MATLAB for Geoscientists

Ángel Rodés

2021

Scottish Universities Environmental Research Centre

Contents

Chapter 1

Introduction to Matlab and Octave

1.1 Installation

Install it!

Install MATLAB ® following the instructions from the IT services

https://www.gla.ac.uk/myglasgow/it/software/statistics/#/matlab

or install GNU Octave© from the Web

https://www.gnu.org/software/octave/

1.2 Matrix-oriented programming

Matrix-oriented programming

- MATLAB and Octave are presented as "MATrix LABoratories", commonly used for plotting of functions and data, implementation of algorithms, creation of user interfaces, and interfacing with programs written in other languages.
- MATLAB® is a proprietary programming language developed by Math-

Works, Inc. GNU Octave© is free software under the terms of the GNU General Public License.

- MATLAB and Octave use languages that are mostly compatible. In this course we will use syntaxes compatible with both programs, unless otherwise stated.

1.3 Why MATLAB/Octave?

MATLAB and/or Octave will allow you to:

- Manage large datasets (raw data, synthetic results, maps, etc.).
- Perform iterative calculations.
- Write self-explained calculations and share them with the scientific community (i.e. suitable to be included in scientific publications).
- Plot exactly what you want.
- Learn a computing language that is easy.
- Learning how to program in MATLAB/Octave makes other languages much easier to learn: Matlab/Octave are similar to R, Python, C++, etc.
- Solve your own problems using your own programs, adapting exactly to your needs!

1.4 The interface

The interface

Matlab and Octave come with very similar interfaces containing, at least, the following elements:

- Address bar
- Browser
- Command window
- Workspace

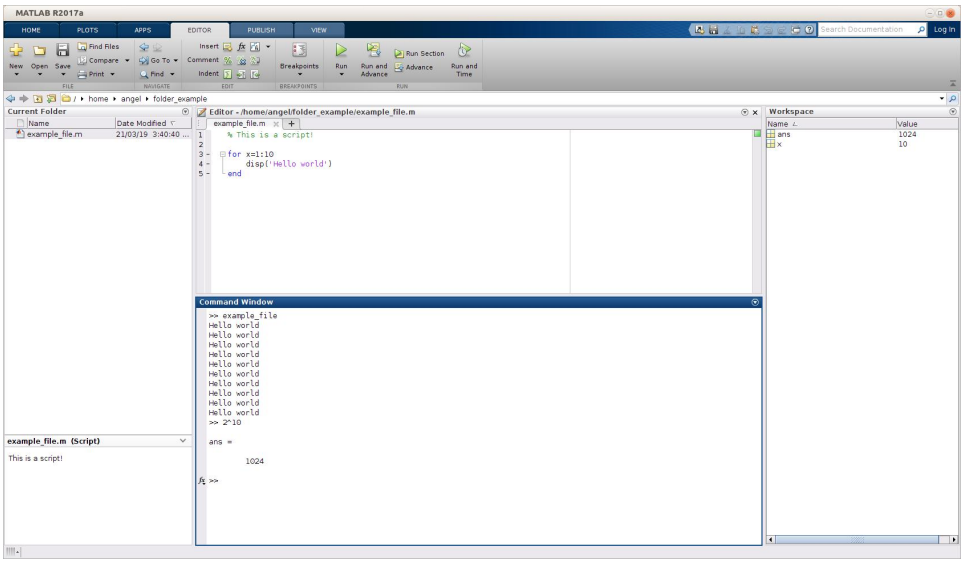


Figure 1.1: MATLAB interface.

- Editor

The command window This is the **brain** of the program. You can use this as a simple calculator or to call functions or scripts. Ultimately, this is the only element you need to use Matlab or Octave!

Try writing the following commands and hit enter:

- `97+6`
- `23-36`
- `23-6*6`
- `(23-6)*6`
- `12742*pi`
- `3^2`
- `sqrt(2)`
- `log(100)`
- `log10(100)`

To clean the command window, use the command `clc`.
As most of the console interfaces, the command window has memory: try using the up arrow key (↑).

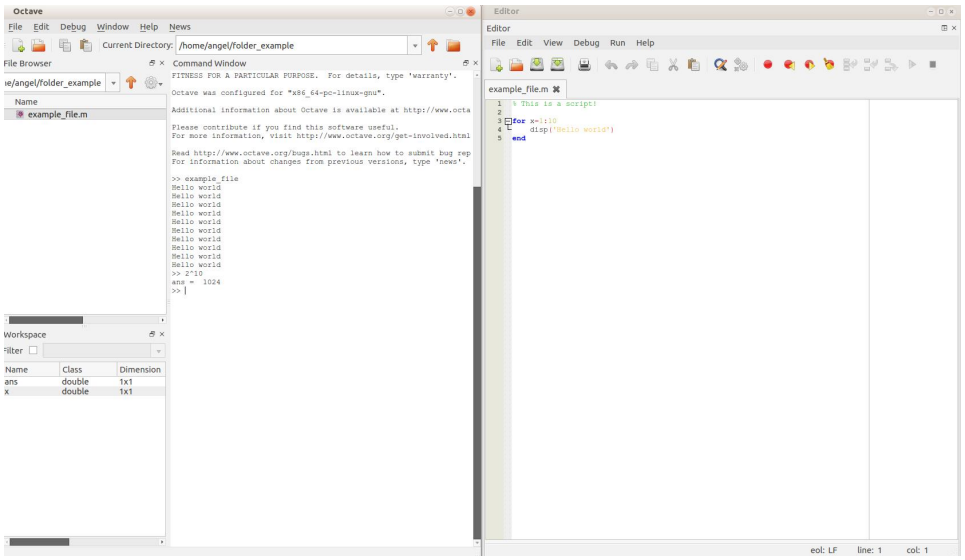


Figure 1.2: Octave GUI.

Current directory and browser

When you want to interact with files (e.g. calling your own scripts or creating files with your results or graphs), you need to know **where** you are working. Therefore, make sure that the address you see at the top of the screen is the folder you want to work in. You can see, create, delete or open your files using your system browser (explorer, finder, nautilus, etc.) or the browser integrated in Matlab or Octave.

Create a new file called ***my-first-file.m****.*
Avoid using spaces, most symbols, or start with numbers when naming your files. ***Instead of spaces, use the underscore symbol (_).***
Also, note that Matlab files always end with `.m`

The editor

This is just a basic text editor. The files that you can edit do not contain any information about formatting. You can open these files using any text editor (e.g. notepad). However, the integrated editor format the text to highlight the meaning of the text in the Matlab language.

Open `my_first_file.m` *and write:*

```
% This is my first Matlab script.
disp('Hello world')
```

then run it using the command `my_first_file` *(no* `.m`*) in the command*

window, or selecting ***run*** *in the menu.*

Workspace

This is the *memory* of Matlab/Octave. The last answer given in the command window is usually stored as `ans`.

Write `x=3+2` *in the command window.*
The parameter `x` *will appear in the Workspace.*

Use the command `who` or `whos` to display a summary of the workspace in the command window, and the command `clear` to remove all the parameters in the workspace.

1.5 What can we put in the Workspace?

Parameters with one number

- `mass=12`
- `C14halflife=5730;`
- `avogadro=6.022*10^23`

As for file names, avoid using spaces, most symbols, or start parameter's names with numbers.
Ending with semicolon (;) prevents the output to be shown in the command window, although the parameter is stored in memory. You can check that the value of C14halflife is in memory by typing `C14halflife` *in the command window.*

Other "special" accepted values:

- `maxtime=Inf`
- `mintime=-Inf`
- `unknownvalue=NaN`

`Inf` *means "Infinite" and* `NaN` *means "Not a Number". You can also generate them by computing* `1/0` *or* `0/0` *in the command window.*

Array of numbers

- `data=[254,782,65,5]`
- `moredata=[23;36;47]`
- `a=1:20`
- `a=1:0.25:10`
- `odds=1:2:100`
- `pairs=2:2:100`
- `emptyarray=[]`

Use `length(data)` to check the size of your array.

Try also `linspace(0,3,20)` and `logspace(0,2,5)`
to get equally distributed numbers in the linear or logarithmic space. Use `odds'` if you want it as a column.
Access a single (`data(3)` or `data(end)`) or several values of an array (`a(7:10)`)

Matrices

- `A=[1,2,3 ; 4,5,6 ; 7,8,9]`
- `B=[99,88,77 ; 66,55,44 ; 33,22,11]`
- `C=ones(4,3) % number of rows,columns`
- `D=zeros(4,3)`

Note that anything you write after the `%` symbol is ignored. ***`%` is used for comments.***

You can also create a matrix by repeating an array using `repmat`:

```
repmat(data,3,1)
```

Use `help repmat` to know more about this.

These matrices are 2-D (rows and columns). However, MATLAB and Octave are also able to handle matrices in multiple dimensions. E.g. `ones(3,2,5)` is a 3-D matrix.

Use `size(B)` to check the size of your matrix (rows and columns), or n`numel(B)` to get the number of elements in `B`.

Strings

Strings are parameters containing text:

- `name='John'`
- `students=[{'Gerry'},{'Trish'},{'Pablo'}]`

Strings are useful when working with sample or location names. MATLAB and Octave can handle strings and provide powerful tools to manipulate and operating with text, such as regular expressions. However, these programming languages were not primarily designed to work with text, and string manipulation can be very frustrating at the beginning. Therefore, we will restrict the use of text to sample names or simple labels.

Sometimes it will be useful to find a sample in a list. For example, use `strcmp` *to find the position of the student named Trish:* `strcmp('Trish',students)`

Small functions

Simple formulas can be defined by using defining the parameters with @(Parameters):

- `temp_fahrenheit = @(temp_celsius)1.8 * temp_celsius + 32`
- `meters=@(ft)ft/3.2808`
- `decay=@(halflife,time)exp(-log(2)/halflife*time)`

Try `temp_fahrenheit(15)` *and* `decay(C14halflife,20000)`

Boolean data

Boolean data is a type of data that has one of two possible values: true (1) or false (0). In MATLAB, logical is usually generated used equalities or inequalities:

- `avogadro>1E23`

- `mass==12`
- `odds<10`
- `odds(odds<10)`
- `A>5`
- `isinf(maxtime)`
- `isnan(B)`
- `isprime(7537)`

Note that `==` and `~=` are used in MATLAB to determine equality or inequality, and `=` to define a parameter.

We can combine boolean data using boolean operators: `&` (and) and `|` (or).

E.g. `( A<5 | B<30 )`.

Boolean data can also be use as indexes if the boolean array or matrix has the same size as the objective array or matrix.

E.g. `A(A>5)` or `B(A<3)` but not `data(A<10)`.

This property is useful to easily create filters for our data:

```
data(data>50 & data<500)
```

clc to clean the Command window

Basic calculations

With numbers: `mass*avogadro`

With arrays and matrices: `odds+pairs` but `odds.*pairs`

Note the difference between `B/A` and `B./A`:

"`.*`", "`./`" and "`.^`" are operators used to perform calculations element by element (array operations). Avoid using "`*`", "`/`" and "`^`" on matrices unless you really want to do matrix operations following the rules of linear algebra.

Call parts of another variable: You can access the number in the second row and third column with `A(2,3)`, the second row with `B(2,:)` or the

first column with `A(:,1)`. MATLAB and Octave always follow the order (**row,column**) in 2D matrices.

Random numbers

- `rand` % any number between 0 and 1
- `rand(1,10)` % a row of 10 random numbers
- `rand(10,1)` % a column of 10 random numbers
- `rand(3,3)` % a 3x3 matrix with random numbers between 0 and 1
- `A.*rand(3,3)` % a matrix with random numbers between 0 and numbers in matrix A
- `normrnd(11000,2000)` % a random number from a gaussian distribution of 11000±2000
- `normrnd(11000,2000,1,5000)` % a row of 5000 random numbers from a gaussian distribution of 11000±2000

Try `hist(normrnd(11000,2000,1,5000))` *and* `hist(rand(1,5000))` *to plot the histograms corresponding to these random distributions.*

1.6 Plots

Air pressure

Let's define a function that calculates the pressure at a certain altitude:

```
pressure = @(altitude)1013.25*...
 exp(-0.03417/0.0065*(log(288.15)-...
 (log(288.15-0.0065*altitude))))
 % standard atmosphere pressure (Lide, 1999)
```

Note that we can use three dots (...) to avoid long lines.

Then define `x` values between 0 (sea level) and 8848 m (Everest) every 100 m:

```
x=0:100:8848
```

And alculate their corresponding pressures:

```
y=pressure(x)
```

Simple plots

Try the following plots:

- `plot(x,y)`
- `plot(x,y,'.r')`
- `plot(x,y,'ob')`
- `plot(x,y,'--k')`
- `plot(x,y,'-g','LineWidth',2)`
- `bar(x,y)`
- `stairs(x,y)`

Figure

Create a figure and plot several things in it:

```
figure % open a new figure
hold on % do not clear when plotting different things
plot(x,y,'-b')
plot(200,pressure(200),'hr')
text(200,pressure(200),'East Kilbride')
xlabel('Altitude')
ylabel('Pressure')
title('My first plot with labels')
```

Make y axis logarithmic: `set(gca, 'YScale', 'log')` (`gca` means "Get current axes") *You can export your plots using the menu* ***File*** $>$ ***Save As*** *in the figure window. Exporting your plots as .eps or .pdf will allow you to edit them with vector graphic editors like Adobe Illustrator or Inkscape.*

1.7 Scripts

Scripts

A script is a text file with a list of orders. In your current directory, create `radiocarbondating.m`. Open it with the editor and write the following orders:

```
%% This is a script that calculates radiocarbon ages and errors
%% By Me, 2019

%% Start with some cleaning
clear % this removes any previous parameter in the workspace
clc % this clears the command window

%% Define the formula that calculates the age from concentrations
C14age=@(modernconcentration,measuredconcentration)-...
8033*log(measuredconcentration./modernconcentration);

%% This is the data we have
modernc=1232;
errormodernc=13;
oldc=[567 1100 20 1252];
erroroldc=[6 20 5 50];

%% Select the data we want to work with
n=1

%% Create 1000 random data based on the normal dristributions
randommodern=normrnd(modernc,errormodernc,1,1000);
randomold=normrnd(oldc(n),erroroldc(n),1,1000);

%% Calculate the ages of the distributions
ages=C14age(randommodern,randomold);

%% Plot the age distribution
figure
hold on
hist(ages)
title(['Sample ' num2str(n)])
xlabel('Age')
ylabel('Probability')

%% Calculate the mean and the average
age=mean(ages)
errorage=std(ages)
```

Now you can change the value of `n` to get the results of other data.

Note that we can make composed strings using brackets `[]` and the function `num2str(n)` to convert numbers into strings.
Also note that we can use `...` to avoid very long lines.

1.8 Loops

Loops

We often need to run a block of code several times. For example, in our program `radiocarbondating.m` we could copy and paste the script 4 times changing `n=1` by `n=2`, `n=3` and `n=4` to get all our ages calculated. However, we avoid repeating code by writing a loop statement that executes the code multiple times.

In `radiocarbondating.m`, we can substitute "`n=1`" by "`for n=[1,2,3,4]`" and write "`end`" at the end of the script to perform the calculations and plotting for the four samples.

The basic form of a loop in Matlab is:

```
for Parameter=List
  % My repeating code
end
```

Error bars

Create a new script called **plot-with-error-bars.m** that use a loop to plot error bars of the individual concentrations:

```
%% This is a script that plots data with error bars
%% By Me, 2019

%% Start with some cleaning
clear % this removes any previous parameter in the workspace
clc % this clears the command window
close all hidden % close any pre vious figure

%% This is the data we have
data=[567 1100 20 1252 326 625];
errors=[6 20 5 50 32 100];

%% Figure
figure
hold on
for n=1:length(data) % start a loop
  plot(n,data(n),'.b')  % Plot data
  x=[n,n]; % x positions of the limits of the error bar line
  y=[data(n)-errors(n),data(n)+errors(n)]; % y positions
  plot(x,y,'-b') % plot the error bar
end % end of the loop
xlabel('Sample')
ylabel('Concentration')
```

Another way of creating a loop is using the statement `while`:

```
n=0;
while n<10
    n=n+1 % add 1 to the value of n
end
```

1.9 Conditional statements

if - end

Conditional statements allow us to select at run time which block of code to execute. The simplest conditional statement is `if`, closed with `end`:

```
n=round(rand*100); % random number between 0 and 100
                   % rounded to the nearest integer
if n/2==round(n/2)
  string=[num2str(n) ' is pair']
end
```

if - elseif - else - end

We can define alternatives using `if`, `elseif`, `else` and `end`:

```
n=round(rand*100);
if n/2==round(n/2)
  string=[num2str(n) ' is pair'];
elseif isprime(n)
  string=[num2str(n) ' is odd and prime'];
else
  string=[num2str(n) ' is odd, but not prime'];
end
disp(string) % disp shows the string in the command window
```

You can also define conditional statements using `switch` *(*`switch`*,* `case`*,* `otherwise` *and* `end`*). Find yourself how to use the* `switch` *statement by typing* `help switch` *in the command window!*

1.10 Functions

Functions

A function is a script that works like a "black box". You only see the final output in the workspace, not all the parameters defined in the function. When writing a function, or converting a script into a function, we have to start the file with

`function OUTPUTS = function_name(INPUTS)`

and write

`end`

at the end of the file.

^{14}C age function

Create a file called **C14agefunction.m** and copy:

```
function [age,errorage]=C14agefunction(oldc,erroroldc,modernc,errormodernc)
  C14age=@(modernconcentration,measuredconcentration)-...
    8033.*log(measuredconcentration./modernconcentration);
  randomold=normrnd(oldc,erroroldc,1,10000);
  randommodern=normrnd(modernc,errormodernc,1,10000);
  ages=C14age(randommodern,randomold);
  age=mean(ages);
  errorage=std(ages);
end
```

Note that the function name has to be the same as the file name. Otherwise you will get an error when running it.

Save the file, and then execute the following in the command window:

`C14agefunction(50,10,1254,20)`

`[age,error]=C14agefunction(50,10,1254,20)`

1.11 Built-in functions

Built-in functions

MATLAB and Octave come with a large number of built-in functions (e.g. `factorial`, `sin`, `sum`, `diff`, `max`, `magic`, `pi`, `median`, `chi2pdf`, `interp1`, `contour`, and many more).

You can learn how to use these functions using `help` (e.g. `help interp1`), selecting the name of the function and pressing F1 in MATLAB.

Also, you can discover more functions in the Internet. Just search for the operation you want to do, including "Matlab" or "Octave" in your search.

We can even see how some of these built-in functions are made with `edit`*. Try* `edit magic` *to see the code of the function that generates magic squares!*

Toolboxes and packages

There are some advances functions, like the ones used to work with maps, that are not included in the basic package of MATLAB and Octave. These "special packages" are called "toolboxes" in MATLAB and just "packages" in Octave.

Toolboxes are installed using the MATLAB installer and they are automatically loaded when you start MATLAB.

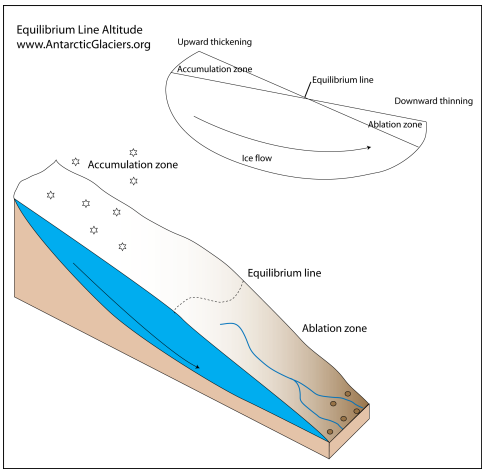
Octave packages can be installed using `pkg install` and the name of the file where the package is. Before we start using an Octave package, we have to load it with `pkg load package_name`.

As one of the objectives of this course is learning to write code we can share, most of the built-in functions that we are using in this course are included in the basic versions of MATLAB and Octave. If a toolbox or package is required, it will be clearly stated.

1.12 Exercises

Snow and glacier modelling

A glacier is a persistent body of dense ice that is constantly moving under its own weight. *(Wikipedia: Glacier)*



(*http://www.antarcticglaciers.org*)

Consider the following climate simplifications:

- Average monthly temperature (°C) at sea level in Scotland:

Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
4	5	7	8	12	14	16	16	13	10	7	5

- Temperature lapse rate: 8°C/Km
- Monthly precipitation (mm) in Scotland:

Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
175	125	150	100	75	100	100	125	125	175	175	175

Consider the following snow/ice behaviour (huge simplifications):

- All precipitation is **snow when temperature is below 5°C**. All precipitation is rain above 5°C.
- Daily temperature range is 5°C, so **day temperature is 2.5°C above the average**.
- Considering thermal conductivity of the snow mantle ∼5 W/K/m^2, a snow latent heat of fusion of 350 kJ/kg and a snow average density of ∼0.3 Kg/l, an average of vertical **5 cm of snow per month will be melted for each degree of day temperature over 0°C**.
- If the snow survives for more than a year (annual mass balance > 0), the snow will flow downhill at an **horizontal speed of 10 inches/day**.

- The average **glacier slope is 15°**.

Mass balance:

1. Write a function that calculate the monthly snow mass balance (snow accumulation-snow melting). *Remember that the melting function should not create snow!*
2. Write a function that calculate the snow accumulated monthly. *Remember that (1) we can have snow inherited from the previous month, and (2) the thickness of the snow mantle cannot be negative!*
3. Write a piece of code that calculates the annual mass balance. Introduce the possibility of emulate past and future climate conditions by changing the temperature and precipitation uniformly (ΔT and ΔP).

The output of the monthly functions should be an array of 12 numbers when the input is one altitude, or a matrix when the input is a "column" of altitude values.

Snow accumulation:

1. Placing a ski resort: what is the lowest altitude with 3 or more months of snow?
2. According to these data, where could we find a glacier in Scotland today? *Note: the highest peak in Scotland is Ben Nevis, 1345 m above sea level.*

The Glenshee ski area is located between 650 and 1050 m of altitude. What impact would these scenarios have on the business by 2100?

Glacier modeling exercises:

1. Write a piece of code that emulate the annual snow/ice mass flow. *Tip: calculate how much the snow/ice moves vertically in a year and discretize the altitude reference accordingly, so the snow packed during the previous year will move one position per year.*

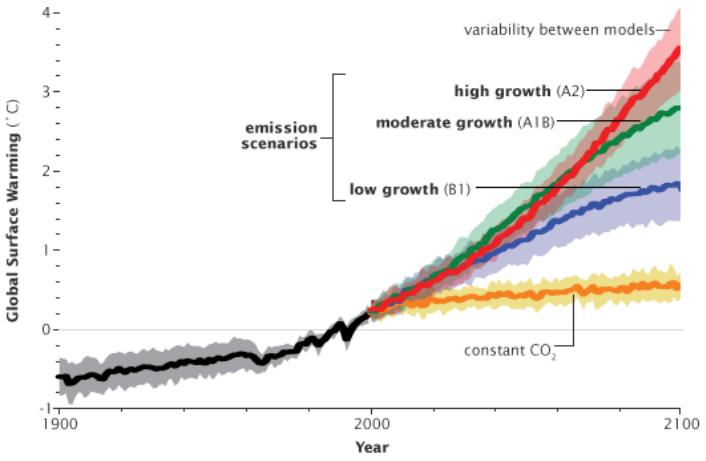
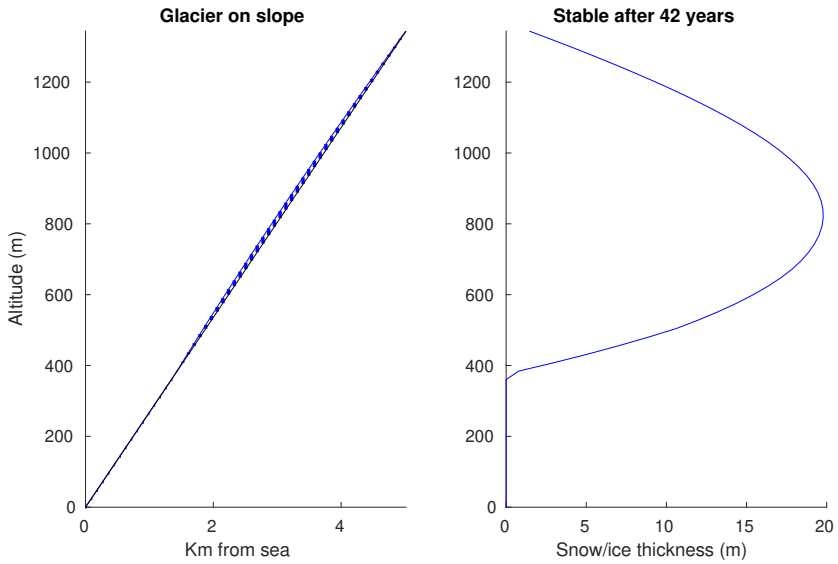


Figure 1.3: earthobservatory.nasa.gov

2. Write a script that runs the previous code until the thickness of the snow/ice is stable.

3. According to this model, where should the glacial fronts have been during the Younger Dryas (ΔT=-4°C)? and during last glaciation (ΔT=-6°C)?

Produce graphical outputs like these:



Chapter 2

Importing data & Statistics

Import functions

We can import our data in many ways. There are lots of built-in functions that can be used to input data from different file types and different formats: `input`, `importdata`, `load`, `xlsread`, `imread`, `geotiffread`, `arcgridread`, `usgsdem`, etc. Here we will learn some of the simplest and more universal ways of doing it: using `csvread`, `textscan` and directly pasting data in a dialog box with `inputdlg`.

2.1 Importing .csv files

.csv files

CSV stands for "comma-separated values". CSV files are text files widely used to store tabular data in a simple format. All spreadsheet manipulation programs, as Microsoft Excel, are able to import and export CSV files. Each line in a CSV file corresponds to a row in a spreadsheet. Values from different columns are separated by commas.

`csvread(filename, row, col)` reads data from the comma-separated value formatted file starting at the specified row and column. The row and column arguments are zero based, so that row=0 and col=0 specify the first value in the file.

Note that only numeric data can be read using `csvread`. For example,

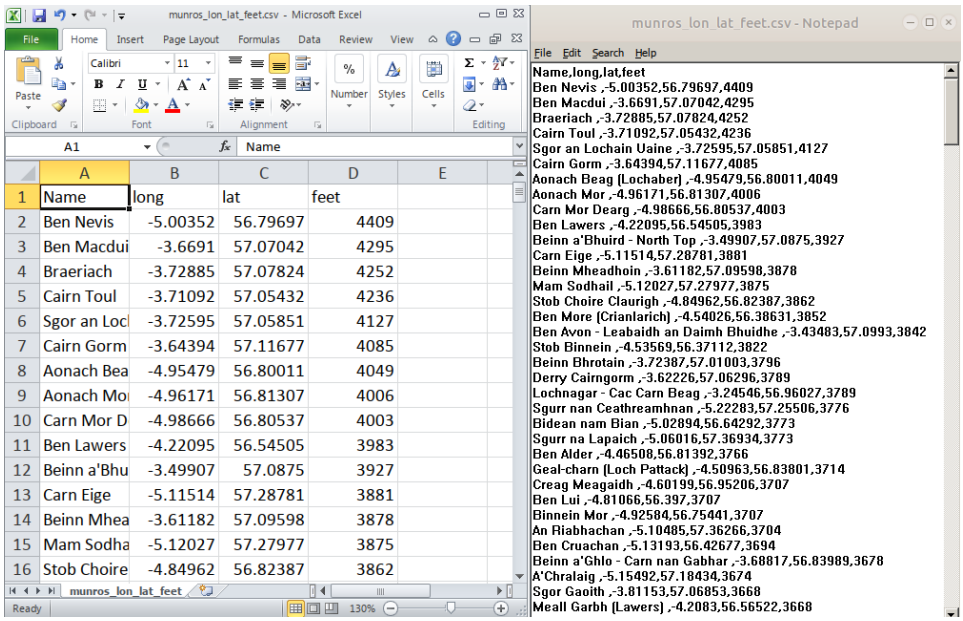


Figure 2.1: Same .csv file in Excel and in a text editor.

the file `munros_lon_lat_feet.csv` contains text and data:

Name,	long,	lat,	feet
Ben Nevis ,	-5.00352,	56.79697,	4409
Ben Macdui ,	-3.6691,	57.07042,	4295
Braeriach ,	-3.72885,	57.07824,	4252
Cairn Toul ,	-3.71092,	57.05432,	4236
...	...	...	...

So the orders `csvread('munros_lon_lat_feet.csv',0,0)` does not work or do not import the Munros' names. To import the numerical data from this file we should use: `munrodata=csvread('munros_lon_lat_feet.csv',1,1)`

2.2 Importing data from text files using `fopen` and `textscan`

`fopen` and `textscan`

We can import tabulated data, including text strings, from any text file. To do so, we need to know how many rows with headers are in the file (in this case: `1`) , what is the symbol that delimiters the columns (in this case: `,`), and the type of data in the different columns. In this case, the first column contains text (`%s` for *string*) and the three next columns contain numbers (`%f` for *floating-point number*):

```
fid = fopen('munros_lon_lat_feet.csv');
munrodata = textscan(fid, '%s %f %f %f',...
   'HeaderLines', 1,'Delimiter',',');
fclose(fid);
```

Once imported our data, we can organize it in different arrays:

```
names=munrodata{1};
lon=munrodata{2};
lat=munrodata{3};
feet=munrodata{4};
```

2.3 Input dialog

Dialogs

We can also input our data copied from a spreadsheet (like Excel) using `inputdlg` as a string and then convert it into a matrix using `textscan`:

```
cstr = inputdlg ('Paste from excel','Input new data');
mydata=textscan(cstr{1}, '%s %f %f %f');
```

When using this method to input data remember that:

- The **text strings** (usually sample names) **should not contain spaces** or certain symbols.
- **Avoid empty cells**: you can use a `0` or `NaN` instead.
- You should only copy rows with data. Avoid copying the headers.

2.4 The normal distribution

Gaussian distribution

Most analytical data are considered **Gaussian (or normal) distributions** (Fig. 2.2). This means that the true value of whatever we are

measuring could be equally higher or lower that the measured central value (the data) and its probability is:

$$P = \frac{1}{\sqrt{2\pi\sigma^2}} \cdot e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

where x are the possible values, μ is the central value (the mean), and σ is the uncertainty (the standard deviation).

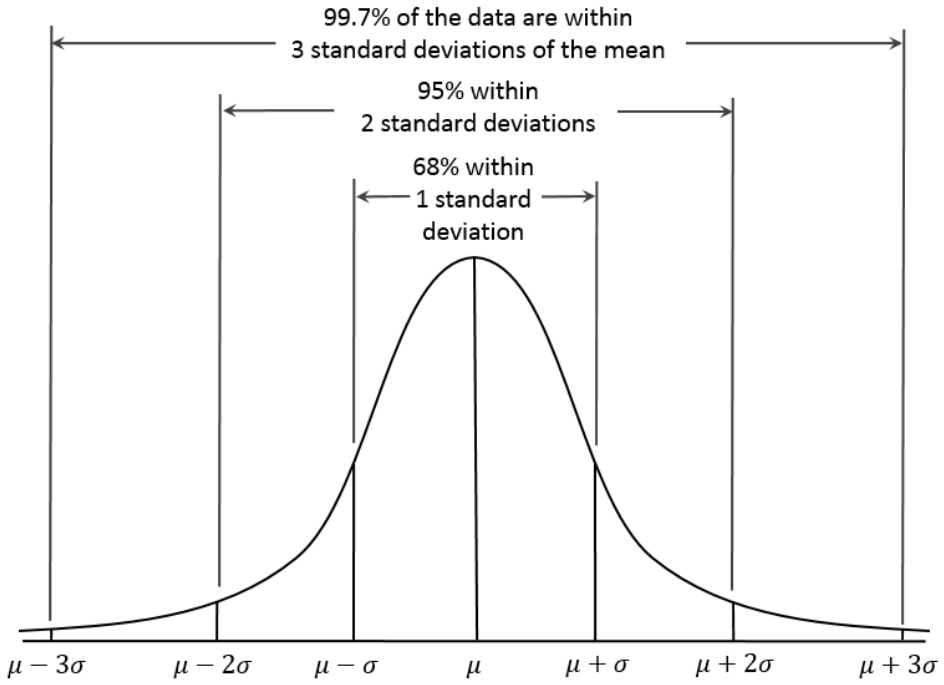


Figure 2.2: For the normal distribution, the values less than one standard deviation away from the mean account for 68.27% of the set; while two standard deviations from the mean account for 95.45%; and three standard deviations account for 99.73%. Author: Dan Kernler. https://en.wikipedia.org/wiki/Normal_distribution

Create a function called **normalprobs.m** that calculates the probabilities of an array x, given a piece of data as $\mu \pm \sigma$:

```
function [P]=normalprobs(x,mu,sigma)
  % Calculates the probability of x based on a gaussian mu +/- sigma
  P=1/(2*pi*sigma^2)^0.5*exp(-(x-mu).^2./(2*sigma^2));
```

```
end
```

Then compare the following plots:

- `hist(normrnd(56,15,1,1000))`
- `plot(1:100,normalprobs(1:100,56,15))`

2.5 Calculating the average

Averages

Geochronologists often produce a set of ages to date one geologic event. **Each of these ages are always the result of fitting a model** to the analytical data, usually some concentration(s) in a rock or mineral. Generally, the relatively simple models used to generate "standard" ages are based in assumptions on how the nature works. But the processes that rule the concentrations in nature are always much more complicate than assumed by our model. Thus, we should always expect some scatter in our apparent ages due to this natural "noise". However, in principle we don't known how much scatter can we attribute to the differences between the nature and our model.

As we have seen before, any analytical data has also associated some uncertainty related to the precision of our measurements (the error bars). This **known** uncertainty should also contribute to the scatter of our data.

Before calculating the average of our ages, we should understand what kind of uncertainty will dominate our averaged age. If we don't have many samples, a simple way of checking this is just plotting our ages. If we have a large dataset, we can also compare our scatter with our individual uncertainties using `std`(ages) and `median`(errors).

For example:

```
%% Group opf ages from LGM moraines
ages=[27311,18071,19698,19868,25357,21515,19486,18784,19311,...
      14342,19412,18064,18554,18092,18194,19647,19390,18634,...
      19900,18069];
errors=[8839,2263,1893,1780,1568,2754,2720,2516,1414,1265,...
```

```
       2239,1389,3249,1287,1385,1323,1482,2044,1787,3392];
%% Plot ages
  figure % start a new figure
  hold on % keep all plotted elements
  for n=1:length(ages)
    % plot the error line
    plot([n,n],[ages(n)-errors(n),ages(n)+errors(n)],'-b')
    % plot the central data
    plot(n,ages(n),'.b')
  end
%% Calculations
  SCATTER=std(ages)
  ANALYTICAL_UNCERT=median(errors)
```

Comparing scatter and analytical uncertainties, we can decide which is the best way of averaging our data:

- If scatter is much bigger (orders of magnitude) than our analytical errors, we can just ignore the analytical uncertainties.
- If the scatter is about the same or a few times bigger than the analytical errors, our final age should reflect both the analytical and model uncertainties.
- If the scatter is much smaller than the analytical errors, we are probably overestimating our analytical uncertainties. We should check our previous calculations.

2.6 Types of "averages"

Average of a group of numbers

The most used type of average is the mean: `mean(ages)`, which is the same as `sum(ages)/length(ages)`. The uncertainty of the mean is the standard deviation: `std(ages)`, which is

```
sqrt(sum((ages-mean(ages)).^2)/(length(ages)-1))
```

The Standard Deviation Of the Mean (SDOM) gives us an idea of how the mean can change with new measurements:

```
std(ages)/sqrt(length(ages))
```

The SDOM is often used as the uncertainty of a large number of analytical measurements on the same material, but **it does not reflect the uncertainty related to the natural variability expected in a group of ages from the same geological formation**.

In large datasets containing extreme values, the median could also be a good choice to represent the data: `median(ages)`. The median is less affected by outliers than the mean, and is often the preferred measure of central tendency when the distribution is not symmetrical. As for the mean, we can calculate its uncertainty as:

```
sqrt(sum((ages-median(ages)).^2)/(length(ages)-1))
```

For analytical data, the standard deviation of the median is considered to be a $\sim 25\%$ higher than the SDOM.

Average of a group of numbers and uncertainties

When our data consist of a group of probability distributions (e.g. ages and errors), we should take into account the errors in the calculation of the average. If our data have different errors, the data with bigger errors should *weight* less than the more precise data. To take this into account, we can use the **weighted mean**:

```
WM=sum(ages./errors.^2)/sum(1./errors.^2)
```

The standard deviation of the weighted mean (SDOWM) average can be calculated as:

```
SDOWM=sqrt(1/sum(1./errors.^2))
```

However, the SDOWM only reflects the uncertainty from the individual errors and not the scatter of the data. A more realistic uncertainty could be calculated as:

```
sqrt(std(ages)^2+SDOWM^2)
```

An alternative method to calculate the average and uncertainty of a group of ages and errors is actually simulating their probability distributions:

```
simulations=1000;
% create a matrix to place data
fakedata=zeros(simulations,length(ages));
for n=1:simulations
  % fill each line with random data
  % based on individual ages and errors
  fakedata(n,:)= normrnd(ages,errors);
end
MEAN=mean(fakedata(:))
% note that (:) converts a matrix into an array
UNCERTAINTY=std(fakedata(:))
hist(fakedata(:),30) % plot the fake data
```

2.7 Error transmission

Error transmission

Simulating the dispersion of the data by generating "fake data" according to gaussian distributions is great trick to operate with probabilistic data, getting a density distribution of the result. However, when the following conditions are met, it is much more efficient to propagate errors mathematically.

- All operators (e.g. analytical data) are well-modeled by **gaussian distributions**.
- We are considering uncertainties of **independent variables** (no covariance).
- The **uncertainty of the result** is small enough to be well represented by a normal distribution (i.e. the result is roughly lineal in the area of the uncertainty and therefore the resulting distribution is **not asymmetrical**).

If we can assume these conditions, the error propagation should be performed by considering the partial derivatives of the result respect the operators:

$$\sigma_{f(a,b)} = \sqrt{\left(\sigma_a \frac{\delta f(a,b)}{\delta a}\right)^2 + \left(\sigma_b \frac{\delta f(a,b)}{\delta b}\right)^2}$$

where $a \pm \sigma_a$ and $b \pm \sigma_b$ are the operators within a standard deviation

(one sigma) uncertainties.
Here are some examples on common operations:

Operation	Formula	Uncertainty
Addition	$a+b$	$\sqrt{\sigma_a^2+\sigma_b^2}$
Subtraction	$a-b$	$\sqrt{\sigma_a^2+\sigma_b^2}$
Multiplication	$a\cdot b$	$\sqrt{(\sigma_a\cdot b)^2+(\sigma_b\cdot a)^2}$
Division	a/b	$\sqrt{\left(\frac{\sigma_a}{b}\right)^2+\left(\sigma_b\cdot\frac{a}{b^2}\right)^2}$
Power	a^b	$\sqrt{\left(\frac{\sigma_a\cdot a^b\cdot b}{a}\right)^2+(\sigma_b\cdot a^b\cdot\ln a)^2}$
Natural logarithm	$a\cdot\ln(b)$	$\sqrt{(\sigma_a\cdot\ln(b))^2+\left(\sigma_b\cdot\frac{a}{b}\right)^2}$
Logarithm to base 10	$a\cdot\log_{10}(b)$	$\sqrt{(\sigma_a\cdot\log_{10}(b))^2+\left(\sigma_b\cdot\frac{a}{b\cdot\ln(10)}\right)^2}$

2.8 Rejecting outliers

Rejecting outliers

In statistics, an outlier is a data point that differs significantly from other observations.

A simple method to identify outliers is using the Tukey's fences based on the data quartiles. This method identify outliers at deviations outside the range from $Q_1-1.5\cdot(Q_3-Q_1)$ to $Q_3+1.5\cdot(Q_3-Q_1)$. We can calculate this limits using the built-in function `quantile`:

```
Q1=quantile(ages,0.25);
Q3=quantile(ages,0.75);
outliers=ages(ages<(Q1-1.5*(Q3-Q1)) | ages>(Q3+1.5*(Q3-Q1)))
```

Other approaches commonly used to identify outliers are based on the goodness of fit. A simple way of measuring how close are our measurements to our mean is the χ^2 value:

$$\chi^2=\left(\frac{x-\bar{x}}{\sigma}\right)^2$$

Using $\sigma=\sqrt{\sigma_x^2+\sigma_{\bar{x}}^2}$ we can calculate the individual values of χ^2 considering all our uncertainties. Values of χ^2 greater than 1 indicate that the two

values we are comparing, the individual data and our average, are different within uncertainties.
Exercise: Use the χ^2 method to identify outliers of the ages respect their weighted mean and standard deviation of the weighted mean.

In geosciences (e.g. studying a set of ages from a landform), outliers are often due to the natural variability of the samples and not to experimental errors. Therefore, ***when rejecting an outlier we are expected to give an explanation of the geological process that caused that outlier****.*

2.9 Box plots

Box plots

A common way of representing a dataset is using box plots. The data is usually represented along the y axis, a box is drawn from Q_1 to Q_3. The box is cut at the median ($= Q_2$) and error bars are drawn outside the box between the limits defined in section 2.8

This code produces a box-plot of the given ages:

```
%% my data
ages=[27311,18071,19698,19868,25357,21515,19486,18784,19311,...
      14342,19412,18064,18554,18092,18194,19647,19390,18634,...
      19900,18069];
%% calculate the cuartiles
Q1=quantile(ages,0.25);
Q2=median(ages);
Q3=quantile(ages,0.75);
maxbar=Q3+1.5*(Q3-Q1);
minbar=Q1-1.5*(Q3-Q1);
outliers=ages(ages<minbar | ages>maxbar);

%% start a figure
figure
hold on
%% plot box at position x=2 and a width of 0.3
plot([2-0.3,2+0.3],[Q1,Q1],'-k')
plot([2-0.3,2+0.3],[Q2,Q2],'-k')
plot([2-0.3,2+0.3],[Q3,Q3],'-k')
plot([2-0.3,2-0.3],[Q1,Q3],'-k')
plot([2+0.3,2+0.3],[Q1,Q3],'-k')
plot([2,2],[Q3,maxbar],'-k')
plot([2,2],[Q1,minbar],'-k')
plot([2-0.15,2+0.15],[minbar,minbar],'-k')
plot([2-0.15,2+0.15],[maxbar,maxbar],'-k')
plot(ones(size(outliers))*2,outliers,'.k')
% ones(size(outliers) is used because
% in principle we do not know the size of
```

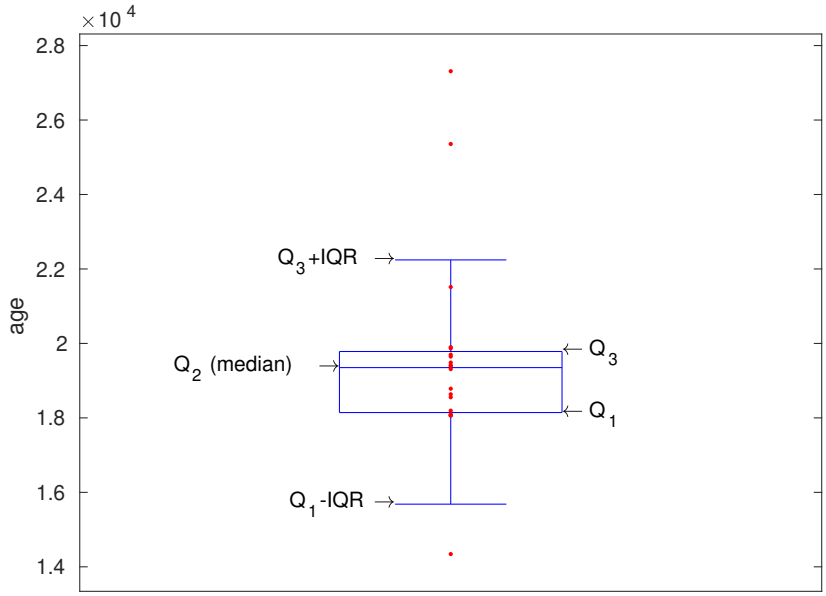


Figure 2.3: In a box-plot, the inter-quartile range (IQR) is defined by the distance between the first and third quartile Q_1 - Q_3, so 50% of our data are in the IQR, 25% is over Q_3 and the rest 25% is under Q_1.

```
% the outliers array
%% beautify the axis
xlim([0 4])
set(gca,'xtick',[]) % remove x ticks
ylim([0 max(ages)*1.5])
ylabel('age')
box on % draw upper and right lines
```

Exercise: Write a function that draws the box plot at a given x position and box width. Make the width of the caps half the width of the box. The input should be: `drawboxplot(data,x,width)`

Then use the function to compare the altitudes of the eastern and western munros from section 2.2.

Remember that you can recycle code using copy & paste!

2.10 Histograms

Histograms

Another way of visualizing a the distribution of a large dataset is plotting an histogram (e.g. `hist(ages)`). We can select the number of "bars" to plot: `hist(feet,30)`. We can also use the built-in function `hist` to generate the counting data and plot it using `plot`:

```
figure
[counts,centers] = hist(feet,20);
plot(centers,counts,'*-r')
```

Exercise: Compare graphically the mean + standard deviation, the box plot and the histogram of these data. Which one reflect better the variability of our ages?

2.11 Camel-plots

Camel-plots

When our data have associated errors (like our `ages` and `errors`), the histogram does not represent the relative weight of our individual data. The probability distribution of the age 12000 ± 1500 can be depicted using the function defined in section 2.4:

```
% Define the x values to plot
xref=linspace(0,50000,500);
% calculate the probability distribution
probs=normalprobs(xref,12000,1500);
% plot
figure
hold on
plot(xref,probs,'-b')
```

Exercise: Write a script that sum all the probabilities of the previously defined ages for each xref and plot it. The plot of this sum should look

similar to the blue line in the next figure.

The generated plot show the density of our data better than the histogram. These plots are sometimes called "probability density plots". However, these diagrams represent the distribution of our data, that highly depends on how we selected the samples. Therefore, they do not necessarily represent the probability distribution of the landform age and, according to Greg Balco, they should be called "normal kernel density estimates."
https://cosmognosis.wordpress.com/2011/07/25/what-is-a-camel-diagram-anyway/

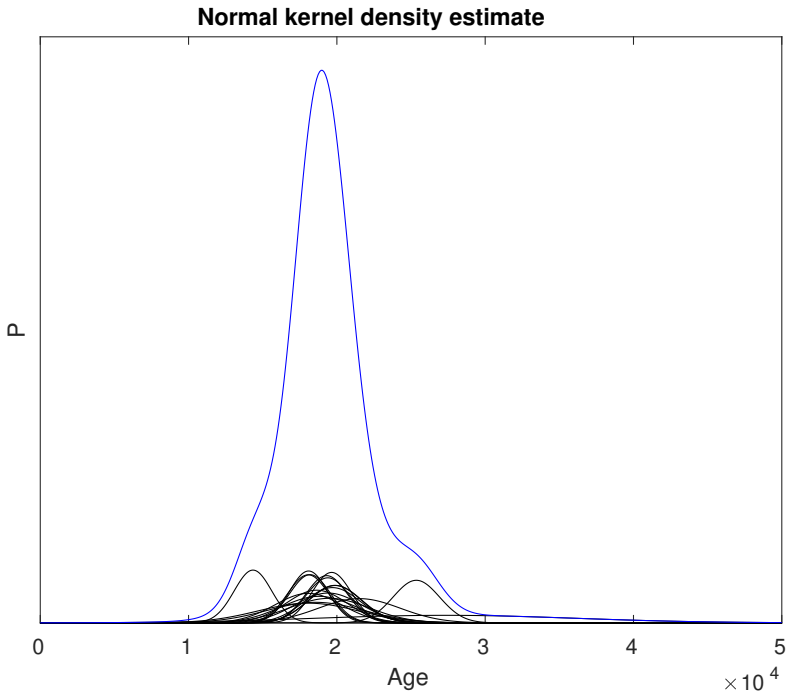


Figure 2.4: The plot of the sum of probability distributions from our data is often called "Camel-plot" (informally).

Chapter 3

Data calibration

3.1 What is a calibration?

Calibrations

Geochemical analysis often require calibrating a machine. A calibration is a method that compares

- Nominal values of something we consider *real* (e.g. known concentrations of Fe in a solution), and
- Directly measured data (e.g. counts per second from an ICP machine)

To perform a calibration we usually need to measure a set of known samples (standards) in a machine.

Our standards will be a group of samples (usually artificial) containing no analyte (e.g a tube with water) or a known value of the analyte (e.g. a tube with liquid containing 10.2 parts per million of iron). Standards containing no analyte are usually called *blanks*, and the known values are often called *nominal* data. Excepting the blanks, the nominal values of the standards are expected to have an associated uncertainty (e.g. [Fe]=10.2 $\pm$ 0.1 ppm).

Also, the machine we are using will be probably repeating the measurement on the same sample for 3 times or more. Therefore, the measured data will also have an associated uncertainty, usually the standard deviation of the measurements from the same sample.

3.2 Calibration tools

Plotting data with x and y errors

The first thing we should do to start calibrating any data is compare it graphically.

Create the script **my_first_calibration.m** and copy the following ICP calibration data:

```
% Nominal concentrations of iron in some standards
STDSnominal=[0,0.98,4.56,10.78,19.34,0,1.05,5.1,9.94,18.95];
% associated uncertainties
STDSnominal_uncert=[0,0.02,0.06,0.11,0.19,0,0.02,0.06,0.10,0.19];
% ICP measured values of standards in counts per second (cps)
STDScps=[425,1724,7443,15221,30973,146,1832,7378,15124,27701];
% associated uncertainties
STDScps_uncert=[214,140,377,329,381,249,311,280,1129,1140];
```

The simplest way of representing these data with uncertainties is:

```
x=STDSnominal;
dx=STDSnominal_uncert;
y=STDScps;
dy=STDScps_uncert;

figure
hold on
for n=1:length(x)
  plot([x(n),x(n)],[y(n)-dy(n),y(n)+dy(n)],'-b')
  plot([x(n)+dx(n),x(n)-dx(n)],[y(n),y(n)],'-b')
end
plot(x,y,'.b')
```

However, when we have both x and y uncertainties, the error bars do not fully represent the probability distribution of the data in the 2-D space. Assuming that σ_x and σ_y are independent (no coviariance), the probability distribution at a point (x_i , y_i) can be defined by its χ^2 value respect our data x, y, σ_x and σ_y:

$$\chi^2 = (\tfrac{x_i - x}{\sigma_x})^2 + (\tfrac{y_i - y}{\sigma_y})^2$$

Assuming that all the points with $\chi^2 = 1$ are at the one-sigma confidence level boundary, we could solve the previous equation as:

$$x_i = x + \sigma_x \cdot \cos(\theta)$$
$$y_i = y + \sigma_y \cdot \sin(\theta)$$

being θ between 0 and $2 \cdot \pi$ (note that $\sin^2(\theta) + \cos^2(\theta)$ is always 1).

We can use this property to draw the ellipses corresponding to our data within uncertainties:

```
figure
hold on
for n=1:length(x)
  theta=linspace(0,2*pi,100);
  xi=x(n)+dx(n)*cos(theta);
  yi=y(n)+dy(n)*sin(theta);
  plot(xi,yi,'-b') % plot ellipse
end
plot(x,y,'+b') % plot central point
```

Note that ellipsis from samples with no uncertainty in one of the axis (e.g. blanks) look exactly as error bars.

Linear regression

A line is the simplest way of relating 2 sets of data (e.g. known concentrations and signals given by a machine). One of the most used methods to fit a line to our dataset is the "least-squares" regression. This method minimizes the square of the distances between the line and our data. Fortunately, there is a direct solution to solve this problem. The general formulas to fit a line $y = a \cdot x + b$ to n data by least-squares are:

$$a = \frac{n \cdot (\sum x_i y_i) - (\sum x_i) \cdot (\sum y_i)}{n \cdot (\sum x_i^2) - (\sum x_i)^2}$$
$$b = \frac{(\sum x_i^2) \cdot (\sum y_i) - (\sum x_i) \cdot (\sum x_i y_i)}{n \cdot \sum x_i^2 - (\sum x_i)^2}$$

In MATLAB/Octave, we can create the function `leastsquares.m` as:

```
function [ myfit ] = leastsquares( x,y )
  a=(length(x)*sum(x.*y)-sum(x)*sum(y))/(length(x)*sum(x.^2)-sum(x)^2);
  b=(sum(x.^2)*sum(y)-sum(x)*sum(x.*y))/(length(x)*sum(x.^2)-sum(x)^2);
  myfit = @(x) a*x+b;
end
```

and the average error of the data **calibrated** using `leastsquares` will be:

```
myfiterror= mean(abs(y-myfit(x))
```

Exercises:

- Plot the ICP data from slide 3.2 together with its linear fit.
- Use this data to calibrate a measurement of 9000 cps.
- How would you propagate the uncertainty of the calibration?

- Does `myfiterror` fully represent the calibration uncertainty?
  We have not used the uncertainties in our calculations!

Interpolation and smoothing

When we have a curve defined as $y = f(x)$ we might be interested in getting the x_i value corresponding to a y_i. This is the case of the function `myfit(x)`, where x represent concentrations and y are signals obtained by ICP. The function `leastsquares` is a line and it would not be difficult to calculte the inverse function mathematically: $y = a \cdot x + b \Rightarrow x = (y - b)/a$.

However, we often need to fill the gaps from *incomplete* datasets. For example, the file `gistemp.csv` contains an estimate of global surface temperature change every 5 years (GISTEMP data: http://data.giss.nasa.gov/gistemp/).

Create a new script,load the data and plot it:

```
gistempdata=csvread('gistemp.csv',1,0);
years=gistempdata(:,1);
temp=gistempdata(:,2);

figure
hold on
plot(years,temp,'*k')
```

If we want to estimate the global surface temperature change every year, we need to interpolate the data. To do so, we can use the built-in function `interp1`. By default `interp1(x,y,x0)` will return the linear interpolation of the `x,y` dataset at `x0`. Try:

```
myyears=min(years):1:max(years);
mytemp=interp1(years,temp,myyears);
plot(myyears,mytemp,'.-r')
```

To avoid errors, the `x` values in texttttinterp1(x,y,x0) should be sorted and not repeated. If your data is not sorted, you can use sort the data using `sort: [x2,order]=sort(x); y2=y(order);`

The linear interpolation is the default method used by `interp1`, so `interp1(x,y,x0)` is equivalent to `interp1(x,y,x0,'linear')`. But we can use other methods, such as `'spline'`, or `'nearest'` to interpolate our data. To see the differences, try:

```
myyears=min(years):1:max(years);
mytemp=interp1(years,temp,myyears,'spline');
plot(myyears,mytemp,'.-b')
```

Apart of the method, we can ask `interp1` to also extrapolate data by adding `'extrap'` after the method. If you want to know more about `interp1`, type `help interp1`.

Exercise: use extrapolation to predict the global surface temperature change during the next century.

In other cases, instead of increasing the resolution of our data, we might be interested in *smoothing* it (e.g. to remove high frequency noise). For example:

```
smoothingtime=50;
yearssmooth=1900:10:2000;
for n=1:length(yearssmooth)
   % select data around the year yearssmooth(n)
   selecteddata=(abs(yearssmooth(n)-years)<smoothingtime/2);
   tempsmooth(n)=mean(temp(selecteddata));
end
plot(yearssmooth,tempsmooth,'-m')
```

This method is called "moving average".

Exercise: extrapolate the smoothed temperatures to predict the global surface temperature change during the next century.

3.3 Spectrometry data

Blank Equivalent Concentration

In spectrometry,the Blank Equivalent Concentration (BEC) is defined as the concentration that would correspond to the signal of the blank. It is usually determined by the following formula:

$$BEC = \frac{I_{blank}}{I_{standard} - I_{blank}} \cdot C_{standard}$$

being I the signals measured, usually in counts per second (cps), and C the nominal concentration. Considering that we are going to be working with datasets involving several standards, we can define the BEC graphically as the negative of the x-intercept of our calibration line.

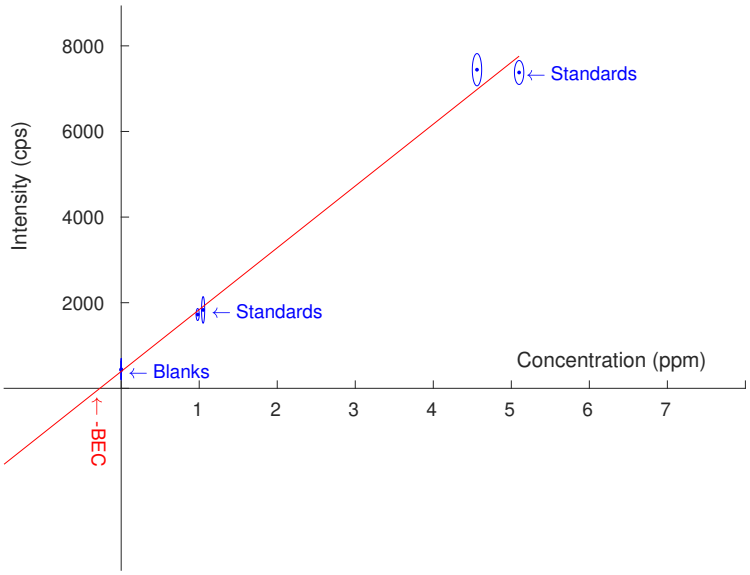


Figure 3.1: Graphical calculation of the Blank Equivalent Concentration (BEC) according to the calibration represented by the red line.

The BEC gives us an idea of how the background level of our machine compares with the measurements of our standards. We could think that a low BEC value implies that our measurements are going to be more precise. However, the precision of the measurements will depend on the **stability** of the background rather than the background value.

The stability of the background is what defines the precision of our measurements, rather then the background value. Similarly, we should calculate the variability of our BEC to get an idea of the noise of our measurements in concentration units. There are different ways of calculating the BEC "noise" (σ_{BEC}). We could just calculate it based on the scatter of our blank data. However, this would not reflect the scatter of our standards. Fig. 3.3 shows the σ_{BEC} calculated from the uncertainty of our calibration.

Exercise: recycle the code generated before (calculation of `myfiterror`*) to calculate the uncertainty of the BEC corresponding to the calibration data shown in the slide 3.2.* Tip: you can use maths to get the inverse of `myfit` or use `interp1`.

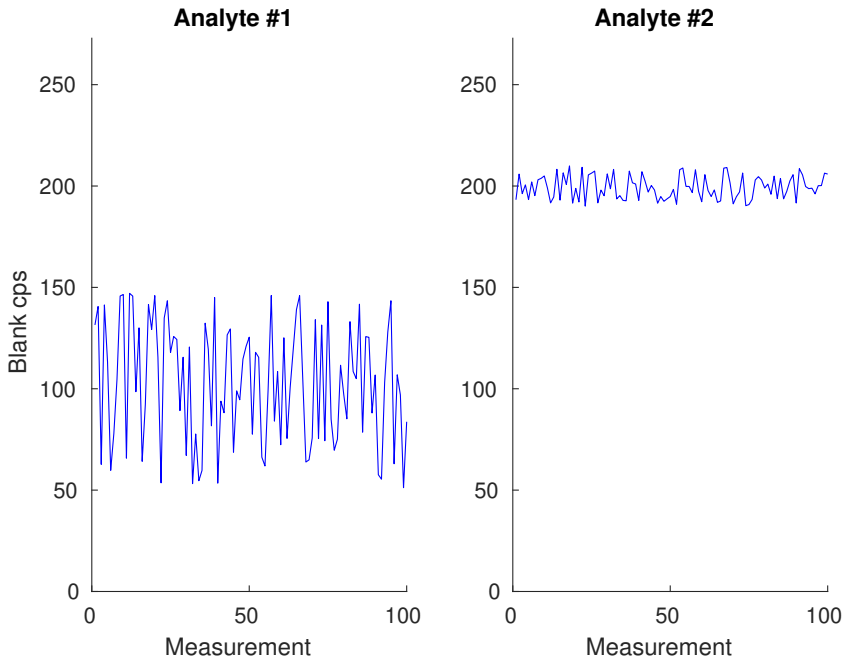


Figure 3.2: Two blank measurements compared. The measurements of the analyte #1 in the blank show a lower background, but the background of the analyte #2 measurements is more precise.

Limits of detection and quantification

The Limit of Detection (LOD) or Detection Limit is defined as the smallest measurable concentration. It is the concentration of a theoretical sample that will produce a signal strong enough to be distinguishable from the background noise. Assuming that this signal is going to have a noise similar to the background, the difference between the signals from the sample and the blank should be bigger than two times the noise. That is why the LOD is numerically defined as $LOD = 3 \cdot noise$. This is often calculated by calibrating the concentration corresponding to $I_{blank} + 3 \cdot \sigma_{I_{blank}}$, being $\sigma_{I_{blank}}$ the standard deviation of the intensities measured on blank samples.

This approach assumes that our theoretical smallest measurable samples will produce a signal as scattered as our blank. However, sometimes, the measures of our samples are more similar to standards than to blanks (e.g. due to matrix effects). This is why considering $LOD = 3 \cdot \sigma_{BEC}$ would be a more conservative way of calculating our LOD.

Likewise, the Limit of Quantification (LOQ) is usually defined as 10 times the blank noise, so the uncertainty associated with the lowest sample that can produce quantitative data is $\sim 10\%$. As for the LOD, we can calculate the LOQ using the BEC uncertainty: $LOQ = 10 \cdot \sigma_{BEC}$.

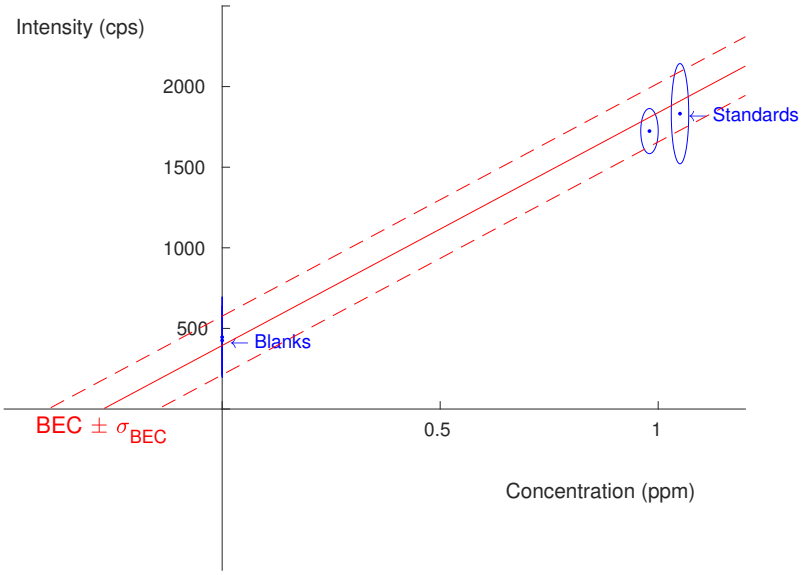


Figure 3.3: Graphical calculation of the Blank Equivalent Concentration (BEC) and its uncertainty σ_{BEC}.

Calibrating data

Once we have calculated the algorithms that relate the ICP signal with concentrations and concentration uncertainties, we are ready to calibrate the signals from our "unknown samples" in our script `my_first_calibration.m`:

```
% ICP measured values of unknowns in counts per second (cps)
SAMPLEScps=[9782,28746,13471,5870,28173,30492,13739,3588,813,12805];
% associated uncertainties
SAMPLEScps_uncert=[181,1042,1214,76,2899,2532,809,243,275,716];
```

To make this easier, we can define our calibration line as a reference `xcal` and `ycal`:

```
xcal=linspace(min(x),max(x),100);
ycal=myfit(xcal);
```

and transform the signals `yunk` into concentrations `xunk` using `interp1`:

```
yunk=SAMPLEScps; dyunk=SAMPLEScps_uncert;
```

```
xunk=interp1(ycal,xcal,yunk,'linear','extrap')
```

As the calibration is a line, we can also transform the measurement uncertainties into concentrations using:

```
measurementuncert=...
    ( interp1(ycal,xcal,yunk+dyunk,'linear','extrap')-...
      interp1(ycal,xcal,yunk-dyunk,'linear','extrap') )/2
```

The measurement uncertainty is the **internal uncertainty** of our data, which is the errors we should use to compare our samples between them. However, as usually we want to compare our data with data that has not been calibrated simultaneously (e.g. samples measured one month ago), we should also include the calibration uncertainty into the **external uncertainty** (`dxunk`):

```
calibrationuncert=...
  ( interp1(ycal,xcal,yunk+myfiterror,'linear','extrap')-...
    interp1(ycal,xcal,yunk-myfiterror,'linear','extrap') )/2
dxunk=sqrt(calibrationuncert.^2+measurementuncert.^2);
```

Including the graphical representation of the unknown data, the script **my_first_calibration.m** could be something similar to this:

```
%% This is my first calibration script

clear % clear all previous data
close all hidden % close all figures

% Nominal concentrations of iron in some standards
STDSnominal=[0,0.98,4.56,10.78,19.34,0,1.05,5.1,9.94,18.95];
% associated uncertainties
STDSnominal_uncert=[0,0.02,0.06,0.11,0.19,0,0.02,0.06,0.10,0.19];
% ICP measured values of standards in counts per second (cps)
STDScps=[425,1724,7443,15221,30973,146,1832,7378,15124,27701];
% associated uncertainties
STDScps_uncert=[214,140,377,329,381,249,311,280,1129,1140];
x=STDSnominal; dx=STDSnominal_uncert;
y=STDScps; dy=STDScps_uncert;

% ICP measured values of unknowns in counts per second (cps)
SAMPLEScps=[9782,28746,13471,5870,28173,30492,13739,3588,813,12805];
% associated uncertainties
SAMPLEScps_uncert=[181,1042,1214,76,2899,2532,809,243,275,716];
yunk=SAMPLEScps; dyunk=SAMPLEScps_uncert;

%% Calibration and uncertainty
myfit = leastsquares(x,y);
```

```
myfiterror= mean(abs(y-myfit(x)));

%% Calibration line
xcal=linspace(min(x),max(x),100);
ycal=myfit(xcal);
ycalerror=myfiterror;

%% BEC, LOD, LOQ
bec=-interp1(ycal,xcal,0,'linear','extrap');
dbec=interp1(ycal,xcal,myfiterror,'linear','extrap')+bec;
LOD=3*dbec;
LOQ=10*dbec;

%% Calibrate unknowns
xunk=interp1(ycal,xcal,yunk,'linear','extrap');
calibrationuncert=...
  interp1(ycal,xcal,yunk+myfiterror,'linear','extrap')-xunk;
measurementuncert=...
  interp1(ycal,xcal,yunk+dyunk,'linear','extrap')-xunk;
dxunk=sqrt(calibrationuncert.^2+measurementuncert.^2);

%% Start a figure
figure
hold on

% plot the unknowns with error-bars
for n=1:length(xunk)
  plot(xunk(n),yunk(n),'.k')
  plot([xunk(n)-dxunk(n),xunk(n)+dxunk(n)],[yunk(n),yunk(n)],'-k')
  plot([xunk(n),xunk(n)],[yunk(n)-dyunk(n),yunk(n)+dyunk(n)],'-k')
end

% plot the standards with ellipsis
for n=1:length(x)
  theta=linspace(0,2*pi,100);
  xi=x(n)+dx(n)*cos(theta);
  yi=y(n)+dy(n)*sin(theta);
  plot(xi,yi,'-b')
end
plot(x,y,'.b')

% plot the calibration
plot(xcal,ycal,'-r')
plot(xcal,ycal+myfiterror,'--r')
plot(xcal,ycal-myfiterror,'--r')

% put labels
ylabel('Intensity (cps)')
xlabel('Concentration (ppm)')
```

And the generated figure will be similar to this:

- As you can see in the figure, the previous script overestimate the uncertainties of the lowest concentrations by considering `myfiterror` as a con-

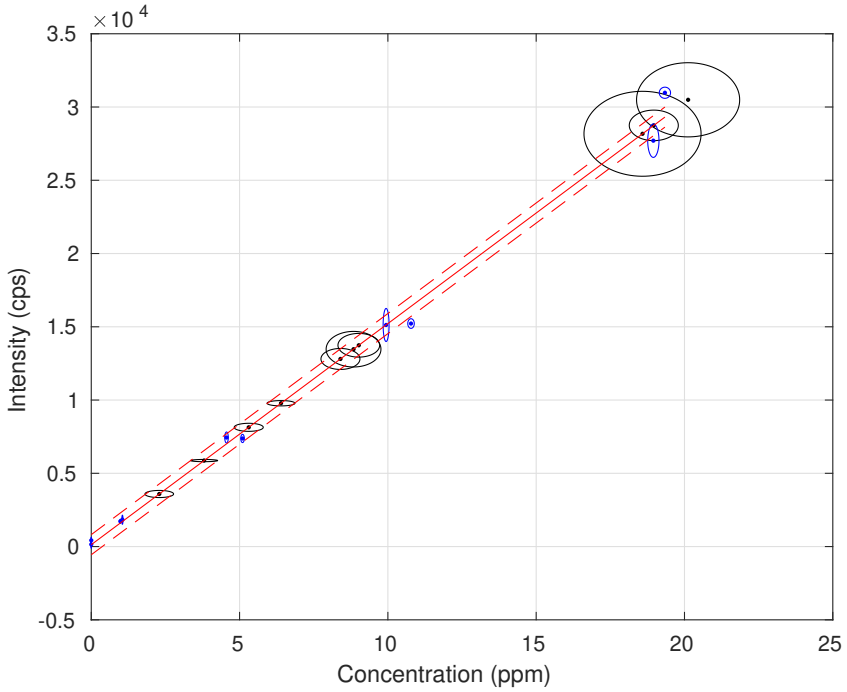


Figure 3.4: Example of figure output of a calibration. Blanks and standards are depicted in blue, unknowns in black and the calibration line within uncertainty in red.

stant, where it should be a function of the ICP signal. Therefore, we obtain overestimated LOD and LOQs. You can try to improve the simplistic `mean(abs(y-myfit(x)))` with some other code.

- Also, note that we have ignored the uncertainties of the calibration data `STDSnominal_uncert` and `STDScps_uncert`. We should also transmit those uncertainties in the external uncertainties. We could do that mathematically (calculating the partial derivatives for the formulas in the `leastsquares.m` function) or programmatically (fitting a large number of different lines with data generated using `normrnd`).

An example of a full propagation of uncertainties is shown in `my_second_calibration.m`.

Finally, you can convert your script into a function that you can use in the future to calibrate your own data:

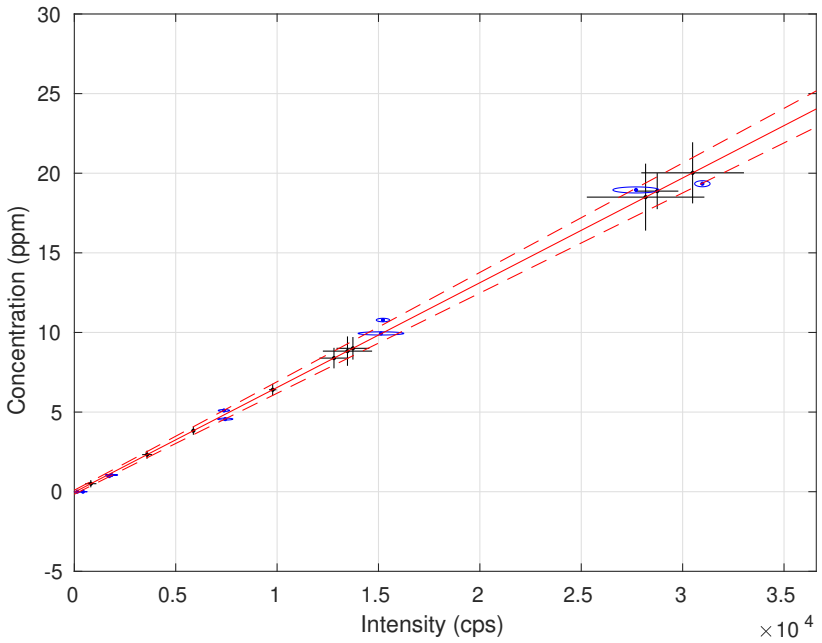


Figure 3.5: Another example of calibration. Here, the scatter and the uncertainties of the standards are fully propagated.

```
function [ SAMPLESppm SAMPLESppm_uncert] = calibfunction(STDSppm,STDSppm_uncert,...
                                                         STDScps,STDScps_uncert,...
                                                         SAMPLEScps,SAMPLEScps_uncert)

    % Paste here some of the code used in my_first_calibration
    %  - Remember no to paste the input data (STDSppm,SAMPLEScps, etc.)
    %  - Do not paste the plotting code unless you need it!

end
```

Reporting data with uncertainties

At the end of our scripts we will probably want to include an easy way of exporting the numerical results we obtained, like the concentrations and uncertainties `xunk` and `dxunk`. The simplest way is by printing what we want in the command window using the built in function `disp`. As we want to mix our numerical parameters with strings, we will have to use `num2str` to transform our numbers into strings.

```
clc % clear the command window
```

```
disp(['Blank equivalent concentration: ' num2str(bec) ' ppm'])
disp(['Limit of detection: ' num2str(LOD) ' ppm'])
disp(['Limit of quantification: ' num2str(LOQ) ' ppm'])
disp(['Concentrations and uncertainties:'])
for n=1:length(xunk)
  disp([num2str(xunk(n)) ' +/- ' num2str(dxunk(n))])
end
```

If we want to be able to paste our copy and paste our data in Excel, we can replace `' +/- '` *by the tab character* `char(9)`.

A common mistake when reporting data with errors is using more digits than the significant figures. For example, the last numbers of 6.3976 from the data 6.3976 ± 0.46537 are meaningless. A couple of significant figures in the uncertainty is usually enough, so in our report we should just write 6.40 ± 0.47, as this distribution is identical to 6.3976 ± 0.46537. We can use `round` to trim useless digits from our `mu±sigma` data:

```
decimalpositions=1-floor(log10(sigma));
newmu=round(mu,decimalpositions);
newsigma=round(sigma,decimalpositions);
```

Exercise: Add the "reporting code" in `my_first_calibration.m` to:

- Displays the concentrations below the LOD as "$< LOD$ ppm".
- Displays the concentrations below the LOQ as "$\sim$ *conc.* ppm".
- Displays the concentrations above the LOQ "*conc.* $\pm$ *uncert.* ppm" using only their significant figures.

3.4 Exercise: ICP-OES data calibration

ICP-OES data calibration

The file `ICPdata_GU20171012.csv` contains real raw ICP-OES data exported from the ICP machine (one line per analyte in chronological order) and an extra column with the nominal concentrations of the standards in ppm.

Write a script that reduces the ICP-OES data and report calibrated concentrations for each analyte.

Tips

Some useful functions that you might need:

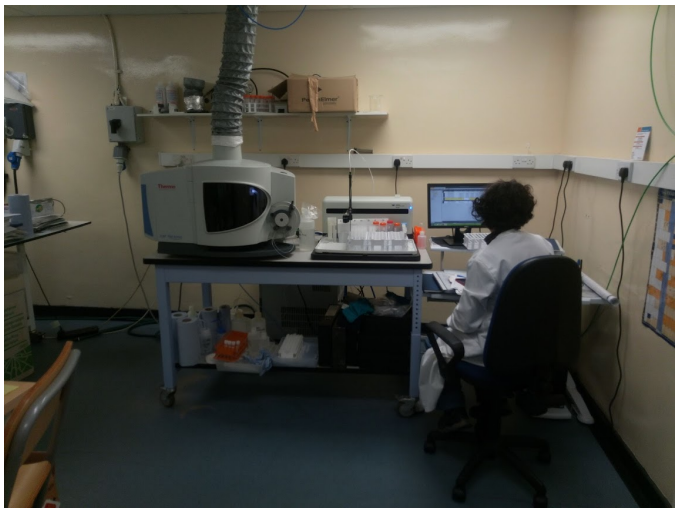


Figure 3.6: ICP-OES at SUERC.

- `fopen` and `textscan` from chapter 2.
- `STD=~isnan(nominal)` select lines containing nominal values (the values in the `nominal` array are **not-not**-a-number).
- `unique(analytename)` returns a list of unique values of `analytename`.
- `strcmp(string, list_of_strings)` returns the positions of the `string` in the array `list_of_strings`. This is useful when you want to work with only the lines that refer to a specific analyte. E.g. inside this loop:
  ```
  for this_analyte=unique(analytename)'
  selectdata=strcmp(this_analyte, analytename);
  ...and work with selectdata in here...
  end.
  ```

Chapter 4

Modeling

Numerical models

Numerical models are widely used to solve physical or chemical problems by describing processes using equations and numbers.

We can generally describe a numerical model as $y = F(x)$, where x are the causes of the process, F are the algorithms that describe the process, and y are the consequences of the process.

In Earth Sciences, we typically *know* the consequences (y) of the process we are studying (e.g. a concentration of something as a result of time, that will be the main *cause* in a geochronology problem), and we want to know the causes x. Therefore, finding or approximating the inverse model $x = F'(y)$ would be very useful for our purposes.

The way we solve a problem involving numerical models will depend on whether or not we can find or approximate the inverse model.

4.1 Forward problem

Forward problem

Forward models as $y = F(x)$ are used to make informed predictions. However, when we can find the inverse of our model $x = F'(y)$, we can solve our problem directly.

In geochronology, this $x = F'(y)$ typically means expressing the *true age* of a sample as a function of concentrations and other known parameters.

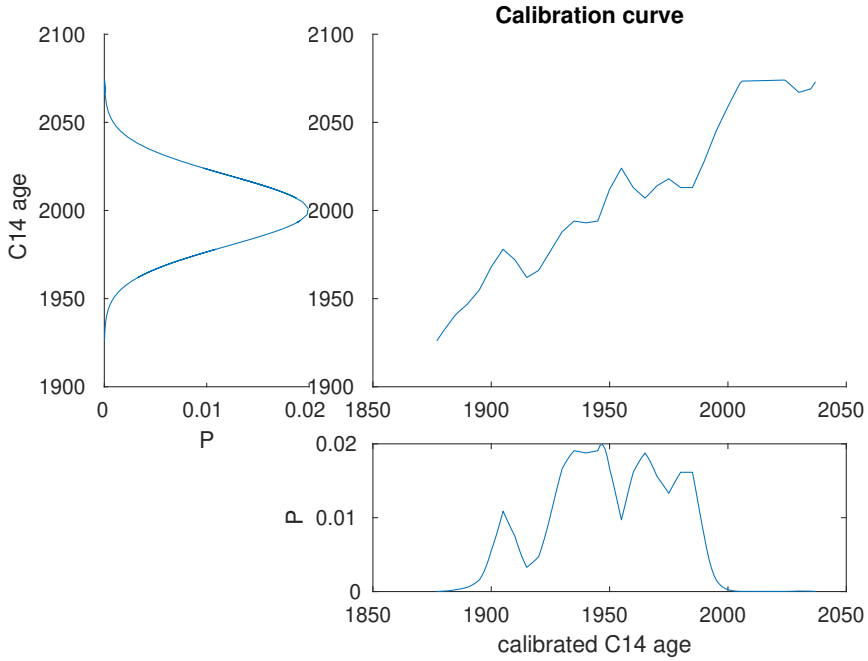
Mathematically, a forward problem is a *well-posed problem*, where a unique solution exists and the solutions change continuously with the known parameters.

Radiocarbon calibration

A good example of a forward problem is the calculation of a calibrated ^{14}C age from an apparent ^{14}C age, as the ^{14}C ages calculated in the slide 1.10.

Apparent radiocarbon ages are calibrated using calibration lines. You can download the data corresponding to a calibration line from here:
https://journals.uair.arizona.edu/index.php/radiocarbon/article/downloadSuppFile/16947/275
You should get a file called `16947-25973-2-SP.14c`. Save it in your working folder and open it as text: you will find out that is a comma separated file (csv).

If we have a radiocarbon age (e.g. 2000 ± 20), we should be able to create a script to calibrate the entire probability distribution of the age (see slide **??**) along the calibration curve using `interp1` and produce an output like this:



(code in the next slides)

```
%% import calibration curve
fid = fopen('16947-25973-2-SP.14c');
imported = textscan(fid, '%f %f %f %f %f',...
'HeaderLines', 12,'Delimiter',',');
fclose(fid);

% select the data from the calibnration curve
calBP=imported{1};
C14=imported{2};

%% interpolate calibration curve to arrays of 1 position per calibrated year
refcalBP=min(calBP):1:max(calBP);
refC14=interp1(calBP,C14,refcalBP);
```

```
%% input our data
C14age=2000;
C14ageerr=20; % one sigma error

% calculate probabilities of my data
C14probs=normalprobs(refC14,C14age,C14ageerr);

%% Plot the calibration curve and our data
% select only the "most" probable data to plot
sel=(C14probs>max(C14probs)/1000);

figure % start figure
% plot the part of the calibration curve that is relevent for us
subplot(3,3,[2 6])
plot(refcalBP(sel),refC14(sel))
title('Calibration curve')

% plot our data
subplot(3,3,[1 4])
plot(C14probs(sel),refC14(sel))
ylabel('C14 age')
xlabel('P')

% plot the data calibrated
subplot(3,3,[8 9])
plot(refcalBP(sel),C14probs(sel))
xlabel('calibrated C14 age')
ylabel('P')
```

Note that:

- We are ignoring the uncertainty of the calibration curve (`imported{3}`). To know how to incorporate all the errors, check the script "MatCal" by Lougheed & Obrochta (2016):
  http://dx.doi.org/10.5334/jors.130
- To represent several subplots in the same window we are using `subplot(r,c,[a b])`, where `r` and `c` are the number of rows and columns, and `a` and `b` are corners of the area where we want to plot. E.g. `subplot(3,4,[7 12])` would start plotting in the blue area:

1	2	3	4
5	6	7	8
9	10	11	12

4.2 Inverse problem

Inverse problem

Sometimes it is not possible to express our problem as $x = F'(y)$, because sometimes it is impossible to get the inverse of $F(x)$.

Most geochemical models used in geochronology allow the calculation of a theoretical concentration or measurable signal C as a function of time t and other parameters: $C = f(t, C_0, ...)$. However, some of these problem cannot be solved for $t = f(C, C_0, ...)$. In this cases, we will need to *guess* the age t corresponding to our known concentrations C, C_0, etc.

Ill-posed problem

Mathematically, this kind of problems are often *ill-posed problems*. Therefore, we cannot assume that they have a unique solution and we should check the sensitivity of our results to a change in our known parameters.

In geochronology, this means that apart of answering the main question:

Which ages are compatible with my data?

but we should also answer the question:

Which ages are **not** compatible with my data?

4.3 Cosmogenic depth-profile dating

Cosmogenic depth-profile dating

The accumulation of ^{10}Be under a sedimentary surface depends on the inherited ^{10}Be concentration (C_0), the different ^{10}Be production rates ($P_{sp.}$, $P_{f\mu}$ and P_{μ^-}) and attenuation lengths ($\Lambda_{sp.}$, $\Lambda_{f\mu}$ and Λ_{μ^-}), the ^{10}Be decay constant (λ), the density of the sediment (ρ), the depth (z), the erosion rate of the surface (ϵ) and the age of the landform (t):

$$C = C_0 + \frac{P_{sp.}}{\frac{\varepsilon}{\Lambda_{sp.}}+\lambda} e^{-\frac{z\cdot\rho}{\Lambda_{sp.}}} \left(1 - e^{-t\left(\lambda+\frac{\varepsilon}{\Lambda_{sp.}}\right)}\right) + \frac{P_{\mu^-}}{\frac{\varepsilon}{\Lambda_{\mu^-}}+\lambda} e^{-\frac{z\cdot\rho}{\Lambda_{\mu^-}}} \left(1 - e^{-t\left(\lambda+\frac{\varepsilon}{\Lambda_{\mu^-}}\right)}\right) + \frac{P_{f\mu}}{\frac{\varepsilon}{\Lambda_{f\mu}}+\lambda} e^{-\frac{z\cdot\rho}{\Lambda_{f\mu}}} \left(1 - e^{-t\left(\lambda+\frac{\varepsilon}{\Lambda_{f\mu}}\right)}\right) \quad (4.1)$$

This equation cannot be solved for t. Also, when we have a dataset of ^{10}Be concentrations under a surface (a ^{10}Be depth-profile), we want to solve the problem for C_0, ϵ and t. *How can we do this?*

^{10}Be accumulation model

The following function calculates theoretical ^{10}Be concentrations:

```
function [ C ] = exposure_model(P,L,l,density,z,C0,erosion,t)
```

```
  C=C0+...
    P(1)./(l+erosion.*density./L(1)).*exp(-z.*density./L(1)).*...
    (1-exp(-(l+erosion.*density./L(1)).*t))+...
    P(2)./(l+erosion.*density./L(2)).*exp(-z.*density./L(2)).*...
    (1-exp(-(l+erosion.*density./L(2)).*t))+...
    P(3)./(l+erosion.*density./L(3)).*exp(-z.*density./L(3)).*...
    (1-exp(-(l+erosion.*density./L(3)).*t));
end
```

^{10}Be data

The following code defines all the known parameters and the ^{10}Be concentrations from the sampled depth-profile for an alluvial fan in Almería (Spain):

```
%% Production rates
P=[4.35,0.0985,0.0855]; % production rates in at/g/a
L=[160,1137,1842]; % attenauation lengths in g/cm^2
l=4.9975E-7; % decay contant in a^(-1)

%% Field data
density=1.8; % g/cm^3
z=[267,195,141,95,46,3]; % depth of the sameples in cm
Be10=[91000,184000,265000,430000,732000,1070000]; % 10Be concentrations in atoms/g
Be10error=[9100,16000,18000,29000,61000,81000]; % 10Be uncertainties in atoms/g
```

Try `exposure_model(P,L,l,density,10,0,0.0001,10000)` to calculate the ^{10}Be concentration accumulated in a sample 10 cm below a 10 ka old surface being eroded at a rate of 1 mm/ka (0.0001 cm/a).

We can reproduce the theoretical depth profile for these conditions along the first 3 m under the surface:

```
zref=0:300; % depth reference in cm
concentrations=exposure_model(P,L,l,density,zref,0,0.0001,10000);
plot(concentrations,-zref,'-b')
```

Now we just need to find which theoretical values of inheritance, age and erosion rates (the last 3 parameters in the `exposure_model` function) match our `Be10` concentrations within `Be10error` uncertainties!

4.4 Monte Carlo methods

Monte Carlo methods

The simplest way of guessing the values for `C0,erosion,t` that fit our data `Be10` at our depths `z` could be just trying *a lot* of random values of `C0,erosion,t` and check which theoretical concentrations are closer to our data. This is called a **Monte Carlo experiment**.

To perform this Monte Carlo experiment, we should define a way of measuring how close is our model to our data. A χ^2 function (similar to the one at the slide 2.8) would do the job:

$$\chi^2 = \sum_{sample=1}^{n} \left(\frac{C_{model}(z_{sample}) - C_{sample}}{\sigma_{C_{sample}}} \right)^2$$

The following code runs a Monte-Carlo experiment of 100 000 models assuming that ϵ is between 0 and 50 m/Ma (0.005 cm/a), the landfom age is between less than 3 Ma (`3E6` a), and C_0 is smaller than the lowest concentration.

```
%% Monte carlo experiment
nummodels=100000; % define how many models
C0i=rand(1,nummodels)*min(Be10); % random inheritences
ti=rand(1,nummodels)*3e6; % random ages
erosioni=rand(1,nummodels)*0.005; % random erosion rates
chisquarevalues=rand(1,nummodels)*NaN; % allocate memory for the chi square array

% calculate the chi squared vales for each model
for n=1:nummodels
    % calculate the model concetratios for the depths z
    Cmodel=exposure_model(P,L,l,density,z,C0i(n),erosioni(n),ti(n));
    % calculate the chi squared for this model
    chisquarevalues(n)=sum(((Cmodel-Be10)./Be10error).^2);
end
```

Which models should we consider to represent the uncertainty of the results?

When fitting a model to data, we have to report how many parameters are we trying to fit and how many data we have. The number of parameters should be lower that the data points and the difference between them are the **Degrees of Freedom** of our model. In our model we have
DOF = 6 - 3 = 3 degrees of freedom.

When performing this kind of inverse modeling, the models that fit the data with a χ^2 value below the minimum χ^2 value plus the degrees of freedom are often considered to fit the data within one sigma confidence level.

Therefore, we can calculate which of the models fit our data within one-sigma, assuming that this is defined by the models with χ^2 values between the minimum χ^2 and χ^2+DOF:

```
DOF=3; % degrees of freedom (# of samples - # of parameters)
minchi=min(chisquarevalues); % minimum chi-square value (best model)
best=find(chisquarevalues==minchi); % location of the best model
% location of the models fitting the data within one-sigma
onesigma=find(chisquarevalues<minchi+DOF);

% display previous infrormation
disp(['Min chi-squared value = ' num2str(minchi)])
disp([num2str(length(onesigma)) ' modles fitting one sigma'])
disp(['Age: ' num2str(min(ti(onesigma))/1e3) ' - '...
    num2str(max(ti(onesigma))/1e3) ' ka'])
disp(['Erosion: ' num2str(min(erosioni(onesigma))*1e4) ' - '...
    num2str(max(erosioni(onesigma))*1e4) ' m/Ma'])
disp(['Inheritance: ' num2str(min(C0i(onesigma))) ' - '...
    num2str(max(C0i(onesigma))) ' atoms/g'])
```

- What is the best result?
- Does it fit the data well? ($\chi^2 \simeq 0$)
- Plot randomized values against their corresponding χ^2 values to get a idea of the distribution of the results. *Tip: plot only the χ^2 values below the best value+10.*
- Plot the sample ^{10}Be concentrations and the theoretical ^{10}Be of the best fit.
- Select the models fitting the data within one-sigma confidence level and plot these models in grey (`'Color',[0.7 0.7 0.7]`).

We should get a high number of fitting models to get an idea of which the distribution of the parameters values that fit our data. *Try increasing the number of random models to get at least 300 fitting models.*

4.5 Convergence methods

Convergence

Another way of getting more models fitting our data is changing the limits of the randomized parameters while generating more models. For example, until now we have been considering ages that are between 0 and 3 Ma, but after running about 10^5 models, it is pretty clear that the ages fitting the data are lower than 1 Ma, and that the erosion rates should be lower than 5 m/Ma (0.0005 cm/a). *Re-run your solver applying better limits.*

We can even program our solver to start converging after a *learning process* of a certain number of models, automatizing what we have just done manually. However, we should be cautious making our random models to converge very fast because we can miss solutions that fit our data. To avoid that, we could make them converge to χ^2 values $< \chi^2_{min.} + 10 \cdot DOF$. Actually, we should also allow our random models to diverge out of the initial limits when we find good solutions close to our limits.

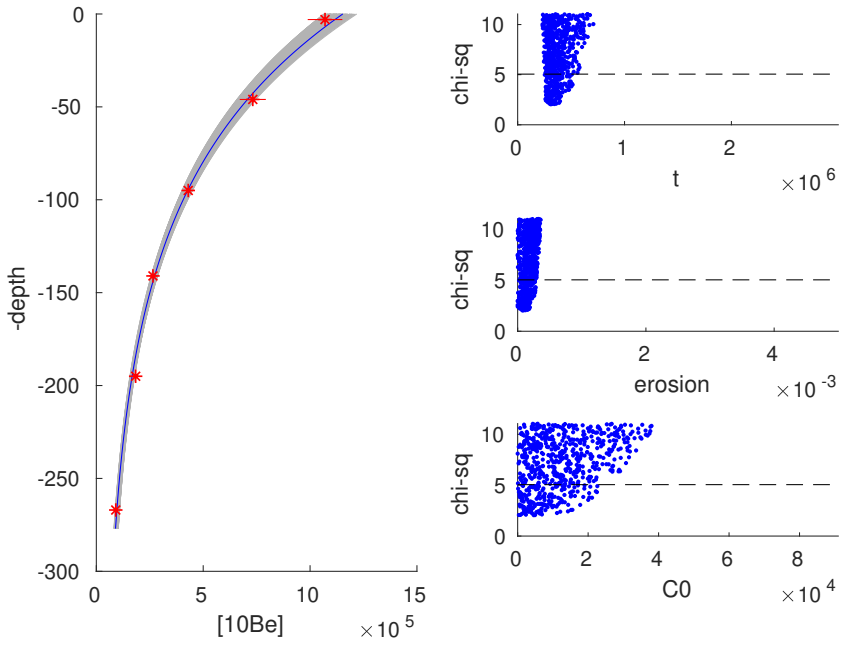


Figure 4.1: Mote Carlo simulations.

4.6 Goal-seeking algorithms

Goal-seeking algorithms

Until now, we have been solving the question *"Which ages are compatible with my data?"*, but are not explicitly answering the question *"Which ages are **not** compatible with my data?"*.

The cosmogenic depth-profile models often show that the fitting models are scattered towards old ages. This is because the fitting area in the $\epsilon - t$ space is a narrow valley that we can easily miss when randomizing the ϵ and t values.

To avoid this, we can randomize only the C_0 and t parameters and make our program to seek actively which is the best ϵ that fit the data for each of the models. This operation is sometimes called *"χ^2 minimization"*. χ^2 minimization slows down our program but will guarantee that **the models outside the fitting age range do not fit our data**.

To minimize the value of χ^2 we can use some built-in functions as `fminunc` or `fminsearch` (type `help fminsearch` for more information). However, the way these algorithms work change in the different versions of MATLAB and Octave, so we will never be sure that our program is going to work the same way in someone else computer. Therefore, it is highly recommended to build our own minimization algorithm.

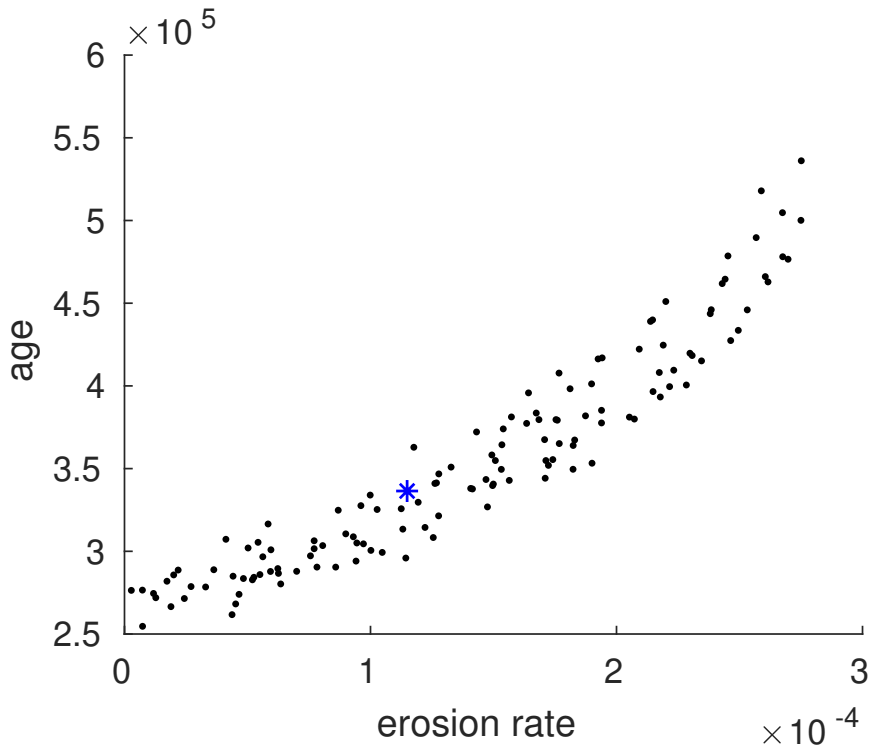


Figure 4.2: Erosion rate-age plot.

An easy solution could be to use `interp1` as a goal seeker of the deviations. The piece of code in the next slide includes this goal seeker in the modeling loop to force getting always the best erosion rate.

How many models do you need to run now to get 300 fitting resutls?

```
% start testing models
for n=1:nummodels
    erosionref=[0,logspace(-5,2,100),10^10]'; % define an array with erosion rates
    % calculate the deviations corresponding to each erosion rate
    % deviations are defined as the sum of (Cmodel-Csample)/Uncertainty
    %     for all the samples
    deviations=...
      sum(...
        (exposure_model(P,L,l,density,z,C0i(n),erosionref,ti(n))-Be10)./...
         Be10error...
      ,2);
```

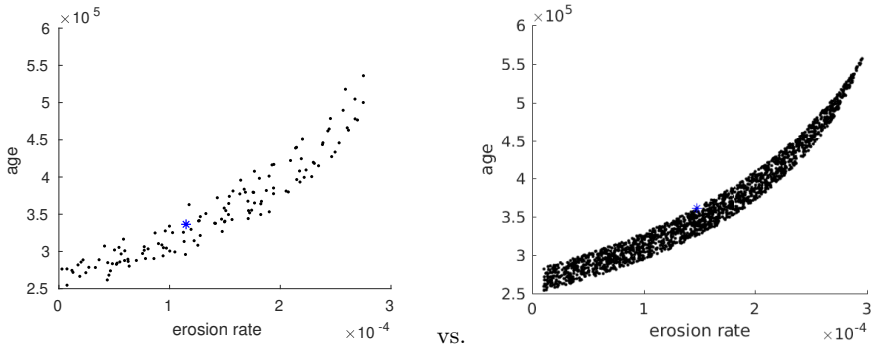
```
    % Interpolate the erosion rate values to find the one the model that
    % fit the data better (for the age and Co corresponding to this random
    % model). THen store the result at erosioni(n), ooverwriting the
    % previously defined value.
    erosioni(n)=interp1(deviations,erosionref,0);

    % calculate the model concetratios for the new erosion rate at the depths z
    Cmodel=exposure_model(P,L,l,density,z,C0i(n),erosioni(n),ti(n));

    % calculate the chi squared for this model
    chisquarevalues(n)=sum(((Cmodel-Be10)./Be10error).^2);
end
```

4.7 Summary

Summary

- Using convergence (and divergence) algorithms help our inverse modeling program to run faster and allow us to set wide starting limits.

- Using goal-seeking algorithms slows down the calculation of each individual simulation. However, it usually allows us to run less models, and also guarantee the reproducibility and accuracy of our results. Compare the two plots representing the solutions in the $\epsilon - t$ space with and without using χ^2 minimization:



Exercise

Use previous models to solve the age of a landform with the following ^{10}Be depth profile:

Sample depth cm	$[^{10}Be]$ 10^3 $atoms/g$
250	25±2
163	45±3
113	60±5
73	100±7
43	140±10
11	200±15

Start testing ages between 0 and 10 Ma and try introducing some convergence code. This will allow you to create a code that will work on any ^{10}Be database.

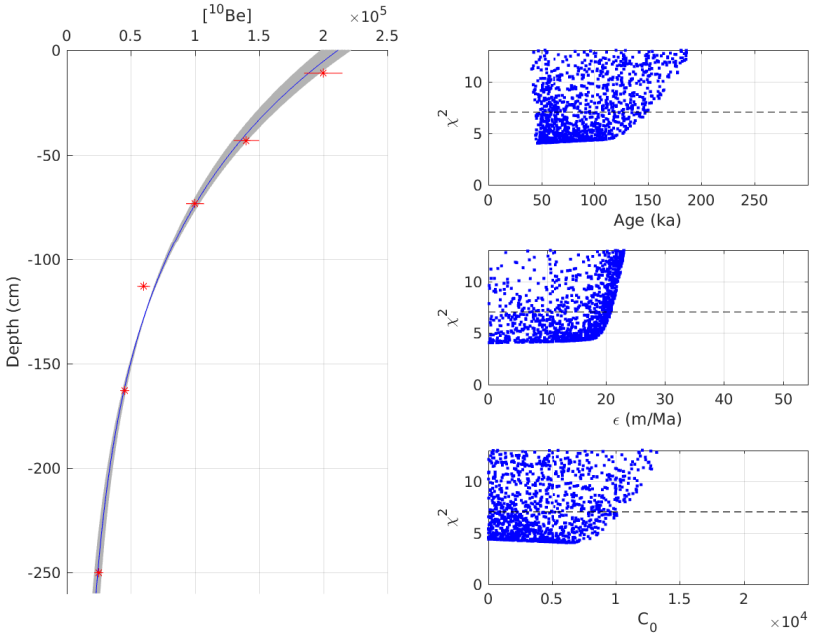


Figure 4.3: Expected result.

Chapter 5

Maps

5.1 Install a toolbox or package

Loading the mapping tools

Most mapping tools are not installed in the basic versions of MATLAB or Octave. To install them, we will need to do the following:

1. In MATLAB we can install the **Mapping Toolbox** if it is not installed yet. Check if it is installed in the Home tab > Add-Ons > Manage Add-Ons. If it is not, you gen get it from *https://www.mathworks.com/products/mapping.html*
2. In Octave you need the **mapping package** (check if you already have it with `pkg list`). If you do not, you can install it by typing *`pkg install -forge mapping`* in the command window or from *https://octave.sourceforge.io/mapping/index.html*
   In Octave, we need to "activate" the packages in every session if we want to use them. To do so type or write at the top of your script *`pkg load mapping`*
   It is also recommended to load the input/output package: *`pkg load io`*

Mapping tools

The mapping tools allow us to:

- Calculate azimuths between two points (*`azimuth`*), get angular distances between coordinates (`distance`), converting angular distances to kilometres (`rad2km`) and many other calculations on maps.
- Read shape files and raster files with `shaperead` and `rasterread`.
- Plot maps using `mapshow`.

However, in geosciences we often need to perform specific calculations on digital elevation models that are only included in the MATLAB Mapping Toolbox (as `gradientm` or

`viewshed`) and other topographic derivates (gradient, flow accumulation, stream order, etc.) that are not included in any official toolbox or package.

5.2 Import a toolbox not included in MATLAB

TopoToolbox

TopoToolbox is a is a MATLAB program for the analysis of digital elevation models (DEMs) developed by Schwanghart & Kuhn (2010) that provides a set of functions that support the analysis of relief and flow pathways in digital elevation models (https://doi.org/10.1016/j.envsoft.2009.

It can be downloaded from https://www.mathworks.com/matlabcentral/fileexchange/50124-topotoolbox

But since the objective of this course is to understand the fundamentals necessary to develop our own tools, we will focus on using the tools included in MATLAB and Octave.

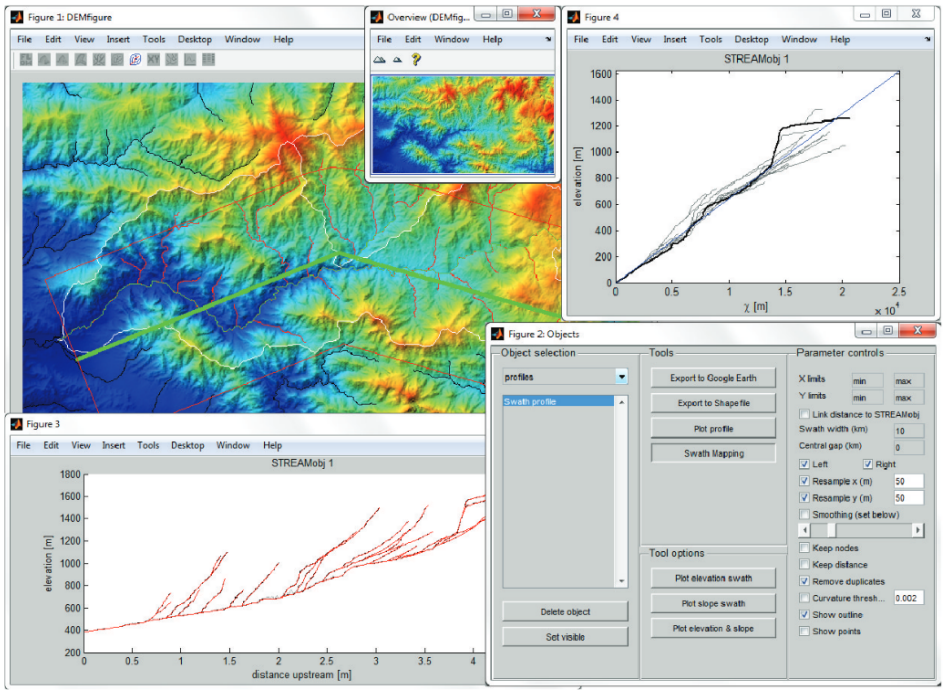


Figure 5.1: Example layout of topoapp, a graphical user interface that enables access to the majority of TopoToolbox functions. See https://doi.org/10.5194/esurf-2-1-2014 for details

5.3 Plotting maps

Plotting maps using the mapping tools

- Calculate the azimuth from SUERC ([lat,lon]=[55.75,-4.16]) and George Square ([lat,lon]=[55.86,-4.25]).
- Calculate the distance in km from SUERC to George Square.
- Download the shapefile of UK from https://biogeo.ucdavis.edu/data/gadm3.6/shp/gadm36_GBR_shp.zip and extract the coastline shape files `gadm36_GBR_0.shp, gadm36_GBR_0.shx, gadm36_GBR_0.prj,` etc. to your folder.
- Plot the map `gadm36_GBR_0.shp` using `mapshow` or `geoshow`.
- Extract the X and Y coast points using `data=shaperead('gadm36_GBR_0.shp')` and plot them using plot(data.X,data.Y,'-r')
- Calculate what is the minimum distance from SUERC to the coast in a straight line. And the maximum!

5.4 Import maps without mapping tools

Import Esri grid files

`.asc` files are widely used to export Digital Elevation Models. They are plain text files (ASCII) containing a matrix of elevations.

At the top of the file they also contain the following information:
- `ncols` and `nrows`: the number of columns and rows of the elevation matrix.
- `xllcorner` and `yllcorner`: the coordinates of the lower left corner of the lower left cell.
- `cellsize`: the actual distance between two cells.

Open the file `Scotland.asc` as text. In this file, the `xllcorner`, `yllcorner` and `cellsize` are in geographic units (decimal degrees of longitude and latitude) and the elevations are in metres, but sometimes all these parameters are in metres. Make sure you know the units before using a `.asc` file!

Import the data in `Scotland.asc` using `textread` to transform the `.asc` data into X, Y and Z coordinates:

```
textdata=textread('Scotland.asc', '%s');
rawelevations=textread('Scotland.asc', '%f','headerlines',6);

ncols=str2double(textdata(2));
nrows=str2double(textdata(4));
xllcorner=str2double(textdata(6));
yllcorner=str2double(textdata(8));
```

```
cellsize=str2double(textdata(10));

X=ones(nrows,ncols).*[xllcorner:cellsize:xllcorner+cellsize*(ncols-1)];
Y=ones(nrows,ncols).*[yllcorner:cellsize:yllcorner+cellsize*(nrows-1)]';
Z=zeros(nrows,ncols);

n=0;
for r=1:nrows
  for c=1:ncols
    n=n+1;
    Z(r,c)=rawelevations(n);
  end
end
```

You have just created a script that reads `.asc` files without toolboxes!

Now you can plot your DEM using any of these plotting tools:

```
surf(X,Y,Z)

meshc(X,Y,Z)

contour(X,Y,Z,[0.1,200:200:2000]) % the 0.1 m contour is the coastline

contour(X,Y,Z,[0.1,500:500:2000],'-b','ShowText','on')

contour3(X,Y,Z,[0.1,50:50:2000])

plot(X(Z>915),Y(Z>915),'.r') % Munros!
```

- Use `interp2` to get the altitudes of SUERC and George Square.
- Calculate the actual distance from SUERC to George Square.
- Plot the elevation transect from SUERC to the closest Munro.
- Plot the elevation histogram of Scotland.

5.5 Geomorphic calculations on DEMs

Calculate slopes

We can calculate the slope vectors using the function `gradient`:

```
% calculate the spacing in metres
xspacing=mean(1000*rad2km(deg2rad(distance(Y(:),X(:),Y(:),X(:)+cellsize))));
yspacing=mean(1000*rad2km(deg2rad(distance(Y(:),X(:),Y(:)+cellsize,X(:)))));
```

```
[dx, dy] = gradient(-Z,xspacing,yspacing);

figure; hold on
contour(X,Y,Z,[0.1,200:200:2000])
quiver(X,Y,dx,dy) % displays slope vectors as arrows
```

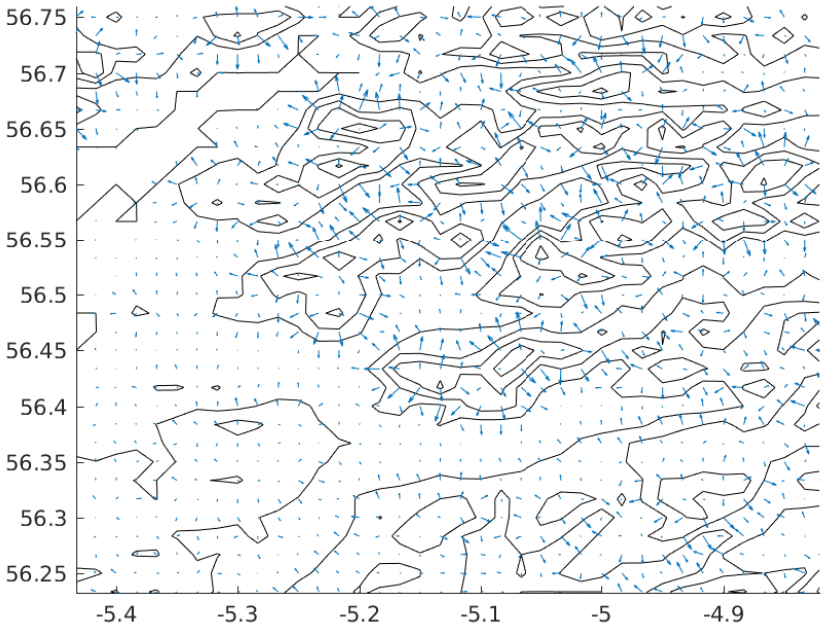


Figure 5.2: `quiver` function displays slope vectors as arrows.

Note that:

- We have to provide the `xspacing` and `yspacing` in metres to get the partial derivatives $\delta z/\delta x$ and $\delta z/\delta y$ (`dx` and `dy`) in m/m (unitless). We could calculate the same as follows:

  ```
  [dxunscaled, dyunscaled] = gradient(-Z);
  dx=dxunscaled/xspacing; dy=dyunscaled/yspacing;
  ```

- The x spacing could not constant in large maps when the points are spaced at constant geographical longitudes (see *`distance(Y(:),X(:),Y(:),X(:)+cellsize)`*). That is why sometimes the `cellsize` is in metres. Here, we are approximating the longitude spacing on the entire map by its average.
- We are calculating the gradient of `-Z` because we want the slope to be positive downhill.

Once we have the gradient, we can calculate the slopes:

$$\nabla z = \sqrt{\left(\frac{\delta z}{\delta x}\right)^2 + \left(\frac{\delta z}{\delta y}\right)^2} \tag{5.1}$$

In Matlab or Octave:

```
slopes=(dx.^2+dy.^2).^0.5;
```

If we want the slope angles in degrees: `rad2deg(atan(slopes))`

Calculate flow direction

We can calculate the azimuth of the flow direction with:

```
flowdir = azimuth(0,0,dy,dx);
```

But for modeling purposes it is useful to simplify the flow directions to the closest neighbouring cell, using 1 of N, 2 for NE, 3 for E and so on:

8	1	2
7	0	3
6	5	4

Using 0 for cells with altitudes $\leq$ than the surroundings. These cells are called *sinks. This simplified way of representing the slopes is very useful to program how the water, or the ice, will move in our map cell by cell.*

We can calculate flow direction following this simple approach:

```
%% Flow direction
FD=0.*Z; %% start with all cells as sinks
% remember that lower rows correspond to lower latitudes
kernelref=[6,5,4;7,0,3;8,1,2]; % direcction references

for r=2:nrows-1 % go though all the cells that are not at the borders
  for c=2:ncols-1
    Zkernel=Z(r-1:r+1,c-1:c+1);
    if Z(r,c)==min(Zkernel(:)) % if sink
      FD(r,c)=0;
    else
      % find the minimum Z
      % get the first one if there are two minimums
      position=find(Zkernel==min(Zkernel(:)),1,'first');
      FD(r,c)=kernelref(position);
    end
  end
end
```

Fill sinks

If you plot the sinks in the map:

```
figure; hold on
sel=(FD==0); % selct sinks
plot(X(sel),Y(sel),'.r') % plot them in red
% overlap the controur plot
contour(X,Y,Z,[0.1,200:200:2000],'-k')
```

You will find that:

1. The borders are considered sinks (as expected)
2. The sea is considered a sink (as expected)
3. There are too many sinks inland! This is a problem for our programming purposes: we know that, even if these sinks are real, the water would fill them and continue its way to the sea.

To better mimic the behaviour of the water in our map, we should be able to produce a flow-direction map where all the *rivers* go to the sea.

DEM manipulating programs usually have an option to fill sinks before calculating flow-direction and flow-accumulation.

We can create our own sink filling algorithm by iteratively adding 1 meter of altitude to all sinks until we reach a DEM with no sinks.

Find an example of sink-filling code in the next slide.

```
%% Fill sinks
disp('Filling sinks...')
Zfilled=Z; % start a new map (identical)
convergence=0;
step=0;
baselevel=0; % sea level
while convergence<1
  step=step+1;
  previoustotal=sum(Zfilled(:));
  for r=2:nrows-1 % ignore map borders
    for c=2:ncols-1 % ignore map borders
      % these are the elevations around Z(r,c), including (r,c)
      Zkernel=Zfilled(r-1:r+1,c-1:c+1);
      % find the minimum Z around (r,c). Ignore central value.
      minimumZkernel=min(Zkernel([1,2,3,4,6,7,8,9]));
      if Zfilled(r,c)>baselevel % do not touch the sea
```

```
        % if it is a sink, add 1 m
        Zfilled(r,c)=max(Zfilled(r,c),minimumZkernel+1);
      end
    end
  end
  % If we reach equilibrium (no sinks)
  if sum(Zfilled(:))==previoustotal
    convergence=1;
    disp(['Sinks filled in ' num2str(step) ' steps'])
    disp([num2str(sum(Zfilled(:))-sum(Z(:))) ' total meters added'])
    disp([num2str((sum(Zfilled(:))-sum(Z(:)))/numel(Z)) ' average meters added'])
  end
end
Z=Zfilled; % replace our map!
```

Flow accumulation

The next "map" that is usually calculated on a DEM is the flow-accumulation.

The flow-accumulation parameter records how many cells are upstream a specific cell. Cells with a a high flow-accumulation correspond to places collecting a lot of water: streams or rivers.

Imagine that there is a short and homogeneous rain on our DEM. Only one *drop* falls on each of our cells. We can programmatically follow the path of each of *drop* from the original cell to a **sink** using the flow-direction map. If every time we move one cell downstream we add up all the *drops* we will be calculating the flow-accumulation on our map.

The code in the next slide calculates the flow-accumulation by following all the water paths.

```
%% FLow-accumulation
FA=zeros(size(Z)); % start with a dry map
kernelref=[6,5,4;7,0,3;8,1,2]; % direcction references
drref=[-1,-1,-1;0,0,0;1,1,1]; % row direction reference
dcref=[-1,0,1;-1,0,1;-1,0,1]; % column direction reference
for r=1:nrows
  for c=1:ncols
    convergence=0;
    prevr=r; prevc=c; % define previuos cell
    FA(prevr,prevc)=FA(prevr,prevc)+1; % leave a drop here
    % Start following the river downstream
    while convergence==0
      if FD(prevr,prevc)==0 % if sink
        convergence=1;
      else
        % select the next celldownhill: new row and column
```

```
        newr=prevr+drref(kernelref==FD(prevr,prevc));
        newc=prevc+dcref(kernelref==FD(prevr,prevc));
        % accumulate flow in the next cell
        FA(newr,newc)=FA(newr,newc)+1;
        prevr=newr;
        prevc=newc;
      end
    end
  end
end
```

Now you can plot the "rivers" as the cells with more *drops* than the average:

```
figure; hold on
sel=(FA>mean(FA(:))); % select cells with many drops
plot(X(sel),Y(sel),'.b') % plot the rivers
% overlap the controur plot
contour(X,Y,Z,[0.1,200:200:2000],'-k')
```

Catchment related calculations

Once we have calculated the flow-direction and flow-accumulation matrices (`FD` and `FA`), we can start calculating parameters that can be useful in Earth sciences.

There are many geological processes (e.g. erosion) that depend on what is happening upstream a certain point.

A useful parameter that we might want to know is the average altitude of the upstream catchment at any point in our DEM.

To calculate the average upstream altitude, we can recycle the structure we used for the flow-accumulation calculation and calculate the "accumulated altitude" at every cell of the map. To calculate the average upstream altitude we will just need to divide the accumulated altitude by the flow accumulation.

```
%% Catchment altitude
AccumZ=Z; % start with the same DEM
kernelref=[6,5,4;7,0,3;8,1,2]; % direcction references
drref=[-1,-1,-1;0,0,0;1,1,1]; % row direction reference
dcref=[-1,0,1;-1,0,1;-1,0,1]; % column direction reference
for r=1:nrows
  for c=1:ncols
    convergence=0;
    prevr=r; prevc=c; % define original cell
    % Start following the river downstream
      while convergence==0
```

```
      if FD(prevr,prevc)==0 % if sink
        convergence=1;
      else
        % select the next celldownhill: new row and column
        newr=prevr+drref(kernelref==FD(prevr,prevc));
        newc=prevc+dcref(kernelref==FD(prevr,prevc));
        % accumulate Z in the next cell
        AccumZ(newr,newc)=AccumZ(newr,newc)+Z(r,c);
        prevr=newr;
        prevc=newc;
      end
    end
  end
end
AverageZ=AccumZ./FA;
```

5.6 Exercise: model glaciations

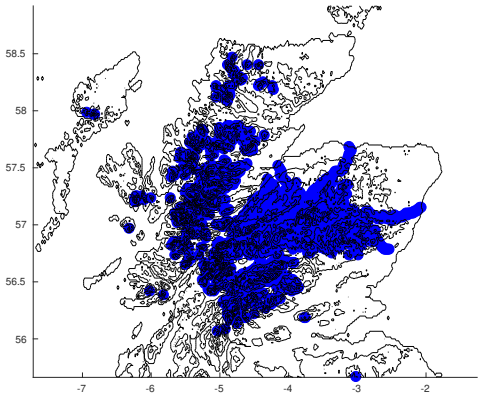
Scottish glaciers (again)

Using the climate and mass balances defined in the first chapter (slide 1.12 and onwards), and assuming average temperatures 4°C below current ones and monthly precipitation 100 mm above current ones, model the Scottish glaciers during Younger Dryas.

Consider that a glacier will be present in places with upstream mass balances above 0.

Plot the results.

The output map should look like this:



Does the output look similar to what we know from geology?

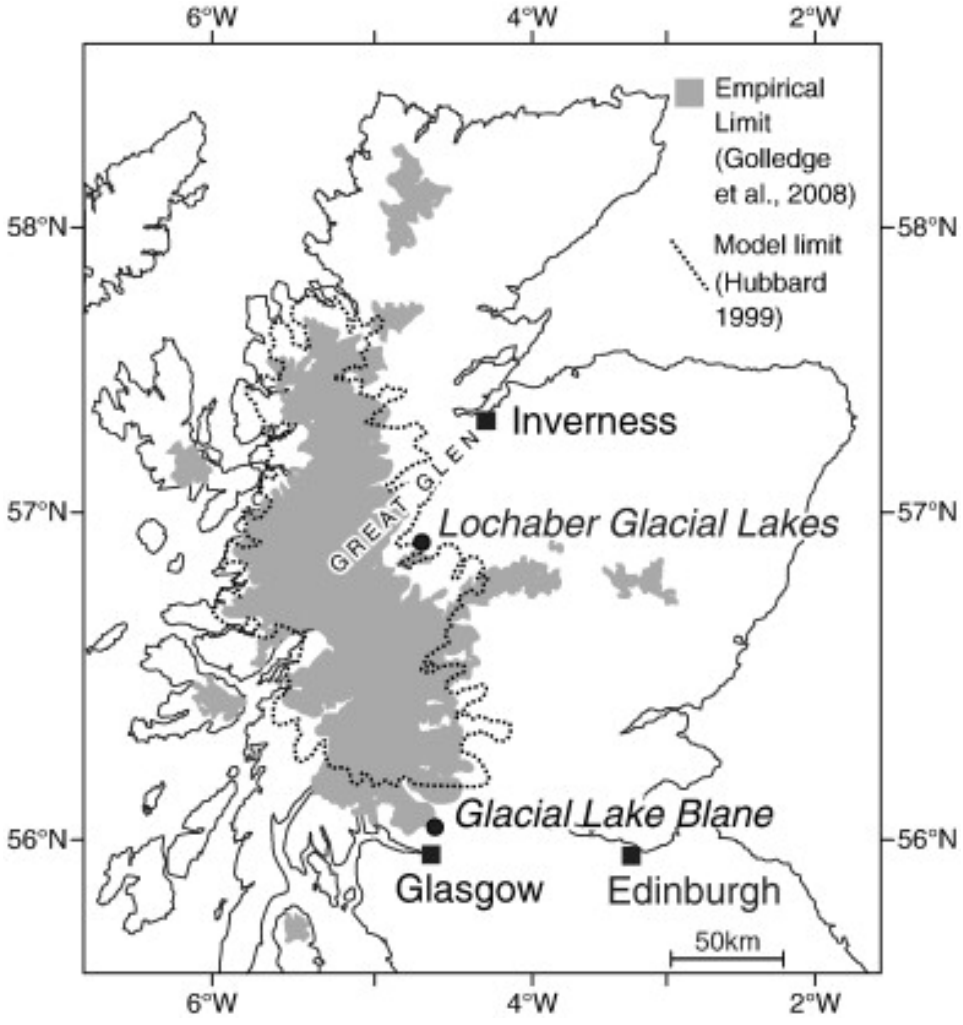


Figure 5.3: MacLeod *et al.* (2011).

Why?