PCS 350 — Computational Methods in Medical Physics

Instructor: Catherine Beauchemin (cbeau@ryerson.ca)

Last modified: September 22, 2010

Contents

1 Introduction

1.1 Learning the Unix commands

In your tutorials, you will be using Linux machines. While you can typically use a Linux machine just like you would a Windows machine (click on the menu/computer icon, etc.), you may also want to learn a few bash terminal commands. To familiarize yourself with the basic bash commands, please follow the online tutorial at http://www.ee.surrey.ac.uk/Teaching/Unix/index.html.

Learning the Unix commands will be particularly useful if you decide to connect to the Ryerson computers from your home computer (using Putty or ssh) to do your work remotely.

1.2 Learning to use Octave

GNU Octave is a high-level language, primarily intended for numerical computations. It is a free, open-source software, and it is mostly compatible with Matlab. It is available for most Unix/Linux platforms, Mac, and Windows. You will find it at http://www.gnu.org/software/octave/download.html.

Octave comes with a series of extra packages that you will find very useful (e.g. GSL, Linear-Algebra, PhysicalConstants). All these packages are distributed separately under the project Octave-Forge. You will find them at http://octave.sourceforge.net.

To familiarize yourself with Octave, please complete the online tutorial at http://www.aims.ac.za/resources/tutorials/octave.

Octave will be used in example codes in class and in tutorials. It has been chosen because it is high-level, interactive, and is the language used by most members of our department. However, if you are familiar with Fortran, C, or python, you are free to complete your assignments in any of these languages instead. Please note that you will receive the same level of assistance with these other languages as you would with Octave, so use them only if you are already familiar with them.

1.2.1 Octave from interactive shell to scripting

In my opinion, one of Octave's big advantage is the option of extending its use from an interactive prompt to a powerful scripting language. To use Octave as a shell, you would type:

```
myprompt% octave -q
octave:1> 2+3
ans =  5
octave:2> a = [1,2,3]
a =
   1   2   3

octave:3> b = [1;2]
b =
   1
   2
octave:4> b'
ans =
   1   2

octave:5> sin(b)
ans =
   0.84147
   0.90930

octave:6> a.^a
ans =
    1    4   27
```

But say you would like to write a script, that you could run unattended, non-graphically, and just come back later to savour the results. After all, isn't that the whole fun of programming? Octave allows you to write executable scripts for that purpose. For example, create a file with content

```
myprompt% which octave
/usr/bin/octave

myprompt% vi hello.oct
#!/usr/bin/octave -q -f

# a sample Octave program
printf("Hello World!\n")
```

```
myprompt% octave hello.oct
Hello World!

myprompt% chmod +x hello.oct
myprompt% ./hello.oct
Hello World!
```

The command `which` lets you know the path of any executable. `vi` is just a command-line editor. You could use anything else, e.g. Notepad, gedit. In the script, the line `#!/usr/bin/octave -q -f` is typical of any executable script. The `#!` has to be the first two characters of an executable, followed by the full path to the programme needed to run that executable. The `-q -f` octave command-line options suppress the printing of the standard octave message at startup, and suppress the reading of any initialization file, respectively. For more info on octave's command-line option, type `octave --help | less` at the prompt, then type `q` to quit.

Finally, you can run a script by typing `octave myscript` or you can first make the file executable with `chmod +x myscript` and then you can run it directly with the command `./myscript`. To verify the permissions of a file (whether it is executable and who can execute it) you can type `ls -l`. For more info, check out some bash documentation on file permissions.

Another important part of scripts is your ability to launch a whole bunch of them with different options. This is especially important when doing stochastic simulations where you need to average several runs for each set of parameters. For this, you'll need to be able to set the value of key parameters from the command-line, when you run your script. Figure 1 shows an example of use of command-line options in an octave script. You would run this script as

```
myprompt% octave -qf cmdline.oct -W 20 -H 12
The area of this rectangle is: 240
```

```
myprompt% chmod +x cmdline.oct
myprompt% ./cmdline.oct -W 20 -H 12
The area of this rectangle is: 240
```

1.2.2 Functions in Octave

Just like any other language, Octave allows you to define functions. One of the powerful features of Octave is its flexibility in the number of input and output arguments. Figure 2 demonstrates usage of these features.

1.3 Adopting good programming practices

Your code should be legible and clear: It should read like English. Writing a programme that compiles and runs is not sufficient. The code has to be written in a way that makes it easy for others to follow and for yourself to maintain over time. Here are a few guidelines I expect you to follow:

- Choose function and variable names which reflect their purpose;
- Follow proper indenting practices;

```
#!/usr/bin/octave -qf

# This code computes the area
# of a W x H rectangle

# Initialize width and height
W = 0; H = 0;

# Parse the command-line args
arg_list = argv();
for i = 1:nargin
   if( strcmp(arg_list{i},"-W") )
      W = str2num( arg_list{++i} );
   elseif( strcmp(arg_list{i},"-H") )
      H = str2num( arg_list{++i} );
   endif;
endfor;

# Compute area
Area = W*H;

# Report output
if( !Area ) # If W or H are zero
   error("Please specify width with -W and height with -H\n");
else
   printf("The area of this rectangle is: %g\n", Area );
end;
```

Figure 1: Example of use and parsing of command-line options in Octave.

```
#!/usr/bin/octave

function [max_elem, maxrow, maxcol] = max_elem(amatrix, rowsubset, colsubset)
   # Don't proceed if amatrix is a vector
   if( min(size(amatrix)) == 1 )
      error( "max_elem: expecting a matrix, got a vector" );
   endif;

   # Set values of unspecified arguments
   if( nargin < 3 )
      colsubset = [1:size(amatrix)(2)];
      if( nargin < 2 )
         rowsubset = [1:size(amatrix)(1)];
      endif;
   endif;
   amatrix = amatrix(rowsubset,colsubset);

   # Find the max element for each column and its row index
   [maxs_in_col, row_of_maxs] = max( amatrix );
   # Find the max element of the whole matrix and its column index
   [max_elem, col_of_max] = max( maxs_in_col );

   # Only computer the requested return values
   if( nargout > 1 )
      maxrow = row_of_maxs(col_of_max);
      if( nargout > 2 )
         maxcol = col_of_max;
      endif;
   endif;
endfunction;

mymatrix = [ 4 2 1; 3 5 9 ];

# Returns the max element
max_elem(mymatrix)

# Returns the max element and the row where it is found
[maxi,maxirow] = max_elem(mymatrix)

# Returns the max element, the row, and the col where it is found
[maxi,maxirow,maxicol] = max_elem(mymatrix)

# Returns the max element in the smaller subset of the matrix's cols
max_elem(mymatrix,[1:2],[1:2])
```

Figure 2: Example of use of function input and output arguments in Octave.

- Comment your code when necessary to make it clear; and
- Maintain a good balance between code that is efficient and code that is clear.

This is a lot harder than it sounds but it is crucial to always strive to follow all these guidelines. Those who do not follow good programming practices simply cannot write good code. Table 1 illustrates bad, good and best programming practices.

1.3.1 Writing efficient Octave code

The following code illustrates 3 different methods of creating a 1-D array of size `len`.

```
#!/usr/bin/octave
len = input( "What size array do you wish to use for the evaluation: " );

clear a;
tic();
for i=1:len
   a(i) = i;
endfor
time1 = toc();
clear a;

tic();
a = [1];
for i=2:len
   a = [a i];
endfor
time2 = toc();
clear a;

tic();
a=zeros( len, 1 );
for i=1:len
   a(i) = i;
endfor
time3 = toc();
clear a;

tic();
a=[1:len];
time4 = toc();

printf( "The time taken for method 1 was %.4f seconds\n", time1 );
printf( "The time taken for method 2 was %.4f seconds\n", time2 );
printf( "The time taken for method 3 was %.4f seconds\n", time3 );
printf( "The time taken for method 4 was %.4f seconds\n", time4 );
```

The functions `tic()` and `tac()` allow you to calculate the time elapsed between the call to `tic()` and the call to `tac()` in your code: a good way to compare code efficiency. The ouput of this code is something like

Bad

```
x = rand(5,6);
foo = -1;
bob = -1;
for a = [1:5]
for b = [1:6]
if( x(a,b) > bob )
foo = a;
bob = x(a,b);
end;
end;
end;
bar = -1;
bob = -1;
for c = [1:5]
for d = [1:6]
if( x(c,d) > bob )
bar = d;
bob = x(c,d);
end;
end;
end;
printf("max element=%f\n", bob);
printf("it is at %d,%d\n",[foo,bar]);
```

Good

```
# Function to find the maximum element of a matrix
# return: value of max element and its row,col position
function elem_and_pos = max_element_and_position( matrix )
   [num_rows, num_cols] = size( matrix );
   maxelem = -1;
   for row = 1:num_rows
      for col = 1:num_cols
         if( matrix(row,col) > maxelem )
            maxelem = matrix( row, col );
            maxpos = [row, col];
         end;
      end;
   end;
   elem_and_pos = [maxelem, maxpos];
endfunction;

# Specify matrix size [#rows, #columns]
msize = [5, 6];
# Create the matrix
rand_matrix = rand( msize );

elem_and_pos = max_element_and_position( rand_matrix );

printf("max element=%g\n", elem_and_pos(1));
printf("it is at %d,%d\n",elem_and_pos(2:3));
```

Best (taking advantage of Octave's matrix manipulation functions)

```
# Function to find the maximum element of a matrix
# return: value of max element and its row,col position
function elem_and_pos = max_element_and_position( matrix );
   # The max element for each column & their row index
   [max_each_col,row_of_maxs] = max( matrix );
   # The max element and its column index
   [max_element,col_of_max] = max(max_each_col);
   elem_and_pos = [ max_element, row_of_maxs(col_of_max), col_of_max ];
endfunction;

# Specify matrix size [#rows, #columns]
msize = [5, 6];
# Create the matrix
rand_matrix = rand( msize );

elem_and_pos = max_element_and_position( rand_matrix );

printf("max element=%g\n", elem_and_pos(1));
printf("it is at %d,%d\n",elem_and_pos(2:3));
```

Table 1: Example code illustrating bad and good programming practices. Can you even tell what the bad code does? While the bad code does exactly what it is supposed to do, it is useless. Nobody can maintain let alone fix code that looks like that. The best code shows how, by taking advantage of Octave's matrix manipulation functions, it is rarely necessary to use `for` loops.

```
myprompt% ./tictoc.oct
What size array do you wish to use for the evaluation: 20000
The time taken for method 1 was 1.6421 seconds
The time taken for method 2 was 2.9851 seconds
The time taken for method 3 was 0.1856 seconds
The time taken for method 4 was 0.0006 seconds
```

So you can see that while all codes are doing the same thing, method 2 is very very costly compared to method 4. In general, you will find that using a `for` loop in Octave is almost always a bad idea: There is almost always a better way to write it.

2 Number representation & rounding error

2.1 Number representation

- In common mathematical notation, numbers are represented as a string of digits which can be of any length, and the location of the ***radix point*** is indicated by placing a "point" character (dot or comma) there (e.g. 12.34)
- In mathematics and computing, a radix point (or radix character) is the symbol used in numerical representations to separate the integer part of the number (left of radix point) from its fractional part (right of radix point).
- If the radix point is omitted then it is implicitly assumed to lie at the right (least significant) end of the string (i.e., is integer).
- In scientific notation, numbers are scaled by a power of 10 so that it lies $\in [1, 10[$ with radix point appearing immediately after the first digit (e.g. 1.234×10^1).

The general form for the representation of a floating point number is:

$$(-1)^\sigma \cdot s \cdot b^e$$

where:

σ is the sign;

s is the significand (or fraction);

b is the base; and

e is the exponent.

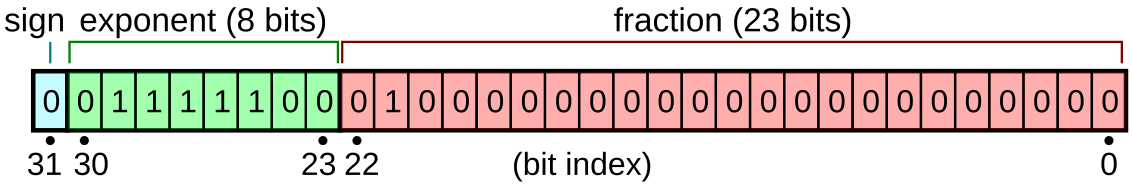
For example

- Base 10: $(-1)^0 \cdot 1234 \cdot 10^{-3} = 1.234$
- Base 2: $(-1)^1 \cdot 1011_2 \cdot 2^{-1} = -101.1_2 = -(2^2 + 2^0 + 2^{-1}) = -5.5$

Quiz: What number (in base 10) is defined by (σ, s, b, e) = (1,9E2AC,16,-4)?

2.1.1 Single-precision representation

In a computer (IEEE 754), single precision (32-bits) numbers (float) are represented as [image taken from Wikipedia]



Where

$$(-1)^{\text{sign}} \times 2^{\text{exponent}-\text{bias}} \times 1.\text{fraction}$$

Sign: 0 (positive) or 1 (negative).

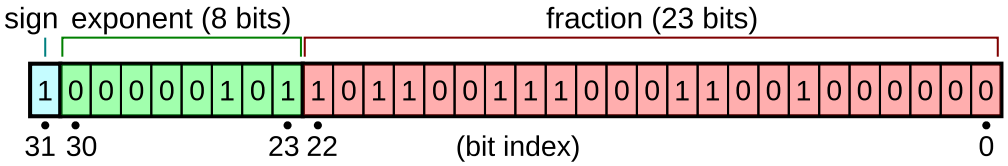
Exponent: $2^8 - 1 = 255$. Bias = 127. Emax = 254-127 = 127. Emin = 1-127 = -126. Mapping: 0 = zero, [1,254] = [-126,127], 255 = Inf or Nan. Bit order: bit $30 = 2^7$, bit $23 = 2^0$.

Fraction: Can represent $2^{23} = 8,388,608$ different 23-digit sequences. Since the first bit is always 1 (011 = 11), there is an assumed (hidden) 24th bit set to 1. Smallest number = 1. Biggest number = $1.111...11_2 \sim 10_2 = 2$. Range = $[1.0_2, 10.0_2[$ or [1,2[. Bit order: bit $22 = 2^{-1}$, bit $0 = 2^{-23}$.

In the example, we have:

- sign: $(-1)^0 = 1$
- exponent: $01111100_2 = 2^2 + 2^3 + 2^4 + 2^5 + 2^6 = 124 \rightarrow 124 - 127 = -3$
- fraction: $+0100...0_2 = 2^{-2} = 0.25 \rightarrow 1 + 0.25 = 1.25$
- Result: $1 \times 2^{-3} \times 1.25 = 0.15625$.

Quiz: What number (base 10) is represented by the following?



2.2 Representation errors

This is just a quick summary. For an indept discussion about floating-point errors, see What every computer scientist should know about floating-point arithmetic. By David Goldberg.

Representation error is the error that can arise as a result of trying to represent a certain number using a finite number of digits. E.g., representing π as 3.1416 is a representation error of

$\pi - 3.1416 = -7.34641020683213... \times 10^{-6}$.

One issue is:

- Humans use the decimal system (base 10: 0,1,2,3,4,5,6,7,8,9)
- Computers use the binary system (base 2: 0,1)

Consequence:

- Numbers which can be represented exactly in one number system, are in general not exactly representable in another system.

Example: One consequence of representation error

Take the very simple example of trying to print the following sequence: 0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2, 0.1, 0.0.

code will never end	code behaves correctly

`x = 1.0;`
`while( x != 0 )`
`  x = x - 0.1`
`end;`
	`x = 10.0;`
	`while( x != 0 )`
	`  x = x - 1;`
	`  x/10`
	`end;`

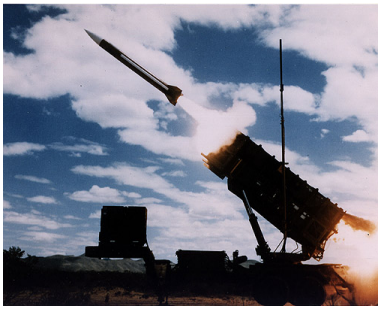
`x =  0.90000`
`x =  0.80000`
`x =  0.70000`
`x =  ...`
`x =  0.20000`
`x =  0.10000`
`x =  1.3878e-16`
`x = -0.100000`
`x =  ...`
	`ans =  0.90000`
	`ans =  0.80000`
	`ans =  0.70000`
	`ans =  ...`
	`ans =  0.20000`
	`ans =  0.10000`
	`ans = 0`

This is because the binary representation 0.1 is:

$$0.1 = 0.0001100110011...{}_2 = 2^{-4} + 2^{-5} + 2^{-8} + 2^{-9} + 2^{-12} + 2^{-13} + ...$$

The particular problem illustrated above could have been avoided by writing good code, namely using `while(x > 0)`. But these sorts of conversion errors can have devastating consequences.

The Patriot missile software problem



During the Gulf War in the early 1990's, Operation Desert Storm used the Patriot missile air defence system to intercept incoming Scuds.

On the night of the 25th of February, 1991, a Patriot missile system operating in Dhahran, Saudi Arabia, failed to track and intercept an incoming Scud. The Iraqi missile impacted into an army barracks, killing 28 U.S. soldiers and injuring another 98. The cause of the missile system failure was traced back to ***a bug*** in Patriot's radar and tracking software.

Because time was measured as the number of tenth-seconds, the value 1/10, which has a non-terminating binary expansion, was chopped at 24 bits after the radix point. The error in precision grows as the time value increases, and the inaccuracy resulting from this is directly proportional to the target's velocity (in this case MACH 5). Read more here.

2.3 Arithmetic round-off errors

One method of computing the difference between two floating-point numbers is to compute the difference exactly and then round it to the nearest floating-point number. This is very expensive if the operands differ greatly in size. For example, $2.15 \times 10^{12} - 1.25 \times 10^{-5}$

$$\begin{aligned} x &= 2.1500000000000000000 \times 10^{12} \\ y &= 0.0000000000000000125 \times 10^{12} \\ x - y &= 2.1499999999999999875 \times 10^{12} = 2.15 \times 10^{12} \end{aligned}$$

The method used by computers is to operate on a fixed number of digits, say $p = 3$, and when the smaller operand is shifted right, digits are simply discarded (as opposed to rounding). Then $2.15 \times 10^{12} - 1.25 \times 10^{-5}$ is

$$\begin{aligned} x &= 2.15 \times 10^{12} \\ y &= 0.00 \times 10^{12} \\ x - y &= 2.15 \times 10^{12} \end{aligned}$$

Here, the answer is exactly the same as if the difference had been computed exactly and then rounded. ***BUT BE CAREFUL...*** If we consider another example: $10.1 - 9.93$. The first method gives us

$$\begin{aligned} x &= 1.010 \times 10^{1} \\ y &= 0.993 \times 10^{1} \\ x - y &= 0.017 \times 10^{1} = 1.70 \times 10^{-1} \end{aligned}$$

which is the correct and exact answer (no rounding-off was even required). Using the second method, however, we get

$$\begin{aligned} x &= 1.01 \times 10^{1} \\ y &= 0.99 \times 10^{1} \\ x - y &= 0.02 \times 10^{1} = 2.00 \times 10^{-1} \end{aligned}$$

This leads to a ***17.6% error*** ($|2.0 - 1.7|/1.7$) and the result from the second method is ***wrong in every digit***!

3 Random numbers

Although each of us probably has an intuitive sense of what we mean by "random number" the formal, mathematical notion can be a bit slippery.

And actually, even our own conception of "what is random" can be quite wrong. E.g. you will feel more confident you can win the 6/49 with a number like `4 12 27 34 38 43` than with `1 2 3 4 5 6`. Yet both sequences are equally likely to occur and both are just as random.

In fact, the notion of random numbers only makes sense when put in the context of an actual sequence of numbers. Here is what a graduate student complaining about issues with the random number generator of his system (RANDU) was told by his computer centre's programming consultant (yes, they use to have that back then):

> "*We guarantee that each number is random individually, but we don't guarantee that more than one of them is random.*"
> [Taken from Numerical Recipes]

3.1 Random numbers generators (RNGs)

A few definitions from Wikipedia:

- ***Randomness*** is a lack of order, purpose, cause, or predictability. A random process is a repeating process whose outcomes follow no describable deterministic pattern, but follow a probability distribution.
- A ***random number generator*** (RNG) is a computational or physical device designed to generate a sequence of numbers or symbols that lack any pattern, i.e. appear random.
- In computing, a ***hardware random number generator*** is an apparatus that generates random numbers from a physical process. Such devices are often based on microscopic phenomena such as thermal noise or the photoelectric effect or other quantum phenomena.
- A ***pseudorandom number generator*** (PRNG) is an algorithm for generating a sequence of numbers that approximates the properties of random numbers.

NOTE: Hardware RNGs can also be built from macroscopic phenomena, such as playing cards, dice, roulette wheels and lottery machines. Even though macroscopic phenomena are deterministic following Newtonian mechanics, real-world systems evolve in ways that cannot be predicted in practice because one would need to know the micro-details of initial conditions and subsequent manipulation or change. BTW, some believe that your mind can affect RNGs. Read more here.

3.2 Pseudorandom numbers generators (PRNGs)

- Consist of an algorithm for generating a sequence of numbers that approximates the properties of random numbers. E.g., RANDU uses the recurrence relation
  $V_{j+1} = (65539V_j) \mod 2^{31}$, where V_0 is odd. Pseudo-random numbers are calculated as $X_j = V_j/2^{31}$.

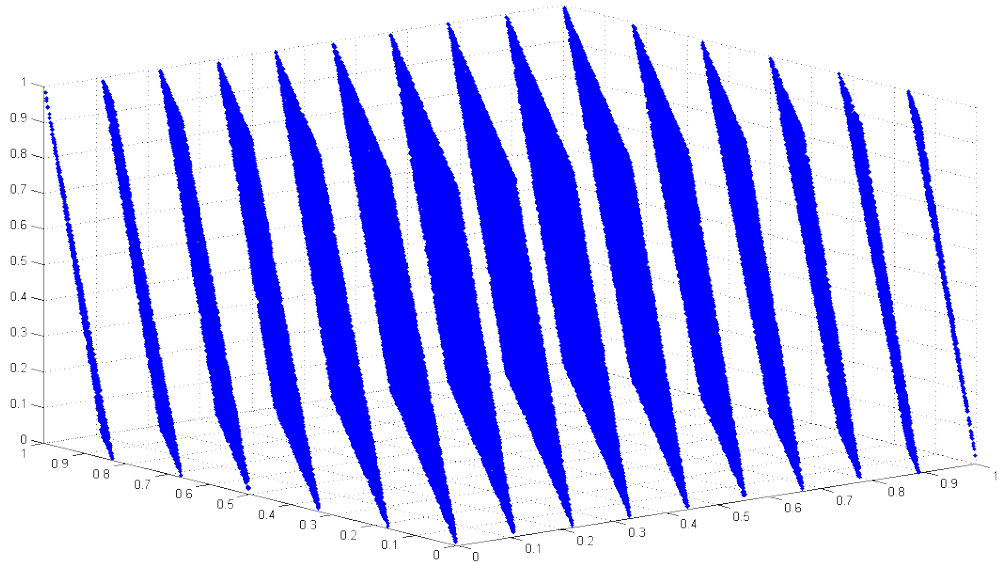


Figure 3: The most infamous PRNG, RANDU. Succesive points fall on a subset of planes because high correlations exist between successive numbers generated by RANDU so using the same random number stream to produce (x, y, z) triplets is ***VERY RISKY***. [Image taken from Wikimedia Commons. Created in MATLAB by generating 100002 values using RANDU and plotting them as 100000 consecutive triples (x, y, z).]

- A PRNG can be started from an arbitrary starting ***state***, using a ***seed*** state. It will always produce the same sequence thereafter, refered to as an ***instance***, when initialized with that state. This is VERY USEFUL when debugging stochastic simulations.
- The maximum length of the sequence before it begins to repeat is called the ***period***.

Much care must go into designing PRNGs to avoid artifacts. The best example of bad PRNG design is certainly the infamous RANDU, for very obvious reasons (see Figure 3). Some common PRNG artifacts include:

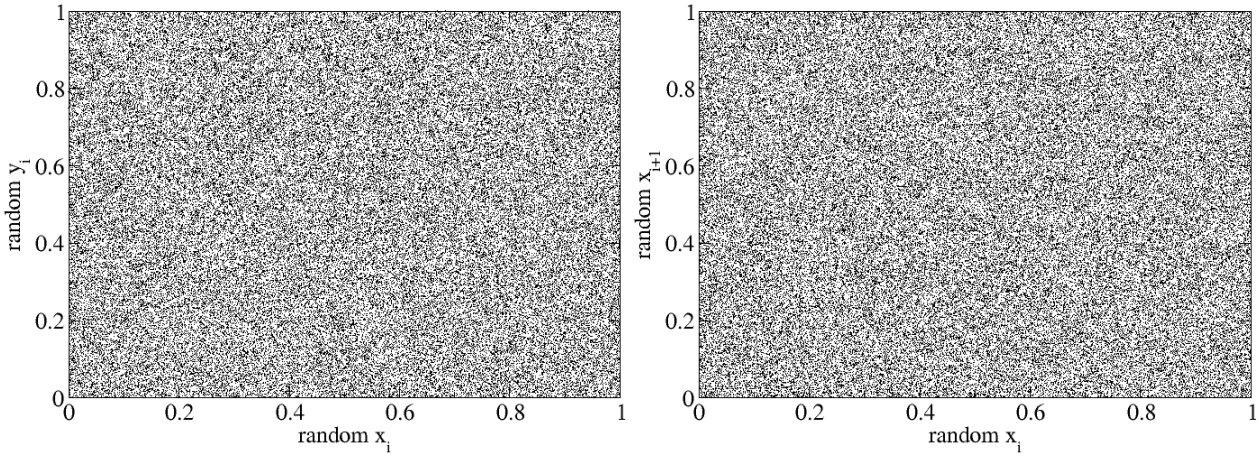
- shorter than expected periods for some seed states; such seed states may be called 'weak' in this context;
- lack of uniformity of distribution (bias); and
- correlation of successive values (poor dimensional distribution of the output sequence).

When using a PRNG, you should always make sure that:

- the period of your PRNG is more than sufficiently long enough for the type of simulations you will be running
- it passes the correlation/spectral tests or otherwise you will need to use as many PRNG instances as you have RN streams in your application. E.g., if you are picking 3-D points at random, you should use 3 streams.

With current computer capabilities, if you run a simulation generating at most 2 calculations per CPU clock cycles running on a 4GHz computer for 100 years, you will require 9.46×10^{18} RNs.

The function `rand` in the most recent Octave version uses the Mersenne Twister with a period of 2^{19937} which is about 4.3×10^{6001}. We can visually verify that our RNG has good spatial distribution and no obvious bias by generating N random numbers, picking successive numbers such that $(x_i, y_i) = (r_{2i-1}, r_{2i})$, and plotting y_i vs x_i (see random y_i vs random x_i, $N = 10^5$). We can also informally look for correlations in the sequence by plotting x_{i+1} vs x_i (see random x_{i+1} vs random x_i, $N = 10^5$).



3.3 Properties of random numbers

In this section, we will use the notation $f(x)$ to designate the probability density function (pdf), $F(x)$ to designate the cumulative distribution function (cdf), and X to designate a random variable produced from $f(x)$.

3.3.1 Probability density function (pdf)

A ***probability density function*** or pdf is a function whose definite integral over a chosen range gives the probability that a random variable take a value in that range. A pdf, $f(x)$, is such that:

- $f(x) \geq 0 \quad \forall \quad x$
- $\int_{-\infty}^{\infty} f(x)\mathrm{d}x = 1$

The actual probability can then be calculated by taking the integral of the function $f(x)$ over a chosen integration interval of the input variable x. For example, the probability of the random variable $X \in [4.3, 7.8]$ is simply

$$\mathrm{P}(4.3 \leq X \leq 7.8) = \int_{4.3}^{7.8} f(x)\,\mathrm{d}x \quad .$$

3.3.2 Cumulative distribution function (cdf)

The ***cumulative distribution function or cdf*** (also called probability distribution function or just distribution function) is a function that gives the probability that a random variable is less than or equal to the independent variable of the function. A cdf, $F(x)$, is such that:

- $0 \leq F(x) \leq 1 \quad \forall \quad x$
- $F(\infty) = 1$

Namely, for every real number x, the cdf of X is given by

$$F(x) = \mathrm{P}(X \leq x) \ ,$$

where $F(x)$ is the cdf, and the right-hand side represents the probability that the random variable X takes on a value less than or equal to x. The probability that X lies in the interval $(a, b]$ is therefore $F(b) - F(a)$ if $a < b$. The cdf, $F(x)$, is related to the pdf, $f(x)$, by the relation

$$F(x) = \int_{-\infty}^{x} f(x') \, \mathrm{d}x'$$

such that

$$F(b) - F(a) = \mathrm{P}(a \leq X \leq b) = \int_{a}^{b} f(x) \, \mathrm{d}x$$

3.3.3 Moments of a random variable

The n^{th} ***moment*** of a random variable with pdf $f(x)$ around a value c is given by

$$\mu_n = \int_{-\infty}^{\infty} (x - c)^n \, f(x) \, \mathrm{d}x \ .$$

A few important concepts/definitions follow

- The moments about zero ($c = 0$) are usually referred to simply as the moments of the function.
- The n^{th} moment (about zero) of a pdf, $f(x)$, is the expected value (or expectation) of X^n. Namely,
$$\mathrm{E}(X^n) = \langle X^n \rangle = \int_{-\infty}^{\infty} x^n \, f(x) \, \mathrm{d}x \ .$$
- The moments about the mean of the pdf, μ, are called central moments. Namely,
$$\mu_k = \Big\langle \big(X - \langle X \rangle \big)^k \Big\rangle = \int_{-\infty}^{\infty} (x - \mu)^k \, f(x) \, \mathrm{d}x \ .$$

The relevant moments are typically

Mean, μ: The 1^{st} moment about zero, such that

$$\mu = \langle X \rangle = \int_{-\infty}^{\infty} x \, f(x) \, \mathrm{d}x \ .$$

Variance, σ^2: The 2^{nd} central moment, such that

$$\sigma^2 = \Big\langle \big(X - \langle X \rangle \big)^2 \Big\rangle = \int_{-\infty}^{\infty} (x - \mu)^2 \, f(x) \, \mathrm{d}x \ .$$

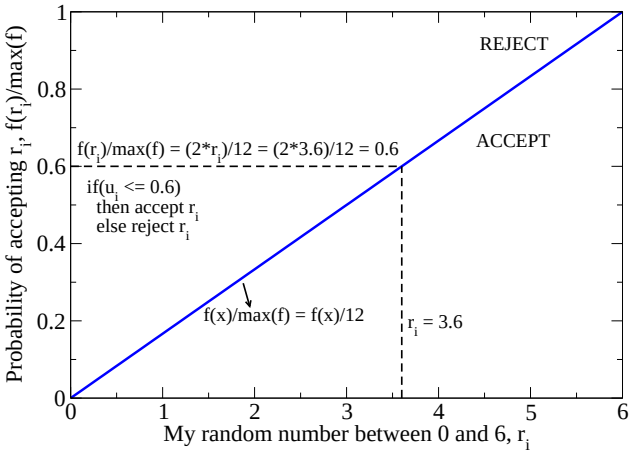
where σ is the standard deviation.

3.4 Generating non-uniform random numbers

3.4.1 Acceptance-rejection method

[Taken from MathWorks]

The idea behind the acceptance-rejection method is to generate uniformly distributed RNs, and reject some such that the remaining ones are distributed following the desired distribution. Say you want RNs distributed such that $f(x) = 2x$ for $x \in [0, 6)$. Here is how you can do it:



Follow these steps

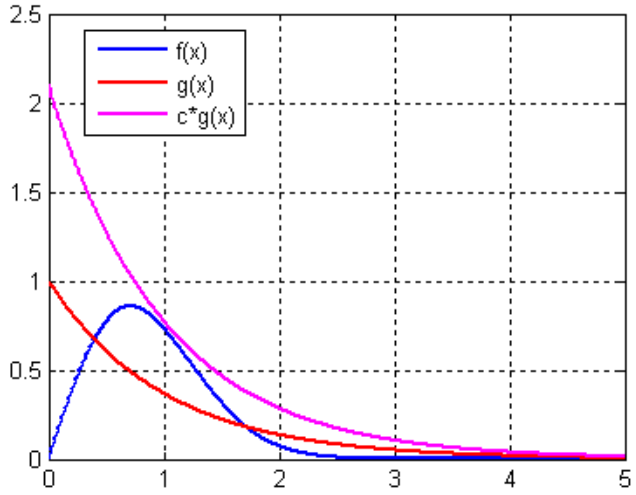
1. Generate a uniform RN, $r_i \in [0, 6)$. This may become one of your RN with distribution $f(x)$. [E.g. $r_1 = 3.6$].
2. Generate a uniform RN, $u_i \in [0, 1)$. This will decide whether or not you accept r_i as your first RN. [E.g $u_1 = 0.512$].
3. If ($u_i \leq \frac{f(r_i)}{c}$) where $c = \max(f)$:

   Then: accept and return r_i. [E.g. since $0.512 \leq \frac{2 \cdot 3.6}{12}$, I accept r_1]

   Else: reject r_i and go back to 1.

But that means that for all $r_i < 3$, you'll reject at least half of the RNs you generate. This sucks! Preferably, you want to generate your r_i RNs so that you don't have to throw out so many.

So more generally, to generate RNs with distribution $f(x)$ for $x \in [x_{\min}, x_{\max})$ with the acceptance-rejection method from RNs r_i with distribution $g(r_i)$, follow these steps:



1. Determine a constant c such that $f(x) \leq c \cdot g(x) \ \ \forall \ \ x \in [x_{\min}, x_{\max})$ for the $g(x)$ you picked smartly.
2. Generate a RN, $r_i \in [x_{\min}, x_{\max})$ following distribution $g(r_i)$.
3. Generate a uniform RN, $u_i \in [0, 1)$
4. If $\left(u_i \leq \frac{f(r_i)}{c \cdot g(r_i)}\right)$

   Then: accept and return r_i.

   Else: reject r_i and go back to 2.

The things to keep in mind

- For efficiency, a "cheap" method is req'd for generating RNs from $g(x)$.

- The scalar c should be small because the expected number of iterations to produce a single RN is c. ***You want to limit the number of rejections***.

3.4.2 Inverse transform method

The idea behind the inverse transform method is to find and apply an algebraic expression to uniformly distributed RNs so as to convert them to produce RNs distributed following the desired distribution. To do this, you'll need:

- The pdf you want your RNs to follow, $f(x)$;
- A uniform random number $u \in [0, 1]$;

Follow these steps

1. Find the cdf, $F(x)$, from the pdf, $f(x)$. Remember that $F(x) = \int f(x)dx$.
2. Set $F(x) = u$ and solve for x in terms of u. Make sure that the solution x lies in the correct range.
3. See if you can simplify $F^{-1}(u)$ given the nature of u.

The things to keep in mind

- Make sure you correctly set your bounds of integration when computing the cdf and that the cdf equals one at the upper bound.
- Sometimes this method fails because $F(x)$ is not an inversible function.

Example: Exponentially distributed RNs using the inverse method

Let's use the inverse transform method to find the function $F^{-1}(u)$ such that we get the pdf $f(x) = \lambda e^{-\lambda x}$ for $x \in [0, \infty]$.

1. Determine the cdf from the pdf

$$\begin{aligned} F(x) &= \int_0^x \lambda\ e^{-\lambda x'}\ \ dx' \\ &= -e^{-\lambda x'}\Big|_{x'=0}^{x} = -e^{-\lambda x} + 1 \\ F(x) &= 1 - e^{-\lambda x} \end{aligned}$$

2. Set $u = F(x)$ and solve for x

$$\begin{aligned} u &= 1 - e^{-\lambda x} \\ 1 - u &= e^{-\lambda x} \\ \ln(1-u) &= -\lambda x \\ x &= -\frac{1}{\lambda}\ \ln(1-u) \end{aligned}$$

3. See if you can simplify the expression. Since u is uniform $\in [0,1]$ then $1-u$ is also uniform $\in [0,1]$ so one might as well define

$$x = -\frac{1}{\lambda}\ \ln(u) \quad \text{for} \quad u \in [0,1]$$

and that's it. You're done.

Quiz: How to produce RNs which follow the triangular distribution $f(x) = \frac{2}{a}\left(1 - \frac{x}{a}\right)$ for $x \in [0,a]$ using the inverse method?

4 Monte Carlo methods

- ***Monte Carlo (MC) methods*** are a class of computational algorithms that rely on repeated random sampling to compute their results.
- MC methods tend to be used when it is infeasible or impossible to compute an accurate result with a deterministic algorithm. E.g. determining all the degrees of freedom that would lead a water molecule to deposit EXACTLY where it did on a snowflake.

4.1 Random walks

- A ***random walk*** is a mathematical formalization of a trajectory that consists of taking successive steps in random directions.
- There are several physical processes, such as Brownian motion, electron transport through metals, or T lymphocytes moving through lymph nodes, in which a particle appears to move randomly.
- A random walk simulation, in the limit of large number of agents/particles taking a large number of steps leads to the model for normal diffusion.
- Representing diffusion (Brownian motion) as a set of discrete particles allows one to capture interesting local dynamics such as particle collisions, binding, interactions, etc.
- Such simulations are needed to answer, for example, how many contacts with other T cells would one T cell make, on average, as it travels a total distance R from its starting location. Katrin Rohlf, in Math, works with these types of models to simulate blood flow and plaque formation.

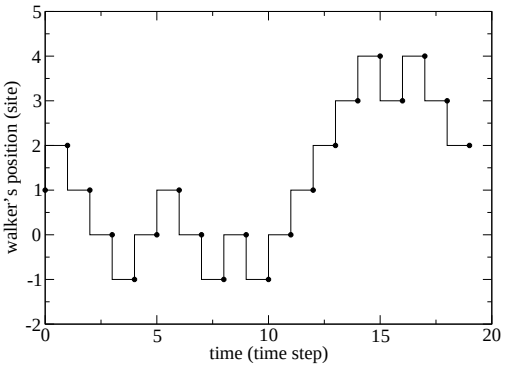
4.1.1 Properties of the simple random walk

Taken from [8]. The traditional unbiased random walk is very simple to implement. A ***random walk*** is defined with a walker starting at position R_0 and moving a step of length s_i at each integer time i, where s_i are independent and identically distributed random variables. More generally,

1-D You simply flip a coin and go either left or right.

2-D You can go up, down, left, or right. You can assign a number to each direction $(0,1,2,3) \mapsto$ (up, down, left, right) and simply sample from an integer RN $\in [0,3]$.

3-D You apply a reasoning similar to that in 2-D except that you now have 6 possible directions.



Here, we will look at the 1-D case and assume each s_i is either $-s$ or $+s$ with equal probability 1/2. A t steps random walk can be expressed as a set of small steps

$$\Delta x_1, \Delta x_2, \Delta x_3, ..., \Delta x_t \ .$$

where Δx_i is either $(-s)$ or $(+s)$. From this definition, we can extract several properties of the random walk. Firstly, let us determine the average position of the walker after a t step random walk. This is given by

$$\begin{aligned}
\langle R_t \rangle &= \langle \Delta x_1 + \Delta x_2 + ... + \Delta x_t \rangle \\
&= \left\langle \sum_i^t \Delta x_i \right\rangle = t \langle \Delta x \rangle \\
&= t \left[\frac{1}{2}(-s) + \frac{1}{2}(+s) \right] \\
\langle R_t \rangle &= 0
\end{aligned}$$

Therefore, the average final position of walkers having taken a t-step random walk is zero. But that is not the full story. A walker having travelled $m \leq t$ steps to the left and a walker having travelled $m \leq t$ steps to the right will have each travelled a distance m, but their average position is zero. So we are not just interested in the average final position, we would also like to know the average radial distance travelled from the starting position. This is called the ***root-mean-squared*** or rms displacement, it is defined as $\sqrt{\langle R^2 \rangle}$ and since $\langle R \rangle = 0$ this is equal to the standard deviation of R or $\sqrt{\left\langle (R - \langle R \rangle)^2 \right\rangle}$. The rms displacement after t steps is

$$\begin{aligned}
\left\langle R_t^2 \right\rangle &= \left\langle (\Delta x_1 + \Delta x_2 + ... + \Delta x_t)^2 \right\rangle \\
&= \left\langle \Delta x_1^2 + \Delta x_2^2 + ... + \Delta x_t^2 + \underbrace{2\Delta x_1 \Delta x_2 + 2\Delta x_1 \Delta x_3 + ... + 2\Delta x_{t-1} \Delta x_t}_{\text{cross terms}} \right\rangle \\
&= \left\langle \sum_{i=1}^{N} \Delta x_i^2 + \underbrace{2 \cdot \sum_{i=1}^{N-1} \Delta x_i \sum_{j=i+1}^{N} \Delta x_j}_{\text{cross terms}} \right\rangle \ .
\end{aligned}$$

If the walk is random, the walker is equally likely to walk in any direction in each step so all cross terms in the above will vanish. For example, if we consider a 2-step walk, we have

$$\begin{aligned}
\left\langle R_2^2 \right\rangle &= \left\langle (\Delta x_1 + \Delta x_2)^2 \right\rangle = \left\langle \Delta x_1^2 + \Delta x_2^2 + 2\Delta x_1 \Delta x_2 \right\rangle = \left\langle \Delta x_1^2 \right\rangle + \left\langle \Delta x_2^2 \right\rangle + \langle 2\Delta x_1 \Delta x_2 \rangle \\
&= \left[\frac{1}{2}(-s)^2 + \frac{1}{2}(+s)^2 \right] + \left[\frac{1}{2}(-s)^2 + \frac{1}{2}(+s)^2 \right] \\
&\qquad +2 \left[\frac{1}{4}(-s) \cdot (-s) + \frac{1}{4}(-s) \cdot (+s) + \frac{1}{4}(+s) \cdot (-s) + \frac{1}{4}(+s) \cdot (+s) \right] \\
&= \left[s^2 \right] + \left[s^2 \right] + \frac{1}{2} \left[s^2 - s^2 - s^2 + s^2 \right] \\
\left\langle R_2^2 \right\rangle &= 2s^2 \ .
\end{aligned}$$

More generally, we have

$$\begin{aligned} \langle R_t^2 \rangle &= \left\langle \sum_{i=1}^{t} \Delta x_i^2 \right\rangle \\ \langle R_t^2 \rangle &= t\, s^2 \\ \sqrt{\langle R_t^2 \rangle} &= \sqrt{t} s \ , \end{aligned}$$

In summary, we have

$$\begin{aligned} \text{mean position} = \quad & \langle R_t \rangle & = 0 \\ \text{variance} = \quad & \langle (R_t - \langle R_t \rangle)^2 \rangle & = ts^2 \\ \text{rms displacement} = \quad & \sqrt{\langle R_t^2 \rangle} & = \sqrt{t} s \ , \end{aligned}$$

where rms is the root-mean-squared displacement. What do these statistics tell us?

mean position = 0: The averaged displacement of several random walks is zero.

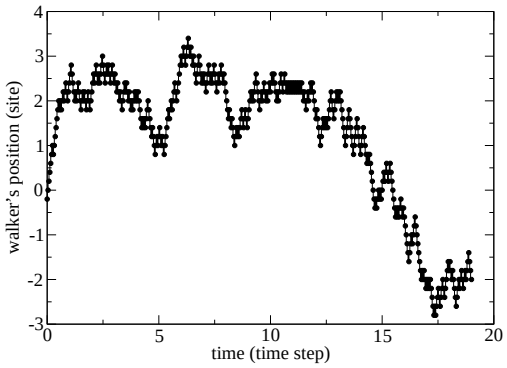
rms displacement = $\sqrt{t}s$: The average displacement for any one random walk consisting of t steps of length s.

So the walker will walk a total path of length ts but will actually only have travelled a distance $\sqrt{t}s$ from its starting point.

Conclusion:

"*Don't drink and walk, unless you have tones of energy to spare*"

4.1.2 Brownian motion



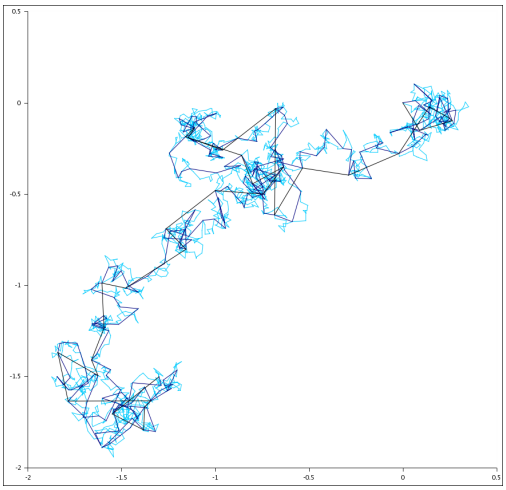
Taken from [8]. Brownian motion corresponds to a random walk in the limit where s, the step size, is infinitesimal. Imagine that double the number of steps are taken per unit time. If a step is taken every 1/2 time unit, the rms displacement of the random walk at time t is now $\sqrt{2t}$ instead of $\sqrt{t}$ since $2t$ steps have been taken. But the rms displacement is a key characteristic of our random walk so it should be conserved as we approach a continuous walk.

In order to keep the rms displacement fixed as we increase the number of steps by a factor k we need to reduce the size of each step such that

$$\text{constant} = rms = \sqrt{t}s = \sqrt{kt}\frac{s}{\sqrt{k}} \ .$$

Thus, as we increase the number of steps by a factor k, we need to change the step size by a factor $1/\sqrt{k}$. In the limit where $k \to \infty$, the random walk becomes continuous: this is called ***Brownian motion***. Let us define the random variable $\lim_{k\to\infty} W_t = B_t$ which has three important properties:

1. For each t, the random variable B_t is normally distributed with mean 0 and variance t.
2. For each $t_1 < t_2$, the normal random variable $B_{t_2} - B_{t_1}$ is independent of the random variable B_{t_1}, and in fact independent of all B_s, $s \in [0, t_1]$.
3. Brownian motion B_t can be represented by continuous paths.



One can reproduce the trajectory of a continous Brownian walk without having to take $k = \infty$ time steps by taking advantage of these properties. Say we want to know the position of our Brownian particle at times $0 = t_0 \leq t_1 \leq t_2 \leq ... \leq t_N$ if it started at $B_0 = 0$ at $t_0 = 0$. Property 2 says that the increment $B_{t_1} - B_{t_0}$ is a normal random variable with mean 0 and variance t_1.

Therefore, we can generate B_{t_1} by choosing from the normal distribution $N(\mu, \sigma^2) = N(0, t_1) = \sqrt{t_1 - t_0}N(0, 1)$, since $\sqrt{s}N(0, 1)$ will convert a normally distributed random variable of variance 1 to one of variance s. More generally, the algorithm will look something like

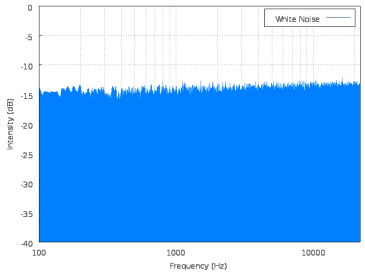
```
B = 0
For each Deltat interval
   B = B + sqrt(Deltat)*Normal(0,1)
End for loop
```

4.1.3 The Wiener process generates Brown noise

Mostly, I just couldn't resist putting in this subsubsection title: way too funny! It also happens to be an important and interesting topic.

It turns out that ***Brownian motion*** is also referred to as the ***Wiener process***. As we have seen above, the Wiener process W_t has independent increments with distribution $W_t - W_s \sim N(0, t-s)$ for $0 \leq s < t$, where, $N(\mu, \sigma^2)$ denotes the normal distribution with expected value μ and variance σ^2. The condition that it has independent increments means that if $0 \leq s_1 \leq t_1 \leq s_2 \leq t2$ then $W_{t1}W_{s1}$ and $W_{t2}W_{s2}$ are independent random variables, and the similar condition holds for n increments. The sequence of random numbers generated by a Wiener process has very important applications in stochastic differential equations, and signal analysis.

Even though noise is a random signal, it can have characteristic statistical properties. Spectral density (power distribution in the frequency spectrum) is such a property, which can be used to distinguish different types of noise. This classification by spectral density is given "color" terminology, with different types named after different colors.



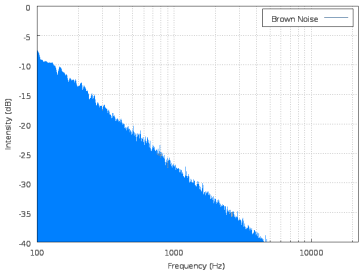
From Wikimedia Commons

You are probably familiar with the term white noise. ***White noise*** is a signal (or process), named by analogy to white light, with a flat frequency spectrum in linear space (equal presence of all frequencies). In practice though, the device you are using to produce the noise has a limited range of frequency so typically, a signal is considered "white" if it has a flat spectrum over a defined frequency band. You can generate white noise by piping the output of

a random number generator to the sound output of your computer using gstreamer, namely

```
myprompt% octave -qf --eval "while(1); fwrite(stdout,rand,'double'); end;" |
  gst-launch fdsrc fd=0 do-timestamp=false blocksize=$((8*256)) !
  audio/x-raw-float, rate=4096, channels=1, endianness=1234, width=64 !
  audioconvert ! alsasink
```

So sequential elements of a white noise signal are completely independent of the preceding elements.



From Wikimedia Commons

Brown noise is a signal with a spectral density proportional to $1/f^2$, meaning it has more energy at lower frequencies. It decreases in power by 6 dB per octave and, when heard, has a "damped" or "soft" quality compared to white noise. Brown noise is not named for a power spectrum that suggests the color brown; rather, the name is a corruption of Brownian motion. It can be generated by an algorithm which simulates Brownian motion or by integrating white noise. Namely,

```
myprompt% octave -qf --eval "b = 0; s=sqrt(1/4096); while(1); b += s*randn;
  fwrite(stdout,b,'double'); end;" |
  gst-launch fdsrc fd=0 do-timestamp=false blocksize=$((8*256)) !
  audio/x-raw-float, rate=4096, channels=1, endianness=1234, width=64 !
  audiowsinclimit mode=1 cutoff=40 !
  audioconvert ! queue max-size-time=1000000000 ! alsasink
```

From Wikipedia. Brown noise is NOT to be confused with the brown note. The brown note is an infrasound frequency that is said to cause humans to lose control of their bowels due to resonance. There is no scientific evidence to support the claim that a "brown note" (transmitted through sound waves in air) exists.

4.1.4 Escape time

Taken from [8]. Many interesting applications of random walks are based on calculations of escape times (or first passage time). Let a, b be positive integers, and consider the first time the random walk (discrete or continuous) starting at 0 reaches the boundary of the interval $[-b, a]$. This is called the ***escape time*** or ***time of first passage***. The expected time for the walker to escape interval $[-b, a]$ is ab. The probability that the escape happens at a rather than $-b$ is $b/(a+b)$.

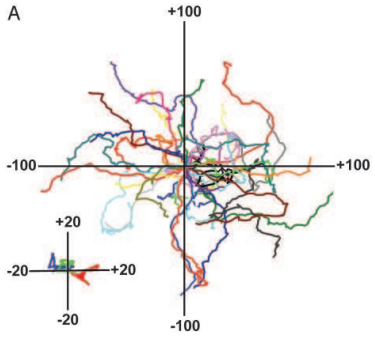
4.1.5 Applications of random walks

Applying additional restrictions to the simple random walks rapidly complicates the mathematics. In fact, this is why processes that behave like restricted random walks are typically modelled as MC processes rather than solved analytically. Two common modification of the simple random walk are:

Self-avoiding RW: Imposes the restriction that the walker cannot cross its own path or that of others. It extends to the notion of self-avoiding polygons, a closed self-avoiding walk on a

lattice. The self-avoiding restriction is common to model the real-life behaviour of chain-like entities such as solvents and polymers, whose physical volume prohibits multiple occupation of the same spatial point. A self-avoiding walk is interesting for simulations because its properties cannot be calculated analytically, thus it is very helpful to understand polymers, e.g. DNA molecules.

Persistent RW: Imposes the restriction that the walker cannot step back. This restriction is common to model animal movement, or many other process where taking a step back is "unatural". For example, the autocatalytic polymerization kinetics of the cytoskeletal actin network provides the basic mechanism for the persistent random walk of a crawling cell.



T cells tracks in lymph nodes (from [6]).

So far we have looked at random walks where the length of the steps is allowed to vary, but directions are very restricted. We are going to relax this and allow steps of fixed length to be taken in ANY direction, requiring only that successive steps be of the same length.

The question here is given the current position of the walker, how does one go about picking the next position such that the walker always takes steps of length r_{rms}. This problem can be thought of as how to distribute points uniformly on a circle (in 2-D) or on a spherical shell (in 3-D).

2-D $x_{i+1} = x_i + r_{\text{rms}} \cos\theta \qquad y_{i+1} = y_i + r_{\text{rms}} \sin\theta$
where θ is a uniform random variable $\in [0, 2\pi]$.

3-D $x_{i+1} = x_i + r_{\text{rms}} \sin\theta \cos\phi \qquad y_{i+1} = y_i + r_{\text{rms}} \sin\theta \sin\phi$
$z_{i+1} = z_i + r_{\text{rms}} \cos\theta$
where ϕ, the angle from the $+x$-axis, is a uniform random variable $\in [0, 2\pi[$, and θ, the angle from the $+z$-axis, is a random variable $\in [0, \pi[$ with $f(\theta) = \frac{1}{2}\sin\theta$.

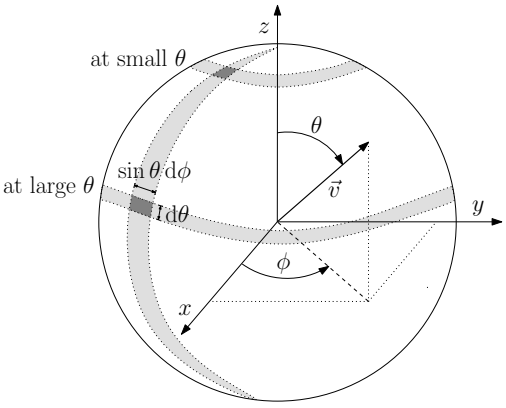


Figure from Beauchemin et al. [**?**]

But where does the pdf $f(\theta) = \frac{1}{2}\sin\theta$ for the 3-D case come from? Why isn't θ, the angle from the $+z$-axis, distributed uniformly $\in [0, \pi[$? This is a simple geometric argument.

A small element on the surface of the sphere defined by $(\mathrm{d}\theta, \mathrm{d}\phi)$ located at a given angle (θ, ϕ) is equal to $\sin\theta\,\mathrm{d}\phi\,\mathrm{d}\theta$. If θ was picked from a uniform distribution, the number of points falling in a $\sin\theta\,\mathrm{d}\phi\,\mathrm{d}\theta$ patch near the poles (where $\sin\theta$ is small) would be the same as that falling in a patch near the equator (where $\sin\theta$ is large). This would lead to a non-uniform distribution with a greater density of points near the poles. To redress this situation, we have to sample θ proportionally compared to the area of these patches, namely pick θ such that

$$f(\theta) \propto \sin\theta \quad , \quad \theta \in [0, \pi[\quad .$$

The formal pdf will have to satisfy

$$\begin{aligned} 1 &= \int_0^\pi f(x)\,\mathrm{d}x = \int_0^\pi C\sin\theta\,\mathrm{d}\theta = C\left[-\cos\theta\right]_0^\pi = C\left[\cos\pi - \cos 0\right] = 2C \quad \rightarrow \quad C = \frac{1}{2} \\ f(\theta) &= \frac{1}{2}\sin\theta\ . \end{aligned}$$

4.2 Radioactive decay

The decay of radioactive particles is the best example of a random process. In fact, some hardware RNGs are actually based on radioactive particle decay (e.g., alpha-radiation from Am 241 source). The particle decay process is described by the ordinary differential equation (ODE)

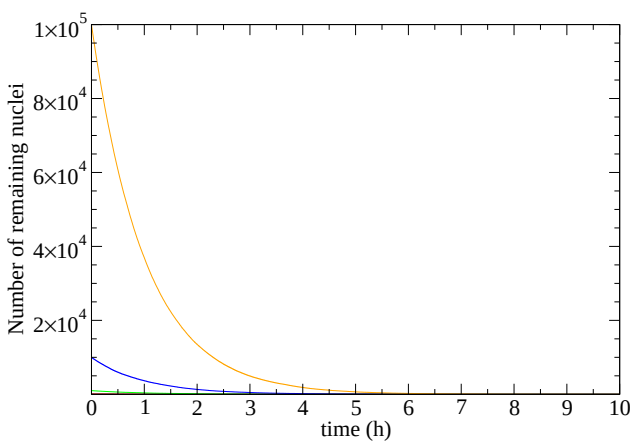
$$\frac{\mathrm{d}N}{\mathrm{d}t} = -\lambda N$$

which describes the decay of particles over time, where N is the number of particles, and λ is the decay rate. The more particles, the more likely one of them will decay in a time step $\mathrm{d}t$.

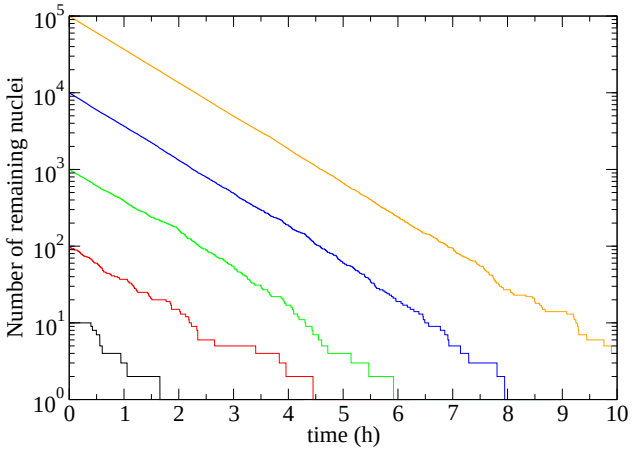
This equation can easily be solved analytically

$$\begin{aligned} \int \frac{\mathrm{d}N}{N} &= \int -\lambda \mathrm{d}t \\ \ln(N) &= \lambda t + C \\ N(t) &= N_0\ \mathrm{e}^{-\lambda t} \end{aligned}$$

but this expression is really only true for N and/or λ large. As N becomes progressively smaller, there will not always be a decay at each step, sometimes, there may be none at all. How do we model such a process at low N values?



Linear scale



Log scale

How do we code it up?

1. For each time step Δt, for each particle p_i, $i \in [1, N]$
2. if($u_i < \lambda\Delta t$) decrement N, increment ΔN
3. End when $N = 0$ (no particles left)

You can try this for different values of N. Notice how the system looks and sounds more and more stochastic as N decreases.

4.3 Monte Carlo integration

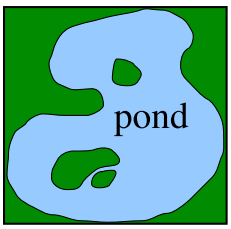
4.3.1 Integration by stone throwing (rejection method)

Have you ever wondered how people use to integrate complicated functions that cannot be integrated analytically before computers were widely used?

How to integrate, OLD SCHOOL

1. Get a nice piece of graph paper and draw your function very very precisely (has to be linear-linear plot).
2. Weigh your piece of paper as precisely as possible
3. Cut VERY carefully along the edge of your function (i.e., cut-out the AUC)
4. Weigh your AUC cut-out
5. Weight of AUC / Weight of full graph = AUC / Area of graph

Yes, you had to be good in arts and crafts to integrate back then! If you wanted to get even more precise results, you would use a device called a planimetre instead of a simple pair of cissors. Once measured, the results of the most common integrals were recorded in reference tables so people could look them up instead of having to do this over and over again. This method of computing an integral would have been used by Sir Isaac Newton and was in use all through the early 80's.



This method is still used today, but with a more modern twist: ***integration by stone throwing*** (which, unfortunately, doesn't involve actual stones). Say you wanted to estimate the area of the pond behind your house (you are VERY lucky if you have a pond behind your house), and you happen to have a lot of pebbles at hand...

1. Draw a shape of known area on the ground (e.g., square or circle) large enough to encircle the pond
2. Start throwing stones into the shape being very careful to distribute them evenly (don't throw them all in one corner).
3. Keep track of how many pebbles you throw, N, and how many of those landed in the pond, s.
4. As $N \to \infty$, you have s/N = area of pond/area of shape

4.3.2 Integration by mean value

From [5]. The standard MC technique for integration is based on the mean-value theorem, namely

$$I = \int_a^b f(x)\,\mathrm{d}x = (b-a)\,\langle f \rangle$$

This is obvious if you think of integrals as areas: the value of the integral of some function $f(x)$ between a and b equals the length of the interval $(b-a)$ times the average value of the function over that interval, $\langle f \rangle$. The integration algorithm uses MC techniques to evaluate the mean of f.

With a sequence x_i of N uniform random numbers $\in [a,b]$, we want to determine the sample mean by sampling the function $f(x)$ at these points:

$$\langle f \rangle \sim \frac{1}{N} \sum_{i=1}^{N} f(x_i)$$

This gives the very simple integration rule

$$I = \int_a^b f(x)\,\mathrm{d}x = \frac{(b-a)}{N} \sum_{i=1}^{N} f(x_i)$$

as $N \to \infty$ or $(b-a)$ gets infinitely small, this scheme approaches the correct answer.

How to code it up:

```
define function f   (of which you seek the integral)
I = 0
For i in 1 to N do
   I = I + f( uniform RN in [a,b) )
End
I = I * (b-a)/N
```

The uncertainty in the value obtained for the integral I after N samples of $f(x)$ is measured by the standard deviation σ_I. The standard deviation of the integrand f in the sampling is an intrinsic property of the function $f(x)$, that is something we cannot alter by taking more samples. For example, if the $f(x_i)$ samples follow a normal distribution, σ_I and σ_f are related by

$$\sigma_I \sim \frac{\sigma_f}{\sqrt{N}} = \sqrt{\frac{\langle f^2 \rangle - \langle f \rangle^2}{N}}$$

In other words, even for very large N, if $f(x)$ is far from smooth over $[a,b]$, this simplistic method will not yield very good results.

4.3.3 Integration by mean value with variance reduction

Wouldn't it be nice if we could somehow remove the variance of $f(x)$ to make it smoother. One good method is the ***variance reduction*** or ***subtraction*** method, in which we construct a flatter function on which to apply the MC integration by mean value. Suppose we construct a function $g(x)$ with the following properties on $[a,b]$:

$$|f(x) - g(x)| \le \epsilon \qquad \text{and we know } \int_a^b g(x)\,\mathrm{d}x = J \ .$$

We now evaluate the integral of $f(x) - g(x)$ and add J to the result to obtain the desired integral

$$I = \int_a^b f(x)\,\mathrm{d}x = J + \int_a^b [f(x) - g(x)]\,\mathrm{d}x = J + \frac{(b-a)}{N} \sum_{i=1}^{N} [f(x_i) - g(x_i)] \ .$$

If we can indeed find a simple $g(x)$ that makes the variance of $f(x) - g(x)$ less than that of $f(x)$, and that we can integrate analytically, we obtain more accurate anwers in less time.

How to code it up:

```
define function f          (of which you seek the integral)
define function g          (which you will subtract from f)
J = the analytical value of the integral of g over [a,b]
I = 0
For i in 1 to N do
   r = uniform RN in [a,b)
   I = I + f(r) - g(r)
End
I = I * (b-a)/N + J
```

4.3.4 Integration by mean value with importance sampling

We can smooth out $f(x)$ by subtracting another function from it, but we can also smooth it out by dividing it by another function. This method is called ***importance sampling*** because it lets us sample the integrand in the most important regions. It takes the form

$$I = \int_a^b f(x)\,\mathrm{d}x = \int_a^b \frac{f(x)}{w(x)} w(x)\,\mathrm{d}x$$

To be useful, the ***weighting function***, $w(x)$, must be such that $f(x)/w(x)$ is smooth. Then, with a small change of variable

$$\mathrm{d}y = w(x)\,\mathrm{d}x \;\rightarrow\; y(x) = \int_a^x w(x')\,\mathrm{d}x' \;\rightarrow\; y(b) = \int_a^b w(x)\,\mathrm{d}x = 1 \;\rightarrow\; y(a) = 0,\; y(b) = 1\ ,$$

we can transform our original integral into the simpler, smoother

$$I = \int_a^b f(x)\,\mathrm{d}x = \int_a^b \frac{f(x)}{w(x)} w(x)\,\mathrm{d}x = \int_0^1 \frac{f(x(y))}{w(x(y))}\,\mathrm{d}y = \frac{1}{N}\sum_{i=1}^{N} \frac{f(x(y_i))}{w(x(y_i))}\ .$$

where $x(y_i) = F^{-1}(y)$, and $F(x)$ is such that

$$y(x) = F(x) = \int_a^x w(x')\,\mathrm{d}x'\ .$$

A good choice of $w(x)$ makes $f(x)/w(x)$ a rather constant function and consequently easier to integrate. You can think of $w(x)$ as increasing the density of points sampled where $f(x)$ is large because $\mathrm{d}y \mapsto w(x)\,\mathrm{d}x$. Alternatively, you can think of $w(x)$ as simply reducing $f(x)$ so that it attenuates the "important areas" in $f(x)$.

Basically, what we are doing is evaluating the integral $f(x)/w(x)$ with x distibuted following the pdf $w(x)$ fo $x \in [a, b]$. As we saw when we studied the inversion method to generate RNs of a given distribution, the requirements on $w(x)$ are such that

$$\int_a^b w(x)\,\mathrm{d}x = 1 \quad \text{and} \quad w(x) > 0\ \forall\ x\ .$$

$w(x)$ also needs to be analytically integratable and the result of its integral needs to be an invertible function.
How to code it up:

```
define function f        (of which you seek the integral)
define function w        (the weigth function)
define function Fx(y)    (the inverse of the integral of w)
I = 0
For i in 1 to N do
   r = uniform RN in [0,1)
   x = Fx(r)
   I = I + f(x)/w(x)
End
I = I / N
```

Quiz: Define the 3 functions you will need ($f(x)$, $w(x)$, and $x(y)$) in order to compute $I = \int_{-1/3}^{3/2} \frac{dx}{1+x^2}$ using MC integration with importance sampling with the function $w(x) = A(4-2x)$ as your weigth, where A is the appropriate normalization constant.

5 Solutions to ordinary differential equations (ODEs)

From [7]. Problems involving ODEs can always be reduced to the study of sets of first-order differential equations. For example, the second-order equation

$$\frac{\mathrm{d}^2 y}{\mathrm{d}x^2} + q(x)\frac{\mathrm{d}y}{\mathrm{d}x} = r(x)$$

can be rewritten as two first-order equations

$$\begin{aligned} \frac{\mathrm{d}y}{\mathrm{d}x} &= z(x) \\ \frac{\mathrm{d}z}{\mathrm{d}x} &= r(x) - q(x)z(x) \ . \end{aligned}$$

The generic problem in ODE equations is reduced to the study of a set of N coupled first-order ODEs for the functions y_i, $i = 0, 1, 2, ..., N-1$, having the general form

$$\frac{\mathrm{d}y_i(x)}{\mathrm{d}x} = f_i(x, y_0, y_1, y_2, ..., y_{N-1}) \ , \quad i = 0, 1, ..., N-1$$

where the functions f_i on the right-hand side are known.

5.1 Initial-value problems

Initial-value problems are problems where the value of $y(t)$ at $t = 0$ is known, and solving the ODE consists in propagating the solution, $y(t)$, in time, yielding the evolution of the system in time.

5.1.1 Euler's method

This method is based on the idea of using the Taylor series expansion of x around t. A quick reminder for those less familar with Taylor series. A Taylor series is a series expansion of a function about a point. A 1-D Taylor series is an expansion of a real function $g(x)$ about a point $x = a$, and is given by

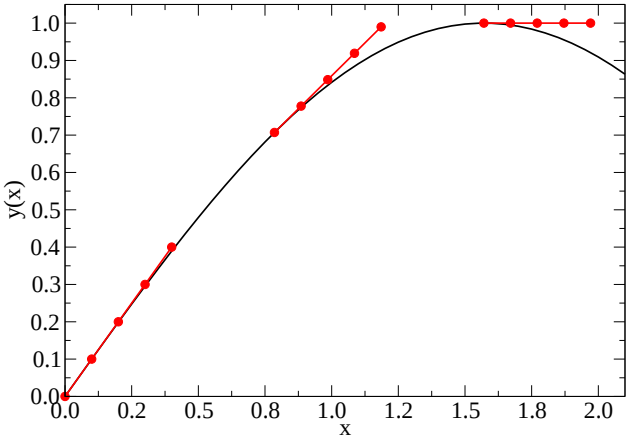
$$g(b) = g(a) + \frac{\mathrm{d}g}{\mathrm{d}x}(b-a) + \frac{1}{2!}\frac{\mathrm{d}^2 g}{\mathrm{d}x^2}(b-a)^2 + \frac{1}{3!}\frac{\mathrm{d}^3 g}{\mathrm{d}x^3}(b-a)^3 + ... + \frac{1}{n!}\frac{\mathrm{d}^n g}{\mathrm{d}x^n}(b-a)^n + ...$$

where all the derivatives ($\mathrm{d}g/\mathrm{d}x$, $\mathrm{d}^2 g/\mathrm{d}x^2$, etc.) are evaluated at $x = a$. Thus, if we have $y(x)$, we can express $y(x + \Delta x)$ using the Taylor series expansion of y about x, namely

$$y(x+\Delta x) = y(x) + \frac{\mathrm{d}y}{\mathrm{d}x}\Delta x + \frac{1}{2!}\frac{\mathrm{d}^2 y}{\mathrm{d}x^2}\Delta x^2 + \frac{1}{3!}\frac{\mathrm{d}^3 y}{\mathrm{d}x^3}\Delta x^3 + ... + \frac{1}{n!}\frac{\mathrm{d}^n y}{\mathrm{d}x^n}\Delta x^n + ...$$

We can truncate this infinite series to an approximation of order $\mathcal{O}(\Delta x^2)$, and write it as

$$y(x+\Delta x) = y(x) + \frac{\mathrm{d}y}{\mathrm{d}x}\Delta x + \mathcal{O}(\Delta x^2)$$



Since $\mathrm{d}y/\mathrm{d}x = f(x, y)$, we can rewrite the above as

$$y(x+\Delta x) \approx y(x) + f(x,y)\Delta x \ .$$

Another way to look at this is to discretized the ODE such that

$$\lim_{\Delta x \to 0} \frac{y(x+\Delta x) - y(x)}{\Delta x} = \frac{\mathrm{d}y}{\mathrm{d}x} = f(x,y)$$

It is easy to see that for large Δx or for functions where the slope of $y(x)$, $f(x, y)$, changes rapidly (namely, where $\mathrm{d}^2 y/\mathrm{d}x^2$ or $\mathrm{d}f/\mathrm{d}x$ is large) this will lead to poor results. For this reason, in most applications, Euler should not be used because

1. the method is not very accurate compared to others; and
2. it is not very stable (numerical solution diverges from true solution).

Typically, the Euler method will be written as

$$y_{i+1} = y_i + f(x_i, y_i)h$$

where h is the step size, $x_i = x_0 + ih$, and $y_i = y(x_i)$. So you would code it up as:

```
x = Initial value
y = Initial value
h = Chosen step size
For i in 1 to N do
   y = y + f(x,y) * h
   x = x + h
   Print x y
End
```

5.1.2 Runge-Kutta method

One big problem with the Euler method is that it advances the solution through an interval h, but uses the derivative information only at the beginning of that interval,

$$y_{n+1} = y_n + hf(x_n, y_n) \ .$$

One obvious way to correct this would be to take half a step to determine $y_{n+1/2}$ and use the slope at $f(x_{n+1/2}, y_{n+1/2})$ to determine y_{n+1}. Namely,

$$\begin{aligned} k_1 &= hf(x_n, y_n) \\ k_2 &= hf\left(x_n + \frac{h}{2}, y_n + \frac{k_1}{2}\right) \\ y_{n+1} &= y_n + k_2 + \mathcal{O}(h^3) \end{aligned}$$

This method is known as the ***second-order Runge-Kutta*** method. Of course, there is no reason to stop at the second-order. Probably the most popular popular one is the ***fourth-order Runge-Kutta***, namely

$$\begin{aligned} k_1 &= hf(x_n, y_n) \\ k_2 &= hf\left(x_n + \frac{h}{2}, y_n + \frac{k_1}{2}\right) \\ k_3 &= hf\left(x_n + \frac{h}{2}, y_n + \frac{k_2}{2}\right) \\ k_4 &= hf\left(x_n + h, y_n + k_3\right) \\ y_{n+1} &= y_n + \frac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right) + \mathcal{O}(h^5) \end{aligned}$$

While this method is more computationally costly than Euler (requires 4 more evaluations per step), its precision is much better and as a result, the step size, h, can be made larger. It is usually the approximation of choice for numerical problems requiring relatively high accuracies.

5.2 Two-point boundary-value problems

From [7]. Boundary-value problems are problems described by a function, $y(x)$, which is static in time, but needs to satisfy some boundary conditions at $x = x_{\min}$ and $x = x_{\max}$. In the case of a first order ODE of the type

$$\frac{\mathrm{d}y}{\mathrm{d}x} = f(x, y) \ ,$$

having a single boundary condition and solving the above using any initial-value scheme (e.g. Euler, RK) is sufficient. But for a second-order ODE such as

$$\frac{\mathrm{d}^2 y}{\mathrm{d}x^2} = f(x, y, y')$$

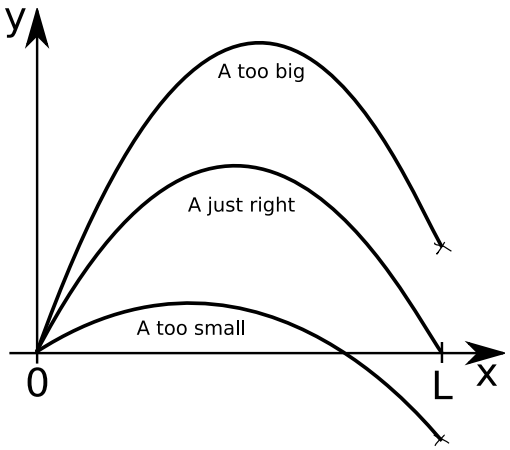
which can be expressed as 2 coupled first-order ODEs as

$$\begin{aligned} \frac{\mathrm{d}y}{\mathrm{d}x} &= z \\ \frac{\mathrm{d}z}{\mathrm{d}x} &= f(x, y, z) \end{aligned}$$

specifying $y(x_{\min})$ is not sufficient to carry out the initial-value integration because one also requires $y'(x_{\min})$ or rather $z(x_{\min})$.

More generally, a boundary-value problem is one where have a set of N coupled first-order ODEs, and we'll be given $n_1 < N$ ICs at one boundary, and $n_2 = N - n_1$ ICs at the other boundary. So we'll always be missing n_2 required values at the starting boundary to be able to carry out the integration as we did with initial-value problems. Solving a boundary-value problem comes down to working around this issue. Several schemes exist, but we will look at the 2 most popular/common ones.

5.2.1 The shooting method



The shooting method is so called because it consists in shooting, correcting your aim, and shooting again, until you get it right at the other boundary within a satisfying precision. To help understand what we are doing, let's look at a concrete example, say $\frac{\mathrm{d}^2y}{\mathrm{d}x^2} = -k^2x$ which we'll write as

$$\begin{aligned} \frac{\mathrm{d}y}{\mathrm{d}x} &= y' \\ \frac{\mathrm{d}y'}{\mathrm{d}x} &= -k^2x \end{aligned}$$

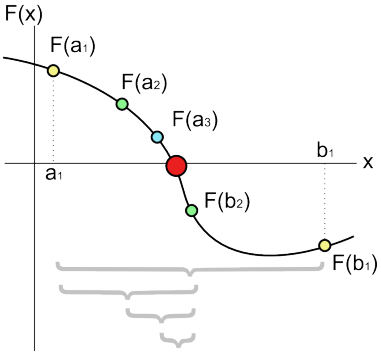
Furthermore, we are given $y(0) = 0$ and $y(L) = 0$. To solve this problem as an initial-value problem, we need $y'(0)$. So we proceed as follows:

1. Take a good guess for $y'(0)$, say $y'(0) = A$
2. Solve your system of first-order ODEs as an initial-value problem with a method of your choice (e.g., 4th order Runge-Kutta, `lsode`)
3. Once you get to $x = x_{\max}$, determine how much you missed your target by and decide by how much to adjust your A.
4. Restart from 2.

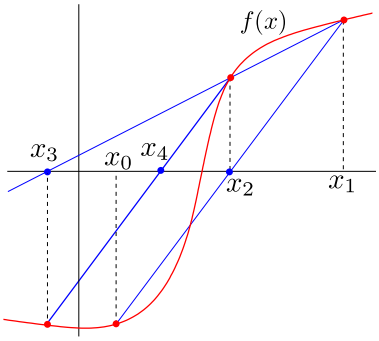
Step 3 above essentially comes down to finding the root of an expression. Namely, define the function F(A) $= y(x_{\max}) - 0$, where 0 is your target $y(L) = 0$, and $y(x_{\max})$ is the value you got for y at $x = x_{\max}$ due to your choice of A.

5.2.2 A brief word on root finding

Many root finding methods can help you pick a nice way to improve your guess for A. Let's illustrate two basic root-finding methods: the bisection and the secant.



Bisection from Wikimedia.



Secant from Wikimedia.

Bisection method From Wikipedia. Suppose we want to solve the equation $f(x) = 0$, where f is a continuous function. The bisection method starts with two points a and b such that $f(a)$ and $f(b)$ have opposite signs. The intermediate value theorem says that f must have at least one root in the interval $[a, b]$. The method now divides the interval in two by computing $c = (a + b)/2$. There are now two possibilities: either $f(a)$ and $f(c)$ have opposite signs, or $f(c)$ and $f(b)$ have opposite signs. The bisection algorithm is then applied recursively to the sub-interval where the sign change occurs. The bisection method converges slowly, but it is guaranteed to find a root.

Secant method From MathWorld and Wikipedia. The secant method assumes the function to be approximately linear in the region of interest. Each improvement is taken as the point where the approximating line crosses the axis. Given x_{n-1} and x_n, we construct the line through the points $(x_{n-1}, f(x_{n-1}))$ and $(x_n, f(x_n))$, as demonstrated in the picture on the left. Note that this line is a secant or chord of the graph of the function f. In finite-difference, it can be represented as

$$y - f(x_n) = \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}(x - x_n) \ .$$

We now choose x_{n+1} to be the root of this line, so x_{n+1} is chosen such that

$$f(x_n) + \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}(x_{n+1} - x_n) = 0 \ .$$

Solving this equation gives the recurrence relation for the secant method

$$x_{n+1} = x_n - \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})} f(x_n) \ .$$

The secant method retains only the most recent estimate, so the root does not necessarily remain bracketed. So while the secant method is faster than the bisection method, it is not guaranteed to converge to a solution.

Root finding in Octave For the purpose of this course, note that the function `fzero` in Octave is often a good choice for root-finding, and depending on your initial guess, the function will pick between some of the best root-finding methods such as Brent's method and Powell's method (a mixture of the secant, the bisection, and other methods/tricks).

The code below demonstrate usage of `fzero` to solve the following problem: find the value of x at which a unit circle centered at the origin ($y = \sqrt{1 - x^2}$) and a parabola centered at the origin ($y = x^2$) will intersect.

```
#!/usr/bin/octave -q

function y = f(x)
   y = sqrt(1 - x^2);
endfunction;

function y = g(x)
   y = x.^2;
endfunction;

function smin = fct_to_minimize(x)
   smin = f(x) - g(x);
endfunction;

# Find minimum in interval x = [0,1]
[minval,dif] = fzero('fct_to_minimize',[0,1]);
printf("Value of (x,y) = (%f,%f), w interval, diff=%g\n", minval, f(minval), dif);

# Find minimum with initial guess x = 0.5
[minval,dif] = fzero('fct_to_minimize',0.5);
printf("Value of (x,y) = (%f,%f) w initial guess, diff=%g\n", minval, f(minval), dif);
```

Running this code will give you:

```
myprompt% ./finding-roots.oct
Value of (x,y) = (0.786151,0.618034), w interval, diff=-4.63721e-11
Value of (x,y) = (0.786151,0.618034) w initial guess, diff=-3.33067e-16
```

5.2.3 The relaxation method

From [2]. This method is particularly important: it is used in many applications, and we will see it again in the context of PDE solving. The idea is simple: express your ODE as a finite-difference (as with Euler), divide your solution space into N sub-steps, each corresponding to a given finite-difference equation, then solve the set of simultaneous equations. You will need a slightly different approach depending on whether you have a linear or nonlinear ODE.

An example of a linear boundary-value problem Let me illustrate what I mean by a simple example.

$$y'' = y + x(x-4), \quad 0 \leq x \leq 4 \tag{1}$$

with $y(0) = y(4) = 0$. We can replace y'' in the above by its finite-difference approximation. Consider the Taylor expansions of the elements to the right and left of a given $y(x)$

$$\begin{aligned} y(x+\Delta x) &= y(x) + y' \cdot \Delta x + \frac{1}{2}y'' \cdot (\Delta x)^2 + \frac{1}{6}y''' \cdot (\Delta x)^3 + \frac{1}{24}y'''' \cdot (\Delta x)^4 + ... \\ y(x-\Delta x) &= y(x) - y' \cdot \Delta x + \frac{1}{2}y'' \cdot (\Delta x)^2 - \frac{1}{6}y''' \cdot (\Delta x)^3 + \frac{1}{24}y'''' \cdot (\Delta x)^4 - ... \end{aligned}$$

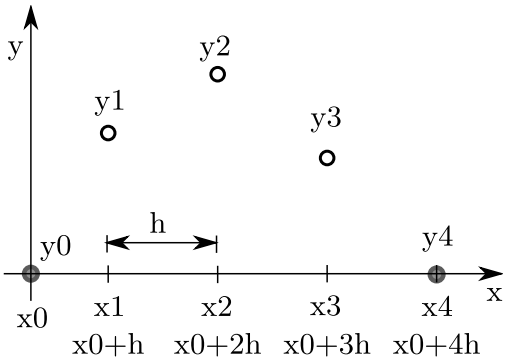
If we add these elements together, we get

$$\begin{aligned} y(x+\Delta x)+y(x-\Delta x) &= 2y(x)+y''\cdot(\Delta x)^2+\frac{1}{12}y''''\cdot(\Delta x)^4+... \\ y'' &= \frac{y(x+\Delta x)-2y(x)+y(x-\Delta x)}{\Delta x^2}+\mathcal{O}(\Delta x^4) \end{aligned}$$

where we will neglect the elements of $\mathcal{O}(\Delta x^4)$. We can write this more concisely as

$$y'' \sim \frac{y_{i+1}-2y_i+y_{i-1}}{h^2}$$

where $y_{i+1}=y(x_i+h)=y(x_0+i\cdot h)$, and h is the ***step size*** chosen to integrate the problem.



We can rewrite (1) using the finite-difference approximation as

$$\begin{aligned} y'' &= y+x(x-4), \quad 0\le x\le 4 \\ \frac{y_{i+1}-2y_i+y_{i-1}}{h^2} &= y_i+x_i(x_i-4) \\ y_{i-1}-(2+h^2)\,y_i+y_{i+1} &= h^2x_i(x_i-4)\ . \end{aligned}$$

If we pick a step size of $h=1$ we have $x_0=0$, $x_4=4$, $y_0=y_4=0$ to satisfy our BCs. With a step size of $h=1$, we have y_0 and y_4 and we are looking for y_1, y_2, and y_3. We can express our problems as a system of algebraic equations

$$\begin{aligned} y_2-2y_1+0 &= y_1+1(1-4) \\ y_3-2y_2+y_1 &= y_2+2(2-4) \\ 0-2y_3+y_2 &= y_3+3(3-4)\ , \end{aligned}$$

such that in matrix form, we have to solve the system $\mathbf{Ay}=\mathbf{b}$, where

$$\begin{bmatrix} -3 & 1 & 0 \\ 1 & -3 & 1 \\ 0 & 1 & -3 \end{bmatrix}\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}=\begin{bmatrix} -3 \\ -4 \\ -3 \end{bmatrix}$$

The solution to this problem is given by $\mathbf{A}^{-1}\mathbf{b}=\mathbf{y}$, where

$$\begin{bmatrix} -3 & 1 & 0 \\ 1 & -3 & 1 \\ 0 & 1 & -3 \end{bmatrix}^{-1}\begin{bmatrix} -3 \\ -4 \\ -3 \end{bmatrix}=\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}=\begin{bmatrix} 13/7 \\ 18/7 \\ 13/7 \end{bmatrix}$$

At $x=1$, the exact solution is 1.8341 (to four decimal places) and the finite-difference relaxation method is $y_1=1.8571$. In Octave, you would do the above problem as

```
octave:1> A = [-3 1 0; 1 -3 1; 0 1 -3];
octave:2> b = [-3; -4; -3];
octave:3> y = inverse(A)*b
y =
   1.8571
   2.5714
   1.8571
```

```
octave:4> y = A \ b
y =
   1.8571
   2.5714
   1.8571
```

The general linear boundary-value problem Let's consider the more general linear two-point boundary value problem

$$y'' = p(x)y' + q(x)y + r(x), \quad a \leq x \leq b \tag{2}$$

with boundary conditions $y(a) = \alpha$, and $y(b) = \beta$. To solve this problem using finite-differences, we divide $[a, b]$ into N subintervals, so that $h = (b - a)/N$, and we have

$$y''(x_i) \approx \frac{y_{i+1} - 2y_i + y_{i-1}}{h^2}, \quad y'(x_i) \approx \frac{y_{i+1} - y_{i-1}}{2h}$$

Such that I can express (2) as

$$\begin{aligned} \frac{y_{i+1} - 2y_i + y_{i-1}}{h^2} &= p_i \frac{y_{i+1} - y_{i-1}}{2h} + q_i y_i + r_i \\ 2y_{i+1} - 4y_i + 2y_{i-1} &= hp_i y_{i+1} - hp_i y_{i-1} + 2h^2 q_i y_i + 2h^2 r_i \end{aligned}$$

where $p_i = p(x_i)$, $q_i = q(x_i)$, $r_i = r(x_i)$. We can group the y_{i-1}, y_i and y_{i+1} terms together and get

$$(2 + hp_i)y_{i-1} - (4 + 2h^2 q_i)y_i + (2 - hp_i)y_{i+1} = 2h^2 r_i$$

where the boundary conditions are such that $y_0 = \alpha$ and $y_N = \beta$. The special cases of the above are for the expressions containing y_0 and y_N which correspond to the lines for $i = 1$ and $i = N - 1$, namely

$$\begin{aligned} i = 1 : (2 + hp_1)\alpha - (4 + 2h^2 q_1)y_1 + (2 - hp_1)y_2 &= 2h^2 r_1 \\ -(4 + 2h^2 q_1)y_1 + (2 - hp_1)y_2 &= 2h^2 r_1 - (2 + hp_1)\alpha \\ i = N - 1 : (2 + hp_{N-1})y_{N-2} - (4 + 2h^2 q_{N-1})y_{N-1} + (2 - hp_{N-1})\beta &= 2h^2 r_{N-1} \\ (2 + hp_{N-1})y_{N-2} - (4 + 2h^2 q_{N-1})y_{N-1} &= 2h^2 r_{N-1} - (2 - hp_{N-1})\beta \end{aligned}$$

Thus, we can put this set of algebraic equations into a matrix of the form $\mathbf{Ay} = \mathbf{b}$ such that

$$\begin{bmatrix} -(4 + 2h^2 q_1) & (2 - hp_1) & 0 & \dots & 0 \\ (2 + hp_2) & -(4 + 2h^2 q_2) & (2 - hp_2) & \ddots & \ddots \\ 0 & (2 + hp_3) & -(4 + 2h^2 q_3) & (2 - hp_3) & \ddots \\ 0 & \ddots & \ddots & \ddots & 0 \\ 0 & \ddots & (2 + hp_{N-2}) & -(4 + 2h^2 q_{N-2}) & (2 - hp_{N-2}) \\ 0 & \dots & 0 & (2 + hp_{N-1}) & -(4 + 2h^2 q_{N-1}) \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \dots \\ y_{N-2} \\ y_{N-1} \end{bmatrix} = \begin{bmatrix} 2h^2 r_1 - (2 + hp_1)\alpha \\ 2h^2 r_2 \\ 2h^2 r_3 \\ \dots \\ 2h^2 r_{N-2} \\ 2h^2 r_{N-1} - (2 - hp_{N-1})\beta \end{bmatrix}$$

As in our example, all that is left to do is to invert $\mathbf{A}$ such that $\mathbf{A}^{-1}\mathbf{b} = y$ to get the value of all y_i. The smaller the step size, h, the larger the matrix, and the longer it will take to solve the problem.

Quiz: Determine the three terms you'll need (A, y, and b) in order to find the numerical solution of $y'' = -2xy'$ using the relaxation method with $h = 2$ with boundary conditions $y(0) = 1$ and $y(10) = 0$.

How to code it up for an arbitrary step size The most efficient and compact way to build the matrix to solve this and other problems involving tri-diagonal matrices is to use an Octave function called `diag`. The function `diag` is such that:

```
octave> diag(5*ones(3,1),-1)
ans =
   0   0   0   0
   5   0   0   0
   0   5   0   0
   0   0   5   0


octave> diag((1-2*[1:4]),0)
ans =
  -1   0   0   0
   0  -3   0   0
   0   0  -5   0
   0   0   0  -7
```

```
octave> diag([2:6],1)
ans =
   0   2   0   0   0   0
   0   0   3   0   0   0
   0   0   0   4   0   0
   0   0   0   0   5   0
   0   0   0   0   0   6
   0   0   0   0   0   0

octave> diag([1:2],2)+diag([4:6],1)
ans =
   0   4   1   0
   0   0   5   2
   0   0   0   6
   0   0   0   0
```

Quiz: Determine the three terms you'll need (A, y, and b) in order to find the numerical solution of $y'' - y/(\mathrm{e}^x + 1) = 0$ using the relaxation method with boundary conditions $y(0) = 1$ and $y(1) = 5$ by divinding space into N intervals.

Quiz: One applied example of a boundary value problem is that of temperature distribution. Let us assume that a very small blood vessel located at the surface of the skin has its inner radius, r_{in}, held at body temperature ($T = 37\ ^\circ\text{C}$) and its outer radius, r_{out}, held at room temperature ($T = 23\ ^\circ\text{C}$). The radial temperature between the inner and outer radii obeys the following equation

$$\frac{\mathrm{d}^2T}{\mathrm{d}r^2} + \frac{1}{r}\frac{\mathrm{d}T}{\mathrm{d}r} = 0$$

where T is the temperature in $^\circ\text{C}$, and r is the radius. Of course, our boundary values are such that $T(r_{\text{in}}) = 37\ ^\circ\text{C}$ and $T(r_{\text{out}} = 23\ ^\circ\text{C}$. Set up both the shooting method (using Euler's method) and relaxation method for a step size h for this problem, doing all the pre-coding calculations, but without working out the actual code.

6 Solutions to partial differential equations (PDEs)

6.1 PDE generalities

6.1.1 Classes of linear second-order PDEs

From [5, 8] and Wikipedia. A partial differential equation or PDE is a differential equation with more than one independent variable. Physical quantities such as temperature and pressure vary continuously in both space and time. We can represent such quantities and their change over time using a function, or field, $U(x, y, z, t)$ which contains independent space and time variables.

We usually come across three types of linear second-order PDEs in mechanics. These are classified as ***elliptic***, ***hyperbolic***, and ***parabolic***.

Each type of PDE has certain characteristics that help determine if a particular finite difference approach is appropriate to the problem being described by the PDE. Interestingly, just knowing the

type of PDE can give us insight into how smooth the solution is, how fast information propagates, and the effect of initial and boundary conditions.

The most general form for a linear second-order PDE with two independent variables is

$$a\frac{\partial^2 U}{\partial x^2} + 2b\frac{\partial^2 U}{\partial x \partial y} + c\frac{\partial^2 U}{\partial y^2} + d\frac{\partial U}{\partial x} + e\frac{\partial U}{\partial y} = f$$

where a, b, c, d, e, and f are arbitrary function of the variables x and y. The class of a linear second-order PDE is defined by the relative value of these functions, namely

Class	Elliptic	Parabolic	Hyperbolic
Condition	$ac > b^2$	$ac = b^2$	$ac < b^2$
Example	Poisson equation	Heat equation	Wave equation
	$\nabla^2 U(x) = -4\pi\rho(x)$	$\nabla^2 U(\vec{x}, t) = a\,\partial U/\partial t$	$\nabla^2 U(\vec{x}, t) = c^{-2}\,\partial^2 U/\partial t^2$

Just for your information:

Poisson equation : describes the value of the electric potential in space as a function of the charge density, ρ.

Heat equation : describes the temperature of a medium over time and space as a function of the heat sources and initial heat distribution. It has the same form as the diffusion equation.

Wave equation : describes the displacement from equilibrium of the elements of the medium in time and space as a function of their ICs, BCs, and the properties of the medium.

At this point, I believe that a brief review of the mathematical differential operators will be helpful:

$\vec{\nabla} f$ **or gradient:** results in a vector quantity. In physical terms, the gradient represent the rate of change (slope) of the function in each coordinate direction. The gradiant points uphill and its magnitude is the slope of the function in that direction. In cartesian coordinates in 3-D, it is

$$\vec{\nabla} f(x, y, z) = \frac{\partial f}{\partial x}\hat{i} + \frac{\partial f}{\partial y}\hat{j} + \frac{\partial f}{\partial z}\hat{k}\ .$$

$\vec{\nabla} \times \vec{f}$ **or curl:** results in a vector quantity. In physical terms, the curl represents the amount and direction of circulation around a point. In cartesian coordinates in 3-D, it is

$$\vec{\nabla} \times \vec{f}(x, y, z) = \left(\frac{\partial f_z}{\partial y} - \frac{\partial f_y}{\partial z}\right)\hat{i} + \left(\frac{\partial f_x}{\partial z} - \frac{\partial f_z}{\partial x}\right)\hat{j} + \left(\frac{\partial f_y}{\partial x} - \frac{\partial f_x}{\partial y}\right)\hat{k}\ .$$

$\vec{\nabla} \cdot \vec{f}$ **or divergence:** results in a scalar quantity. In physical terms, the divergence of a function is the extent to which the function behaves like a source or a sink at a given point. In cartesian coordinates in 3-D, it is

$$\vec{\nabla} \cdot \vec{f} = \frac{\partial f_x}{\partial x} + \frac{\partial f_y}{\partial y} + \frac{\partial f_z}{\partial z}\ .$$

$\nabla^2 f$ or Δf or laplacian: results in a scalar quantity. In physical terms, the gradient-part of the laplacian determines the rate of change of the function at each point, creating a vector field pointing uphill everywhere. The subsequent application of the divergence to that vector field determines whether a point will act more like a sink (all gradient vectors pointing inwards, top of hill) or a source (all gradient vectors pointing outwards, bottom of valley). In cartesian coordinates in 3-D, it is

$$\Delta f = \nabla^2 f = \vec{\nabla} \cdot \vec{\nabla} f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} + \frac{\partial^2 f}{\partial z^2} \ .$$

6.1.2 Types of boundary conditions

Before one starts to look for a solution to any PDE, one requires ***initial conditions*** or ICs, as well as ***boundary conditions*** or BCs. The ICs specify the value of the function over all space at the initial time, namely $U(x, y, z, t_0)$ in 3-D cartesian coordinates, where t_0 is the initial time.

The BCs specify how you expect your solution to behave at the edges of its domain. For example, the heat equation is used to describe the temperature everwhere on a plate of metal held at room temperature on three of its sides, but held at 80° on the fourth side, so my solution will have to match these conditions along the edges of the plate for all times. There are severaly types of BCs depdending on the physical system to be represented. For linear second-order PDEs, these are

Dirichlet BCs: specify the values that the solution is to take on the boundary of the domain.

Neumann BCs: specifies the values that the derivative of a solution is to take on the boundary of the domain.

Cauchy BCs: specify both the values of the solution and its derivative on the boundary of the domain.

Although having adequate BCs is necessary for a unique solution, having too many BCs may be an overspecification for which no solution exists.

6.1.3 Initial value problem vs boundary value problems

From [7].

	Initial value problems
Description	Describes how $U(\vec{x}, t)$ propagates itself forward in time, given U at some initial time t_0 for all $\vec{x}$. In other words, the equation describes time evolution.
Goal	Track the time evolution of the solution with some desired accuracy.
Concern	**Stability**: Many reasonable-looking algorithms for initial value problems just don't work — they are numerically unstable.
Classes	Parabolic and Hyperbolic

	Boundary value problems
Description	Describes a single static function $U(\vec{x})$ which satisfies the equation within some (x, y) region of interest and which has some desired behaviour on the boundary of the region of interest.
Goal	To somehow converge on the correct solution everywhere at once. In general it is not possible stably to just integrate in from the boundary in the same sense that an initial value problem can be integrated forward in time.
Concern	**Efficiency**: The fact that all conditions on a boundary value problem must be satisfied ***simultaneously*** requires solving a large numbers of simultaneous algebraic equation which is both CPU and RAM intensive. Stability is relatively easy to achieve.
Classes	Elliptic

This, then, is the most important classification from a computational point of view: Is the problem at hand an ***initial value problem*** (time evolution) or is it a ***boundary value problem*** (static solution)?

6.2 Parabolic PDEs

From [7, 8]. The "model" equation for parabolic PDEs is the heat or diffusion equation

$$\frac{\partial U}{\partial t} = D\nabla^2 U$$

where D here is the diffusion coefficient which does not vary over space. In one dimension, the diffusion equation is simply

$$\frac{\partial U(x,t)}{\partial t} = D\frac{\partial^2 U(x,t)}{\partial x^2}$$

In general, for diffusion problems, one is given the solution $U(x, 0)$ over all space at time zero, as well as the boundary conditions $U(x_{\min}, t) = \alpha(t)$ and $U(x_{\max}, t) = \beta(t)$ valid at all times. Let's see how we can numerically determine the solution to this equation in space and time.

6.2.1 Forward difference method

From [7, 8]. Let us go back to our old friend the finite difference method. We have seen in Section 5.2.3 on the relaxation method that one can express a second order differential equation as:

$$\frac{\mathrm{d}^2 y}{\mathrm{d}x^2} \approx \frac{y_{i+1} - 2y_i + y_{i-1}}{(\Delta x)^2} + \mathcal{O}(\Delta x^4)$$

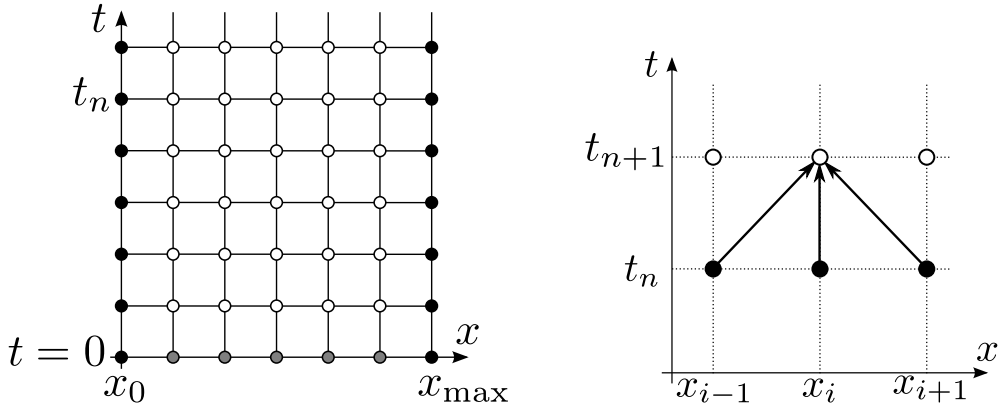
and that one can express a first order differential equation as:

$$\frac{\mathrm{d}y}{\mathrm{d}x} \approx \frac{y_{i+1} - y_i}{\Delta x} + \mathcal{O}(\Delta x^2)$$

Thus, we can express our diffusion equation as a forward difference scheme

$$\begin{aligned} \frac{\partial U(x,t)}{\partial t} &= D\,\frac{\partial^2 U(x,t)}{\partial x^2} \\ \frac{u_i^{n+1} - u_i^n}{\Delta t} &= D\,\frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{(\Delta x)^2} \end{aligned}$$

where n is the time subscript such that $t_{n+1} = t_n + \Delta t$, and i is the space subscript such that $x_{i+1} = x_i + \Delta x$. This scheme is of order $\mathcal{O}(\Delta x^4 + \Delta t^2)$. We can represent the mesh and stencil of the forward difference method as



Along the bottom of the mesh, we can see the ICs, and along the edges we can see the BCs. This method is an ***explicit*** method because there is a way to determine new values (in the sense of time) directly from the previously known values.

For solving the forward difference expression, it is helpful to rearrange the expression to separate the terms at time n from those at time $n+1$ as

$$\begin{aligned} \frac{u_i^{n+1} - u_i^n}{\Delta t} &= D\,\frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{(\Delta x)^2} \\ u_i^{n+1} &= \frac{D\Delta t}{(\Delta x)^2}\left[u_{i+1}^n - 2u_i^n + \frac{(\Delta x)^2}{D\Delta t}u_i^n + u_{i-1}^n\right] \\ u_i^{n+1} &= \sigma u_{i-1}^n + (1-2\sigma)u_i^n + \sigma u_{i+1}^n \end{aligned}$$

where $\sigma = D\Delta t/(\Delta x)^2$. Now let's turn this into a linear algebra problem as we did with the relaxation method.

Example case To help understand how to use this expression, let us look at the particular cases where we have a spatial step size $\Delta x = (x_{\max} - x_{\min})/5$ such that our range $[x_{\min}, x_{\max}]$ at time step n is represented by $u_0^n, u_1^n, u_2^n, u_3^n, u_4^n$, and u_5^n. This means that $u_0^n = U(x_{\min}, t) = \alpha^n$ and $u_5^n = U(x_{\max}, t) = \beta^n$, our boundary conditions at time $t = t_n$, and we know u_i^0 for all i as these

are our initial conditions. Then, our first step (i.e., going from u_i^0 (the solution at $t = 0$, the initial condition) to u_i^1 (the solution at $t = \Delta t$) will be

$$\begin{aligned} i = 1 : u_1^1 &= \sigma\alpha^0 + (1 - 2\sigma)u_1^0 + \sigma u_2^0 \\ i = 2 : u_2^1 &= \sigma u_1^0 + (1 - 2\sigma)u_2^0 + \sigma u_3^0 \\ i = 3 : u_3^1 &= \sigma u_2^0 + (1 - 2\sigma)u_3^0 + \sigma u_4^0 \\ i = 4 : u_4^1 &= \sigma u_3^0 + (1 - 2\sigma)u_4^0 + \sigma\beta^0 \end{aligned}$$

we can express this in matrix form as

$$\begin{bmatrix} 1-2\sigma & \sigma & 0 & 0 \\ \sigma & 1-2\sigma & \sigma & 0 \\ 0 & \sigma & 1-2\sigma & \sigma \\ 0 & 0 & \sigma & 1-2\sigma \end{bmatrix} \begin{bmatrix} u_1^0 \\ u_2^0 \\ u_3^0 \\ u_4^0 \end{bmatrix} + \sigma \begin{bmatrix} \alpha^0 \\ 0 \\ 0 \\ \beta^0 \end{bmatrix} = \begin{bmatrix} u_1^1 \\ u_2^1 \\ u_3^1 \\ u_4^1 \end{bmatrix}$$

General case In general, if we choose $\Delta x = (x_{\max} - x_{\min})/M$, we can express our system of equations as a

$$\begin{bmatrix} 1-2\sigma & \sigma & 0 & \dots & 0 \\ \sigma & 1-2\sigma & \sigma & \ddots & \vdots \\ 0 & \sigma & 1-2\sigma & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & \sigma \\ 0 & \dots & 0 & \sigma & 1-2\sigma \end{bmatrix} \begin{bmatrix} u_1^n \\ u_2^n \\ u_3^n \\ \vdots \\ u_{M-1}^n \end{bmatrix} + \sigma \begin{bmatrix} \alpha^n \\ 0 \\ \vdots \\ 0 \\ \beta^n \end{bmatrix} = \begin{bmatrix} u_1^{n+1} \\ u_2^{n+1} \\ u_3^{n+1} \\ \vdots \\ u_{M-1}^{n+1} \end{bmatrix}$$

which is of the form $\mathbf{A}\mathbf{u}^\mathbf{n} + \sigma\mathbf{b}^\mathbf{n} = \mathbf{u}^{\mathbf{n+1}}$, where $\mathbf{b}^\mathbf{n}$ is the vector holding the boundary conditions at time $t = t_n$. One obtains the solution of the PDE over time by repeatedly iterating the system from $n = 0$ to $n = N - 1$ where $\Delta t = (t_{\max} - t_{\min})/N$.

Stability The stability criterion for the forward difference method is

$$\frac{2D\Delta t}{(\Delta x)^2} \leq 1 \quad \text{or} \quad 2\sigma \leq 1$$

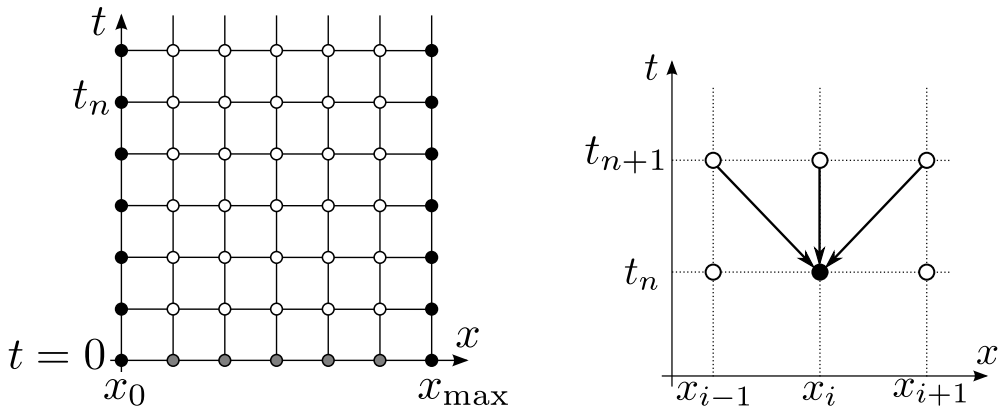
This means that one has to either sacrifice spatial resolution (pick a large Δx) or take a very large number of time steps (pick a small Δt) in order to keep the integration stable. Since the stability depends on $(\Delta x)^2$ but on Δt, it is usually better to sacrifice spatial resolution.

6.2.2 Backward difference method

From [7, 8]. Consider instead the following approximate solution for our parabolic PDE

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = D\,\frac{u_{i+1}^{n+1} - 2u_i^{n+1} + u_{i-1}^{n+1}}{(\Delta x)^2}\,.$$

We can represent the mesh and stencil of the backward difference method as



Now to understand how to use this finite difference expression, let us separate the terms at time n from those at time $n+1$ as

$$\begin{aligned}
\frac{u_i^{n+1} - u_i^n}{\Delta t} &= D\,\frac{u_{i+1}^{n+1} - 2u_i^{n+1} + u_{i-1}^{n+1}}{(\Delta x)^2} \\
u_i^{n+1} - u_i^n &= \frac{D\Delta t}{(\Delta x)^2}\left[u_{i+1}^{n+1} - 2u_i^{n+1} + u_{i-1}^{n+1}\right] \\
u_i^n &= \sigma\left[-u_{i+1}^{n+1} + 2u_i^{n+1} - u_{i-1}^{n+1}\right] + u_i^{n+1} \\
u_i^n &= -\sigma\, u_{i-1}^{n+1} + (1+2\sigma)\, u_i^{n+1} - \sigma\, u_{i+1}^{n+1}
\end{aligned}$$

where $\sigma = D\Delta t/(\Delta x)^2$. The backward difference method is an ***implicit*** method because we express known values as a function of unknown values. In contrast, a fully explicit scheme (such as the forward difference method we saw before) expresses unknown values as a function of only known values.

Example case To help understand how to use this expression, let us look at the particular cases where we have a spatial step size $\Delta x = (x_{\max} - x_{\min})/5$ such that our range $[x_{\min}, x_{\max}]$ at time step n is represented by $u_0^n, u_1^n, u_2^n, u_3^n, u_4^n$, and u_5^n. This means that $u_0^n = U(x_{\min}, t) = \alpha^n$ and $u_5^n = U(x_{\max}, t) = \beta^n$, our boundary conditions at time $t = t_n$, and we know u_i^0 for all i as these are our initial conditions. Then, our first step (i.e., going from u_i^0 (the solution at $t = 0$, the initial condition) to u_i^1 (the solution at $t = \Delta t$) will be

$$\begin{aligned}
i = 1 : u_1^0 &= -\sigma\alpha^1 + (1+2\sigma)u_1^1 - \sigma u_2^1 \\
i = 2 : u_2^0 &= -\sigma u_1^1 + (1+2\sigma)u_2^1 - \sigma u_3^1 \\
i = 3 : u_3^0 &= -\sigma u_2^1 + (1+2\sigma)u_3^1 - \sigma u_4^1 \\
i = 4 : u_4^0 &= -\sigma u_3^1 + (1+2\sigma)u_4^1 - \sigma\beta^1
\end{aligned}$$

we can express this in matrix form as

$$\begin{bmatrix} 1+2\sigma & -\sigma & 0 & 0 \\ -\sigma & 1+2\sigma & -\sigma & 0 \\ 0 & -\sigma & 1+2\sigma & -\sigma \\ 0 & 0 & -\sigma & 1+2\sigma \end{bmatrix} \begin{bmatrix} u_1^1 \\ u_2^1 \\ u_3^1 \\ u_4^1 \end{bmatrix} - \sigma \begin{bmatrix} \alpha^1 \\ 0 \\ 0 \\ \beta^1 \end{bmatrix} = \begin{bmatrix} u_1^0 \\ u_2^0 \\ u_3^0 \\ u_4^0 \end{bmatrix}$$

but unfortunately, we don't know u_i^1, we only know u_i^0. Fortunately, this simply requires a little bit of algebra, namely

$$\begin{aligned}
\mathbf{A}\mathbf{u}^{\mathbf{n+1}} - \sigma\mathbf{b}^{\mathbf{n+1}} &= \mathbf{u}^{\mathbf{n}} \\
\mathbf{A}\mathbf{u}^{\mathbf{n+1}} &= \mathbf{u}^{\mathbf{n}} + \sigma\mathbf{b}^{\mathbf{n+1}} \\
\mathbf{A}^{-1}\mathbf{A}\mathbf{u}^{\mathbf{n+1}} &= \mathbf{A}^{-1}\left[\mathbf{u}^{\mathbf{n}} + \sigma\mathbf{b}^{\mathbf{n+1}}\right] \\
\mathbf{u}^{\mathbf{n+1}} &= \mathbf{A}^{-1}\left[\mathbf{u}^{\mathbf{n}} + \sigma\mathbf{b}^{\mathbf{n+1}}\right]
\end{aligned}$$

such that in order to express what we don't know as a function of what we do know, we'll have to iterate the system $\mathbf{A}^{-1}\left[\mathbf{u}^{\mathbf{n}} + \sigma\mathbf{b}^{\mathbf{n+1}}\right] = \mathbf{u}^{\mathbf{n+1}}$, namely

$$\begin{bmatrix} 1+2\sigma & -\sigma & 0 & 0 \\ -\sigma & 1+2\sigma & -\sigma & 0 \\ 0 & -\sigma & 1+2\sigma & -\sigma \\ 0 & 0 & -\sigma & 1+2\sigma \end{bmatrix}^{-1} \left(\begin{bmatrix} u_1^0 \\ u_2^0 \\ u_3^0 \\ u_4^0 \end{bmatrix} + \sigma \begin{bmatrix} \alpha^1 \\ 0 \\ 0 \\ \beta^1 \end{bmatrix} \right) = \begin{bmatrix} u_1^1 \\ u_2^1 \\ u_3^1 \\ u_4^1 \end{bmatrix}$$

General case In general, if we choose $\Delta x = (x_{\max} - x_{\min})/M$, we can express our system of equations as a

$$\begin{bmatrix} 1+2\sigma & -\sigma & 0 & \dots & 0 \\ -\sigma & 1+2\sigma & -\sigma & \ddots & \vdots \\ 0 & -\sigma & 1+2\sigma & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & -\sigma \\ 0 & \dots & 0 & -\sigma & 1+2\sigma \end{bmatrix}^{-1} \left(\begin{bmatrix} u_1^n \\ u_2^n \\ u_3^n \\ \vdots \\ u_{M-1}^n \end{bmatrix} + \sigma \begin{bmatrix} \alpha^{n+1} \\ 0 \\ \vdots \\ 0 \\ \beta^{n+1} \end{bmatrix} \right) = \begin{bmatrix} u_1^{n+1} \\ u_2^{n+1} \\ u_3^{n+1} \\ \vdots \\ u_{M-1}^{n+1} \end{bmatrix}$$

which is of the form $\mathbf{A}^{-1}\left[\mathbf{u}^{\mathbf{n}} + \sigma\mathbf{b}^{\mathbf{n+1}}\right] = \mathbf{u}^{\mathbf{n+1}}$, where $\mathbf{b}^{\mathbf{n+1}}$ is the vector representing the boundary conditions at time $t = t_{n+1}$. One obtains the solution of the PDE over time by repeatedly iterating the system from $n = 0$ to $n = N - 1$ where $\Delta t = (t_{\max} - t_{\min})/N$.

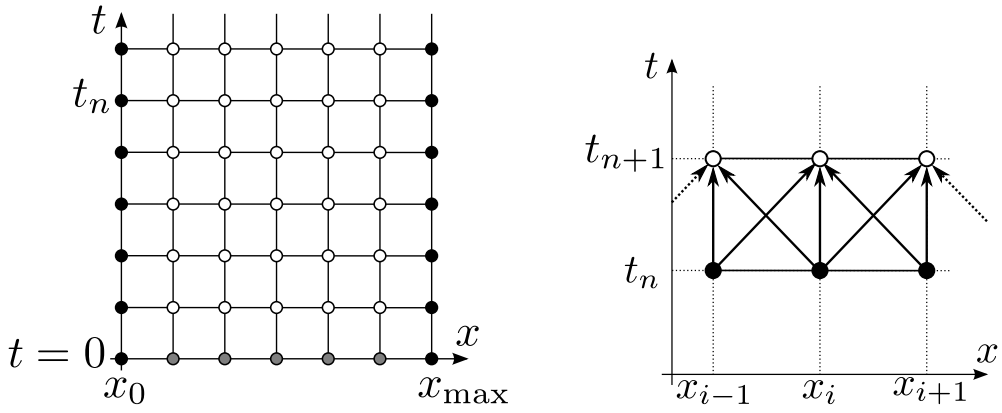
Stability The backward difference method is ***unconditionally stable***. This means that the method is stable for all choices of time and space step sizes, Δt and Δx. However, this does not mean that you can pick any Δx or Δt: there is still the issue of precision (accuracy).

6.2.3 Crank-Nicolson method

From [7, 8]. So far, our methods for solving parabolic PDEs consist of an explicit method that is sometimes stable and an implicit method that is always stable. Both methods have errors of size $\mathcal{O}(\Delta t + \Delta x^2)$ when stable. The time step size Δt needs to be fairly small to obtain good accuracy. Could we think of a method to improve accuracy? How about a mixture of the two method, an average of the explicit and implicit methods, namely

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = \frac{D}{2}\left[\frac{(u_{i+1}^{n+1} - 2u_i^{n+1} + u_{i-1}^{n+1}) + (u_{i+1}^n - 2u_i^n + u_{i-1}^n)}{(\Delta x)^2}\right] .$$

As it turns out (we will not show this here), this method has error $\mathcal{O}(\Delta t^2 + \Delta x^2)$ which allows us to take bigger time steps. We can represent the mesh and stencil of the backward difference method as



As before, we can rearrange the equation to separate the terms at time n from those at time $n+1$ as

$$\begin{aligned}
\frac{u_i^{n+1} - u_i^n}{\Delta t} &= \frac{D}{2}\left[\frac{(u_{i+1}^{n+1} - 2u_i^{n+1} + u_{i-1}^{n+1}) + (u_{i+1}^n - 2u_i^n + u_{i-1}^n)}{(\Delta x)^2}\right] \\
2u_i^{n+1} - 2u_i^n &= \frac{D\Delta t}{(\Delta x^2)}\left[(u_{i+1}^{n+1} - 2u_i^{n+1} + u_{i-1}^{n+1}) + (u_{i+1}^n - 2u_i^n + u_{i-1}^n)\right] \\
-\sigma u_{i-1}^{n+1} + (2+2\sigma)u_i^{n+1} - \sigma u_{i+1}^{n+1} &= \sigma u_{i-1}^n + (2-2\sigma)u_i^n + \sigma u_{i+1}^n
\end{aligned}$$

where $\sigma = D\Delta t/(\Delta x)^2$. And now, let's see what our matrix is going to look like.

Example case To help understand how to use this expression, let us look at the particular cases where we have a spatial step size $\Delta x = (x_{\max} - x_{\min})/5$ such that our range $[x_{\min}, x_{\max}]$ at time step n is represented by $u_0^n, u_1^n, u_2^n, u_3^n, u_4^n$, and u_5^n. This means that $u_0^n = U(x_{\min}, t) = \alpha^n$ and $u_5^n = U(x_{\max}, t) = \beta^n$, our boundary conditions at time $t = t_n$, and we know u_i^0 for all i as these are our initial conditions. Then, our first step (i.e., going from u_i^0 (the solution at $t = 0$, the initial condition) to u_i^1 (the solution at $t = \Delta t$) will be

$$\begin{aligned}
i = 1 : -\sigma\alpha^1 + (2+2\sigma)u_1^1 - \sigma u_2^1 &= \sigma\alpha^0 + (2-2\sigma)u_1^0 + \sigma u_2^0 \\
i = 2 : -\sigma u_1^1 + (2+2\sigma)u_2^1 - \sigma u_3^1 &= \sigma u_1^0 + (2-2\sigma)u_2^0 + \sigma u_3^0 \\
i = 3 : -\sigma u_2^1 + (2+2\sigma)u_3^1 - \sigma u_4^1 &= \sigma u_2^0 + (2-2\sigma)u_3^0 + \sigma u_4^0 \\
i = 4 : -\sigma u_3^1 + (2+2\sigma)u_4^1 - \sigma\beta^1 &= \sigma u_3^0 + (2-2\sigma)u_4^0 + \sigma\beta^0
\end{aligned}$$

we can express this in matrix form as

$$\begin{bmatrix} 2+2\sigma & -\sigma & 0 & 0 \\ -\sigma & 2+2\sigma & -\sigma & 0 \\ 0 & -\sigma & 2+2\sigma & -\sigma \\ 0 & 0 & -\sigma & 2+2\sigma \end{bmatrix} \begin{bmatrix} u_1^1 \\ u_2^1 \\ u_3^1 \\ u_4^1 \end{bmatrix} - \sigma \begin{bmatrix} \alpha^1 \\ 0 \\ 0 \\ \beta^1 \end{bmatrix} = \begin{bmatrix} 2-2\sigma & \sigma & 0 & 0 \\ \sigma & 2-2\sigma & \sigma & 0 \\ 0 & \sigma & 2-2\sigma & \sigma \\ 0 & 0 & \sigma & 2-2\sigma \end{bmatrix} \begin{bmatrix} u_1^0 \\ u_2^0 \\ u_3^0 \\ u_4^0 \end{bmatrix} + \sigma \begin{bmatrix} \alpha^0 \\ 0 \\ 0 \\ \beta^0 \end{bmatrix}$$

Once again, turning this into an easy to use iteratable system requires just a little bit of algebra, namely

$$\begin{aligned}
\mathbf{A}\mathbf{u}^{\mathbf{n+1}} - \sigma\mathbf{b}^{\mathbf{n+1}} &= \mathbf{B}\mathbf{u}^{\mathbf{n}} + \sigma\mathbf{b}^{\mathbf{n}} \\
\mathbf{A}\mathbf{u}^{\mathbf{n+1}} &= \mathbf{B}\mathbf{u}^{\mathbf{n}} + \sigma\mathbf{b}^{\mathbf{n}} + \sigma\mathbf{b}^{\mathbf{n+1}} \\
\mathbf{A}^{-1}\mathbf{A}\mathbf{u}^{\mathbf{n+1}} &= \mathbf{A}^{-1}\left[\mathbf{B}\mathbf{u}^{\mathbf{n}} + \sigma\left(\mathbf{b}^{\mathbf{n}} + \mathbf{b}^{\mathbf{n+1}}\right)\right] \\
\mathbf{u}^{\mathbf{n+1}} &= \mathbf{A}^{-1}\left[\mathbf{B}\mathbf{u}^{\mathbf{n}} + \sigma\left(\mathbf{b}^{\mathbf{n}} + \mathbf{b}^{\mathbf{n+1}}\right)\right]
\end{aligned}$$

such that in order to express what we don't know as a function of what we do know, we'll have to iterate the system $\mathbf{A}^{-1}\left[\mathbf{B}\mathbf{u}^{\mathbf{n}} + \sigma\left(\mathbf{b}^{\mathbf{n}} + \mathbf{b}^{\mathbf{n+1}}\right)\right] = \mathbf{u}^{\mathbf{n+1}}$, namely

$$\begin{bmatrix} 2+2\sigma & -\sigma & 0 & 0 \\ -\sigma & 2+2\sigma & -\sigma & 0 \\ 0 & -\sigma & 2+2\sigma & -\sigma \\ 0 & 0 & -\sigma & 2+2\sigma \end{bmatrix}^{-1} \left(\begin{bmatrix} 2-2\sigma & \sigma & 0 & 0 \\ \sigma & 2-2\sigma & \sigma & 0 \\ 0 & \sigma & 2-2\sigma & \sigma \\ 0 & 0 & \sigma & 2-2\sigma \end{bmatrix} \begin{bmatrix} u_1^0 \\ u_2^0 \\ u_3^0 \\ u_4^0 \end{bmatrix} + \sigma \left\{ \begin{bmatrix} \alpha^0 \\ 0 \\ 0 \\ \beta^0 \end{bmatrix} + \begin{bmatrix} \alpha^1 \\ 0 \\ 0 \\ \beta^1 \end{bmatrix} \right\} \right) = \begin{bmatrix} u_1^1 \\ u_2^1 \\ u_3^1 \\ u_4^1 \end{bmatrix}$$

General case In general, if we choose $\Delta x = (x_{\max} - x_{\min})/M$, we can express our system of equations as a

$$\begin{bmatrix} 2+2\sigma & -\sigma & 0 & \dots & 0 \\ -\sigma & 2+2\sigma & -\sigma & \ddots & \vdots \\ 0 & -\sigma & 2+2\sigma & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & -\sigma \\ 0 & \dots & 0 & -\sigma & 2+2\sigma \end{bmatrix}^{-1} \left(\begin{bmatrix} 2-2\sigma & \sigma & 0 & \dots & 0 \\ \sigma & 2-2\sigma & \sigma & \ddots & \vdots \\ 0 & \sigma & 2-2\sigma & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & \sigma \\ 0 & \dots & 0 & \sigma & 2-2\sigma \end{bmatrix} \begin{bmatrix} u_1^n \\ u_2^n \\ u_3^n \\ \vdots \\ u_{M-1}^n \end{bmatrix} + \sigma \left\{ \begin{bmatrix} \alpha^n \\ 0 \\ \vdots \\ 0 \\ \beta^n \end{bmatrix} + \begin{bmatrix} \alpha^{n+1} \\ 0 \\ \vdots \\ 0 \\ \beta^{n+1} \end{bmatrix} \right\} \right) = \begin{bmatrix} u_1^{n+1} \\ u_2^{n+1} \\ u_3^{n+1} \\ \vdots \\ u_{M-1}^{n+1} \end{bmatrix}$$

which is of the form $\mathbf{A}^{-1}\left[\mathbf{B}\mathbf{u}^{\mathbf{n}} + \sigma\left(\mathbf{b}^{\mathbf{n}} + \mathbf{b}^{\mathbf{n+1}}\right)\right] = \mathbf{u}^{\mathbf{n+1}}$, where $\mathbf{b}^{\mathbf{n}}$ is the vector representing the boundary conditions at time $t = t_n$. One obtains the solution of the PDE over time by repeatedly iterating the system from $n = 0$ to $n = N - 1$ where $\Delta t = (t_{\max} - t_{\min})/N$.

Stability The Crank-Nicolson method is ***unconditionally stable***. This means that the method is stable for all choices of time and space step sizes, Δt and Δx. However, this does not mean that you can pick any Δx or Δt: there is still the issue of precision (accuracy). However, since the precision is in $\mathcal{O}(\Delta x^2 + \Delta t^2)$ as opposed to $\mathcal{O}(\Delta x^2 + \Delta t)$ for the forward and backward difference methods, you can use a bigger Δt and keep the same level of precision. As a result, Crank-Nicolson is the recommended scheme for any simple diffusion problem.

6.3 Hyperbolic PDEs

From [2, 8]. The "model" equation for hyperbolic PDEs is the wave equation

$$\frac{\mathrm{d}^2 U}{\mathrm{d}t^2} = c^2 \nabla^2 U$$

where c is the wave velocity which does not vary over space or time. In one dimension, the wave equation is simply

$$\frac{\mathrm{d}^2 U}{\mathrm{d}t^2} = c^2 \frac{\mathrm{d}^2 U}{\mathrm{d}x^2}$$

In general, for the wave equation, one is given the initial conditions $U(x,0) = f(x)$ and $\partial U(x,0)/\partial t = g(x)$ over all space at time zero. These initial conditions represent the initial displacement and initial transverse velocity of the elements of the medium over space. One is also given the boundary conditions $U(x_{\min}, t) = \alpha(t)$ and $U(x_{\max}, t) = \beta(t)$ valid at all times. For example, in the case of a wave on a string, these represent the restriction on the ends of the string and $U(x_{\min}, t) = U(x_{\max}, t) = 0$ would mean that the ends of the string are solidly attached to a wall.

Why are we doing this? We already know the solution to this equation! While numerical methods are not warranted for the one dimensional problem with simple boundary and initial conditions, the methods we'll see are applicable to higher-dimensional problems with any BCs or ICs such that a numerical method is more likely to be worthwhile.

6.3.1 Explicit finite difference method

From [2]. As with Parabolic equations, we can express our time and space derivatives as centered difference approximations such that

$$\begin{aligned} \frac{\mathrm{d}^2 U}{\mathrm{d}t^2} &= c^2 \frac{\mathrm{d}^2 U}{\mathrm{d}x^2} \\ \frac{u_i^{n-1} - 2u_i^n + u_i^{n+1}}{\Delta t^2} + \mathcal{O}(\Delta t^2) &= c^2 \, \frac{u_{i-1}^n - 2u_i^n + u_{i+1}^n}{\Delta x^2} + \mathcal{O}(\Delta x^2) \end{aligned}$$

where n is the time subscript such that $t_{n+1} = t_n + \Delta t$, and i is the space subscript such that $x_{i+1} = x_i + \Delta x$. This scheme is of order $\mathcal{O}(\Delta x^2 + \Delta t^2)$.

We can rearrange the equation to separate the most advanced time from the solution at past times as

$$\begin{aligned} \frac{u_i^{n-1} - 2u_i^n + u_i^{n+1}}{\Delta t^2} &= c^2 \, \frac{u_{i-1}^n - 2u_i^n + u_{i+1}^n}{\Delta x^2} \\ u_i^{n-1} - 2u_i^n + u_i^{n+1} &= \frac{c^2 (\Delta t)^2}{(\Delta x)^2} \, \left(u_{i-1}^n - 2u_i^n + u_{i+1}^n \right) \\ u_i^{n+1} &= \sigma \, u_{i-1}^n - 2\sigma \, u_i^n + \sigma \, u_{i+1}^n + 2 \, u_i^n - u_i^{n-1} \\ u_i^{n+1} &= \sigma \, u_{i-1}^n + (2 - 2\sigma) \, u_i^n + \sigma \, u_{i+1}^n - u_i^{n-1} \end{aligned}$$

where $\sigma = c^2 \Delta t^2 / \Delta x^2$. For $n = 0$, this expression is

$$u_i^1 = \sigma \, u_{i-1}^0 + (2 - 2\sigma) \, u_i^0 + \sigma \, u_{i+1}^0 - u_i^{-1} \; .$$

We have been given $u_i^0 = f_i$, the initial displacement of each element at time $t = 0$, but not u_i^{-1}, the initial displacement of each element at time $t = -\Delta t$. Fortunately, we can obtain u_i^{-1} from the expression for the initial velocity of each element, $\partial U/\partial t = g(x)$, by expressing this initial condition itself as a centered finite difference such that

$$\begin{aligned} \frac{\partial U}{\partial t} = g(x) &\approx \frac{u_i^1 - u_i^{-1}}{2\Delta t} = g_i \\ u_i^1 - u_i^{-1} &= 2\Delta t g_i \\ u_i^{-1} &= u_i^1 - 2\Delta t g_i \; . \end{aligned}$$

Putting this back into our equation for $n = 0$, we get

$$\begin{aligned} u_i^1 &= \sigma u_{i-1}^0 + (2-2\sigma)u_i^0 + \sigma u_{i+1}^0 - u_i^1 + 2\Delta t g_i \\ 2u_i^1 &= \sigma u_{i-1}^0 + (2-2\sigma)u_i^0 + \sigma u_{i+1}^0 + 2\Delta t g_i \\ u_i^1 &= \frac{\sigma}{2}\, u_{i-1}^0 + (1-\sigma)\, u_i^0 + \frac{\sigma}{2}\, u_{i+1}^0 + \Delta t g_i \ . \end{aligned}$$

This means that we'll be using

$$\begin{aligned} n = 0: \quad u_i^1 &= \frac{\sigma}{2}\, f_{i-1} + (1-\sigma)\, f_i + \frac{\sigma}{2}\, f_{i+1} + \Delta t g_i \\ n > 0: \quad u_i^{n+1} &= \sigma\, u_{i-1}^n + (2-2\sigma)\, u_i^n + \sigma\, u_{i+1}^n - u_i^{n-1} \end{aligned}$$

where $f_i = f(x_i)$ and $g_i = g(x_i)$ are our initial conditions for displacement and transverse velocity of the elements of the medium at time $t = 0$.

Example case To help understand how to use this expression, let us look at the particular cases where we have a spatial step size $\Delta x = (x_{\max} - x_{\min})/5$ such that our range $[x_{\min}, x_{\max}]$ at time step n is represented by $u_0^n, u_1^n, u_2^n, u_3^n, u_4^n$, and u_5^n. This means that $u_0^n = U(x_{\min}, t) = \alpha^n$ and $u_5^n = U(x_{\max}, t) = \beta^n$, our boundary conditions at time $t = t_n$, and we know $u_i^0 = f_i$ for all i as this is our initial conditions for displacement.

Our first step ($n = 0$) is simple as it is simply

$$\begin{bmatrix} 1-\sigma & \sigma/2 & 0 & 0 \\ \sigma/2 & 1-\sigma & \sigma/2 & 0 \\ 0 & \sigma/2 & 1-\sigma & \sigma/2 \\ 0 & 0 & \sigma/2 & 1-\sigma \end{bmatrix} \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \end{bmatrix} + \frac{\sigma}{2} \begin{bmatrix} \alpha^0 \\ 0 \\ 0 \\ \beta^0 \end{bmatrix} + \Delta t \begin{bmatrix} g_1 \\ g_2 \\ g_3 \\ g_4 \end{bmatrix} = \begin{bmatrix} u_1^1 \\ u_2^1 \\ u_3^1 \\ u_4^1 \end{bmatrix}$$

Similarly, our next step ($n = 1$) will be

$$\begin{bmatrix} 2-2\sigma & \sigma & 0 & 0 \\ \sigma & 2-2\sigma & \sigma & 0 \\ 0 & \sigma & 2-2\sigma & \sigma \\ 0 & 0 & \sigma & 2-2\sigma \end{bmatrix} \begin{bmatrix} u_1^1 \\ u_2^1 \\ u_3^1 \\ u_4^1 \end{bmatrix} + \sigma \begin{bmatrix} \alpha^1 \\ 0 \\ 0 \\ \beta^1 \end{bmatrix} - \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \end{bmatrix} = \begin{bmatrix} u_1^2 \\ u_2^2 \\ u_3^2 \\ u_4^2 \end{bmatrix}$$

General case In general, if we choose $\Delta x = (x_{\max} - x_{\min})/M$, we can express our system of equations for the first step ($n = 0$) as

$$\begin{bmatrix} 1-\sigma & \sigma/2 & 0 & \dots & 0 \\ \sigma/2 & 1-\sigma & \sigma/2 & \ddots & \vdots \\ 0 & \sigma/2 & 1-\sigma & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & \sigma/2 \\ 0 & \dots & 0 & \sigma/2 & 1-\sigma \end{bmatrix} \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ \vdots \\ f_{M-1} \end{bmatrix} + \frac{\sigma}{2} \begin{bmatrix} \alpha^0 \\ 0 \\ \vdots \\ 0 \\ \beta^0 \end{bmatrix} + \Delta t \begin{bmatrix} g_1 \\ g_2 \\ g_3 \\ \vdots \\ g_{M-1} \end{bmatrix} = \begin{bmatrix} u_1^1 \\ u_2^1 \\ u_3^1 \\ \vdots \\ u_{M-1}^1 \end{bmatrix}$$

And that for all subsequent step ($n > 0$) as

$$\begin{bmatrix} 2-2\sigma & \sigma & 0 & \dots & 0 \\ \sigma & 2-2\sigma & \sigma & \ddots & \vdots \\ 0 & \sigma & 2-2\sigma & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & \sigma \\ 0 & \dots & 0 & \sigma & 2-2\sigma \end{bmatrix} \begin{bmatrix} u_1^n \\ u_2^n \\ u_3^n \\ \vdots \\ u_{M-1}^n \end{bmatrix} + \sigma \begin{bmatrix} \alpha^n \\ 0 \\ \vdots \\ 0 \\ \beta^n \end{bmatrix} - \begin{bmatrix} u_1^{n-1} \\ u_2^{n-1} \\ u_3^{n-1} \\ \vdots \\ u_{M-1}^{n-1} \end{bmatrix} = \begin{bmatrix} u_1^{n+1} \\ u_2^{n+1} \\ u_3^{n+1} \\ \vdots \\ u_{M-1}^{n+1} \end{bmatrix}$$

which is of the form $\mathbf{A}\mathbf{u^n} + \sigma\mathbf{b^n} - \mathbf{u^{n-1}} = \mathbf{u^{n+1}}$, where $\mathbf{b^n}$ is the vector representing the boundary conditions at time $t = t_n$. One obtains the solution of the PDE over time by repeatedly iterating the system from $n = 0$ to $n = N - 1$ where $\Delta t = (t_{\max} - t_{\min})/N$.

Stability The stability criterion for the explicit finite difference method used here is

$$\sqrt{\sigma} = \frac{c\Delta t}{\Delta x} \leq 1$$

This means that one has to either sacrifice spatial or take smaller steps in order to keep the algorithm under check.

6.3.2 Implicit finite difference method

From [2]. This method is very similar to the Crank-Nicolson method we saw for the parabolic equation, but instead of averaging the second order spatial derivative at $n+1$ and n, we'll average it at $n+1$ and $n-1$, such that

$$\begin{aligned} \frac{\mathrm{d}^2U}{\mathrm{d}t^2} &= c^2\frac{\mathrm{d}^2U}{\mathrm{d}x^2} \\ \frac{u_i^{n-1} - 2u_i^n + u_i^{n+1}}{\Delta t^2} &= \frac{c^2}{2}\left[\frac{u_{i-1}^{n-1} - 2u_i^{n-1} + u_{i+1}^{n-1}}{\Delta x^2} + \frac{u_{i-1}^{n+1} - 2u_i^{n+1} + u_{i+1}^{n+1}}{\Delta x^2}\right] \end{aligned}$$

where n is the time subscript such that $t_{n+1} = t_n + \Delta t$, and i is the space subscript such that $x_{i+1} = x_i + \Delta x$.

We can rearrange the equation to separate the most advanced time from the solution at past times as

$$\begin{aligned} \frac{u_i^{n-1} - 2u_i^n + u_i^{n+1}}{\Delta t^2} &= \frac{c^2}{2}\left[\frac{u_{i-1}^{n-1} - 2u_i^{n-1} + u_{i+1}^{n-1}}{\Delta x^2} + \frac{u_{i-1}^{n+1} - 2u_i^{n+1} + u_{i+1}^{n+1}}{\Delta x^2}\right] \\ u_i^{n-1} - 2u_i^n + u_i^{n+1} &= \frac{c^2\Delta t^2}{2\Delta x^2}\left[u_{i-1}^{n-1} - 2u_i^{n-1} + u_{i+1}^{n-1} + u_{i-1}^{n+1} - 2u_i^{n+1} + u_{i+1}^{n+1}\right] \\ 2u_i^{n-1} - 4u_i^n + 2u_i^{n+1} &= \sigma\ u_{i-1}^{n-1} - 2\sigma\ u_i^{n-1} + \sigma\ u_{i+1}^{n-1} + \sigma\ u_{i-1}^{n+1} - 2\sigma\ u_i^{n+1} + \sigma\ u_{i+1}^{n+1} \\ -\sigma\ u_{i-1}^{n+1} + (2+2\sigma)\ u_i^{n+1} - \sigma u_{i+1}^{n+1} &= \sigma\ u_{i-1}^{n-1} - (2+2\sigma)\ u_i^{n-1} + \sigma\ u_{i+1}^{n-1} + 4u_i^n \end{aligned}$$

where $\sigma = c^2\Delta t^2/\Delta x^2$. As with the explicit method, for $n = 0$, since we don't have u_i^{-1}, we'll have to use the finite difference approximation $u_i^{-1} = u_i^1 - 2\Delta t g_i$ such that

$$\begin{aligned}
-\sigma\ u_{i-1}^1 + (2+2\sigma)\ u_i^1 - \sigma u_{i+1}^1 &= \sigma\left(u_{i-1}^1 - 2\Delta t g_{i-1}\right) - (2+2\sigma)\left(u_i^1 - 2\Delta t g_i\right) + \sigma\left(u_{i+1}^1 - 2\Delta t g_{i+1}\right) + 4u_i^0 \\
-2\sigma\ u_{i-1}^1 + (4+4\sigma)\ u_i^1 - 2\sigma u_{i+1}^1 &= -2\sigma\Delta t g_{i-1} + (4+4\sigma)\Delta t g_i - 2\sigma\Delta t g_{i+1} + 4u_i^0 \\
-\sigma\ u_{i-1}^1 + (2+2\sigma)\ u_i^1 - \sigma u_{i+1}^1 &= -\sigma\Delta t\ g_{i-1} + (2+2\sigma)\Delta t\ g_i - \sigma\Delta t\ g_{i+1} + 2\ u_i^0
\end{aligned}$$

Since we are now quite experienced, we'll have no problem expressing the matrix for the general case directly.

General case In general, if we choose $\Delta x = (x_{\max} - x_{\min})/M$, we can express the first line ($i = 1$) for the case $n = 0$ as

$$\begin{aligned}
n = 0, i = 1: \quad -\sigma\ u_0^1 + (2+2\sigma)\ u_1^1 - \sigma u_2^1 &= -\sigma\Delta t\ g_0 + (2+2\sigma)\Delta t\ g_1 - \sigma\Delta t\ g_2 + 2\ u_1^0 \\
-\sigma\ \alpha^1 + (2+2\sigma)\ u_1^1 - \sigma u_2^1 &= -\sigma\Delta t\ g_0 + (2+2\sigma)\Delta t\ g_1 - \sigma\Delta t\ g_2 + 2\ f_1
\end{aligned}$$

which will turn into the matrix

$$\begin{bmatrix}
(2+2\sigma) & -\sigma & 0 & \dots & 0 \\
-\sigma & (2+2\sigma) & -\sigma & \ddots & \vdots \\
0 & -\sigma & (2+2\sigma) & \ddots & 0 \\
\vdots & \ddots & \ddots & \ddots & -\sigma \\
0 & \dots & 0 & -\sigma & (2+2\sigma)
\end{bmatrix}
\begin{bmatrix} u_1^1 \\ u_2^1 \\ u_3^1 \\ \vdots \\ u_{M-1}^1 \end{bmatrix} =$$

$$\begin{bmatrix}
(2+2\sigma) & -\sigma & 0 & \dots & 0 \\
-\sigma & (2+2\sigma) & -\sigma & \ddots & \vdots \\
0 & -\sigma & (2+2\sigma) & \ddots & 0 \\
\vdots & \ddots & \ddots & \ddots & -\sigma \\
0 & \dots & 0 & -\sigma & (2+2\sigma)
\end{bmatrix}
\begin{bmatrix} g_1 \\ g_2 \\ g_3 \\ \vdots \\ g_{M-1} \end{bmatrix} \Delta t + 2 \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ \vdots \\ f_{M-1} \end{bmatrix} + \sigma \begin{bmatrix} \alpha^1 \\ 0 \\ \vdots \\ 0 \\ \beta^1 \end{bmatrix} - \sigma\Delta t \begin{bmatrix} g_0 \\ 0 \\ \vdots \\ 0 \\ g_M \end{bmatrix}$$

Then, we can express the first line ($i = 1$) for the case $n > 0$ as

$$\begin{aligned}
n > 0, i = 1: \quad -\sigma\ u_0^{n+1} + (2+2\sigma)\ u_1^{n+1} - \sigma u_2^{n+1} &= \sigma\ u_0^{n-1} - (2+2\sigma)u_1^{n-1} + \sigma\ u_2^{n-1} + 4u_1^n \\
-\sigma\ \alpha^{n+1} + (2+2\sigma)\ u_1^{n+1} - \sigma u_2^{n+1} &= \sigma\ \alpha^{n-1} - (2+2\sigma)u_1^{n-1} + \sigma\ u_2^{n-1} + 4u_1^n
\end{aligned}$$

which will turn into the matrix

$$\begin{bmatrix} (2+2\sigma) & -\sigma & 0 & \dots & 0 \\ -\sigma & (2+2\sigma) & -\sigma & \ddots & \vdots \\ 0 & -\sigma & (2+2\sigma) & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & -\sigma \\ 0 & \dots & 0 & -\sigma & (2+2\sigma) \end{bmatrix} \begin{bmatrix} u_1^{n+1} \\ u_2^{n+1} \\ u_3^{n+1} \\ \vdots \\ u_{M-1}^{n+1} \end{bmatrix} =$$

$$\begin{bmatrix} -(2+2\sigma) & \sigma & 0 & \dots & 0 \\ \sigma & -(2+2\sigma) & \sigma & \ddots & \vdots \\ 0 & \sigma & -(2+2\sigma) & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & \sigma \\ 0 & \dots & 0 & \sigma & -(2+2\sigma) \end{bmatrix} \begin{bmatrix} u_1^{n-1} \\ u_2^{n-1} \\ u_3^{n-1} \\ \vdots \\ u_{M-1}^{n-1} \end{bmatrix} + 4 \begin{bmatrix} u_1^{n} \\ u_2^{n} \\ u_3^{n} \\ \vdots \\ u_{M-1}^{n} \end{bmatrix} + \sigma \left(\begin{bmatrix} \alpha^{n-1} \\ 0 \\ \vdots \\ 0 \\ \beta^{n-1} \end{bmatrix} + \begin{bmatrix} \alpha^{n+1} \\ 0 \\ \vdots \\ 0 \\ \beta^{n+1} \end{bmatrix} \right)$$

6.4 Elliptic PDEs

From [8]. Elliptic PDEs typically model steady states. For example, the elliptic PDE describes the steady state distribution of heat on a plane region whose boundary is being held at specific temperatures, or the value of the electric potential over a plane region at the boundary of which a potential difference is applied.

The "model" equation for elliptic PDEs is the Poisson equation

$$\nabla^2 U = \rho(\vec{x}) \ .$$

When $\rho(\vec{x}) = 0$ the Poisson equation is referred to as Laplace's equation.

6.4.1 The finite difference method

For simplicity, we will operate in 2-D cartesian coordinate and write

$$\frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2} = \rho(x, y) \ . \tag{3}$$

To solve our 2-D PDE, we divide space up using a finite difference approximation as

$$\frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{\Delta x^2} + \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{\Delta y^2} = \rho_{i,j} \ .$$

where $\Delta x = (x_{\max} - x_{\min})/M$, $\Delta y = (y_{\max} - y_{\min})/N$, and $\rho_{i,j} = \rho(x_{\min} + i\Delta x, y_{\min} + j\Delta y)$. This means that we have subdivided the space of the problem into an $M \times N$ grid point mesh. Of these $M \times N$ points, we know the value of the function at $2(M + N)$ of these points and are seeking the value of the function at the remaining $(M - 1) \times (N - 1)$ points. For simplicity, let's define $m = M - 1$ and $n = N - 1$.

By setting $\sigma = \Delta x^2/\Delta y^2$ in the above, we can simplify it to

$$\begin{aligned} u_{i-1,j} - 2u_{i,j} + u_{i+1,j} + \sigma \left(u_{i,j-1} - 2u_{i,j} + u_{i,j+1} \right) &= \Delta x^2 \rho_{i,j} \\ \sigma \ u_{i,j-1} + u_{i-1,j} - 2(\sigma + 1) \ u_{i,j} + u_{i+1,j} + \sigma \ u_{i,j+1} &= \Delta x^2 \rho_{i,j} \end{aligned}$$

The $m \times n$ points where we seek the value of the function can be subdivided into 3 types:

corners These are the only points in contact with 2 boundaries, namely $u_{1,1}, u_{m,1}, u_{1,n}, u_{m,n}$. There are 4 such points.

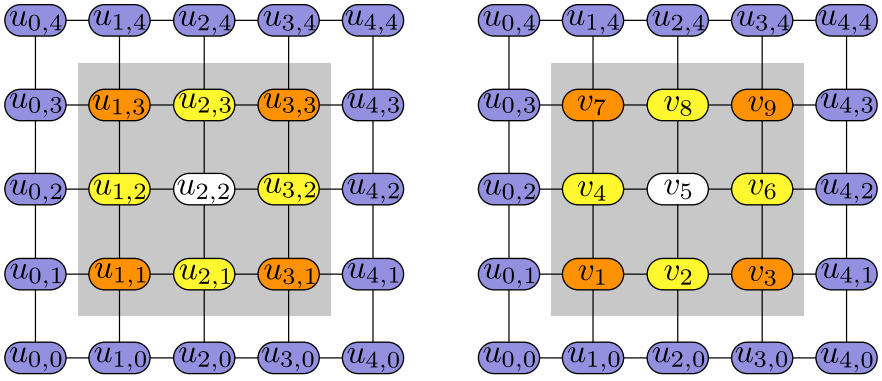
edges These are the points in contact with 1 boundary, namely $u_{i,1}, u_{i,n}, u_{1,j}, u_{m,j}$ where $1 < i < m$ and $1 < j < n$. There are $2(m+n) - 8$ such points.

cores These are the points that are not in contact with a boundary, namely $u_{i,j}$ where $1 < i < m$ and $1 < j < n$. There are $(m-2) \times (n-2)$ such points.

Example case In order to better illustrate how this will all come together, let us apply to finite difference method with $M = N = 4$ to approximate the solution of the Laplace equation $\nabla^2 U = 0$ for $x \in [0, 1]$ and $y \in [1, 2]$ with the following Dirichlet boundary conditions:

$$\begin{aligned} U(0, y) &= 2\ln(y) & \qquad U(x, 1) &= \ln(x^2 + 1) \\ U(1, y) &= \ln(y^2 + 1) & \qquad U(x, 2) &= \ln(x^2 + 4) \end{aligned}$$

The exact solution is $U(x, y) = \ln(x^2 + y^2)$. For our choice of M and N, we have 9 unknowns: 4 corners, 4 edges, and 1 core. Let us look at the grid representing this problem.



In order to make it easier to handle indices and to treat the system as a matrix, it is easier to express all unknowns as a single 1-D vector. To facilitate this, we will use the alternate notation illustrated in the figure, namely $u_{i,j} = v_k$ where $k = i + m(j-1)$, such that the finite difference equation becomes

$$\begin{aligned} \sigma\ u_{i,j-1} + u_{i-1,j} - 2(\sigma+1)\ u_{i,j} + u_{i+1,j} + \sigma\ u_{i,j+1} &= \Delta x^2 \rho_{i,j} \\ \sigma\ v_{i+m(j-2)} + v_{i-1+m(j-1)} - 2(\sigma+1)\ v_{i+m(j-1)} + v_{i+1+m(j-1)} + \sigma\ v_{i+mj} &= \Delta x^2 \rho_{i+m(j-1)} \end{aligned}$$

where $\vec{v}$ is a vector of length $m \times n$. The system of equation for the 4 corner points is

$$
\begin{aligned}
\text{bottom corners}: \; u_{1,1},\ v_1: \quad \sigma\ u_{1,0} + u_{0,1} - 2(\sigma+1)\ u_{1,1} + u_{2,1} + \sigma\ u_{1,2} &= \Delta x^2 \rho_{1,1} \\
-2(\sigma+1)\ u_{1,1} + u_{2,1} + \sigma\ u_{1,2} &= \Delta x^2 \rho_{1,1} - \sigma\ u_{1,0} - u_{0,1} \\
-2(\sigma+1)\ v_1 + v_2 + \sigma\ v_4 &= \Delta x^2 \rho_{1,1} - \sigma\ b(x_1) - l(y_1) \\
u_{3,1},\ v_3: \quad \sigma\ u_{3,0} + u_{2,1} - 2(\sigma+1)\ u_{3,1} + u_{4,1} + \sigma\ u_{3,2} &= \Delta x^2 \rho_{3,1} \\
u_{2,1} - 2(\sigma+1)\ u_{3,1} + \sigma\ u_{3,2} &= \Delta x^2 \rho_{3,1} - \sigma\ u_{3,0} - u_{4,1} \\
v_2 - 2(\sigma+1)\ v_3 + \sigma\ v_6 &= \Delta x^2 \rho_{3,1} - \sigma\ b(x_3) - r(y_1) \\
\text{top corners}: \; u_{1,3},\ v_7: \quad \sigma\ u_{1,2} + u_{0,3} - 2(\sigma+1)\ u_{1,3} + u_{2,3} + \sigma\ u_{1,4} &= \Delta x^2 \rho_{1,3} \\
\sigma\ u_{1,2} - 2(\sigma+1)\ u_{1,3} + u_{2,3} &= \Delta x^2 \rho_{1,3} - u_{0,3} - \sigma\ u_{1,4} \\
\sigma\ v_4 - 2(\sigma+1)\ v_7 + v_8 &= \Delta x^2 \rho_{1,3} - l(y_3) - \sigma\ t(x_1) \\
u_{3,3},\ v_9: \quad \sigma\ u_{3,2} + u_{2,3} - 2(\sigma+1)\ u_{3,3} + u_{4,3} + \sigma\ u_{3,4} &= \Delta x^2 \rho_{3,3} \\
\sigma\ u_{3,2} + u_{2,3} - 2(\sigma+1)\ u_{3,3} &= \Delta x^2 \rho_{3,3} - u_{4,3} - \sigma\ u_{3,4} \\
\sigma\ v_6 + v_8 - 2(\sigma+1)\ v_9 &= \Delta x^2 \rho_{3,3} - r(y_3) - \sigma\ t(x_3)
\end{aligned}
$$

where $b(x)$, $t(x)$, $l(y)$, $r(y)$ are the functions representing the BCs at the bottom, top, left, and right edge of the domain, respectively.

For the 4 edges we have

$$
\begin{aligned}
\text{bottom edge}: \; u_{2,1},\ v_2: \quad \sigma\ u_{2,0} + u_{1,1} - 2(\sigma+1)\ u_{2,1} + u_{3,1} + \sigma\ u_{2,2} &= \Delta x^2 \rho_{2,1} \\
u_{1,1} - 2(\sigma+1)\ u_{2,1} + u_{3,1} + \sigma\ u_{2,2} &= \Delta x^2 \rho_{2,1} - \sigma\ u_{2,0} \\
v_1 - 2(\sigma+1)\ v_2 + v_3 + \sigma\ v_5 &= \Delta x^2 \rho_{2,1} - \sigma\ b(x_2) \\
\text{left edge}: \; u_{1,2},\ v_4: \quad \sigma\ u_{1,1} + u_{0,2} - 2(\sigma+1)\ u_{1,2} + u_{2,2} + \sigma\ u_{1,3} &= \Delta x^2 \rho_{1,2} \\
\sigma\ u_{1,1} - 2(\sigma+1)\ u_{1,2} + u_{2,2} + \sigma\ u_{1,3} &= \Delta x^2 \rho_{1,2} - u_{0,2} \\
\sigma\ v_1 - 2(\sigma+1)\ v_4 + v_5 + \sigma\ v_7 &= \Delta x^2 \rho_{1,2} - l(y_2) \\
\text{right edge}: \; u_{3,2},\ v_6: \quad \sigma\ u_{3,1} + u_{2,2} - 2(\sigma+1)\ u_{3,2} + u_{4,2} + \sigma\ u_{3,3} &= \Delta x^2 \rho_{3,2} \\
\sigma\ u_{3,1} + u_{2,2} - 2(\sigma+1)\ u_{3,2} + \sigma\ u_{3,3} &= \Delta x^2 \rho_{3,2} - u_{4,2} \\
\sigma\ v_3 + v_5 - 2(\sigma+1)\ v_6 + \sigma\ v_9 &= \Delta x^2 \rho_{3,2} - l(y_2) \\
\text{top edge}: \; u_{2,3},\ v_8: \quad \sigma\ u_{2,2} + u_{1,3} - 2(\sigma+1)\ u_{2,3} + u_{3,3} + \sigma\ u_{2,4} &= \Delta x^2 \rho_{2,3} \\
\sigma\ u_{2,2} + u_{1,3} - 2(\sigma+1)\ u_{2,3} + u_{3,3} &= \Delta x^2 \rho_{2,3} - \sigma\ u_{2,4} \\
\sigma\ v_5 + v_7 - 2(\sigma+1)\ v_8 + v_9 &= \Delta x^2 \rho_{2,3} - \sigma\ t(x_2)
\end{aligned}
$$

And finally for the core point we have

$$
\begin{aligned}
\text{core point}: \; u_{2,2},\ v_5: \quad \sigma\ u_{2,1} + u_{1,2} - 2(\sigma+1)\ u_{2,2} + u_{3,2} + \sigma\ u_{2,3} &= \Delta x^2 \rho_{2,2} \\
\sigma\ v_2 + v_4 - 2(\sigma+1)\ v_5 + v_6 + \sigma\ v_8 &= \Delta x^2 \rho_{2,2}
\end{aligned}
$$

With this, we are now ready to construct our matrix

$$\begin{bmatrix} -4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & -4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & -4 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & -4 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & -4 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & -4 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & -4 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & -4 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & -4 \end{bmatrix} \begin{bmatrix} u_{1,1} \\ u_{2,1} \\ u_{3,1} \\ u_{1,2} \\ u_{2,2} \\ u_{3,2} \\ u_{1,3} \\ u_{2,3} \\ u_{3,3} \end{bmatrix} = \begin{bmatrix} -\sigma\ b(x_1) - l(y_1) \\ -\sigma\ b(x_2) \\ -\sigma\ b(x_3) - r(y_1) \\ -l(y_2) \\ 0 \\ -r(y_2) \\ -l(y_3) - \sigma\ t(x_1) \\ -\sigma\ t(x_2) \\ -r(y_3) - \sigma\ t(x_3) \end{bmatrix} = \begin{bmatrix} -0.5069 \\ -0.2231 \\ -1.3873 \\ -0.8109 \\ 0 \\ -1.1787 \\ -2.5210 \\ -1.4469 \\ -2.9197 \end{bmatrix}$$

where $b(x)$, $t(x)$, $l(y)$, $r(y)$ are the functions representing the BCs at the bottom, top, left, and right edge of the domain, respectively. If you've done everything right, you should get the following solution for the above:

$$\begin{bmatrix} -4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & -4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & -4 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & -4 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & -4 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & -4 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & -4 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & -4 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & -4 \end{bmatrix}^{-1} \begin{bmatrix} -0.5069 \\ -0.2231 \\ -1.3873 \\ -0.8109 \\ 0 \\ -1.1787 \\ -2.5210 \\ -1.4469 \\ -2.9197 \end{bmatrix} = \begin{bmatrix} u_{1,1} \\ u_{2,1} \\ u_{3,1} \\ u_{1,2} \\ u_{2,2} \\ u_{3,2} \\ u_{1,3} \\ u_{2,3} \\ u_{3,3} \end{bmatrix} = \begin{bmatrix} 0.4847 \\ 0.5944 \\ 0.7539 \\ 0.8376 \\ 0.9159 \\ 1.0341 \\ 1.1390 \\ 1.1974 \\ 1.2878 \end{bmatrix} \quad \underbrace{\begin{bmatrix} 0.4855 \\ 0.5947 \\ 0.7538 \\ 0.8383 \\ 0.9163 \\ 1.0341 \\ 1.1394 \\ 1.1977 \\ 1.2879 \end{bmatrix}}_{\text{true soln}}$$

which, as you can see, is very close to the true solution.

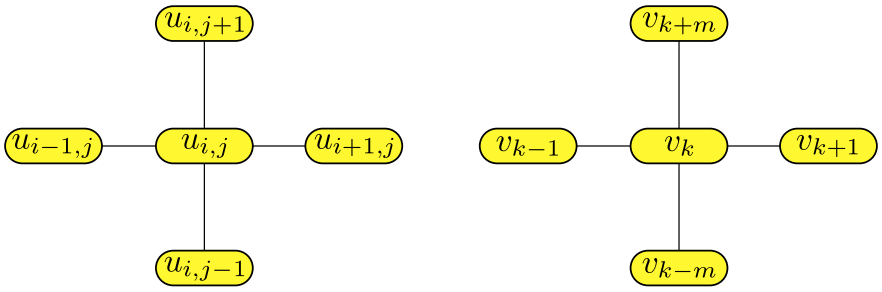
General case In general, it is better to think of the meaning of each of the 5 non-zero diagonals of the matrix. There are 3 sets of diagonals:

main diagonal: This diagonal represent the central site.

off-by-1 diagonals: These 2 diagonals on either side of the main diagonal represent the left (-1) and right ($+1$) neighbours of the central site.

off-by-m diagonals: These 2 diagonals represent the bottom ($-m$) and top ($+m$) neighbours of the central site.

This can be illustrated as follows:



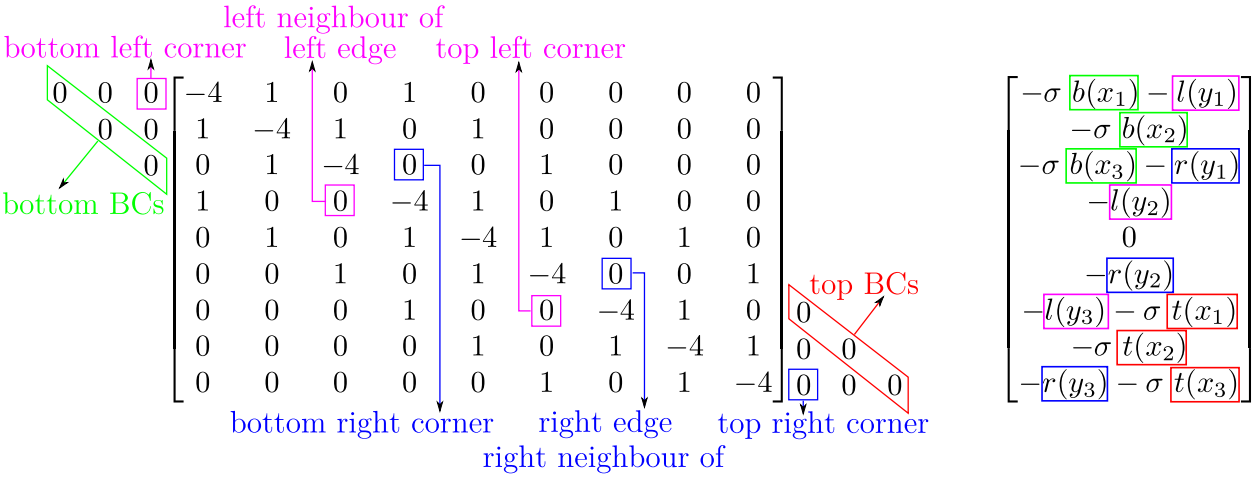
Our finite difference equation can be expressed as

$$\sigma\ v_{k-m} + v_{k-1} - 2(\sigma+1)\ v_k + v_{k+1} + \sigma\ v_{k+m} = \Delta x^2 \rho_k\ ,$$

and we can contruct the matrix as 5 different diagonals:

neighbour	diagonal	value	size of `vec`	index k of BCs
bottom	`diag(vec,-m)`	σ	$m \times n - m$	`[1:m]`
left	`diag(vec,-1)`	1	$m \times n - 1$	`[1:m:(m*n)-m+1]`
centre	`diag(vec,0)`	$-2(\sigma+1)$	$m \times n$	none
right	`diag(vec,1)`	1	$m \times n - 1$	`[m:m:m*n]`
top	`diag(vec,m)`	σ	$m \times n - m$	`[(n-1)*m+1:m*n]`

Perhaps it will be clearer if I illustrate this using the matrix and solution vector from our example



You can clearly see that on each row (for each k) where v_k has one or more BCs, we put a zero in the diagonal and subtract the BC from the solution vector. It takes some getting used to, but it's pretty straight forward. The full Octave code to do this is shown in Figure 4.

7 ODE modelling of real systems

System of ODEs are commonly used to represent complex dynamical systems to try and understand how they work, and how we can manipulate them to our advantage. For example, ODE models are the cornerstone of immunological, ecological, and biological modelling. My research being in the area of modelling the spread of infectious disease within a host or cell culture, I will be using this as an example.

7.1 ODEs as models for an in-host viral infection

When using ODE to represent a true system, much thought has to go into writing out the ODE system, and understanding the underlying assumption of the chosen model. In general, you want your model to be sufficiently complex to capture the behaviour you are interested in, but not so complicated that you have to make many guesses as to how some of its mechanisms work. As Albert Einstein said it:

```
#!/usr/bin/octave -qf

xmin = 0; xmax = 1; M = 4; dx = (xmax-xmin)/M; m = M-1;
ymin = 1; ymax = 2; N = 4; dy = (ymax-ymin)/N; n = N-1;
sigma = (dx/dy)^2;

# Convert k index to [i j] index
function ind = k2ij(m,k)
   j = ceil(k/m);
   ind = [ k-m*(j-1); j]';
end;

# Value of function at each boundary
lbc = inline('2.*log(y)');   # left BC @ xmin
rbc = inline('log(y.^2+1)'); # right BC @ xmax
bbc = inline('log(x.^2+1)'); # bottom BC @ ymin
tbc = inline('log(x.^2+4)'); # top BC @ ymax

# Building the result vector without the BCs
bc = zeros(m*n,1); # Only true for rho(x,y) = 0

##### Building left neighbour diagonal & result vector
k = [1:m:(m*n)-m+1];
lvec = ones(m*n-1,1);   lvec(k(2:end)-1) = zeros(n-1,1);
bc(k) -= lbc( ymin + dy.*k2ij(m,k)(:,2) );
##### Building right neighbour diagonal & result vector
k = [m:m:m*n];
rvec = ones(m*n-1,1);   rvec(k(1:end-1)) = zeros(n-1,1);
bc(k) -= rbc( ymin + dy.*k2ij(m,k)(:,2) );
##### Building bottom neighbour diagonal & result vector
k = [1:m];
bvec = sigma*ones(m*n-m,1);
bc(k) -= sigma*bbc( xmin + dx.*k2ij(m,k)(:,1) );
##### Building top neighbour diagonal & result vector
k = [(n-1)*m+1:m*n];
tvec = sigma*ones(m*n-m,1);
bc(k) -= sigma*tbc( xmin + dx.*k2ij(m,k)(:,1) );
##### Building main (central) diagonal
cvec = -2*(1+sigma).*ones(m*n,1);

# Building the matrix by putting it all together
A = diag(cvec,0) + diag(lvec,-1) + diag(rvec,1) + diag(bvec,-m) + diag(tvec,m);

# The solution everywhere in space is simply given by
res = A \ bc;
```

Figure 4: Code to solve the elliptic Laplace equation for our example case.

"Things should be made as simple as possible, but not any simpler."

For example, a very commonly used model for a viral infection is:

$$\begin{aligned} \text{target cells}: \quad \frac{\mathrm{d}T}{\mathrm{d}t} &= -\beta TV \\ \text{infected cells}: \quad \frac{\mathrm{d}I}{\mathrm{d}t} &= \beta TV - \delta I \\ \text{virus}: \quad \frac{\mathrm{d}V}{\mathrm{d}t} &= pI - cV \end{aligned}$$

The model is made-up of 3 different "species":

target cells, T : These cells are the target of the infection. The more there are, and the more virus (V) there is, the more target cells become infected (I).

infected cells, I : When a virus infects a cell, the cell is converted from a target cell (T) to an infected cell (I), at a rate which depends on how many cells are available for infection, T, how many virus is around, V, and the rate at which a virus can successfully infect a cell, β. Once infected, cytotoxic effects from viral production as well as potential viral attacks not explicitly represented in the model will lead to the infected cells dying at a rate δ.

virus, V : The virions, or virus particles, are the agents responsible for the infection of target cells ($T \to I$). They are produced at a certain rate (p) by infected cells and are cleared out of the system by immune responses (e.g., antibodies, mucus, fever) at a rate c.

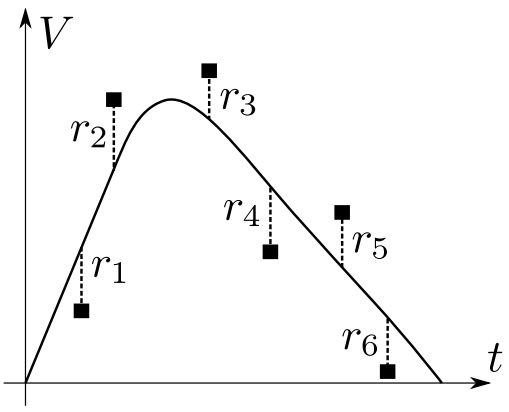
Quiz: What does the $-\delta I$ and $-cV$ terms represent? [Tip: think about radioactive decay, $\mathrm{d}N/\mathrm{d}t = -\lambda N$]

Quiz: Do you think these 2 terms make sense?

What are the assumptions of the infection model?

- All cells see all virus, and vice-versa. This is only true in a well-mixed system. This means that the model assumes very high diffusion and no spatial heterogeneity.
- Target cells do not divide or die in significant numbers over the course of the infection.
- Either the immune system plays an insignificant role, or its action is sufficiently constant to be taken into account implicitly by the δI and cV terms.

7.2 Comparing a model against experimental data



Once you have formulated a model you think is appropriate for your problem, you need to compare it against your experimental data. There are two important measures of how good you model fits the data:

SSR or sum of squared residuals is the sum of the squared difference between your model's curve and your experimental data points over all data points.

It gives you a measure of how good your fit matches the experimental data. Namely, it is given by

$$\mathrm{SSR} = \sum_{i=1}^{N} (y_{\mathrm{exp},i} - y_{\mathrm{model},i})^2 = \sum_{i=1}^{N} r_i^2$$

AIC or Akaike's "An Information Criterion" is a measure that allows you to compare two different models with a different number of parameters. While in general the more parameters you have, the better a fit you'll get, there is a limit to how many parameters your experimental data can support/set. The AIC is a measure of a model's agreement with experimental data which takes into consideration the SSR, but adds a penalty for additional parameters as a function of the number of experimental data points available. The general expression for the AIC corrected for small sample sizes (the $\mathrm{AIC_C}$) is given by

$$\mathrm{AIC_C} = N_{\mathrm{pts}} \log_{\mathrm{e}} \left(\frac{\mathrm{SSR}}{N_{\mathrm{pts}}} \right) + \frac{2(N_{\mathrm{par}} + 1) N_{\mathrm{pts}}}{N_{\mathrm{pts}} - N_{\mathrm{par}} - 2} \ ,$$

where N_{par} is the number of parameters of the fitted model and the +1 term arises because we estimate the variance as $\sigma^2 = \mathrm{SSR}/N_{\mathrm{pts}}$, where N_{pts} is the number of data points fitted by the model, and SSR is the sum of squared residuals [1]. The model with the lowest $\mathrm{AIC_C}$ is considered to be the better model given the experimental data it is approximating.

There are two things you will typically be seeking when using these measures:

1. To compare different proposed models
2. To determine the values of a model's parameters

Say you are trying to figure out a model for how a fish's age depends on its length. For whatever reason, it finally came down to two possible models:

$$\mathrm{Age} = p_1 L^{p_2} \quad \text{and} \quad \mathrm{Age} = p_3^L$$

You don't know which model is most appropriate, and to make matters worse, you don't know the values of any of the parameters, p_i. What to do?

1. For each model, find the parameters that will cause it to best fit the data (minimize the SSR).
2. Once you've got your best-fit parameter values, and your minimum SSR for each model, compare the $\mathrm{AIC_C}$ obtained with each and the one with the smallest $\mathrm{AIC_C}$ is the model best supported by your data.

CAUTION: The model with the smallest $\mathrm{AIC_C}$ is NOT NECESSARILY the best model. It is the model best supported by the data. If you have very limited data, you have to keep the number of parameters of your model very small. This means that if the correct model has more parameter than can be justified by the numbe of data points, the $\mathrm{AIC_C}$ may indicate that a less correct model with fewer parameters is statistically better supported by the data.

7.3 Fitting a model against experimental data

In the model I introduced above for viral infections, I had 3 variables (T, I, and V) and 4 parameters, β, δ, p, and c, which represent the infection rate of target cells by virus, the death rate of infected cells, the production rate of virus per infected cell, and the clearance rate of virus, respectively. Well, as it turns out, for influenza, we have no idea what the value of any of these parametres might be. To try and find out, we try to pick parameters to minimize the SSR. This is essentially a minimization problem. In a sense, we are trying to find β, δ, p, c such that $\text{SSR}(\beta, \delta, p, c)$ is a minimum. Because in fact, the SSR I will obtain depends on the shape of my solution curve which in turns depends on the parameters of the model.

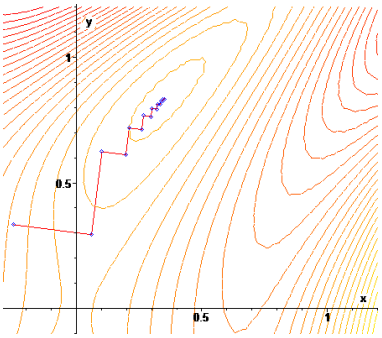
So finding the best fit parameter is analogous to finding the minimum of a function. In our case, that function is the complicated function which makes up the SSR. How do you do this?

1. Take a guess for value of each parameter (e.g., $\beta = \delta = p = c = 1$)
2. Improve your guesses for each parameter until you find (β, δ, p, c) which minimize the SSR.

This latter problem is called an optimization problem, and in this particular case it is a minimization problem.

7.3.1 A brief word on minimization methods

Our world is full of examples of problems that require that you find the optimal solution (minimum or maximum) by adjusting a set of parameters. For example, if I want to minimize the number of people who die in the next flu pandemic and I have a limited amount of drug doses, a limited number of health care workers, of hospital beds, etc., how do I best distribute my resources (e.g., give a little drug to everyone vs a lot of drug to a few key people). There are several algorithms for finding the minimum of a function, and I'll mention a few of them here briefly.

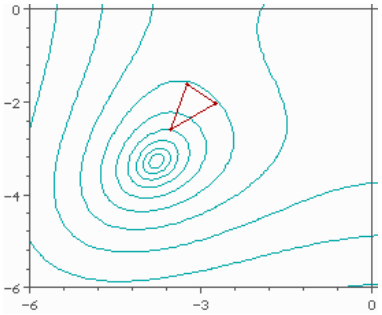


From Wikimedia

Steepest descent From Wikipedia. The method of steepest descent, also known as the gradient descent method, is a method by which the gradient of the function to be minimized is used to find its minimum (or maximum if steepest ascent). Gradient descent is based on the observation that if function $f(\vec{x})$ is defined and differentiable in a neighborhood of a point $\vec{x_0}$, then $f(\vec{x})$ decreases fastest if one goes from $\vec{x_0}$ in the direction opposite to the gradient, namely the direction of the negative of the gradient, such that

$$\vec{x}_{n+1} = \vec{x}_n - \gamma_n \nabla f(\vec{x}_n), \quad n \geq 0.$$

so hopefully the sequence $(\vec{x}_n)$ converges to the desired local minimum. Note that the value of the step size γ_n is allowed to change at every iteration.



From Wikimedia

Nelder-Mead or downhill simplex method From Wikipedia. The method uses the concept of a simplex, which is a polytope of $N + 1$ vertices in N-dimensions. Examples of simplexes include a line segment on a line, a triangle on a plane, a tetrahedron in three-dimensional space and so forth. The simplest step is to replace

the worst point with a point reflected through the centroid of the remaining N points. If this point is better than the best current point, then we can try stretching exponentially out along this line. On the other hand, if this new point isn't much better than the previous value, then we are stepping across a valley, so we shrink the simplex towards the best point. [Go see the very nice animated gif on Wikimedia].

Minimization problems in Octave For the purpose of this course, note that the function `leasqr` in Octave is often a good choice for root-finding. It uses the Levenberg-Marquardt algorithm which, in simple terms, essentially makes use of the steepest descent method. It estimates the gradient using a finite-difference approximation, and a few other tricks. The Levenberg-Marquardt algorithm is designed especially to be used for least squares curve fitting by minimizing the SSR. Aha! Just what you needed!

8 Cellular automata

Cellular automaton (CA) models are very powerful tools commonly used in statistical mechanics (e.g. the Ising spin model [4]). What makes the beauty of CA is that through the implementation of simple local rules, one can reproduce the behaviour exhibited by the highly complex system modelled. The level of interactions involved in self-organizing systems is usually quite well described by CA.

Cellular automata were originally introduced by John von Neumann and Stanislaw Ulam under the name of "cellular spaces" as possible idealization of biological systems. They sought to show that biological processes such as the reproduction and evolution of organized forms could be modelled by simple cells following local rules for changing a cell parameter with time [3]. Traditional CA usually consist of a regular uniform N-dimensional grid that can either be finite or infinite (periodic boundaries) in extent. The grid contains a discrete variable (or cell) at each site that can assume m possible discrete values. The state of a CA is completely specified by the values of all variables at each site. The CA evolves in discrete space with discrete time steps with the value of a variable at a given site being affected by the values of variables at sites in its neighbourhood at the previous time step. The neighbourhood of a site can be defined in numerous ways and can be as extended as one wishes. At a new time step, the variables at all sites are updated based on their own value and that of their defined neighbourhood at the preceding time step according to a definite set of local rules.

8.1 Elementary One-dimensional Cellular Automata

Stephen Wolfram has done most of the early work on CA. In the '80s, he published an extensive series of articles on CA [9, 10, 11]. He concentrated most of his work on elementary CA, namely one-dimensional CA with only two possible values at each site: 0 or 1. Typically, the neighbourhood he chose was defined to be the cell itself plus its two immediate neighbours: the cells to its left and right. This type of CA as been thoroughly studied by Wolfram.

From the definition of the neighbourhood, it follows that the local rules for the evolution of the CA will have to be defined for each combination of states of the 3 sites involved in defining the subsequent state of a single site. For the sake of clarity, let us label lc, cc and rc — for left, centre,

	patterns	rules
Type 1	disappear with time	0, 32, 72, 104, 128, 160, 200, 232
Type 2	single non-zero seed forever	4, 36, 76, 108, 132, 164, 204, 236
Type 3	uniform type of rows	50, 122, 178, 222, 250, 254
	uniform pairs of rows	54, 94
Type 4	nontrivial patterns	18, 22, 90, 126, 146, 150, 182, 218

Table 2: Classification of the 32 legal rules according to the type of patterns they developed when the one-dimensional CA is initialized with a single non-zero seed.

and right cell, respectively — the 3 cells involved in the determination of the state of cc in the next time step. The set of possible configurations of lc, cc, rc is then defined as, for example,

$$\begin{array}{cccccccccc} lc, cc, rc & 111 & 110 & 101 & 100 & 011 & 010 & 001 & 000 \\ cc & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \end{array} , \tag{4}$$

where the bottom line indicates the state of cc at the next time step given the top line's lc, cc, rc configuration at the previous step. This means that any rule can be defined by an eight digit binary number where each digit would represent the next state of the cc given its current state and that of its immediate neighbours. The rule detailed on the second line of (4) is called "Rule 90", from the binary number 01011010 defining it. Recall that a binary number is read in the following manner:

$$01011010 = 0 \cdot 2^7 + 1 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 90 \ .$$

For a binary CA based on a 3-cell local rule, there are $2^3 = 8$ possible configurations for which a result must be specified which makes for a total of $2^8 = 256$ possible rules. However, those rules are usually reduced to a total number of 32 "legal" rules. The legal rules come from the fact that the CA were initially used for modelling biological processes such as the reproduction of organized forms. This meant, for example, that a starting configuration of all-zero sites, corresponding to the absence of any cells, should remain an all-zero configuration as it evolves in time — the triplet 000 should always give 0. Secondly, the rules should be symmetric so that the left and right cells are interchangeable. This forces the rules for configurations 100 and 001 and for configurations 110 and 011 to be equal [10]. Thus, the legal rules can be written as

$$\alpha_1\alpha_2\alpha_3\alpha_4\alpha_2\alpha_5\alpha_4 0 \ , \tag{5}$$

where each α_i can be 0 or 1, hence the $2^5 = 32$ legal rules.

If one decides to start the one-dimensional elementary CA with a single non-zero seed at its centre, interesting patterns develop. Based on these patterns, the 32 legal rules can be divided into 4 types as described in Table 2. Example patterns formed by rules from each of the 4 types are presented in Figure 5.

8.2 Conway's Game of Life

From Wikipedia. The Game of Life, also known simply as Life, is a cellular automaton devised by the British mathematician John Horton Conway in 1970. It is the best-known example of a cellular automaton.

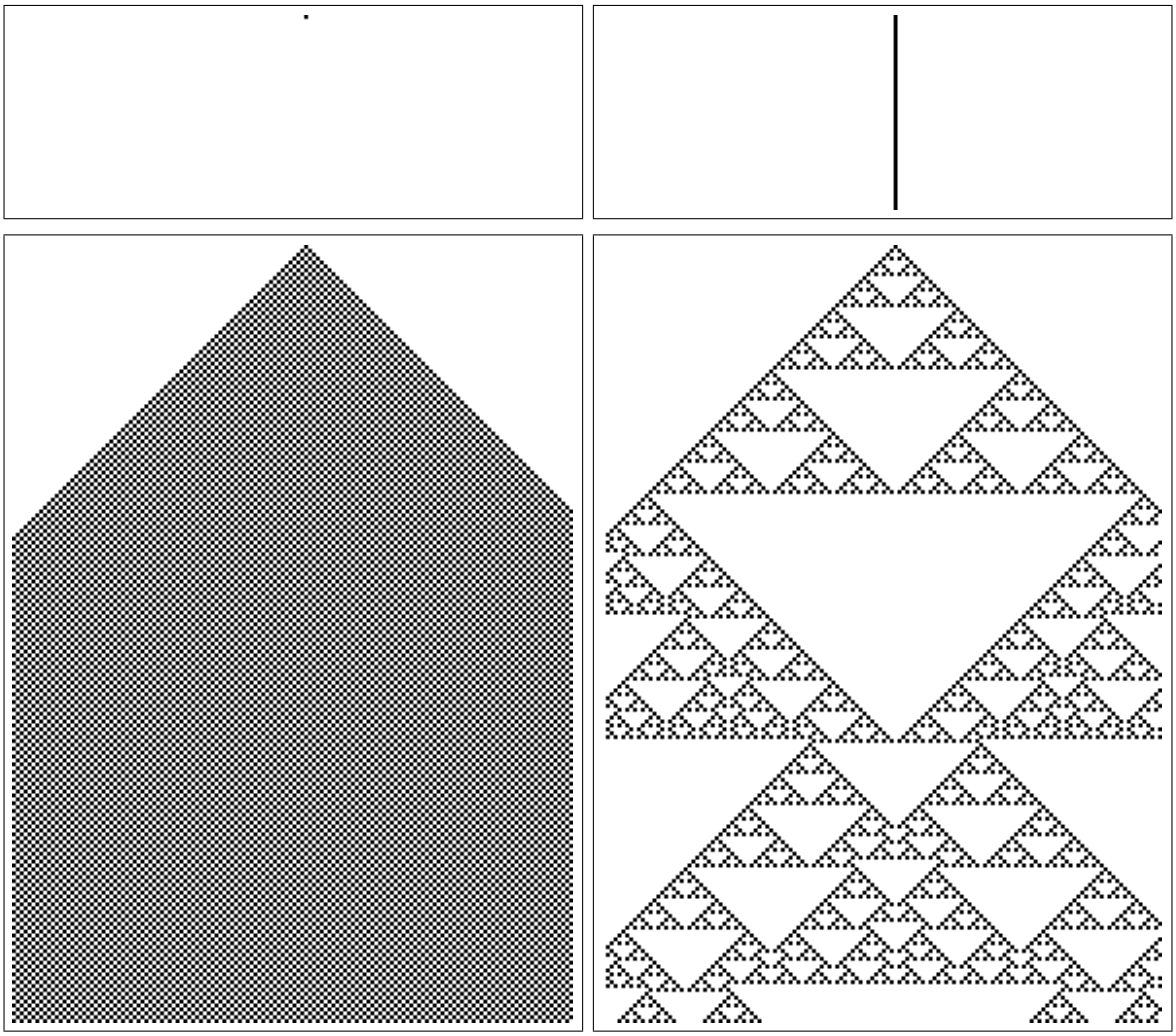


Figure 5: Evolution of a one-dimensional CA initialized with a single seed and submitted to Type 1 Rule 232 (top left), Type 2 Rule 36 (top right), Type 3 Rule 178 (bottom left), and Type 4 Rule 90 (bottom right). Each row represents the state of the 150-sites periodic CA at a given time step with black representing 1s and white representing 0s. The top row of each panel constitutes the initial single-seed configuration and subsequent rows represent successive time steps.

From Wikimedia

The universe of the Game of Life is an infinite two-dimensional orthogonal grid of square cells, each of which is in one of two possible states, live or dead. Every cell interacts with its eight neighbours, which are the cells that are directly horizontally, vertically, or diagonally adjacent. At each step in time, the following transitions occur:

1. Any live cell with fewer than two live neighbours dies, as if by needs caused by underpopulation.
2. Any live cell with more than three live neighbours dies, as if by overcrowding.
3. Any live cell with two or three live neighbours lives, unchanged, to the next generation.
4. Any dead cell with exactly three live neighbours becomes a live cell.

The initial pattern constitutes the 'seed' of the system. The first generation is created by applying the above rules simultaneously (synchroneously) to every cell in the seed — births and deaths happen simultaneously, and the discrete moment at which this happens is sometimes called a tick. In other words, each generation is a pure function of the one before. The rules continue to be applied repeatedly to create further generations.

Some of the shapes or patterns commonly generated in Conway's Game of Life can be divided into categories such as static patterns (still lives), repeating patterns (oscillators, a superset of still lives), and patterns that translate themselves across the board (spaceships).

Check out Wikipedia for a list of the common patterns and some very nice animated gif.

References

[1] K. P. Burnham and D. R. Anderson. *Model Selection and Multimodel Inference: A Practical Information-Theoretic Approach*. Springer-Verlag New York Inc., New York, USA, 2nd edition, 2002.

[2] L. V. Fausett. *Applied Numerical Analysis Using MATLAB*. Pearson Education, Prentice Hall, Upper Saddle River, NJ, 2nd edition, 2008.

[3] C. E. Hecht. *Statistical Thermodynamics and Kinetic Theory*. W.H. Freeman and Company, New York, 1990.

[4] R. H. Landau and M. J. Pàez. *Computational Physics: Problem Solving with Computers*. Wiley-Interscience, New York, 1997.

[5] R. H. Landau, M. J. Pàez, and C. C. Bordeianu. *Computational Physics. Problem Solving with Computers*. Wiley-VCH, Weinheim, Germany, 2nd edition, 2007.

[6] M. J. Miller, S. H. Wei, M. D. Cahalan, and I. Parker. Autonomous T cell trafficking examined *in vivo* with intravital two-photon microscopy. *P. Natl. Acad. Sci. USA*, 100(5):2604–2609, March 2003.

[7] W. H. Press et al. *Numerical Recipes in (Language): The Art of Scientific Computing*. Cambridge University Press, Cambridge, UK, any year.

[8] T. Sauer. *Numerical Analysis*. Addison Wesley, 2006.

[9] S. Wolfram. Statistical mechanics of cellular automata. *Rev. Mod. Phys.*, 55, 1983.

[10] S. Wolfram. Cellular automata as models of complexity. *Nature*, 311(5985):419–424, 4 October 1984.

[11] S. Wolfram. Cellular automata. *Los Alamos Science*, 9:2–21, Fall 1983. available at: `http://www.stephenwolfram.com/publications/articles/ca/83-cellular/`.