Structured Programming, EE20084

EE20084

Dr. Steve Pennock

2East-2.26, S.R.Pennock@bath.ac.uk

Semester 2, 2023-24

Department of Electronic and Electrical Engineering
University of Bath

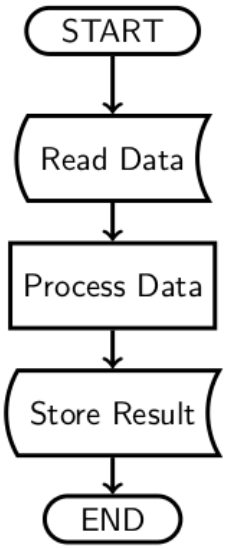
Contents

1 Program Structure

Structured programming should produce better software in a shorter development cycle than unstructured programming.

1. Programme structure is all about organisation:
    - Keep related functions together in a file/unit
    - Put reusable sections in a library (more on libraries later)
    - Arrange your programme flow in a logical manner
2. Programme structure is done upfront, not as a last ditched reorganisation to try and get things working.

Software will typically read in some data, process the data and output a result.

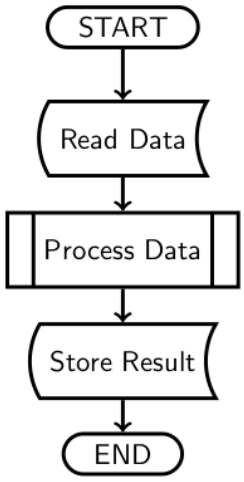


When creating the sub-tasks the inputs and outputs are identifed, along with an overall description of what 'process' is going to be applied to the data. *This is easily documented at the start.*

Each sub-task is designed along with the test to verify proper operation of the sub-task.

Putting the working sub-tasks together should give properly working software. If it does not work something was wrong in the design - the overall design or the design of the sub-tasks. Then re-design and update the version numbers on the sub-tasks.

Processing the data may in itself be a fairly complex operation with many sub-tasks or *functions.*



'Process Data' is shown here as a 'predefined process' which needs its own description or flow chart to explain its operation.

'Process Data' probably has its own sub-tasks or functions, along with documentation and test methods to verify proper operation of each sub-task.

Breaking the overall problem down into a set of functions or sub-tasks is often referred to as *heirarchial design*.

Its is generally suggested that the functions or sub-tasks are 5-20 lines of code at most. In such lengths it is easier to understand tha code and *find all the faults.*

Note

We all make mistakes.

Observation

Suppose we have an 'error rate' of 1%. In a function with 20 lines of code, if each line has 5 variables and operations on those variables then there is quite likely (50:50?) that there is an error in that function.

If we write 500 such lines of code, there are likely to be 25 errors *somewhere*. Dividing into functions and testing the functions *localises* the error.

How NOT to programme

Examples of poor approaches in writing software:

1. Open a file and start writing
2. Get it written and add the comments later
3. Start with variable a
4. Make all your variables global
5. Assume all file and keyboard inputs are perfect
6. Assume that a programme that compiles is finished
7. Assume that a programme that works once is finished
8. Write code in as small a space as possible
9. Sort out indentation once the code works
10. Write code and then write/record tests once it all works
11. Write lots of code and add some tests as a last job

Consider:

Do you think you might follow some or all of the above traits?

Software implementation

Design vt: to draw: to form a plan of: to contrive: to intend: to set apart or destine. **[n:]** a drawing or sketch: a plan in outline: plot: intention.

Contrive vt: to plan: to invent: to bring about or effect: to manage, arrange: to plot: to conceive.

Software implementation - a detailed process:

1. Requirements Analysis and Specification
   - What functions is the software to do?
   - Are there time constraints?
   - Is there a design process to follow, such as ISO 13407 or ISO 15504 and their checklist ISO TR 18529?
2. Design - top down and bottom up.
   - Consider alternative ways of satisfying the Requirement Specification.
   - Rank the alternatives to find the best solution.

- Think through how to test the design.

3. Decomposition in the design
   - Factorise the problem into well defined (simple?) functions. Comment/document their definitions.
   - Work out how to test the functions - need to debug each function and then test/debug the whole thing.
   - Factorise the data storage/handling. Comment/document their definitions.

4. Implementation
   - Coding (25-30% of the time?) to implement the functions and the overall program. Amend comments & documentation as things develop and the code improves into something better. Testing the functions and program to show it works to yourself *and your management/customer*.
   - Module testing - recording test results. Show it works to yourself *and your management/customer*.
   - Application testing - recording test results. Show it works to yourself *and your management/customer*.

Starting the coding process early makes the project longer!

Note

It should be that the Design can be given to a competent 'coder' to then implement and test the code to meet the Design. Modifications to the Design may become apparent/necessary during the implementation process.

Rules of thumb:

50% to 80% of defects can be traced to poor Requirements definition.

Fixing things at final testing, or due to customer complaints, is 200-400 times more expensive than if spotted at Requirements definition.

Good Practice

Plan:

Interaction: I/O with the user, hardware or operating system.

Algorithm: If you can't write it down, you can't code it!

Data Structures: Organise your data into logical groups.

Functions: Do one thing at a time. Each action has its own function.

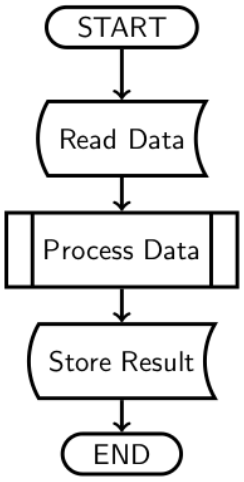
Testing: *Prove* that your functions work individually, and are robust - as they are debugged.

Prove that your overall design is working and robust - to yourself and your management/customer.

Testing

Testing should identify that the function is performing the desired actions/operations.

When designing functions we need to make a 'statement' about what is the proper action/operation of a function. Then we can test against this.



Consider the 'Store data' part. By inserting known data with a predictable result just before Store Data is called the output can be tested for conformance with the predicted output.

If differences are apparent the Store Data function *is not* working.

If the output is exactly what is predicted Store Data *may be* working. The test might not be sufficient to show up all possible errors.

Test the individual functions.

Testing helps you debug the individual functions, isolating errors within a function.

If individual functions work OK but the overall program does not there may be an error in the logic of the interaction between the functions or the flow through them.

Recording tests convinces a customer/boss that you are working properly and professionally. You have a record to back up your assertion that parts of the complete program are working.

If you haven't tested it, it doesn't work!

If it is not tested the customer is unlikely to pay for it - and your boss may not be very pleased.

Test, test and test again.

Oh, and then test some more.

Do lots of testing before the customer is anywhere near the software (even when operating under ISO 13407, ISO 9241-210, ISO 15504 schemes).

If it fails on a presentation day?

The testing was insufficient - it did not identify the fault now encountered.

Mouse project suffers from this as well!

All projects suffer from this as well!

Conclusions

1. Structured programming is not about opening an editor and starting to type
    - That is coding!
2. Planning and organising your programme and data is vital.
3. So is knowing what your programme is supposed to do. Needs to be tested to show that it is doing what it is supposed to do.
4. Write the testing strategy for a function when the function is first outlined. *(additional tests may need to be added later - use version numbers)*
5. Proper commenting and documenting is there to help define the functions and programme from the outset.

Don't forget to test!

Ideally every function (MyFunction?) has an associated function (Test_MyFunction which tests its operation against known outcomes.

2 Basic Input/Output and string functions

Any program must be able to interact with the users of the program in some way.

The general run of all programs is to

1. Read in data from user or from a file.
2. Do something to the data.
3. Write out the result to screen or to a file.

The input and output of information is frequently done using strings. This is the natural way to deal with text descriptions. With data values when they are rendered into strings the values become easy to read on a computer screen or in data files - often called 'human readable'.

The Python functions to handle strings are discussed later.

The basic Input/Output via the keyboard and screen produces an interaction with the user.

```
name=input('Please enter your name: ')
print("Hello "+name)
```

This is rather tedious when there are many variables to input. When outputting the values of many variables there may well be a good reason for storing the output. File input/output is most frequently used.

File I/O

In order to produce input or output with files the files need to be opened first. This is done via the python function fopen:

```
filename="Datafile_example.dat"
fp=open( filename,'rt')
```

The string variable filename holds the name of the file to be opened. The second parameter to the fopen function is the mode for the connection to the file.

The mode string contains one or more characters from the following:

r Reading mode, file is to be read from.

w Writing mode, file is for writing to. Old versions are overwritten.

a Append mode, file is for writing to, adding new data to the end of the previous version of the file.

t Text mode, file contains 8-bit ASCII characters (simple .txt file).

b Binary mode - file contains raw binary format, 4 byte integer, 4 byte float, 8 byte double, etc.

Once the input/output is finished the file should be closed:

```
inputfilename="Datafile_example.dat"
fin=open( inputfilename,'rt')
outputfilename="Outputfile_example.dat"
fout=open( outputfilename,'wt')

# rest of the program where files are used

fout.close()
fin.close()
```

Handling file opening errors

The opening of files can go wrong for a number of reasons, such as 'File system Full' or 'File does not exist'. Using a try:except construction this error can be identified and reported to the user.

```
import sys
import os

inputfilename="Datafile_example.dat"
# fin=open( inputfilename,'rt')
try:
    inputfilename="Datafile_example.dat"
    fin=open( inputfilename,'rt')
except FileNotFoundError:
    print('File <%s> not found'%(inputfilename))
    current_location=os.getcwd()
    # gets the directory where the program is executing from the operating
    # system as this is where the file is expected to be
    print("executing program in directory: "+current_location)
    sys.exit(2)  # exits the program, returning a value of 2 to the operating
     system
```

Writing to the file

To write to a file a string can be created which is then written to the file.

```
xx=17.325
nn=99
message='No place like home'

fout=open( 'Output.txt','wt')
op_string="X=%g, N=%d, Message=%s"%(xx,nn,message)
fout.write(op_string)

fout.close()
```

The format used to create the output string is easily controlled using standard formatting codes:

Code	Data Type	Example
%d or %i	integer	1024
%f	float	376.9911
%8.2f	float	376.99
%e	float	3.769911e+02
%12.3e	float	3.770e+02
%g	float	%f or %e if %f too wide
%s	string	The quick brown fox
%o	octal	0o1212
%x	Hex(lowercase)	2f
%X	Hex(uppercase)	2F
%5.4X	Hex(uppercase)	002F

The fields widths can be left at their default values or explicitly set, as in the second and third outputs:

```
xx=17.325
nn=99
message='No place like home'

fout=open( 'Output.txt','wt')
op_string="X=%g, N=%d, Message=<%s>\n"%(xx,nn,message)
fout.write(op_string)
op_string="X=%7.1f, N=%5d, Message=<%s>\n"%(xx,nn,message)
fout.write(op_string)
op_string="X=%12.4e, N=%3d, Message=<%s>\n"%(xx,nn,message)
fout.write(op_string)

fout.close()
```

Output.txt contains:

```
X=17.325, N=99, Message=<No place like home>
X=    17.3, N=   99, Message=<No place like home>
X=  1.7325e+01, N= 99, Message=<No place like home>
```

Reading from the file

read()

Using read() the entire contents of a file are read into a python variable.

```
fin=open( 'Datafile_example.dat','rt')
all_file=fin.read()
print("File contents are:")
print(all_file)
fin.close()
```

Output is:

```
File contents are:
x=3.2 n=4 Name=Bath
x=7.4 n=1 Name=Birmingham
x=1.3 n=2 Name=Edinburgh
x=3.2, n=4, Name=Bath
n=4 x=3.2 Name=Bath
```

Having read in the entire file it needs to be split into individual items, so the structure of the data file needs to be known. The number of characters, words and lines in the data can be identified:

```
fin=open( 'Datafile_example.dat','rt')
all_file=fin.read()
chars=len(all_file)
words=len(all_file.split())  # splits on space character
lines=len(all_file.splitlines())
print("File contained %d characters, %d words and %d lines"%(chars, words,
    lines))
fin.close()
```

The resulting output is

```
File contained 114 characters, 15 words and 5 lines
```

For large data files it may be cumbersome and memory intensive to read in the whole file. Data files are generally a set of separate lines.

readline()

Lines in the data file can be read in one by one using readline():

```
fin=open( 'Datafile_example.dat','rt')
first_line=fin.readline()
second_line=fin.readline()
print("First line=<%s>"%(first_line))
print("Second line=<%s>"%(second_line))
fin.close()
```

The output produced is

```
First line=<x=3.2 n=4 Name=Bath
>
Second line=<x=7.4 n=1 Name=Birmingham
>
```

Note each line contains a newline character at the end of the line.

readlines()

Many lines in the data file can be read in using readlines():

```
fin=open( 'Datafile_example.dat','rt')
file_lines=fin.readlines()
for index,line in enumerate(file_lines):
    print("Line[%d] =<%s>"%(index,line))
fin.close()
```

The output is:

```
Line[0] =<x=3.2 n=4 Name=Bath
>
Line[1] =<x=7.4 n=1 Name=Birmingham
>
Line[2] =<x=1.3 n=2 Name=Edinburgh
>
Line[3] =<x=3.2, n=4, Name=Bath
>
Line[4] =<n=4 x=3.2 Name=Bath
>
```

Extracting values from text lines

Once lines are read in they generally have to be interpreted to extract particular values into variables. The split() function will split a string at the space characters, while split(',') will split at the comma characters.

```
filename="Datafile_example.dat"
fp=open( filename,'rt')

Aline=fp.readline()      #read a line
print("Line read from file is <%s>"%(Aline))
Splitline=Aline.split()    #split line at the space characters
print("Split the line gives "+str(Splitline))
N_terms=len(Splitline)
print("Line has split into %d elements"%(N_terms))
first=Splitline[0].split('=')  #Split part of the line at = character
print("First part of Splitline splits on = to "+str(first))
Varname=first[0]
xvalue=float(first[1])
print("Found variable <%s> with value %g"%(Varname,xvalue))
second=Splitline[1].split('=')
print("Second part of Splitline splits on = to "+str(second))
Varname2=second[0]
nvalue=int(second[1])
print("Found variable <%s> with value %d"%(Varname2,nvalue))
third=Splitline[2].split('=')
print("Third part of Splitline splits on = to "+str(third))
Varname3=third[0]
place=third[1]
print("Found variable <%s> with value <%s>"%(Varname3,place))
fp.close()
```

Listing 1: 'Basic interpretation of input string'

The output produced is:

```
Line read from file is <x=3.2 n=4 Name=Bath
>
Split the line gives ['x=3.2', 'n=4', 'Name=Bath']
Line has split into 3 elements
First part of Splitline splits on = to ['x', '3.2']
Found variable <x> with value 3.2
Second part of Splitline splits on = to ['n', '4']
Found variable <n> with value 4
Third part of Splitline splits on = to ['Name', 'Bath']
Found variable <Name> with value <Bath>
```

This has then identified the variable name and its value. Note that when reading the line into a string (Aline in this case) the 'newline' character is contained in the string. There are many inbuilt Python functions to operate on strings, so a wide range of strategies can be implemented to extract values from the text strings read in. Python string functions are included later in the notes.

Setting filenames

In the above examples the filename to read from or write to is set in the Python code. File names can be read in from the terminal:

```
input_filename=input('Please enter the input filename: ')
output_filename=input('Please enter the output filename: ')
print("Input file is <%s>, output file is <%s>"%(input_filename,
    output_filename))
```

Filenames can also be given on the command line:

```
python MyProgram.py input.dat output.dat
```

The Python program then needs to act on the input arguments which are stored in a list called argv:

```
import sys
print "This is the name of the script: ", sys.argv[0]
print "Number of arguments: ", len(sys.argv)
print "The arguments are: " , str(sys.argv)
input_filename=argv[1]
output_filename=argv[2]
```

The output will be

```
This is the name of the script:  MyProgram.py
Number of arguments:  3
The arguments are:  ['Myprogram.py', 'input.dat', 'output.dat']
```

Strings

Strings are implicitly included within Python. In C or C++ string.h needs to be 'included' in code to bring in the library of string functions.

st.capitalize() : Capitalize st eg. 'hello' => 'Hello'

st.lower() : Lowercase st eg. 'HELLO' => 'hello'

st.swapcase() : Swap cases of all characters in st eg. 'Hello' => "hELLO"

st.title() : Titlecase st eg. 'hello world' => 'Hello World'

st.upper() : Uppercase st eg. 'hello' => 'HELLO'

s2 in st : Return true if st contains s2

st + s2 : Concatenate st and s2

len(st) : Length of st

min(st) : Smallest character of st

max(st) : Largest character of st

s2 not in st : Return true if st does not contain s2

st * integer : Return integer copies of st concatenated eg. 'hello' => 'hellohellohello'

Indexing : st[i] gives character at index i of st. st[i:j]gives the slice of st from i to j. st[i,j,k] gives the slice of st from i to j with step k.

st.count(s2): Count of s2 in st

st.center(width) : Center st with blank padding of width eg. 'hi' => ' hi '

st.isspace() : Return true if st only contains whitespace characters

st.ljust(width) : Left justify st with total size of width eg. 'hello' => 'hello '

st.rjust(width) : Right justify st with total size of width eg. 'hello' => ' hello'

st.strip() : Remove leading and trailing whitespace from st eg. ' hello ' => 'hello'

st.index(s2, i, j) : Index of first occurrence of s2 in st after index i and before index j

st.find(s2) : Find and return lowest index of s2 in st

st.index(s2) : Return lowest index of s2 in st (but raise ValueError if not found)

st.replace(s2, s3) : Replace s2 with s3 in st

st.replace(s2, s3, count) : Replace s2 with s3 in st at most count times

st.rfind(s2) : Return highest index of s2 in st

st.rindex(s2) : Return highest index of s2 in st (raise ValueError if not found)

st.islower() : Return true if st is lowercase

st.istitle() : Return true if st is titlecased eg. 'Hello World' => true

st.isupper() : Return true if st is uppercase

st.endswith(s2) : Return true if st ends with s2

st.isalnum() : Return true if st is alphanumeric

st.isalpha() : Return true if st is alphabetic

st.isdecimal() : Return true if st is decimal

st.isnumeric() : Return true if st is numeric

st.isdigit() : Return true if st is digit

st.isidentifier(): Return true if st is a valid identifier

st.isprintable() : Return true is st is printable

st.startswith(s2) : Return true is st starts with s2

st.endswith((s1, s2, s3)) : Return true if s ends with any of string tuple s1, s2, and s3

st.join('123') : Return st joined by iterable '123' eg. 'hello' => '1hello2hello3'

st.partition(sep) : Partition string at sep and return 3-tuple with part before, the sep itself, and part after eg. 'hello' => ('he', 'l', 'lo')

st.rpartition(sep) : Partition string at last occurrence of sep, return 3-tuple with part before, the sep, and part after eg. 'hello' => ('hel', 'l', 'o')

st.rsplit(sep, maxsplit) : Return list of st split by sep with rightmost maxsplits performed

st.split(sep, maxsplit) : Return list of st split by sep with leftmost maxsplits performed

st.splitlines() : Return a list of lines in st eg. 'hello\nworld' => ['hello', 'world']

st.center(width, pad): Center s with padding pad of width eg. 'hi' => 'padpadhipadpad'

st.expandtabs(N): Replace all tabs with N spaces eg. 'hello\tworld' => 'hello world'

st.lstrip(): Remove leading whitespace from st eg. ' hello ' => 'hello '

st.rstrip(): Remove trailing whitespace from st eg. ' hello ' => ' hello'

st.zfill(width): Left fill s with ASCII '0' digits with total length width eg. '42' => '00042'

Finally

Input and output is an important part of any program - otherwise there is no interaction and the program generally has nothing to do.

Input and output with the user at a terminal is used to create interactive programs. If there is a lot of data to be input or output data files are often preferred.

To set up file input and output the files need to be opened so that the program can interact with the data files held by the host operating system. This process can lead to errors, which need ot be detected and corrected.

Formatting of data output is often used in order to make the output files easier to read and understand by humans.

Python has many functions to deal with input and output in various ways. In particular there is a very rich collection of functions to operate on strings.

3 Coursework Task

The complete description of the Coursework or Mini-Project is in a separate document and on the Moodle site for EE20084. The Coursework is 50% of the assessment for EE20084. In this you are to produce:

Design Document (20%): Describe the design and test plan for the program that is to be developed.

Code submission (40%): The .py python files for your program.

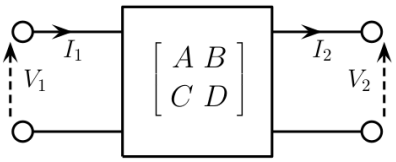
Report submission (40%): A report describing the final code that has been developed and its testing.

The Coursework is to write a program that will analyse very general electrical circuits connected in the very common *cascade* connection. This is most easily done using the ABCD or chain matrix analysis method.

ABCD or Chain Matrix

The ABCD matrix relates voltages and currents at the input and output connection ports of a circuit element. The ABCD matrix is also known as the chain matrix or voltage transmission matrix.

Consider a linear two-port circuit with voltages V_1 and V_2 and currents I_1 and I_2 at the two ports.



$$\begin{aligned} V_1 &= AV_2 + BI_2 \\ I_1 &= CV_2 + DI_2 \end{aligned}$$

The equation set is cast into the ABCD matrix form as

$$\begin{bmatrix} V_1 \\ I_1 \end{bmatrix} = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} V_2 \\ I_2 \end{bmatrix} = [T] \begin{bmatrix} V_2 \\ I_2 \end{bmatrix} \tag{1}$$

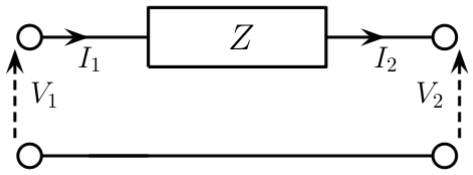
This can be simply inverted to find:

$$\begin{bmatrix} V_2 \\ I_2 \end{bmatrix} = [T]^{-1} \begin{bmatrix} V_1 \\ I_1 \end{bmatrix} = \frac{1}{AD - BC} \begin{bmatrix} D & -B \\ -C & A \end{bmatrix} \begin{bmatrix} V_1 \\ I_1 \end{bmatrix} \tag{2}$$

The four parameters of the matrix can be determined by examination of the circuit with open and short circuit load conditions.

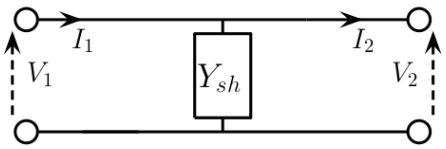
Examples of ABCD Matrices

Series Impedance and Shunt Admittance



In the case of a series impedance, Z, the ABCD matrix is:

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} = \begin{bmatrix} 1 & Z \\ 0 & 1 \end{bmatrix} \quad (3)$$



In the case where the shunt element is an admittance $Y_{sh} = 1/Z_{sh}$:

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ Y_{sh} & 1 \end{bmatrix} \quad (4)$$

Cascade of several circuit elements

The cascade connection is a *very common* way of connecting components. Consider the case where there are several two-port circuits in cascade.

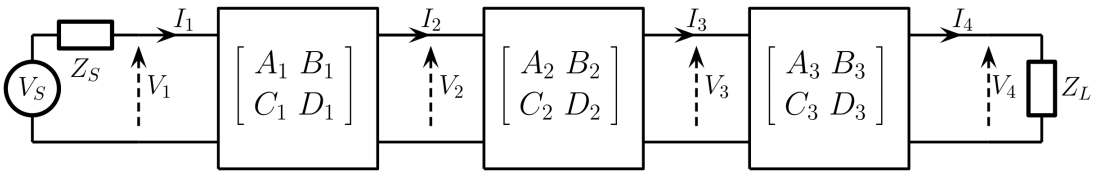


Figure 3.1: Cascade of three 2-port circuits.

The behaviour of the overall circuit is determined by the ABCD matrices of the elements of the cascade:

$$\begin{aligned}
\begin{bmatrix} V_1 \\ I_1 \end{bmatrix} &= \begin{bmatrix} A_1 & B_1 \\ C_1 & D_1 \end{bmatrix} \begin{bmatrix} V_2 \\ I_2 \end{bmatrix} = [T_1] \begin{bmatrix} V_2 \\ I_2 \end{bmatrix} \\
\begin{bmatrix} V_2 \\ I_2 \end{bmatrix} &= \begin{bmatrix} A_2 & B_2 \\ C_2 & D_2 \end{bmatrix} \begin{bmatrix} V_3 \\ I_3 \end{bmatrix} = [T_2] \begin{bmatrix} V_3 \\ I_3 \end{bmatrix} \\
\begin{bmatrix} V_3 \\ I_3 \end{bmatrix} &= \begin{bmatrix} A_3 & B_3 \\ C_3 & D_3 \end{bmatrix} \begin{bmatrix} V_4 \\ I_4 \end{bmatrix} = [T_3] \begin{bmatrix} V_4 \\ I_4 \end{bmatrix}
\end{aligned} \quad (5)$$

Relating the voltages and currents from left to right:

$$\begin{aligned}\begin{bmatrix} V_1 \\ I_1 \end{bmatrix} &= [T_1]\begin{bmatrix} V_2 \\ I_2 \end{bmatrix} \\ &= [T_1]\,[T_2]\begin{bmatrix} V_3 \\ I_3 \end{bmatrix} \\ &= [T_1]\,[T_2]\,[T_3]\begin{bmatrix} V_4 \\ I_4 \end{bmatrix} = \begin{bmatrix} A_C & B_C \\ C_C & D_C \end{bmatrix}\begin{bmatrix} V_4 \\ I_4 \end{bmatrix}\end{aligned} \tag{6}$$

So the ABCD matrix of the cascade network is determined by multiplying together the ABCD matrices of the elements in the order they appear in the cascade circuit. This is why the matrix is sometimes referred to as the chain or voltage transmission matrix.

Observation

For a general cascade circuit the overall ABCD matrix is simply the multiple of the ABCD matrices of the individual components of the cascade, in the order that they occur in the cascade:

$$[T_C] = [T_1]\,[T_2]\,[T_3]\,\ldots[T_M] \tag{7}$$

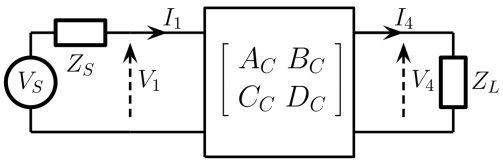


Figure 3.2: Loaded 2-port circuit described by ABCD matrix.

From the ABCD matrix of the complete cascade network various features of the complete network are easily found, such as input impedance, voltage gain, current gain, power gain.....

Input and output impedance

When a load, Z_L, is connected at the end of the cascade $\frac{V_4}{I_4} = Z_L$ and the input impedance as seen by the source, Z_{in} is

$$Z_{in} = \frac{V_1}{I_1} = \frac{A_C V_4 + B_C I_4}{C_C V_4 + D_C I_4} = \frac{A_C \frac{V_4}{I_4} + B_C}{C_C \frac{V_4}{I_4} + D_C} = \frac{A_C Z_L + B_C}{C_C Z_L + D_C} \tag{8}$$

When a source, Z_S, is connected the output impedance as seen at the load end, Z_{out} is

$$Z_{out} = \frac{D_C Z_S + B_C}{C_C Z_S + A_C} \tag{9}$$

Voltage, current and power gain

As $\frac{V_4}{I_4} = Z_L = \frac{1}{Y_L}$ using the base definition of the ABCD matrix:

$$V_1 = A_C V_4 + B_C I_4 = A_C V_4 + B_C V_4 Y_L \tag{10}$$

the voltage gain is:

$$A_V = \frac{V_4}{V_1} = \frac{1}{A_C + B_C Y_L} = \frac{Z_L}{A_C Z_L + B_C} \tag{11}$$

Likewise

$$I_1 = C_C V_4 + D_C I_4 = C_C I_4 Z_L + D_C I_4 \tag{12}$$

and the current gain is:

$$A_I = \frac{I_4}{I_1} = \frac{1}{C_C Z_L + D_C} \tag{13}$$

The power gain is:

$$A_P = A_V A_I^* \tag{14}$$

where the complex conjugate of the current gain is needed.

Transmittance

The transmittance from source to load is:

$$T = \frac{2}{A_C Z_L + B_C + C_C Z_L Z_S + D_C Z_S} \tag{15}$$

Familiarisation Exercises

It is often easier to develop functions or programs to implement an algorithm if you can perform the operations yourself 'by-hand'. Examples that you are convinced are correct often (always?) form test cases for you to prove your software programs against.

These familiarisation exercises illustrate the basic steps that are needed to perform an ABCD matrix circuit analysis.

Example 1

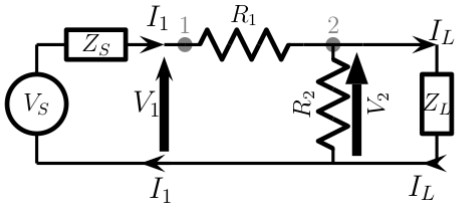


Figure 3.3: A two element cascade circuit of series and shunt resistances.

Question 1

In the case where $Z_S = R_1 = R_2 = Z_L = 50\Omega$ calculate the ABCD matrix, Z_{in} and A_v.

Question 2

In the case where $Z_S = Z_L = 50\Omega$ and $R_1 = R_2 = 150\Omega$ calculate the ABCD matrix, Z_{in} and A_v.

Question 3

In the case where $Z_S = Z_L = 50\Omega$, $R_1 = R_2 = 150\Omega$ and $V_s = 5\,Volts$ calculate $V_1,\ I_1,\ V_2$ and I_L.

Example 2

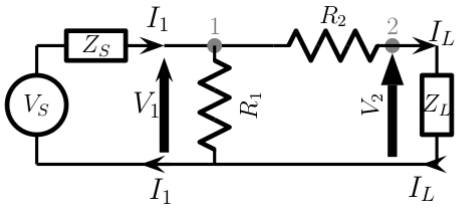


Figure 3.4: A two element cascade circuit of shunt and series resistances.

Question 4

In the case where $Z_S = R_1 = R_2 = Z_L = 50\Omega$ calculate the ABCD matrix, Z_{in} and A_v.

Question 5

In the case where $Z_S = Z_L = 50\Omega$ and $R_1 = R_2 = 150\Omega$ calculate the ABCD matrix, Z_{in} and A_v.

Question 6

In the case where $Z_S = Z_L = 50\Omega$, $R_1 = R_2 = 150\Omega$ and $V_s = 5\ Volts$ calculate $V_1,\ I_1,\ V_2$ and I_L.

The task - core part

You should write a program that analyses cascade circuits, such as shown in Figure 3.5, where series and shunt impedances and admittances of any value can be connected in any order between a source and a load.

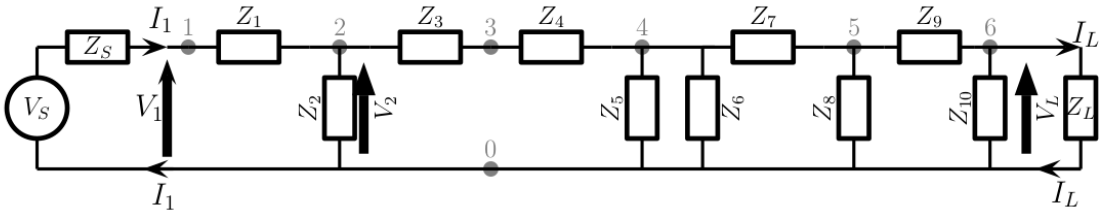


Figure 3.5: A cascade circuit of series and shunt impedances. Node numbers, 1 to 6 in this case, show how components are connected. Node zero is the common (ground?) connection.

The program is to read a data file, say 'test.net', and output to a results file, say 'test.csv'. These files are to be specified in the command line, so to run 'MyCode.py' on these files the command line should be

```
python MyCode.py test.net test.csv
```

The input data file, 'test.net', describes the circuit, and will be in the following format where there must be a CIRCUIT block, a TERMS block and an OUTPUT block. The OUTPUT block defines what is to be printed into the output file 'test.csv'.

Note that lines starting with the hash character, #, are comment lines.

```
# define a circuit between <CIRCUIT> and </CIRCUIT> delimiters
# Elements have two node numbers and component value is a
# resistance R=1/G (Ohms), a conductance G=1/R (Seimens), an inductance (Hen
# or a capacitance (Farads).
<CIRCUIT>
n1=1 n2=2 R=8.55
n1=2 n2=0 R=141.9
n1=2 n2=3 R=8.55
n1=3 n2=4 L=1.59e-3
n1=4 n2=0 C=3.18e-9
n1=4 n2=0 L=7.96e-6
n1=4 n2=5 C=6.37e-7
n1=5 n2=0 R=150.5
# components do not have to follow their order in the circuit
n1=6 n2=0 R=150.5
n1=5 n2=6 G=0.02677
</CIRCUIT>

# define the terminations between <TERMS> and </TERMS> delimiters
<TERMS>
# A 5V Thevenin voltage source with RS=50 ohms connected
# between node 1 and the implicit common (0) node
VT=5 RS=50
# A 2.5 Amp Norton current source with RS=25 Ohms
#IN=2.5 RS=25
#IN=2.5 GS=0.04
# Load connected between last node (6 in this case) and the implicit common
RL=75
# Frequency range and number of frequencies to evaluate at.
Fstart=10.0 Fend=10e+6 Nfreqs=10
</TERMS>
```

```
# define the outputs between <OUTPUT> and </OUTPUT> delimiters
# Order of parameters defines the order they appear in the columns
# output file and their units.
<OUTPUT>
Vin V
Vout V
Iin A
Iout A
Pin W
Zout Ohms
Pout W
Zin Ohms
Av
Ai
</OUTPUT>
```

The CIRCUIT, TERMS and OUTPUT blocks can appear in any order, but all three must be present and have valid content.

In a <CIRCUIT> block the components can be specified in any order. The node numbers determine their interconnection order.

In the <OUTPUT> block the data to be output can be Vin, Iin, Pin, Zin, Vout, Iout, Pout, Zout, Av or Ai. The order that the output variables appear in the output file is defined by the order given in the input data file. The units for a variable are linear values by default.

The variables to be output can be:

- Vin - Input Voltage
- Iin - Input Current
- Pin - Input Power
- Zin - Input Impedance
- Vout - Output Voltage
- Iout - Output Current
- Pout - Output Power
- Zout - Output Impedance

- Av - Voltage Gain
- Ai - Current Gain

The first line of the output file should be the variable type. The second line of the output file should be the units for the output variables and the third and subsequent lines are the variable values. Apart from the frequency the output values are all complex numbers in either real and imaginary format, or when using decibel outputs in magnitude and phase. The commas following the labels and the data should all align. The units for a linear gains should be L. The following output in file 'a_Test_Circuit_1.csv' results from the example input file 'a_Test_Circuit_1.net'.

```
     Freq,    Re(Vin),    Im(Vin),   Re(Vout),   Im(Vout),    Re(Iin),    Im(Iin),   Re(Iout),   I
       Hz,          V,          V,          V,          V,          A,          A,          A,
1.000e+01,  1.247e+00,  5.032e-03, -4.487e-08,  1.894e-10,  7.506e-02, -1.006e-04, -5.983e-10,  2.
1.111e+06,  3.753e+00,  1.130e-02, -2.138e-03, -9.601e-03,  2.495e-02, -2.260e-04, -2.850e-05, -1.
2.222e+06,  3.753e+00,  5.649e-03, -2.041e-03, -1.054e-03,  2.494e-02, -1.130e-04, -2.721e-05, -1.
3.333e+06,  3.753e+00,  3.764e-03, -9.215e-04, -2.779e-04,  2.494e-02, -7.528e-05, -1.229e-05, -3.
4.444e+06,  3.753e+00,  2.822e-03, -5.175e-04, -1.122e-04,  2.494e-02, -5.645e-05, -6.899e-06, -1.
5.556e+06,  3.753e+00,  2.258e-03, -3.306e-04, -5.624e-05,  2.494e-02, -4.515e-05, -4.407e-06, -7.
6.667e+06,  3.753e+00,  1.881e-03, -2.293e-04, -3.218e-05,  2.494e-02, -3.763e-05, -3.057e-06, -4.
7.778e+06,  3.753e+00,  1.612e-03, -1.683e-04, -2.012e-05,  2.494e-02, -3.225e-05, -2.244e-06, -2.
8.889e+06,  3.753e+00,  1.411e-03, -1.288e-04, -1.342e-05,  2.494e-02, -2.822e-05, -1.717e-06, -1.
1.000e+07,  3.753e+00,  1.254e-03, -1.017e-04, -9.397e-06,  2.494e-02, -2.508e-05, -1.356e-06, -1.
```

The Task - extension parts

Exponent prefixes

Add the exponent prefixes (p, n, u, m, k, M, G) and use in all input and output variables.

Symbol	Prefix	Factor
p	pico	10^{-12}
n	nano	10^{-9}
u	micro (μ)	10^{-6}
m	milli	10^{-3}
k	kilo	10^{3}
M	mega	10^{6}
G	giga	10^{9}

Decibels and phase

Add the ability to express the outputs in decibels. The <OUTPUT> block would be of the form:

```
<OUTPUT>
Vin V
Vout dBV
Iin A
Iout uA
Pin dBW
Pout dBmW
Zin Ohms
Zout kOhms
Av dB
Ai dB
</OUTPUT>
```

The example input file "Ext_a_Test_Circuit_1.net" generates the output file "Ext_a_Test_Circuit_1.csv". The output should be decibels and the argument or phase in radians of the complex value. The separator between the magnitude in dB and the phase should be /_. Example output is:

```
     Freq,    Re(Vin),    Im(Vin),     |Vout|,     /_Vout,    Re(Iin),    Im(Iin),   P
       Hz,          V,          V,        dBV,       Rads,          A,          A,
1.000e+01,  1.247e+00,  5.032e-03, -1.470e+02,  3.137e+00,  7.506e-02, -1.006e-04, -5.
4.642e+01,  3.753e+00,  1.130e-02, -4.014e+01, -1.790e+00,  2.495e-02, -2.260e-04, -2.
2.154e+02,  3.753e+00,  5.649e-03, -5.278e+01, -2.665e+00,  2.494e-02, -1.130e-04, -2.
1.000e+03,  3.753e+00,  3.764e-03, -6.033e+01, -2.849e+00,  2.494e-02, -7.528e-05, -1.
4.642e+03,  3.753e+00,  2.822e-03, -6.552e+01, -2.928e+00,  2.494e-02, -5.645e-05, -6.
2.154e+04,  3.753e+00,  2.258e-03, -6.949e+01, -2.973e+00,  2.494e-02, -4.515e-05, -4.
1.000e+05,  3.753e+00,  1.881e-03, -7.271e+01, -3.002e+00,  2.494e-02, -3.763e-05, -3.
4.642e+05,  3.753e+00,  1.612e-03, -7.542e+01, -3.023e+00,  2.494e-02, -3.225e-05, -2.
2.154e+06,  3.753e+00,  1.411e-03, -7.776e+01, -3.038e+00,  2.494e-02, -2.822e-05, -1.
1.000e+07,  3.753e+00,  1.254e-03, -7.982e+01, -3.049e+00,  2.494e-02, -2.508e-05, -1.
```

Logarithmic frequency sweep

Add the ability to calculate at logarithmically spaced frequencies as specified in the <TERMS> block as

```
LFstart=10.0 LFend=10e+6 Nfreqs=10
```

The 10 values of $\log(f)$ are then evenly spaced, but the linear frequencies, f, are to be output. Such output is shown in the last example of an output file.

Program execution time

Programs will be tested for speed of execution. The execution time is the time taken for the complete Python program to execute from the command line to the program terminating. Best results will be obtained for a program that produces valid output files in the shortest execution time.

4 Classes

Collecting or grouping data

Arrays are a useful way of collecting a set of data where the elements are associated with each other *and* they are all of the same data type. For example the x and y values for drawing a graph.

x[10]
y[10]

For collecting associated data of *differing data types* a Class is used. This helps the programmer organise complicated sets of data together rather than having the data known by separate identities/variable names.

A traditional example of a structure is the payroll record. An employee is described by a set of attributes that we store in instances of a class:

```
class record_card:
   """ class to store a name, address, insurance_number, salary,
   number_of_years_service and age
   """
   def __init__(self, p_name, p_address, p_insurance_number, p_salary,
    p_number_of_years_service, p_age):
      self.name=p_name
      self.address=p_address
      self.insurance_number=p_insurance_number
      self.salary=p_salary
      self.number_of_years_service=p_number_of_years_service
      self.age=p_age
```

The class has an initiating function called __init__ which sets the values of the member fields to the parameters passed when the instance of the class is created.

```
# create two instances, each with their own values
Mystery=record_card("Me.E", "The Big Hoose, The Trossachs", "YY0101893BRXT",
    10000.10, 12, 28)
ANOther=record_card("Me.O", "The Wee Hoose, The Trossachs", "VDH985702EXTN",
    50000.10, 32, 58)
```

The fields of a class are accessed using the fullstop operator:

```
# Access values
average_salary=0.5*(Mystery.salary+ANOther.salary)
# update a value
Mystery.age=35
```

It is often useful to be able to print out the values of all of the member fields, so a function can be added to the definition.

```
class record_card:
   """ class to store a name, address, insurance_number, salary,
number_of_years_service and age
   """
   def __init__(self, p_name, p_address, p_insurance_number, p_salary,
    p_number_of_years_service, p_age):
      self.name=p_name
      self.address=p_address
      self.insurance_number=p_insurance_number
      self.salary=p_salary
      self.number_of_years_service=p_number_of_years_service
      self.age=p_age
   def print_contents(self):
      print("Record : Name=<%s>"%(self.name))
      print("Address=<%s>"%(self.address))
      print("Salary=%.2f, Years=%d, age=%d"%(self.salary, self.
    number_of_years_service,self.age))
```

Using the print_contents function:

```
Mystery.print_contents()
ANOther.print_contents()
```

A second example

A class to facilitate using cartesian vectors will store the components of the vector, but in addition can provide useful features such as addition, subtraction, dot product, cross product, print the vector, normalise the vector.....

The initial declaration of the class sets up the fields of the class when an instance of the class is created, with the fields of the class referred to through 'self':

```
class cartesian_vector:
   """ class to store a cartesian vector and provide basic operations
   """
   def __init__(self, p_x, p_y, p_z):
      self.x=p_x
      self.y=p_y
      self.z=p_z
```

Listing 2: "Basic definition of a class to handle cartesian vectors."

Creating instances of the vector with initial values is simple:

```
# create two instances, each with their own values
a=cartesian_vector(1.0,2.0,3.0)
b=cartesian_vector(4.0,-5.0,6.0)
```

To observe the values in a vector add a print_contents function or method:

```
class cartesian_vector:
   """ class to store a cartesian vector and provide basic operations
   """
   def __init__(self, p_x, p_y, p_z):
      self.x=p_x
      self.y=p_y
      self.z=p_z
   def print_contents(self, name):
      print("Vector: Name=<%s>, x=%11.4e, y=%11.4e, z=%11.4e"%(name, self.x,
    self.y, self.z))
   def write_file(self, fileptr, name):  # fileptr = already opened file
      st=("Vector %s=(%11.4e, %11.4e, %11.4e)"%(name, self.x, self.y, self.z)
    )
      fileptr.write(st)
```

Listing 3: "A class to handle cartesian vectors with print and write_file functions."

Calling the print function:

```
a.print_contents("a")
b.print_contents("b")
```

produces the output:

```
Vector: Name=<a>, x= 1.0000e+00, y= 2.0000e+00, z= 3.0000e+00
Vector: Name=<b>, x= 4.0000e+00, y=-5.0000e+00, z= 6.0000e+00
```

Overloading operators

In Python (and C++) it is possible to add definitions to the standard operators ($+ \quad - \quad * \quad / \quad < \quad > \quad ==$ etc) to make them perform operations on a new class. This is already done in Python and C++ so that complex numbers can be handled with simple expressions like $(z + y)/(z * y)$.

Overload the addition operator '+' in the vector class definition:

```
class cartesian_vector:
   """ class to store a cartesian vector and provide basic operations
   """
   def __init__(self, p_x, p_y, p_z):
      self.x=p_x
      self.y=p_y
      self.z=p_z
   def print_contents(self, name):
      print("Vector: Name=<%s>, x=%11.4e, y=%11.4e, z=%11.4e"%(name, self.x,
    self.y, self.z))
   def write_file(self, fileptr, name):  # fileptr = already opened file
      st=("Vector %s=(%11.4e, %11.4e, %11.4e)"%(name, self.x, self.y, self.z)
   )
      fileptr.write(st)
   def __add__(self, other):  # self + other
      rtn=cartesian_vector(0.0,0.0,0.0)
      rtn.x=self.x+other.x
      rtn.y=self.y+other.y
      rtn.z=self.z+other.z
      return(rtn)
```

Using addition is then simple:

```
c=a+b
c.print_contents("c=a+b")
```

producing the output

```
Vector: Name=<c=a+b>, x= 5.0000e+00, y=-3.0000e+00, z= 9.0000e+00
```

Note that an instance of the class cartesian_vector called 'c' is automatically created for us by the Python interpreter. It recognises that the addition operator when called with cartesian_vector parameters either side of the '+' sign will return a cartesian_vector.

Adding operator definitions for subtraction ($-$), dot product as multiply ($*$) and cross product as divide ($/$) :

```
    def __sub__(self, other):  #self - other
        rtn=cartesian_vector(0.0,0.0,0.0)
        rtn.x=self.x-other.x
        rtn.y=self.y-other.y
        rtn.z=self.z-other.z
        return(rtn)
    def __mul__(self, other):  # self.other
        """ dot product is a magnitude """
        rtn=float(0.0)
        rtn=self.x*other.x+self.y*other.y+self.z*other.z
        return(rtn)
    def __truediv__(self, other):  # self X other
        """ use divide to represent cross product """
        rtn=cartesian_vector(0.0,0.0,0.0)
        rtn.x=self.y*other.z-self.z*other.y
        rtn.y=self.x*other.z-self.z*other.x
        rtn.z=self.x*other.y-self.y*other.x
        return(rtn)
```

The operators are simple to use:

```
d=a-b
d.print_contents("d=a-b")
dot=a*b
print("a.b=%g"%(dot))
e=a/b
e.print_contents("e=aXb")
f=b/a
f.print_contents("f=bXa")
```

producing the output

```
Vector: Name=<d=a-b>, x=-3.0000e+00, y= 7.0000e+00, z=-3.0000e+00
a.b=12
Vector: Name=<e=aXb>, x= 2.7000e+01, y=-6.0000e+00, z=-1.3000e+01
Vector: Name=<f=bXa>, x=-2.7000e+01, y= 6.0000e+00, z= 1.3000e+01
```

Further functionality is added with a 'norm' function to find the magnitude of a vector and a 'hat' function to find the unit vector of a vector:

```
    def norm(self):
      rtn=math.sqrt(self.x*self.x + self.y*self.y + self.z*self.z)
      return(rtn)
   def hat(self):
      wk=1.0/self.norm()  # calls the norm function of this class
      rtn=cartesian_vector(0.0,0.0,0.0)
      rtn.x=self.x*wk
      rtn.y=self.y*wk
      rtn.z=self.z*wk
      return(rtn)
```

Using these:

```
ma=a.norm()
mb=b.norm()
mc=c.norm()
md=d.norm()
print("||a||=%g, ||b||=%g, ||c||=%g, ||d||=%g"%(ma, mb, mc, md))
a_hat=a.hat()
b_hat=b.hat()
a_hat.print_contents("hat(a)")
b_hat.print_contents("hat(b)")
```

produces the output

```
||a||=4.3589, ||b||=9.38083, ||c||=13.6748, ||d||=5.19615
Vector: Name=<hat(a)>, x= 2.2942e-01, y= 4.5883e-01, z= 6.8825e-01
Vector: Name=<hat(b)>, x= 4.2640e-01, y=-5.3300e-01, z= 6.3960e-01
```

Finally

1. Classes allow the programmer to collect associated information of different types together in a logical manner. They also gather the functions needed to implement desired actions on the collection of data.
2. Once a class is defined, and debugged, it is quite easy to use it in a variety of programs. Software re-use is then much easier.
3. Object Orientated Programming using classes is used in many structured programming languages - Python, C++, Java, ...

5 Introducing comments and inclusion of tests

Comments are needed to explain how code operates. The audience for these include:

Author: To remind the author what the code does, or is supposed to do.

Non-authors: Other people who read the code, including co-authors, other users of the code, managers, customers...

Authors: To remind the author what the code does when returning to the code some weeks, months or years later.

Consider a function to square a value:

```
def square(x):
    """Return the square of x.
    :param x: the value to be squared, or anything with the * operator defined
    :type x: real, int, complex or other
    :return: the square of x
    :rtype: the same type as the paramerter
    """
    return x * x

xvalue=2.1
y = square(xvalue)
print("(%g)^2=%g"%(xvalue,y))
```

The param and return statements show what is to be passed to the function and what the function should return.

To test the function we call it with known values to see if the output is as expected:

```
xvalue=2.1
y = square(xvalue)
print("(%g)^2=%g"%(xvalue,y))

ok=( y==(xvalue*xvalue) )
if ok:
    print("Square worked")
else:
    print("Square failed")
```

Using doctest

Testing can also be included in the docstring commenting:

```
def square(x):
    """Return the square of x.
    :param x: the value to be squared, or anything with the * operator defined
    :type x: real, int, complex or other
    :return: the square of x
    :rtype: the same type as the paramerter

    >>> square(2)
    4
    >>> square(-2)
    4
    """
    return x * x

if __name__ == '__main__':
    import doctest
    doctest.testmod()

xvalue=2.1
y = square(xvalue)
print("(%g)^2=%g"%(xvalue,y))
```

The output expected when running the program from the command line is included in the comments, and the doctest facility in Python can recognise this and apply the tests. There is no error here so the output is

```
(2.1)^2=4.41
```

Introducing an error:

```
def square(x):
    """Return the square of x.
    :param x: the value to be squared, or anything with the * operator defined
    :type x: real, int, complex or other
    :return: the square of x
    :rtype: the same type as the paramerter

    >>> square(2)
    4
    >>> square(-2)
    4
    """
    return x * x +0.12345

if __name__ == '__main__':
    import doctest
    doctest.testmod()

xvalue=2.1
y = square(xvalue)
print("(%g)^2=%g"%(xvalue,y))
```

the output is:

```
**********************************************************************
File "H:/TEACH/Struct_Prog_EE20084/Python_Notes/Python_code/Testing/
    Doctest_03.py", line 15, in __main__.square
Failed example:
square(2)
Expected:
4
Got:
4.12345
**********************************************************************
File "H:/TEACH/Struct_Prog_EE20084/Python_Notes/Python_code/Testing/
    Doctest_03.py", line 17, in __main__.square
Failed example:
square(-2)
Expected:
4
Got:
4.12345
**********************************************************************
1 items had failures:
2 of   2 in __main__.square
***Test Failed*** 2 failures.
(2.1)^2=4.53345
```

i.e. a clear indication of what went wrong.

Considerations

1. In designing a program or class we can define the functions, parameters and return types, including comments and testing strategies. In the implementation stage the actual code is included and dubugged using the testing.
    - It may become apparent that other functions and/or tests are needed as the code is implemented.
2. Using the doctest facility provides checks on the code each time it is run. Helps with, or even is, debugging.
3. When testing many functions, and to make the test of many functions more readable/comprehensible, a more succinct output is likely to be needed. This can be created with your own 'Test_MyFunction' routines to test 'MyFunction'.

6 Libraries

Libraries contain pre-written code for developers to call to prevent re-inventing the wheel.

The library system selects just those parts of the pre-written code that are needed for the current application/program and includes them. This is similar to just selecting a few books form your local Library.

To make a library accessible to a Python program it needs to be imported with a statement like:

```
import math
x=4.0
y=math.cos(x)
```

In the C language the library needs to be included:

```
#include<math>
.
.

x=4.0
y=cos(x)
```

Examples of important libraries

Libraries that are almost always used include:

math: mathematical functions.

cmath: functions to handle mathematical functions using complex numbers.

random: functions to generate random numbers and sequences of random numbers.

statistics: functions to calculate various statistical distributions and evaluations.

numpy: create and use arrays, and various numerical methods including Fast Fourier Transforms (FFT)

scipy: wide variety of scientific functions

Math

This module provides access to the mathematical functions. These functions cannot be used with complex numbers; use the functions of the same name from the cmath module if you require support for complex numbers.

Except when explicitly noted otherwise, all return values are floats.

math.fabs(x): Return the absolute value of x.

math.ceil(x): Return the ceiling of x, the smallest integer greater than or equal to x.

math.floor(x): Return the floor of x, the largest integer less than or equal to x.

math.remainder(x, y): Return the IEEE 754-style remainder of x with respect to y.

math.trunc(x): Return the Real value x truncated to an Integral (usually an integer).

math.frexp(x): Return the mantissa and exponent of x as the pair (m, e). m is a float and e is an integer such that x == m * 2**e exactly.

math.comb(n, k): Return the number of ways to choose k items from n items without repetition and without order.

math.perm(n, k=None): Return the number of ways to choose k items from n items without repetition and with order.

math.factorial(x): Return x factorial as an integer.

math.fsum(iter): Return an accurate floating point sum of values in the iterable iter.

math.gcd(a, b): Return the greatest common divisor of the integers a and b.

math.isclose(a, b, *, rel_tol=1e-09, abs_tol=0.0): Return True if the values a and b are close to each other and False otherwise.

math.isfinite(x): Return True if x is neither an infinity nor a NaN, and False otherwise.

math.isinf(x): Return True if x is a positive or negative infinity, and False otherwise.

math.isnan(x): Return True if x is a NaN (not a number), and False otherwise.

math.isqrt(n): Return the integer square root of the nonnegative integer n.

math.ldexp(x, i): Return x * (2**i).

math.modf(x): Return the fractional and integer parts of x.

math.exp(x): Return e raised to the power x, where e = 2.718281 is the base of natural logarithms.

math.expm1(x): Return e raised to the power x, minus 1.

math.log(x): Return the natural logarithm of x (to base e).

math.log(x, B): Return the natural logarithm of x (to base B).

math.log1p(x): Return the natural logarithm of 1+x (base e).

math.log2(x): Return the base 2 logarithm of x.

math.log10(x): Return the base 10 logarithm of x.

math.pow(x, y): Return x raised to the power y.

math.sqrt(x) : Return the square root of x.

math.acos(x) : Return the arc cosine of x, in radians.

math.asin(x) : Return the arc sine of x, in radians.

math.atan(x): Return the arc tangent of x, in radians.

math.atan2(y, x): Return atan(y / x), in radians. The result is between $-\pi$ and π.

math.cos(x): Return the cosine of x radians.

math.sin(x): Return the sine of x radians.

math.tan(x): Return the tangent of x radians.

math.degrees(x): Convert angle x from radians to degrees.

math.radians(x): Convert angle x from degrees to radians.

math.acosh(x): Return the inverse hyperbolic cosine of x.

math.asinh(x): Return the inverse hyperbolic sine of x.

math.atanh(x): Return the inverse hyperbolic tangent of x.

math.cosh(x): Return the hyperbolic cosine of x.

math.sinh(x): Return the hyperbolic sine of x.

math.tanh(x): Return the hyperbolic tangent of x.

math.erf(x): Return the error function at x.

math.erfc(x): Return the complementary error function at x, 1.0 - erf(x).

math.gamma(x): Return the Gamma function at x.

math.lgamma(x): Return the natural logarithm of the absolute value of the Gamma function at x.

math.dist(p, q): Return the Euclidean distance between two points p and q, each given as a sequence (or iterable) of coordinates.

math.pi : The mathematical constant $\pi = 3.141592\ldots$, to available precision.

math.e : The mathematical constant $e = 2.718281\ldots$, to available precision.

math.tau : The mathematical constant $\tau = 2 * \pi$, to available precision.

math.inf : A floating-point positive infinity. In a 32-bit float sets the exponent bits all to 1 and fraction bits to 0: s111 1111 1000 0000 0000 0000 0000 0000

math.nan : A floating-point “not a number” (NaN) value. In a 32-bit float sets the exponent bits all to 1: s111 1111 1xxx xxxx xxxx xxxx xxxx xxxx

Cmath

Setting a complex number is straightforward:

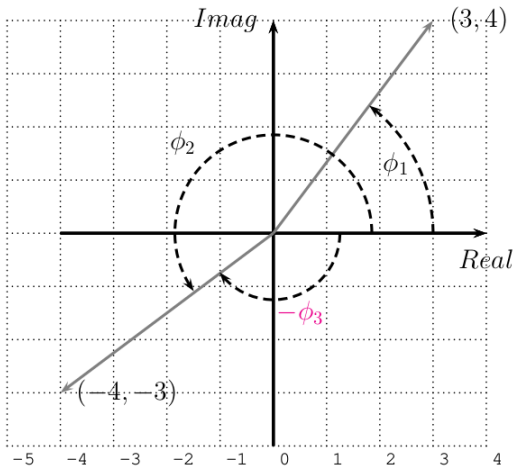
```
import cmath

re=1.5
im=-7.3
z=complex(re,im) # sets real part to re and imaginary part to im
print("Z=%g + j %g"%(z.real,z.imag))
```

where the real and imag operators split the complex value into separate floats for printing.

The modulus (absolute value) of a complex number x can be computed using the built-in abs() function. There is no separate cmath module function for this operation.

Polar coordinates give an alternative way to represent a complex number where the complex number z is defined by a modulus $r = \sqrt{Real^2 + Imag^2}$ and the phase angle ϕ.



The phase

$$\phi = \arctan 2(Imag, Real)$$

is the counterclockwise angle, measured in radians, from the positive real axis to the line segment that joins the origin to z.

cmath.phase(z): Return the phase of x (or the argument of z), as a float. phase(z) is equivalent to math.atan2(z.imag, z.real). The result lies in the range $[-\pi : \pi]$

cmath.polar(z) : Return the representation of z in polar coordinates. Returns a pair (r, phi) where r is the modulus of z and phi is the phase of z. polar(z) is equivalent to (abs(z), phase(z)).

cmath.rect(r, phi) : Return the complex number x with polar coordinates r and phi. Equivalent to r * (math.cos(phi) + math.sin(phi)*1j).

cmath.isinf(x): Return True if the real or the imaginary part of x is positive or negative infinity.

cmath.isnan(x): Return True if the real or imaginary part of x is not a number (NaN).

cmath.exp(z): Return the exponential value $e^z = \exp(z)$.

cmath.log(z): Returns the natural (base e) logarithm of z.

cmath.log(z, B): Returns the logarithm of x to the base B.

cmath.log10(z): Return the base-10 logarithm of z.

cmath.sqrt(z): Return the square root of x.

cmath.acos(z): Return the arc cosine of z.

cmath.asin(z): Return the arc sine of z.

cmath.atan(z): Return the arc tangent of z.

cmath.cos(z): Return the cosine of z.

cmath.sin(z): Return the sine of z.

cmath.tan(z): Return the tangent of z.

cmath.acosh(z): Return the inverse hyperbolic cosine of z.

cmath.asinh(z): Return the inverse hyperbolic sine of z.

cmath.atanh(z): Return the inverse hyperbolic tangent of z.

cmath.cosh(z): Return the hyperbolic cosine of z.

cmath.sinh(z): Return the hyperbolic sine of z.

cmath.tanh(z): Return the hyperbolic tangent of z.

Random

random.seed(a=None, version=2) : Initialize the random number generator. If a is omitted or None, the current system time is used.

random.getrandbits(k): Returns a Python integer with k random bits.

random.randrange(start, stop[, step):] Return a randomly selected element from range(start, stop, step).

random.randint(a, b): Return a random integer N such that a <= N <= b.

random.random(): Return the next random floating point number in the range [0.0, 1.0).

random.uniform(a, b): Return a random floating point number N such that a <= N <= b for a <= b and b <= N <= a for b < a.

random.betavariate(alpha, beta): Beta distribution. Conditions on the parameters are alpha > 0 and beta > 0. Returned values range between 0 and 1.

random.expovariate(lambd): Exponential distribution. lambd is 1.0 divided by the desired mean. It should be nonzero. (The parameter would be called "lambda", but that is a reserved word in Python.) Returned values range from 0 to positive infinity if lambd is positive, and from negative infinity to 0 if lambd is negative.

random.gammavariate(alpha, beta): Gamma distribution. (Not the gamma function!) Conditions on the parameters are alpha > 0 and beta > 0.

random.gauss(mu, sigma): Gaussian distribution. mu is the mean, and sigma is the standard deviation.

random.lognormvariate(mu, sigma): Log normal distribution. A variable x has a log-normal distribution if log(x) is normally distributed. The distribution of log(x) has a mean mu and standard deviation sigma. mu can have any value, and sigma must be greater than zero.

random.normalvariate(mu, sigma): Normal distribution. mu is the mean, and sigma is the standard deviation.

random.vonmisesvariate(mu, kappa): mu is the mean angle, expressed in radians between 0 and 2π, and kappa is the concentration parameter, which must be greater than or equal to zero. If kappa is equal to zero, this distribution reduces to a uniform random angle over the range 0 to 2π.

random.paretovariate(alpha): Pareto distribution. alpha is the shape parameter.

random.weibullvariate(alpha, beta): Weibull distribution. alpha is the scale parameter and beta is the shape parameter.

Statistics

statistics.mean(data): Return the sample arithmetic mean of data which can be a sequence or iterable.

statistics.fmean(data): Convert data to floats and compute the arithmetic mean.

statistics.geometric_mean(data): Convert data to floats and compute the geometric mean.

statistics.harmonic_mean(data): Return the harmonic mean of data, a sequence or iterable of real-valued numbers.

statistics.median(data): Return the median (middle value) of numeric data, using the common“mean of middle two” method.

statistics.mode(data): Return the single most common data point from discrete or nominal data.

statistics.multimode(data): Return a list of the most frequently occurring values in the order they were first encountered in the data.

statistics.pstdev(data, mu=None): Return the population standard deviation (the square root of the population variance).

statistics.pvariance(data, mu=None): Return the population variance of data, a non-empty sequence or iterable of real-valued numbers.

statistics.stdev(data, xbar=None): Return the sample standard deviation (the square root of the sample variance).

statistics.quantiles(data, *, n=4, method='exclusive'): Divide data into n continuous intervals with equal probability. Returns a list of n - 1 cut points separating the intervals.

Numpy

A major feature in Numpy is the ability to create arrays of values (rather than lists).

```
import numpy as np  # abbreviates calls to np.

a = np.array([1, 2, 3])   # Create a rank 1 array
print(type(a))            # Prints "<class 'numpy.ndarray'>"
print(a.shape)            # Prints "(3,)"
print(a[0], a[1], a[2])   # Prints "1 2 3"
a[0] = 5                  # Change an element of the array
print(a)                  # Prints "[5, 2, 3]"

b = np.array([[1,2,3],[4,5,6]])    # Create a rank 2 array
print(b.shape)                     # Prints "(2, 3)"
print(b[0, 0], b[0, 1], b[1, 0])   # Prints "1 2 4"
```

```
import numpy as np  # abbreviates calls to np.

np.zeros((3,4))         # Create a 3 by 4 array, all values zero
np.ones((2,4),int)      # Create a 2 by 4 array of integers
np.zeros(25, complex)  # Create a 25 element array of complex, all zero
d = np.arange(10,25,5) # Create an array of evenly spaced values (step value)
np.linspace(0,2,9)      # Create an array of 9 evenly spaced values from 0 to
    2
e = np.full((2,2),7.0) # Create a constant array
f = np.eye(2)           # Create a 2 by 2 identity matrix
np.random.random((2,2)) # Create a 2 by 2 array with random values
np.empty((3,2))         # Create an empty array
```

Array Maths

```
tot = np.add(b,a)         # Addition of arrays
diff = np.subtract(a,b)   # Subtraction of arrays
prod = np.multiply(a,b)   # Multiplication of arrays
ratio= np.divide(a,b)     # Division of arrays
eb = np.exp(b)          # Exponentiation - element by element
rt = np.sqrt(b)         # Square root - element by element
SS = np.sin(a)          # Sine - element by element
CC = np.cos(b)          # Cosine - element by element
ln = np.log(a)          # Natural Logarithm - element by element
dotp = a.dot(ab)        # Dot product a.ab - row by column
S = a.sum()               # Array-wise sum of all elements
Mi = a.min()              # Array-wise minimum value
Ma = b.max(axis=0)      # Maximum value of an array row[0]
Cs = b.cumsum(axis=1)   # Cumulative sum of the elements in row[1]
Mn = a.mean()           # mean of the array
Me = np.median(arr)       # Median of the array
Co = np.corrcoef(arr)     # Correlation coefficient of the array
St = np.std(arr)          # Standard deviation of the array

print("a=",a)
print("b=",b)
print("a*b=",prod)    # element by element - not maths!
print("a/b=",ratio)   # element by element - not maths!
```

Listing 4: "Array functions in Numpy"

Fourier transforms - numpy.fft

```
# a is an array
# optional parameters are: n is the number of points,
# axis selects a row of data in a
# if norm="ortho" the FFT and IFFT are scaled by 1/sqrt(n) otherwise
# norm="None" (default) FFT is not scaled and IFFT is scaled by 1/n
fft(a[, n, axis, norm])     # Compute the one-dimensional FFT.
ifft(a[, n, axis, norm])    # Compute the one-dimensional inverse FFT.
fft2(a[, s, axes, norm])    # Compute the 2-dimensional FFT
ifft2(a[, s, axes, norm])   # Compute the 2-dimensional IFFT.
fftn(a[, s, axes, norm])    # Compute the N-dimensional FFT.
ifftn(a[, s, axes, norm])   # Compute the N-dimensional IFFT.
```

For purely real input data:

```
rfft(a[, n, axis, norm])   # Compute the one-dimensional FFT.
irfft(a[, n, axis, norm])  # Compute the inverse of the n-point FFT.
rfft2(a[, s, axes, norm])  # Compute the 2-dimensional FFT.
irfft2(a[, s, axes, norm]) # Compute the 2-dimensional IFFT.
rfftn(a[, s, axes, norm])  # Compute the N-dimensional FFT.
irfftn(a[, s, axes, norm]) # Compute the inverse of the N-dimensional FFT.
```

For data with Hermitian symmetry (a real spectrum):

```
hfft(a[, n, axis, norm])   # Compute the FFT of the signal.
ihfft(a[, n, axis, norm])  # Compute the inverse FFT of the signal.
```

Helper routines:

```
fftfreq(n[, d])     # Return the FFT sample frequencies.
rfftfreq(n[, d])  #  Return the FFT sample frequencies (for usage with rfft,
    irfft).
fftshift(x[, axes])    # Shift the zero-frequency component to the center of
    the spectrum.
ifftshift(x[, axes])  # The inverse of fftshift.
```

Scipy

Scipy contains a very wide selection of items for 'scientific' calculations.

There is some overlap with the older standard Python functions and those in numpy.

A simple import statement makes all of this available:

```
import scipy
```

constants: most of the physical constants and conversion factors

cluster: hierarchical clustering, vector quantization, K-means

fft: Discrete Fourier Transform algorithms, as per numpy

fftpack: Legacy interface for Discrete Fourier Transforms

integrate: numerical integration routines such as quadrature, trapezoidal, simpson, ODE integration

interpolate: interpolation tools for 1-D, multivariate, spline methods

io: data input and output for MatLab(.mat), IDL, Fortran, Netcdf, .wav and other files

lib: Python wrappers to external libraries

linalg: linear algebra routines for matrix operations, solving sets of equations, eivenvalue problems, decomposing matrices, etc.

misc: miscellaneous utilities (e.g. image reading/writing)

ndimage: various functions for multi-dimensional image processing

optimize: optimization algorithms including linear programming

signal: signal processing tools - convolution, filtering, windows, standard waveforms, linear systems, wavelets, spectral analysis

sparse: sparse matrix and related algorithms

spatial: KD-trees, nearest neighbors, distance functions

special: special functions: Airy, Elliptic, Bessel, Struve, statistical, Gamma, Error, Legendre, Orthogonal polynomial, Kelvin....

stats: statistical functions: continuous and discrete distributions, statistical, correlation, statistical tests...

weave: tool for writing C/C++ code as Python multiline strings

Graphics

Graphics was not part of the original definition of many languages, such as C or C++, but is now an important feature for programs. When graphical output is generated by a C/C++ program additional libraries have to have been used. There are many of them and they all operate in slightly different ways.

The Windows or X-windows systems will produce the usual 'windows' that are presented to us on a PC/laptop, but they will not draw a data plot for us. For this the additional Libraries have to supply functions for us to call to set up the axes, plot lines, add legends....

In Python there are several ways of generating graphs, matplotlib being a quite popular scheme that mirrors graphics in MatLab.

```
import matplotlib.pyplot as plt
import numpy as np
import math
M=12
npts=int(pow(2.0,M))     #sets n=2^M, good for FFT calcs
L=0.5e-6              # an inductance
C=6.0e-9              # a capacitance
R=3.3                 # a resistance

fmin=10.0             #minimum frequency for calculations
fmax=10.0e+6          #maximum frequency
df=(fmax-fmin)/(npts-1)      #uniform step between frequencies
freqs=np.zeros(npts,float)   #array to hold frequencies
Z=np.zeros(npts,complex)     #array to hold complex impedances
Y=np.zeros(npts,complex)     #array to hole complex admittances
for ii in range(0,npts):
    freqs[ii]=fmin+df*ii          #frequency
    omega=2.0*math.pi*freqs[ii] #circular frequency
    Z[ii]=R+complex(0.0,omega*L)+1.0/complex(0.0,omega*C)
    Y[ii]=1.0/Z[ii]               #calcs of a series LCR circuit
```

Listing 5: "Matplotlib example - forming the data"

Now create the plot of the impedance data as a function of frequency:

```
figno=1
plt.figure(figno) # an instance of figure for the graphics system
plt.plot(freqs, Z.real,color='k', linestyle='-', label='Re(Z)')
plt.plot(freqs, Z.imag,color='g', linestyle='-.', label='Im(Z)')
plt.legend()                  # adds the legend to the plot
plt.ylabel('Z (Ohms)')        # axis title
plt.xlabel('Freq (Hz)')
#plt.xlim([0.10, 20.0])
plt.ylim([-100.0, 50.0])      # sets min & max limits on y-axis
plt.title('Series LCR resonance')  # sets the main title on the plot
plt.grid(True)                #adds a grid on the plot
plt.savefig("LCR_impedance.eps",format="eps")  #outputs to file
#plt.show()          # output to 'screen'
```

Listing 6: "Matplotlib example - impedance plot"

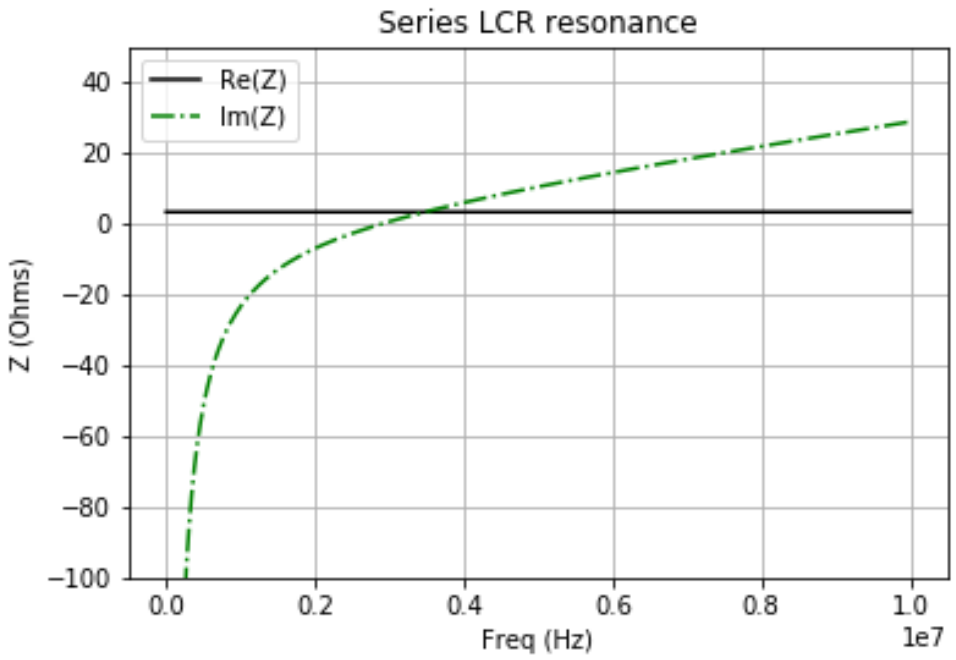


Figure 6.1: Impedance plot generated by Matplotlib.

Then create a plot of the admittance data as a function of frequency:

```
figno+=1
plt.figure(figno)            # a second instance of figure
plt.plot(freqs, Y.real,color='r', linestyle=':', label='Re(Y)')
plt.plot(freqs, Y.imag,color='b', linestyle='—', label='Im(Y)')
plt.legend()
plt.ylabel('Y (Siemens)')
plt.xlabel('Freq (Hz)')
#plt.xlim([0.10, 20.0])
#plt.ylim([−0.5, 7.5])
plt.title('Series LCR resonance')
plt.grid(True)                  #adds a grid on the plot
plt.savefig("LCR_admittance.eps",format="eps")
#plt.show()
```

Listing 7: "Matplotlib example - admittance plot"

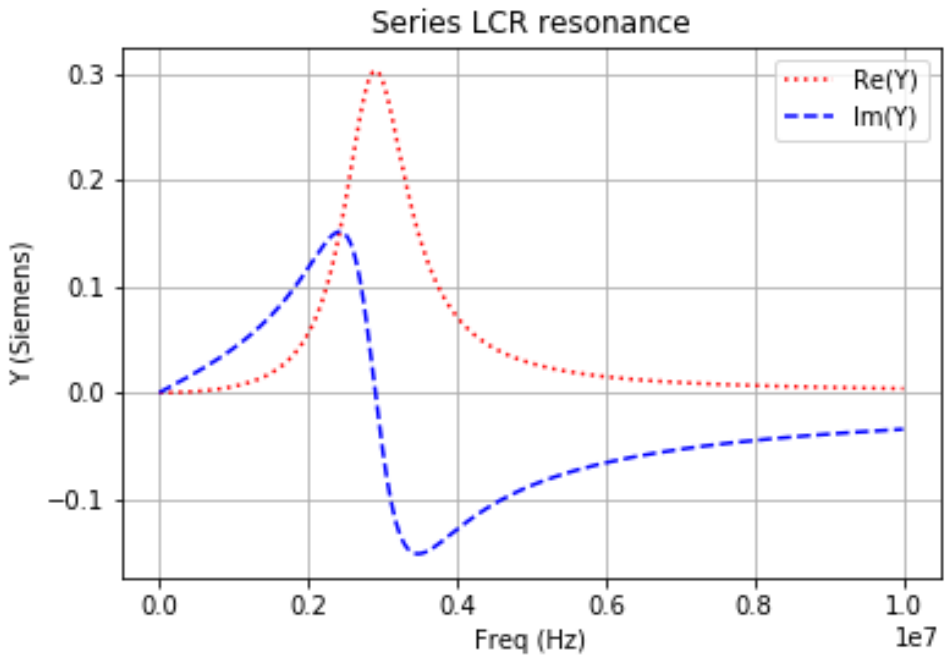


Figure 6.2: Admittance plot generated by Matplotlib.

Finally

Summary

1. Libraries provide a reliable way of packaging code together in logical groups.
2. Libraries are especially useful for code that is used in multiple projects and programs.
3. Libraries reduce the amount of time spent redeveloping and recompiling code.
4. Libraries are universally used when distributing tested code across many institutions or programmers.

7 Advanced Input/Output

Command line interpretation

A simple command line example looked at previously allowed for two filenames to be given on the command line:

```
python MyProgram.py input.dat output.dat
```

The Python program interprets the input arguments that are stored in a list called argv:

```
import sys
print "This is the name of the script: ", sys.argv[0]
print "Number of arguments: ", len(sys.argv)
print "The arguments are: " , str(sys.argv)
input_filename=argv[1]
output_filename=argv[2]
```

Which produces the output:

```
This is the name of the script:  MyProgram.py
Number of arguments in:  3
The arguments are:  ['Myprogram.py', 'input.dat', 'output.dat']
```

More sophisticated command lines, as are commonly used in a lot of applications, have additional parameters that may or may not be present. For example the command 'python –help' shows the usage of the python command and its many optional parameters:

```
usage: python [option] ... [-c cmd | -m mod | file | -] [arg] ...
Options and arguments (and corresponding environment variables):
-b     : issue warnings about str(bytes_instance), str(bytearray_instance)
       and comparing bytes/bytearray with str. (-bb: issue errors)
-B     : don't write .pyc files on import; also PYTHONDONTWRITEBYTECODE=x
-c cmd : program passed in as string (terminates option list)
-d     : debug output from parser; also PYTHONDEBUG=x
-E     : ignore PYTHON* environment variables (such as PYTHONPATH)
-h     : print this help message and exit (also --help)
-i     : inspect interactively after running script; forces a prompt even
       if stdin does not appear to be a terminal; also PYTHONINSPECT=x
-I     : isolate Python from the user's environment (implies -E and -s)
-m mod : run library module as a script (terminates option list)
-O     : remove assert and __debug__-dependent statements; add .opt-1 before
       .pyc extension; also PYTHONOPTIMIZE=x
```

The 'getopt' system was developed through the 1970's for Unix systems.

Getopt

In the getopt system options, such as '-n' or '-o filename', are included allowing parameters to be set from the command line and switching various options on or off in a program. The desired options are set using a very simple string. For the Python program to take notice of these options it has to interrogate the command line sent in sys.argv to see what has been set.

Getopt example

In this example valid options are:

-v Makes the program more verbose, producing additional output as it runs

-h or –help Makes the program display help information screens. Generally information on how to run the program.

-n 3 Sets a (numeric?) parameter to 3

-i input.file or –input input.file Sets the input file to be "input.file"

-o output.file or –output output.file Sets the output file to be "output.file"

The getopt option string to set this up is just "ho:vi:n:"

- The -h and -v options have no parameters and are specified by single characters in the getopt option string.
- The -n -i and -o options have parameters and this is specified by a letter followed by a colon in the getopt option string.
- The order in the getopt option string has no effect. The order the options are executed in is set within the Python program that receives the options.

Example program

```
import sys
import getopt

def usage():
    """ Function to display how to run the program """
    print("\n\nTo run GetOpt_01.py the command line should be:")
    print("python GetOpt_01.py [options]")
    print("Valid options are -i -o -v -n -h")

try:
    opts, args = getopt.getopt(sys.argv[1:],"ho:vi:n:",["help","output=","
    input="])
except getopt.GetoptError as err:    # print help information and exit:
    print("\n\n\tError")
    print(err) # will print something like "option -a not recognized"
    usage()
    sys.exit(2)
```

The try-except clause checks that there are valid options passed to the program. If there are not it calls the usage() function to print out a help message on how to run the program. If the options are valid the program then interprets these as follows:

```
output_file = None      # need to initialise variables that may not be set
input_file = None
verbose = False
N_str=""
for o, a in opts:
    if o == "-v":
        verbose = True
    elif o in ("-h", "--help"):  # option string has --help as well as -h
        usage()
        sys.exit()
    elif o in ("-o", "--output"):# option string has --output= and o:
        output_file = a    # a contains the output file name in this case
    elif o in ("-i", "--input"): # option string has --input= and i:
          input_file = a     # a contains the input file name in this case
    elif o in ("-n"):
        N_str = a     # a contains the value for N in this case
    else:
        assert False, "unhandled option"

if verbose:
    print("input=<%s>\noutput=<%s>\nN=<%s>"%(input_file,output_file,N_str))
```

Using Comma Separated Variables (CSV) files

Comma Separated Variables (CSV) files are widely used, where variables are written in flat text (ASCII or UTF-8) with commas in between the variables. We could just set up output formats such as

```
fp=open("My_data.csv")
st="X,Y,Z"     # column headings
fp.write(st)
st="%10.2e,%f,%g",%(x,y,z))     # data
fp.write(st)
```

and read this back in using the split(',') function to split the input text at each comma.

This potentially has a problem if a field includes a comma. If we need to output a quoted string "Pultney St, Bath" the split(',') function would try to split up the quoted string.

Python has a module for handling this and many other eventualities, called csv.

The CSV file records information in a simple table format - a set of columns. Each column may well have a column heading and corresponding values in the following rows. The various column headings and the data values in the later rows are all separated by commas.

```
import csv
output_file = open('file.csv', 'wt', newline='')

data_1=['Id','Age','Salary',"Surname, Forename",'Grade']
data_2=[215, 32, 35052, "Newton, Issac",'A']
data_3=[384, 52, 58934, "Einstein, Albert",'C']
data_4=[12, 70, 23756, "Meldrew, Victor",'B']
data=[data_1,data_2,data_3,data_4]
csv.writer(output_file).writerows(data)

data_1=['Id','Cost']
data_2=[215, 25052]
data_3=[384, 68934]
data_4=[12,  21756]
data=[data_1,data_2,data_3,data_4]
csv.writer(output_file).writerows(data)
output_file.close()

input_file = open('file.csv', 'rt')
ip_data=csv.reader(input_file)
for row in ip_data:
   print("Row=",row)
input_file.close()
```

The CSV file contains:

```
Id,Age,Salary,"Surname, Forename",Grade
215,32,35052,"Newton, Issac",A
384,52,58934,"Einstein, Albert",C
12,70,23756,"Meldrew, Victor",B
Id,Cost
215,25052
384,68934
12,21756
```

The output generated to the screen is:

```
Row= ['Id', 'Age', 'Salary', 'Surname, Forename', 'Grade']
Row= ['215', '32', '35052', 'Newton, Issac', 'A']
Row= ['384', '52', '58934', 'Einstein, Albert', 'C']
Row= ['12', '70', '23756', 'Meldrew, Victor', 'B']
Row= ['Id', 'Cost']
Row= ['215', '25052']
Row= ['384', '68934']
Row= ['12', '21756']
```

The data written out and read in is a list of strings. To recover values which can be used in computations the input data has to be converted to integers or floats as needed.

Binary Input/Output

Binary IO is much faster than ASCII/UTF-8 or text IO. Binary IO literally puts the binary representation of an integer as 4 bytes to a file. Likewise a 32-bit float is written as 4 bytes which can hold about 8 significant digits, while a 64-bit float is written as 8 bytes which can hold about 16 significant digits. The UTF-8 representation of the 4-byte float that can easily be read on a computer might be '-1.345678e+07' which is 13 characters, so 13 bytes need to be written. The UTF-8 representation of the 8-byte float that can easily be read on a computer might be '-1.34567890123456e+007' which is 23 characters, so 23 bytes need to be written. Not only are more bytes being written in the UTF-8 case, which takes more time, but there must also be a conversion process from the 8-byte float to the output string:

```
print("%22.16e",%(x))
```

This takes some time to execute while it generates the 23 character long string '-1.34567890123456e+007'.

ASCII IO produces files that are easily read using Notepad/Word/any text editor. Binary IO puts the binary representation of data directly into a

file, which is not easily read.

Binary Input/Output for integers and floats

The int type in Python has two functions for the conversion between integers and bytes, int.to_bytes and int.from_bytes. For converting floats the module struct needs to be imported.

```
 f=open("Test.bin","wb")
ii=42
ibin = ii.to_bytes(4,byteorder='big') # convert to 4 bytes
f.write(ibin)
 f.close
 f=open("Test.bin","rb")
ribin=f.read(4)                       # read 4 bytes
ival=int.from_bytes(ibin,byteorder='big')
```

Here ii is converted to a 4-byte binary value with the most significant byte being the first byte. If the most significant byte is to be the last byte-order needs to be set to 'little'.

For converting floats the module struct needs to be imported and use its pack and unpack functions.

```
import struct
 f=open("Test.bin","wb")
a=float(12.5)
abin = struct.pack('d',a)    # 'd' creates 8 byte output, 'f' creates 4
f.write(abin)
f.close
f=open("Test.bin","rb")
rabin=f.read(8)              # read 8 bytes
wk=struct.unpack('d',rabin)  # returns an array
aval=float(wk[0])            # find first element
```

A simple example, and a simple problem

Binary IO does not know what has been written, there are just a sequence of bytes in the output file. When reading in it is easy to misinterpret the data.

Suppose 4-byte integer I and then two 8-byte floats Xa and Xb are written out to the binary file in the order $I\ Xa\ Xb$. There are 20 bytes written to the file. In the following representation each bit of the integer Ia is denoted as 'i', while the floats have a sign bit 's', an 11-bit exponent 'e' and a 52-bit mantissa 'm'. The binary file contents are:

Write	Byte 1	Byte 2	Byte 3	Byte 4	Read
Ia	iiii iiii	iiii iiii	iiii iiii	iiii iiii	Ya
Xa	seee eeee	eeee mmmm	mmmm mmmm	mmmm mmmm	
	mmmm mmmm	mmmm mmmm	mmmm mmmm	mmmm mmmm	Yb
Xb	seee eeee	eeee mmmm	mmmm mmmm	mmmm mmmm	
	mmmm mmmm	mmmm mmmm	mmmm mmmm	mmmm mmmm	J

Reading this in as $I\ Xa\ Xb$ the data is recovered accurately from the 20 bytes read in. However, if the file is read in as $Ya\ Yb\ J$, i.e. two floats and an integer, there are 20 bytes read in. This is exactly the right length so there is no simple indication of error - but the data is *corrupted!* Ya is now made up of the 4 bytes of I and the first half of Xa. Yb is made of the last half of Xa and the first half of Xb. J is made from the second half of Xb.

This is illustrated in the Python program Basic_Binary_02.py on the EE20084 Moodle site.

A simple array example, and a simple problem

It is easy to output arrays as binary values using the 'array' module available in Python and its tofile() function:

```
from array import *
output_file = open('file.bin', 'wb')

divline="-"*80  # create a line of 80 minus signs to break up output
print(divline)
A = array('d', range(10))  # array of 8-byte double precision floats
B = array('f', range(5))   # array of 4-byte single precision floats
A.tofile(output_file)
B.tofile(output_file)
output_file.close()
```

Here two arrays, one with 8-byte (52 bit mantissa, 11 bit exponent) floats and one with 4-byte (23 bit mantissa, 8 bit exponent) floats are created and output to a file. Reading this in requires the file to be re-opened and then use the fromfile() function.

```
input_file = open('file.bin', 'rb')
ip_A = array('d') # array to hold 8-byte double precision floats
ip_B=array('f')   # array to hold 4-byte single precision floats
ip_A.fromfile(input_file, 10)  #read in in order data was written
ip_B.fromfile(input_file, 5)
input_file.close()
print("   A=",A)
print("IP A=",ip_A)
A_eq = A == ip_A    # test if data read in same as originally written
if A_eq:
   print("A written and read OK")
else:
   print("Error: A not written and read accurately")
print("   B=",B)
print("IP B=",ip_B)
B_eq = B == ip_B    # test if data read in same as originally written
if B_eq:
   print("B written and read OK")
else:
   print("Error: B not written and read accurately")
```

Reading in works here, producing the output

```
   A= array('d', [0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0])
IP A= array('d', [0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0])
A written and read OK
   B= array('f', [0.0, 1.0, 2.0, 3.0, 4.0])
IP B= array('f', [0.0, 1.0, 2.0, 3.0, 4.0])
B written and read OK
```

If however the order of the read is reversed, the correct number of bytes are read in, but 4-byte and 8-byte data is confused and corrupted.

```
input_file = open('file.bin', 'rb')
ip_A = array('d')
ip_B=array('f')
ip_B.fromfile(input_file, 5) #read in in reverse order
ip_A.fromfile(input_file, 10)
input_file.close()
```

The output is:

```
   A= array('d', [0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0])
IP A= array('d', [5.304989477e-315, 5.307579804e-315, 5.31017013e-315,
    5.311465295e-315, 5.31276046e-315, 5.31405562e-315, 5.315350785e-315,
    5.315998367e-315, 2.000000473111868, 512.0001225471497])
Error: A not written and read accurately
   B= array('f', [0.0, 1.0, 2.0, 3.0, 4.0])
IP B= array('f', [0.0, 0.0, 0.0, 1.875, 0.0])
Error: B not written and read accurately
```

The error has to be detected by the Python program - which if you don't know what the data was in the file is impossible.

To alleviate this problem we could include extra binary data to indicate whether an integer or a float is being stored. This only needs one extra bit

before each value, but binary IO is byte based so an extra byte should be output. In that byte, or bytes, we could also store the variable name. There is no defined standard in Python for doing this, until more sophisticated IO schemes are used, such as in numpy.

Pickle

The pickle module allows for simple binary I/O of instances of a class.

```
import pickle
class MyClass:
    def __init__(self,p_name,p_address,p_wage,p_age):
        self.name = p_name
        self.address = p_address
        self.wage = p_wage
        self.age = p_age
    def display(self):
        print("Name:%s\nAddress:%s\nWage=%.2f, Age=%d"%(self.name,self.address,
     self.wage,self.age))

my = MyClass("Mr. E","The Mansion, Oil Drum Lane", 45872.0, 52)
pickle.dump(my, open("myobject.bin", "wb"))
my.display()
me = pickle.load(open("myobject.bin", "rb"))
me.display()
```

Examining the binary file myobject.bin in Notepad shows that the name and address fields are stored as viewable text strings, i.e. as UTF-8, while the float and int values are stored as binary. Space and IO time savings are mainly encountered when storing many floats and integers.

IO Features in numpy

savetxt() and loadtxt()

Text output and input for a single numpy array is facilitated by the savetxt() and loadtxt() functions.

```
import numpy as np  # abbreviates calls to np.
a=np.array([[1.0, 2, 3.5], [4, 5.2, 6]])
b=np.array([1, 2])

np.savetxt('my_file.txt', a)  # writes a single numpy array to a file
AA=np.loadtxt('my_file.txt')  # reads from file into AA
print("AA=",AA)
diff=a-AA       # should be zeros if IO worked OK
print("diff=",diff)
```

The file my_file.txt contains:

```
1.0000000000000000e+00 2.0000000000000000e+00 3.5000000000000000e+00
4.0000000000000000e+00 5.2000000000000178e+00 6.0000000000000000e+00
```

while the output to the screen is

```
AA= [[1.  2.  3.5]
[4.  5.2 6. ]]
diff= [[0. 0. 0.]
```

The savetxt and loadtxt functions have worked, even though there is some corruption to the data in the file. The corruption is after the 17th significant figure, which loadtxt cannot interpret as a float only contains 16 digits.

save() and load()

Binary output and input for a single numpy array is facilitated by the save() and load() functions.

```
np.save('my_file.npy', a)  # writes single array to my_file.npy
AAy=np.load('my_file.npy')
print("AAy=")
print(AAy)
```

The output file is binary, so largely unreadable using standard editors/viewers. Opening with Notepad shows that it is a NUMPY file of some sort which is not using Fortran ordering and has a shape of 2 rows by 3 columns.

```
1  "NUMPYSOHNULvNUL{'descr': '<f8', 'fortran_order': False, 'shape': (2, 3),
   }
2  NULNULNULNULNULNULð?NULNULNULNULNULNULNUL@NULNULNULNULNULNULBSS@NULNULNULNULNULNULDLE
   @ÍÌÌÌÌÌÌDC4@NULNULNULNULNULNULCAN@
```

Figure 7.1: my_file.npy interpreted as UTF-8 text.

This is however successfully read back in by the load() function.

savez() and load()

The savez() function stores multiple numpy arrays:

```
np.savez('my_Zfile.npz', a=a, b=b) # writes multiple numpy arrays to file
data = np.load('my_Zfile.npz')
AAz=data['a']       #extracts 'a' from data
print("AAz=", AAz)
BBz=data['b']       #extracts 'b' from data
print("BBz=",BBz)
#ZZ=data['z']   # returns an error as z is not in the file
```

The output file is binary, and is largely unreadable. Inspection with Notepad shows an element a.npy holds something with shape (2,3) - 2 rows and 3 columns, while b.npy holds something with shape(2,) - 2 elements.


```
PKETXEOTDC4NULNULNULNULNULNULNUL!NULÍ3¢Û°NULNULNUL°NULNULNULENONULDC4NULa.npySOHNUL
DLENUL°NULNULNULNULNULNULNUL°NULNULNULNULNULNULNUL“NUMPYSOHNULvNUL{'descr': '<f8',
'fortran_order': False, 'shape': (2, 3),
}
NULNULNULNULNULNULð?NULNULNULNULNULNULNUL@NULNULNULNULNULNULFF@NULNULNULNULNULNULDLE
@ÍÌÌÌÌÌDC4@NULNULNULNULNULNULCAN@PKETXEOTDC4NULNULNULNULNULNULNUL!NULVTùª-^NULNULNUL^
NULNULNULENONULDC4NULb.npySOHNULDLENUL^NULNULNULNULNULNULNUL^NULNULNULNULNULNULNUL“NU
MPYSOHNULvNUL{'descr': '<i4', 'fortran_order': False, 'shape': (2,),
}
SOHNULNULNULSTXNULNULNULPKSOHSTXDC4NULDC4NULNULNULNULNULNULNUL!NULÍ3¢Û°NULNULNUL°NUL
NULNULENONULNULNULNULNULNULNULNULNULNULNUL€SOHNULNULNULNULa.npyPKSOHSTXDC4NULDC4NUL
NULNULNULNULNULNUL!NULVTùª-^NULNULNUL^NULNULNULENONULNULNULNULNULNULNULNULNULNULNUL€
SOHçNULNULNULb.npyPKENOACKNULNULNULNULSTXNULSTXNULfNULNULNUL¦SOHNULNULNULNUL
```


Figure 7.2: my_Zfile.npz interpreted as UTF-8 text.

The values are read back in OK using load(). In this case all of the values in the file are read into the variable ’data’. This has to be interrogated for each variable, so long as the original variable name is known.

Note that there is some overhead added into the file in addition to the numpy array values. The text string
"’descr’: ’<f8’, ’fortran_order’: False, ’shape’: (2,3), "
is added twice and the filename "a.npy" and b.npy" are also added twice. With large arrays there is a clear saving in filesize using this binary format, but for shorter arrays the overhead may be more than the basic saving from using the binary 8-byte format.

Finally

1. The widely used command line options are easily included, allowing simple control of what a program does from the command line.
2. CSV files are quite easy to generate and interpret in Python. Files are easily readable in Excel or any other spreadsheet program.
3. Binary I/O is far faster and generates far smaller files - but it is not easily readable. Binary I/O can be misinterpreted quite easily.
4. Numpy provides a Binary I/O scheme that annotates the data in the binary file, so misinterpretation is much less likely.

   Files are however larger and it takes longer to process, so some of the advantage of using Binary I/O is lost.

5. Testing I/O is very important. If the I/O is not known to be working the program is operating in RIRO mode - *Rubbish-In Rubbish-Out*.

8 Testing software

1. Testing is needed to convince *ourselves*, and probably the *customer*, that the software is working.
2. Testing is an implicit part of developing and debugging software.
   - We naturally have to test the code as it is being developed.
   - We test relatively a small section of code or a function to debug that part before moving onto other parts/functions.
3. Software testing strategies are very similar to testing strategies for electrical/electronic circuits.

An example receiver chain might contain various components.

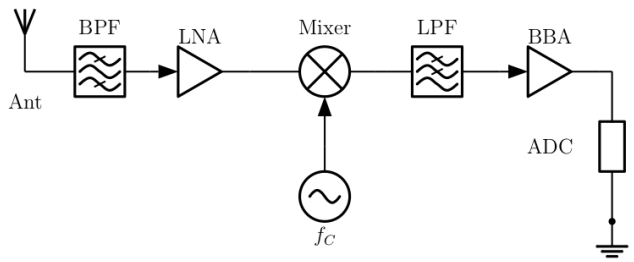


Figure 8.1: Example receiver chain.

Do the filters (BPF and LPF) have the right passband and stopband loss?

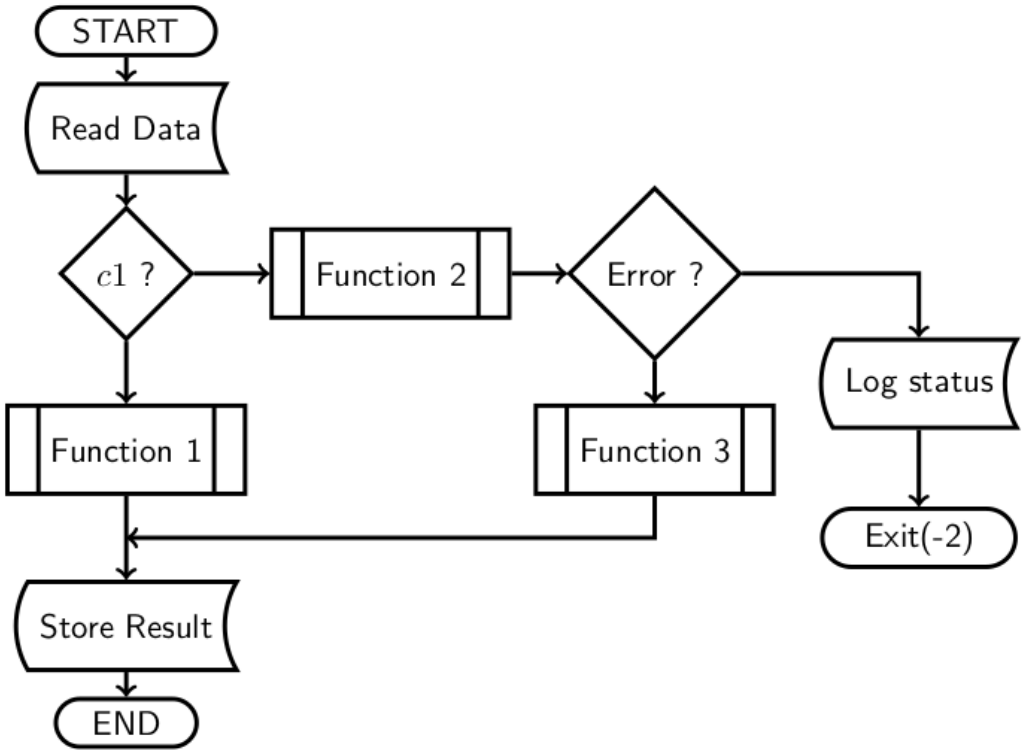
Do the amplifiers (LNA, BBA) have the designed gain and bandwidth?

Does the mixer have the designed conversion loss/-gain?

Testing the individual components reveals where an error is. Testing the whole receiver just shows that there is an error *somewhere* in the whole receiver. Software is similarly split into sub-tasks or functions. These are combined to form the complete program/application.

It is not uncommon for there to be several ways that the execution of a program is terminated.

In the following examples the Functions themselves have internal structure, possibly calling other functions. Each Function may be described by its own flow chart.



Testing

Testing should identify that the function is performing the desired actions/operations.

When designing functions we need to make a 'statement' about what is the proper action/operation of a function. Then we can test against this.

Test the individual functions.

Testing is what we do to debug functions and programs.

Record the tests - then you know how far through debugging a complete program you are, and you can show your management and/or customer.

If you haven't tested it, it doesn't work!

If it is not tested the customer is unlikely to pay for it - and your management may not be very pleased.

Black and White Box testing

In many cases the testing is categorised into black box and white box testing:

Black-box: This treats the system as a "black-box", so it doesn't explicitly use knowledge of the internal structure. Black-box test design is usually described as focusing on testing functional requirements. Synonyms for black-box include: behavioral, functional, opaque-box, and closed-box.

In the circuit example this would be like applying an input signal to the Antenna and testing that the output at the ADC is what is expected.

White-box: This allows us to peek inside the "box", and it focuses specifically on using internal knowledge of the software to guide the selection of test data. Should test for *all eventualities*. Synonyms for white-box include: structural, glass-box and clear-box. In software engineering this may well include Path testing, Loop testing and Condition testing.

In the circuit example this looks at all AC and DC paths within a component asking if voltages and currents are as expected throughout that component.

Grey-box: A combination of black-box tests with some knowledge of the expected behavior of internal routing paths in the software but not the detailed knowledge of the code needed for white-box testing.

Requirements-based testing could be called "black box" because it makes sure that all the customer requirements have been verified. Code-based testing is often called "white box" because it makes sure that all the code (the statements, paths, or decisions) is exercised.

Black and White Tests

Imagine you're testing an electronics system. Its housed in a black box with lights, switches, and dials on the outside. You must test it without opening it up, and you can't see beyond its surface. You have to see if it works just by flipping switches (inputs) and seeing what happens to the

lights and dials (outputs). This is black box testing. Black box software testing is doing the same thing, but with software. There is still a question as to how you define the boundary of the box and what kind of access the "blackness" is blocking.

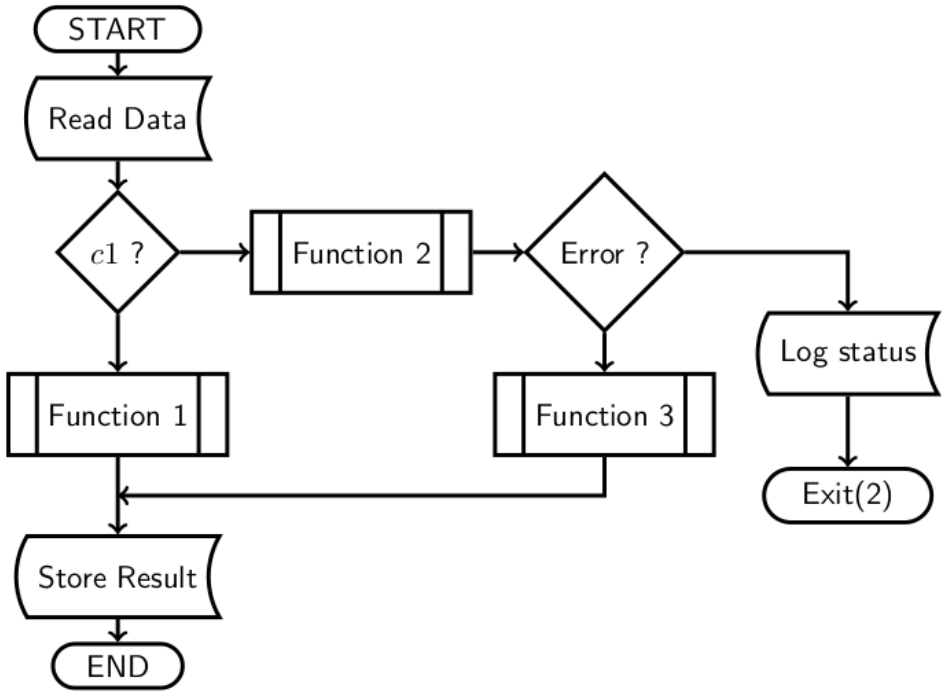
An opposite test approach would be to open up the electronics system, see how the circuits are wired, apply probes internally and maybe even disassemble parts of it. By analogy, this is called white box testing, but already the metaphor is faulty. It is just as hard to see inside a white box as a black one. In the interests of logic, some people prefer the term "clear box" testing, but even a clear box keeps you from probing the internals of the system. It might be even more logical to call it "no box" testing. The metaphor operates beyond logic, but "white box" testing is the most commonly used term.

Think of a drinks vending machine. A black box test would involve inserting the money into the machine and verifying that a drink drops out and that correct change is given. A white box test might involve opening the back panel on the machine and manually triggering the switch that drops the drink.

White box testing is much more time consuming and therefore *expensive* than black box testing. It requires the source code to be produced before the tests can be planned and is much more laborious in the determination of suitable input data and the determination if the software is or is not correct. The advice given is to *start* test planning with a *black* box test approach as soon as the specification is available. *White* box planning can start once the *design* phase has broken the code up into modules, functions or units and we know how the software is *internally structured*.

Flow chart example

The data read into the program needs to be chosen to route through all paths within the program.



The data must be able to set the $c1$ condition to be true or false and the $error$ condition to be true or false. The exit(2) return value needs to be trapped by an IDE, such as Spyder or Visual Studio, or by the Operating System in a command script.

The data also needs to trigger all paths within Function 1, 2 and 3. This could be rather cumbersome, so a better strategy is to test Functions 1, 2 and 3 separately with data that exercises all their internal paths.

Trapping a return value in a bash command script is quite simple:

```
# bash example catching exit value from myprog
#
myprog
return_value=$?  # most recent return value from executing a program is  $?
echo "Return value is " $return_value
if [ $return_value -eq 0]
then
echo "myprog returns 0, so ran OK"
else
echo "myprog failed, and returned " $return_value
fi
```

Coverage

This term is essentially used to describe how thorough the testing has been - have all aspects of the software's performance been covered by the testing strategy. A *greater* coverage means that *more aspects* have been tested.

When White-box testing all paths and outcomes should be observed by covering all possibilities.

Once the tests have been performed, and passed, does this guarantee that the software is now 'perfect'?

No

It means that you have not yet come up with the test that reveals other bugs or imperfections.

Numeric cases

The Functions may also have internal conditional variables, and so the input data needs to be devised to check for all such cases when white-box testing.

For example if a function evaluates $sin(x)/x$ the $x = 0$ situation needs to be considered. If $x = 0$ then the return value is likely to be **'Inf'**, or more generally **'NaN'**. If the function asks if $(x == 0.0)$ then it should set the return value to be 1.0. This should then be tested for by evaluating with the $x = 0$ case in the testing sequence along with x values that return known finite values.

Question: In the case of the $sin(x)/x$ function what values of x should we avoid in order to get meaningful tests?

Other numerical issues will arise, for example:

1. Dividing by zero - **Inf** should be returned.
2. Logarithm of zero - **Inf** should be returned.

For real number 'math' functions:

1. Logarithm of a negative value - **NaN** should be returned.

2. Arcsin or arccos of a number outside of the range $[-1 : 1]$ - **NaN** should be returned.

3. Square root of a negative value - **NaN** should be returned.

If using cmath these will return a valid complex number.

Other numerical evaluations may also produce problems. Some (older?) functions will return values that are clearly impossible but these impossible values then have to be trapped for in the calling program. Returning **Inf** or **Nan** is a clearer indication of the problem.

File opening example

Consider the function in EE20084_functions_01.py for opening a file:

```
import sys
import os

def My_open_file(filename, mode):
    """ Function that attempts to open a file in a specified mode
        On failing to open the file the current directory is printed to
        the screen showing the directory that the Python program can see
        :param filename – the name of the file ot be opened
        :type string
        :param mode – the file opening mode, r, w, t, b, a
        :type string
        :return file pointer
        :rtype pointer
    """
    try:
        fp=open( filename, mode)
    except FileNotFoundError:
        print('File <%s> not found'%(filename))
        current_location=os.getcwd()
        # gets the directory where the program is executing
        print("executing program in directory: "+current_location)
        sys.exit(2)  # exits the program, returning 2 to the operating system
    return(fp)
```

Listing 8: "Function implementation in EE20084_functions_01.py"

To test this the various possibilities all need to be examined:

Open an existing file: Use a file that in known to exist in the current directory. The function should run and then let the test program read in some *known* values and test for these values.

Try to open a non-existent file: The test program should call the 'dir' command and list out the contents of the current directory. Exit value

should be 2 which can by trapped in an IDE or captured if the program is executed from a command shell.

Numeric tests

Tests may often make a comparison between a reference/known value and what the current software has determined. Comparisons can then be numerical:

```
import numpy as np  # abbreviates calls to np.

f_reference=4.0
f_calculated = square(2.0)

f_diff=f_reference - f_calculated
if ((f_diff==0.0)):
   print("square() function is OK")
else:
   print("Error in square() function, expected f=%g, got f=%g"%(f_reference,
    f_calculated))
```

Listing 9: "Testing a function compared to a known value"

When dealing with integers the difference should be zero. The float difference may not identically be zero due to the finite precision of floating point calculations. However we could expect the difference to be within a small tolerance.

Using numpy.isclose() allows for a equality within an absolute or relative fractional accuracy

```
absolute_tolerance=1.0e-6
relative_tolerance=1.0e-6
square_ok=np.isclose(f_reference, f_calculated, atol=absolute_tolerance, rtol
    =relative_tolerance)
if (square_ok):
   print("square() function is within abs tol=%g, rel tol=%g"%(
    absolute_tolerance,relative_tolerance))
else:
   print("Error in square() function, expected f=%g, got f=%g"%(f_reference,
    f_calculated))
```

Listing 10: "Testing a function using numpy.isclose()"

Displaying differences to the programmer leaves the test as a 'visual' inspection, where the programmer may still make mistakes. Converting to the boolean value 'square_ok' lets the program act on the pass/fail output. Actions can include outputting the pass/fail summary messages to a 'log' or 'error' file to provide a record of the result of the test.

Many pass/fail tests can then be combined to give a pass/fail evaluation

of a function or program.

Passing the overall pass/fail evaluation of a set of functions or a program via the return value of the testing program can then communicate success/failure to an external script.

This is what happens in many systems when software is installed. Messages output to the user and log files indicate whether all tests have been passed or not. It is still down to the user to view these, which is still a 'visual' test, *and act on it*.

Automatically testing file similarity

Using system commands such as fc or diff, or using python functions, can provide ways of testing to see if an output file is the same as a 'perfect' file or not. Suppose there is a perfect version of an output file and a copy of it. Comparing these will produce no difference. If there is a corrupted version that is a copy of the perfect file but with some corruption there is a difference. The **fc** command can be run from within a Python script using the **os** module to make comparisons. The Python **cmp()** function from the **filecmp** module can also be used to identify if there are differences between two files.

```
import sys
import os
import filecmp
os.system('fc a_Test_Circuit_1_perfect.out a_Test_Circuit_1.out >Perfect.log
    2>&1')
os.system('fc a_Test_Circuit_1_perfect.out a_Test_Circuit_1_corrupted.out >
    Corrupted.log 2>&1')
```

*Listing 11: "Running **fc** file comparisons through Python"*

The Windows/DOS command **fc** compares two files. In the case where the files are the same the output in the log file, "Perfect.log" is

```
Comparing files a_Test_Circuit_1_perfect.out and A_TEST_CIRCUIT_1.OUT
FC: no differences encountered
```

*Listing 12: "Output from **fc** file comparison with identical files"*

This clearly shows the files are the same.

In the case where the files differ the output in the log file, "Corrupt.log" contains

```
Comparing files a_Test_Circuit_1_perfect.out and A_TEST_CIRCUIT_1_CORRUPTED.
    OUT
***** a_Test_Circuit_1_perfect.out
5.029e+01+j 0.000e+00  2.566e-05+j 0.000e+00  1.306e+02+j 0.000e+00  1.213e
    -02+j 0.000e+00  2.112e-02+j 0.000e+00
 5.556e+06  3.615e+00+j 0.000e+00  4.387e-02+j 0.000e+00  2.769e-02+j 0.000e
    +00  5.849e-04+j 0.000e+00  1.001e-01+j 0.000e+00
5.029e+01+j 0.000e+00  2.566e-05+j 0.000e+00  1.306e+02+j 0.000e+00  1.213e
    -02+j 0.000e+00  2.112e-02+j 0.000e+00
 6.667e+06  3.615e+00+j 0.000e+00  4.387e-02+j 0.000e+00  2.769e-02+j 0.000e
    +00  5.849e-04+j 0.000e+00  1.001e-01+j 0.000e+00
***** A_TEST_CIRCUIT_1_CORRUPTED.OUT
5.029e+01+j 0.000e+00  2.566e-05+j 0.000e+00  1.306e+02+j 0.000e+00  1.213e
    -02+j 0.000e+00  2.112e-02+j 0.000e+00
 5.556e+06  3.615e+00+j 0.000e+00 <BREXIT>  4.387e-02+j 0.000e+00  2.769e-02+
    j 0.000e+00  5.849e-04+j 0.000e+00  1.001e-01+j 0.
000e+00  5.029e+01+j 0.000e+00  2.566e-05+j 0.000e+00  1.306e+02+j 0.000e+00
     1.213e-02+j 0.000e+00  2.112e-02+j 0.000e+00
 6.667e+06  3.615e+00+j 0.000e+00  4.387e-02+j 0.000e+00  2.769e-02+j 0.000e
    +00  5.849e-04+j 0.000e+00  1.001e-01+j 0.000e+00
*****
```

*Listing 13: "Output from **fc** file comparison with differing files"*

In this case the **fc** command outputs the line before a discrepancy, the line where there is a discrepancy and the following two lines. This shows the line where there is a difference and its surrounding 'context' in the files being compared.

Using the Python filecmp package

Using the Python **filecmp** package the file comparison function **cmp()** returns True if the files are the same and False if they differ.

```
correct_files=0
files_examined=0
op=filecmp.cmp('a_Test_Circuit_1_perfect.out', 'a_Test_Circuit_1.out')
print("For files %s and %s same=%r"%('a_Test_Circuit_1_perfect.out', '
    a_Test_Circuit_1.out', op))
files_examined+=1
if (op):
    correct_files+=1
op=filecmp.cmp('a_Test_Circuit_1_perfect.out', 'a_Test_Circuit_1_corrupted.
    out')
print("For files %s and %s same=%r"%('a_Test_Circuit_1_perfect.out', '
    a_Test_Circuit_1_corrupted.out', op))
files_examined+=1
if (op):
    correct_files+=1
incorrect_files=files_examined-correct_files
```

```
print("%d files tested, %d correct, %d incorrect"%(files_examined,
    correct_files,incorrect_files))
```

*Listing 14: "Running **filecmp.cmp()** file comparisons in Python"*

Here the number of correct and incorrect files is counted and the output is:

```
For files a_Test_Circuit_1_perfect.out and a_Test_Circuit_1.out same=True
For files a_Test_Circuit_1_perfect.out and a_Test_Circuit_1_corrupted.out
    same=False
2 files tested, 1 correct, 1 incorrect
```

Note

This is the basis on which the basic functional performance of the Python Coursework submission is made in the Autotester. Output from your Python code is tested for equality against all of the output files given on the Moodle site - and some other more complicated/testing examples.

Conclusions

1. Structured programming is not about opening an editor and starting to type - *that is coding!*
2. Planning and organising the programme and data is vital.
3. So is knowing what the programme is supposed to do. Programs/-functions need to be tested to show that they are doing what they are supposed to do.
4. Write the testing strategy for a function when the function is first outlined. *(additional tests may be added later)*. How else are you going to debug the code?
5. Proper commenting and documenting is there to help define the functions and programme from the outset.
6. 'Automatic' tests on working functions should be repeated as later functions are added to see if they still work. The later functions may have corrupted the functions developed earlier.
7. 'Visual' tests are rather weak. Output logged to a file is a better test and provides a *record of testing*.

9 Developing your own modules

It often makes sense to group together related functions and structures into a module. Doing this also has the benefit that several programs can use these functions, and if a function is updated all of the programs will pick up the new version. The standard Python packages such as math or numpy are all written in this fashion. Likewise the standard C/C++ libraries and other available libraries are also written in this fashion.

Thinking about some of the functions already discussed in the course these can be grouped together as "EE20084_functions_01.py" and "Vectors.py".

1. The definitions of the functions are put into the file EE20084_functions_01.py
2. The Python compiler then compiles this module into a compiled python file EE20084_functions_01.pyc.
3. Similarly Vectors.py is compiled to Vectors.pyc.
4. The main programs that refer to these module must import the module. The Python compiler then links together what is needed from the module, the main program and any other modules (math, numpy...) that are imported and executes the program.

In this example EE20084_functions_01.py contains a function to open a file while catching if it opened correctly:

```
import sys
import os

def My_open_file(filename, mode):
```

Listing 15: "Function to open a file with error check"

and a function to search a string for a variable "Npts" in a string like "Npts=75" and extract, and return, the integer value 75:

```
def find_int(string_to_search,name,noisy):
```

Listing 16: "Function to find a parameter in a string and extract its integer"

Similarly the class developed to handle cartesian vectors is defined in "Vectors.py" this contains many functions, including printing vectors and adding them:

```
import math
class cartesian_vector:
    """ class to store a cartesian vector and provide basic operations
    """
    def __init__(self, p_x, p_y, p_z):
        self.x=p_x
        self.y=p_y
        self.z=p_z
    def print_contents(self, name):
        print("Vector: Name=<%s>, x=%11.4e, y=%11.4e, z=%11.4e"%(name, self.x
    , self.y, self.z))
    def write_file(self, fileptr, name):  # fileptr = already opened file
        st=("Vector %s=(%11.4e, %11.4e, %11.4e)"%(name, self.x, self.y, self.
    z))
        fileptr.write(st)
    def __add__(self, other):  # self + other
        rtn=cartesian_vector(0.0,0.0,0.0)
        rtn.x=self.x+other.x
        rtn.y=self.y+other.y
        rtn.z=self.z+other.z
        return(rtn)
```

Listing 17: "Some of the Vector class for cartesian vectors"

If these files are in the same directory as the main python program to use them is a simple matter of importing them and using them:

```
import Vectors as V
import EE20084_functions_01

op_file= EE20084_functions_01.My_open_file('test_Vector_EE20084.out','wt')
a=V.cartesian_vector(1.0,2.0,3.0)
b=V.cartesian_vector(4.0,-5.0,6.0)
a.write_file(op_file,'a')
b.write_file(op_file,'b')
op_file.close()
a.print_contents("a")
b.print_contents("b")
```

Listing 18: "Importing and using modules"

The use of the full module name, such as "EE20084_functions_01" can be rather cumbersome. Abbreviating this to "EE84" is simple using

```
from My_Module import EE20084_functions_01 as EE84     # abbreviates to EE84
```

Listing 19: "Abbreviating a module name"

Grouping modules

In some cases it may be useful to group modules together for form a larger module. A grouping is very simply created. For example in "My_Module.py" two modules are simply imported:

```
import Vectors
import EE20084_functions_01
```

Listing 20: "Grouping modules into My_Module.py"

Using My_Module is very simple:

```
import math
#import Vectors as v                    # abbreviates Vectors to v
from My_Module import Vectors as v     # abbreviates Vectors to v
from My_Module import EE20084_functions_01 as EE84     # abbreviates to EE84
#import My_Module.Vectors as v          # does not work, not a package

# create two instances, each with their own values
a=v.cartesian_vector(1.0,2.0,3.0)
b=v.cartesian_vector(4.0,-5.0,6.0)
op_file=EE84.My_open_file('Test_Module.out','wt')
a.write_file(op_file,'a')
b.write_file(op_file,'b')
op_file.close()
a.print_contents("a")
b.print_contents("b")
```

Listing 21: "Using the modules grouped in My_Module.py"

In this case "Vectors.py" is in the same directory, and could have been imported directly, but this example illustrates simple grouping. The 'from' statement makes the Python compiler pull Vectors out of My_Module, and then introduce the abbreviation **v** for Vectors.

To use the construct "My_Module.Vectors" instead the code needs to be collected as a package rather than modules. This kind of grouping is done in the various Python packages that we can use, such as numpy, scipy,..... Recall that the IFFT calculation is called using "numpy.fft.ifft()", i.e. the ifft function in the fft module in numpy.

Conclusions

1. Modules and Packages or Libraries provide extra functionality over and above the basic languages.
2. Some are hardware specific and are essential.
3. Some are just programmed algorithms to relieve you of the effort of implementing the algorithm. *[don't redesign the wheel]*
4. Design re-use is assisted by keeping code in modules and libraries for use in many programs.
5. The modules are not recompiled every time the main program is compiled - this can save a lot of re-compilation of already compiled modules.
6. Modules and Packages or Libraries assist and promote using a 'Structured Programming' strategy. *[Greatly helps debugging/development by breaking things into smaller, testable, functions]*

10 Program Execution Time and Speed up

- For short programs the execution time may not be too much of an issue.
- For longer more complex programs the user may find the time the program takes to run is an issue.
- Here the execution time of a program is considered, along with techniques to speed up the execution of a program.
- The processor on a computer executes machine code. This code does basic memory shifts, stack control, addition, subtraction, multiplication, division - see:

  https://en.wikipedia.org/wiki/X86_instruction_listings#Original_8086/8088_ins
- Execution speed is governed by how well our Python, C, C++, Java, MatLab text is converted into machine code.

Compiler Approach

Source code: This is typed into a file by the user and is the text version of the program. This could be in a variety of languages such as C, C++, Fortran, Pascal, Algol, Java.... For example this might be in a file MyProg.c

```
#include<stdio>
#include<math>
.
.
x=4.0
y=cos(x)
printf("x=%g, y=%g\n",x,y)
```

Compiler: This takes our source code, checks the syntax, and if that is correct compiles our code into object code. The object code in MyProg.obj is closer to machine code than the original source code.

Where the source code makes reference to a pre-defined function like cosine or printf the object code makes a call to something called cos and printf. However at this point there is no machine code for this as our source code does not define how to calculate the cosine of an angle or print something to the screen.

Linker: This links the object code with libraries for math, strings, standard input/output, numeric methods, graphics. These libraries contain the code for the predefined functions such as cos in math and printf in stdio.

If all of the functions called in MyProg.obj are found in the libraries then all 'references' have been resolved. The linker then generates and links the executable code needed for MyProg.obj and those parts needed from the libraries.

Executable code: The executable code might be in the file MyProg.exe, and typing MyProg.exe at the command line then instructs the operating system to run our code.

Code Optimisation

Many compilers have an optimisation option, which optimises the code generated in some ways.

In traditional, perhaps Unix, based compilers there has always been a drive towards optimising for speed of operation. Runtime of a well written program can be halved. Poorly written programs can have even larger reductions in their execution time.

In the GCC compiler code can be optimised with -O, -O1, -O2, -O3 options. Further optimisation to utilise features of the specific processor on the machine can be made with -march=native. The code can then not be used on machines with a different processor to the one that the code was compiled on.

In DOS/Windows compilers there was more of a drive towards creating small executables. The original DOS system may only have had 64 kBytes of memory so smaller executables left more space for data and other programs.

Interpreter Approach

Typically used for Basic, MatLab and Python. Python, and MatLab with extra steps, can have a hybrid approach using both interpreter and compiler.

The interpreter takes the source code, checks the syntax, and if that is correct interprets the source code into executable code and executes the code. This includes resolving calls to libraries where necessary.

While it seems a simpler approach to the compiler/linker scheme the entire code has to be interpreted every time a change is made to the source code. Large parts of the code may not have changed but it is being re-interpreted every time - slowing down the whole process.

The interpreter system can be better when developing short programs, but as program complexity increases the long re-interpretation times can become very apparent to the developer. When the code is finalised the source code needs to be distributed to users for them to re-interpret every time it is run.

Hybrid approach - interpret with compilation

Python can use a hybrid approach where debugged code can be compiled and stored in an intermediate bytecode form so it does not have to be recompiled every time changes are made elsewhere in the program. In many ways this is what the libraries do - they contain the compiled executable version of things like the cosine function.

Modules

A Python module contains functions. When the python command is run on these modules the source code is syntax checked, and if that is correct the functions are compiled into bytecode. The bytecode is closer to executable code, but not as close as the object code that the compiler approach generates.

Python scripts that call the modules are themselves compiled, but the modules are not as they have already been compiled. The bytecodes from the python script and the modules are combined and turned into executable code for the processor to execute. This speeds up the process as the modules are not recompiled.

Timing Program Execution

Determining execution time for a program is easily included by importing a module called time and then calling its time() function.

```
import time

start_time=time.time()

# rest of the python program

stop_time=time.time()

print("Calculations took %s sec"%(round( (stop_time-start_time),3) ) )
```

Good programming practice for faster execution

There are many potential ways of speeding up code. Some relate to organisation of the source code, and there are examples in the following notes. Others relate to changing the flow of data through the machine code on the processor.

Remove repeated assignments

Suppose a program needs to determine the function $\sin(r)/r$ at positions (x_i, y_j) where the radial distance is $r = \sqrt{(x_i^2 + y_j^2)}$. The function evaluations are to be stored in a 2-dimensional array.

Numpy has useful array functions, so use is made of these.

The first part of the program defines parameters and the data arrays.

```
8  import numpy as np
9  import math
10 import time
11 start_time=time.time()
12 nxpts=251
13 nypts=301
14 xstart=-5.0
15 xfinish=5.0
16 ystart=-6.0
17 yfinish=6.0
18 dx=(xfinish-xstart)/nxpts
19 dy=(yfinish-ystart)/nypts
20 x=np.zeros(nxpts,float)
21 y=np.zeros(nypts,float)
22 data=np.zeros((nxpts,nypts),float)
```

Listing 22: "Program to evaluate $\sin(r)/r$ - data and array setup."

The program then performs the calculations

```
23 for i in range(0,nxpts):
24     for j in range(0,nypts):
25         x[i]=xstart+i*dx
26         y[j]=ystart+j*dy
27         data[i,j]=math.sin(math.sqrt(x[i]*x[i]+y[j]*y[j]))/math.sqrt(x[i]*x[i
       ]+y[j]*y[j])
28 calc_time=time.time()
29 for i in range(0,nxpts):
30     for j in range(0,nypts):
31         print("D[%d,%d]=%10g "%(i,j,data[i,j]))
32     print("\n")
33 print_time=time.time()
34 print("Calculations took %s sec"%(round( (calc_time-start_time),3) ) )
35 print("Print took %s sec"%(round( (print_time-calc_time),3) ) )
```

Listing 23: "Program to evaluate $\sin(r)/r$*"*

The calculations take 0.234 seconds and the print to screen takes 4.984 seconds. *Output to a screen is time consuming.*

Problems with the program

Line 25: This is evaluated for all values of the counter j, and it does not need to be as it is independent of j.

Line 27: The radius is evaluated twice - only needs to be done once.

Repeated indexing of the x[] and y[] array elements - only needs to be done once.

A better solution

```
23 for ix in range(0,nxpts):
24     x[ix]=xstart+ix*dx       # set x[] once
25 for iy in range(0,nypts):
26     y[iy]=ystart+iy*dy()     # set y[] once
27 for ix in range(0,nxpts):
28     xv=x[ix]                 # reference x[] array once per loop
29     for iy in range(0,nypts):
30         yv=y[iy]             # reference y[] array once per loop
31         r=math.sqrt(xv*xv+yv*yv)    #calculate r once
32         data[ix,iy]=math.sin(r)/r
33 calc_time=time.time()
34 for ix in range(0,nxpts):
35     for iy in range(0,nypts):
36         print("D[%d,%d]=%10g "%(ix,iy,data[ix,iy]))
37     print("\n")
38 print_time=time.time()
```

```
39 print("Calculations took %s sec"%(round( (calc_time-start_time),3) ) )
40 print("Print took %s sec"%(round( (print_time-calc_time),3) ) )
```

Listing 24: "Program to evaluate $\sin(r)/r$ - less repeated evaluations"

The calculations now take 0.077 seconds and the print to screen takes 4.987 seconds. *Output to a screen is time consuming.*

The improvements all stem from making the program only do calculations once and storing values in an additional variable.

Statements like $x[i]$ involve the processor finding the value at index i and returning it. Once this is done storing the value in xv and using that is far more efficient than repeatedly working out $x[i]$.

Arrays, lists, dictionaries, sets all suffer from this. If the value needed from these is needed more than once assign it into another variable and use that variable instead.

Screen Output

Outputting data to the screen, and drawing things onto the screen in a Graphical User Interface is time consuming.

Make sure all output to a screen/GUI is actually necessary.

Output to a file is rather faster, particularly if it is binary output - see the Advanced IO section of the notes.

'Monitoring' output showing progress of the program, which is mainly useful to observe if something goes wrong, is better written to a .log file. If things go wrong the user needs to look at the .log file.

Localise data

To use data the processor needs to fetch it from memory, perform calculations, and then put the answer somewhere in memory.

A processor typically had a quite slow hard disk, faster RAM and much faster Cache memory on the processor chip.

If data is localised within functions or even just declared within a short code block it is likely to be 'colocated' in the computer memory. If the data can all be stored in the processor cache this gives least time for the

memory operations. If the data is larger than the cache the processor may have to fetch from and put to the slower RAM, or the very slow hard disk.

Global data is likely to sit a long way from local data, so may well be in RAM when the program needs to reference it.

Speeding up Python

Python in its basic form is a relatively slow interpreter.

Partial compilation in modules helps reduce compilation time.

To speed up execution time the interpretation parts need to be replaced with compiled part.

Best compiled executable code comes from C, C++, Fortran.... Hence to speed up a program the best use of precompiled C, C++, Fortran... code is needed.

The import libraries like math, cmath, numpy are all pre-compiled, debugged, and generally optimised. Execution is fast, and the programmer does not need to write and debug new code.

Cython

Cython provides a way of translating Python into C/C++. This can then be compiled to produce the quickest code.

The Cython scheme requires the Python compiler/interpreter *and* a C/C++ compiler.

The C/C++ compiler should be the *same one* that Python was created with in order to avoid binary incompatibility errors.

In Unix/Linux systems when Python is installed, like most other software, it is compiled from source code using the system's host C/C++ compiler. Hence all code is naturally binary compatible.

In DOS/Windows system there is no host C/C++ compiler, you need to install a C/C++ compiler yourself. The original definition of a C/C++ compiler is relatively loose, there is not even a definition as to whether the LSB or MSB of an integer is stored first. Hence binary compatibility is very unlikely between C/C++ compilers.

Example - python text

A simple example taken from PythonTutorials. The Cython system operates on files with the extension .pyx rather than .py

```
#example_original.py
def test(x):
    y = 0
    for i in range(x):
        y += i
    return y
```

Listing 25: "Initial python file example_original.py "

```
#example_original.pyx
def test(x):
    y = 0
    for i in range(x):
        y += i
    return y
```

Listing 26: "Initial python file converted for cython use: example_original.pyx"

To run cython the python script setup.py is needed

```
from distutils.core import setup
from Cython.Build import cythonize

setup(ext_modules = cythonize('example_original.pyx'))
```

Listing 27: "setup.py operating on example_original.pyx"

This is run at the command line:

```
python setup.py build_ext —inplace
```

Example - cython text

Cython produces C/C++ code, where variables have an explicit type. Variables have to be given a type:

```
cdef int x,y,z       # integers
cdef char *s           # string of characters
cdef float x = 5.2     # single precision float
cdef double x = 40.5   # double precision float
```

The modified Cython script is

```
#example_original.py
def test(x):
    y = 0
    for i in range(x):
        y += i
    return y
```

Listing 28: "Initial python file example_original.py "

```
#example_cython.pyx
def test(int x):
    cdef int y = 0
    cdef int i
    for i in range(x):
        y += i
    return y
```

Listing 29: "Initial python file converted for cython use: example_cython.pyx"

To run cython the python script setup.py is needed

```
from distutils.core import setup
from Cython.Build import cythonize

#setup(ext_modules = cythonize('example_cython.pyx', extra_compile_args=["-O3"]))
setup(ext_modules = cythonize('example_cython.pyx'))
```

Listing 30: "setup_cython.py operating on example_original.pyx"

This is run at the command line:

```
python setup_cython.py build_ext --inplace
```

Example - full cython text

Return types in C/C++ code include pointers and the full conversion to cython is:

```
#example_original.py
def test(x):
    y = 0
    for i in range(x):
        y += i
    return y
```

Listing 31: "Initial python file example_original.py "

```
cpdef int test(int x):
    cdef int y = 0
    cdef int i
    for i in range(x):
        y += i
    return y
```

Listing 32: "Initial python file converted for full cython use: example_cython_full.pyx"

To run cython the python script setup.py is needed

```
from distutils.core import setup
from Cython.Build import cythonize

setup(ext_modules = cythonize('example_cython_full.pyx'))
```

Listing 33: "setup_cython_full.py operating on example_original.pyx"

This is run at the command line:

```
python setup_cython_full.py build_ext --inplace
```

Testing

To run the python and cython versions the python script Testing_things.py is used

```
#testing_things.py
import timeit

cy = timeit.timeit('''example_cython.test(5000000)''',setup='import
    example_cython',number=100)
cyf = timeit.timeit('''example_cython_full.test(5000000)''',setup='import
    example_cython_full',number=100)
py = timeit.timeit('''example_original.test(5000000)''',setup='import
    example_original', number=100)
```

Listing 34: "testing_things.py"

This is run at the command line:

```
python testing_things.py
```

Having run the code 100 times with the parameter 5,000,000 this produces the output

```
Original py= 23.97049373597838
Cython cy= 0.1541433539823629
Cython_full= 0.15412238001590595
Cython is 155.5077991797239x faster
Cython_full is 155.52896168294666x faster
Cython_full is 1.0001360864428306x faster than cython
```

The full cython case here produces little difference as pointers are not very relevant here. The speed up can be quite spectacular.

The C/C++ code that is produced is very hard to read. It will run with almost any C/C++ compiler on any system, provided all necessary libraries are available to the C/C++ compiler. Math is readily available, but numpy and others are not so readily available.

The example here compiles functions to create a library which other programs can then call.

Numba

The numba scheme uses just-in-time (jit) compilation to compile code. This is very effective when creating modules as once debugged the compiled code is imported into the Python scripts that reference it.

To use numba it needs to be imported, and then before each function reference an @jit line is included to tell numba to operate on that function.

The same example calculation as considered with Cython is used here.

```
1  from numba import jit
2  import time
3  @jit(nopython=True)
4  def test_numba(x,nl):
5      for ii in range(0,nl):
6          y = 0
7          for i in range(x):
8              y += i
9      return y
10 def test(x,nl):
11     for ii in range(0,nl):
12         y = 0
13         for i in range(x):
14             y += i
15     return y
```

Listing 35: "Function definitions in the first part of example_numba.py"

The import statement makes the just in time feature of numba available to this script.

The function test_numba will have jit applied, while the function test is the standard Python implementation.

The number of repeat calculations (nl) is a parameter to both functions.

The rest of the code sets parameters and measures times.

```
16 param=500000
17 nloop=600
18 time_a=time.time()
19 test_numba(param,nloop)
20 time_b=time.time()
21 test(param,nloop)
22 time_c=time.time()
23 t_python=time_c-time_b
24 t_numba=time_b-time_a
25 print("Python took ",t_python)
26 print("Numba took ",t_numba)
27 print("Numba is %g times faster"%((1.0*t_python)/(1.0*t_numba)))
```

Listing 36: "example_numba.py"

Having run the code the output shows an impressive speed up:

```
Python took  19.962265253067017
Numba took  0.06246471405029297
Numba is 319.577 times faster
```

Finally

1. Compiled code executes faster than interpreted code.
2. The standard Python libraries that are imported are debugged, compiled, optimised code. Best practice is to make full use of them.
3. In writing code eliminate repeated calculations and indexing of arrays, lists and dictionaries.
4. Reduce I/O to the screen to just what is needed, and use log files where extra output may be necessary.
5. Cython can produce C/C++ code which compiles to optimised code if there is a compatible C/C++ compiler.
6. Numba uses just in time compilation that can provide similar speed up. This is all in Python so no second compiler is needed and there are no binary incompatibility issues.

11 Instrument Control

Instruments can be controlled by a processor or computer system of some description. Indeed many current laboratory instruments have their own built in microcontroller, microprocessor or in some cases a complete Windows or Linux PC.

Generally commands will be issued by the controlling processor and the instrument produces responses, often in the form of blocks of data.

Memory or Port mapped devices

Simple instruments such as basic ADCs or DACs may have an enable or trigger pin and data pins. A program can control this to initiate the action of the ADC or DAC by setting the enable pin to high and writing or reading data bits at other pins. In a memory mapped device these pins are mapped, by interface circuits and the main computer data/control bus, to a particular address in the memory of a PC. In the controlling program there could be statements like:

```
DAC_value = 0b11001          # binary value 11001
DAC_address = 0x00008800     # HEX value 8800
ADC_address = 0x00008820     # HEX value 8820
output(DAC_address, DAC_value)
ADC_value = input(ADC_address)
ADV_volts = 5.0*ADC_vlaue/65536
```

A value is output to the DAC, provided the operating system has configured correctly that the DAC is at the memory address 8800. A 16 bit binary value is then read in from the pins of the ADC at address 8820, which is a 16 bit converter. This binary value is converted into a floating point number for further calculations.

The memory mapped devices generally have to occupy a limited section of memory, perhaps HEX 8800 to 88FF. The number of devices is then limited as they can't use the same memory addresses. It is quite easy to program in the wrong addresses so values intended for the DAC are sent to the ADC, a graphics card or some other device. With the memory mapped scheme it could be difficult to verify what device was actually present at the ADC_address - it was up to the user to make sure the right device sat at the right address.

Serial interface devices

Serial devices send command and data as a serial series of bits. The devices at either end of the link need some processing power, but not much, to interpret the serial bit stream to and from characters and values.

To communicate via the serial port on a Laptop/PC in Python the typical interfaces used are pyserial or usblib.

The instrument being controlled needs to receive messages via the USB connection and respond to them. Some programmable intelligence needs to be built into the instrument. This will involve more hardware and programming on the instrument.

More sophisticated instruments such as the Keysight instruments in the Undergraduate Labs have a serial interface which can receive commands and respond to them. The commands and responses are generally short text strings, originally limited to four characters. This helps in debugging code as the text strings are easy to read or print out. Each command text string typically mimics the pressing of a button on the instrument's control panel.

To enable Python for such instrument control need to install pyvisa and the Agilent IO Libraries. To communicate with Arduino devices there is the Firmita module.

Arduino Example

Arduinos can be programmed from Python via the Firmita interface. This interface provides the connection from Python scripts through to individual I/O pins on the Arduino.

```
import pyfirmata
import time

board = pyfirmata.Arduino('/dev/ttyACM0')

while True:
    board.digital[13].write(1)
    time.sleep(1)
    board.digital[13].write(0)
    time.sleep(1)
```

Here the Arduino is identified through the Linux operating system as being at /dev/ttyACM0 in the file system. The 'digital' member of 'board'

accesses the digital I/O pins, and in this case pin 13 is repeatedly toggled 1010101010 with a one second duration for each bit.

Pyvisa example

Pyvisa provides the link between Python scripts and many, many instruments. To communicated with the Keysight equipment in the Undergraduate Labs the Agilent IO library also needs to be present. Many equipment manufacturers have their own libraries to provide the low level interface driver between a PC and their equipment.

Connection to equipment

The pyvisa module needs to be imported. Then the ResourceManager can be called to list out what visa resources are visible.

```
import pyvisa
rm = pyvisa.ResourceManager()
print(rm.list_resources())
lst=rm.list_resources()
print("List_resources done")  #"returns <%s>"%(st))
n_resources=len(lst)
print("%d resources found"%(n_resources))
```

Listing 37: "Establishing devices seen by visa."

The resource lists returned by the ResourceManager is easily printed out to the screen.

```
('USB0::0x0957::0x2707::MY58000772::0::INSTR', 'USB0::0x2A8D::0x1787::
    CN57344226::0::INSTR')
List_resources done
2 resources found
```

Listing 38: "Output generated from using ResourceManager."

Each visa device has quite a long complex identifier, which does not easily identify the individual instruments to the program. Also the order in which equipment is found depends on the PC and the operating system. The list can also contain a list of all of the visa devices you have ever encountered. More code is needed to identify the equipment found in the list.

The ResourceManager also has a function "list_resources" which will produce a list of devices that can be interrogated individually.

```
for ii in range(0,n_resources):
    resource=lst[ii]
    print("Resource[%d]=<%s>"%(ii,resource))
    try:
        DSO_OK=True
        DSO = rm.open_resource(resource)  #'USB0::0x2A8D::0x1787::CN57344491
    ::0::INSTR')
        print("\tResource[%d] opened, now querying"%(ii))
        print(DSO.query("*IDN?"))
    except:
        print("Attempt to open DSO on <%s> failed"%(resource))
        DSO_OK=False
        DSO_OK=False
st_DSO=input("Enter resource number for DSO :")
st_SGEN=input("Enter resource number for SGEN :")
```

Listing 39: "Enquire which devices are present and to be used."

The try/except clause is used to see if each of the elements in the list produces by list_resources can be opened. The user needs to monitor the output to see which index the valid devices are listed at.

```
Resource[0]=<USB0::0x0957::0x2707::MY58000808::0::INSTR>
   Resource[0] opened, now querying
Agilent Technologies,33511B,MY58000808,5.00-1.19-2.00-58-00

Resource[1]=<USB0::0x2A8D::0x1787::CN57344194::0::INSTR>
   Resource[1] opened, now querying
KEYSIGHT TECHNOLOGIES,DSO-X 1102A,CN57344194,01.01.2016092800


Enter resource number for DSO :1

Enter resource number for SGEN :0
```

Listing 40: "Using ResourceManager."

Having noted the index for the resources that exist the user enters the relevant index for the 33511B signal generator and the DSO-X 1102A oscilloscope. In this case the 33511B signal generator is index 0 and the DSO-X 1102A is index 1.

The Python script then attempts to open the oscilloscope and signal generator that the user has provided the index for.

```
j_DSO=int(st_DSO)
j_SGEN=int(st_SGEN)
DSO_str=lst[j_DSO]   #Example:  'USB0::0x2A8D::0x1787::CN57344242::0::INSTR'
try:
    DSO_OK=True
    DSO = rm.open_resource(DSO_str)   #'USB0::0x2A8D::0x1787::CN57344491::0::
    INSTR')
except:
    print("Attempt to open DSO on <%s> failed"%(DSO_str))
    DSO_OK=False
SGEN_str=lst[j_SGEN]   #'USB0::0x0957::0x2707::MY58000809::0::INSTR'
try:
    SGEN_OK=True
    SGEN = rm.open_resource(SGEN_str)
except:
    print("Attempt to open SGEN on <%s> failed"%(SGEN_str))
    SGEN_OK=False
if (not(SGEN_OK and DSO_OK)):
    print("\n\tFailed to find devices, exiting\n")
    sys.exit(2)
```

Listing 41: "Test devices can be opened."

Having identified and 'opened' the DSO and the signal generator the devices are queried by asking for their identifiers:

```
print("ResourceManager returns:\n"+str(rm))
print(DSO.query('*IDN?'))
print(SGEN.query('*IDN?'))
```

Listing 42: "Query devices for their identifiers."

The DSO and the signal generator return their identifiers, which confirms to the user that the correct devices have been located and opened:

```
ResourceManager returns:
Resource Manager of Visa Library at C:\windows\system32\visa32.dll
KEYSIGHT TECHNOLOGIES,DSO-X 1102A,CN57344194,01.01.2016092800

Agilent Technologies,33511B,MY58000808,5.00-1.19-2.00-58-00
```

Listing 43: "Query identifier responses."

Set up limits for the instruments

In the Python script the signal generator is set to produce 21 logarithmically spaced frequencies between 100 Hz and 1 MHz. The amplitude of the output from the signal generator will be 5 Volts. Some data arrays are also needed to store results.

```
nfreq=21          # no. of frequencies to measure
fstart=100.0      # start frequency
flast=1.0e6       # stop frequency
amplit=5.0        # amplitude for signal generator
lfstart=math.log(fstart)
lflast=math.log(flast)
lfstep=(lflast-lfstart)/(nfreq-1)
V1=np.zeros(nfreq,float)
freq1=np.zeros(nfreq,float)
V2=np.zeros(nfreq,float)
freq2=np.zeros(nfreq,float)
fask=np.zeros(nfreq,float)
```

Listing 44: "Setup measurement data and storage arrays."

Set up Waveform Generator and DSO

The signal generator is then instructed to produce a sine wave output with a peak value of +5 Volts and a minimum value of -5 Volts. Its output is then switched on. The oscilloscope is set to measure with an averaging factor of 32. Note these commands are the same as pressing buttons on the front panels of the instruments.

```
# signal generator and DSO set up
SGEN.write('FUNC SIN')              #select sinewave
st="VOLTAGE:HIGH %.1f"%(amplit)     # max voltage
SGEN.write(st)
st="VOLTAGE:LOW %.1f"%(-amplit)     # min voltage
SGEN.write(st)
SGEN.write('PHASE 0.0')
SGEN.write('OUTPUT ON')             # switch it on!
DSO.write('ACQuire:TYPE AVERage')   # Set DSO to average
DSO.write('ACQuire:COUNt 32')       # Averaging factor=32
sleep1=1.5                          # settle time for Autoscale
sleep2=0.1                          # settle time between data requests
```

Listing 45: "Setup Generator oscilloscope and delays."

Loop through measurements at many frequencies

The measurements loop through the frequency range that has been set up. The frequency is sent to the signal generator and then the oscilloscope is commanded to AutoScale. This takes some time to complete, so the program waits for the time set in "sleep1" (1.5 seconds) before proceeding.

```
for jj in range(0,nfreq):
    lff=lfstart+jj*lfstep
    ff=math.exp(lff)              # calculate frequency
    fask[jj]=ff                   # store for later use
    st='FREQ %.1f'%(ff)
    SGEN.write(st)                # sets signal gen frequency to ff
    DSO.write('AUTOSCALE')        # put DSO into Autoscale
    time.sleep(sleep1)            # wait enough time for Autoscale to finish
```

Listing 46: "Set frequency and Autoscale the DSO."

The measurements for channel 1 are triggered and the frequency and RMS of the waveform are requested. The return from the instruments is a text string, which has to be converted into a float value for calculations.

```
    print("Setup done, now digitize freq[%d/%d]"%(jj,nfreq))
    DSO.write(':DIGITIZE CHAN1')    # digitize Channel 1
    time.sleep(sleep2)              # give DSO time to complete
    DSO.write('MEASURE:SOURCE CHANNEL1')
    time.sleep(sleep2)              # give DSO time to complete
    buffer=DSO.query('MEASURE:FREQUENCY?')   # text is returned from
   instrument
    freq1[jj]=float(buffer)                  # convert text to float
    DSO.write('MEASURE:VRMS CHANNEL1')
    time.sleep(sleep2)              # give DSO time to complete
    buffer=DSO.query('MEASURE:VRMS?')
    V1[jj]=float(buffer)
```

Listing 47: "Measure Channel 1."

The process is then repeated for Channel 2. Finally a summary of the measured frequencies and voltages is printed.

```
    DSO.write('MEASURE:SOURCE CHANNEL2')
    time.sleep(sleep2)
    buffer=DSO.query('MEASURE:FREQUENCY?')
    freq2[jj]=float(buffer)
    DSO.write('MEASURE:VRMS CHANNEL2')
    time.sleep(sleep2)              # give DSO time to complete
    buffer=DSO.query('MEASURE:VRMS?')
    V2[jj]=float(buffer)
    print("F1=%9.2f, V1=%9.4fV, F2=%9.2f, V2=%9.4fV"%(freq1[jj],V1[jj],freq2[
   jj],V2[jj]))
```

Listing 48: "Measure Channel 2."

Output data tables

The data is finally written out to a comma separated variables format file, and printed to the screen.

```
fname='FrequencyResponse.csv'
fout=open(fname,"wt")
fout.write("f ask, f1 meas, V1, f2 meas, V2\n")
for jj in range(0,nfreq):
    print("F[%3d]=%10.1f, F1=%10.1f, V1=%9.4fV,  F2=%10.1f, V2=%9.4fV"%(jj,
    fask[jj],freq1[jj],V1[jj],freq2[jj], V2[jj]))
    st=("%13.6e, %13.6e, %13.6e, %13.6e, %13.6e\n"%(fask[jj],freq1[jj],V1[jj
    ],freq2[jj], V2[jj]))
    fout.write(st)
fout.close()
```

Listing 49: "Output data tables to screen and file."

Close down and finish

Finally the signal generator output is switched off and the DSO is instructed to finish outputting data. Then the connections to the instruments are closed.

```
# Finished, so now turn off
DSO.write('SOUR:OUTP OFF')
SGEN.write('OUTPUT OFF')
# Close the VISA connection.
DSO.close()
SGEN.close()
print("Done, data written to file <%s>\n\t\tBye bye"%(fname))
```

Listing 50: "Close down the signal generator and DSO."

On the EE20084 Moodle pages the Python script "Freq_resp_02.py", the output printed to the screen captured into the file "Run_freq_resp_03.txt" and the output file "FrequencyResponse.csv" are there for you to examine.

Finally

1. Instrument control via memory or port mapping can be very fast, but needs to be set up and handled carefully.
2. Much more sophistication can be built into serial interface devices, and these have become much more common.
3. A major feature is that the device can be identified quite easily, rather than relying on the assumption that the right device is in the right place.
4. Most serial interfaces use short (4 character?) text commands and data is sent as standard UTF-8 text strings.

1. In programming instruments such as those in the Undergraduate Keysight Labs the text commands mimic button presses on the instrument. To program the instrument you need to be familiar with the 'manual' usage of the instrument.
   (a) You need to plan or DESIGN the sequence of button presses needed to perform the operations/measurements you want, and then code it into a Python script.

12 Optimisation and use in Artificial Intelligence(AI)

In Artificial Intelligence (AI) systems, and many other systems, an output or end condition is sought as a function of some control or input variables. The outputs could be a speed, a velocity if direction of motion is known, a (GPS?) position, an output (loudness, brightness) level, ...

Generally there is a desired state that we wish to achieve and there is the current state. The optimisation process tries to move the current state to be the desired state.

Considering a current position (x_c, y_c) and a desired position (x_d, y_d) the simple linear sum of differences between these is:

$$\Delta x \ + \ \Delta y \ = \ (x_t - x_d) \ + \ (y_t - y_d)$$

The adjacent diagram shows this could easily be zero even though there is a difference in the two positions.

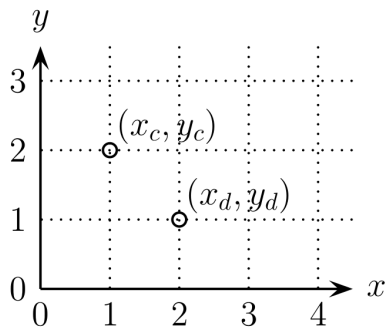


Figure 12.1: Desired (x_d, y_d) and current (x_c, y_c) positions.

Instead the sum of the squares of the differences, or radial distance, is used.

$$\Delta x^2 \ + \ \Delta y^2 \ = \ (x_t - x_d)^2 \ + \ (y_t - y_d)^2$$

This produces a positive value whenever there is a difference in the positions.

The mean square difference or error between the current (C) state and the desired (D) state provides a cost function. If we are taking N measures:

$$O = \frac{1}{N} \sum_{n}^{N} (C_n \ - \ D_n)^2$$

Hence all differences between the C_n and D_n contribute to the cost or objective function with no cancellation.

We may wish for some differences to have a greater influence, in which case a weight W_n is applied to each difference calculation:

$$O = \frac{1}{N} \sum_{n}^{N} W_n \, (C_n \ - \ D_n)^2$$

The objective function may have a simple variation, such as in the adjacent figure. There is a single minimum to find which is quite well defined. In fact the objective function is zero showing that an exact achievement of the target value is possible.

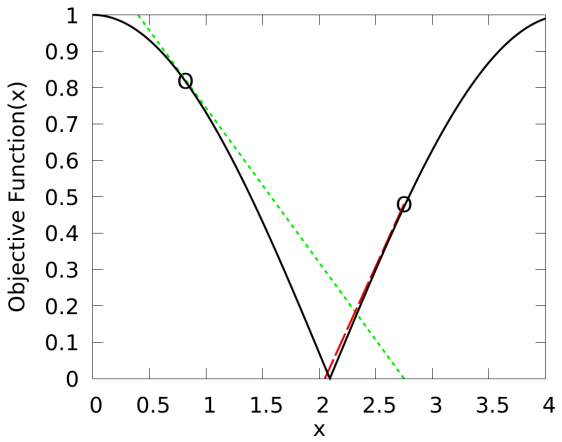


Figure 12.2: Simple objective function and Newton-Raphson iterative solution

Sampling the Objective Function at many x points and choosing the case where the Objective Function is least provides a simple solution. This can take much computation time - the adjacent graph is plotted with 50 points.
The Newton-Raphson iteration technique can be used to find the minimum of the objective function more efficiently.

Newton-Raphson

The Newton-Raphson iteration produces an updated (improved?) estimate of x_{i+1} from a previous estimate x_i and the gradient of the objective function:

$$x_{i+1} = x_i - \frac{d\,O(x)}{dx}$$

If the initial estimate of x is 0.8, the objective function is 0.82. The slope of the objective function gives an improved estimate of $x = 2.75$. The objective function at $x = 2.75$ is 0.48. This is lower than the original 0.82 so the estimate of x has improved.

Evaluating another Newton-Raphson iteration the slope at $x = 2.75$ points to a new x value of about 2.05 where the objective function is about 0.02. This is very close to the position where the objective function is zero.

The Newton-Raphson iteration process will repeat until it reaches termination conditions. These are generally made with respect to:

Function size $Ftol$**:** Once the objective function $O(x_{i+1}) < Ftol$ the solution is considered to be found.

Step size $Xtol$**:** Once the step between iterations $(x_{i+1} - x_i) < Xtol$ the solution is considered to be found. Some idea of how accurate the x value needs to be is needed to set $Xtol$.

$Xtol$ needs to be greater than the smallest increment in a floating point value that the computer can evaluate. In Python floating point values can generally be distinguished in the 16^{th} decimal place.

More commonly the objective function can have several minima, which are not so sharply defined.

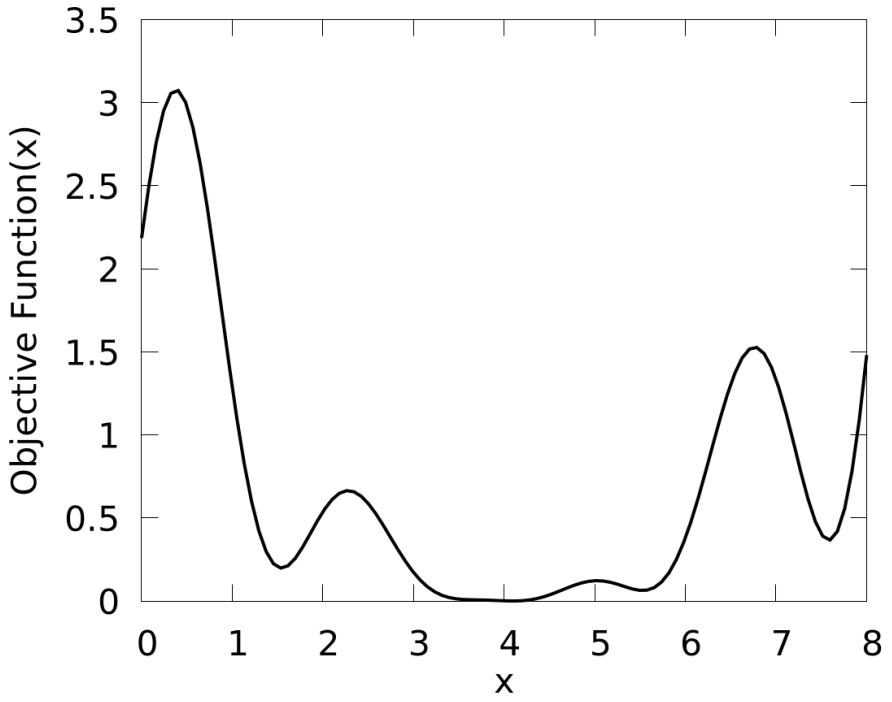


Figure 12.3:

Minima in objective functions are often quite smooth. The gradient evaluation near the minima can become low.

Evaluating at $x > 4$ near the 'global minimum' the gradient is low and this could produce new estimates that are near 4.0 or 1.5. In the latter case the Newton-Raphson scheme could find the minima near 1.5. Further a subsequent very low gradient evaluation around $x = 1.5$ could

point back to the minima near 4, or 5.5, or 7.5. Indeed as the minima are approached the Newton-Raphson scheme may well step well away from minima.

Two and multidimensional cases

Considering the case where a desired position (x_d, y_d) is sought the square error between all possible positions and the desired position is

$$\Delta x^2 \, + \, \Delta y^2 \; = \; (x_t - x_d)^2 \, + \, (y_t - y_d)^2$$

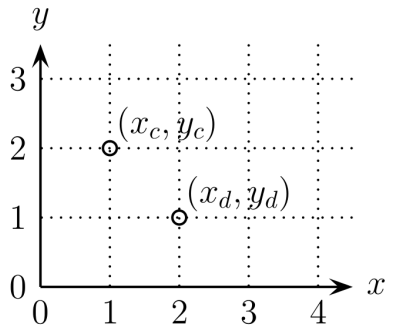


Figure 12.4: Desired (x_d, y_d) and current (x_c, y_c) positions.

This difference varies in both x and y, which is quite easily shown on a two dimensional surface.

In the N-dimensional cases there are N independent variables $X = (x_1, x_2, \ldots x_N)$ rather than just x. To provide improved estimates of X in a minimisation scheme there need to be M functions or measurements where $M \geq N$. These functions are denoted $F = (f_1, f_2, \ldots f_M)$. The gradient of the objective function with respect to X is determined from the Jacobian matrix:

$$J_F \; = \; \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_N} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_N} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_M}{\partial x_1} & \frac{\partial f_M}{\partial x_2} & \cdots & \frac{\partial f_M}{\partial x_N} \end{pmatrix}$$

Here the gradient of every function f_j with every variable x_j is evaluated. This is used to find a better estimate of X_{i+1} from X_i

In some cases the derivatives in the Jacobian matrix can be identified analytically by a formula. In other cases this may not be possible. Here the optimisation scheme can use numeric differentiation to determine the Jacobian matrix. Small variations in the $x_1, x_2, \ldots x_N$ are used and the resulting changes in the $f_1, f_2, \ldots f_M$ to find the $\partial f_1/\partial x_1, \ldots \partial f_M/\partial x_N$. This can take a lot of computation time.

Plotting the (square error) objective function with two independent variables x and y as a surface shows how a gradient following process can find the minimum of the objective function.

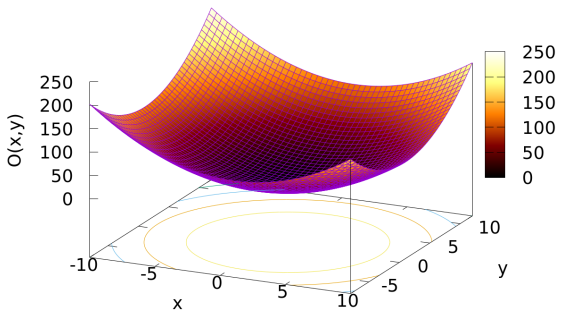


Figure 12.5: Example Objective function with two independent variables with simple behaviour.

There is still an issue with the gradient becoming very low near the final solution minimum.

In the general case the objective function can have many minima. As in the case of the single independent variable only one of these might be the minimum that is desired - around (0,0) in the adjacent diagram.

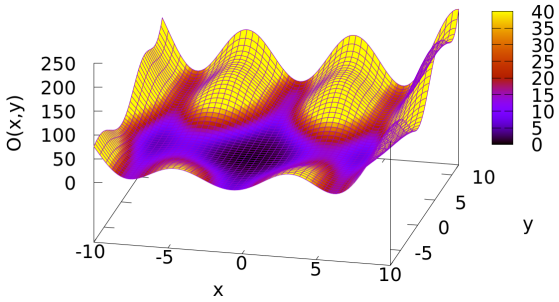


Figure 12.6: Example Objective function with two independent variables with more complicated behaviour.

In this case in the range $-5 \leq x \leq 5$ the gradient following algorithm will find the minimum near (0,0). Outside of this range it will find the higher value minima near $x = 7.5$ or $x = -7.5$.

Optimisers generally need to be constrained in the range over which they can vary X by putting bounds on X.

Searching for a solution should terminate if:

$X_{i+1} - X_i \leq Xtol$**:** once the X value is sufficiently accurate.

$O(X_{i+1}) \leq Ftol$**:** once the objective function value is sufficiently small.

$|J_F| \leq Gtol$**:** if the gradient is small. This helps to prevent the gradient optimiser from stepping too far away from its last estimate of X.

Number of iterations > $Nmax$ **:** Limits the number if iterations that can be taken.

Diode Law example

The Diode Law links the voltage V applied to a diode to I_d the current flow through the diode.

$$I_d \;=\; I_s \;\left[\exp\left(\frac{q\,V}{\eta\;k\;T}\right)-1\right] \tag{16}$$

where

$$\begin{aligned} I_s &= \text{Reverse saturation current} \\ q &= \text{Charge on an electron } = 1.6\times10^{-19}\text{ Coulombs} \\ k &= \text{Boltzmann's Constant } = 1.38\times10^{-23}\text{ Joules/K} \\ T &= \text{Temperature in degrees Kelvin} \\ \eta &= \text{Ideality factor} \quad 1\le\eta\le2 \end{aligned}$$

Taking the logarithm of the Diode Law gives:

$$\log_e\left(I_d\right) \;=\; \log_e\left(I_s\right) \;+\; \log_e\left[\exp\left(\frac{q\,V}{\eta\;k\;T}\right)-1\right] \tag{17}$$

If $I_d \gg I_s$ the equation approximates to

$$\begin{array}{rcccc} \log_e\left(I_d\right) & = & \log_e\left(I_s\right) & + & \left(\frac{q}{\eta\;k\;T}\right)\;V \\ Y & = & C & + & M\;\;V \end{array} \tag{18}$$

i.e. a straight line where $C=\log_e\left(I_d\right)$ and $M=\left(\frac{q}{\eta\;k\;T}\right)$.

In this example there are some measurements of the diode current for particular voltages. The optimiser is used to adjust T, η and I_s, the independent variables, to find the minimum square difference between the measured currents and the currents given by the diode law at the measured voltages.

In Diode_opt_01.py the optimisation code is imported from scipy, and the least squares method is to be used.

```
import math
import numpy as np
from scipy.optimize import minimize
from scipy.optimize import least_squares
```

Listing 51: "Imports for the optimisation example."

The linear and logarithmic forms of the diode law are determined in functions. These take an array of voltages as an input and return the corresponding array of diode currents.

```
def Diode_Eq(Temp,eta,Is,V):
    alpha=1.6e-19/(eta*1.38e-23*Temp)
    nV=len(V)  # V is an array in general
    Id=np.zeros(nV,float)
    for i in range(0,nV):
        Id[i]=Is*(math.exp(alpha*V[i])-1.0)
    return(Id)

def Log_Diode_Eq(Temp,eta,LIs,V):
    alpha=1.6e-19/(eta*1.38e-23*Temp)
    nV=len(V)   # V is an array in general
    Id=np.zeros(nV,float)
    for i in range(0,nV):
        Id[i]=LIs + math.log10( (math.exp(alpha*V[i])-1.0) )
    return(Id)
```

Listing 52: "Linear and logarithmic forms of the Diode Law."

The Python program sets x to contain $x[0] = T$, $x[1] = \eta$, $x[2] = I_s$. These independent variables are varied by the optimiser to minimise the Residual_Function. The Residual_Function finds the difference between the values of current produced by the Diode function and the measured values of current held in the array y.

```
def Diode(x,Voltages):
    Idiode=Diode_Eq(x[0],x[1],x[2],Voltages)
    return(Idiode)

def Residual_Function(x,u,y):
    return( Diode(x,u)-y )

def Log_Diode(x,Voltages):
    Idiode=Log_Diode_Eq(x[0],x[1],x[2],Voltages)
    return(Idiode)

def Log_Residual_Function(x,u,y):
    return( Log_Diode(x,u)-y )
```

Listing 53: "Residual functions calling interface functions to the Diode Law functions."

The initial_parameters array sets $T = 290$, $\eta = 1.5$ and $I_s = 10^{-9}$. The parameter_bounds array restricts the ranges the optimiser may use to $250 \leq T \leq 400.4$, $1.0 \leq \eta \leq 2.1$ and $10^{-11} \leq I_s \leq 10^{-5}$.

The voltages where the currents were measured are held in Vmeasurements and the corresponding currents are in Imeasurements.

```
initial_parameters=(290.0, 1.5, (1.0e-9))
parameter_bounds=( (250.0, 1.0, (1e-11)), (400.4, 2.1, (1e-5)) )
Log_initial_parameters=(290.0, 1.5, math.log10(1.0e-9))
Log_parameter_bounds=( (250.0, 1.0, math.log10(1e-11)), (400.4, 2.1, math.
    log10(1e-5)) )
```

```
Vmeasurements=(    1e-6,     1e-5, 1.0e-4, 1.0e-3, 1e-2, 1e-1, 2.0e-1, 3.0e-1,
    4.0e-1, 5.0e-1)
Imeasurements=(9.7e-11, 9.7e-10, 9.7e-9, 9.7e-8, 1e-6, 3e-5, 2.3e-4, 1.6e-3,
    1.1e-2, 7.9e-2)
```

Listing 54: "Initial estimates of the unknown parameters and limits on the ranges the optimiser can use."

The least_squares function from scipy is then called to operate on Residual_Function based on the initial_parameters, the bounds on the parameters, the measured voltages and currents are held in args. Verbose is set to 2 to provide some print out to the screen as the least_squares function executes.

```
res = least_squares(Residual_Function, initial_parameters, bounds=
    parameter_bounds, args=(Vmeasurements, Imeasurements), verbose=2)
print(str(res.x))
Tf=res.x[0]
etaf=res.x[1]
Isf=res.x[2]
print("Linear data gives T=%g, eta=%g, Is=%g"%(Tf,etaf,Isf))
```

Listing 55: "Calling the least_squares optimisation function using linear data and unpacking the solution."

After least_squares has executed the temperature, ideality factor and saturation current are 'unpacked' from the output 'res' and printed out.

The minimisation is then performed for a second time but using the logarithmic form of the diode law and minimising to the logarithm of the measured currents.

```
nMeas=len(Imeasurements)
log_Imeasurements=np.zeros(nMeas,float)
for j in range(0,nMeas):
    log_Imeasurements[j]=math.log10(Imeasurements[j])
res = least_squares(Log_Residual_Function, Log_initial_parameters, bounds=
    Log_parameter_bounds, args=(Vmeasurements, log_Imeasurements), verbose=2)
print(str(res.x))
lTf=res.x[0]
letaf=res.x[1]
llsf=pow(10.0, res.x[2])
print("Logarithmic form gives T=%g, eta=%g, Is=%g"%(lTf,letaf,llsf))
```

Listing 56: "Calling the least_squares optimisation function using logarithmic data and unpacking the solution."

The output produced by least_squares shows how the cost function reduces over 35 iterations and exited as it met the default limit for the gradient, gtol. Final values were $T = 294$, $\eta = 2$ and $I_s = 4.2\mu A$.

```
runfile('H:/TEACH/Struct_Prog_EE20084/Python_Notes_2020_21/Python_code/Optimise/Diode_opt_01.py', wdir='H:/TEACH/
    Struct_Prog_EE20084/Python_Notes_2020_21/Python_code/Optimise')
   Iteration     Total nfev        Cost      Cost reduction    Step norm     Optimality
       0              1         3.1336e-03                                  4.85e-01
       1              2         2.8618e-05      3.10e-03       4.68e-01      2.31e-03
       2              3         1.5874e-05      1.27e-05       7.17e-01      8.17e-05
       3              6         1.5664e-05      2.10e-07       3.96e-02      7.59e-03
```

Listing 57: "Start ofoptimisation on linear data."

```
      30             33         2.5714e-09      1.98e-10       3.64e-02      3.85e-07
      31             34         2.5682e-09      3.26e-12       6.96e-03      1.35e-08
      32             35         2.5682e-09      1.29e-15       1.10e-04      2.83e-12
`gtol` termination condition is satisfied.
Function evaluations 35, initial cost 3.1336e-03, final cost 2.5682e-09, first-order optimality 2.83e-12.
[2.94012554e+02 2.00231389e+00 4.17885008e-06]
Linear data gives T=294.013, eta=2.00231, Is=4.17885e-06
```

Listing 58: "End of optimisation on linear data."

Running on the logarithmic form took 7 iterations, exiting because the Residual_Function was less than Ftol. Final values were $T = 341$, $\eta = 1.76$ and $I_s = 5\mu A$.

```
   Iteration     Total nfev        Cost      Cost reduction    Step norm     Optimality
       0              1         5.0642e+01                                  1.26e+02
       1              2         1.0801e+01      3.98e+01       9.18e+00      3.00e+01
       2              3         3.0134e+00      7.79e+00       5.34e+00      7.04e+00
       3              4         5.7715e-01      2.44e+00       2.43e+01      1.41e+00
       4              5         7.7347e-02      5.00e-01       1.75e+01      2.62e-01
       5              6         4.9734e-03      7.24e-02       9.45e+00      4.13e-02
       6              7         5.2092e-04      4.45e-03       3.02e+00      3.82e-03
       7              8         4.8456e-04      3.64e-05       3.06e-01      3.91e-05
       8              9         4.8456e-04      3.19e-09       3.04e-03      3.59e-08
       9             12         4.8456e-04      1.91e-14       1.79e-02      8.08e-07
`ftol` termination condition is satisfied.
Function evaluations 12, initial cost 5.0642e+01, final cost 4.8456e-04, first-order optimality 8.08e-07.
[340.72285631   1.76195112  -5.30498273]
Logarithmic form gives T=340.723, eta=1.76195, Is=4.9547e-06
```

Listing 59: "Running diode optimisation logarithmic example."

Finally the program outputs the voltages, measured currents, the 'fitted' currents from the optimised values of T, η and I_s and the percentage differences between the measured and fitted currents.

```
  Voltage,      Meas I  |     Linear I, lin %err,       Log I, log %err
    1e-06,      9.7e-11  | 8.23008e-11,     -15.2, 9.56901e-11,      -1.4
    1e-05,      9.7e-10  | 8.23081e-10,     -15.1, 9.56985e-10,      -1.3
   0.0001,      9.7e-09  | 8.23811e-09,     -15.1, 9.57817e-09,      -1.3
    0.001,      9.7e-08  | 8.31157e-08,     -14.3, 9.66192e-08,      -0.4
     0.01,        1e-06  | 9.09635e-07,      -9.0, 1.05554e-06,       5.6
      0.1,        3e-05  | 2.57696e-05,     -14.1, 2.92246e-05,      -2.6
      0.2,      0.00023  | 0.000210451,      -8.5, 0.000230826,       0.4
      0.3,       0.0016  |    0.001534,      -4.1,  0.00162154,       1.3
      0.4,        0.011  |   0.0110195,       0.2,   0.0112152,       2.0
      0.5,        0.079  |   0.0789986,      -0.0,   0.0773955,      -2.0
```

Listing 60: "Comparing linear and logarithmic optimisation output results."

Fitting with the linear form of the diode law gives quite poor accuracy at low currents but good at high currents. Fitting with the logarithmic form of the diode law gives quite good accuracy at all currents.

Using the linear data the values of low 'nanoamp' currents are almost negligible compared to the higher currents of tens of milliamps and the influence the nanoamp currents have on the objective function is almost negligible. The higher currents dominate the calculations. In the logarithmic form 1 nano is -9 while 10 milli is -2. These logarithmic values are of very similar size, so all currents have a significant effect on the objective function. Using the logarithmic form the lower currents are emphasised or weighted more than the higher currents in determining the objective function values.

In general the measures contributing to the objective function may well be of significantly different sizes. Lower values can be weighted (multiplied) by a large weight factor W_{low} while higher values can be weighted by a small weight W_{high} to equalise the importance of the inputs.

Optimisations generally have a weight that can be applied to every input. In a PID controller different weights can be applied to the Proportional, Differential and Integral measures.

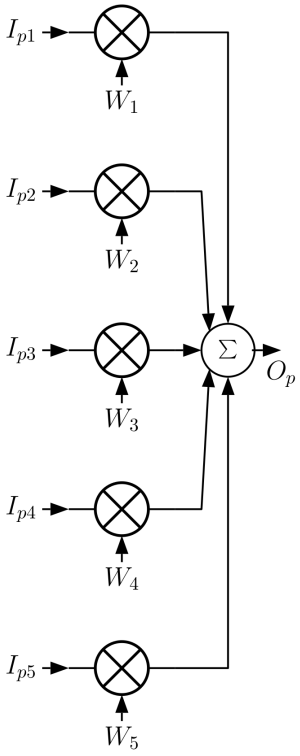


Figure 12.7: Summation of input variables $I_{p1} \ldots I_{p5}$ weighted by the weights $W_1 \ldots W_5$

The relative weighting of data in optimisers can be *quite critical* to the result achieved! When using linear data in the diode law example the optimisation was weighted towards high values of current, while the logarithmic form weighted towards the lower currents.

The spacing between samples or sampling rate of data can also be quite critical. Consider the set of data in Fig. 12.8. These look like they are on a straight line, and Fig. 12.9 shows a very good straight line fit can

be achieved.

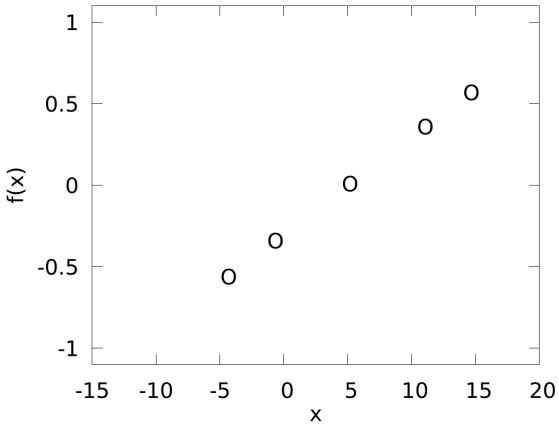


Figure 12.8: Some simple data

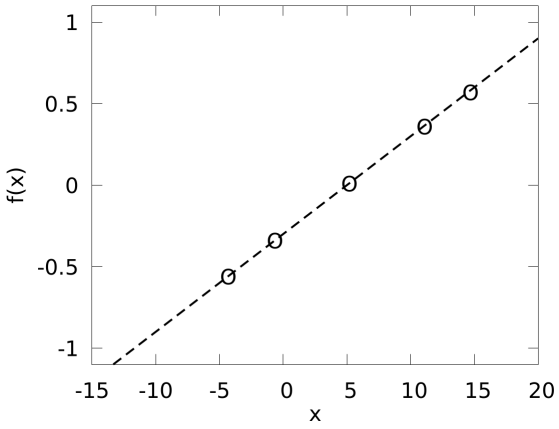


Figure 12.9: Straight line fit to data.

However that data could also be from a sine wave.

Intermediate samples are needed to find out if the trend is a straight line or a sine wave - or something else.

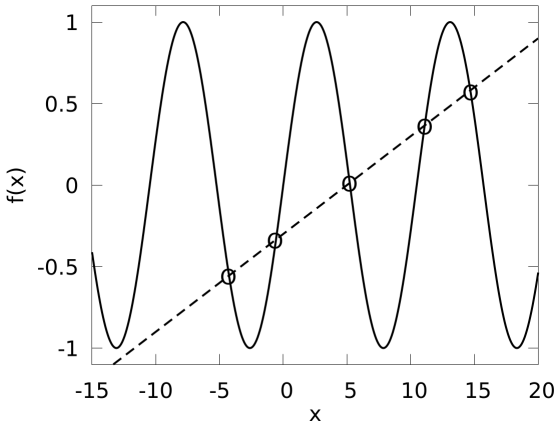


Figure 12.10: Straight line and sine wave fit to the data points.

At the sample points the fit to data is best. In between sample points, and above and below the sample range, the fit can be rather poor - if not misleading.

Finally

1. Numeric optimisation can find parameter values that push the current state or evaluation towards a desired state.
2. The simple approach of making many evaluations of the Objective Function and finding the lowest value can be computationally expensive.
3. Gradient optimisers are quite efficient in terms of the number of iterations taken, but are prone to moving away from a solution.
4. Weighting data so that all parameters are treated equally, or emphasising some, can be important.
5. Data is generally good at the sample points, but (much?) less reliable elsewhere.
6. Sampling density and width can influence results significantly. Some knowledge of expected variations can be important.

A Base Input/Output Tutorial

This tutorial looks at the base file input/output features in Python. Taking a simple example program you are to extend its functionality.

Example Program

The initial example program is **IO_example_01.py**, and it has an associated data file **file.txt**. These files are both on the Moodle site under the Labs topic in the file **Base_Input_Output.zip**.

Download and unpack the .zip file into a directory on your H-drive. The code can be run in Visual Studio Code or Spyder.

Spyder: Start Spyder (under Anaconda on the Windows Start Menu) and then open the file **IO_example_01.py**. Under **Run** select **Configuration per file** and enter **file.txt** in the **Command line options** box. It can be useful to also select **Remove all variables before execution**. Then click OK and **Run** (or press **F5**) the program.

Visual Studio Code: Start Visual Studio Code and using Open Folder navigate to the folder containing the python file **IO_example_01.py** and the file **file.txt**. Selecting **IO_example_01.py** from the Visual Studio Code Explorer opens the file for editing. The Python program needs filename file.txt to be passed to it as a command argument. To set this select **Run** and then **Open Configurations**. Under the "program" line in "configurations" add the line

```
"args": ["${workspaceFolder}/file.txt"],
```

and select save. Then select **IO_example_01.py** and **Run** and **Start Debugging** or **F5** to execute the program. The output from the program will appear in the **Terminal** window.

The program will run and try to read in variables from every line in 'file.txt', some lines succeed and others fail. Observe the lines that fail and note why they fail and where these cases are handled in the code.

File error

Rename **file.txt** to some other name such as **misnamed.txt** and run the program. The file will now not be found as the program command line is still setting the program to look for **file.txt**, and so the program terminates with error messages.

Rename **misnamed.txt** back to **file.txt**.

Extending I/O

The functionality of the program needs to be extended to also read in float and string variables. In extending the program it is wise to keep the original, working, program as **IO_example_01.py** and then operate on a new version, say **IO_example_02.py**. Make a copy of **IO_example_01.py** call it **IO_example_02.py** and make Visual Studio or Spyder operate on this new 'Version 02' of the program. Running this new program now should produce the same action as the old program, as nothing has changed apart from the file name.

Using the **find_int** function as an example create **find_float** and **find_string** functions to find float or string values following the variable name and equals sign. Using these your program should be able to read in all of the values in the correctly formatted lines in **file.txt**.

B Python String Functions

This Tutorial is to look at the various functions Python has that operate on strings, and apply them to the Input/Output parts of the Coursework Task.

Strings are implicitly included within Python. Input and output generally requires the interpretation of input strings to read in data values and the output of formatted strings to present results. This Tutorial session is for you to consider how the various string functions could be applied to implement the input and output parts of the Coursework Task. You should then proceed to implement parts of the input and output for the Coursework Task.

The following list includes many of the more commonly used string functions available in Python, and the string variables **st**, **s2**, **s3** and **pad** are used throughout the list.

Basic operations

len(st) determines the length of st while **st[i]** returns the character at index i of st.

Finding substrings

s2 in st : Return true if st contains s2

s2 not in st : Return true if st does not contain s2

st.count(s2): Count of s2 in st

st.find(s2) : Find and return lowest index of s2 in st

st.index(s2) : Return lowest index of s2 in st (but raise ValueError if not found)

st.index(s2, i, j) : Index of first occurrence of s2 in st after index i and before index j

st.rfind(s2) : Return highest index of s2 in st

st.rindex(s2) : Return highest index of s2 in st (raise ValueError if not found) j

Dividing/splitting strings

st[i:j] : Slice of st from i to j **st[i:] :** Slice of st from i to end **[st[:j] :** Slice of st from start to j **st[i:j:k] :** Slice of st from i to j with step k

st.strip() : Remove leading and trailing whitespace from st eg. ' hello ' => 'hello'

st.lstrip(): Remove leading whitespace from st eg. ' hello ' => 'hello '

st.rstrip(): Remove trailing whitespace from st eg. ' hello ' => ' hello'

st.partition(sep) : Partition string at sep and return 3-tuple with part before, the sep itself, and part after eg. 'hello' => ('he', 'l', 'lo')

st.rpartition(sep) : Partition string at last occurrence of sep, return 3-tuple with part before, the sep, and part after eg. 'hello' => ('hel', 'l', 'o')

st.split(sep) : Return list of st split by sep

st.split(sep, maxsplit) : Return list of st split by sep with leftmost maxsplits performed

st.rsplit(sep, maxsplit) : Return list of st split by sep with rightmost maxsplits performed

st.splitlines() : Return a list of lines in st eg. 'hello\nworld' => ['hello', 'world']

Forming strings

st + s2 : Concatenate st and s2

st="X=%9.2e"%(x): Set st using standard print formatting eg. x=42 => 'X= 4.20e+01'

st.center(width) : Center st with blank padding of width eg. 'hi' => ' hi '

st.center(width, pad): Center st with padding pad of width eg. 'hi' => 'padpadhipadpad'

st.ljust(width) : Left justifiy st with total size of width eg. 'hello' => 'hello '

st.rjust(width) : Right justify st with total size of width eg. 'hello' => ' hello'

C Using Numpy

This Tutorial looks at using a few of the features offered by Numpy. The Numpy arrays are often needed in computations and their use in simple matrix calculations are investigated here. Binary IO produces smaller output files and faster operations, and Numpy has some built in functions that simplify Binary IO. This is also examined here.

Arrays to hold Matrices

Create a Python program that

1. imports numpy
2. creates two 2×2 arrays matrices initially populated with all values being 3.0
3. sets the matrices to:

$$[A] = \begin{bmatrix} 1.0 & 3.0 \\ -2.0 & 1.0 \end{bmatrix} \qquad [B] = \begin{bmatrix} -2.0 & 1.0 \\ 2.0 & 3.0 \end{bmatrix}$$

4. uses numpy.matmul() or numpy.dot() to multiply the two matrices together to produce $[C] = [A][B]$ and prints out the matrix $[C]$.
5. uses numpy.add() to add the two matrices together to produce $[D] = [A] + [B]$ and prints out the matrix $[D]$.
6. uses numpy.multiply() to multiply the individual elements of $[A]$ and $[B]$ to produce $[E]$ and prints out the matrix $[E]$. *Note this is 'array' multiplication rather than matrix multiplication.*
7. creates a 2×4 array matrix $[P]$ and a 4×2 array matrix $[Q]$ and populates them as

$$[P] = \begin{bmatrix} 1.0 & 3.0 & -2.0 & 4.0 \\ -2.0 & 1.0 & 4.0 & -1.0 \end{bmatrix} \qquad [Q] = \begin{bmatrix} -2.0 & 1.0 \\ 2.0 & 3.0 \\ -4.0 & 1.0 \\ 1.0 & -3.0 \end{bmatrix}$$

8. uses numpy.matmul() or numpy.dot() to multiply the two matrices together to produce $[R] = [P][Q]$ and prints out the matrix $[R]$.

9. uses numpy.add() to add the two matrices together to produce $[S] = [P] + [Q]$ and prints out the matrix $[S]$.

10. uses numpy.multiply() to multiply the individual elements of $[P]$ and $[Q]$ to produce $[T]$ and prints out the matrix $[T]$. *Note this is 'array' multiplication rather than matrix multiplication.*

Binary IO and speed test

The Python program Numpy_IO_bintest.py uses Numpy functions to save and read back in data in three forms:

Text: using savetxt and loadtxt

Binary - single array: using save and load

Binary - multiple array: using savez and load

The program also uses timing functions to record how long the three methods take to execute.

The timings are gained using the **time** module. Once imported the time.time() function call finds out what the time is from the operating system. Calling this before and after a section of code is executed allows for simple evaluation of how long that code took to execute.

Run the program and observe the sizes of the files produced, and the times taken for the text and binary parts to execute. Run the program several times and you will note that the times vary somewhat.

Increase the array sizes to 500×500 and 2000×2000 and note the new times. These are longer and a little more consistent. Determine how many times faster the binary IO is and how many times smaller the files are.

D Simple event driven programs

This tutorial looks at examples of simple event driven programs. In the example used here the events are the interaction with the user, and this influences the execution of the program. In addition there is an opportunity to observe how numeric errors are observed.

Looking for errors with an Event driven program

The example program, available on the Moodle site, 'Events_Errors_01.py' can run from the command line, within Spyder or within a Visual Studio project. The code defines some functions to make very simple text based choice menus. The user's choices provide the events that drive the code.

Outline of the program

The program will evaluate a function $f(x)$ from a lower limit for x up to an upper limit for x in a number of steps, $nstep$. There is a choice of five functions that can be investigated. At each evaluation the program will print out the value of x and the corresponding value of the function $f(x)$.

The program defines some simple functions that display menus. These menus allow the user to select an option and to evaluate a chosen function across a range of ordinate values.

ActionMenu: Produces a top level menu where the user selects certain actions for the program to perform.

LimitsMenu: Allows the user to set the ordinate (x) limits for evaluations and the number of evaluations to make.

FunctionMenu: Allows the user to choose one of a set of functions to calculate.

evaluate: Performs the calculations of the chosen function over the ordinate range that has been set.

Running an example

Run the program. The 'Main Menu' is displayed on the screen or output window. The ordinate range and functions to evaluate need to be set through this menu, and these are initially not set. While these are not set entering **3** to Evaluate a function will just return you to the initial Main Menu.

To set the evaluation limits enter select **1** on the Main Menu. For example the program can be set to evaluate between x=-2.0 and x=2.0 at 21 points. The program first prompts for the lowest and highest x ordinate values. Enter **-2.0** for the lowest and **2.0** for the highest. Then enter **21** for the number of x evaluations. You are the returned to the Main Menu. If you make a mistake in these values just select **1** on the Main Menu to run through setting the evaluation limits again.

To set the function to evaluate enter **2** at the Main Menu. You can then choose one of the five functions by entering the corresponding choice, say **1** to choose the cosine function. You are the returned to the Main Menu.

Now select **3** on the main menu and the cosine function is evaluated at 21 points between x=-2.0 and x=2.0.

You can then go on and set the program to evaluate other functions, or use other ordinate ranges.

Running the program to investigate $\arcsin(x)$ and other functions

Now set the program to evaluate the $\arcsin$ function by selecting **2** from the Main menu, and **2** from the Function menu. Select **3** on the Main menu to evaluate.

In this case where x is outside of the range $[-1.0 : 1.0]$ the $\arcsin$ function can not be evaluated. When x is out of range the value returned by $\arcsin(x)$ is 'nan', i.e. not a number

Note: in a program once a variable has become 'nan' all calculations that depend on this variable also become 'nan'. Likewise if a function returns 'inf' (infinity) all other calculations using this are also 'inf'.

Now investigate all the other functions. Pay attention to the circumstances that give rise to 'nan' or 'inf' values. When evaluating $\exp(x)$ large numbers, say x from 100 to 1000, are needed.

Events triggering actions

This simple program is controlled by the events of the user inputting values at a prompt. In a Graphical User Interface there are many other ways of entering numbers and selecting icons to make the program perform calculations or tasks. However the same basic principle holds of the program waiting for the event of a selection and prompt before continuing based on what was selected.

Interaction with hardware devices such as Oscilloscopes or digital data bus connected devices is very similar - the program waits for some data and/or a prompt from the device and then acts on it. Some programs will continuously scan for inputs from devices. When the devices have some data or instruction ready for the main program the main program receives this data - the event - and then processes based on the received data.