lecture notes of lesson 1 and the listing of `FreqEqError` in lesson 2. Show to the grader that

   ```
   help DrumFreqEqError
   ```

   gives clear and complete information on your function to any user.

   (c) Next change k to 2 and remake the plot of the error versus ω now using `DrumFreqEqError` with k equal to 2.

   Bare solution script

5. Finally, use `DrumFreqEqError` to find the first four frequencies ω_n for $n = 1$ to 4 and $k = 2$. Print these out neatly.

   (a) First do it from initial ballparks. As initial ballpark use some suitable odd multiple of $\pi/4$ for each case. Your ballpark should be very accurate if n is high enough.

   Display the results in a neat table using `fprintf` as

   ```
   For k = 1.1, omega1 is: 12.1234567 (12.123 approximate).
   ```

   where 12.1234567 means 2 digits in front of the decimal point and 7 behind it, and similarly for 1.1 and 12.123. The print out must be achieved using the correct format specifiers inside the `fprintf` FORMATSTRING. (Actual data numbers inside FORMATSTRING are *never* allowed!)

   Warning: the format `%1.1f` does *not* mean 1 digit before the decimal point and 1 behind it, as some students think. The first 1 should be the total number of print positions, including leading spaces, digits before the decimal point, the decimal point itself, and the single digit behind it. So `%1.1f` is not just wrong, but impossible.

(b) Next do it by having `fzero` search in a range. As end points of the ranges use integer multiples of π.

In this case, print the results out as

```
For k = 1.1, omega1 is: 12.1234567 (in [*.123 12.123]).
```

using similar requirements as before. Here *.123 means as many digits in front of the decimal point as `fprintf` needs, and 3 behind it. The range you selected must be in the square brackets.

If you do it correctly, the decimal points of the frequencies should align in the tables, for a neat appearance.

Bare solution script

Lesson 3 INTERPOLATION

Before doing the below exercises, you should first read and try to understand the relevant parts of posted lesson 3. If you did not, neither the instructor nor any TA will help you with problems until you do.

1. You measured a function f that, "unknown to you," is exactly equal to

$$f_{\text{exact}}(t) = \frac{1}{2} + \sin\left(\frac{t}{2}\right) + \frac{1}{3}e^{t/3}$$

   Create a handle `fExactFun` to an anonymous function that evaluates f at a given time t as above. Check that at time $t = 1$, your function gives you 1.44 as it should.

   Now assume you did 8 measurements of function f at equally spaced times from -3 to 3. Put the measured times in an array `tMeasured`. Determine what the measured f-values should have been according to `fExactFun` and put them in array `fMeasured`.

   Now, using linear and spline interpolation of these "measured" data, evaluate f at times -3.5, -1.5, 0, 1.5, and 3.5.

   To get the errors in these values, take the absolute value of the difference between interpolated and exact f-values at that time.

   (Note: Do each of these times separately, -3.5 first. After you have done -3.5, use cut, paste, and modify to do the other times.)

   Print the obtained values and their errors out in the format

```
At t = 12.1:
the linear interpolate is 12.123, with error *.12E*;
the spline interpolate is 12.123, with error *.12E*;
the error in the spline is *.1 times smaller.
```

   using an `fprintf` with suitable FORMATSTRING format specifiers for each line. (Actual data numbers inside FORMATSTRING are *never* allowed!) Here 1 means 1 digit, 12 means 2 digits (or a sign or space and a digit), etcetera, and `*` means whatever number of digits `fprintf` wants.

   For the linear interpolation, do not forget that the times -3.5 and 3.5 are extrapolations, not interpolations. Modify the `interp1` call appropriately for these two times. And in the print-outs for these two times, change `interpolate` into `extrapolate`.

   Bare solution script

2. Continuing the previous exercise:

   (a) Create a plot of the interpolations. In particular, plot the exact solution as a black dashed line. Use 241 plot points from -4.5 to 4.5 to plot it. Also plot the measured data as black circles in the plot. And also plot the linear and spline interpolates at the 241 plot points in the plot. Use blue for the linear interpolate and red for the spline one.

   Your horizontal axis should go from -4.5 to 4.5 and your vertical axis from -1 to 3. Use title "Comparison of Interpolations", and axis labels "t" and "f". Include legend

entries "Exact", "Measured", "Linear" and "Spline." (Use `help legend` to figure out how to put the legend in a better position in the graph than on top of the curve.) On the horizontal axis, use tickmarks *only* at $t = -3$ and $t = 3$; this will make the interpolated range stand out from the extrapolated parts.

Use mark-up text to comment on your observations from the plot.

(b) Next evaluate the maximum error in the each of the two interpolation methods in the complete plot range. Use the interpolated values at the plot points to do so.

Also evaluate the maximum error in the each of the two interpolation methods in just the interpolation range. The easiest way for you to do this at your current knowledge is to create 161 "Test" points from -3 to 3, and evaluate the interpolations and then their errors at these points.

Use suitable `fprintf` commands to print out the errors neatly. Use mark-up text to compare the errors in the two ranges and comment on that.

Bare solution script

3. Continuing the previous exercises, now assume that we have done measurements at 64 equally-spaced points from -3 to 3, instead of just 8. But also assume that the measured f-values at these points have random errors that are on average 0.1 large (so about 3% of the total range of f)

In particular, Matlab users should create the new measured f-values as follows:

```
rng('default')
fMeasured=fExactFun(tMeasured)+0.1*randn(size(tMeasured));
```

Octave users should use instead:

```
randn("seed",4)
fMeasured=fExactFun(tMeasured)+0.1*randn(size(tMeasured));
```

You should also increase the number of plot points from 241 to 721 and the number of test points from 161 to 481.

Now repeat the previous exercise. Use cut and paste and then make appropriate changes, including in your conclusions.

Bare solution script

. . . . . .

4. Repeat the previous exercise, but instead of *interpolating* the measured data, *fit* a straight line, a cubic, and a quintic to the data.

Plot the fitted line in green, the cubic in cyan, and the quintic in magenta (all in the same graph). Return to using 241 plot points and 161 test points.

Comment appropriately.

Bare solution script

5. Continuing the previous exercises, integrate the linear and spline interpolates of the 8 exact data points from a starting point $t_1 = -1$ to an end point $t_2 = 3$. Compare with the exact result

$$\int_{t_1}^{t_2} f_{\text{exact}}(t)\,\mathrm{d}t = \frac{1}{2}(t_2 - t_1) - 2\cos\left(\frac{t_2}{2}\right) + 2\cos\left(\frac{t_1}{2}\right) + e^{t_2/3} - e^{t_1/3}$$

Print out the results as

```
For the integral with exact value 1.12345678
linear interpolation  gives 1.123, with error 1.12E*;
spline interpolation  gives 1.123, with error 1.12E*;
the straight line fit gives 1.123, with error 1.12E*;
the cubic  curve  fit gives 1.123, with error 1.12E*;
the quintic curve fit gives 1.123, with error 1.12E*.
```

Comment on the results.

Note: in Octave integration of the linear interpolate is slow.

Repeat for the three polynomials fitted to the 64 noisy data. Use format

```
For the integral with exact value 1.12345678
the straight line fit gives 1.123, with error 1.12E*;
the cubic  curve  fit gives 1.123, with error 1.12E*;
the quintic curve fit gives 1.123, with error 1.12E*.
```

Bare solution script

6. Continuing the previous exercises, for the case with the 64 noisy data, find the derivatives of the three fitted polynomials and plot them all three together, with the same colors as before (green, cyan, magenta). Also plot the exact derivative as a black broken line. Comment on the quality of the approximations. Similar plot requirements as before. The vertical axis range should be a bit bigger than the vertical range of the exact derivative; take it from -0.4 to 0.8.

Bare solution script

Lesson 4 ORDINARY DIFFERENTIAL EQUATIONS

Before doing the below exercises, you should first read and try to understand the relevant parts of posted lesson 4. If you did not, neither the instructor nor any TA will help you with problems until you do.

1. Your city is being attacked by a ship. You want to shoot cannon balls at it, but the ship may be too far away. To answer that exercise, first you need to create a system of ordinary differential equations that describes the motion of the cannon ball in air.

   To do so, the first step is identify the equations of motion of a cannon ball in air, but without significant spin. They are:

$$\frac{\mathrm{d}x}{\mathrm{d}t} = u \tag{1}$$
$$\frac{\mathrm{d}y}{\mathrm{d}t} = v \tag{2}$$
$$\frac{\mathrm{d}u}{\mathrm{d}t} = (-F_{\text{air}}u/V)/m \tag{3}$$
$$\frac{\mathrm{d}v}{\mathrm{d}t} = (-F_{\text{gravity}} - F_{\text{air}}v/V)/m \tag{4}$$

   where x is the horizontal distance traveled from the cannon, y the height above the cannon, u the horizontal velocity component in the x-direction and v the vertical velocity component in the y-direction.

   Also m is the mass of the cannon ball, which has radius r and is made of iron with a density ρ_{iron} equal to 7,272 kg/m^3.

   Also

$$V = \sqrt{u^2 + v^2} \qquad F_{\text{gravity}} = mg \qquad F_{\text{air}} = C_D \pi r^2 \tfrac{1}{2} \rho_{\text{air}} V^2$$

   where the acceleration of gravity g at your particular city is 9.81 m/s^2, the drag coefficient C_D can be taken to be 0.5, and the density of air ρ_{air} at your city to be 1.225 kg/m^3.

   Write a well-documented Matlab function CannonBall that takes as inputs (a) the time t from firing the cannon, (b) a vector consisting of values of the four unknowns (x, y, u, v), and (c) the cannon ball radius r. Your function should output the corresponding values of the derivatives of the unknowns, as above, in a *column* vector.

   Warning: The provided script `14_ODE_x1.m` already contains code to test your function. Do not modify that script. Just run it to make sure that your function works properly. If it does not, fix the problems before attempting the next exercise.

   Bare solution script

2. Continuing the previous exercise, the next step is to figure out how far your cannon can really shoot. Assume that your cannon can shoot the cannon ball out at an initial total speed V_0 of 100 m/s. Also assume that you shoot the cannon ball out at an angle α from the horizontal. In that case the *initial* velocity components are equal to

$$u_0 = V_0 \cos(\alpha) \qquad v_0 = V_0 \sin(\alpha)$$

Ignoring air resistance, the subsequent motion of the cannon ball is described in typical basic physics classes. These basic physics classes show that without air resistance, your cannon ball will travel farthest when the initial angle α is equal to 45 degrees. The cannon ball will then travel 1,019 m in 14.42 seconds to a ship at the same height as the cannon.

But that ignores air resistance. So you need to know how the distance travelled changes due to air resistance.

To find out, use the function `CannonBall` of the previous exercise to find the motion of a cannon ball of 10 cm radius in 14.42 seconds using the initial data as described above. Let `ode45` produce the solution at 100 equally spaced times in that time range.

Plot the found path of the cannon ball as y versus x (not against time), in red. To do so neatly, you will want to take the x and y values out of `unknownValues`.

Also put the solution without air resistance in the graph,

$$\begin{aligned} x_{\text{nodrag}} &= V_0 \cos(\alpha) t \\ y_{\text{nodrag}} &= V_0 \sin(\alpha) t - \tfrac{1}{2} g t^2 \end{aligned}$$

as a broken black line. Evaluate this at the same 100 equally spaced times as described above.

Add an appropriate title, axis labels, and legend of course. Put the x-axis location at the origin.

Bare solution script

Lesson 5 LINEAR ALGEBRA

Before doing the below exercises, you should first read and try to understand the relevant parts of posted lesson 5. If you did not, neither the instructor nor any TA will help you with problems until you do.

1. Consider a power line between two poles. It sags down under its own weight. Let h be the height of the power line above the ground. The heights h_1, h_1, $\ldots h_6$ at 6 points equally distributed along the first half of the power line satisfy the approximate equations

$$\begin{aligned} h_1 &= H \\ h_1 - 2h_2 + h_3 &= \rho/25 \\ h_2 - 2h_3 + h_4 &= \rho/25 \\ h_3 - 2h_4 + h_5 &= \rho/25 \\ h_4 - 2h_5 + h_6 &= \rho/25 \\ 2h_5 - 2h_6 &= \rho/25 \end{aligned}$$

   Here H is the scaled height of the power line at the pole; assume it to be 1 (that is how it was scaled). Also ρ, `rho`, is the scaled mass of the power line per unit length; take this to be 1 too.

   Write this system of 6 equations in 6 unknown heights in matrix-vector form as $M\vec{h} = \vec{r}$. Here $\vec{h}$ is the vector containing the unknown heights h_1, h_2, $\ldots h_6$, as a column.

   The vector $\vec{r}$ must contain the right hand sides of the equations given above, as a column. Create this vector in Matlab and name it `rhv`.

   Also create the matrix M in Matlab, naming it `Mat`. To create the matrix, note that each row in A contains the coefficients in the left hand side of the corresponding equation shown above. The first coefficient in each row in A is the coefficient multiplying the first unknown, h_1, or is zero if h_1 does not appear. Similarly, the second coefficient is the coefficient multiplying h_2 or zero if h_2 does not appear, etcetera.

   Check whether the equations have a meaningful solution; use mark up text to discuss this. In particular, estimate the relative error in the solution that will be caused by the fact that floating point numbers in Matlab are stored to only about 16 significant digits. State how many significant digits in the solution that error does about correspond to. The grader should see the final matrix `Mat`, the right hand side vector `rhv`, and the estimated relative error due to errors in Matlab processing `relErrMatlab`. Do not throw large amounts of junky other info at the grader; use semicolons where appropriate.

   If you conclude that a reasonably or highly accurate solution $\vec{h}$ to the above equations will be found, (and you should, but explain clearly *why*), find it. Call the solution $\vec{h}$ `heightValues` in Matlab.

   Then plot these heights against the horizontal position of the points. (The points are located at 6 equally spaced x-positions from 0 to 50% of the distance between the poles. Call the array of these x-values `xValues`) Plot the points as black circles connected by straight black lines. Provide appropriate axes labels and title, and extend the horizontal axis for no more

than the 50%. For horizontal axis label, use "position, percent". For vertical axis label use "height." Does it look roughly like half a sagging power line? It should.

As a final twist, replace the first equation $h_1 = H$ by the equation $h_1 - h_2 = 0$, which requires that h_1 and h_2 are the same (which they *are* by approximation). Do *not* create a new matrix and right-hand side vector from scratch. Just change the few coefficients in the existing matrix and right hand side vector that are no longer the same. If you find that the new system has no longer a meaningful solution, (and you should), say so. However try to solve anyway and comment on what you get using `disp`. If you get some numbers but they are no good, explain why. If you get no numbers, say so. Do not try to plot this.

Bare solution script

2. Suppose you know the values of some function f at n x-values x_1, x_2, ..., x_n. Then you can interpolate a single polynomial in x of degree $n-1$ through the known values $f(x_1)$, $f(x_2)$, ..., $f(x_n)$. This polynomial will take the general form

$$p_{\text{int}} = C_1 x^{n-1} + C_2 x^{n-2} + \ldots C_{n-1} x + C_n$$

From that you can see that the coefficients of the polynomial C_1, C_2, ..., C_n need to satisfy the following n equations:

$$\begin{pmatrix} x_1^{n-1} & x_1^{n-2} & \ldots & x_1^3 & x_1^2 & x_1 & 1 \\ x_2^{n-1} & x_2^{n-2} & \ldots & x_2^3 & x_2^2 & x_2 & 1 \\ x_3^{n-1} & x_3^{n-2} & \ldots & x_3^3 & x_3^2 & x_3 & 1 \\ \vdots & \vdots & \ldots & \vdots & \vdots & \vdots & \vdots \\ x_n^{n-1} & x_n^{n-2} & \ldots & x_n^3 & x_n^2 & x_n & 1 \end{pmatrix} \begin{pmatrix} C_1 \\ C_2 \\ C_3 \\ \vdots \\ C_n \end{pmatrix} = \begin{pmatrix} f(x_1) \\ f(x_2) \\ f(x_3) \\ \vdots \\ f(x_n) \end{pmatrix}$$

We will be looking at solving this system of equation. Along the way, we will pick up some clues on why we did not cover this in the lesson on interpolation, instead of, say, spline interpolation.

First some notations. The set of x-values will be called $\vec{x}$, or `xValues` in Matlab. The corresponding known values of function f will be called $\vec{f}$, or `fValues` in Matlab. Note that $\vec{f}$ is the right hand side vector in the system of equations above. The set of still unknown coefficients $C_1, C_2, \ldots, C_n$ of the polynomial will be called $\vec{C}$, or `Coefs` in Matlab. Note that $\vec{C}$ is the vector of unknowns in the system of equations above. Finally the matrix in the left hand side of the above system of equations, consisting of the various powers of the various x values, is called a "Vandermonde" matrix. (Or rather, it is an horizontally flipped Vandermonde matrix, because of the stupid way Matlab orders the coefficients of polynomials.) This matrix will be called V for Vandermonde, or `VDM` in Matlab. So in vector-matrix notation, we want to solve the system

$$V\vec{C} = \vec{f}$$

To solve this system so will require evaluating vector $\vec{f}$ and matrix V and then using left division to get $\vec{C}$ as usual.

(a) Take $n = 10$ and take the n corresponding x-values equally spaced from 0 to 1. Make sure $\vec{x}$ is a *column* vector. For testing purposes, we will take function f equal to

$$f = 1 - \cos x$$

so you can use that with the given x values to find the f values in vector $\vec{f}$. From the given x values, you can also evaluate the Vandermonde matrix V as shown above. Note that if you do not want to write out the 100 individual coefficients in Matlab one by one without making any typos, you can write it more concisely as

$$V = \left(\vec{x}^{n-1} \, \vec{x}^{n-2} \, \ldots \, \vec{x}^2 \, \vec{x} \, \vec{1} \right)$$

where $\vec{1}$ is a column vector of n ones, `ones(n,1)` in Matlab. The above does assume that the powers of $\vec{x}$ are evaluated *elementwise*. So make sure Matlab does that.

Next solve the system of equations for the set of coefficients $\vec{C}$ using the appropriate Matlab procedures as taught in the lesson. Observe that the found coefficients `Coefs` are quite accurately equal to the Taylor series coefficients. Then evaluate the found polynomial at 100 equally spaced plot points `xPlot` between 0 and 1. Do that using `polyval`. Call the corresponding values of the polynomial `pIntPlot`. Also evaluate the given exact function f at the plot points, call it `fPlot`. Plot `fPlot` as a black broken line, the 10 interpolated f-values `fValues` as black circles, and `pIntPlot` as a blue solid line. Use an appropriate legend, labels, title, etcetera, of course. Use a horizontal axis from 0 to 1 and a vertical axis from -0.1 to 0.5.

The interpolated polynomial will be right on top of the exact curve. So far so good, but even the fourth degree Taylor series would do that; the challenge is very minor here.

(b) Next repeat the above, but with some small errors added to the 10 f-values. The perturbed f-values will be called $\vec{f}_2$ or `fValues2` in Matlab. In particular, in Matlab take

```
fValues2=fValues+0.005*[-0.1 0.5 -1 1 -0.6 0.2 0 0 0 0]';
```

Find the precise relative error in $\vec{f}$ that we introduced here as follows:

$$\varepsilon_f \equiv \frac{|\vec{f}_2 - \vec{f}|}{|\vec{f}|}$$

Here $|\ldots|$ means the length of the vector in between the bars. In Matlab, the length of a vector can be found using the `norm` function (among other methods). You should find that the introduced relative error is only about 1%.

Based on this relative error, and the properties of matrix V, predict what the maximum relative error in the corresponding coefficients $\vec{C}_2$ can be. Use `fprintf` to print it out as

```
Predicted error in the coefficients: *
```

where * means all digits before the decimal point but none behind it. Can these very small errors in the f-values really make a noticeable difference in the coefficients??

To see that, solve for the coefficients $\vec{C}_2$, `Coefs2`, and evaluate their relative error by comparing with $\vec{C}$. Use `fprint` to print it out as

```
Actual error in the coefficients: *
```

Use mark up text to compare this result with the maximum error that you predicted based on the relative error in the f values. Also observe that the found coefficients are now *nowhere close* to the Taylor series coefficients.

Next plot `fPlot` again as a black broken line, the exact 10 `fValues` again as black circles, the perturbed 10 `fValues2` as red circles, and the perturbed polynomial interpolate `pIntPlot2` as a red solid line. Other requirements as before.

You should find that the perturbed polynomial provides a very bad approximation to the exact function; far, far, worse than you would get from linear or spline interpolation of the same perturbed data.

Do note however that the interpolate *does* approximate the 10 exact f-values, the black circles, well. In terms of linear algebra the 10 equations are still satisfied well, even if the solution for the coefficients of the polynomial is no good at all. This again tells you that just because your found solution *satisfies the equations well,* it does not mean that *all* is well. You must look at the condition number for that.

Note that the problem we gave polynomial interpolation here was almost trivial. If you go to still higher degree, and/or functions that vary a bit less trivially over the range, you will get errors like the above, and in fact, far, far worse, even if you *do not* perturb the f-values. This is called the "Runge Phenomenon." The bottom line is to stick to your cubic spline or curve fit from the lesson on interpolation. Polynomial interpolation of high degree is bad news.

(c) One additional problem with the way we did things here is that Vandermonde matrices can easily become very ill-conditioned. As an example, simply change your vector $\vec{x}$ to 10 equally spaced x-values between 4 and 5, rather than between 0 and 1.

Recompute the Vandermonde matrix and comment on whether Matlab can still find meaningful coefficients for interpolating polynomials with these x-values.

Bare solution script

3. Given the matrices

$$A = \begin{pmatrix} -2 & 2 \\ 0 & 1 \\ 14 & 2 \\ 6 & 8 \end{pmatrix} \qquad B = \begin{pmatrix} 3 & 4 \\ 2 & 1 \\ 14 & 16 \\ 1 & 25 \end{pmatrix}$$

find, if it exists (else note that it does not using mark up text),

(a) $-5A + 3B$;
(b) A^{T};
(c) AB, BA, $A^{\mathrm{T}}B$, and BA^{T} (note that the latter two are not equal);
(d) the non-matrix (elementwise) products AB, BA, $A^{\mathrm{T}}B$, and BA^{T};

Bare solution script

. . . . . .

4. Given the matrix

$$A = \begin{pmatrix} -2 & 2 \\ 0 & 1 \\ 14 & 2 \\ 6 & 8 \end{pmatrix}$$

find,

(a) the unit matrix that A can be pre-multiplied by (i.e. as IA). Demonstrate that this does not change A.

(b) the unit matrix that A can be post-multiplied by (i.e. as AI). Demonstrate that this does not change A.

(c) the zero matrix that can be added to A. Demonstrate that this does not change A.

(d) the square zero matrix that A can be pre-multiplied by. Demonstrate that this produces a zero matrix the size of A.

(e) the square zero matrix that A can be post-multiplied by. Demonstrate that this produces a zero matrix the size of A.

(f) a zero row vector that A can be pre-multiplied with. Demonstrate that this produces a zero row vector with the same number of columns as A.

(g) a zero column vector that A can be post-multiplied with. Demonstrate that this produces a zero column vector with the same number of rows as A.

Bare solution script

5. Reconsider the power line between two poles. In addition to sagging down under its own weight, it can vibrate like a string. In this case we will use 8 points along the entire line from pole to pole, and number them from 0 to 7. Each point has an amplitude of vibration of the cable a_i where i is the number of the point. But since the line is attached to the poles at points 0 and 7, the amplitudes of vibration a_0 and a_7 of these points are zero and can be eliminated. That leaves $n = 6$ unknown amplitudes of vibration $a_1, a_2, \ldots, a_6$ for the six interior points. These amplitudes can be shown to satisfy the equations:

$$\begin{aligned}
2a_1 - a_2 &= \frac{\omega^2}{49} a_1 \\
-a_1 + 2a_2 - a_3 &= \frac{\omega^2}{49} a_2 \\
-a_2 + 2a_3 - a_4 &= \frac{\omega^2}{49} a_3 \\
-a_3 + 2a_4 - a_5 &= \frac{\omega^2}{49} a_4 \\
-a_4 + 2a_5 - a_6 &= \frac{\omega^2}{49} a_5 \\
-a_5 + 2a_6 &= \frac{\omega^2}{49} a_6
\end{aligned}$$

where ω is the (scaled) frequency of the vibration.

Formulate this as an eigenvalue problem

$$K\vec{e} = \lambda\vec{e}$$

for an appropriate "stiffness" matrix K, eigenvector $\vec{e}$ equal to $(a_1, a_2, \ldots, a_6)^{\mathrm{T}}$, and eigenvalue λ. What would matrix K be? And what would the eigenvalue be, in terms of the quantities above? To figure that out, note that the eigenvalue problem $K\vec{e} = \lambda\vec{e}$ can be written out as

$$\begin{aligned}
k_{11}e_1 + k_{12}e_2 + k_{13}e_2 + \ldots &= \lambda e_1 \\
k_{21}e_1 + k_{22}e_2 + k_{23}e_2 + \ldots &= \lambda e_2 \\
k_{31}e_1 + k_{32}e_2 + k_{33}e_2 + \ldots &= \lambda e_3 \\
\vdots &= \vdots
\end{aligned}$$

Here the k_{ij} are the components of the matrix K. Compare this system of equations with the previous one, noting that $e_1 = a_1$, $e_2 = a_2$, $e_3 = a_3$, ...

From the comparison of the left hand sides of the two systems of equations above, it should be obvious what λ is equal to. And you can see what the matrix components k_{ij} of K are by comparing right hand sides, noting that row number i is the equation number, and column number j is the number of the eigenvector component.

Matrix K should be symmetric; check that it is. Note further that the eigenvalues better be positive real numbers! Or you would get imaginary frequencies. A symmetric matrix that has all eigenvalues positive is called *positive definite*.

Now let Matlab solve the eigenvalue problem to find the 6 eigenvalues, as an array called `lambda`

From the smallest (first) three eigenvalues in `lambda`, compute and print out the corresponding frequencies `omega1`, `omega2`, and `omega3`, in the generic format

```
Frequency I: *.123
```

where I ranges from 1 to 3. No data numbers in FORMATSTRING allowed.

Note: the frequencies for larger values of I than 3 will be very inaccurate. To get these more accurate, you would need more than 6 interior points along the cable.

Bare solution script

6. This continues the previous exercise

   (a) For the same stiffness matrix as in the previous exercise, now find *both* the eigenvectors, as the columns of a matrix E (`E` in Matlab), and the corresponding eigenvalues, on the main diagonal of a matrix Λ (`Lambda` in Matlab). So eigenvector number j, $\vec{e}_j$ is in column j of matrix E and the corresponding eigenvalue number j, λ_j, is at $\Lambda_{j,j}$. Compute again the frequencies of the first three eigenvalues, this time getting the eigenvalues from matrix Λ.

   (b) The eigenvectors describe the "mode shapes." In other words, for each eigenvalue, the corresponding eigenvector gives the amplitudes of vibration of the 6 interior points when the cable is vibrating at that frequency. Eventually, we would like to plot them.

   Plotting the values of each of the mode shapes against the position of the points along the cable (in percent) will show the mode shape of vibration at that frequency graphically. In other words, it shows you how the power line looks (except for sag) when it is vibrating at that single frequency alone.

   However, it is ugly that the eigenvectors do not include the two end points. While the end points have zero amplitude, still to get a complete picture of the mode shapes, you should include the end points. So take the first 3 eigenvectors out of the matrix of eigenvectors and put them into a matrix `modes` that includes the zero end point values.

   To do so, first create an 8 row by 3 column matrix `modes` of zeros using the appropriate Matlab function. Then replace the middle 6 rows of that zero matrix with the first three eigenvectors that Matlab gave you. To do so use START:END specifications that specify the center-left 6 by 3 part of the matrix `modes`, and the first three eigenvectors of the matrix `E`. Leave out START and END where possible.

   Next take the three individual mode shapes out of `modes`, calling them `mode1Vals`, `mode2Vals`, and `mode3Vals`.

(c) Next check whether `mode1Vals`, `mode2Vals`, and `mode3Vals`, considered as 8-dimensional vectors, are really length 1 and mutually orthogonal as they should be. To check the length, try both the Matlab `norm` function and matrix multiplication and check that they produce the same answer. To check that any two different modes are orthogonal, try both the Matlab `dot` function and matrix multiplication and check that they produce the same answer.

(d) After that, first multiply each of the three mode shapes by 7. This corrects for Matlab scaling the eigenvectors to length one, which is not really desired here. (When you increase the number of points along the power line for greater accuracy, you would instead like the amplitudes to be still about the same.)

Then plot these three mode shapes versus the x-values of the 8 points, where x ranges from 0 to 100% along the length of the power line. (In other words, the array of x-values `xVals` consists of 8 equally spaced percent values from 0 to 100. It should be a column vector, not a row vector, as the modes are.)

Provide appropriate title, labels, and legend. Format the legend entries as

```
legend(['omega1 = ' num2str(omega1,3)],...
       ['omega2 = ' num2str(omega2,3)],...
       ['omega3 = ' num2str(omega3,3)],...
       'location','southeast')
```

where you may have to modify the location depending on the sign of the mode shapes. The above works on Octave. (Note that Matlab considers a combination [STRING1 STRING2] to be simply a bigger combined string.)

Bare solution script

Lesson 6 FOR, IF, WHILE

Before doing the below exercises, you should first read and try to understand the relevant parts of posted lesson 6. If you did not, neither the instructor nor any TA will help you with problems until you do.

Note: there are a lot of homework exercises in this lesson, but most only require relatively minor modifications of earlier exercises, many with solutions that should already have been posted.

1. In Lesson 2, Zeros, exercise 5, you printed out the roots ω_1, ω_2, ω_3, and ω_4, of the following equation:

   $$J_0(\omega) - k\omega J_1(\omega) = 0$$

   where J_0 and J_1 were Bessel functions of the first kind and the given constant k was a nondimensionalized flexibility of the membrane attachment.

   You did the above for both the guessed root method, where you provided `fzero` a guessed approximate root near which to search for the exact one, and using the range method, where you provided `fzero` with a range to search in for the root.

   Repeat printing out the first few roots, using both methods, but now no longer use separate code for each frequency that you print out. Instead for each method, use a `for` loop over counter n, going from 1 to value $n_{\max} = 9$ (instead of 4), to find and print the first 9 roots ω_n. (So you must use one piece of code, not 9, for the 9 frequencies in each method.) Take $k = 2$ again.

   The correct *old* solution should already be posted as file `l2_Zeros_x5.pdf`. Just copy the "SOLUTION:" part into your solution script for the current exercise and modify it as requested.

   Of course, this exercise does require you to write the guessed frequency and the frequency range in terms of the frequency number n. Look what it is for $n = 1$ and then see what you need to add to get it to work for larger values of n. Also, you will need to make a small change in the format of the range method to keep the decimal points aligned.

   Note: This exercise requires a working copy of the function file `DrumFreqEqError.m`. The correct file contents can also be found in the mentioned solution file.

   Bare solution script

2. In Lesson 5, Linear Algebra, exercise 1 (whose correct solution should already be posted) examined the system of equations

   $$\begin{aligned} h_1 &= H \\ h_1 - 2h_2 + h_3 &= \rho/(n-1)^2 \\ h_2 - 2h_3 + h_4 &= \rho/(n-1)^2 \\ h_3 - 2h_4 + h_5 &= \rho/(n-1)^2 \\ \vdots &= \vdots \\ h_{n-2} - 2h_{n-1} + h_n &= \rho/(n-1)^2 \\ 2h_{n-1} - 2h_n &= \rho/(n-1)^2 \end{aligned} \qquad i = \begin{Bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ \vdots \\ n-1 \\ n \end{Bmatrix}$$

for the unknown heights h_1, h_2, ..., h_n of n points along half a sagging power line. Here $H = 1$ was the scaled height of the power line at the pole, and $\rho = 1$ the scaled mass of the power line per unit length. Finally, n, the number of points along the half power line, was taken to be 6 in that exercise.

Copy the solution part from that original exercise into the solution file of the current exercise. However, leave out the "final twist" part of the original exercise, since we are not interested in that anymore.

What you are asked to do now is change the solution so that it works correctly for *any* value of n, not just $n = 6$. That will require, of course, that various numbers (in particular ones close to 6) are re-expressed in terms of n.

But more significantly, it requires that the matrix M, or `Mat`, of the system, and the right hand side vector $\vec{r}$, or `rhv`, of the system are created completely differently. Simply writing the matrix out as before only works for a single value of n, which is not acceptable anymore.

To get the matrix M for general n, first think about how it looks. The matrix has one row for each equation in the system of equations above, and there are n equations, so the matrix has n rows. It also has n columns, because there must be a column for each unknown height h_1, h_2, ..., h_n. So it is an $n \times n$ matrix.

Also the matrix is mostly zeros. So after setting a desired value of n, (`n` in Matlab; give it value 6 initially), initialize M to an $n \times n$ matrix of zeros using the appropriate Matlab function. This ensures that matrix M has the correct size, and it gets all zero coefficients right already. Similarly initialize the vector of right hand sides $\vec{r}$ to a single column of n zeros to make it a column of the right size.

That leaves the task to fix up the nonzero coefficients. To do so, note that the equations with equation numbers (or matrix row numbers) i from 2 to $n-1$ all have a similar structure. Therefore these can be set in an appropriate `for` loop on i. In particular on the "main diagonal" of M, where the unknown number (or matrix column number) j is the same as i, the equations above show that the coefficient is always -2. When the column number j is one more or one less, the coefficient is always 1. And the right hand side is always $\rho/(n-1)^2$.

After you have put most nonzero coefficients in using this `for` loop, the only thing left is to put the nonzero coefficients of the first and last equation in matrix M, and the right hand sides of these equations into $\vec{r}$. These values can simply be set explicitly by writing them out. It is neatest to put in the coefficients of the first equation before the `for` loop and those of the last equation behind it, so that all equations are processed in their normal order.

First run the program with n equal to 6 to check that you get the same results as before. Then change n into 11 without changing anything else and rerun the program. If all is right, you should get a similar sagging power cable as before, but much better looking. Publish and hand in the $n = 11$ version.

Bare solution script

3. In Lesson 5, Linear Algebra, exercise 2 examined a flipped Vandermonde matrix. That is a

matrix V of the general form

$$V = \begin{pmatrix} x_1^{n-1} & x_1^{n-2} & \dots & x_1^3 & x_1^2 & x_1 & 1 \\ x_2^{n-1} & x_2^{n-2} & \dots & x_2^3 & x_2^2 & x_2 & 1 \\ x_3^{n-1} & x_3^{n-2} & \dots & x_3^3 & x_3^2 & x_3 & 1 \\ \vdots & \vdots & \dots & \vdots & \vdots & \vdots & \vdots \\ x_n^{n-1} & x_n^{n-2} & \dots & x_n^3 & x_n^2 & x_n & 1 \end{pmatrix}$$

In this exercise we want to write a *function* (using "New," "Function") `MakeVDM`, with header

```
function VDM = MakeVDM(xValues)
...
```

that, given an input array $\vec{x}$ (`xValues` in Matlab) containing n (`n` in Matlab) x-values $x_1, x_2, \dots x_n$, returns the $n \times n$ flipped Vandermonde matrix V (`VDM` in Matlab) corresponding to these x-values.

Function `MakeVDM` must first find the value of n; this can be done by applying the `size` function on `xValues`. Since `size` returns not simply n, but the array $[n\ 1]$, (or $[1\ n]$ in case a user puts in `xValues` as a row vector), you will want to apply the `max` function on the result of the `size` function to get just n.

Then function `MakeVDM` must initialize the Vandermonde matrix V as an $n \times n$ matrix of ones. This gives the Vandermonde matrix the correct size and as an additional benefit, it gets the final column of ones correct already.

After that, function `MakeVDM` must put in the correct coefficients $V_{i.j}$ (`VDM(i,j)`) in the other columns. It must do that for every row number i from 1 to n, so you will want to use a `for` loop on i with those limits. But for a given row i, `MakeVDM` will still need to put in the coefficients $V_{i,j}$ (`VDM(i,j)`) for all the column numbers j in the row except the last. So you will need a `for` loop on j "nested" inside the one on i.

The most straightforward way to evaluate the coefficients is now to use the formula:

$$V_{i,j} = x_i^{n-j}$$

Check this expression by putting in some i and j values to compare with the general form of the Vandermonde matrix given above.

Note however that if x_i happens to be zero, and $j = n$ (corresponding to the final column), you get 0^0 and that is undefined. So Matlab will crash. The solution is simple: since the final column already has the correct values in it, just leave it as is. Simply make sure to terminate your loop on column number j at $j = n - 1$ instead of at $j = n$.

Be sure to include complete documentation on your function in its comment statements. Then run the script of this exercise: this script already contains a test whether your function gives the right Vandermonde matrix for the same case as in exercise 2 in the Linear Algebra homework. It also tests whether `help MakeVDM` works.

After you get this to work correctly, you are asked to make one additional improvement. Raising x_i-values to powers is not a very efficient thing to do for Matlab. Matlab would much rather multiply than raise to a high power. And the following formula allows you to compute a coefficient $V_{i,j}$ from the one in the next column $j+1$:

$$V_{i,j} = V_{i,j+1} \times x_i$$

Note that this only requires a multiplication. You should check that this formula is also correct according to the general form of the Vandermonde matrix given above.

Of course, you can only compute $V_{i,j}$ this way if the coefficient in the next column is already known. And initially only the last column, of ones, in the matrix is correct. So initially you can only compute the second-last column this way. But if you have the second-last column, then you can compute the third last column, and so on. The bottom line is that you need to compute starting at the second last column and going backwards. So you need to reverse the `for` loop on j so that it starts at $j = n - 1$ and ends at $j = 1$. You may want to refresh your memory about arrays at the end of lesson 1.

(Honesty compels me to admit that there is a still more efficient way to compute this particular type of matrix. You can use elementwise operations to do an entire column at a time. In particular, if $\vec{v}_j$ is a column in V and $\vec{v}_{j+1}$ the next column, then elementwise

$$\vec{v}_j = \vec{v}_{j+1} \times \vec{x}$$

or in Matlab `VDM(:,j)=VDM(:,j+1).*xValues`. Matlab can probably use advanced features of the computer hardware to evaluate this without actually having to use a `for` loop on i at all, and in parallel, i.e. compute more than one coefficient at the same time. At the very least, Matlab would do the `for` loop on i internally, rather than using its code interpreter. However, I want you to practice nested `for` loops, and this way there would not be a loop on i anymore. So do not do this.)

After you have successfully modified your function `MakeVDM`, and the exercise script finds no problem with it anymore, it is time to have some fun with Vandermonde matrices! Use function `MakeVDM` to explore the ill conditioning of Vandermonde matrices further. In particular, *using the command window*, check that for 10 equally spaced x-values from -1 to 1, instead of from 0 to 1, you get a much better condition number than $1.5\,10^7$. Since all it takes is just a shift and rescaling of the original x-values, this improvement in condition number comes essentially for "free." Then, still using the command line, check that if you use 10 x-values from approximately -1 to 1 as given by the "Chebyshev nodes"

```
xValues=cos(linspace(-pi+pi/(2*n),-pi/(2*n),n));
```

you get a condition number that is much better still.

Then add this at the end of your solution script `l6_ForIfWhile_x3.m`. In particular, put in code that for a given value of n, computes the Vandermonde matrix and its condition number for n equally spaced x values from 0 to 1, and also for n equally spaced x values from -1 to 1, and also for n x-values spaced as Chebyshev nodes as above.

Enclose the above code in a `for` loop on n in which n gets the values 10, 20, and 40. So you should be able to see how each of the described three x-value distributions works for these values of n.

Behind the `for` loop, use mark up text to comment on which of the three distributions of points produces the best Vandermonde matrices. Is even the best without problems?

Bare solution script

4. Some mathematician claims that the sum

$$\sum_{n=1}^{n_{\max}} \frac{1}{n}$$

in other words

$$\frac{1}{1}+\frac{1}{2}+\frac{1}{3}+\frac{1}{4}+\frac{1}{5}+\ldots+\frac{1}{n_{\max}}$$

becomes infinite when $n_{\max}$ becomes infinite. Let's check that out.

In your work space, give $n_{\max}$ (`nMax` in Matlab) the initial value 2. Next create a new script (using "New", "Script"). In it, put Matlab code that performs the sum above up to a given value of $n_{\max}$.

Note: call the sum `total` in Matlab. Do *not* use the name `sum` as that name is already used for something else really important in Matlab.

Note: Do *not* give a value to $n_{\max}$ in the script, just assume that some value has already been set, like you did in the workspace.

At the end of the script use an `fprintf` command to print out the number of terms summed and the obtained sum as

```
For nMax = *, the sum is *.123
```

Here `*` means whatever digits of the number before the decimal point `fprintf` wants to print and the .123 means 3 digits behind the decimal point.

Save the script as `Sum1.m` (the `.m` will be inserted by Matlab if you leave it away).

Now in the command window, where you already set `nMax` to 2, run the script. To do so, just type `Sum1` without the `.m` and hit Enter. The script should, of course, say that the sum is 1.5 (the first term is 1 and the second term is 1/2, making 1.5 total). If not, fix the script and try again. Next set `nMax` to 10 in the command window and run the script again; now you should get 2.929. If not, fix so that both `nMax` = 2 and 10 work correctly.

Next edit the solution script `l6_ForIfWhile_x4.m` itself. Put in a `for` loop in which you give `nMax` the successive values [10, 100, 1000, 10000, 100000]. Inside the `for` loop, call the script `Sum1` to print out the sum for the current value of `nMax`.

After studying the values you get for the sum using these five $n_{\max}$ values, use mark up text behind the `for` loop to comment on whether it looks like the sum converges to a definite value when $n_{\max}$ becomes bigger and bigger, or whether it looks like the value of the sum seems to keep getting bigger and bigger without ever settling down on a definite value.

In arguing this, it is helpful to initialize variable `total` to 1 before the `for` loop on $n_{\max}$ (1 is the value of the sum for $n_{\max} = 1$). Then inside the loop you can save the current value of `total` as `totalOld` before you run `Sum1`. After `Sum1` you can then `fprintf` the difference between the current sum `total` and the previous one `totalOld`, as

```
The increase in value is *.123
```

This shows you how much the sum has changed with the increase in `nMax`. And of course, the difference between successive sums must eventually become smaller and smaller, and finally zero, if the sum converges to a finite limit.

Warning: Summing 100,000 terms may be a bit slow on some computers. You may want to wait with that one until everything works OK. Initially just do [10, 100, 1000, 10000] then.

Bare solution script

5. The following important function, the "sine integral",

$$\mathrm{Si}(x) = \int_0^x \frac{\sin\xi}{\xi}\,\mathrm{d}\xi$$

cannot be expressed in terms of simpler functions. You cannot find the needed antiderivative in terms of normal functions, even though the integrand *seems* so simple.

However, the Taylor series of the Si function is easy to find. (Just write the Taylor series for $\sin\xi$, divide by ξ, and integrate.) The result is

$$\mathrm{Si}(x) = \frac{x}{1!\,1} - \frac{x^3}{3!\,3} + \frac{x^5}{5!\,5} - \frac{x^7}{7!\,7} + \dots$$

which can be written as

$$\mathrm{Si}(x) = \sum_{\substack{n=1\\ n\ \text{odd}}}^{\infty} t_n \qquad t_n = (-1)^{(n-1)/2}\,\frac{x^n}{n!\,n}$$

In Matlab (using "New", "Script,"), create a script that, assuming that x (`x` in Matlab) and $n_{\max}$ (`nMax`) have already been set, sums the Si sum above to a last term number $n_{\max}$. Call the sum `total`, as the name `sum` is already used for something else in Matlab. Do not use the seemingly logical name `Si` for the sum either; we want to use that name for something else later. Just stick to `total`.

Hint: You can skip the even values of n in the `for` loop using a START:STEP:END construct in the `for` command for a suitable value of STEP. (STEP is the difference between successive n-values.)

Initialize the sum not to zero, but to the first term $t_1 = x/(1!\,1) = x$, then start the `for` loop at $n = 3$ to add the other terms t_3, t_5, t_7, …, $t_{n_{\max}}$. Doing the first term outside the `for` loop is slightly more efficient. More importantly, it simplifies meeting later follow-up requirements.

At the end of the script, print the obtained sum out as

```
x = *.123 and nMax = * gives Si = *.123456
```

where `*` means as many digits as `fprintf` want to print and .123 means 3 digits behind the decimal point and similar for .123456.

On saving, name the summation script `SiScript.m`.

Next in your command window, set x to 5π. The correct value of $\mathrm{Si}(5\pi)$ is about 1.633965 according to my table book. Set $n_{\max}$ to say 10, call the script `SiScript` by invoking its name (without `.m`), and see whether you get the correct value from the table. If not, increase $n_{\max}$ a bit, like maybe double it, and try again. Find the very smallest value of $n_{\max}$ for which the result agrees with all digits of the table value above.

Now you want to make one more improvement to your script `SiScript.m`. Instead of evaluating the terms t_n straightforwardly in the form shown above, note that for n at least 3, you can compute term t_n from the previous one t_{n-2} using

$$t_n = -\frac{(n-2)x^2}{(n-1)n^2}t_{n-2}$$

Check that this is correct based on the formula for t_n given above. Then use this formula instead of the straightforward one to evaluate t_n for n equal to 3 and higher. (There is no previous term for $n = 1$.) This formula does not overflow like $n!$ and x^n for $x > 1$ will readily do for larger n, nor underflow, as x^n for $x < 1$ will readily do for larger n. It is also much more easy to evaluate for Matlab than factorials and powers at high n. Check that the modified script still works OK.

Next in your solution script `l6_ForIfWhile_x5.m` first make sure to set the value 5π for `x`, and put the value 1.633965 from my table in a variable `Table`. Also set the value you got for `nMax` using the command window. Then invoke your summation script `SiScript` to compute the sum. Compare with the table value so that it looks like

```
x = *.123 and nMax = * gives Si = *.123456
                       Table Si = *.123456
```

Note: Si is actually a quite important function, and Matlab provides this function as `sinint`. (It is apparently within the symbolic logic package; at least Octave says that it is.) Octave does not have the function, and numerical evaluation of `sinint` in Matlab seems to have limited accuracy.

Actually, it is quite difficult to find a really good numerical evaluation of the sine integral. However, it turns out that in 2015 Barnaby Rowe *et al* needed an accurate sine integral for their galaxy simulation software "GalSim", noticed that there was none, and wrote their own; see ArXiv:1407.7676. To find Si(x) for $|x|$ values up to 4, they used an adulterated Taylor series. For larger values of $|x|$, there are round-off issues in the Taylor series (as we will explore in later exercises) and you want to do something else. (We will also explore a simplified version of what they did for large x in a later exercise.)

Bare solution script

. . . . . .

6. In exercise 1 (whose correct solution should already be posted), you printed out the frequencies ω_1 through ω_9 that were roots of the equation:

$$J_0(\omega) - k\omega J_1(\omega) = 0$$

We want to do this again, using the initial guess method, but now stop printing frequencies as soon as the initial guess has become within some tolerable error `tol` of the exact value.

In your solution script, first set `k` again to 2, and ensure that under no circumstance the `for` loop will print out more than 7 frequencies. Then:

(a) In your section "`a)`," first set `tol` to 0.01 and print it out as

```
Requested accuracy: *.1E*
```

Then copy the part from exercise 1 that prints the frequencies using the initial guess method into the current script. Change the formats so that all frequencies in the table are printed with only 3 digits behind the decimal point. Also modify it using `if` and `break` statements so that it prints the message

```
The approximate value is good enough.
```

as soon as the difference of the initial guess from the exact frequency is no more than `tol`, and then stops printing further frequencies. Also use an `if` command to print the error message

```
*** The requested accuracy was not achieved!
```

if the printing of frequencies terminated without ever achieving the requested accuracy `tol`. Use the `error` command, not `fprintf` to print this error message.

(b) After the above works OK, copy your your section "`a)`" into a section "`b)`" (make sure that they are separate sections for publishing purposes). Then change `tol` to 0.001 in section "`b)`". This tolerance will not be achieved, so this time the error message should be printed.

If the script works OK from the command line, publish it and check that the output is acceptable. In particular, Octave users will find that they want a double-percent line before the `if` statement that prints out the error message.

Bare solution script

7. In exercise 2 (whose correct solution should already be posted), you solved a system of equations for the heights of a sagging power line. See the posted solution `l6_ForIfWhile_x2.pdf` for more.

   Copy over the solution part of that exercise into the script of this exercise. Then make the following improvements using appropriate `if` statements:

   - Add a variable `rhoRelerr` at the start that contains the expected relative error in the scaled mass per unit length of the power line. Give it a value of 2%.
   - If the number of points `n` is greater than 11, printing out the complete matrix `Mat` becomes a mess. So change the code so that it only prints out the full matrix and right hand side vector if `n` is no more than 11. Otherwise, the program should only print out the 5×5 top left corner and the 5×5 bottom right corner of the matrix `Mat`, and the first 5 and last 5 coefficients of the right hand side vector `rhv`. Do not use two separate `if` statements to handle the two ways of printing, that is bad programming. Use the proper single compound `if` statement to handle both n up to 11 and n greater than 11.
   - Remove the mark-up text commenting on the condition number of the matrix. This comment might not apply for a different value of `n`, which would produce a different condition number.

     Instead add a compound `if` statement that (1) if the estimated relative error in the heights due to the fact that numbers are only accurate to about 16 digits in Matlab is 100% or more throws the error message

     ```
     *** Bad matrix!  There is no meaningful solution!
     ```

     (using the `error` function instead of `fprintf` so that execution is terminated), or else (2) if this estimated relative error is greater than 0.1% throws the warning (using `fprintf`)

     ```
     ** Estimated solution error in the heights *.1%!
     ```

     After all, certainly you would want to know if the solution process on Matlab introduces significant errors. And large condition numbers are inaccurate; I would hardly trust

the results blindly if the estimated error was over say a percent (or even a tenth of a percent). Note that to print an actual percent in `fprintf`, you must double it in FORMATSTRING.

If neither of the two above problems applies, there is still the matter of (3) what the relative error in the density does. Very pessimistically, you can estimate the relative error in the heights that it causes by multiplying the relative error in the density by the condition number. If this relative error is more than 5%, your lawyer will demand that you put in a warning to that effect. Use the format

```
** Variations in density might theoretically cause an
   error in the heights of up to *%.
   (But in real life the error should be no bigger
   than about *.1%).
```

where the second cited error is the same as the relative error in the density.

Make sure that at most one of the three possible messages can ever appear.

Octave users should put a double percent line in front of the above compound `if` statement to prevent earlier output being lost due to the `error` function aborting execution.

There is no reasonable way to get the matrix to be singular to machine precision, so the grader will just have to look at your source code for that.

- If n is at least 41, do not show the individual computed heights as circles in the graph, do that only for lower values of n.

Check that your program still works as before for $n = 11$. Then change n into 41 for the version you publish and hand in.

Bare solution script

8. Some mathematician claims that the sum

$$\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \dots$$

or in other words

$$\sum_{n=1}^{\infty} t_n \qquad t_n = \frac{1}{n}$$

is infinite. This was already explored in exercise 4, whose solution should already have been posted, using a clever experiment.

However, in this exercise we want to try a more straightforward approach to see whether the mathematician is right: we will try to sum the sum to some reasonably small estimated error ϵ, (`tol` in Matlab), and fail.

So copy your script `Sum1.m` of exercise 4 into a script `Sum1Tol.m` and make appropriate changes into it. First of all, make the script stop summing when the appropriate estimated error in the sum is no more than some tolerable error `tol`. Take the formula for the estimated error to be the one appropriate for a non-alternating series (see the posted lesson). Do not set a value for `tol` in the script; assume that that has already been set before the script is run.

Then behind the summation loop, if the tolerable estimated error was achieved print out the message

```
At n = *, the sum is: *.123
The estimated error is: *.1E*
```

Otherwise print the message

```
*** The tolerated error *.1E* was not achieved,
    even after summing * terms!
    Estimated remaining error: *.1E*
```

Since you do not have infinite time to complete this exercise, do make sure that even if the requested tolerance is never achieved, the script will eventually stop summing after some large number of terms `nMax` rather than continue summing to the end of the semester or to the next power failure. Again do not set a value for `nMax` in the script, it is to be set before running the script.

Now in the command window set `tol` to 0.01 (clearly a very non demanding value) and `nMax` to say 1,000 and run the script. If it does not succeed, and it will not, try 10,000 terms, then 100,000 or whatever you are quite comfortable waiting on. If you get too optimistic, use `Ctrl+c` to stop the summing.

Put the value of `tol` and the largest number of terms `nMax` that you are comfortable waiting for in your solution script `l6_ForIfWhile_x8.m`, along with the `Sum1Tol` call. Behind it, put an `if` statement that prints

```
The mathematician seems to be wrong
```

or

```
The mathematician seems to be right
```

depending on the result from the `Sum1Tol` script.

Note: It is true that the estimated error is not really accurate here, nor is it in general. But accuracy is not the point. The purpose of the estimated error is to prevent you from thinking you have found an accurate value for the sum when you are nowhere close. It does that quite nicely here and in most cases.

Warning: Students who end up with frozen homework programs, or messages that Java or Adobe are misbehaving, should severely reduce the maximum number of terms summed below 100,000, like to 10, then figure out what is wrong. Trying to `publish` 100,000 error messages (or other junk output) is a sure recipe for crashing something.

Bare solution script

9. The same mathematician as in the previous exercise claims that the sum

$$\frac{1}{1} - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \frac{1}{5} - \cdots$$

or in other words

$$\sum_{n=1}^{\infty} t_n \qquad t_n = (-1)^{n-1} \frac{1}{n}$$

is finite, and in fact equal to ln(2). To check this, copy script `Sum1Tol.m` over into `Sum1AltTol.m`. Then in `Sum1AltTol.m` make the following changes: (1) Change the estimated error into the one appropriate for an alternating series like this. (2) Evaluating the sign as $(-1)^{n-1}$ is not efficient, hard to read, and ugly. Therefore, before the `for` loop initialize a variable `sgn` to -1. Then inside the loop, use a statement `sgn=-sgn;`. This ensures that `sgn` will alternate between $+1$ and -1. That produces the $(-1)^{n-1}$ in an efficient way. Then you only need to divide `sgn` by n to get term t_n.

You may want to check in the command window that in this case you can easily achieve an estimated error of 0.01 and that the resulting sum is close to ln(2).

In your solution script `l6_ForIfWhile_x9.m`, set the maximum number of terms to sum to 20,000, then use a `for` loop in which you try values of `tol` equal to 0.01, 0.0001, and `1.E-6`. Inside this `for` loop, print out `tol`, then run `Sum1AltTol`.

Compute the true error in the result (i.e. the absolute difference from $\ln 2$). Print the sum, $\ln 2$, and the true error with `fprintf`, below each other so that it is easy to compare the values.

Then run a couple of sanity checks. In particular: (1) You summed too many terms if the number of terms summed n is greater than one *and* $1/(n-1)$ is no greater than `tol`. (2) You summed too few terms if n is less than $n_{\max}$ *and* $1/n$ exceeds `tol`. Check for each possibility separately using a simple `if` statement, with a compound condition, that prints an error message if the problem exists. Check the proper order of the parts of the compound conditions; you do not want a possibility of dividing by zero. Use function `error` instead of `fprintf` to print the error messages so that further program execution is aborted.

Next, if the requested tolerance was not achieved (which should only happen for the final value of `tol`), print (with `fprintf`)

```
*** Too few terms for this tolerance.
```

and terminate the `for` loop with the appropriate command for terminating a loop.

Otherwise, if the true error is no more than the estimated error, print

```
The mathematician seems to be right:
The sum *.123456 differs from ln(2) by only *.1E*;
less than the estimated error *.1E* in the sum.
```

Otherwise there is something wrong, because (1) this is a monotonous alternating series so the estimated error should *overestimate* the true error, (2) the sum is really ln(2) according to Calculus, and (3) Matlab round-off errors are not significant here. So then print the error message

```
*** Oops, something is wrong and I have no clue.
```

using function `error`.

(Octave students: do not try to use a double percent line in the middle of a `for` loop. It won't work. Instead, if an `error` call is triggered, run `l6_ForIfWhile_x9.m` in the command window to figure out what is wrong and fix it there. Of course, you should always do things that way anyway.)

Warning: Students who end up with frozen homework programs, or messages that Java or Adobe are misbehaving, should severely reduce the maximum number of terms summed below 20,000, then figure out what is wrong. Trying to `publish` 20,000 message lines is a sure recipe for crashing something. Check that your `break` statement is there and correctly implemented.

Bare solution script

10. Continuing exercise 5, if you would actually want to use the sine integral, you would probably want to have it in the form of a function. So create a function Si(x) (function file `Si.m`, capitalized) that evaluates the sine integral using the general procedure of exercise 5.

    However, unlike in the previous exercises, sum until the maximum possible accuracy, rather than to a given tolerance. Not only does that give the best possible result, it also frees the user from having to specify a tolerance.

    Also, instead of using a `break` command to terminate the summation loop when the maximum accuracy has been achieved, use a `return` command to finish the execution of the function completely.

    Also, just hard-code the maximum number of terms to ever sum as 1,000 inside the function. Since the sum should never require anywhere near that number of terms, it is pretty useless to have to specify a maximum number of terms explicitly when using the function. But to be safe against programming and instructor errors, behind the summation loop, do use an `error` command that throws an error

    ```
    *** Si: Summation not converged!
    ```

    if the loop has terminated, implying that the `return` command has never been executed even after 1,000 terms.

    Check in the command window that Si(5π) is still about 1.633965.

    After that, there is one additional refinement you want to implement. If any term t_n in the summation becomes too big, it causes problems. This can happen if x is too big. The problem is that numbers in Matlab have a maximum *relative* error ε which is about 10^{-16}, or more generally equal to `eps`. As a result, any term t_n in the sum has a maximum *absolute* error $\varepsilon|t_n|$. So if a term t_n in the sum has a magnitude of about $1/\varepsilon$ or more, such a term will introduce an error in the sum of order 1. That would completely mess up the true value of Si, which is about $\pm\pi/2$ at large x.

    The bottom line is that if you find a term in the sum whose magnitude is more than, let's say, `0.0005/eps`, you should conclude that the result of the sum will probably have an error of about 0.0005 or so. That is obviously becoming pretty bad. So if that happens, just set the total sum equal to the following *approximate* value (from Wikipedia)

    $$\text{sign}(x)\frac{\pi}{2} - \frac{\cos x}{x}$$

    (where sign(x) is ± 1 depending on the sign of x) valid for large $|x|$ and `return`. This formula will have a maximum error of about 0.0008 in the range of x-values where it is used here. And that then assures that the error in our function Si will never be more than about 0.001 (or less than 0.1%) however large x may be.

    The script `l6_ForIfWhile_x10.m` already has the needed tests in it to test whether your function `Si` is working properly. But first you may want to test it yourself using the command

window. Try evaluating `Si(5*pi)` and compare with the "exact" value 1.633964846102835, using `format long`. You should see that your value is not quite accurate, but pretty good. Also try `Si(100)` with "exact" value 1.56222.

Then run script `l6_ForIfWhile_x10.m` and examine the results. If you get error messages starting with `***`, investigate!

Bare solution script

11. Repeat the previous exercise. However, this time use a `while` loop instead of a `for` loop to do the summation. Call the modified function `SiWhile.m`. It should produce the exact same results as `Si.m`.

    In case the `while` loop was not covered in class, you will have to look up how it is done at the end of lesson 6 yourself.

    Bare solution script

Lesson 7 SYMBOLIC MATHEMATICS

Before doing the below exercises, you should first read and try to understand the relevant parts of posted lesson 7. If you did not, neither the instructor nor any TA will help you with problems until you do.

Note for Octave users: To do symbolic math, you will first need to load the symbolic package as

```
pkg load symbolic
syms initpython
sympref display flat
```

1. Answer using symbolic math:

   (a) Consider the equation for the area of a cylindrical container:

   $$A = 2\pi r^2 + 2\pi r\ell$$

   i. Solve this equation symbolically for the length ℓ in terms of A and r. Also show the result with `pretty`.

   ii. Test out the symbolic solution by verifying that if you take $A = \pi$ and $r = 2/3$, you get $1/12$ exactly. Be sure to use `sym('...')` wherever Matlab would provide a 16 digit floating point number otherwise.

   iii. Convert the symbolic solution into a handle to a standard Matlab anonymous function.

   iv. Check that that function too returns $1/12$ for the example data, but only to about 16 significant digits. Do so by using `fprintf` to print the result out to 30 digits behind the point, as

   ```
   Test length numeric:  1.123456789012345678901234567890
   Test length from vpa: 1.123456789012345678901234567890
   ```

   Here the second line should be the test length to 30 correct digits behind the decimal point, as obtained from the symbolic solution using `vpa`. Make sure this number is printed aligned with the above approximate one, so that you can easily compare digits. To do so, use `fprintf` to print "Test length from vpa:" without a Newline, then show the relevant `vpa` output using `disp`.

   (b) Consider the cubic

   $$x^3 - 5x^2 + 5x + 3$$

   i. Let Matlab find the exact roots of the expanded cubic. Also show the result with `pretty`.

   ii. Let Matlab factor the cubic. You should produce a product of factors here, not a vector. Octave does that by default, but not Matlab. Also show the result with `pretty`. (Note that the remaining quadratic with irrational roots is not factored. In Matlab you could force it using `'factormode'`, but not in Octave.)

iii. Let Matlab expand the factored expression again. This should restore the original cubic. Also show the result with `pretty`.

Bare solution script

2. Answer using symbolic math:

   (a) Let Matlab find the Taylor series of $\ln(1+x)$ up to and including power x^{10}. Also show the result with `pretty`. Note that when $x = 1$, corresponding to $\ln(2)$, you get the first few terms of a sum that you should have fond memories of. So substitute in $x = 1$ to get `ln2Partial`. Next `fprintf` that, converted to a plain Matlab number, as well as $\ln(2)$ in the form

   ```
   ln(2) partial: *.123  ln(2): *.123
   ```

   Note also that when $x = -1$, for $\ln(0)$, you get the first few terms of minus a sum that you should have even more fond memories of. So substitute in $x = -1$ to get `ln0Partial`. Print that out similarly as above in the form

   ```
   ln(0) partial: *.123  ln(0): -Inf
   ```

   (b) Let Matlab symbolically integrate

   $$\int \ln(x)\,\mathrm{d}x$$

   and then differentiate the result again, twice. Check that all is OK, (except for the integration constant, that is).

   (c) Let Matlab symbolically integrate

   $$\int_0^1 \ln(x)\,\mathrm{d}x \qquad \text{and} \qquad \int_{-3}^0 \frac{x}{x-b}\,\mathrm{d}x$$

   (Also show the second integral with `pretty`.) The exact answers are

   $$\int_0^1 \ln(x)\,\mathrm{d}x = -1 \qquad \int_{-3}^0 \frac{x}{x-b}\,\mathrm{d}x = 3 + b\ln\left(\frac{b}{b+3}\right)$$

   (The second solution given by Matlab and Octave is not quite ideal; the two logarithms should have been combined as above for at least real b. As is, the expression becomes complex for positive real b. Matlab, but not Octave, will also blather about the singular case that the pole $x = b$ is in the domain of integration.)

   (d) Show all results of this exercise also with `pretty`. Consider the ratio

   $$\frac{s^3 - 5s^2 + 2s - 5}{s^4 - 4s^3 + 5s^2 - 4s + 4}$$

   Let Matlab factor it; you should get a combination of factors, not a vector. Octave does that by default, but in Matlab you need to fix it with `prod`. Also find the partial fraction expansion of the ratio above. Simplify the partial fraction expansion and check that you get the factored ratio back in Matlab, or the original ratio in Octave.

Bare solution script

Lesson 8 PLOTS

Before doing the below exercises, you should first read and try to understand the relevant parts of posted lesson 8. If you did not, neither the instructor nor any TA will help you with problems until you do.

1. Make the following plots using the `ez...` functions. Make sure to use subexercise headers or each plot will overwrite the previous. (You could also avoid that using the `figure(N)` function to number the figures.)

   (a) Use `ezplot` to plot both $J_0(\omega)$ and $\frac{1}{2}\omega J_1(\omega)$ in the same graph without actually defining either function or any plot points. (Plot one function at a time, using `hold on` to prevent the first curve from being lost. Turn hold off again after the second curve.) The horizontal axis must extend from 0 to 3.5π, with tick marks at multiples of $\pi/2$ and corresponding labels. Use title "Drum Frequency Functions" and legend "$J_0(\omega)$" and "$0.5\,\omega J_1(\omega)$" where ω is `omega` in Matlab. Put the legend in an empty corner of the graph. The intersection points of the curves give the valid frequencies.

   (b) Plot the "lemniscate of Bernoulli," given by the equation

   $$(x^2 + y^2)^2 - (x^2 - y^2) = 0$$

   Use title "Lemniscate of Bernoulli" and hidden axes with sizes from -1 to 1, respectively -0.5 to 0.5 with equal x and y scalings. Note: on Octave, I first have to set equal scaling on the axis, and after that I have to separately restore the messed-up axis sizes. That is a bug. Note: on Octave you may want to increase the resolution from the default 60 points along the axes to 240 to see that the curve crosses itself at the origin.

   (c) Plot the curve

   $$x = t \qquad y = t^2 \qquad z = t^3$$

   from $t = 0$ to 2. Use title "Powers of Time". Use equally scaled axes of the appropriate size.

   (d) Plot the "saddle" function

   $$z = x^2 - y^2$$

   as a mesh surface. Limit both x and y to the interval from -1 to 1, with equally scaled axes. Note: using Octave on my Vista computer, the surfaces take forever to publish (well, 5 minutes or so). Note: using Matlab, you might need a final `view(3)` to get a 3D view.

   (e) Plot the saddle function as above, but now as a solid surface. Make sure that the color shading is interpolated (not bad looking flat or facetted). Note: using Matlab, you might need a final `view(3)` to get a 3D view.

   (f) Plot the saddle function as above, but now as contour lines. Note: on Matlab you may need to increase the resolution from the default 60 points along the axes to 240 to see that the straight lines cross at the origin.

   Bare solution script

2. We have the following measured data on masses of spheres versus their terminal velocities in a liquid:

m:	12.9	16.6	21.5	27.8	35.9	46.4	59.9	77.4
v:	6.7	8.2	9.5	10.2	11.6	12.8	15.8	16.6

   (a) Examine graphically whether the relationship between m and v seems to follow some power law of the form $v = Cm^p$. Use appropriate axis sizes and labels. Comment using mark up text on why this may be a power relationship.

   (b) Try to find approximate values for the constants C and p.

   (c) Compare your found relationship to the measured data in a `loglog` plot. Plot the power relationship from $x = 10$ to 100. Make the vertical axis range from 5 to 20. Add appropriate title, axes labels, and a legend in the southeast corner. Comment using mark up text on how well you think the agreement is.

   Bare solution script

3. A rectangular plate has length 2 and height 1.5. The bottom edge of the plate (the x-axis $y = 0$) is in contact with boiling water, so the temperature T there is 100 degrees Centrigrade (in Tallahassee at least). The other three edges are in contact with ice water, so their temperature T is 0 degrees Centrigrade. Your task is to find and plot the temperature distribution in the interior of the plate under those conditions.

   (a) First create a set of n x-values and m y-values covering the length and height of the plate (`xValues` and `yValues` in Matlab). To get decent accuracy but not use excessive resources, set variable n (`n`) to 29 and m (`m`) to 22 in the solution you hand in. Create a full two-dimensional mesh using these x and y values, and plot these mesh points on the plate as circles. Use appropriate title and labels, x-axis from 0 to 2 and y-axis from 0 to 1.5, scaled equally.

   (b) The temperatures T at the mesh points may be found using function `SimplePoisson`. You should already have obtained that function in the lecture. If not, you can find it in the bare homework templates folder. Or try the link above.

   In using this function, you will need to define an array `forcing` describing the problem "forcing." In doing so note that in this problem, in the interior of the plate the forcing is zero. (In this problem, forcing would correspond to heat radiated away from the interior of the plate, and that can be ignored at 100°C or less.) As already noted, the boundary values are also all zero, except for the 100°C bottom boundary $y = 0$, corresponding to row number $i = 1$ in array `forcing`. (This row is printed out at the *top* of the array if you print out array `forcing`. Sorry about that.) So all the `forcing(i,j)` values are zero except at $i = 1$. There is a slight problem with mesh points $(1, 1)$ and $(1, n)$, as these points are both on the 100°C boundary and on a 0°C boundary. To fix that, give those two special points the average temperature of 50°C.

   Now get the grid temperatures from `SimplePoisson` and plot them as stems. From the plot, check that they seem generally sane. Use appropriate title and labels on all three axes, x-axis from 0 to 2, y-axis from 0 to 1.5, and z-axis from 0 to 100.

   (c) Next plot the temperature distribution as a three-dimensional surface above the plate. Use appropriate title and labels on all three axes, x-axis from 0 to 2, y-axis from 0 to 1.5, and z-axis from 0 to 100.

(d) Next plot the isotherms (lines of constant temperature) 10°C, 20°C, ..., and 90°C within the plate. Note that the top three boundaries together form the 0°C isotherm and the bottom boundary the 100°C isotherm. Use appropriate title and labels on both axes, x-axis from 0 to 2 and y-axis from 0 to 1.5, scaled equally.

If you did this exercise correctly, the temperatures are generally OK, except near the mentioned two special corner points where the temperature jumps from 0°C to 100°C. To make this less noticeable, you could use much bigger values of m and n, but that would make the computation very slow. Especially the surface shading.

"Mesh-stretching" to the rescue. Redefine your mesh point x and y values as follows:

```
xValues=1-cos(linspace(0,pi,n))
yValues=(1-cos(linspace(0,0.5*pi,m)))*1.5
```

This increases the point density near the problem points, as you will see in plotting the mesh. Rerun the program. You should get much better results near the problem corners. Publish and hand in this final version, the one with mesh stretching, *only*.

Bare solution script

4. The so-called "streamfunction" ψ for slow flow of superfluid liquid helium around a circular cylinder, (or of Hele-Shaw flow around a circular cylinder, for that matter), is given in polar coordinates r and θ by

$$\psi = U\left(r - \frac{a^2}{r}\right)\sin(\theta)$$

where U is the incoming flow velocity far from the cylinder and a is the radius of the cylinder. Take both to be one. Now contour lines of the streamfunction are "streamlines" that show you in which direction the fluid flows. The purpose here is to plot them.

Because the flow is outside a cylinder (i.e. a circle when seen in the two-dimensional cross-section), to get a decent plot you will need to use polar coordinates. So define a polar mesh where r takes 201 values from a to $5a$, and θ takes 201 values from 0 to 2π. Find the streamfunction at those points from the expression given above.

But you want to plot the streamlines in Cartesian coordinates. So you will need to convert your polar mesh to Cartesian coordinates. Then plot the contour lines of the streamfunction in terms of these.

Note: to get the streamlines you really want to see, you need to specify their values of ψ in function `contour`. To do so, use the following vector

```
[-5.0001:0.25:-0.0001 0.0001:0.25:5.0001]
```

(This somewhat weird way of writing `[-5:0.25:5]` is designed to be robust against round off errors and such.) Use axis sizes from -3 to 3 in both directions, and make sure you use square axes so that the cylinder looks like a circle and not like an ellipse. (On my version of Octave, `axis('equal')` is buggy.) Add appropriate labels and title.

To make the cylinder look like a solid cylinder, instead of like a circular piece of flow field, see the `patch` function.

Bare solution script