Step 0: Installing Python

Complete the following steps on the computer that you will use at the school **before** you travel to the school.

The tutorials are based on the programming language Python which is growing increasingly popular within the scientific community.

0.1 I already have Python installed

If you already have Python installed on the computer that you are going to bring to the school, you still need to check whether the Python packages that the tutorials depend upon are also installed. These packages are iPython, NumPy, SciPy, and matplotlib. Using whatever method recommended for your version of Python, install these packages. Then make sure that everything works by proceeding to Step 0.3.

0.2 I DO NOT have Python installed

Python is expandable through packages, and other people have spend a significant amount of time building packages that help in a scientific context; the packages upon which this tutorial is based are called iPython, NumPy, SciPy, and matplotlib.

Depending on your operating system, you should do the following to install Python and the necessary packages:

- on Windows, download WinPython which already includes all the packages necessary. On the website, click the Downloads link of the latest version of the 3.4 branch. You will be presented with several options: no special name, Qt5, and Zero. Choose the file with no special ending. Make sure that the number of bits (32bit or 64bit) matches that of your operating system - nowadays it is probably safe to assume that all operating systems are working with 64bit. Download the installer and launch it. Use the default installation path.
- on Linux follow the instructions here; essentially, use your distributions package manager to install the necessary packages.
- on Mac, download and run the 64bit Anaconda graphical installer for Python 2.7 which already includes all the packages necessary.

0.3 Making sure it works

The following steps will make sure that all the different parts of Python are correctly installed on your system.

1. Start iPython
   a) Under Mac and Linux, open a shell (terminal), enter `ipython`, and hit Enter.
   b) Under Windows, double click the file `IPython Qt Console.exe` in the WinPython installation directory.
2. Type `%pwd` and hit enter. The directory in which you started iPython will appear (see Fig. 1).

Figure 1: iPython shell.

3. In that directory - the working directory - use your favorite text editor (on Windows Notepad++, on Linux `gedit`, `nedit`, `vim`, `kate`) to create a file called `helloworld.py` with the following content:

```
# The standard hello world program in Python
print('Hello World!')
```

4. In the iPython shell, type `%run helloworld.py` and hit enter. The words `Hello World!` should appear (see Fig. 1). If something like `ERROR: File 'u'helloworld.py'' not found.` appears, make sure the directory in which you saved `helloworld.py` matches the output of `%pwd`.

5. In the working directory create a file called `testinstall.py` with the following content:

```
# Test the installation of NumPy, SciPy and matplotlib
# Import the necessary libraries
import numpy as np
import scipy.fftpack as fp
import matplotlib.pyplot as plt

# Make an array with numbers ranging from 1...100
data = np.arange( 100 )

# Calculate the fast Fourier transform of that data
spectrum = fp.fft( data )

# Create a plotting canvas
plt.figure(1)

# divide the canvas into to subplots
# Select the upper panel
plt.subplot(211)

```

```
20 # Plot the original data
21 plt.plot ( data )
22 plt.title('Data')
23
24 # Select the lower panel
25 plt.subplot(212)
26
27 # Plot the FFT power (on a semi-logarithmic axis)
28 plt.semilogy( abs( spectrum )**2 )
29 plt.title('Spectrum')
30
31 # Make the plot visible
32 plt.show()
```

6. In the iPython shell, enter `%run testinstall.py`. A new window should open, displaying two plots looking something like Fig. 2 (depending on your operating system):

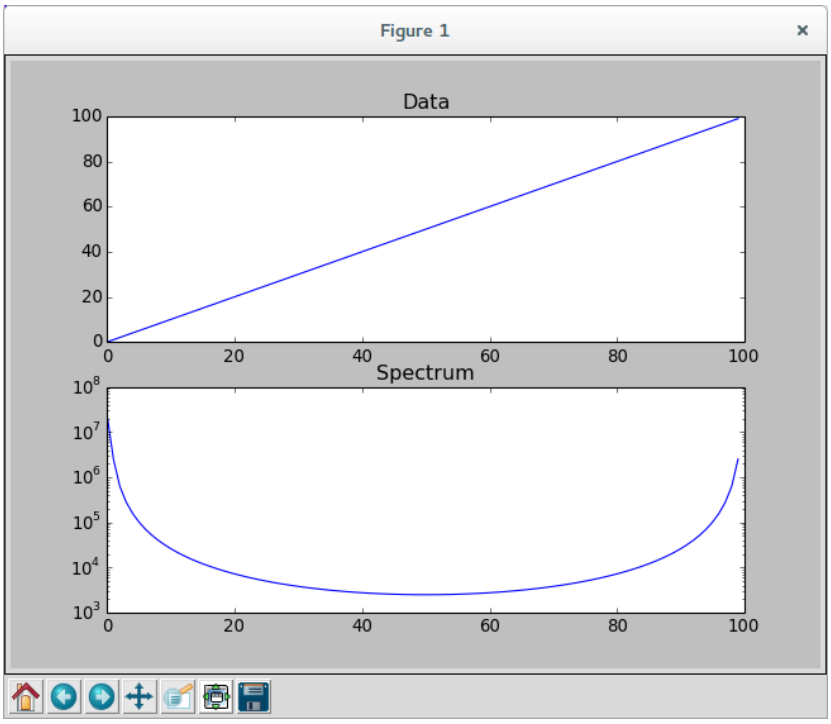

Figure 2: Output of `testinstall.py`.

7. You can exit the iPython shell by entering `quit()` (see Fig. 1).

If you see the words `Hello World!` appear on your screen, and the plotting window opens as expected, you are ready to start the tutorial session.

Step 1: Basic programming terminology

Programming is the act of telling a computer what to do. *Actually* telling a computer what to do is very difficult, because the commands a computer understands are very basic. Luckily, nowadays there are a plethora of high level languages around which make programming relatively easy. In what follows we are going over some of the basic concepts of programming; these are not exclusive to Python, the programming language that we are going to use in this course, but instead will have an equivalent in (almost) every other language you might encounter.

As with natural languages, the best way to learn a programming language is to use it. So in order to get a feel for programming in general and Python in particular, make sure you do the exercises.

Variables

One of the fundamental building blocks of programming are variables. Essentially, they are names/value pairs. Use them to save values for later. Variables can be of different data types: most importantly numbers, boolean (only two possible values, true or false) and strings (or words, combination of characters). Python is not too picky when it comes to changing the data type of variables or combining variables of different data type; but beware, other languages are.

```
# define a floating point number
number = 5.34
# define a string
name = 'Bob'
# change name to type integer
# such a statement can cause trouble in other languages
name = 7
```

Lists

Once you have a variable, the step to "a collection of variables" is not far. Lists are exactly that; they combine under one name many values. In Python, list elements can be of any data type:

```
# define an empty list
mylist = []
# add some stuff, can be any data type or variables
mylist = [ 'Alice', 34.234, name, number ]
# you can check whether a list contains a certain
# elements by using the in construct
bool = 'Hurz' in mylist
# the variable bool will contain False because there is no element 'Hurz' in
    mylist
print( bool )
# use the append function to add an element at the end of the list
mylist.append( 'Hurz' )
# show the elements of mylist
print( mylist )
# check again
bool = 'Hurz' in mylist
# now bool is True
```

```
print( bool )
# of course, a list can be an element within a list
newlist = [ 1, 2, 3, 4 ]
mylist.append( newlist )
```

You can access individual items within the list by giving its position - the first item has position 0. So to access the third item in `mylist`, use

```
# print the third entry
print( mylist[2] )
# overwrite the third entry
mylist[2] = 'palim palim'
```

You can also access several elements in a single command by separating the first and last element index MINUS ONE by a : (colon). This is called slicing in Python terminology.

```
mylist = [ 1, 2, 3, 4, 5, 6 ]
# to get the second and third element, write
print( mylist[1:3] )
# remember, list[start:end] selects all values
# between index start and end-1
# you can also access values from the back.
# this prints the last element
print( mylist[-1] )
# or even from the second to the second last
print( mylist[1:-2] )
```

A useful mnemonic of remembering which elements you obtain when slicing, is the following picture.

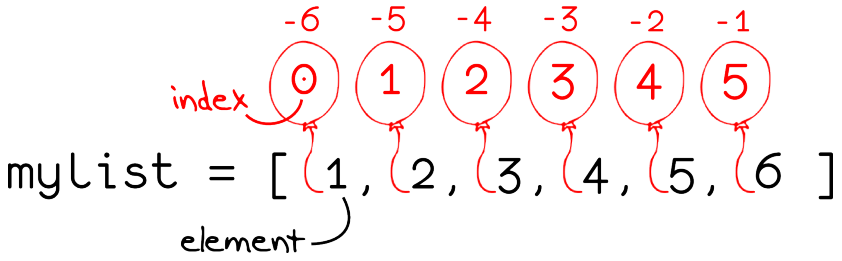

Figure 3: Remembering indices for slicing.

The numbers in the red balloons are the indices of the elements, and they are attached to the lower left corner of each element[1]. So if you do

```
mylist = [ 1, 2, 3, 4, 5, 6 ]
# to get the second and third element, write
print( mylist[1:3] )
# or, in fact
print( mylist[-5:-3] )
```

you get everything between index 1 and 3, which are just the numbers 1 and 2. The reason I am stressing this is because every other programming language I know, given a statement like `list[start:end]`, returns all elements between `start` and `end`, *including* the one at `end`; for those languages, then, the balloon would be attached to the center of the element. Also note that Python does not complain about indices out of bounds but instead just returns an empty list:

[1]Thanks, Alois Mahdal, for that image.

```
mylist = [ 1, 2, 3, 4, 5, 6 ]
# clearly, there are no elements at these indices
# so Python simply returns an empty list
a = mylist[-2372:-423]
```

Another list related function to remember is `len(mylist)`, which returns the number of elements in a list.

Variables and Lists

In some programming languages, the left hand side of a variable assignment must be a single variable name, i.e.:

```
a = 12
```

In the case of 1-dimensional variables, such a construct is obvious. Consider, however, the following:

```
a = ['My mother', 68]
```

Python allows you another way to assign variables, like so:

```
name, age = ['My mother', 68]
```

After this assignment, the variable `name` contains the string `My mother`, and the variable `age` contains the integer `68`. Such behavior is especially useful when dealing with return values from functions - more on that later.

Dictionaries

In Python, dictionaries are an unordered collection of key/value pairs - other programming languages call such a construct a hash (Perl) or hashtable (Java); dictionaries are also somewhat similar to structures in C. The keys are typically strings but the value can be of any data type. Dictionaries are very helpful when storing data. Imagine you have two arrays, one holding voltage measurements `vs` and one filled with the associated time stamps `tm`; you could combine these data in a dictionary like this:

```
# define a time series of "measurements"
# range is a built-in function that returns a list of
# numbers from 0 to the number you provide as the argument
# in this case, then, tm is a list containing all integer
# values between 0 and 999
tm = range( 1000 )
vs = tm
# define an empty dictionary
data = {}
# add the data, giving them names - the keys
data['voltage'] = vs
data['time'] = tm
# accessing the data uses the same syntax
print( data['time'] )
```

The advantage of dictionaries (vs. plain variables) is that the connection key/value is permanent, i.e., once created you cannot change the key `'voltage'` to something else and hence you preserve the connection between the description and the data.

Conditional statements

Often you want your program to execute different statements depending on some condition. For that, use `if` statements (note the indentation!):

```
number = 5
if number == 5:
  print('The number is 5!')
else:
  print('The number is NOT 5!')
```

The comparison operators Python knows are `==`, `!=`, `<`, `<=`, `>`, and `>=` with the expected meaning. But as you saw in the secion about lists, there are other expressions that result in a boolean value that can be used in `if` statements. You can combine conditions using `and` and `or`:

```
# Python is case sensitive, i.e., numberA and numbera are
# two different variable names
numberA = 5
numberB = 7
if numberA == 5 and numberB != 7:
  print('The number A is 5, and the number B is not 7!')
else:
  print('The number A is not 5 or the number B is 7!')
```

Or something more advanced:

```
mylist = range( 100 )
if 8 in mylist:
  print( 'Indeed, the list contains an 8.' )
else:
  print( 'But now it is gone.' )
# remove an object from a list
mylist.remove(8)
if 8 in mylist:
  print( 'Indeed, the list contains an 8.' )
else:
  print( 'But now it is gone.' )
```

Indentation

In the case of `if` statements, you might want to execute several statements depending on one condition. In Python, statements are grouped into blocks by indentation; other languages mark blocks with `{` and `}` or keywords like `BEGIN` and `END`. Consider the following:

```
number = 5
if number == 5:
  number = number + 5
  number = number*3
print(number)
```

and compare its outcome to the outcome of

```
number = 5
if number == 5:
  number = number + 5
number = number*3
print(number)
```

In the second case the statement

```
number = number*3
```

is executed regardless of the outcome of the `if` condition whereas in the first it is not.

Loops

Once you have data saved in lists (or dictionaries), you might want to manipulate them; for such a task, use loops. Essentially, they will help you repeat a specific task for a certain number of times.

The most basic loop is the `while` loop which executes the statements within the loop as long as the condition after the `while` keyword evaluates as true:

```
x = 0
while x < 10:
  print(x)
  x = x + 1
```

With regard to indentation, we note that

```
x = 0
while x < 10:
  print(x)
x = x + 1
```

would never terminate, because x is never incremented within the `while` loop and hence `x` is always `0`, and the condition `x < 10` will always be true. (If you actually try this example, hit Ctrl+c to stop the execution of the loop.)

Another type of loop is the `for` loop:

```
# the range( n1, n2 ) function creates a list with values from n1 to n2-1
mylist = range( 0, 10 )
for x in mylist:
  print(x)
```

All loops can be exited prematurely using the command break:

```
for x in range( 0, 10 ):
  if x == 5:
    break
  print(x)
```

Functions

Often when using data you find that you are doing certain tasks over and over again. For this situation, Python (and all other programming languages) provide the possibility to combine functionality within a function:

```
def functionname( argument1, argument2, ... ):
  statement
  statement
  ...
```

For example, you might need to count the number of lines in a file which can be achieved using this function:

```
def filelines( filename ):

  count = 0
  f = open ( filename, 'r' )
  for line in f:
    count+=1
  f.close()

  return count
```

The only argument is a string containing the file's name. The function initiates a counter, opens the file, increases the counter for each line encountered, closes the file, and returns the value of the counter. If your function does something but there is no need to return anything (like the `close` function) just omit the `return` statement.

Modules

Python provides the possibility to combine a set of functions in one file and make it available for future use. For example, instead of including the `filelines` function into every script where you need it, you can put it in a file named `filehelpers.py` and place it in your working directory. You can then use the functions of this module (currently only one) by importing the module

```
import filehelpers
```

and then using it with

```
cnt = filehelpers.filelines('lala.dat')
```

A combination of modules is called a "package".

Arrays

The difference between lists and arrays is subtle. Basically, lists can be easily expanded and shrunk depending on the context; use them if you are unsure how many elements you will need for your programming task. If, however, you know the amount of values you are going to encounter, using arrays to save them is much more efficient. In Python, arrays are provided by the NumPy package:

```
import numpy
# this creates an array with 23 elements, all set to 0
arr = numpy.zeros( 23 )
```

Specific elements of the array are accessed using square brackets:

```
# set the 13th elements (indexing starts at 0!) to 23./134.
arr[12] = 23./134.
```

Just like slicing lists, ranges of elements are accessed using the colon operator:

```
# print the seventh to 13th element of arr
print( arr[6:13] )
```

You can also select elements from an array using a list (or an array) of indices:

```
import numpy as np
# define array
mylist = np.array( [ 1, 2, 3, 4, 5, 6 ] )
```

```
# define list of indices
idx = [ 2, 4, 2, 2, 2, 1, 4 ]
print( mylist[ idx ] )
```

However, the same would *not* work when trying to extract elements from a list:

```
# define array
mylist = [ 1, 2, 3, 4, 5, 6 ]
# define list of indices
idx = [ 2, 4, 2, 2, 2, 1, 4 ]
# this will create an error
print( mylist[ idx ] )
```

returns an error.

Note that Python makes array handling easy, for example

```
# fill elements 7 to 14 with the numbers 4 to 12
arr[6:13] = range(4, 13)
# multiply each elements with 4
brr = arr*4
# square arr and subtract brr
crr = arr**2 - brr
```

produces the expected result.

While the above is true, consider the following:

```
# create a list
mylist = range( 5 )
print( mylist*3 )
# create an array
myarray = np.arange( 5 )
print( myarray*3 )
```

Here you see that arrays and lists are quite difference, in that the `*` operator for arrays performs the mathematical operation per element while for lists it repeats the list as many times.

Useful helpers

There are a few elements of the python shell that make life easier. For example, if you have lost track of the type of a variable but remember its name, type `?variablename` to get information about the variable on the shell. For example:

```
In [1]: number = 5.34
In [2]: ?number
Type:       float
String Form:5.34
Docstring:
float(x) -> floating point number

Convert a string or number to a floating point number, if possible.
```

While that might not seem helpful for single valued variables, it is very helpful for quickly finding the size of arrays or dictionaries.

Another useful thing to remember is the `keys()` function of dictionaries:

```
In [17]:  newdict = {'somekey':0, 'camel':0, 'horst':0}

In [18]: newdict.keys()
Out[18]: ['somekey', 'horst', 'camel']
```

because you will forget what you named the keys in your dictionary.

Documentation

In the following tutorials we are going to use functions from the Python standard library and additional packages. If you are unsure about the way a certain function is used or what it does specifically, just google `python functionname` and you will quickly find out.

Exercise 1

1. Write a Python program that fills an array of size 12 with values ranging from 0 to 33 in steps of 3, i.e., the i^{th} element should contain the value `i*3`.
2. Change your solution into a function named `third` that excepts one argument: a number that is used as the size of the resulting array.
3. Use the function from 2. to create an array of size 1001 and set every seventh element to -103.
4. Calculate the sum of all array elements from 3.
5. For the array from 3., calculate the sum of elements 65 through 173.
6. Again calculate the sum of all array elements from 3., but now count those elements twice whose index is divisible by 13 and 23. (Hint: Look at the modulo operator `%`).
7. Loop through all elements from 3., and calculate the sine of every 3^{rd} element (in degree!). If the result is positive, add 42.42 to the original value. If the sine is between -0.2 and +0.2, however, replace the value with the logarithm (base 10) of the next value in the array. (Hint: Check out Python's math library)

Step 2: Reading data

Before you start this tutorial, open iPython, change to your working directory (`cd dirname1/dirname2/`), and call the magic `%matplotlib`.

The objective of this step is to read real-life data from a file using Python. The data originates from a webservice called OMNI that provides solar wind parameters; here we will look at the interplanetary magnetic field (IMF). The data is available in the form of a tabulated text file and can be obtained here: 20120305-20120315_imf.txt Originally, the data was downloaded from this website: OMNIWeb.

We will read the data into an array, which is a data structure provided by the package NumPy. Before we can do that, we need to know how many data points we need to read in. For that, we will use a function which simply tells us how many rows (lines) of data the file includes. We can count the number of lines in a file with this function:

```
def filelines( filename ):

  count = 0
  f = open ( filename, 'r' )
  for line in f:
    count+=1
  f.close()

  return count
```

`filelines.py` opens the file given as an argument and loops through the lines, increasing the counter `count`. There are, of course, other ways to count the number of lines in a file using Python. Save this file in your working directory.

In the program reading the data, we will use the function `filelines`, so we need to make the program aware of its existence with the line

```
  import filelines
```

Because `filelines` is the module name, to use the function we need to write

```
  nlines = filelines.filelines( filename )
```

The following function will read in the data first into the arrays `imfbx`, `imfby`, `imfbz`, and `imfbt`. The dates are read into a list called `date`. After the reading is complete, the data will be returned as a dictionary.

```
import numpy as np
import datetime as dt
import filelines

def read_imf( ):

  # Name of the file
  filename = '../data/20120305-20120315_imf.txt'

  # We know that the file contains a header, i.e.,
  # information at the top of the file about the
  # data. Ignore that information.
  nheader = 13

  # There are also some lines of footer, i.e.,
  # information at the end of the file
```

```
  nfooter = 12

  # Count number of lines
  nlines = filelines.filelines( filename )
  ndata = nlines - nheader - nfooter

  # Initialize variables
  date = []
  imfbx = np.zeros( ndata )
  imfby = np.zeros( ndata )
  imfbz = np.zeros( ndata )
  imfbt = np.zeros( ndata )

  # Open file for reading
  f = open( filename, 'r' )

  # Read and ignore header lines
  for i in range( 0, nheader ):
    dummy = f.readline()

  # Loop over lines and extract variables of interest
  for i in range( 0, ndata ):
    # read one line of data as a string
    line = f.readline()
    # create a datetime from the first 14 characters
    date.append( dt.datetime.strptime( line[0:14], '%Y %j %H %M' ) )
    # split the rest of the line at whitespaces
    columns = line[14:].split()
    # Convert strings to floats and save them in the variables
    imfbt[i] = float( columns[0] )
    imfbx[i] = float( columns[1] )
    imfby[i] = float( columns[2] )
    imfbz[i] = float( columns[3] )

  # Close the file
  f.close()

  # Convert the Python list into a NumPy array
  date = np.array( date )

  return { 'date': date, 'imfbx': imfbx, 'imfby': imfby, 'imfbz': imfbz, 'imfbt':
     imfbt }
```

After opening the file and ignoring the header, we read the data line by line. Each line contains two different types of data: the first part is the date and time of each measurement, the second part is the IMF measurement itself. In order to convert the first part of each line into a Python `datetime` object, we can use the `strptime` function. This function takes a string containing the date as first input and a format string as second input. The format string contains information of how the date information (year, month, day, hour minute, seconds, etc.) is arranged in the string (see `strptime`). In the case of the OMNI data, each date/time string contains the year, followed by a space, the day-of-year (day number after January 1st), a space, the hour, a space, and the minute. As a format string, this translates to `'%Y %j %H %M'`.

Once the datetime is extracted, we can splitting the remainder of each line into columns using the standard Python function `split()`. Once all the data is read in, the list of datetimes

is converted to an array and everything is returned in a dictionary.

Step 3: Plotting data

Just as with `filelines` we can use the function `read_imf` after importing the module. We can then use standard `matplotlib` functionality to create our first plot:

```
import read_imf
import matplotlib.pyplot as plt

# read the data
data = read_imf.read_imf(  )

# create plotting canvas
fig = plt.figure(1)

#create plot
plt.plot(data['date'], data['imfbx'])

#show plot
plt.show()
```

Although the data displays, it looks horrible. The reason is that there are data gaps in the OMNI data, i.e., times when no IMF data was available, and that in that case the value `9999.99` is written into the file. We can rectify this by substituting each IMF value that is `9999.99` with a special value called `Not a Number`, or `NaN` for short. In `read_imf.py`, after line 49 include

```
# However, if the data is 9999.99, set the
# value to Not a Number (NaN)
if imfbt[i] == 9999.99:
  imfbt[i] = np.nan
if imfbx[i] == 9999.99:
  imfbx[i] = np.nan
if imfby[i] == 9999.99:
  imfby[i] = np.nan
if imfbz[i] == 9999.99:
  imfbz[i] = np.nan
```

Because `matplotlib` automatically ignores `NaN`s, the plot looks a lot nicer, However, it it still missing key features plus a few minor things:

1. **No plot is complete without axis labels and proper units. Never, ever, ever, ever!**

   ```
   # get the axis
   ax = fig.gca()
   # add labels
   ax.set_xlabel('Time UT')
   ax.set_ylabel('IMF Bx [nT]')
   ```

2. Plot the curve in black

   ```
   plt.plot(data['date'], data['imfbx'], color='black')
   ```

3. Make the x axis start and end at 00:00

   ```
   import datetime as dt
   bt = dt.datetime(2012, 3, 5, 0, 0, 0)
   ft = dt.datetime(2012, 3,15, 0, 0, 0)
   ax.set_xlim(bt, ft)
   ```

4. Make the y axis symmetric

```
ax.set_ylim(-30,30)
```

5. Change the format of the dates on the x axis

```
# the string '%b %d, %H:%M' determines the date formatting
# here we choose short month name, day, hour, minute
ax.xaxis.set_major_formatter(mdates.DateFormatter('%b %d, %H:%M'))
fig.autofmt_xdate()
```

6. Overlay a gray dashed line at 0 nT

```
xli = ax.get_xlim()
plt.plot( xli, [0,0], '--', color='gray' )
```

Exercise 2

1. Use the `matplotlib` function `subplot` to create a overview of all four IMF components.

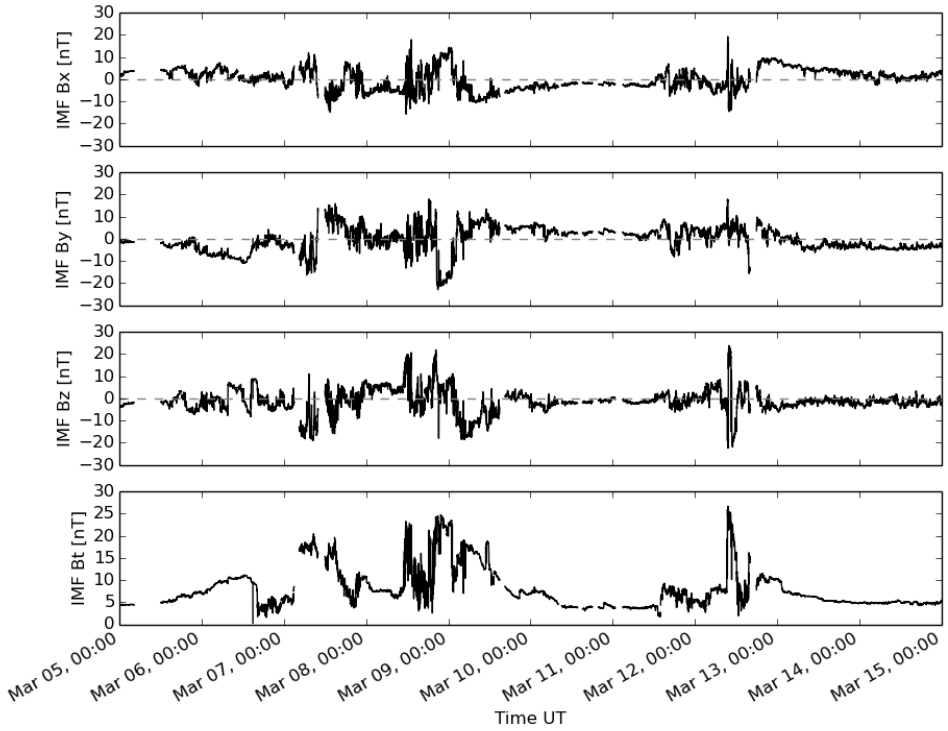

Figure 4: Overview of the IMF between 20120305 and 20120315.

Make sure to plot the right data from the data dictionary.

2. Navigate to OMNIWeb, and download the high-resolution solar wind flow speed and proton density for the same time period (2012-03-05 until 2012-03-15). Based on `read_imf.py` create a script that reads the solar wind plasma parameters (speed and density). In that script, replace missing values with `NaN`s and from the velocity v and proton density n calculate the dynamic pressure

$$p = \frac{1}{2}\rho v^2, \tag{1}$$

where ρ is the mass density, i.e., $\rho = n * m$ where m is the proton mass. Finally, write a Python script that plots all four IMF and all three plasma parameters on one page, like in Fig. 4 (in portrait format!).

3. Navigate to OMNIWeb, and download the high-resolution activity indexes AE and SYM/H. Based on `read_imf.py` create a Python function that will read the indexes into a dictionary. Then create a plot over the same time span as before (2012-03-05 until 2012-03-15).

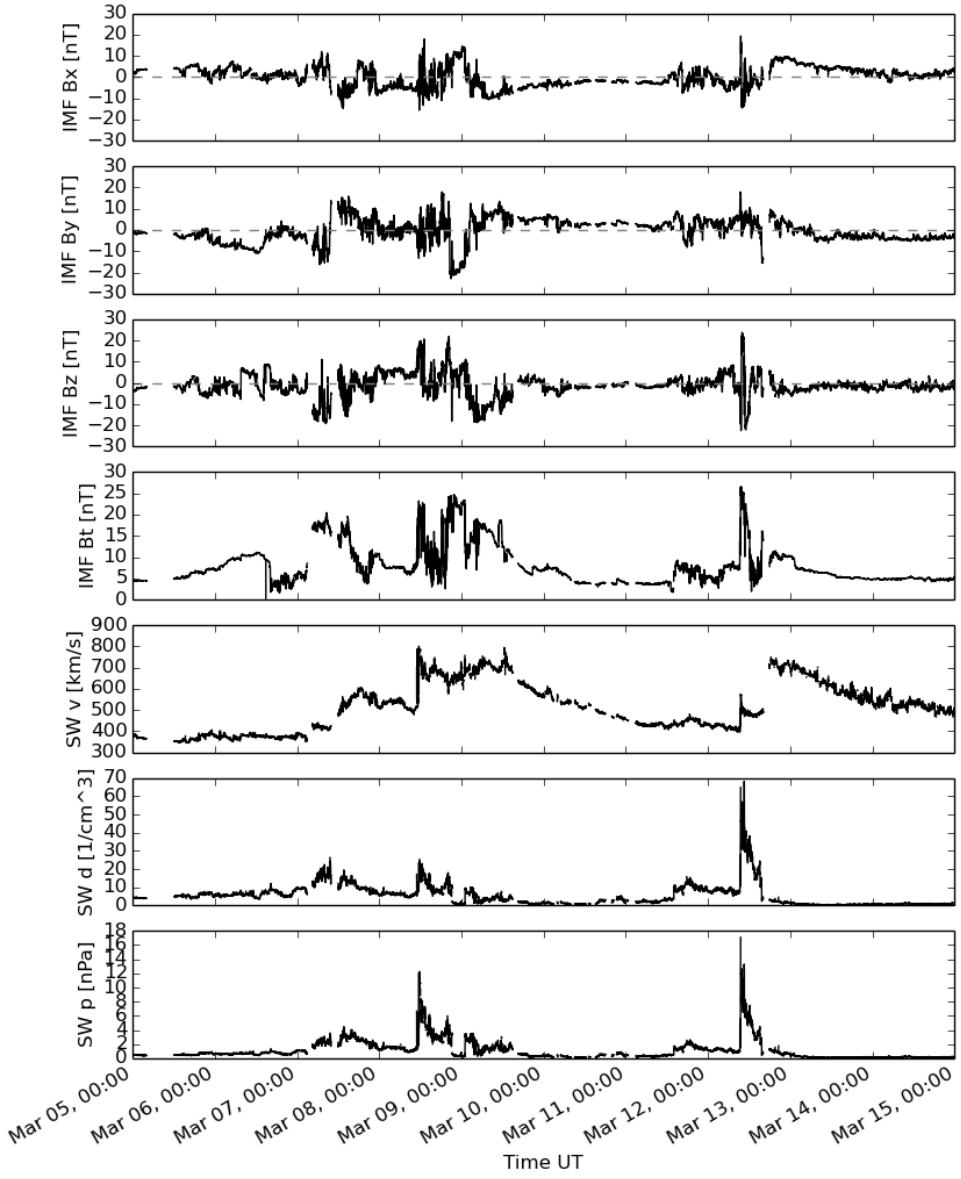

Figure 5: Overview of the IMF and solar wind plasma parameters between 20120305 and 20120315.

Step 3: Selecting data

Now that we know how to read data, we might also be interested to select data depending on certain conditions; for example, we might be interested to compare solar wind values of times when the SYM/H is below -40 nT to those values when SYM/H is above that threshold.

Before we discuss selecting data using the OMNI data, let's look at a few ways to select data from a list depending on some condition. Say that we have a list of names and ages and we want to collect the names of those persons who are over 18. The "classic programming" approach would be using a `for` loop, like so:

```
# define names
names = ['Alice', 'Bob', 'Charlie', 'Donna', 'Evan', 'Frieda']
# define ages
age = [ 12, 43, 23, 6, 28, 3 ]
#create empty list for those persons over 18
over18 = []
for i in range( len( names ) ):
  if ages[i] > 18:
    over18.append( names[i] )

print( over18 )
```

A more pythonic way of doing this is

```
# define names
names = ['Alice', 'Bob', 'Charlie', 'Donna', 'Evan', 'Frieda']
# define ages
age = [ 12, 43, 23, 6, 28, 3 ]

over18 = [ n for (n,a) in zip(names,ages) if a > 18 ] #BOOM!
```

At first, this might look strange. So let's look at the command step-by-step:

- `zip`: A built-in function that combines lists into one big list where the first element of the combined list is a list of the first elements:

  ```
  comb = zip(names, ages)
  print( comb[0] )
  ```

  returns `('Alice', 12)`.

- We then loop over all elements of the combined list, saving the current values of `names` and `ages` in the variables `n` and `a`.

- While looping over the combined list, we select only those `n`'s where `a > 18`.

- Finally, the whole expression is in square brackets to turn all the selected `n`s into a list.

We can use this approach to go back to our original task, selecting those IMF values when the SYM/H index is below -40 nT. We could use:

```
import read_imf_nan
import read_ind_nan

# read the IMF data
imfdata = read_imf_nan.read_imf_nan()
# read the index data
```

```
inddata = read_ind_nan.read_ind_nan()

bx_low = [ bx for (bx,symh) in zip( imfdata['imfbx'], inddata['symh'] )
  if symh < -40 ]
```

Note that this only works if the two lists you are concatenating with `zip` have the same length!

While this achieves what we originally wanted to do, it becomes cumbersome for more complicated conditions; what, for example, if we also wanted to restrict the time interval? Or if we wanted to select more data; we are also interested in the values of `imfdata['imfby']` and `imfdata['imfbz']`. In such cases it is usually best to first create a list of the indices of those elements where the condition is fulfilled. NumPy has a function for that: `extract`.

```
import numpy as np
import read_imf_nan
import read_ind_nan

# read the IMF data
imfdata = read_imf_nan.read_imf_nan()
# read the index data
inddata = read_ind_nan.read_ind_nan()

# create a list containing all array indices
all_idx = range( len( imfdata['imfbx'] ) )
# create a boolean array of the results of the test of the condition
# this only works because inddata['symh'] is an array
# had it been a list, this would not work
condition = inddata['symh'] < -40

# extract from all indices those where condition is true
tidx = np.extract( condition, all_idx )

# extract from the IMF measurements those where condition is true
bx_low = imfdata['imfbx'][tidx]
by_low = imfdata['imfby'][tidx]
bz_low = imfdata['imfbz'][tidx]

# extract from all indices those where condition is NOT true
fidx = np.extract( ~condition, all_idx )

# extract from the IMF measurements those where condition is NOT true
bx_high = imfdata['imfbx'][fidx]
by_high = imfdata['imfby'][fidx]
bz_high = imfdata['imfbz'][fidx]
```

Exercise 3

1. Suppose you use `read_imf.py` to read the IMF data into the dictionary `imfdata`. Use any of the approaches above to extract those IMF Bz measurements that occurred between 2012-03-10 06:00 and 2012-03-11 06:00.

2. Using the `hist` function, plot a histogram of the IMF Bz values that occurred between 2012-03-10 06:00 and 2012-03-11 06:00.

3. Compare the histogram of 2. with the distribution of IMF Bz values between 2012-03-08 12:00 and 2012-03-09 12:00. What does the difference mean in a geophysical context?

4. Download from OMNIWeb the four IMF components, the two plasma parameters (flow speed and proton density), and the two indexes (AE and SYM/H) for the entire year 2012 into one file. Use your results from `read_imf.py` to write a Python function that reads the data into a single dictionary.

5. Download the following list of magnetic storms from 2012. Extract from the year of data those SYM/H measurements occurring between ±2 days of the magnetic storm and plot all SYM/H measurements on a single x axis ranging from -2 to 2.

Interlude 1: Interpolation

As you have seen when plotting the IMF data between 20120305 and 20120315, there are data gaps. For some analyses (like FFT), however, the data must be gapless.

In order to close these gaps, data must be created where no data was before; there are many strategies to do that but the important thing to remember is that when you fill data gaps, you are always inventing data. It is then up to you to investigate how much your analysis is affected by such data invention.

The most obvious strategy to fill data gaps is linear interpolation. Consider the following:

```
import numpy as np
import scipy.fftpack as fp
import math
import matplotlib.pyplot as plt

# number of points in the test time series
nn = 1000.
# length of data gap, relative to nn
gaplength = 0.01

# create original time series
rg = np.arange( nn )
# x values, for 0...1
xs = rg/(nn-1.)
# oscilating y values
ys = np.sin( 2.*math.pi*6*xs )

# new y values, containing NaNs in a gap of length 0.1*nn
ys_nan = [ np.nan if r > 0.3*nn and r < (0.3+gaplength)*nn else y for (r,y) in
    zip(rg,ys) ]

# extract only finite values
xsi = np.extract( np.isfinite( ys_nan ), xs )
ysi = np.extract( np.isfinite( ys_nan ), ys )

# interpolate the y values ysi from xsi to xs
ysi = np.interp( xs, xsi, ysi )

# create figure
fig = plt.figure(1, figsize=( 16, 9 ) )

# create four subplots
```

```
ax1 = plt.subplot( 2, 2, 1 )

# plot original time series
ax1.plot( xs, ys, color='black')
ax1.set_xlim( 0, 1 )
ax1.set_ylim( -1.2, 1.2 )
ax1.set_title('Original')

# next subplot
ax2 = plt.subplot( 2, 2, 2 )

# plot time series containing NaNs
ax2.plot( xs, ys_nan, color='black')
ax2.set_xlim( 0, 1 )
ax2.set_ylim( -1.2, 1.2 )
ax2.set_title('Original + NaNs')

# next subplot
ax3 = plt.subplot( 2, 2, 3 )

# interpolated version
ax3.plot( xs, ysi, color='red')
ax3.set_xlim( 0, 1 )
ax3.set_ylim( -1.2, 1.2 )
ax3.set_title('Interpolated')

# next subplot
ax4 = plt.subplot( 2, 2, 4 )

# plot the FFTs of the original and interpolated time series
ax4.semilogy( np.arange(1000), abs( fp.fft( ys ) )**2, color='black' )
ax4.semilogy( np.arange(1000), abs( fp.fft( ysi ) )**2, color='red' )
ax4.set_xlim( 0, 100 )
ax4.set_ylim( 1e-3, 1e5 )
ax4.set_title('FFT')

plt.show()
```

Interpolation is also very useful when your analysis involves two time series which were measured at different time intervals. Consider, for example, the earlier discussion of selecting IMF values based on the value of SYM/H. The solution was as straight forward, because both parameters were measured at the same time. But what, if that is not the case. There exists a similar index to SYM/H, called Dst, which is only available every 3 hours. How do you select your IMF values in such a case?

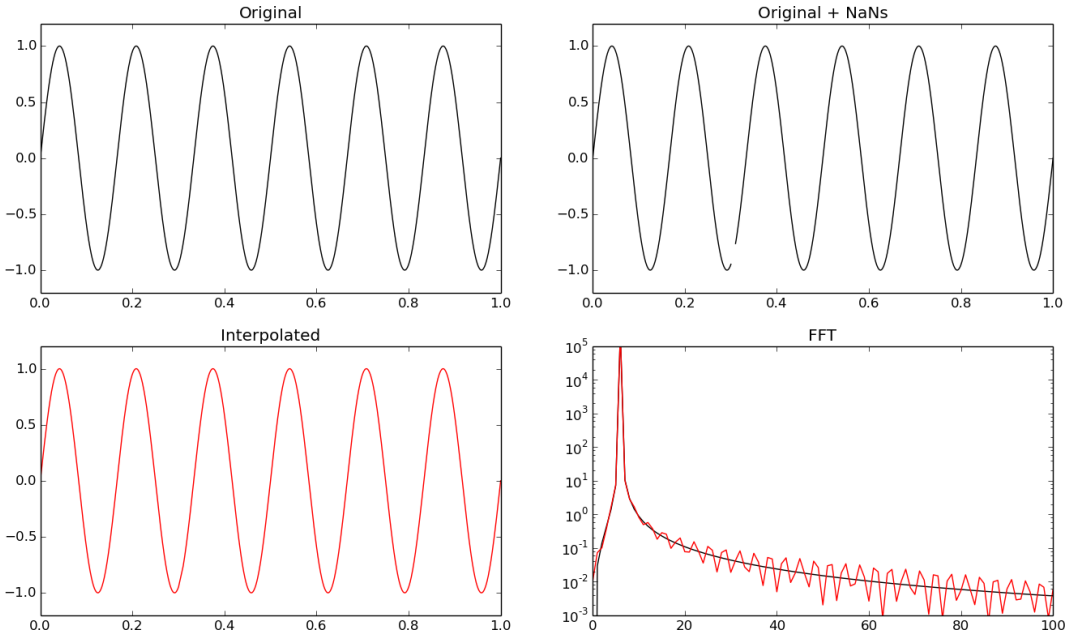

Figure 6: The effect of interpolated data on FFTs.

2-dimensional data

So far, all the data we have encountered was 1-dimensional; one parameter dependent on time. Next, we are going to take a closer look at 2-dimensional data.

Interlude 2: Color mapping

An image, no matter what instruments produced it, is a 2-dimensional array of values. If you take a picture with your mobile phone, the sensor behind the lens has a certain number of pixels in two orthogonal directions, defining an area which is sensitive to light; the number of pixels in the two directions then give the dimension of the array.

Alternatively, every 2-dimensional array of data can be treated (and displayed) an as image. For example, if you have measurements of altitude above sea level measured in regular intervals over a rectangular area, you can display that data as an image.

The essential ingredient to displaying a 2-dimensional array is colormapping, i.e., the process of assigning to each value within the array a color.

Consider the following 2D array

```
1 import numpy as np
2 # use Numpy's array function to convert a list to an array
3 my2Darr = np.array( [ [ 15, 4, 23 ], [ 8, 42, 16 ] ] )
```

and we want to display values between 0 and 10 as magenta, 10 to 20 as blue, 20 to 30 as cyan, 30 to 40 as green, 40 to 50 as yellow, and 50 to 60 as red. That is, we want to use the following colormap

Using this mapping, we would expect to see the 15 to be colored blue; Figure 8 shows the resulting image.

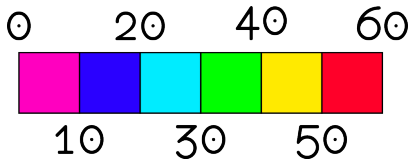

Figure 7: An example colormap.

```
my2Darr = [[15,4,23],[8,42,16]]
```

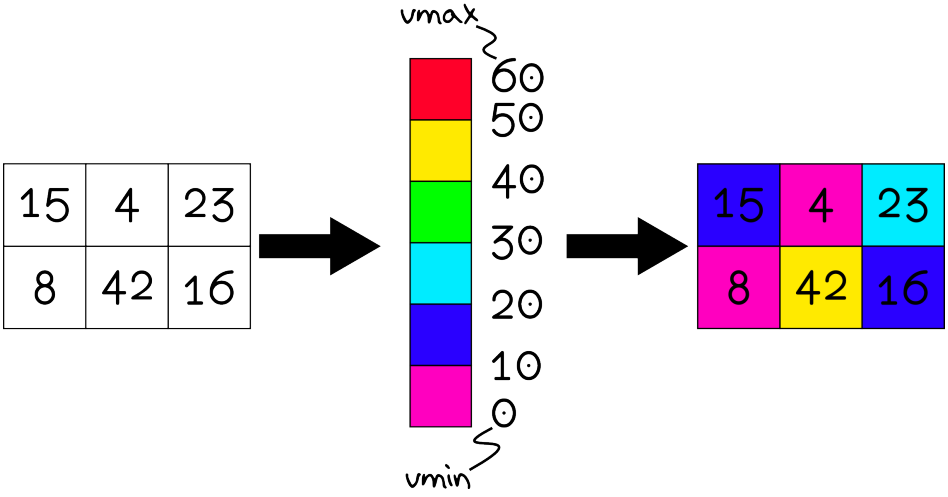

Figure 8: Example of colormapping.

Typically, a colormap is linear and defined by three values, `vmin`, `vmax`, and `N`: `vmin` gives the value which is mapped to the lowest color; `vmax` gives the value which is mapped to the highest color; and `N` is the number of steps between `vmin`, `vmax`. In the above case `vmin=0`, `vmax=60`, and `N=6`.

Using Python, this is how you display a 2D data array as an image:

```
import numpy as np
import matplotlib.pyplot as plt

# define array
my2Darray = np.array( [[15,4,23],[8,42,16]] )
# open display window
fig = plt.figure(1)
# imshow is the function to display 2D data (images)
cax = fig.gca().imshow( my2Darray )
plt.show()
```

If you run this piece of code, however, the result looks nothing like you would expect. Partly, that is expected because you have not told Python what colormapping to use so it decided on some default which could be anything. But more worryingly, the 6-element 2D array looks washed out. The reason for that is that in order to display the 6-pixel image with a reasonable size on your screen, `imshow` uses interpolation to scale the original image up. By default, this interpolation is linear and this results in the washed out version. We can change that to `nearest` to get the expected result:

```
cax = fig.gca().imshow( my2Darray, interpolation='nearest' )
```

Now we see the 6 pixels, however, the mapping is not what we wanted. To get the result we are after, we need to use the following piece of code:

```
import numpy as np
import matplotlib.pyplot as plt

# define array
my2Darray = np.array( [[15,4,23],[8,42,16]] )

# choose colors of the colormap and the number of intervals
cm = plt.get_cmap('gist_rainbow_r', 6)

# create figure
fig = plt.figure(1)
# display the image with interpolation off,
# the colopmap and defining vmin and vmax
cax = fig.gca().imshow( my2Darray, interpolation='nearest',
  cmap=cm, vmin=0, vmax=60 )
# plot a colorbar next to the image.
cbar = fig.colorbar( cax, ticks=np.arange(7)*10 )

plt.show()
```

There are a number of colormaps built into matplotlib but you can also define your own mappings. You can find more information in the matplotlib documentation.

Auroral images

The auroral images we are going to look at were taken at Resolute Bay in northern Canada. While some of the issues we are going to discuss are only relevant for this particular imager system, many other things are generic procedures you would use, whether you display data from this imaging system, the Hubble space telescope, are an electron microscope.

The auroral imager at Resolute Bay is typically located within the polar cap. It records the brightness within narrow ranges of wavelengths associated with the aurora - here we are going to use the "red line" at 630.0 nm.

The data is saved as a series of numbers in binary format; you can download the example file here. The first four numbers are of type integer and can be ignored. Then follow 65536 uint values that represent the brightness measured in 256×256 pixels by the camera (if you do not understand what that means, don't worry, the important thing to take away is that we are reading 65536 brightness values from a file). Since we are reading 65536 values from a file sequentially, we need to use Numpy's `reshape` function to rearrange the values into a 256×256 2D array. We choose a colormap called `jet`.

```
import numpy as np
import matplotlib.pyplot as plt

# this is the filename
fname = '../data/C62_131229091438_0030.abs'

# open the file (read, binary)
f = open(fname, "rb")

# read some unimportant header data
```

```
header = np.fromfile( f, dtype=np.int16, count=4 )

# read the brightness and adjust for binning
data = np.fromfile( f, dtype=np.uint16, count=256*256 )

# close the file
f.close()

# data is a 1D array which has 65536 elements
# we need to rearrange this into a 256x256 2D array
image = data.reshape( 256, 256 )

# because of an averaging procedure, we get the brightness by
# mulitplying all values with 0.25
brightness = 0.25*image

# choose colors of the colormap and the number of intervals
cm = plt.get_cmap('gist_heat', 256)

# create figure
fig = plt.figure(1)
ax = fig.gca()
# display the image
cax = ax.imshow( brightness, vmin=1e2, vmax=1e3, cmap=cm )
# plot colorbar
cbar = fig.colorbar( cax )

plt.show()
```