

Chapter 1

Introduction

1.1 computer programming

A computer program is a set of **instructions** that tell the machine to perform a certain set of tasks reliably, in a desired way. The notion of automated machines is an ancient one and we modern computer users are the beneficiaries of many generations of careful thought and development. The Greek mathematician Hero of Alexandria built an automated holy water dispenser, with a coin deposit providing the instruction, way back in the first century of the Common Era. Try to comport yourself accordingly.

Writing programs is problem-solving. Your goal is always to develop and implement a plan to accomplish a certain task using certain tools in the most clear and efficient way possible. The first step in writing a program should always be the creation of an **algorithm**, a well-developed set of instructions for solving a problem in a finite sequence of steps, starting from an initial state and eventually terminating at a final state. The oldest surviving collection of algorithms is found in Euclid of Alexandria's **Elements**, written around 300 BCE. Euclid's *Propositions* in that text describe the process for solving a wide range of problems in geometry and number theory.

Algorithms may be written in many ways: as a prose statement of the problem and how to solve it; as a prose description of an implementation plan; as *psuedocode*; as a flow chart; or in a programming language. Psuedocode is an intermediate step between a prose algorithm and a set of instructions written in a programming language. It is written using the structural conventions of the programming language you are using but intended for human reading, not machine reading.

An example of a prose algorithm (or problem statement) for $a + b = c$ where $a = 2$ and $b = 4$ would be *I wish to sum the values of two variables, a and b, and store the result in a third variable c. The values of a and b are 2 and 4, respectively.* An implementation algorithm might be:

1. define variables a, b, c
2. assign known values to a, b
3. sum a and b and assign result to c

Throughout the following chapters you will be asked to write implementation and psuedocode algorithms. It's a good habit to develop.

The basic goals for writing any computer program are: reliability (the machine always does what you expect it to do when the program runs); usability (the program is easy to use in the intended way); robustness (the program can handle user errors); efficiency (the program uses the smallest possible amount of system resources such as memory space and processor time); and maintainability (the program can be understood by others can be modified as requirements change in the future).

1.2 Matlab

MATLAB[®] is an interactive computation and visualization environment. The name stands for MATrix LABoratory and while the software began as a gateway into a collection of matrix solvers, it has grown into a comprehensive suite of “toolboxes” designed to assist in the solution of many types of problems.

Here, MATLAB is a vehicle that allows us to create and use our own models simply and rapidly, and to present the results with high-quality graphics. MATLAB is a scientific programming tool that frees us from many of the complications of more rigorous computer programming languages. MATLAB's toolboxes free us from writing many common functions for ourselves. Expressions written in MATLAB code look much like the mathematical equations they represent.

Our objectives in Chapter 1 are to become familiar with the MATLAB environment, perform a few calculations, review vectors and matrices, and discuss programming style.

When you start the MATLAB software, graphical user interface (a GUI) with several windows, a directory menu, and a number of pull-down menus is displayed. A brief introduction to the desktop environment is provided here.

1.2.1 the command window

The command window is used to interact directly with the MATLAB engine, by typing commands at the prompt or by copying and pasting a command (or a series of commands) into the window from a text file.

The simplest way to use MATLAB is to type commands at the command window prompt. The prompt looks like this:

```
>>
```

Pressing the return key causes MATLAB to execute whatever statements appear after the prompt. For example, MATLAB can be used as a calculator:

```
>> 2 + 3
```

would return

```
ans =  
      5
```

ans is MATLAB's default variable name. The default is used when the user does not specify a target variable name for the result of an operation. The same example with a defined variable name:

```
>> a = 2 + 3
```

would return

```
a =  
      5
```

The echo to the screen is suppressed by adding a semicolon at the end of the line, such that

```
>> a = 2 + 3;
```

would return

```
>>
```

That is, nothing.

Simple operations can be typed directly into the command window:

```
>> x = [1 2 3 4];           % create a 4 x 1 array named x  
>> y = 6*x + 3;           % calculate corresponding vector named y  
>> figure(1)              % open a figure window  
>> clf                    % clear the figure window  
>> plot(x,y, 'b-')        % plot the line defined by (x,y)  
>> title('a line')       % give the figure a title
```

This sequence of commands defines an array called *x* that contains four numbers, uses *x* to compute a corresponding set of values that are stored in an array called *y*, and then plots the line (*x*, *y*) in a figure window. The commands are executed one by one as they are typed into the command window. The same set of commands could be saved in a text file and executed as MATLAB program (section 1.4). The variables *x* and *y* are arrays, collections of numbers. Arrays of different dimensions, as well as the special characters used to work with them, are discussed in more detail later.

1.2.2 workspace

The MATLAB workspace represents the section of the computer's memory that is allocated to your MATLAB process. Each new MATLAB session you start is a blank slate until you either create a new variable or load an existing set of variables (from data files) into the computer's RAM. When you perform a calculation (for example, inverting a matrix or defining the axes for a figure), you are using the workspace.

Variable names, dimensions, sizes in memory, and classes are listed in the workspace window. MATLAB allows you to use many variable classes, as would other programming languages. In this class, we will use double-precision numeric, logical, and character variables.

Information about workspace contents can also be accessed in the command window using several MATLAB functions:

```
>> who           % lists variable stored in memory
>> whos         % lists variable stored in memory and size
>> size()       % reports the size (rows, columns) of the
                % variable named within the parentheses
>> length()     % reports the longest dimension of a
                % variable (the length of a vector)
>> clear var    % clears var from the workspace
>> clear        % clears the entire workspace (be careful)
```

Some examples using the variables defined above:

```
>> who
```

Your variables are:

```
a v
```

```
>> whos
      Name      Size      Bytes      Class
      a         1x1         8         double array
      v         3x1         24        double array
```

```
Grand total is 4 elements using 32 bytes
```

```
>> size(a)
```

```
ans =
      1      1
```

```
>> size(v)
```

```
ans =
      3      1
```

Some or all of the variables in the workspace in a .mat file either using a pull-down menu or at the command line:

```
>> save spooge.mat      % save everything
>> save spooge.mat x y % save x and y only
```

This is a MATLAB “native” file format. Everything in the workspace is cleared when you quit.

Previously saved workspaces can be loaded via a pull-down menu or at the command line:

```
>> load spooge.mat
```

Data stored in other file formats may be imported via the “Import Wizard” in the Workspace Browser.

1.2.3 current directory

The current directory window shows you both the current directory and its contents. Any file you want to access must be in the current directory or in a directory that is on the search path. Add the directories containing files you create to the MATLAB search path. By default, the files supplied with MATLAB and its toolboxes are included in the search path.

1.2.4 command history

A record of all the commands executed via the command line is stored and displayed in the **command history** window at the lower left. In the command window, you may step backward through that history one command at a time by placing the cursor at the prompt and repeatedly pressing the up arrow key

1.3 help!

1.3.1 read the book

Chapter 2 of *Numerical Methods with Matlab (Recktenwald)* offers a good overview of the MATLAB basics. Chapter 3 introduces the basics of programming using MATLAB.

1.3.2 online help

The MATLAB software offers several avenues for finding help:

1. the help function is accessed by typing `help` and the name of the function you’d like to know about

```
>> help quiver
```

2. if you don’t know the name of a function but know what you want to do, you can try to find the right function using **lookfor**

```
>> lookfor interpolate
```

3. online help is available within an active MATLAB session

```
>> helpdesk
```

or at the Mathworks website: www.mathworks.com. In addition to the standard documentation, the website also features a searchable archive of answers to user questions.

1.4 m-files

Simple calculations can be typed directly into the command window

```
>> x = [1:1:4];           % create a 1 x 4 vector named x
                           % containing the numbers 1,2,3,4
>> y = 6*x + 3;          % calculate a corresponding vector named y
>> figure(1)             % plot the line defined by (x,y)
>> clf
>> plot(x,y, 'b-')
>> title('a line')
```

or they can be saved in as MATLAB program, either a “script” or “function” file. These files, saved with a .m extension, are text files that can be edited with any text editor. Collectively, these files are called m-files.

MATLAB starts its own editor automatically when a file with a .m extension is opened. M-files may be opened using the “file” pull-down menu or using MATLAB’s edit function at the command line.

The % signifies a comment in an m-file. Anything appearing after the % and before the next carriage return will not execute.

M-files are much like programs written in traditional programming languages but you don’t have to compile them yourself, MATLAB does that for you. The commands in an m-file may be executed by typing the name of the script file in the command window or by copying text from the editor and pasting it into the command window. The latter is good for testing new lines in a program as you key it into an M-file. You should, of course, have written the program out on paper first. In this class, you should save all but the simplest calculations as M-files.

1.4.1 scripts

Scripts are collections of commands that run using variables available within the command window. New variables defined when a script is executed become part of the collection of variables stored in memory and accessible via the command window.

The example above could be saved as a file called `straightline .m`

```
x = [1:1:4];
y = 6*x + 3;
figure(1)
clf
plot(x,y, 'b-')
title('a line')
```

1.4.2 functions

Functions are M-files with designated input and output variables. Variables created while a function is running are not saved unless they are identified as output. MATLAB's built-in functions are M-file functions.

To convert the calculation above into a function, we would save the following two lines (plus any comments we wish to make) in the file called `straightline.m`

```
function yf=straightline(x)
yf = 6*x + 3;
```

To run the function in the command window, we would type:

```
>> x = [1:1:4];
>> y=straightline(x);
```

We “pass” the input variable `x` to the function `straightline` and it returns information to the variable `y`. We could then plot the result as indicated above. Note that MATLAB's **plot** is a function as well, albeit a more complicated one than our equation for a straight line. The output variable `y` is sometimes called the “target.”

It is often very useful to write your scripts as functions. Functions simply manipulate whatever you give them and produce a standard result. Once you write a function to perform a specific operation, you use it in many different programs, often saving a great deal of time and effort. Functions should be as short and specific as possible.

When I write functions for my research projects, I start the name with an `f_`: `f_meaningfulname.m`. This shows me that the M-file is a function and the meaningful name helps me to remember what the M-file does.

1.5 variables

1.5.1 arrays

Every variable in MATLAB takes the form of an n -dimensional array of any size. A single-valued variable, such as `a` in section 1.2.1, is a one-dimensional array of size 1 by 1. A multi-valued variable, such as `x` in section 1.2.1, is a 1-dimensional array of size 1 by N , where N is the number of elements in the array. Such numbers are often called scalars and vectors, though we must be careful to note that those terms have particular mathematical meanings. The best way to avoid confusion between the linear algebra (the meaning used here) and the physics meanings of the word vector is to use the term **vector-valued** when we want to indicate the physics meaning (a quantity with magnitude and direction). A matrix is a two-dimensional array of size N by M where N is the number of rows and M is the number of columns in the matrix.

Some examples:

```

x = 1                % is a 1 x 1 array
x = [1 2 3 4]       % is a 1 x 4 array with N=1 rows
                    % and M=4 columns
X = [1 2 3 4
     2 3 4 5
     4 8 9 10]      % is a 3 x 4 array with N=3 rows
                    % and M=4 columns

```

1.5.2 defining variables

One of MATLAB's many virtues is its forgiving nature when it comes to defining variables. In special cases (for example, some loops involving multidimensional arrays), the dimensions of a variable must be defined in advance of its use but in general, variables can be created "on the fly." It is, however, generally good practice to define variables ahead of their use. A new variable is created when it first appears on the left-hand side of an equation. Back in section 1.4, the variable x was defined to be a 1×4 array containing a specified set of values and the variable y was created on the fly, when x was used to compute the values y . All variables on the right-hand side of an equation must have been created earlier in the program or have been loaded into memory before their use.

Several MATLAB functions are used to dimension variables:

```

>> x=zeros(1,4)      % create an array full of zeros with
                    % dimension (rows, columns)
                    [0 0 0 0]
>> x=ones(1,4)       % create an array full of ones with
                    % dimension (rows, columns)
                    [1 1 1 1]
>> x=20*ones(1,4)
                    [20 20 20 20]

>> x=linspace(1,10,4) % create a row vector (1 by N array) of N linearly
                    % spaced values between x1 and x2
                    % ( x1, x2, N)
                    [1 4 7 10]

>> x=logspace(1,4,4) % create a row vector of N logarithmically
                    % spaced values between 10^x1 and 10^x2
                    % (x1, x2, N)
                    [10 100 1000 10000]

```

These functions can be useful for setting up model domains and for placing initial values in variables and for variable arrays. We'll talk more about that later.

1.5.3 local variables

Variables created while a function is running (such as `yf` above) are local to the function. They never appear in the command window's collection of variables. Similarly, variables defined in the command window are not accessible to the function unless they are passed to the function.

1.5.4 global variables

Global variables are recognized by both scripts run in the command window and by functions. Sometimes they are useful (for example, if you want all the functions in your program to know that the acceleration due to gravity in MKS units is 9.81). Most of the time, global variables are a sign of hastily-written code and lead to nothing but confusion as functions you thought had nothing to do with each other become linked by some obscure quantity.

1.5.5 built-in variables

MATLAB has a small number of special variable names. You can find them listed in *Recktenwald* and in the MATLAB help.

1.5.6 naming variables

Choosing variable names is a matter of style. **Matlab variable names must begin with a letter**, which may be followed by any combination of letters, digits, and underscores. Variable names may be any length but MATLAB only uses the first N characters, where N is the number returned by the MATLAB function `namelengthmax`. In general, shorter is better (if for no other reason than it is less to type) but descriptive variable names are useful reminders of what it is that your program does.

In my own programs, I strive to use a consistent style and use certain key letters to help me remember what a variable is supposed to contain. I use lower case for scalar and vector variables and upper case for matrices. Whatever you do, be consistent and write a comment in your script when you define a new variable.

1.5.7 accessing and using information stored in variables

The data stored in variables can be used in many ways. The manner in which the stored information is accessed depends on the dimension of the array stored in the variable and the way in which it is to be used.

If all the elements in an array are to be used, the variable name may be used without reference to specific elements:

```
>> x = [1 2 3 4 5 6 7 8];           % create a 1 x 4 array named x
>> y = 6*x + 3;                     % calculate a corresponding array named y
```

The array `y` is created using all the elements of `x`.

1.5.7.1 index notation

If only certain elements of an array are to be used, index notation is needed. Indexes reference specific locations within a vector, matrix, or other array. Index notation is analogous to tensor notation used in mathematics. Suppose we wish to use the 3rd element of `x`:

```
>> yp = 6*x(3) + 3
```

```
yp =
    21
```

A range of indexes may be specified using colon notation. Subsets of elements within arrays are accessed using the starting element number and the ending element number, separated by a colon. The default behavior is an increment of 1 but you can specify a different increment size by including it between a set of colons. Suppose we wish to use the first through third elements of `x`:

```
>> yp = 6*x(1:3) + 3
```

```
yp =
     9    15    21
```

Suppose we wish to use every second element of `x` beginning at 1 (the odd number indexes 1, 3, and so on):

```
>> yp = 6*x(1:2:7) + 3
```

```
yp =
     9    21    33    45
```

Note that ending the index numbers at 8 will produce the same result.

The parentheses tell MATLAB to access some subset of the elements in the array `x` and the numbers and colon indicate which elements to access. A subset of elements in an array are accessed in a similar way, with indexes for both the rows and columns of interest.

1.5.7.2 some examples

Suppose you wish to work with an N by M array:

```
>> A = [1 2 3 4
        2 3 4 5
        4 8 9 10];
```

```
>> ap=A(1,3)*2           % use the element at row 1, column 3
```

```

ap =
    6

>> Ap=A(1:2, 1)*3      % use elements in rows 1 & 2, column 1

Ap =
    3
    6

>> Aq=A(1:2, 1:3)*1    % use elements in row 1, columns 1 to 3
                        % and row 2, columns 1 through 3

Aq =
    1     2     3
    2     3     4

>> Ar=A(:, 1)*1        % use all rows in column 1

Ar =
    1
    2
    4

```

Specific numbers are used as indexes in the preceding examples. In a program, such index values are said to be “hard coded,” they can’t change as the program runs. There are times when this is desirable and times when it limits functionality of your program. Quite often, programs require index values to change over the course of a calculation. In that case, an index variable is defined and used to access elements of an array.

```

% compute the sums of the columns of A and store
% them in an array

[N M] = size(A);      % measure the number of rows & columns
b=zeros(1,M);        % define array to hold result
for m=1:M             % for each column
    for n=1:N         % sum the values stored in each row
        b(m) = b(m)+ A(n,m);
    end
end
end

```

1.5.7.3 a more complicated example

Suppose we want to find the set of y defined by a polynomial, $y = c_1x^n + c_2x^{n-1} + \dots + c_nx + c_{n+1}$, of order $n=2$:

$$y = c_1x^2 + c_2x + c_3$$

The calculation can be made in several ways. One simple approach requires us to know the order of the polynomial in advance:

```
>> M=20; % number of elements in arrays x, y
>> c1=1; % a polynomial constant
>> c2=2; % a polynomial constant
>> c3=3; % a polynomial constant

>> x = linspace(1,100,M); % array for the independent variable
>> y = zeros(1,M); % the dependent variable

>> y = c1*x.^2 + c2*x + c3; % second order polynomial
```

The constants are stored as separate scalar values and the expression for y is written specifically for an order 2 polynomial. The period in the first term of the polynomial equation tells MATLAB to take the square of each element of the array x . If the expression had been written without the period, it would indicate that the vector is to be squared, an operation that cannot be performed because it violates the rules of linear algebra.

A better approach is to write a script that could be used for a polynomial of any order. If we don't know the order of the polynomial before the script runs, then we don't know how many terms it will contain and thus must write an algorithm that can accommodate any number. The computer must step through however many terms there are. The steps to follow are:

- define domain (values of independent variable)
- define coefficients for polynomial
- use number of coefficients to identify order of polynomial
- allocate space for dependent variable (same number of values as independent variable)
- step through values of independent variable (for loop)
- sum terms in polynomial for each

Here, we will use a **for** loop. The for loop controls the *flow* of the program, causing a group of statements to be repeated a specified number of times. In MATLAB, the opening **for** statement is matched by a closing **end** statement.

```
>> M=20;
>> x=linspace(1,100,M); % the independent variable
>> c=[1 2 3]; % all the constants in one array
```

```

>> N=length(c)-1;                                % order of the polynomial
>> y2 = zeros(1,M);                               % the dependent variable
>>
>> for m=1:M
>>     for n=1:N+1
>>         y2(m) = y2(m) + c(n)*x(m)^(N+1-n);
>>     end
>> end

```

The polynomial calculation uses two *nested for* loops. The first, or outside, loop steps through the independent variable x . The second, or inside, loop steps through the terms of the polynomial for each x . Flow control will be discussed in more detail later. The variables n and m are used as indexes for the arrays c and x , respectively.

A simple figure can be made to compare the results of the two calculations:

```

>> figure(1)
>>                                     % define plot axes
>> axis([min(x) max(x) min(y) max(y)])
>> hold on                             % keep these axes
>> plot(x, y, 'r+')                    % plot the first set of y's
>> plot(x, y2, 'bo')                  % plot the second set of y's
>> title('second order polynomial computed two ways')
>> legend('second-order equation', 'any order for loops')

```

Suppose a subset of (x, y) is of particular interest. Plotting the 10th through 20th (x,y) is accomplished using index notation:

```

>> plot(x(10:20), y(10:20), 'ms')

```

1.6 control of flow

A control flow statement in a computer program results in a choice (*the control*) about which of two or more paths (*the flow*) should be followed. Implementation can be a simple repetition of a set of commands for a predetermined number of times, or a more sophisticated evaluation of the progress of a calculation using relational operators and logic. The latter might be used to stop a calculation when a certain threshold is met. The flow of control in numerical models can become quite involved. As always, the time you spend writing implementation and pseudocode algorithms will be saved when it comes to writing your program. The more complicated the flow becomes, the more likely the programmer is to make a mistake.

The MATLAB control commands include: **if**, **else**, **elseif**, **end**, **for**, **while**, **switch**.

A paired **for** and **end** causes a statement or group of statements to be repeated a fixed number of times. A paired **while** and **end** repeats a group of statements an indefinite number of times, according to a logical condition. A paired **if** and **end** evaluates a logical expression and executes

a group of statements when the expression is *true*. Optional **elseif** and **else** commands allow the execution of alternate groups of statements.

Control statements are accompanied by relational and logical operators that are used to determine when some criterion has been met.

1.7 graphics

MATLAB's graphics capability is very robust. We will use fairly straightforward functions in Geology 326, such as the **plot** function in the examples above. Several graphics functions are described here and your text book offers a short introduction in section 5 of Chapter 2. The online help resources will guide you to more sophisticated routines. The "see also" list at the end of the help file is often quite useful for finding more detailed information or for guiding yourself toward the function you really wanted to find when you typed **help some_function**.

MATLAB plotting functions direct their output to a window that is separate from the command window. In MATLAB this window is referred to as a figure. MATLAB's **figure** function is used to open a new figure window or to make an existing figure window active.

```
>> figure(1)           % open figure window and call it '1'
>> clf                % clear the figure window of any
                       % pre-existing plot
```

MATLAB's **plot** function is used to generate a linear plot. From MATLAB's help function: **plot(x, y)** plots vector *y* versus vector *x*. If *x* or *y* is a matrix, then the vector is plotted versus the rows or columns of the matrix, whichever line up. If *x* is a scalar and *y* is a vector, **length(y)** disconnected points are plotted.

Various line types, plot symbols and colors may be obtained with **plot(x, y, s)** where *s* is a character string made from one element from any or all the following 3 columns:

b	blue	.	point	-	solid
g	green	o	circle	:	dotted
r	red	x	x-mark	-.	dashdot
c	cyan	+	plus	-	dashed
m	magenta	*	star		
y	yellow	s	square		
k	black	d	diamond		
		v	triangle (down)		
		^	triangle (up)		
		<	triangle (left)		
		>	triangle (right)		
		p	pentagram		
		h	hexagram		

Use MATLAB's help function to learn about more complicated uses of the **plot** function. A few functions used in creating simple plots are introduced below.

xlabel and **ylabel** are used to place labels on the axes of a linear plot. Three dimensional graphics may also be made, in which case **zlabel** would be used. The simplest usage of the label functions requires a character string to be passed to the function:

```
>> xlabel('independent variable')
```

The single quotes around *independent variable* tells MATLAB that they are a string of characters.

title is used to place a title at the top of a plot:

```
>> title('a polynomial function')
```

subplot is used to place more than one set of plot axes in a figure window.

```
>> subplot(p, q, r)
```

generates a p by q array of axes and selects the r 'th axis for the subsequent graphics functions. The numbering of the subplots goes from top to bottom and left to right. For example, the upper left set of axes in a 2 by 2 set of axes would be specified:

```
>> subplot(2, 2, 1)
```

The first set of axes would then remain the active set until the next use of the **subplot** function.

axis is used to define the plot axes.

```
>> axis([xmin xmax ymin ymax])
```

generates plot axes with an abscissa that begins at $xmin$ and ends at $xmax$ and an ordinate that begins at $ymin$ and ends at $ymax$. MATLAB will assign its own axes when the plot function is used alone.

hold on fixes the current plot axes.

1.8 toolboxes

Toolboxes are collections of application-specific functions that extend the MATLAB's core capabilities. Toolboxes include function collections for solving partial differential equations, signal and image processing, control system design, optimization, symbolic math, neural networks, and many other others.

1.9 exercises

The answers to exercises that ask for a script, or just a few commands, should include that script or sequence of commands and the result (numeric or graphic).

1. Some practice using MATLAB.

- (a) Create the array A as in section 1.5.7.2. Write a command that would echo the value stored in the second column of the third row of A to the command line. Include the command and its result in your answer to this problem.
- (b) Write a script that uses a **for** loop to compute the sums of the rows of A and store them in an array called c . Use index variables, not explicit numbers, to access locations in the array. Include your implementation algorithm, script and the result produced by the script in your answer to this problem.
- (c) Write a script that uses a **for** loop to calculate the mean of each row in A and store the result in an array b .
- (d) Many basic mathematical functions (trigonometric functions, the exponential and logarithmic functions, the error function, and so on) are built into MATLAB. *Note that all angles are taken to be in radians.* Compute the cosine of all elements in your array A and store the result in a variable called Ac .
- (e) Use the MATLAB function **max** to find the maximum value in the second row of Ac . Use **max** to find the maximum value in all the elements of Ac . You may wish to use MATLAB's **help** function to learn how **max** works.
- (f) Vectors and matrices may be concatenated as long as the dimensions are compatible. For example,

```
>> x=[1 2 3];
>> y=[4 5 6];
>> z=[x y];
```

would yield

```
>> z=[1 2 3 4 5 6];
```

- i. Use colon notation to define a vector $z2$ that is identical to z above.
 - ii. Use colon notation to define a vector that contains the elements $[2\ 4\ 6\ 8\ 10]$.
 - iii. Use **linspace** to define a vector that contains the elements $[2\ 4\ 6\ 8\ 10]$.
 - iv. Use colon notation to define a vector containing values from 10 to 200 in intervals of 10 and from 200 to 1000 in intervals of 100.
- (g) Write a for loop to calculate $n!$. Include your implementation algorithm with your answer.

- (h) Section 1.5.7.3 considered two different ways to compute a dependent variable y for a polynomial with known coefficients, c . Rewrite the script containing two for loops as a function. *What is the function's return value? What are the input variables?*
- (i) Run the function created in exercise 1h with two arbitrary sets of input variables and coefficients. Plot the results in a figure with two **subplots**, one above the other. The plot should include appropriate labels and a title that indicates the order of each polynomial and its coefficients.
2. The following function finds the greatest common denominator of a set of two numbers following the method of Euclid in his treatise *Elements*, Book VII Proposition 2. Write an implementation algorithm that describes how the function works.

```
function R=greatestcommonD(a, b)
    if a == 0
        R=b;
    else
        while b ~= 0
            if a > b
                a = a - b;
            else
                b = b - a;
            end
        end
    end
end
R=a;
```

3. Consider the tensor

$$\sigma_{ij} \quad i, j = x, y, z$$

The subscripts (i,j) are like indexes to an array containing magnitudes of tensor components. For a three-dimensional cartesian coordinate system, the tensor, written in matrix form, is:

$$\begin{bmatrix} \sigma_{xx} & \sigma_{xy} & \sigma_{xz} \\ \sigma_{yx} & \sigma_{yy} & \sigma_{yz} \\ \sigma_{zx} & \sigma_{zy} & \sigma_{zz} \end{bmatrix}$$

In the nomenclature of math and physics, this is a tensor of second rank. Scalars are tensors of zero rank and vectors are tensors of first rank.

If the variable σ represents the full stress tensor in an earth material then the components $i = j$ are the normal stresses and the components $i \neq j$ are the shear stresses. In geodynamics, deviatoric stresses

$$\sigma'_{ij} = \sigma_{ij} - \frac{1}{3}\sigma_{kk}\delta_{ij}$$

are often of more important than the full stresses because they are the part of the full stress tensor associated with deformation of the material. As you can see in the definition above,

in which δ_{ij} is the Kroneker delta, the deviatoric stress is the full stress less the mean normal stress.

Write a MATLAB script to compute the deviatoric stresses for a given stress tensor. Use the variable names `sigma` and `sigmaprime` for σ and σ' . Assign the following magnitudes for σ : $\sigma_{xx} = 0, \sigma_{yy} = 0, \sigma_{zz} = 1$ MPa, $\sigma_{ij} = 0$ for $i \neq j$

As always, write an implementation or pseudocode algorithm *before* you type anything into the m-file editor. Identify what variables you will need. Is there more than one way to perform the calculation? What will you use as the index equivalents of (x, y, z) ?

4. Exponential functions are common in mathematical descriptions of natural processes. In the temporal domain, they arise from the observation that the instantaneous rate at which many processes occur depends on the value of the dependent variable at earlier times. In the spatial domain, exponential functions describe a change in the influence of a constant source with distance from from that source.

Soils are estimated to store more than 2700 Gt of carbon (Lal, 2008) and the rate of carbon exchange between soils, plants, and the atmosphere is relatively rapid. Together, these attributes make soils an important reservoir in the global carbon cycle. Experimental studies show that the rate of soil organic carbon decomposition

$$\frac{dc}{dt} = -kc(t) \quad (1.1)$$

in which c represents the soild organic carbon (SOC) concentration and t represents time, depends on temperature T according to the decay constant

$$k(T) = A \exp\left(\frac{-E}{RT}\right) \quad (1.2)$$

in which E represents an activation energy, R represents the universal gas constant, and A represents a theoretical decay rate at $E = 0$.

Carbon is held in different *pools* within the soil, some more easily decomposed than others. Thus, the total SOC is more correctly represented by a set of equations, one for each pool i

$$\frac{dc_i}{dt} = -k_i c_i(t) \quad (1.3)$$

with the k_i each requiring appropriate coefficients E_i for equation 1.2. Opinions differ widely regarding appropriate values for A of different pools and understanding the fate of soil carbon in a warming world is a topic of considerable research interest.

The exact analytical solution for an equation of the form in (1.1) is:

$$c(t) = c_0 \exp(-kt) \quad (1.4)$$

where c_0 represents the carbon concentration at an initial value of t .

- (a) Write an implementation algorithm for a script that uses equation (1.4) to model a single-pool $c(t)$. Use an initial value $c(t = 0) = 1$ gC per kg soil and a recalcitrant (slow) decay constant $k = 0.018 \text{ a}^{-1}$.
- (b) Write a MATLAB script following your implementation algorithm.
- (c) Modify your script to include a second, labile, decay from the same pool with $k = 0.027 \text{ a}^{-1}$. As before, begin with an implementation algorithm and create a figure to show the results of the two calculations.

Your answer to this exercise should include implementation algorithms, scripts, and figures.

5. Tritium is a radioactive isotope of hydrogen, with 2 neutrons instead of 0 (T or ^3H). Tritium decays into ^3He with a decay constant of 0.0557 a^{-1} . A small amount of Tritium is produced by cosmic ray bombardment of Nitrogen in the atmosphere but most of Earth's Tritium was produced by atmospheric nuclear weapons testing in the late 1950s and early 1960s, mainly in the northern hemisphere. Most of the Tritium entered the ocean by water vapor exchange between the atmosphere and the ocean, and to a smaller extent by direct precipitation on the ocean surface. Much of the remainder went to terrestrial reservoirs such as snow and lakes. The "Cold War" atmospheric fallout Tritium has been used as a tracer to study a variety of environmental problems.

The interstitial water in peat samples collected in 1969 in a bog north of Upper Red Lake, Minnesota were found to have elevated Tritium concentrations (Gorham and Hofstetter, 1971). The maximum concentration, found at about 18 centimeters depth, was 388 tritium units (T.U.). One T.U. is one atom of Tritium per 10^{18} atoms of Hydrogen. Write a MATLAB script to predict when the maximum Tritium concentration in the bog will fall below 10 T.U., assuming no mixing of the interstitial water with other waters and no diffusion of the Tritium.

6. The answers to these questions are due at the beginning of your registered lab time on the date specified at the course website. Suppose this lab is worth a total of 24 points. What fraction of the initial total points will be available if you turn the lab in 20 hours after it is due?

1.10 References

Gorham and Hofstetter, 1971, Penetration of bog peats and lake sediments by tritium from atmospheric fallout, *Ecology*, 52 (5), 898-902.

Lal, R., 2008, Sequestration of atmospheric CO₂ in global carbon pools, *Energy and Environmental Science*, 1(1), 86-100.

