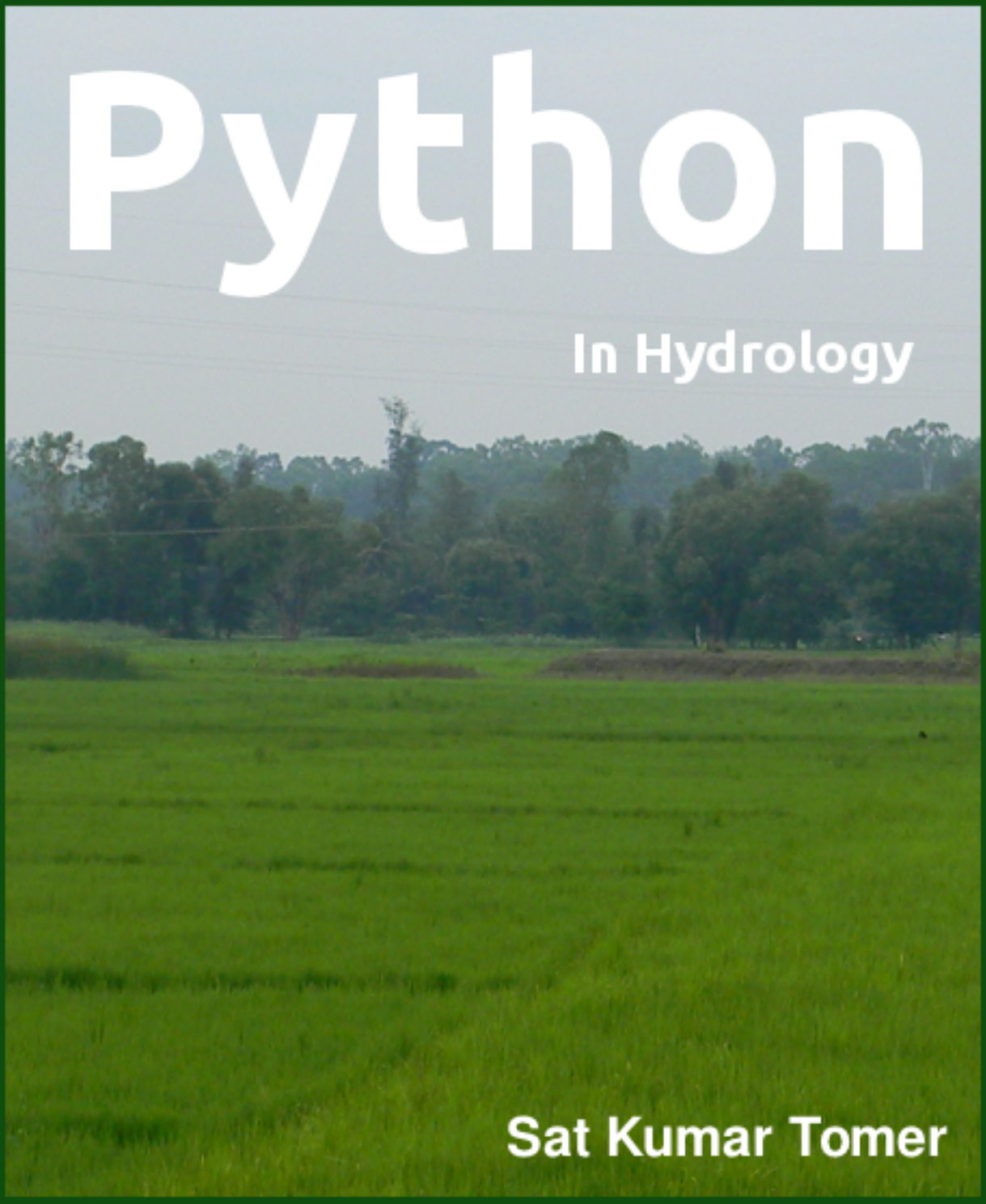

Green Tea Press

Python

In Hydrology

Version 0.0.0

Sat Kumar Tomer

Printing history:

November 2011: First edition of *Python in Hydrology*.

The original form of this book is LaTeX source code. Compiling this LaTeX source has the effect of generating a device-independent representation of a textbook, which can be converted to other formats and printed.

Preface

History

I started using Python in July 2010. I was looking for a programming language which is open source, and can combine many codes/modules/software. I came across Python and Perl, though there might be many more options available. I googled the use of Python and Perl in the field of general scientific usage, hydrology, Geographic Information System (GIS), statistics, and somehow found Python to be the language of my need. I do not know if my conclusions about the Python versus Perl were true or not? But I feel happy for Python being my choice, and it is fulfilling my requirements.

After using Python for two-three months, I become fond of open source, and started using only open source software, I said good-bye to Windows, ArcGis, MATLAB. And even after one year, I do not miss them. I felt that I should also make a small contribution into the free world. I tried to contribute in few areas, but could not go ahead because of my less knowledge in those areas. After spending extensive one year with Python, I thought to make some contribution into Python world. I wrote few small packages for the Python, and started writing this book.

I always have been scared of reading books, especially those having more than 200 pages. I do not remember if I have read any book completely which had more than 200 pages. Though the list of books, that I have read is very small, even for the books which had pages less than 200. I do not like much of the text in the book, and like to learn from examples in the book. I am a student, not a professor, so does not have idea about what students like except my own feeling which I know many of my fellow students do not like.

I hope that you will find this book helpful and enjoyable.

Sat Kumar Tomer
Bangalore, India

Sat Kumar Tomer is a Phd Student in the department of civil engineering at Indian Institute of Science, Bangalore, India.

Acknowledgements

I am thankful to you for reading the book, and hope to receive feedback from you.

I am thankful to Allen B. Downey who provided the latex source code of his books, which helped me in formatting the book in better way. Apart from it, he has written a nice history for his book titled 'Think Python: How to Think Like a Computer Scientist', which encouraged me for writing this book. And, I **copied** many sentences from his book, mainly the basic about Python.

I am thankful to the Free Software Foundation for developing the GNU Free Documentation License.

I am thankful to Green Tea Press.

I am thankful to my friends Aditya Singh, Ganjendara Yadav, Himanshu Joshi, Katarina Turcekova, Shadi Davarian, and Shobhit Khatyan, for their continuous support.

Contributor List

If you have a suggestion or correction, please send email to `satkumartomer@gmail.com`. If I make a change based on your feedback, I will add you to the contributor list (unless you ask to be omitted).

If you include at least part of the sentence the error appears in, that makes it easy for me to search. Page and section numbers are fine, too, but not quite as easy to work with. Thanks!

Contents

Chapter 1

Getting started

1.1 Why Python?

Python is a simple and powerful programming language. By simple I mean, that it is much more forgiving than languages like C though slow also. By powerful, I mean it can glue many existing code which were written in C, C++, Fortran etc. easily. This has a growing user community which makes many tools easily available. Python Package Index which is a major host of the Python code, has more than 15,000 packages listed, which speaks about it popularity. Use of Python in hydrology community is not so fast as compared to other field, but now a days many new hydrological packages are being developed. Python provides access to a nice combination of GIS tools, Mathematics, Statistics etc., which make it a useful language for the hydrologist. Following are the major advantages of Python for the hydrologist:

1. A clear and readable syntax.
2. Modules can be easily written in C, C++.
3. It can be used on all major platforms (Windows, Linux/Unix, Mac etc.)
4. Easy to learn.
5. And it is free.

1.2 Python Installation

Usually all the Linux/Unix system has Python by-default. If it is not there, or for non-linux/unix users, the basic version of Python can be downloaded by following the instructions provided below. The basic Python version includes minimal packages required to run the python, and some other additional packages. For most of the hydrological applications, these packages are not enough, and we require additional packages. In the next section, I will be describing how to install additional packages. Throughout the book, the chevron (>>>) represents the Python shell, and `$` represents the Unix/Linux shell or window's command line. The installation of Python for the various operating system is done in the following way.

1.2.1 Ubuntu/Debian

In the Ubunt/Debian, the Python is installed by running the usual installation command, i.e.:

```
$ sudo apt-get install python
```

1.2.2 Fedora

On Fedora, the installation of the Python is performed in the following way.

```
$ sudo yum install python
```

1.2.3 FreeBSD

Python is installed on FreeBSD by running:

```
$ sudo pkg_add install python
```

The Linux user might be familiar with `sudo`. It allows user to run programs with the security privileges of root or administrator. Window user can ignore `sudo`, as they do not need to specify this.

1.2.4 Windows

For Windows users, the suitable version of Python can be downloaded from `http://www.python.org/getit/`. It provides a .msi file, which can be easily installed by double clicking on it.

1.2.5 Mac OS

Mac OS users also can download a suitable version of Python from `http://www.python.org/getit/` and install it.

1.3 Install additional packages

Pip is a useful program to install additional packages in Python. Before installing pip, `distribute` should be installed. To do so, first we need to download `Distribute`, which is done by downloading `distribute_setup.py` file from `http://python-distribute.org/distribute_setup.py`, and running the following command:

```
$ sudo python distribute_setup.py
```

Now download the `get-pip.py` from `https://github.com/pypa/pip/raw/master/contrib/get-pip.py`, and run as root:

```
$ sudo python get-pip.py
```

Note, that the `distribute_setup.py` and `get-pip.py` should be in your current working directory while installing, otherwise give the full path name of file. If you did not get any error in this procedure, `pip` should be ready to install new packages. The procedure to install packages is simple e.g. suppose you want to install package `SomePackage`, then run the following command:

```
$ sudo pip install SomePackage
```

In hydrology, frequently used packages are `Numpy`, `Scipy`, `xlrd`, `xlwt`, and `gdal`, so these should be installed at this stage. Later whenever a new package/library will be needed, instructions to download them will be given there.

```
$ sudo pip install Numpy
$ sudo pip install Scipy
$ sudo pip install xlrd
$ sudo pip install xlwt
$ sudo pip install gdal
```

These packages/libraries can be installed by specifying all packages name in one line, i.e. `$ sudo pip install Numpy Scipy xlrd xlwt gdal`. But, at this time it is better to install them in seperate line, so that if you get some error, you can easily find out which pakcage is giving error. Most common problem with `pip` is that, it is not able to download library/package from internet. In that case, you can download `*.tar.gz` library using internet browser, and then run the `pip` in the following way:

```
$ sudo pip install /path/to/*.tar.gz
```

Additionally, window user can download `*.exe` or `*.msi` file if available and then download by double clicking it.

If this also fails then, as a last option, you can download `*.tar.gz` file and extract it. Then, go to the folder where you have extracted the file. You should see `setup.py` file there. Now, run the following command:

```
$ sudo python setup.py install
```

If you see any error in this, which could possibly come because of some dependent package/library is not available in your computer. You should read the `README` file provided with the package, it can give you details of required package/library, and how to install them.

The package/libraries are upgraded using the pip in the following way.

```
$ sudo pip install --upgrade some_package
```

1.4 Interactive Development Environment

Simple text editors can be used to write Python programs, but these do not provide options for easy formatting of text, debugging options etc. IDE (Interactive Development Environment) provides many options to quickly format the program in Python way, and easily debugging them. There are

various IDE available for use e.g. PyDev, Spyder, IDLE, and many many others. A list of them can be found at `http://wiki.python.org/moin/IntegratedDevelopmentEnvironments`. I am using Spyder for my work, which is similar to MATLAB. The reason to use Spyder was since earlier I used to work on MATLAB, and Spyder is similar to it, and I found myself to be familiar with it. However you can use any IDE, and after being familiar, it doesn't matter which one you use.

1.5 Execute the program

Python is an interpreted language as the programs are executed by an interpreter. There are two ways to use the interpreter: **interactive mode** and **script mode**. In the interactive mode, you type Python programs (after invoking the python, which is done by typing `python` in a terminal or command window) and interpreter prints the result, e.g. we do `1+1` in it.

```
>>> 1 + 1
2
```

The chevron, `>>>`, is the **prompt** which interpreter uses to indicate that it is ready. If you type `1 + 1`, the interpreter replies `2`. Alternatively, you can store code in a file and use the interpreter to execute the contents of the file, which is called a **script**. By convention, Python scripts have names that end with `.py`. Suppose you have named you script as `myscript.py`, and you want to execute it, in a Unix/Linux shell, you would do:

```
$ python myscript.py
```

or, you can give your script executable permission and simply run the script. The syntax to do is:

```
$ chmod +x myscript.py
$ ./myscript.py
```

In IDE's, the details of executing scripts are different. You can find instructions for your environment at the Python website `python.org`.

Working in interactive mode is convenient for testing small pieces of code because you can type and execute them immediately. But for anything more than a few lines, you should save your code as a script so you can modify and execute it in the future.

1.6 Type of errors

Three kinds of errors can occur in a program: syntax errors, runtime errors, and semantic errors. It is useful to distinguish between them in order to track them down more quickly.

1.6.1 Syntax errors

Python can only execute a program if the syntax is correct; otherwise, the interpreter displays an error message. **Syntax** refers to the structure of a program and the rules about that structure. For example, parentheses have to come in matching pairs, so `(1 + 2)` is legal, but `8)` is a **syntax error**.

1.6.2 Runtime errors

The second type of error is a runtime error, so called because the error does not appear until after the program has started running. These errors are also called **exceptions** because they usually indicate that something exceptional (and bad) has happened.

1.6.3 Semantic errors

The third type of error is the **semantic error**. If there is a semantic error in your program, it will run successfully in the sense that the computer will not generate any error messages, but it will not do the right thing.

1.7 The first program

Traditionally, the first program you write in a new language is called "Hello, World!" because all it does is display the words, "Hello, World!". In Python, it looks like this:

```
>>> print 'Hello, World!'
```

This is an example of a `print` statement, which doesn't actually print anything on paper. It displays a value on the screen. In this case, the result is the words:

```
Hello, World!
```

The quotation marks in the program mark the beginning and end of the string.

Chapter 2

A bit of Python

Before jumping into application of Python into hydrology, which would involve writing many lines of coding, manipulating arrays, etc. It is better to learn some basics of Python, e.g. the types of data, looping (performing same task many times), and writing functions etc. First thing to know is the type of data.

2.1 Data type

There are basically two types of data; numbers and strings. The `type` function returns the type of data.

2.1.1 Numbers

There are three types of number in Python: integers, floating point and complex numbers. Integers are needed for indexing the arrays (vector, matrix), for counting etc. In Python there is no need to define the variable type a priori, and it is allowed to even change the data type later in the program, wherever needed.

```
>>> a = 5
>>> type(a)
<type 'int'>
```

This means that, the data type is integer. The lines in the program without chevron (>>>) represents the output by the Python. Another most commonly used data type is float. Most of the hydrological variables belongs to this category of data type.

```
>>> b = 7.8
>>> type(b)
<type 'float'>
```

This means the data type is floating point. Another data type is complex, which is not frequently needed in day to day hydrological life.

```
>>> c = 5 + 2j
>>> type(c)
<type 'complex'>
```

The `c` represents the complex data type.

2.1.2 Strings

A string is a sequence of characters. There are three way to specify a string.

single quotes: The text written inside single quotes is treated as string by Python.

```
>>> foo = 'my name is Joe'
>>> print(foo)
my name is Joe
```

double quotes: Double quotes are also used to define a string. If single quotes are able to define why is double quotes needed? Let us try to write `What's your name?` using single quotes.

```
>>> foo = 'what's your name?'
File "<stdin>", line 1
    foo = 'what's your name?'
                ^
SyntaxError: invalid syntax
```

This produces SyntaxError. Let us try using double quotes.

```
>>> foo = "what's your name?"
>>> print(foo)
what's your name?
```

Double quotes provide an easy way to define strings which involve single quotes. However, the same task can be performed using single quote also. The same string can be written using single quote only by using the `\` before `'` .

```
>>> foo = 'what\'s your name?'
>>> print(foo)
what's your name?
```

triple quotes: When the strings spans over more than one line, triples quotes are best to define them. Multi-line strings can also be specified using escape sequence `\n` in single or double quote strings, triple quotes make it easier to write. Triple quotes are useful for other things (making help content for functions) also, which you will read later in the book.

```
>>> foo = """My name is Sat Kumar.
... I am in PhD """
>>> print foo
My name is Sat Kumar.
I am in PhD
```

2.2 Data structures

Data structures are able to contain more than one data in it. There are four built-in data structures in Python: `list`, `tuple`, `dictionary` and `set`. Apart from these built-in data structure, you can define your own data type also like `numpy.array` defined by `numpy`, which is very useful. I did not feel any need to use the set in hydrology, so I am skipping set here, if you are interested you can learn about it from some other source.

2.3 Data structures

2.3.1 List

A list is a sequence of items (values). The items in it could belong to any of data type, and could be of different data type in the same list.

```
>>> a = ['Ram', 'Sita', 'Bangalore', 'Delhi']
>>> b = [25, 256, 2656, 0]
>>> c = [25, 'Bangalore']
```

The items in the list are accessed using the indices. The variable `a` and `b` hold items of similar data types, while `c` holds items of different data types. In Python, the indices starts at `0`. So, to get the first and third item, the indices should be 0 and 2.

```
>>> a = ['Ram', 'Sita', 'Bangalore', 'Delhi']
>>> print a[0]
Ram
>>> print a[2]
Bangalore
```

Negative indices are also allowed in Python. The last item in the list has `-1` indices, similarly second last item has indices of `-2` and so on.

```
>>> a = ['Ram', 'Sita', 'Bangalore', 'Delhi']
>>> print a[-1]
Delhi
```

Likewise, second last item in the list can be accessed by using the indices `-2`.

2.3.2 Dictionary

In the list, the indices are only integers. Dictionary has the capability to take any data type as indices. This feature of dictionary makes it very suitable, when the indices are name etc. For example, in hydrology the name of field stations and their corresponding variables are given for each station. Let us try to retrieve the value of variable by using list first, and then by using dictionary. We can use one list to store the name of stations, and one for the variable. First, we need to find the indices of station, and then use this indices to access the variable from the list of variables.

```
>>> names = ['Delhi', 'Bangalore', 'Kolkata']
>>> rainfall = [0, 5, 10]
>>> print(rainfall[ind])
5
```

Now, let us try this using dictionary.

```
>>> rainfall = {'Delhi':0, 'Bangalore':5, 'Kolkata':10}
>>> rainfall['Bangalore']
5
```

The same thing could have been done using list in one line, but dictionary provides a neat and clean way to do this.

2.3.3 Tuple

A tuple is a sequence of values, similar to list except that tuples are immutable (their value can not be modified).

```
>>> foo = 5,15,18
>>> foo[2]
5
>>> foo[1] = 10
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

While trying to modify the items in the tuple, Python issues an error. Tuples are useful there is a need to specify some constants, and to make sure that these constants do not change. The immutable property of tuples ensures that during executions of the program the value of constants will not change.

A tuple having only one item is defined by using the `','` after this, e.g. :

```
>>> foo = 5
>>> type(foo)
<type 'int'>
>>> foo = 5,
>>> type(foo)
<type 'tuple'>
```

You might have noticed that without using the comma (’), Python does not take it as tuple.

2.3.4 Numpy.array

NumericalPython (NumPy) is a library/package written mainly in C programming language, but application programming interface (API) is provided for Python. The library provided `numpy.array` data type, which is very useful in performing mathematical operation on array. It is the type of data, that we would be dealing most of the time. This library is not a part of the standard Python

distribution, hence before using this, NumPy have to be installed in the system. We can check if NumPy is installed in our system or not, by using the following command:

```
$ python -c'import numpy'
```

If this command gives no output (no error), then it means that NumPy is installed. If NumPy is not installed in the system, you will see some message (error) like following:

```
$ python -c'import numpy'
Traceback (most recent call last):
  File "<string>", line 1, in <module>
ImportError: No module named numpy
```

This means, that numpy is not installed in the system. You can install NumPy by following the steps provided in the section 1.3. The python -c'import numpy' is a way to run some simple code without invoking the python. This is useful when you want to do something small, quickly. This is very helpful when you want to check if some package is installed or not in your system.

Before using any library, it should be imported into the program. The import can be used to import the library. There are three ways to import a complete library or some functions from the library. By importing complete library.

```
>>> import numpy
>>> x = [1, 2, 5, 9.0, 15]  # list containing only numbers (float or integers)
>>> type(x)
<type 'list'>
>>> x = numpy.array(x) # convert the list into numpy array
>>> type(x)
<type 'numpy.ndarray'>
```

We imported the complete library numpy, and after doing so, whenever we need any function (i.e. array) from this library, we need to provide name along with the name of library (i.e. numpy.array). The array function converts a list of integers or/and float into numpy array. Often the library name are quiet long, and it can be abbreviated using as in the following manner.

```
>>> import numpy as np
>>> x = np.array(x) # convert the list into numpy array
>>> type(x)
<type 'numpy.ndarray'>
```

If only few functions are needed then they can be imported by explicitly defining their name.

```
>>> from numpy import array
>>> x = array(x) # convert the list into numpy array
>>> type(x)
<type 'numpy.ndarray'>
```

If all the functions are needed, and you do not want to use numpy or np before them, then you can import in the following way.

```
>>> from numpy import *
>>> x = array(x) # convert the list into numpy array
>>> type(x)
<type 'numpy.ndarray'>
```

Anything written after # is comment for program, and Python does not execute them. Comments are useful in making your code more readable. The comments can be in full line also. A numpy array is a homogeneous multidimensional array. It can hold only integer, only float, combination of integers and float, complex numbers and strings. If combination of integers and float are specified in numpy.ndarray, then integers are treated as floats. The data type of numpy.ndarray can be checked using its attribute `dtype`.

```
>>> import numpy as np
>>> x = np.array([1,5,9.0,15]) # np.array can be defined directly also
>>> x.dtype
dtype('float64')
>>> x = np.array([1,5,9,15]) # this is holding only integers
>>> x.dtype
dtype('int64')
>>> x = np.array(['Delhi', 'Paris']) # this is holding strings
>>> x.dtype
dtype('|S5')
```

The mean of the array can be computed using method `mean`, in the following manner.

```
>>> import numpy as np
>>> x = np.array([1,5,9.0,15])
>>> x.sum()
30.0
```

Did you notice the difference between calling attributes and methods? The methods perform some action on the object, and often action needs some input, so methods are called with brackets `()`. If there is some input to be given to method, it can be given inside brackets, if there is no input then empty brackets are used. Try using the methods (e.g. sum) without giving the bracket, you will see only some details about it, and no output.

As Python is object oriented programming (OOP) language, and attributes and methods are used quiet commonly. It is better to know briefly about them before jumping into Python. `Attributes` represent properties of an object, and can be of any type, even the type of the object that contains it. `methods` represent what an object can do. An attribute can only have a value or a state, while a method can do something or perform an action.

2.4 Choosing the name of variable

Then name of the variables should be meaningful, and possibly should be documented what the variable is used for.

Variable names can be arbitrarily long. They can contain both letters and numbers, but they have to begin with a letter. It is legal to use uppercase letters, but it is a good idea to begin variable names with a lowercase letter, as conventionally uppercase letters are used to denote classes.

The underscore character (_) can appear in a name. It is often used in names with multiple words, such as `my_name` or `airspeed_of_unladen_swallow`. A variable name can be started with underscore, but should be avoided because this is used for something else conventionally.

Python has some of the reserved keywords, which can not be used as variable name. If the interpreter gives some error about one of your variable names and you don't know why, you should check if your variable name is not in the reserved keyword list. It is a good idea to remember the list, or keep it handy. But I, being a lazy person, do not remember this list, and in fact even never tried to remember. I just type the name of variable, I want to use in python, and look for the output by python, and then decide whether to use this name for variable or not.

```
>>> 500_mangoes
  File "<stdin>", line 1
    500_mangoes
              ^
SyntaxError: invalid syntax
>>> class
  File "<stdin>", line 1
    class
        ^
SyntaxError: invalid syntax
>>> np
<module 'numpy' from '/usr/lib/pymodules/python2.7/numpy/__init__.pyc'>
>>> foo
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'foo' is not defined
```

First variable name `500_mangoes` gives `SyntaxError`, so we cant use this name as variable. The `class` gives the `SyntaxError` too, so it also can not be used. The `np` gives some output (which means np is referring to something), so if we use this as variable name, the reference will be destroyed. The `foo` gives `NameError` that the variable is not defined, this makes it a valid choice for the variable name. Apart from these scenarios, one more output is possible.

```
>>> a = 5
>>> a
5
```

This means that variable `a` is defined before, now it is upto you, if you want to change its old value.

2.5 Operators and operands

Operators are special symbols that represent computations like addition and multiplication. The values the operator is applied to are called **operands**. Assume variable *a* holds 2 and variable *b* holds 5.

Operator	Name	Example
+	Plus	>>> a+b 7
-	Minus	>>> a-b -3
*	Multiply	>>> a*b 10
/	Divide	>>> a/b 0 (for Python 2.x) 0.4 (for Python 3.x)
**	Power	>>> a**b 32
%	Modulo	>>> b\%a 1
==	Equal to	>>> a==b False
<	Less than	>>>a<b True
>	Greater than	>>>a>b False
<=	Less than or Equal to	>>>a<=b True
>=	Greater than or Equal to	>>>a>=b False
!=	Not equal to	>>>a!=b True
and	And	>>> True and False False
or	Or	>>> True or False True
not	Not	>>> not True False
+=	Add AND assignment	>>> a += b 7
-=	Subtract AND assignment	>>> a -= b 3
*=	Multiply AND assignment	>>> a *= b 10
/=	Divide AND assignment	>>> a /= b 0
%=	Modulus AND assignment	>>> a %= b 2
**=	Exponent AND assignment	>>> a **= b 32
//=	Floor division AND assignment	>>> a //= b 0

2.6 Expressions

An **expression** is a combination of values, variables, and operators. A value all by itself is considered an expression, and so is a variable, so the following are all legal expressions (assuming that the variable `x` has been assigned a value:

```
>>> x = 17
>>> x + 15
32
```

If you type an expression in interactive mode, the interpreter **evaluates** it and displays the result:

```
>>> 1 + 1
2
```

But in a script, an expression all by itself doesn't do anything! This is a common source of confusion for beginners.

2.7 Control Flow

If we want to do same task many times, restrict the execution of task only when some condition is met, **control flow** is the way to do it.

2.7.1 for

`for` is used to repeatedly execute some code. It also can be used to iterate over some list. Suppose you have some list, and you want to square the each item in list and print.

```
>>> foo = [5, 11, 14, 0, 6, 0, 8] # define the list
>>> for item in foo:
...     print item**2
...
25
121
196
0
36
0
64
```

The item in the list can be iterated by defining another list having integers, and iterating over it. Let us try this way to perform the above task.

```
>>> foo = [5, 11, 14, 0, 6, 0, 8] # define the list
>>> a = range(7) # define the list having integers
>>> a
[0, 1, 2, 3, 4, 5, 6]
>>> for item in a:
...     print foo[item]**2
...
```

```
25
121
196
0
36
0
64
```

2.7.2 while

`while` statement is used, when we want to keep on executing the some code unless some condition is met or violated. Suppose, we want to print some numbers ranging from 15 to 20, we could do like this.

```
>>> n = 15 # initialize the n
>>> while n<=20:
...     print n
...     n = n+1
...
15
16
17
18
19
20
```

2.7.3 if

`if` statement execute some portion of the code, if the conditions are met, otherwise it skips that portion. Suppose you have some list, and you want to compute its inverse, but want to skip if the entry in list is zero:

```
>>> foo = [5, 11, 14, 0, 6, 0, 8] # define the array
>>> for item in foo:
...     if item is not 0:
...         print 1.0/item
...
0.2
0.0909090909091
0.0714285714286
0.166666666667
0.125
```

The `if-else` allows alternative portions of code to execute depending upon the condition. In `if-else` only one portion of code is executed from the given alternatives. Suppose in the previous example, you want to issue some statement when there is 0 in the list.

```
>>> foo = [5, 11, 14, 0, 6, 0, 8] # define the array
>>> for item in foo:
...     if item is not 0:
...         print 1.0/item
...     else:
...         print '0 found in list'
...
0.2
0.0909090909091
0.0714285714286
0 found in list
0.166666666667
0 found in list
0.125
```

`if-elif-else` is used when depending upon the condition, you want to execute some portion of code. You can specify as many conditions you want, and their corresponding code to execute. Lets take one example, suppose we have one list, and we want to print some statement if the item in list is negative, positive or 0.

```
>>> foo = [5, -11, 14, 0, -6, 0, 8] # define the array
>>> for item in foo:
...     if item < 0:
...         print 'item is negative'
...     elif item>0:
...         print 'item is positive'
...     else:
...         print 'item is 0'
...
item is positive
item is negative
item is positive
item is 0
item is negative
item is 0
item is positive
```

2.7.4 break

The `break` statement, breaks out of the loop. Suppose you want to print all the items in the list, but you want to stop the loop if you encounter 0.

```
>>> for item in foo:
...     if item==0:
...             print('zero found in the list, stopping iterations')
...             break
...     else:
...             print(item)
...
5
```

```
-11
14
zero found in the list, stopping iterations
```

The `break` statement becomes useful when you want to want if something strange happens to your program, and in that condition you want to stop the execution.

2.7.5 continue

The `continue` statement provides opportunity to jump out of the current loop (iteration) when some condition is met. Suppose you do not want to print items in the list which are negative.

```
>>> foo = [5, -11, 14, 0, -6, 0, 8] # define the array
>>> for item in foo:
...     if item<0:
...         continue
...     print item
...
5
14
0
0
8
```

2.7.6 pass

The `pass` statement does nothing. It can be used when a statement is required syntactically but the program requires no action.

```
>>> foo = [5, -11, 14, 0, -6, 0, 8] # define the array
>>> for item in foo:
...     pass
...
```

This is often used, when you are developing your code, and intent to put something later. If you leave without `pass`, Python will issue the error.

2.8 Function

Function is a some sequence of statements that does some processing. When you define a function, you specify the name and the sequence of statements. Later, you can call the function by name. There are many built in functions in the Python, and each library provides some functions. You can also specify your functions. When you need to do some task many times, it is better to define function to do that task, and later call the function. The thumb rule is that, whenever you feel to define one function define it.

2.8.1 In-built functions

Python has some in-built functions, some of them we have already used.

```
>>> foo = [5, -11, 14, 0, -6, 0, 8] # define the array
>>> type(foo)
<type 'list'>
>>> len(foo)
7
>>> max(foo)
14
```

Here, `type`, `len`, `max` are in-built functions, which returns the `type`, `length` of the list, and `maximum` value in the list respectively.

2.8.2 User defines functions

If you do not find the function that you intent to use, you can define one. In fact, it is a good practice to define functions whenever they are needed, it increase the readability of the codes. A function definition specifies the name of a new function and the sequence of statements that execute when the function is called.

Suppose, you want to define a function which adds the 2 into the input.

```
>>> def add2(temp):
...     return temp+2
...
>>> add2(5)
7
>>> foo = np.array( [5, -11, 14, 0, -6, 0, 8]) # define the array
>>> new_foo = add2(foo)
>>> print new_foo
array([ 7, -9, 16,  2, -4,  2, 10])
```

The `return` defines the output from the function. `return` is optional, some functions may not return anything.

```
>>> foo = [5, -11, 14, 0, -6, 0, 8] # define the array
>>> def add2(temp):
...     print temp[-1]+2 # add 2 to only the last entry in the list
...
>>> add2(foo)
10
>>> new_foo = add2(foo)
10
>>> new_foo
```

This is clear from this example that functions need not to return any values. Like in this example, function only print the last entry of the list after adding 2 to it, and returns none.

2.9 Plotting

There are various library which provides plotting capabilities in Python. I liked `Matplotlb` library, and it is installed in the following manner.

```
$ sudo pip install matplotlib
```

Let us make our first plot which plots y versus x. The x contains values between 0 and 2π with 50 intervals, and y is the sin of x.

```
>>> # import the required modules
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> # generate x over the range from 0 to 2 pi with 50 intervals
>>> x = np.linspace(0,2*np.pi,50)
>>> y = np.sin(x) # compute the sin of x
>>> plt.plot(x,y)
>>> plt.xlabel('x')
>>> plt.ylabel('y')
>>> plt.legend(['sin(x)'])
>>> plt.show()
```

The `plt` is the given abbreviated name, which refers to the `matplotlib.pyplot` library. All the function of this library should be called by using `plt.` while using them. The `plot` makes the continuous line plot, `xlabel` puts the label for the x-axis, and `ylabel` puts the label for y-axis. The `legend` displays the legend on the graph. `show()` displays the graph, the graph can be save by using the `savefig` and can be closed by using the `close()`. Fig. 2.1 shows the plot of y versus x. The function `np.linspace` is used to generate vector over the range 0 to 2π having 50 equally spaced elements. More on generating this kind of vectors is given in the next chapter.

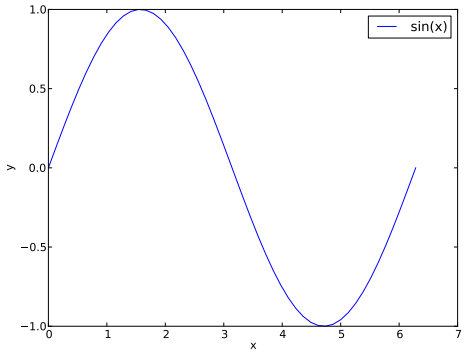

Figure 2.1: The first plot which shows $y = \sin(x)$ versus x.

Chapter 3

Array

3.1 Generating sequential arrays

Often we need vectors whose elements follow a simple order, for example a vector containing elements [10, 11, 12, 13] or [5, 10, 15, 20] or [1.0, 1.2, 1.4, 1.6, 1.8, 2.0]. We see that in these vectors, items follow some simple order, so it would be nicer if there are easy way to define these kinds of vectors. Some of the way to create these vectors are following:

3.1.1 linspace

If we are interested in generating the vector, whose elements are uniformly spaced and we know the upper, lower limit and the number of elements, then in that case `linspace` is the preferred choice.

```
>>> import numpy as np
>>> np.linspace( 0, 2, 9 )
array([ 0.  ,  0.25,  0.5 ,  0.75,  1.  ,  1.25,  1.5 ,  1.75,  2.  ])
```

Because `linspace` lies in `numpy` library, so first we have imported the library and have given it an abbreviated name. Then we call the `linspace` with lower limit, upper limit and the number of element to be generated. In this example, 0 is the lower limit, 2 is the upper limit, and number of elements are 9. Let us generate one more vector to understand more about this function, this time we take lower limit as 0, upper limit as 2π, and number of elements to be 100.

```
>>> x = np.linspace( 0, 2*pi, 100 )
```

By default the number of elements are 50, so if we do not specify the number of elements, we get 50 elements with equal spacing. We can use `len` function to get the length of any array.

```
>>> foo = np.linspace(0,1)
>>> len(foo)
50
```

3.1.2 arange

Suppose again we want to generate a vector whose elements are uniformly spaced, but this time we do not know the number of elements, we just know the increment between elements. In such situation `arange` is used. `arange` also requires lower and upper bounds. In the following example we are generating the vector having lower element as 10, upper element as 30 and having an increment of 30. So from the knowledge of `linspace` we will do something like this.

```
>>> np.arange( 10, 30, 5 )
array([10, 15, 20, 25])
```

Oh! What happened? Why did Python not print 30. Because `arange` function does not include second argument in the elements. So we want to print upto 30, we would do.

```
>>> np.arange( 10, 31, 5 )
array([10, 15, 20, 25, 30])
```

This time we get the required output. The `arange` can also take a float increment. Let us generate some vector with lower bound of 0, upper bound of 2 and with an increment of 0.3.

```
>>> np.arange( 0, 2, 0.3 )                 # it accepts float arguments
array([ 0. ,  0.3,  0.6,  0.9,  1.2,  1.5,  1.8])
```

In the case of float increment also, the maximum value of generated elements is lesser than the second argument given to the `arange`.

3.1.3 zeros

`zeros` is used when we want to generate all the items in vector as 0.

```
>>> foo = np.zeros(5)
>>> foo
array([ 0.,  0.,  0.,  0.,  0.])
```

3.1.4 ones

`ones` is used when all the required elements in vector are 1. Let us say, we want to generate a variable `foo` which has all the elements equal to one, and has the dimension of 3×2.

```
>>> foo = np.ones((3,2))
>>> foo
array([[ 1.,  1.],
       [ 1.,  1.],
       [ 1.,  1.]])
```

Remember that if the number of dimensions are more than one, the dimension are given as `tuple`, e.g. `(2,5)`.

3.1.5 empty

`empty` is useful in initializing the variables. This assigns the garbage values to the elements, which are to be modified later.

```
>>> foo = np.empty((2,5))
>>> foo
array([[  6.94573181e-310,   2.76947193e-316,   2.74957018e-316,
          0.00000000e+000,   0.00000000e+000],
       [  0.00000000e+000,   0.00000000e+000,   6.94573152e-310,
          6.34874355e-321,   0.00000000e+000]])
```

Additionally in `zeros, ones, empty`, the data type (e.g. int, float etc.) also can be defined.

```
>>> foo = np.empty((2,5),int)
>>> foo
array([[     140583176970856,             56931856,             59487840,
        -3617040655747907584,                    0],
       [                   0,                    0,      140583171090560,
                        1285,                    0]])
```

You can see that all the elements of `foo` are now integer, even though the values are useless.

3.1.6 rand

`rand` is used to generate uniformly distributed random variables over the range of 0 to 1.

```
>>> foo = np.random.rand(3,2)
>>> foo
array([[ 0.75644359,  0.07754619],
       [ 0.50267515,  0.91460249],
       [ 0.85992345,  0.58662329]])
```

3.1.7 randn

`randn` is used to generate random variable having normal distribution with mean equal to zero and variance equal to one.

```
>>> foo = np.random.randn(2,4)
>>> foo
array([[-0.66317015, -1.80129451,  0.56398575, -1.11912727],
       [ 0.19218091,  0.21957804, -1.10891128, -0.87418933]])
```

3.2 Useful attributes and methods

The `ndarray` (array generated using numpy) provides attributes to perform commonly used task quickly. These attributes are used to quickly get properties of ndarray. So let us first generate some vector whose elements are normally distributed random numbers, and try these attributes. Here I

am using normally distributed random variable to demonstrate, but these attributed can be used with any numpy array. We are generating a 2 dimensional vector of size 5×100.

```
>>> foo = np.random.randn(5,100)
```

Let us check the number of dimension (not the size, or shape of the array). Number of dimension means how many dimensions are associated with array. For example, in mathematics terminology vector has one dimension, matrix has two dimension.

```
>>> foo.ndim
2
```

The dimension of the array is accessed by using `shape` attribute.

```
>>> foo.shape
(5, 100)
```

The `size` attribute provides the total number of elements in the array. This is simply the multiplication of all the elements given by `shape` attributes.

```
>>> foo.size
500
```

The data type (i.e. float, integer etc.) is extracted using the attribute `dtype`.

```
>>> foo.dtype
dtype('float64')
```

This tells us that, the variable `foo` is float, and has 64 bits. The average or mean of the variable is computed by using `mean` method.

```
>>> foo.mean()
-0.11128938014455608
```

This provides the mean of entire array (i.e. 500 elements in this case). Suppose we want to estimate the mean across some dimension say second (1) dimension, then in this case we need to provide additional parameter to `mean`, i.e. axis.

```
>>> foo.mean(axis=1)
array([-0.07311407,  0.0705939 , -0.09218394,  0.0775191 ,  0.01026461])
```

The minimum, maximum, standard deviation and variance of the array are estimated using `min`, `max`, `std`, and `var` methods.

```
>>> # to get the minimum vale
>>> foo.min()
-3.5160295160193256
>>> # to get the maximum value
>>> foo.max()
3.0989215376354817
```

```
>>> # to get the standard deviation
>>> foo.std()
0.9528004743319175
>>> # to get the variance
>>> foo.var()
0.90782874388712709
```

Remember that the line starting with # represents the comments. Comments make it easier to read and understand the code. So put comments whenever you do something, which is not easy to interpret from the code.

The trace of the matrix represent the sum of diagonal elements, and has meaning in case of square matrix. Python even allows to estimate the trace even when matrix is not square, and trace is computed by using the `trace` attributes.

```
>>> foo.trace()
1.081773080044246
```

There are number of attributes associated with each class, `dir` function is a useful tool in exploring the attributes and method associated with any variable, class, library etc. Let us see what all methods and attributes our variable `foo` have.

```
>>> # to get the list of all the attributes associated with foo variable
>>> dir(foo)
['T', '__abs__', ..... 'flat', 'view']
```

The output of `dir(foo)` is very long, and is omitted for brevity. The attributes/method starting with _ are supposed to be the private attributes and are often not needed.

3.3 Indexing

In this section, we will discuss how to refer to some elements in the numpy array. Remember that in Python first indices is 0. We shall generate some array, say some array whose elements are powered to 3 of the sequence [0,1, ..., 9].

```
>>> foo = np.arange(10)**3
>>> foo
array([  0,   1,   8,  27,  64, 125, 216, 343, 512, 729])
```

Print the third item in the array. Third item means we need to put indices as 2.

```
>>> foo[2]
8
```

Suppose, we would like to print some sequence of array, say at indices of 2,3, and 4.

```
>>> foo[2:5]
array([ 8, 27, 64])
```

We used `2:5` to get the values at indices of 2,3 and 4. This is same as saying that `foo[np.arange(2,5,1)]`. When we do not specify the third value in the indices for array, it is by default taken as 1. If we want to print value at 2 to 8, with an interval of 3. Now because the interval is not 1, so we need to define it.

```
>>> foo[2:10:3]
array([  8, 125, 512])
```

If we leave the first entry in the index as blank i.e. to get array elements form the beginning of array with an interval of 2 and upto 6, we issue the following command:

```
>>> foo[:6:2]    # gives the element at 0,2,4
array([ 0,  8, 64])
```

We get element upto the indices of 4, because `arange` does not go upto the second argument. We can use indices also to modify the existing elements in the array, in the same way as we accessed them. Let us replace the existing value of elements at 0,2 and 4 indices, by -1000.

```
>>> foo[:6:2] = -1000                          # modify the elements at 0,2,4
>>> foo
array([-1000,     1, -1000,    27, -1000,   125,   216,   343,   512,   729])
```

We get the last elements of an array by indices `-1`. We can also use this to reverse the array, by giving the increment of -1.

```
>>> foo[::-1]                                  # reversed a
array([  729,   512,   343,   216,   125, -1000,    27, -1000,     1, -1000])
```

We can perform the calculation on entire numpy array at once. Suppose we are interested in estimating the square root of the numpy array, we can use `sqrt` function of numpy library.

```
>>> np.sqrt(foo) # compute the square root
array([         nan,   1.        ,          nan,   5.19615242,
                nan,  11.18033989,  14.69693846,  18.52025918,
        22.627417  ,  27.        ])
Warning: invalid value encountered in sqrt
```

`nan` represents that the element is 'Not A Number'. So when the value of element is negative the output of `sqrt` become `nan`. The Warning issued by Python tells that there were some invalid values in the input for which `sqrt` can not produce any sensible output, and it provides warning (not errors). In reality, the square root of negative number is complex number, but because we did not define the variable as complex, numpy can not perform operations of complex numbers on this. We need library which handles complex number for such situation.

3.4 Array Manipulation

Often we need to change the array, transpose it, get some elements, or change some elements. This is illustrated by this example, in which first we create the array and then play with it. We have already seen in the previous section, that we can change the value of any element by calling it by the

indices, and then assigning new value to it. First, we generate normally distributed random number of size (2×5) to create an array, which we would like to manipulate.

```
>>> foo = np.random.randn(2,3)
>>> foo
array([[ 1.02063865,  1.52885147,  0.45588211],
       [-0.82198131,  0.20995583,  0.31997462]])
```

The array is transposed using `T` attributes.

```
>>> foo.T
array([[ 1.02063865, -0.82198131],
       [ 1.52885147,  0.20995583],
       [ 0.45588211,  0.31997462]])
```

We can access some elements of the array, and if we want, new values also can be assigned to them. In this example, we shall first access element at (0,1) indices, and then we shall replace it by 5. Finally we will print the variable to check if the variable got modified.

```
>>> foo[0,1]
-0.82198131397870833
>>> foo[0,1]=5
>>> foo
array([[ 1.02063865,  5.        ],
       [ 1.52885147,  0.20995583],
       [ 0.45588211,  0.31997462]])
```

The shape of any array is changed by using the `reshape` method. During reshape operation, the change in number of elements is not allowed. In the following example, first we shall create an array having size of (3×6), and the we shall change its shape to (2×9).

```
>>> foo = np.random.randn(3,6)
array([[ 2.01139326,  1.33267072,  1.2947112 ,  0.07492725,  0.49765694,
         0.01757505],
       [ 0.42309629,  0.95921276,  0.55840131, -1.22253606, -0.91811118,
         0.59646987],
       [ 0.19714104, -1.59446001,  1.43990671, -0.98266887, -0.42292461,
        -1.2378431 ]])
>>> foo.reshape(2,9)
array([[ 2.01139326,  1.33267072,  1.2947112 ,  0.07492725,  0.49765694,
         0.01757505,  0.42309629,  0.95921276,  0.55840131],
       [-1.22253606, -0.91811118,  0.59646987,  0.19714104, -1.59446001,
         1.43990671, -0.98266887, -0.42292461, -1.2378431 ]])
```

Like we can access the any elements of the array and change it, in similar way we can access the any attributes, and modify them. However, the modification is only allowed if the attributes is writeable, and the new value makes some sense to the variable. We can use this behaviour, and change the shape of variable using the `shape` attributes.

```
>>> foo = np.random.randn(4,3)
```

```
>>> foo.shape
(4, 3)
>>> foo
array([[-1.47446507, -0.46316836,  0.44047531],
       [-0.21275495, -1.16089705, -1.14349478],
       [-0.83299338,  0.20336677,  0.13460515],
       [-1.73323076, -0.66500491,  1.13514327]])
>>> foo.shape = 2,6
>>> foo.shape
(2, 6)
>>> foo
array([[-1.47446507, -0.46316836,  0.44047531, -0.21275495, -1.16089705,
        -1.14349478],
       [-0.83299338,  0.20336677,  0.13460515, -1.73323076, -0.66500491,
         1.13514327]])
```

In the above example, first an array is defined with a size of (4×3) and then its shape is assigned a value of (2,6), which makes the array of size (2×6). As we can not change the number of elements, so if we define one dimension of the new variable, second dimension can be computed with ease. Numpy allow us to define `-1` for the default dimension in this case. We can make the desired change in the shape of variable by using default dimension also.

```
>>> foo.shape = -1,6
>>> foo.shape
(2, 6)
```

We can flatten the array (make array one dimensional) by using the `ravel` method, which is explained in the following example:

```
>>> foo = np.random.rand(2,3)
>>> foo
array([[ 0.82866532,  0.99558838,  0.58213507],
       [ 0.48877906,  0.67700479,  0.35975352]])
>>> foo.shape
(2, 3)
>>> a = foo.ravel()
>>> a.shape
(6,)
>>> a
array([ 0.82866532,  0.99558838,  0.58213507,  0.48877906,  0.67700479,
        0.35975352])
```

Chapter 4

Basic applications in Hydrology

4.1 Introdution

This chapter will provide applications of python in hydrology. Most of the problems given in this chapter are taken from the book titled "Applied Hydrology" by Chow et al, and for detailed description of them, you should refer to the book. These examples include the equations commonly encountered in the hydrology. I have choose these problems to teach Python by using examples, and additionally in every example we will be learning new things about Python.

4.2 Water Vapor

Approximately, the saturation vapor pressure (e_s) is related to the air temperature (T) by the following equation,

$$e_s = 611 \exp\left(\frac{17.27T}{237.3+T}\right), \tag{4.1}$$

where, e_s is in pascals and T is in degrees Celcius. Let us calculate the value of e_s at $T = 50$.

```
>>> T = 50
>>> es = 611*np.exp(17.27*T/(237.3+T))
>>> print(es)
12340.799081
```

Let us plot the variation of e_s versus T over the range of $-100 \leq T \leq 100$. The `plt.plot(x,y)` makes the line plot of y versus x, with default color of blue. The `plt.xlabel()` and plt.ylabel()" are used to write labels on x and y axis respectively. The input to `xlable` and `ylabel` must be a string, or a variable which contains a string. The `plt.show()` displays the graph on computer screen.

```
>>> import numpy as np
>>> T = np.linspace(-100,100,50)
>>> es = 611*np.exp(17.27*T/(237.3+T))
>>> plt.plot(T,es)
>>> plt.xlabel('T (degree Celcius)')
```

```
>>> plt.ylabel('es (Pa)')
>>> plt.show()
```

The resulted plot is shown in Fig. 4.1. This example demonstrates how to graphically visualize the variation of one variable with respect to the another variable, while former is explicit function of later.

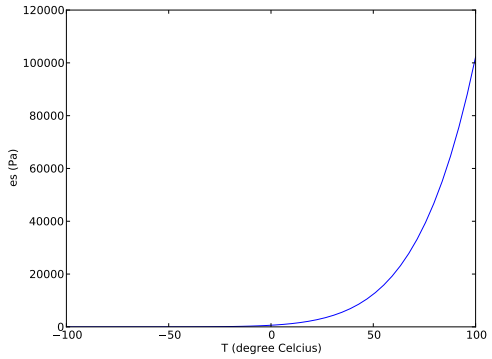

Figure 4.1: The variation of saturation vapor pressure (e_s) versus temperature (T).

4.3 Precipitation

The terminal velocity (V_t) of a falling raindrop is given by:

$$V_t = \left[\frac{4gD}{3C_d}\left(\frac{\rho_w}{\rho_a} - 1\right)\right]^{1/2}, \tag{4.2}$$

where, g is the acceleration due to gravity, D is the diameter of the falling raindrop, ρ_w is the density of water, ρ_a is the density of air, and C_d is the drag coefficient. The *Stoke's law* can be used to calculate drag coefficient $(C_d = 24/Re)$, which is valid for raindrop having diameter less than 0.1 mm. Re is the Reynold number, which can be calculated as $\rho_a VD/\mu_a$. Let us assume, that the Re is given as 5.0, and the raindrop has diameter of 0.05 mm, and we want to estimate the V_t. ($\rho_w = 998$, $\rho_a = 1.2$).

```
>>> import numpy as np
>>> Re = 5.0; rho_w = 998; rho_a = 1.2; g = 9.8; D = 0.05E-3
>>> Cd = 24/Re
>>> Vt = np.sqrt((4*g*D)/(3*Cd)*(rho_w/rho_a-1))
>>> Vt
0.3362483649967134
```

In this example we see that '`;`' allows us to define many expressions in one line.

4.4 Rainfall

Often, we are given a rainfall recorded by a rain gauge which provides the rainfall depths recorded for successive interval in time, and we want to compute the cumulative rainfall. In this example first we shall create rainfall using the random numbers, and we shall also create time variable having values [0,5,10, ..., 100].

```
>>> import numpy as np
>>> time = np.linspace(0,100,21) # create time variable
>>> time
array([   0.,    5.,   10.,   15.,   20.,   25.,   30.,   35.,   40.,
         45.,   50.,   55.,   60.,   65.,   70.,   75.,   80.,   85.,
         90.,   95.,  100.])
>>> rainfall = np.random.rand(21) # generate rainfall
>>> rainfall
array([ 0.08155645,  0.88821997,  0.33355457,  0.49600859,  0.6315054 ,
        0.0722053 ,  0.06165701,  0.96105307,  0.56483934,  0.5194715 ,
        0.35780167,  0.98950575,  0.67866578,  0.31274527,  0.80022389,
        0.53321842,  0.82370443,  0.73212013,  0.77039599,  0.06392391,
        0.53481488])
```

Now we make a bar plot using the `plt.bar()`, for the rainfall which depicts temporal behaviour of the rainfall.

```
>>> import matplotlib.pyplot as plt
>>> plt.bar(time,rainfall)
>>> plt.xlabel('Time')
>>> plt.ylabel('Incremental rainfall')
>>> plt.savefig('/home/tomer/articles/python/tex/images/rain.png')
```

The resulted bar plot of rainfall is shown in Fig 4.2. You might have noticed that in the section 4.2, we used the `plt.show()`, while in the above example we used `plt.savefig`. The `plt.show()` shows the graph on computer screen, which can be saved later, while the `plt.savefig()` saves the graphs in computer, which can be viewed after opening the file. It is just matter of taste, what you like, optionally both can be done on same graph. I prefer to save the figures in the computer and then see them.

The cumulative sum is calculated by using the `cumsum` function of the `numpy` library.

```
>>> cum_rainfall = np.cumsum(rainfall)
```

Now we plot the cumulative rainfall. The resulted cumulative rainfall is shown in Fig. 4.3. The `plt.clf()` clears the current figure, and is quiet useful when making multiples plots, and there is any existing plot in the python memory. Just don't use the `clf` in this, and see the difference.

```
>>> plt.clf()
>>> plt.plot(time,cum_rainfall)
>>> plt.xlabel('Time')
>>> plt.ylabel('Cummulative rainfall')
>>> plt.savefig('/home/tomer/articles/python/tex/images/cum_rain.png')
```

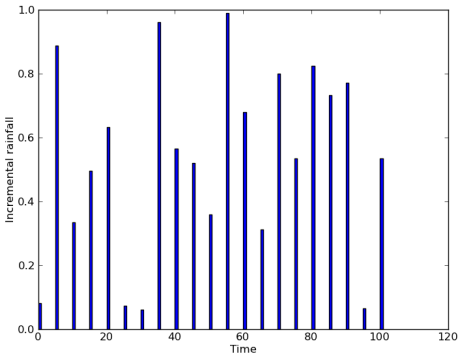

Figure 4.2: Temporal variation in the incremental rainfall.

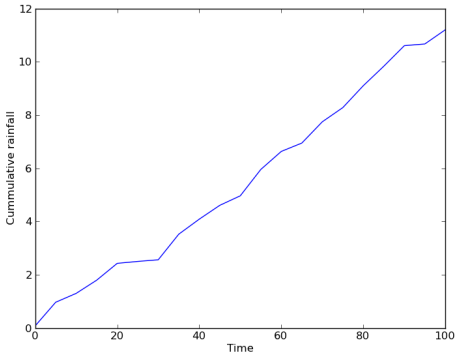

Figure 4.3: Temporal behaviour of the cumulative rainfall .

Usually, we are given the rainfall at some rain gauges, and we want to make the isohyete (contour) plot of the rainfall. To demonstrate this situation, fist we shall generate locations (x,y) and rainfall for ten stations using random numbers. The generated locations of the rain gauges is shown in Fig. 4.4.

```
>>> # import required modules
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>>
>>> #genrate locations and rainfall
>>> x = np.random.rand(10)
>>> y = np.random.rand(10)
>>> rain = 10*np.random.rand(10)
>>>
>>> #plot the locations
>>> plt.scatter(x,y)
>>> plt.xlabel('X')
>>> plt.ylabel('Y')
>>> plt.savefig('/home/tomer/articles/python/tex/images/loc.png')
```

I prefer to add blank lines after a section of code, and comment on the top of section what it is doing. This increases the readability of the code. The `plt.scatter()` makes the scatter plot, i.e. the dots are plotted instead of lines. When there is no order in the data with respect to their position in the array, then scatter plot is used. Like in this case, it is possible that two stations which are close by, but might be placed at distant in the array.

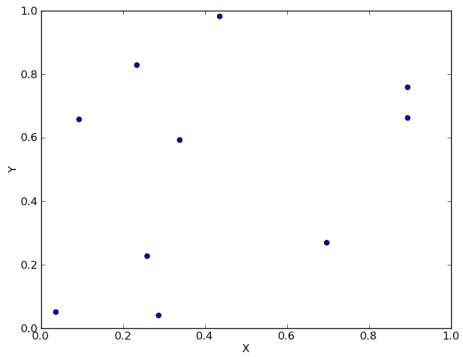

Figure 4.4: Spatial distribution of the rain gauges.

The flow chart of preparing contour map is given in Fig. 4.5. First, we need to generate the grid with regular spacing having the same extent as of the locations of rainfall gauges. Then, from the given location and rainfall data, we need to compute data at regular grid using some interpolation scheme. After this contour maps can be obtained. The `griddata` function of the `scipy.interpolate` library is useful in obtaining the gridded data (data at regular grid). When we need only one or few functions from the library, it is better to call them explicitly, e.g. `from scipy.interpolate import griddata`, like in the following example. We use `meshgrid` function of numpy library, to create the mesh from the given x and y vectors.

```
>>> from scipy.interpolate import griddata
>>> #generate the desired grid, where rainfall is to be interpolated
>>> X,Y = np.meshgrid(np.linspace(0,1,1000), np.linspace(0,1,1000))
>>>
>>> #perform the gridding
>>> grid_rain = griddata((x,y), rain, (X, Y))
```

Now, we can make the contour plot of the gridded data, which is made by `plt.contourf()` function. The `contourf` makes filled contours, while `contour()` provides simple contour. Try using the `contour` instead of `contourf`, and you will see the difference. We begin by clear current figure by using the `plt.clf()`, as there might be some existing figure in the memory especially if you are following all the examples in the same session. We are also overlaying the locations of rainfall gauges using the `plt.scatter()`. The `s` and `c` are used to define the size and color of the markers respectively. The `plt.xlim()` and `plt.ylim()` limits the extent of the x and y axis respectively.

```
>>> plt.clf()
>>> plt.contourf(X,Y,grid_rain)
>>> plt.colorbar()
>>> plt.xlabel('X')
>>> plt.ylabel('Y')
```

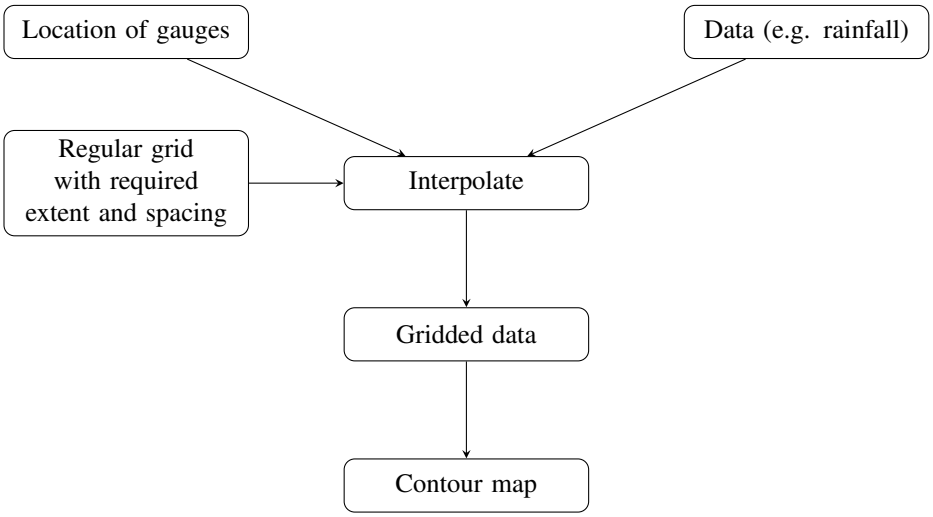

Figure 4.5: Flowchart of making contour map from the data of rainfall gauges

```
>>> plt.scatter(x, y, s=30, c='r')
>>> plt.xlim((0,1))
>>> plt.ylim((0,1))
>>> plt.savefig('/home/tomer/articles/python/tex/images/grid_rain.png')
```

Fig. 4.6 shows the gridded rainfall along with the location of rain gauges. The `gridata` does not perform extrapolation, so the data outside the location of the rain gauges is assigned a value of `nan`. There are other function which can be used to extrapolate the data, which would be discussed later.

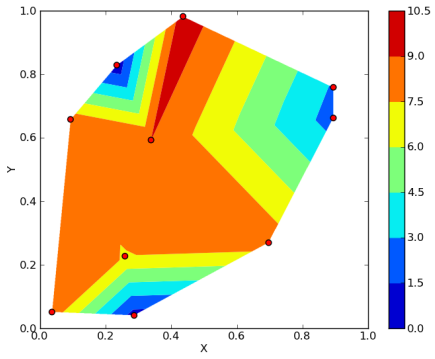

Figure 4.6: Gridded rainfall along with the location of rain gauges.

4.5 Evaporation

Based on the energy balance, the evaporation rate (E_r) after neglecting the sensible heat flux and ground heat flux, can be calculated as,

$$E_r = \frac{R_n}{l_v \rho_w}, \tag{4.3}$$

where, R_n is the net radiation, l_v is the latent heat of vaporization, and ρ_w is the water density. The l_v can be approximated as,

$$l_v = 2500 - 2.36 \times T, \tag{4.4}$$

where, T is the temperature in Celcius.

Based on the aerodynamic, the evaporation rate (E_a) can be calculated as,

$$E_a = B\left(e_{as} - e_a\right), \tag{4.5}$$

where,

$$B = \frac{0.622k^2\rho_a u_2}{p\rho_w\left[\ln(z_2/z_0)\right]^2}, \tag{4.6}$$

e_{as} is the saturated vapor pressure, e_a is the vapor pressure, k is the von Karnman coefficient, u_2 is wind velocity measured at z_2 m height, p is the air pressure, and z_0 is the roughness height.

Usually, evaporation is calculated by combining the energy balance and aerodynamic method. In this case the E becomes,

$$E = \frac{\Delta}{\Delta+\gamma}E_r + \frac{\gamma}{\Delta+\gamma}E_a, \tag{4.7}$$

where, Δ is the gradient of the saturated vapor pressure curve, and is,

$$\Delta = \frac{4098e_s}{(273.3+T)^2}, \tag{4.8}$$

and, the γ is the psychrometric constant, and is defined as,

$$\gamma = \frac{C_p K_h p}{0.622 l_v K_w}, \tag{4.9}$$

k_h and k_w are the heat and vapor diffusivities respectively.

Let us first generate the synthetic data using random numbers. We know that `np.random.rand` provides uniformly distributed random number over an interval of [0,1]. If we want to get the uniformly distributed random number over some other range, say [a,b], we can transform the variable in the following way:

$$x_{new} = a + (b-a) * x_{old}, \tag{4.10}$$

where, X_{old} is uniformly distributed random variable over [0,1], and x_{new} has the range of [a,b]. The `np.random.randn` gives normally distributed random variables having zero mean and standard deviation equal to one. If we are interested in normally distributed random variable having mean μ and standard deviation equal to σ. We can do the following transformation.

$$y_{new} = \mu + \sigma * y_{old}, \tag{4.11}$$

where, y_{new} is transformed variable having mean equal to μ and standard deviation equal to σ, and y_{old} is the normally distributed random variable with mean zero and standard deviation equation to one, as generated by the `np.random.randn` function.

In the following example, we shall generate variable in their usual range. The comment after the

variable provides details of the lower and upper range in case of uniformly distributed random variable, mean and standard deviation when the variable is normally distributed.

```
>>> from __future__ import division
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>>
>>> # generate the synthetic data
>>> Rn = 150+100*np.random.rand(100) # lower bound = 150, upper boun = 250
>>> T = 25+3*np.random.randn(100) # mean = 25, std = 3
>>> Rh = 0.2+0.6*np.random.rand(100) # lower bound = 0.2, upper boun = 0.8
>>> u2 = 3+np.random.randn(100) # mean = 3, std = 1
>>>
>>> # define constants
>>> rho_w = 997; rho_a = 1.19; p = 101.1e3; z2 = 2
>>> z0 = 0.03e-2; k = 0.4; Cp = 1005
```

Now, we apply the energy balance based method to estimate the evaporation.

```
>>> lv = (2500-2.36*T)*1000 # multiplied by thousand to convert from KJ/kg to J/kg
>>> Er = 200/(lv*997)
>>> Er *= 1000*86400 # convert from m/s to mm/day
```

We are using multiplication and assignment operator to convert the units. We could have done this by simply multiplication also i.e. `Er = Er*1000*86400`. The multiplication and assignment operator is fast, as it does not create any temporary variable in the memory. In fact all the assignment operator are faster than simple operator, and should be used whenever there is scope to use them. Now we estimate the evaporation using the aerodynamic method.

```
>>> B = 0.622*k**2*rho_a*u2/(p*rho_w*(np.log(z2/z0))**2)
>>> e_s = 611*np.exp(17.27*T/(237.3+T))
>>> e_a = Rh*e_s
>>> Ea = B*(e_s-e_a)
>>> Ea *= 1000*86400 # convert from m/s to mm/day
```

Now, we combine energy balance and aerodynamic method to get improved estimate of the evaporation.

```
>>> gamma = Cp*p/(0.622*lv) # since kh/kw = 1, hence they are dropped form eq.
>>> delta = 4098*e_s/(237.3+T)**2
>>> w = delta/(delta+gamma)
>>> E = w*Er + (1-w)*Ea
```

Now, we have got four important variables; evaporation using energy balance method (E_r), evaporation using aerodynamics method (E_a), combined evaporation (E), and the ratio of evaporation from energy balance method by combined method (Er/E). We can plot these four variables in four different plot, or we can put them in one figure by making figure into four section. `subplot` is such a function to make figure into subsection. The first argument to `subplot` is the desired number of rows, second argument is the desired numbers of columns in the figure. The third argument is the position of subplot in the figure, which is measured from left to right and top to bottom.

```
>>> plt.clf()
>>> plt.subplot(2,2,1)
>>> plt.plot(Er)
>>> plt.xlabel('Time')
>>> plt.ylabel('Er')

>>> plt.subplot(2,2,2)
>>> plt.plot(Ea)
>>> plt.xlabel('Time')
>>> plt.ylabel('Ea')

>>> plt.subplot(2,2,3, axisbg='y')
>>> plt.plot(E)
>>> plt.xlabel('Time')
>>> plt.ylabel('E')

>>> plt.subplot(2,2,4, axisbg='g')
>>> plt.plot(w)
>>> plt.xlabel('Time')
>>> plt.ylabel('Er/E')
>>> plt.savefig('/home/tomer/articles/python/tex/images/E.png')
```

The estimated E_r, E_a, E and Er/E are shown in the Fig. 4.7. We have additionally used the argument `axisbg` to define the background color for the subplots.

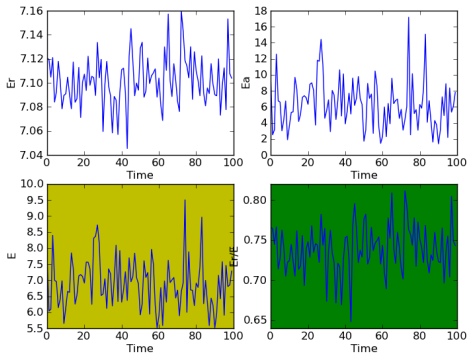

Figure 4.7: The estimated E_r, E_a, E and Er/E.

In the Fig. 4.7, the `ylabel` of subplot 4 is overlapping with the subplot 3. This can be corrected by changing the `wspace`. Which is demonstrated below. Fig. 4.8 shows the improved plot.

```
>>> fig = plt.figure()
>>> fig.subplots_adjust(wspace=0.6)
>>> plt.subplot(2,2,1)
>>> plt.plot(Er)
>>> plt.xlabel('Time')
>>> plt.ylabel('Er')
```

```
>>>
>>> plt.subplot(2,2,2)
>>> plt.plot(Ea)
>>> plt.xlabel('Time')
>>> plt.ylabel('Ea')
>>>
>>> plt.subplot(2,2,3, axisbg='y')
>>> plt.plot(E)
>>> plt.xlabel('Time')
>>> plt.ylabel('E')
>>>
>>> plt.subplot(2,2,4, axisbg='g')
>>> plt.plot(w)
>>> plt.xlabel('Time')
>>> plt.ylabel('Er/E')
>>> plt.savefig('/home/tomer/articles/python/tex/images/corr_E.png')
```

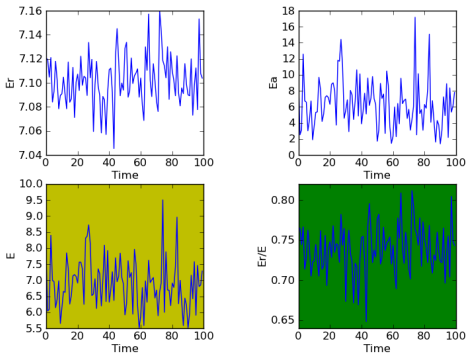

Figure 4.8: Estimated E_r, E_a, E and Er/E with corrected ylabel.

4.6 Infiltration

The cumulative infiltration given by Green-Ampt method is written as,

$$F(t) - \psi\Delta\theta \ln\left(1 + \frac{F(t)}{\psi\Delta\theta}\right) = Kt, \tag{4.12}$$

where, $F(t)$ is the cumulative infiltration after t time, ψ is the suction head , $\Delta\theta$ is given as,

$$\Delta\theta = (1 - S_e)\theta_e, \tag{4.13}$$

wherein, S_e is the degree of saturation, and θ_e is the effective porosity, K is the hydraulic conductivity. To solve the equation using iterative procedure, the Eq. 4.12 is rewritten as,

$$F(t) = \psi\Delta\theta \ln\left(1 + \frac{F(t)}{\psi\Delta\theta}\right) + Kt. \tag{4.14}$$

We use `while` function to iterate till we achieve required accuracy. The iterated value of F are stored using the `append` method. `append` appends the array by one one item, and puts the input variable into it.

```
>>> from __future__ import division
>>> import numpy as np

>>> # define the variables
>>> theta_e = 0.486
>>> psi = 16.7
>>> K = 0.65
>>> S_e = 0.3
>>> t = 1
>>>
>>> #calculate dtheta
>>> dtheta = (1-S_e)*theta_e
>>>
>>> # initial guess of F
>>> F_old = K*t
>>> epsilon = 1
>>> F = []
>>> while epsilon > 1e-4:
>>>     F_new = psi*dtheta * np.log(1+F_old/(psi*dtheta)) + K*t
>>>     epsilon = F_new - F_old
>>>     F_old = F_new
>>>     F.append(F_new)
```

Now, we make a plot of the iterated value of F to see how F is getting updated with iterations. We are also using `-ok` in the `plot` function. The `-o` represents the continuous line with filled dots, and `k` tells that the color of plot is black. We are also specifying the font size for `xlabel` and `ylabel`. We have used '25' for `ylabel` and '20' for `xlabel`, just to demonstrate that different font sizes can be used for different texts. Of course, there is no need to define a different font size for `ylabel` and `xlabel`. The same argument `fontsize` can be used to define the font size for legend also.

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(F,'-ok')
>>> plt.xlabel('Number of iteration',fontsize=25)
>>> plt.ylabel('F',fontsize=20)
>>> plt.savefig('/home/tomer/articles/python/tex/images/F.png')
```

Fig. 4.9 shows the variation of the F over time. The F is becoming almost constant after approximately 12 iterations.

4.7 Surface water

The flow depth in a rectangular channel is given by,

$$Q = \frac{1.49}{n} S_o^{1/2} \frac{(By)^{5/3}}{(B+y)^{2/3}}, \tag{4.15}$$

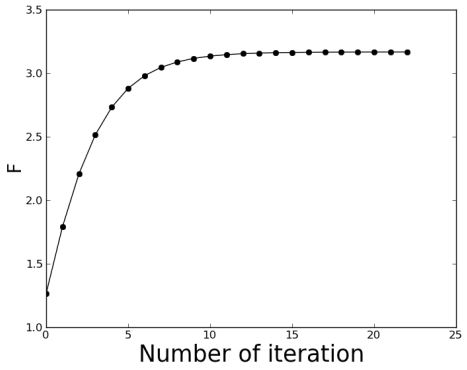

Figure 4.9: The variation of the F with iterations.

where, Q is the flow, n is the Manning's coefficient, S_0 is slope of water surface, B is the width of channel, and y is the flow depth. This is a nonlinear equation in y, and the explicit solution of this is not yet found. This can be solved iteratively like in the last section, or using methods like Newton-Raphson. In this, we will solve using the `fmin` function of the `Scipy.optimize` library. First we will import required libraries. Then we will define a function that takes the flow depth (y) as input and gives the error in the flow estimated based on this y and the given Q. We are taking absolute value of error, other options are like square of error etc. After specifying the function, we can give this function as a input to `fmin` and some initial guess of the y.

```
>>> # import required modules
>>> from __future__ import division
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> from scipy.optimize import fmin
>>>
>>> # define the variables
>>> n = 0.015
>>> S0 = 0.025
>>> Q = 9.26
>>> B = 2
>>>
>>> # define the flow function
>>> def flow(y):
>>>     Q_estiamted = (1.49/n)*(S0**0.5)*((B*y)**(5/3))/((B+y)**(2/3))
>>>     epsilon = np.abs(Q_estiamted - Q)
>>>     return epsilon
>>>
>>> y_optimum = fmin(flow,0.5)
```

`fmin` will give us the required y value. We can also get details of the iterations, and error value at final iterations. We use `print` function to see the details. The output is given below.

```
>>> print(y_optimum)
Optimization terminated successfully.
         Current function value: 0.000078
```

```
         Iterations: 13
         Function evaluations: 26
[ 0.52770386]
```

The optimization terminated successfully, i.e. the required accuracy was achieved within the default maximum number of iterations allowed. The output tells that it took 13 iterations to achieve the required accuracy, and that the function was evaluated 26 times in the process.

4.8 River Routing–Muskingum method

The outflow using Muskingum method is calculated as,

$$Q_{j+1} = C_1 I_{j+1} + C_2 I_j + C_3 Q_j. \tag{4.16}$$

We are given the value of C_1, C_2, C_3, and Q_0, and we are interested in getting the value of Q from time ($t = 0$) to ($t = 19$). In this example, first we define variables, and then we iterate using `for` loop. The list `I` is quiet long, and will not fit into one line. In such cases we can go to second line also, the beginning of brackets tells Python that list has started, and the end brackets tells Python that this is end of the list. We can write list in as many line as we want, no need to specify anything else to tell Python that list is defined in multi-line.

```
>>> from __future__ import division
>>> import numpy as np
>>> # define the variables
>>> I = np.array([93, 137, 208, 320, 442, 546, 630, 678, 691, 675, 634, 571, 477,
>>>               390, 329, 247, 184, 134, 108, 90])
>>> C1 = 0.0631
>>> C2 = 0.3442
>>> C3 = 0.5927
>>>
>>> Q = np.empty(20) # define the empty array
>>> Q[0] = 85 # initial value of Q
>>>
>>> # loop over
>>> for i in range(1,20):
>>>     Q[i] = C1*I[i] + C2*I[i-1] + C3*Q[i-1]
```

Now we can use `matplotlib.pyplot` to plot the inflow and outflow. `-*` means a continuous line with starts, and `--s` means dashed line with square.

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(I, '-*', label='Inflow')
>>> plt.plot(Q, '--s', label='Outflow')
>>> plt.xlabel('Time', fontsize=20)
>>> plt.ylabel('Flow', fontsize=20)
>>> plt.legend()
>>> plt.savefig('/home/tomer/articles/python/tex/images/muskingum.png')
```

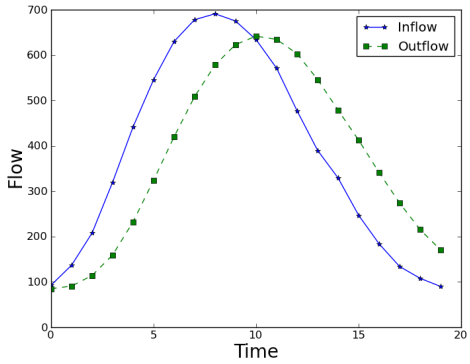

Figure 4.10: The variation of inflow and outflow with time.

Fig. 4.10 shows the variation of inflow and outflow with time. The outflow shows a lag with respect to inflow, and the peak in the outflow is slightly lesser than the inflow.

Chapter 5

Statistics

When there is not a clear understanding of the physics process, or the variables required to do physical modelling are not available, then statistics plays a vital role. There are various modules available in python to deal with statistics, the most commonly used is `scipy.stats`. Additionally there is one more useful modules `statistics`, which can be downloaded from `http://bonsai.hgc.jp/~mdehoon/software/python/Statistics/manual/index.xhtml`. `statistics` is not available for download using the `pip`, and hence you should download it using internet browser, and then install it using either `pip` or `python setup.py install`.

5.1 Empirical distributions

Most of hydrological variables are continuous, but because of our measurement capability we measure them discretely. The classification of discrete data using bins, provides mean to treat the discrete data as continuous. Visualization of the underlying distribution of data is done by plotting the Histogram. The histogram depicts the frequency over discrete intervals (bins). So let us begin with histogram. In the following example, first we will generate some synthetic data, and then compute and plot the histogram.

```
>>> from __future__ import division
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> import scipy.stats as st
>>>
>>> x = np.random.randn(100) # generate some synthetic data
>>>
>>> # compute histogram
>>> n, low_range, binsize, extrapoints = st.histogram(x)
>>> upper_range  = low_range+binsize*(len(n)-1)
>>> bins = np.linspace(low_range, upper_range, len(n))
```

The `st.histogram` provides the number of bins in each interval (n), lower range of the bin (low_range), width of the bins (binsize), and points not used in the calculation of histogram. Since bin size is same for all bins, Python provides only one bin size. We used lower range of bin and bin size to first compute the upper range of the bin, and then compute the mid value for all the bins. Now we can use `bar` to make the histogram. We shall also define the width and color of the bars.

```
>>> plt.bar(bins, n, width=0.4, color='red')
>>> plt.xlabel('X', fontsize=20)
>>> plt.ylabel('number of data points in the bin', fontsize=15)
>>> plt.savefig('/home/tomer/articles/python/tex/images/hist.png')
```

The histogram of the data is shown in Fig. 5.1. In this example, because we have just created 100 random number from the normal distribution, the histogram is not showing the behaviour that normal distribution should show.

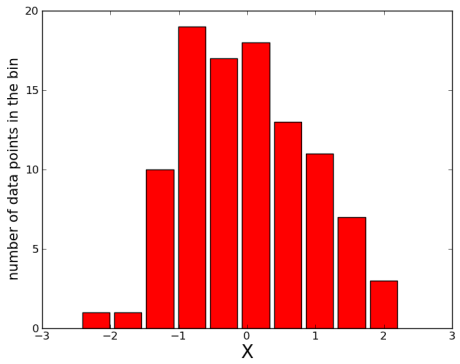

Figure 5.1: The Histogram of x.

Each bar in histogram tells us that how many time the data was in particular bin. A better way to look at the behaviour of data is to look into relative histogram, which tells us about the probability with which data was in some range. The relative histogram or relative frequency is obtained by dividing the frequency in each bin by the sum of frequencies in all the bins. The relative histogram represents the probability of data occurring in the bins. Either we can use the `histogram` function to first compute histogram, and then divide by total number of frequency, or we can directly use the `relfreq` function. `relfreq` provides the relative frequency, along with other output which are similar to that of `histogram`.

```
>>> relfreqs, lowlim, binsize, extrapoints = st.relfreq(x)
>>> plt.bar(bins, relfreqs, width=0.4, color='magenta')
>>> plt.xlabel('X', fontsize=20)
>>> plt.ylabel('Relative frequencies', fontsize=15)
>>> plt.savefig('/home/tomer/articles/python/tex/images/relfreq.png')
```

Because we are using the same x that was used in previous example, hence we are not re-calculating the `bins`. In this bar plot, we are using `magenta` color. The different color of plot, is just to make you familiar with colors and font sizes. And, it does not mean that we should use different color for relative frequency diagram than the histogram; nothing prevents us from using the same color.

The relative histogram is shown in Fig. 5.2. The relative histogram tells us how experimental data behaves; how many times (or with what probability) the data was in some range. The relative histograms only tell about experimental data, about the data we have. What about the data we are going to get into future? The PDF is a better indicator to say something about the future data. The probability density function (pdf) of a random variable is a function that describes the relative

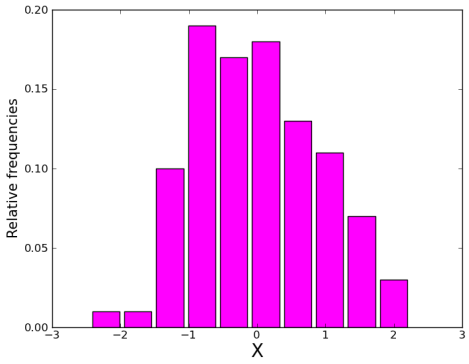

Figure 5.2: The relative histogram of x.

likelihood for this random variable to occur at a given point. The probability for the random variable to fall within a particular region is given by the integral of this variables density over the region. The probability density function is non-negative everywhere, and its integral over the entire space is equal to one. We can divide the relative frequency by the bin size, and get the pdf. A simple way to compute pdf is to use `hist` function. `hist` generates the plot, and also returns the value of pdf over each bin. The number of bins are controlled by giving second argument, in the following example it is set at 10. The `bins` provide the lower and upper ranges of the bin, hence its length is one extra than the number of bins. Apart from the color we are specifying alpha value to `hist` function. `alpha` value controls the transparency of the plot; 0.0 means fully transparent and 1.0 is fully opaque. Fig. 5.3 shows the bar plot of the PDF.

```
>>> n, bins, patches = plt.hist(x, 10, normed=1, facecolor='yellow', alpha=0.5)
>>> plt.xlabel('X', fontsize=15)
>>> plt.ylabel('PDF', fontsize=15)
>>> plt.savefig('/home/tomer/articles/python/tex/images/pdf.png')
```

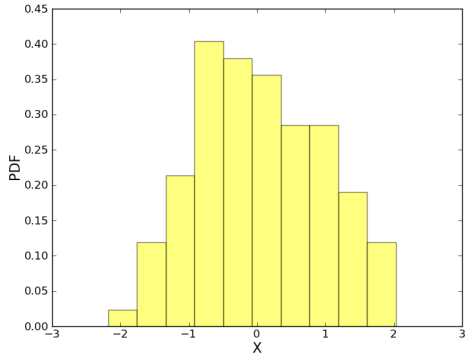

Figure 5.3: The pdf of x.

The cumulative distribution function (CDF) describes the probability that a real-valued random variable X with a given probability distribution will be found at a value less than or equal to x. `cumfreq`

provides cumulative frequency of the data, which can be used to compute the CDF. If we divide cumulative frequency by the total frequency, we get the CDF. The last value of cummulative frequency is equal to the total frequency, hence we are using this to compute CDF. The CDF is shown in Fig. 5.4.

```
>>> cumfreqs, lowlim, binsize, extrapoints = st.cumfreq(x)
>>> plt.bar(bins[:-1], cumfreqs/cumfreqs[-1], width=0.4, color='black', alpha=0.45)
>>> plt.xlabel('X', fontsize=15)
>>> plt.ylabel('CDF', fontsize=15)
>>> plt.savefig('/home/tomer/articles/python/tex/images/cdf.png')
```

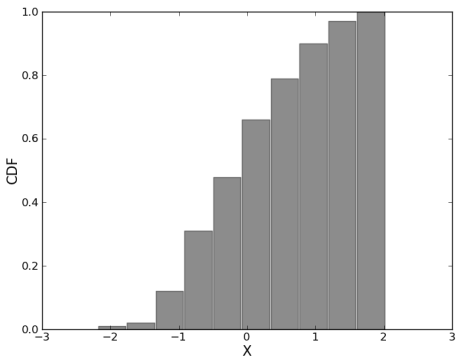

Figure 5.4: The CDF of x.

We can use `ECDF` function of the `scikits` library to directly estimate the empirical cumulative distribution function (ECDF). The information about `scikits` is give at `http://pypi.python.org/pypi/scikits.statsmodels`. The empirical distribution function (CDF) is the cumulative distribution function associated with the empirical measure of the sample. This cdf is a step function that jumps up by $1/n$ at each of the n data points. The empirical distribution function estimates the underlying cdf of the points in the sample. In the previous example, we have estimated CDF (ECDF) after classifying data into some bins. This means, we assumed that the data behaves in a statistical similar way over some small range. We can estimate CDF without making this assumption also, which would be done using `ECDF` function. The output form `ECDF` function is a object which store the value of data and their corresponding ECDF. The data is retrieved using `ecdf.x` and their corresponding ECDF is retrieved using `ecdf.y`. The `ecdf` is the name of variable that you have defined to store the output of `ECDF` function, if you use some other name, you need to use same name to retrieve `x` and `y` attributes.

```
>>> import numpy as np
>>> import scikits.statsmodels.api as sm
>>> import matplotlib.pyplot as plt
>>>
>>> # generate some data
>>> data = np.random.randn(100)
>>>
>>> # estimate ecdf
>>> ecdf = sm.tools.tools.ECDF(data)
```

We should plot ECDF as a step plot, because every ECDF is over some small interval. The ECDF plot is shown in Fig. 5.5.

```
>>> plt.step(ecdf.x, ecdf.y)
>>> plt.xlabel('data', fontsize=20)
>>> plt.ylabel('Empirical CDF', fontsize=15)
>>> plt.savefig('/home/tomer/articles/python/tex/images/ecdf.png')
```

We can also use `ecdf` to evaluate ECDF at any value of data. Let us evaluate and print value of ECDF at some data point (say at 0).

```
>>> print(ecdf(0))
0.48999999999999999
```

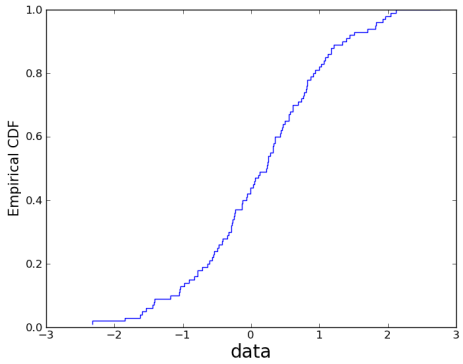

Figure 5.5: The empirical CDF estimated using ordinary method.

The empirical CDF estimated using the method mentioned above results in the step function, which does not look so nice. A better way of estimating ECDF is by using kernel functions. This can be done by `statistics` module. The `statistics` library provides functions to estimate PDF and CDF using various kernel functions. `cpdf` function is used for estimating CDF. We have also defined the name of kernal (Epanechnikov). The available kernal are given at website of library. Inside `legend` we are defining location (loc) of the legend as `best`, which means Python will try to put the legend in the way, as to minimize the interference with plot. The resulted graph after using this curve is shows in Fig. 5.6. It is evident from this figure, that this shows a smoother variation compared to ordinary method of ECDF estimation.

```
>>> import statistics
>>> y,x = statistics.cpdf(data, kernel = 'Epanechnikov')
>>> plt.plot(ecdf.x, ecdf.y, label='Ordinary')
>>> plt.plot(x, y, label='Kernel')
>>> plt.xlabel('data', fontsize=20)
>>> plt.ylabel('Empirical CDF', fontsize=15)
>>> plt.legend(loc='best')
>>> plt.savefig('/home/tomer/articles/python/tex/images/kernel.png')
```

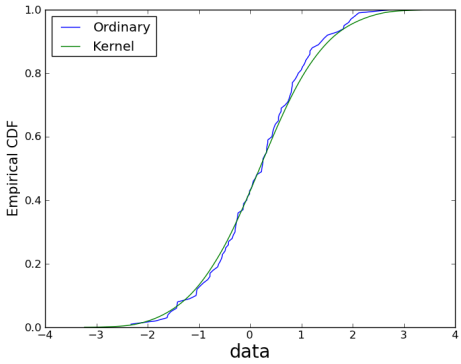

Figure 5.6: Comparison of the ECDF estimated using ordinary method and based on kernal functions.

5.2 Theoretical distributions

Theoretical distributions are based upon mathematical formulas rather than empirical observations. There are various types of theoretical distributions, commonly used in hydrology are: Normal, Uniform, Exponential, Chi, Cauchy. The parameters of distributions are termed as location, scale, and shape parameter. The location parameter is the one, who change the location of pdf without affecting other attributes. Shape parameter is the one, who change the shape of distribution without affecting other attributes. The parameter which stretch or shrink the distribution is called the scale parameters.

First we will generate normally distributed random variables. The input required for this are location and scale parameter, which are mean and standard deviation in case of normal distribution. We can also use `np.random.randn` to generate normally distributed random variables, but `scipy.stats` provides many other utilities (methods). So we shall use `scipy.stats` library.

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> import scipy.stats as st
>>>
>>> # generate instances of normaly distributed random variable
>>> rv1 = st.norm(loc=0, scale=5)
>>> rv2 = st.norm(loc=0, scale=3)
>>> rv3 = st.norm(loc=0, scale=7)
```

Now these instances of variables can be used to evaluate PDF at any value. In the following example, we are computing pdf from -50 to 50 for plotting purpose.

```
>>> x = np.linspace(-50,50, 1000)
>>> y1 = rv1.pdf(x)
>>> y2 = rv2.pdf(x)
>>> y3 = rv3.pdf(x)
```

Now, we have estimated PDF, it can be plotted. On the $x-axis$ we will keep the variable, and on the $y-axis$ keep the PDF. We are also supplying additional argument `lw` to `plot` function, which represents `line width`, and is used to control the widths of the plot. Fig. 5.7 shows the PDF for three normally distributed random variables with varying scale parameters. The figure

illustrates the effect of scale parameter on the PDF. In case of less scale parameter, more mass of pdf is concentrated in the center, as the scale parameter is increasing, the spread is increasing.

```
>>> plt.plot(x, y1, lw=3, label='scale=5')
>>> plt.plot(x, y2, lw=3, label='scale=3')
>>> plt.plot(x, y3, lw=3, label='scale=7')
>>> plt.xlabel('X', fontsize=20)
>>> plt.ylabel('PDF', fontsize=15)
>>> plt.legend()
>>> plt.savefig('/home/tomer/articles/python/tex/images/norm_pdf.png')
```

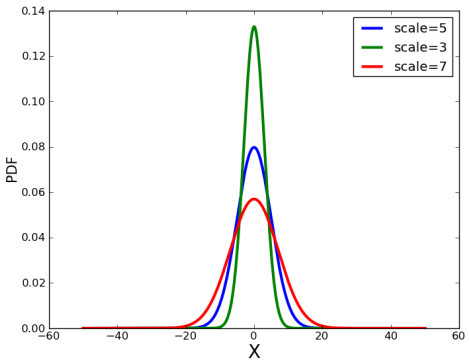

Figure 5.7: PDF for normal distribution with various scale parameter.

We can use same instance to also get the CDF. The `cdf` method gives the CDF at given input, which could be a scalar or an array. The CDF is shown in Fig. 5.8. CDF also shows the effect of scale parameter, but PDF provides a better inside. Hence it is always better to plot PDF to see the behaviour of the distribution or of the empirical data.

```
>>> y1 = rv1.cdf(x)
>>> y2 = rv2.cdf(x)
>>> y3 = rv3.cdf(x)
>>>
>>> # plot the pdf
>>> plt.clf()
>>> plt.plot(x, y1, lw=3, label='scale=5')
>>> plt.plot(x, y2, lw=3, label='scale=3')
>>> plt.plot(x, y3, lw=3, label='scale=7')
>>> plt.xlabel('X', fontsize=20)
>>> plt.ylabel('PDF', fontsize=15)
>>> plt.legend()
>>> plt.savefig('/home/tomer/articles/python/tex/images/norm_cdf.png')
```

There are other quiet commonly used distributions in hydrology, e.g. Cauchy, Chi, Exponential and Uniform etc. Let us, play with them also. First we will generate instance of these distributions. Chi distribution also require degree of freedom parameter apart from the location and scale parameters. In case of uniform distribution, the location parameter is defined as lower range, and scale parameter

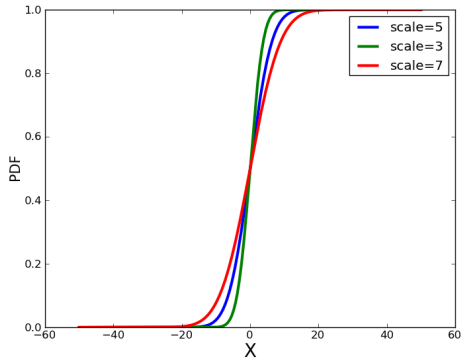

Figure 5.8: CDF of normal distribution for various scale parameter.

is defined as upper range, which is not true mathematically, and is defined just to make things easier in providing input to function. Fig.5.9 shows the PDF for these distribution.

```
>>> rv1 = st.cauchy(loc=0, scale=5)
>>> rv2 = st.chi(2, loc=0, scale=8)
>>> rv3 = st.expon(loc=0, scale=7)
>>> rv4 = st.uniform(loc=0, scale=20)
>>>
>>> # compute pdf
>>> y1 = rv1.pdf(x)
>>> y2 = rv2.pdf(x)
>>> y3 = rv3.pdf(x)
>>> y4 = rv4.pdf(x)
>>>
>>> # plot the pdf
>>> plt.plot(x, y1, lw=3, label='Cauchy')
>>> plt.plot(x, y2, lw=3, label='Chi')
>>> plt.plot(x, y3, lw=3, label='Exponential')
>>> plt.plot(x, y4, lw=3, label='Uniform')
>>> plt.xlabel('X', fontsize=20)
>>> plt.ylabel('PDF', fontsize=15)
>>> plt.legend()
>>> plt.savefig('/home/tomer/articles/python/tex/images/pdf_all.png')
```

We generate a number of random variable from some distribution to represent the distribution. To explore the impact of the number of samples on the empirical distribution, we will generate random number from same distribution but with various number of samples, and see how it is effecting the empirical distribution. We will not be using any quantitative measure to check this, as till this stage we have not talked about them, rather we will just visualize graphically. First we will play with normal distribution which is most commonly used distribution. We will use `hist` function of the `matplotlib.pyplot` library to compute the PDF. We are specifying `normed=1` for the `hist` function, which means that the area of histogram should be made equal to one, and which happens to be the PDF. We are also using `plt.axis` to specify the limits of the plot, and we are keeping it same so that we can compare plots easily. The argument for `plt.axis` are $[x_{min}, x_{max}, y_{min}, y_{max}]$. Fig.

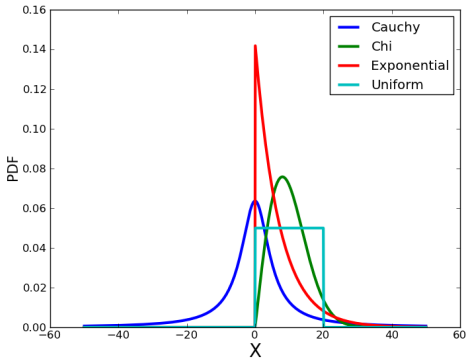

Figure 5.9: PDF for various distributions.

5.10 shows the empirical and theoretical PDF for sample equal to 100, 1000, 10,000, and 100,000. It is clear from the figure that as the number of sample are increasing, the empirical distribution is approaching near to theoretical one. At 100 sample, the distribution is represented very poorly by the data, while in other case it is relatively better.

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> import scipy.stats as st
>>>
>>> # normal distribution
>>> rv = st.norm(loc=0, scale=5)
>>>
>>> x1 = np.linspace(-20, 20, 1000)
>>> y1 = rv.pdf(x1)
>>>
>>> # compute and plot pdf
>>> fig = plt.figure()
>>> fig.subplots_adjust(wspace=0.4)
>>>
>>> plt.subplot(2,2,1)
>>> x = rv.rvs(size=100)
>>> n, bins, patches = plt.hist(x, 20, normed=1, facecolor='yellow', alpha=0.5)
>>> plt.plot(x1, y1, 'r', lw=3)
>>> plt.xlabel('X', fontsize=15)
>>> plt.ylabel('PDF', fontsize=15)
>>> plt.axis([-20, 20, 0, 0.10])
>>> plt.text(-18,0.08,'n=100')
>>>
>>> plt.subplot(2,2,2)
>>> x = rv.rvs(size=1000)
>>> n, bins, patches = plt.hist(x, 20, normed=1, facecolor='green', alpha=0.5)
>>> plt.plot(x1, y1, 'r', lw=3)
>>> plt.xlabel('X', fontsize=15)
>>> plt.ylabel('PDF', fontsize=15)
```

```
>>> plt.axis([-20, 20, 0, 0.10])
>>> plt.text(-18,0.08,'n=1000')
>>>
>>> plt.subplot(2,2,3)
>>> x = rv.rvs(size=10000)
>>> n, bins, patches = plt.hist(x, 20, normed=1, facecolor='black', alpha=0.5)
>>> plt.plot(x1, y1, 'r', lw=3)
>>> plt.xlabel('X', fontsize=15)
>>> plt.ylabel('PDF', fontsize=15)
>>> plt.axis([-20, 20, 0, 0.10])
>>> plt.text(-18,0.08,'n=10000')
>>>
>>> plt.subplot(2,2,4)
>>> x = rv.rvs(size=100000)
>>> n, bins, patches = plt.hist(x, 20, normed=1, facecolor='magenta', alpha=0.5)
>>> plt.plot(x1, y1, 'r', lw=3)
>>> plt.xlabel('X', fontsize=15)
>>> plt.ylabel('PDF', fontsize=15)
>>> plt.axis([-20, 20, 0, 0.10])
>>> plt.text(-18,0.08,'n=10000')
>>>
>>> plt.savefig('/home/tomer/my_books/python_in_hydrology/images/rand_theo.png')
```

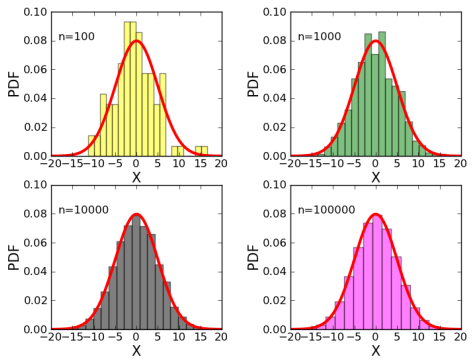

Figure 5.10: Effect of the number of samples (n) on empirical PDF versus theoretical PDF.

We can also see the effect of the number of sample on the empirical distribution apart from the normal distribution, say Laplace distribution. In this example, we are controlling the limits of the axis using the `plt.xlim` and `plt.ylim` separately. In case of $y-axis$ we are only defining `ymax` to control the maximum limit of the axis, while for $x-axis$ we are defining both the limits. The limit of the `axis` could have been fixed using the `axis` as used in the last example, this is just to show that we can control only one limit, and leave the other limit to `plt`. Fig. 5.11 shows the empirical and theoretical pdf for various number of samples.

```
>>> rv = st.laplace(loc=0, scale=15)
>>> x1 = np.linspace(-100, 100, 1000)
>>> y1 = rv.pdf(x1)
```

```
>>>
>>> # compute and plot pdf
>>> plt.clf()
>>> fig = plt.figure()
>>> fig.subplots_adjust(wspace=0.4)
>>>
>>> plt.subplot(2,2,1)
>>> x = rv.rvs(size=100)
>>> n, bins, patches = plt.hist(x, 20, normed=1, facecolor='yellow', alpha=0.5)
>>> plt.plot(x1, y1, 'r', lw=3, label='scale=5')
>>> plt.xlabel('X', fontsize=15)
>>> plt.ylabel('PDF', fontsize=15)
>>> plt.ylim(ymax=0.04)
>>> plt.xlim((-100,100))
>>> plt.text(-80,0.035,'n=100')
>>>
>>> plt.subplot(2,2,2)
>>> x = rv.rvs(size=1000)
>>> n, bins, patches = plt.hist(x, 20, normed=1, facecolor='green', alpha=0.5)
>>> plt.plot(x1, y1, 'r', lw=3, label='scale=5')
>>> plt.xlabel('X', fontsize=15)
>>> plt.ylabel('PDF', fontsize=15)
>>> plt.ylim(ymax=0.04)
>>> plt.xlim((-100,100))
>>> plt.text(-80,0.035,'n=1000')
>>>
>>> plt.subplot(2,2,3)
>>> x = rv.rvs(size=1000)
>>> n, bins, patches = plt.hist(x, 20, normed=1, facecolor='black', alpha=0.5)
>>> plt.plot(x1, y1, 'r', lw=3, label='scale=5')
>>> plt.xlabel('X', fontsize=15)
>>> plt.ylabel('PDF', fontsize=15)
>>> plt.ylim(ymax=0.04)
>>> plt.xlim((-100,100))
>>> plt.text(-80,0.035,'n=10000')
>>>
>>> plt.subplot(2,2,4)
>>> x = rv.rvs(size=10000)
>>> n, bins, patches = plt.hist(x, 20, normed=1, facecolor='magenta', alpha=0.5)
>>> plt.plot(x1, y1, 'r', lw=3, label='scale=5')
>>> plt.xlabel('X', fontsize=15)
>>> plt.ylabel('PDF', fontsize=15)
>>> plt.ylim(ymax=0.04)
>>> plt.xlim((-100,100))
>>> plt.text(-80,0.035,'n=100000')
>>>
>>> plt.savefig('/home/tomer/my_books/python_in_hydrology/laplace_rand.png')
```

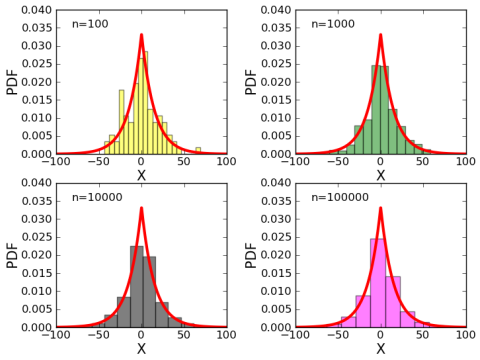

Figure 5.11: Effect of the number of samples on empirical PDF versus theoretical PDF for Laplace distribution.

5.3 The t-test

In statistics, we make hypothesis like two different random variable have the same mean, or have equal variance, or they follow same distribution. To test these hypothesis, test statistic is derived, and based on the test statistic, the hypothesis is rejected or accepted. When the test statistic follow a Student's t distribution, the t-test is used to test hypothesis. This test is available in the `scipy.stats` library. Let us first test if the mean of random variable is same as we expected or not. `st.ttest_1samp` function is used for this purpose. We will generate normally distributed random variable having mean equal to 5, and standard deviation equal to 10. And we will test if the mean of this generated random variable is 5 or not. Because we are talking 1000 number of sample, we expect that the mean will be approximately equal to 5 most of the time (but not always). The hypothetis is rejected or accepted based on the p-value. The p-value close to one means that hypothesis is true; a value closer to zero means that the hypothesis is rejected. The significance level (α) is used to define the threshold, which is often taken as 0.05 or 0.01. If the p-value is greater than this than we can accept the hypothesis.

```
>>> import scipy.stats as st
>>> rvs1 = st.norm.rvs(loc=5,scale=10,size=1000)
>>> # t-test
>>> t, p = st.ttest_1samp(rvs1,5)
>>> print(p)
0.882877605761
```

We see in this example that p-value is 0.88, which means the mean of generated random variable is close to 5. The t-test is also used to test if the mean of two independent random number is equal or not. Let us generate two normally distributed random variable with same mean, say 5. We would like to see if the mean of these two independent random variable is same or not. We can use `st.ttest_ind` for this purpose. In this example the p-value is 0.96, which means means are same.

```
>>> rvs1 = st.norm.rvs(loc=5,scale=10,size=1000)
>>> rvs2 = st.norm.rvs(loc=5,scale=10,size=1000)
>>> # t-test
>>> t, p = st.ttest_ind(rvs1,rvs2)
>>> print(p)
```

```
0.963392592667
```

In the previous example, we tested two independent sample for the mean. We can also test if the mean is same or not, when the samples are related or come from same experiment. We can use `st.ttest_rel` for this purpose. We get p-value 0.57, which means that the means are same.

```
>>> rvs1 = st.norm.rvs(loc=5,scale=10,size=1000)
>>> rvs2 = st.norm.rvs(loc=5,scale=10,size=1000)
>>> # t-test
>>> t, p = st.ttest_rel(rvs1,rvs2)
>>> print(p)
0.569900697821
```

5.4 KS test

KolmogorovSmirnov (KS) test is a non-parametric test to compare the equality of two continuous one dimensional probability distributions. In this test, we quantify the distance (absolute difference) between distributions. These two distributions could be two different sample, or one could be sample and another one a theoretical distribution. Let us test if our generated normal random variable follow normal distribution or not. `st.kstest` is the function to to perform KS test.

```
>>> import numpy as np
>>> import scipy.stats as st
>>> x = np.random.randn(1000)
>>> # KS test
>>> D, p = st.kstest(x,'norm')
>>> print(p)
0.652473310995
```

We get a p-value equal to 0.65, which means that our generated normally distributed random variable is in fact normal. We can also test if the the generated uniformly distributed random variable are not normal by chance. In this we get a p-value equal to 0, which means that our generated random numbers in this case are not normal.

```
>>> y = np.random.rand(1000)
>>> D, p = st.kstest(y,'norm')
>>> print(p)
0.0
```

5.5 The chi square test

We can also compare distribution by comparing their PDFs. In this case we use the Chi square (χ^2) test. In this test, χ^2 statistics is computed first, and based on this we say if distributions are same or not. We can compare sample with the theoretical distribution, or two samples. We will take two pdfs, in which one is assumed to be observed and another one is expected. We will use the `chisquare` function from `scipy.stats.mstats` library. This function gives χ^2 statistics and

p-value of the test. In the following example, we get a p-value close to zero, it means that these two frequency comes from different distributions.

```
>>> from scipy.stats.mstats import chisquare
>>> import numpy as np
>>> f_obs = np.array([10, 15, 20, 30]) # observed pdf
>>> f_exp = np.array([10, 5, 15, 30]) # expected pdf
>>> # chi square test
>>> c, p = chisquare(f_obs, f_exp)
>>>
>>> print(c,p)
(21.666666666666668, 7.6522740548062336e-05)
```

5.6 Measure of statistical dependence

Often we are interested in knowing if two hydrological variables are dependant or not. In this section, it will be described to check their statistical dependency. If two variable are statistically dependent, it does not mean that they are physically also dependent. First we will generate two variables having different relationship between them. Few with perfect relationship, and few with some noise added. In the following example, we are creating six variables:

- Perfect linear relationship ($y = a + bx$),
- Linear relationship with some noise ($y = a + bx + \varepsilon$),
- Quadratic relationship which is monotonic ($y = x^2$),
- Quadratic relationship with some noise ($y = x^2 + \varepsilon$),
- Quadratic relationship but this one is not monotonic ($y = (x-5)^2$), and
- Noise added to previous one ($y = (x-5)^2 + \varepsilon$).

Fig. 5.12 shows these variables. Out of the six variable, three have perfect relationship, and three has some noise. We would expect our measure of statistical dependence to reveal this.

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>>
>>> x = np.linspace(0,10)
>>> y1 = 2*x
>>> y2 = y1 + 2*np.random.randn(50)
>>> y3 = x**2
>>> y4 = y3 + 2*np.random.randn(50)
>>> y5 = (x-5)**2
>>> y6 = y5 + 2*np.random.randn(50)
>>>
>>> plt.subplot(2,3,1)
>>> plt.plot(x, y1, '.')
>>> plt.text(2,15,'(a)')
```

```
>>>
>>> plt.subplot(2,3,2)
>>> plt.plot(x, y2, '.')
>>> plt.text(2,15,'(b)')
>>>
>>> plt.subplot(2,3,3)
>>> plt.plot(x, y3, '.')
>>> plt.text(2,80,'(c)')
>>>
>>> plt.subplot(2,3,4)
>>> plt.plot(x, y4, '.')
>>> plt.text(2,100,'(d)')
>>>
>>> plt.subplot(2,3,5)
>>> plt.plot(x, y5, '.')
>>> plt.text(2,20,'(e)')
>>>
>>> plt.subplot(2,3,6)
>>> plt.plot(x, y6, '.')
>>> plt.text(2,25,'(f)')
>>> plt.savefig('/home/tomer/my_books/python_in_hydrology/images/corr.png')
```

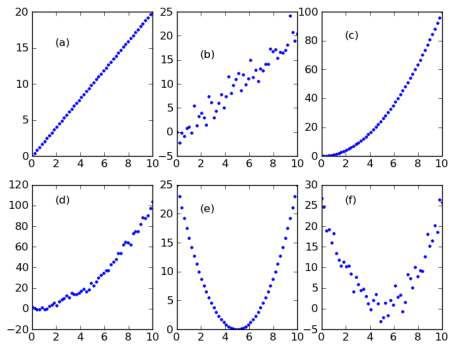

Figure 5.12: Different kind of relationship between two variables.

Unfortunately there is no measure to reveal the strength of relationship in case on non-linearty. The reason for this is that we can have any form of non linear relationship which is not possible for measure to quantity. Having said that, there are some measure which work well in some case. We will explore few of them. First begin with Pearson's correlation coefficient, which provides the strength of linear relationship. `st.pearsonr` function can be used to compute Pearson's correlation coefficient. This function also gives the p-value, which can be used to quantity the significance of the relationship. We are using `%` operator for formatting the output. `.2f` tells to print the output till second decimal places.

```
>>> import scipy.stats as st
>>> r1, p1 = st.pearsonr(x,y1)
>>> r2, p2 = st.pearsonr(x,y2)
```

```
>>> r3, p3 = st.pearsonr(x,y3)
>>> r4, p4 = st.pearsonr(x,y4)
>>> r5, p5 = st.pearsonr(x,y5)
>>> r6, p6 = st.pearsonr(x,y6)
>>>
>>> # print r's
>>> print('%.2f %.2f %.2f %.2f %.2f %.2f')%(r1,r2,r3,r4,r5,r6)
1.00 0.97 0.97 0.96 0.00 -0.02
```

We get 1.0 for first case, and a value slightly lesser than 1.0 for second case, because we perturbed the relationship. In third case, we get a value of 0.97, while in reality the relationship is perfect though not linear. The value is 0 in fifth case, even though the relationship perfect. So we can conclude that Pearson's correlation coefficient is good **only** to measure the linear relationship. Now we will compute Spearman's correlation coefficient for all these six cases using `st.spearman`.

```
>>> rho1, p1 = st.spearmanr(x,y1)
>>> rho2, p2 = st.spearmanr(x,y2)
>>> rho3, p3 = st.spearmanr(x,y3)
>>> rho4, p4 = st.spearmanr(x,y4)
>>> rho5, p5 = st.spearmanr(x,y5)
>>> rho6, p6 = st.spearmanr(x,y6)
>>>
>>> # print rho's
>>> print('%.2f %.2f %.2f %.2f %.2f %.2f')%(rho1,rho2,rho3,rho4,rho5,rho6)
1.00 0.97 1.00 0.99 0.01 -0.04
```

Spearman's correlation coefficient is providing the similar output like one by Spearman's except that it is able to recognize the relationship in third and fourth case better. In the third and fourth case, the relationship was non-linear but monotonic. Spearman's correlation coefficient is useful measure when the data has monotonic behaviour. But this is also not working properly in case when the relationship is well defined, but not monotonic. Kendall's tau correlation coefficient is a statistics to measure the rank correlation. Kendall's tau can be computed using the `st.kendalltau` function.

```
>>> tau1, p1 = st.kendalltau(x,y1)
>>> tau2, p2 = st.kendalltau(x,y2)
>>> tau3, p3 = st.kendalltau(x,y3)
>>> tau4, p4 = st.kendalltau(x,y4)
>>> tau5, p5 = st.kendalltau(x,y5)
>>> tau6, p6 = st.kendalltau(x,y6)
>>>
>>> # print tau's
>>> print('%.2f %.2f %.2f %.2f %.2f %.2f')%(tau1,tau2,tau3,tau4,tau5,tau6)
1.00 0.86 1.00 0.95 0.01 -0.05
```

This provides measure similar to that of Spearman's correlation coefficient, and is unable to reveal non-monotonic relationship that we have in fifth and sixth case.

5.7 Linear regression

Linear regression is an approach to model the relationship between two variables using linear function. We will use `st.linregress` function to perform linear regression. We will first generate some synthetic data using a known linear model, and will also add some noise using normally distributed random variable. `linregress` provides correlation, p-value, and standard error of estimate apart from model coefficients.

```
>>> import numpy as np
>>> import scipy.stats as st
>>> # generate the data
>>> n = 100 # length of the data
>>> x = np.random.rand(n)
>>> y = 3 + 7*x + np.random.randn(n)
>>> # perform linear regression
>>> b, a, r, p, e = st.linregress(x, y)
>>> print(a,b)
(2.9059642495310403, 7.3015273619236618)
```

We generated data using linear model ($y = 3 + 7x + \varepsilon$), while linear regression ($y = 2.91 + 7.3x$). The difference in the fitted model and true model, is because of the noise. As you add more noise, you will see that the fitted model departs more from the reality. Fig. 5.13 shows the true line ($y = 3 + 7x$), corrupted measurement ($y = 3 + 7x + \varepsilon$), fitted line ($y = 2.91 + 7.3x$), and prediction interval for the fitted line. The fitted line and true line are matching reasonably. The prediction interval are also quiet reasonable.

The variance of a predicted Y_{pred} is given by,

$$\sigma^2_{pred} = E[(Y_{pred} - \widehat{Y})^2] = \sigma^2_{\varepsilon}\left(1 + \frac{1}{n} + \frac{(X_0 - \overline{X})^2}{\sum_{i=1}^{n}(X - \overline{X})^2}\right). \tag{5.1}$$

Where, the σ^2_{ε} is estimated by s^2 the classic unbiased estimator of the residual variance. The σ^2_{pred} is used to generate prediction interval using a Students t distribution with $n-2$ degrees of freedom (because s^2 is an estimator). The confidence interval around Y_{pred} is given by,

$$PI = \sigma_{pred} * z \tag{5.2}$$

where, PI is the prediction interval, z is the value of Students t distribution at α significance level.

```
>>> eps = y - a - b*x  # error of fitting and measured data
>>> x1 = np.linspace(0, 1) # x axis to plot the PI
>>> # variace of fitting error
>>> e_pi = np.var(eps)*(1+1.0/n + (x1-x.mean())**2/np.sum((x-x.mean())**2))
>>> # z value using the t distribution and with dof = n-2
>>> z = st.t.ppf(0.95, n-2)
>>> # prediction interval
>>> pi = np.sqrt(e_pi)*z
>>> zl = st.t.ppf(0.10, n-2) # z at 0.1
>>> zu = st.t.ppf(0.90, n-2) # z at 0.9
>>> ll = a + b*x1 + np.sqrt(e_pi)*zl # 10 %
>>> ul = a + b*x1 + np.sqrt(e_pi)*zu # 90 %
```

Finally, we can plot the true line, fitted line, measurement corrupted with noise and prediction intervals.

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(x,y,'ro', label='measured')
>>> plt.plot(x1,ll,'--', label='10%')
>>> plt.plot(x1,ul,'--', label='90%')
>>> plt.xlabel('x')
>>> plt.ylabel('y')
>>> plt.legend(loc='best')
>>> plt.savefig('/home/tomer/my_books/python_in_hydrology/images/lin_regress.png')
```

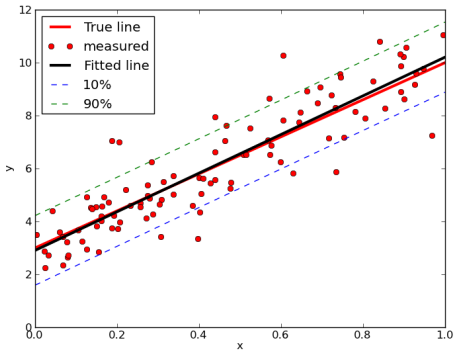

Figure 5.13: The fitted line along with the uncertainty intervals.

5.8 Polynomial regression

We can do the polynomial regression using the `np.polyfit`. This provides the fitted coefficients. We can define the order of polynomial as third argument to the `np.polyfit` function. First, we are generating a second degree polynomial ($y = 1 + 2x - 3x^2$), then we are adding noise into it.

```
>>> import numpy as np
>>> # generate data
>>> x = np.linspace(0,10)
>>> y = 1 + 2*x - 3*x**2 + 15*np.random.randn(50)
>>> # fit the polynomial
>>> z = np.polyfit(x,y,2)
>>> print(z)
[-3.03571947  1.34263078  4.58790632]
```

The `np.polyfit` function is providing fitted polynomial as $y = 4.58 + 1.34x - 3.03x^2$, while the coefficient of true polynomials were different. Only the third parameter is computed reasonably. Other two parameters differs a lot compared to the true one. Let us look into the behaviour of fitted polynomials compared to the true polynomial. `np.poly1d` function is used to evaluate the polynomial using the fitted coefficient returned by `np.polyfit`. Fig. 5.14 shows the resulted plot.

Though the fitted coefficients differed than real coefficients, but the fitted polynomial is quiet close to the true one. The parameter associated with second degree was computed quiet reasonably by the `np.polyfit`, this means that this is the most sensitive parameters compared to other one.

```
>>> import matplotlib.pyplot as plt
>>> # evaluate polynomial
>>> p = np.poly1d(z)
>>> z_true = np.array([-3, 2, 1]) # coefficient of true polynomial
>>> p_true = np.poly1d(z_true) # true polynomial
>>> # plot
>>> plt.plot(x, y,'.r', label='noisy data')
>>> plt.plot(x, p_true(x), label='True curve')
>>> plt.plot(x, p(x), label='Fitted curve')
>>> plt.xlabel('x')
>>> plt.ylabel('y')
>>> plt.legend()
>>> plt.savefig('/home/tomer/my_books/python_in_hydrology/images/cuve_regre.png')
```

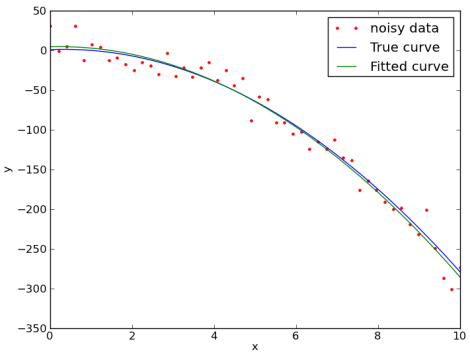

Figure 5.14: Fitted curve along with true curve, and data with noise.

5.9 Interpolation

There are various way of doing interpolation. Commonly used methods are piecewise linear and non-linear, splines, and radial basis functions. In this section, we will use piecewise linear and radial basis function to interpolate the data.

We will first generate few data points having exponential relationship. Then we will interpolate using `interp1d` function of `scipy.interpolate` library. This function returns an object, which can be used later to evaluate the fitted piecewise linear curve at required data points. Fig. 5.15 shows the fitted piecewise polynomial along with the data used to generate it.

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> from scipy.interpolate import interp1d
```

```
>>> # generate data
>>> x = np.linspace(0,1,5)
>>> y = np.exp(-x)
>>> f = interp1d(x, y)
>>> xnew = np.linspace(x.min(), x.max())
>>> ynew = f(xnew)   # use interpolation function returned by `interp1d`
>>> # plot
>>> plt.plot(x, y, 'ro', label='y')
>>> plt.plot(xnew, ynew, '-', label='ynew')
>>> plt.xlabel('x')
>>> plt.ylabel('y')
>>> plt.legend()
>>> plt.savefig('/home/tomer/my_books/python_in_hydrology/images/inter.png')
```

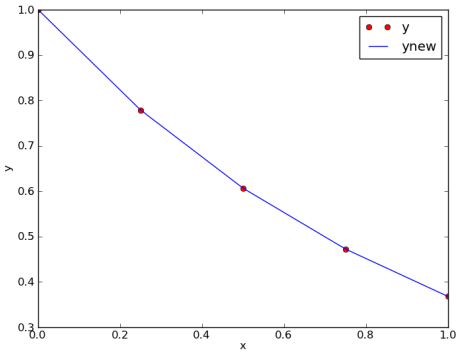

Figure 5.15: Interpolated curve versus the measured data.

The `inerp1d` does not do extrapolation i.e. it will issue an error if we want to fit the data outside input data range. We can suppress the error by specifying the `bounds_error=None` argument. In this case, it will give `nan`, if we want to interpolate outside input data range. To interpolate outside the input data range, we can use `Rbf` function of the `scipy.interpolate` library. Remember in section 4.4, the interpolated data was only in the range of location of input data. We will use `Rbf` function to interpolate outside these range. We are using `plt.imshow` to make the 2D plot. Fig. 5.16 shows the plot. IT is clear from the figure, that it is able to interpolate outside input data range also.

```
>>> x = np.random.rand(5)
>>> y = np.random.rand(5)
>>> pet = 2+2*np.random.rand(5)
>>>
>>> rbfi = sp.interpolate.Rbf(x, y, pet)  # radial basis function interpolation instance
>>>
>>> xi = np.linspace(0,1)
>>> yi = np.linspace(0,1)
>>> XI, YI = np.meshgrid(xi,yi)  # gridded locations
>>>
>>> di = rbfi(XI, YI)   # interpolated values
```

```
>>>
>>> plt.imshow(di, extent=(0,1,0,1), origin='lower')
>>> plt.scatter(x,y, color='k')
>>> plt.xlabel('X')
>>> plt.ylabel('Y')
>>> plt.axis((0,1,0,1))
>>> plt.savefig('/home/tomer/my_books/python_in_hydrology/images/rbf.png')
```

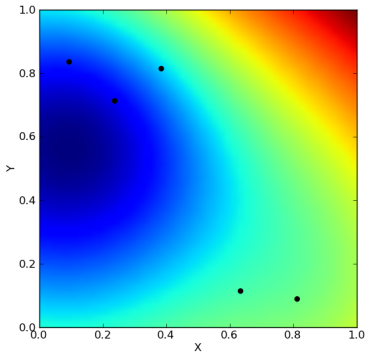

Figure 5.16: Interpolation in 2D.

5.10 Autocorrelation

Autocorrelation is the correlation of a signal with itself. This tells the how a signal is related in time or space. This can be used to see the periodicity in the signal. To demonstrate this, we will first generate a signal using sine with a periodicity of 4π and magnitude of 2. Fig. 5.17 shows the signal in upper panel, and autocorrelation in lower panel. The autocorrelation is plotted using the `plt.acorr` function. We have shown the grid in the plot using the `plt.grid` function. The horizontal lines at 0 and e^{-1} are plotted using the `plt.axhline`. Autocorrelation is showing a nice periodic behaviour with a periodicity of 4π.

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> x = 2*np.sin(np.arange(100)/2.0) # periodic signal
>>> x += np.random.randn(len(x)) # corrupted with noise
>>>
>>> plt.subplot(2,1,1)
>>> plt.plot(x, '-s')
>>> plt.ylabel('x', fontsize=20)
>>> plt.grid(True)
>>> plt.xlabel('Time')
>>>
>>> plt.subplot(2,1,2)
>>> c = plt.acorr(x, usevlines=True, normed=True, maxlags=50, lw=2)
>>> plt.grid(True)
>>> plt.axhline(0, color='black', lw=2)
```

```
>>> plt.axhline(1/np.exp(1), color='red')
>>> plt.ylabel('Autocorrelation')
>>> plt.xlim(xmin=0,xmax=100)
>>> plt.xlabel('Lag')
>>> plt.savefig('/home/tomer/my_books/python_in_hydrology/images/corr_0.png')
```

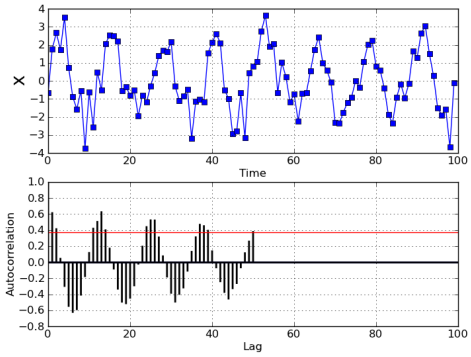

Figure 5.17: A plot showing 100 random numbers generated using sine function, and an autocorrelation of the series.

Autocorrelation function is also used to compute the correlation length. Correlation length is the distance from a point beyond which there is no further correlation of a physical property associated with that point. Mathematically, the correlation length is the lag at which autocorrelation is equal to e^{-1}, which is shown by a horizontal red line in the plot. Let us make a plot with higher periodicity to compute correlation length. The resulted plot is shown in Fig. 5.18. Graphically we see that correlation is approximately 9.

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> x = 2*np.sin(np.arange(100)/10.0) # periodic signal
>>> x += np.random.randn(len(x)) # corrupted with noise
>>>
>>> plt.subplot(2,1,1)
>>> plt.plot(x, '-s')
>>> plt.ylabel('x', fontsize=20)
>>> plt.grid(True)
>>> plt.xlabel('Time')
>>>
>>> plt.subplot(2,1,2)
>>> c = plt.acorr(x, usevlines=True, normed=True, maxlags=50, lw=2)
>>> plt.grid(True)
>>> plt.axhline(0, color='black', lw=2)
>>> plt.axhline(1/np.exp(1), color='red')
>>> plt.ylabel('Autocorrelation')
>>> plt.xlim(xmin=0,xmax=100)
>>> plt.xlabel('Lag')
>>> plt.savefig('/home/tomer/my_books/python_in_hydrology/images/corr_1.png')
```

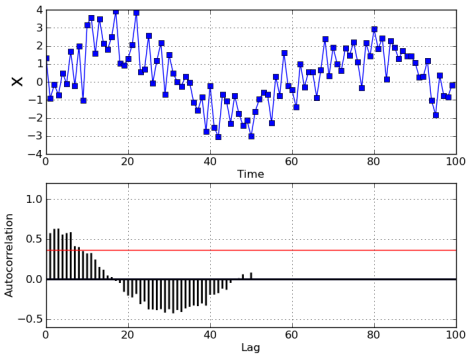

Figure 5.18: A plot showing 100 random numbers generated using sine function, and an autocorrelation of the series.

To precisely determine the correlation length, we would be fitting a interpolation function between lag and correlation length, and then determine the lag where autocorrelation becomes e^{-1}. The `plt.acorr` function return lags and autocorrelation at these lags. First we will assign lags and correlation to separate variables. We also print the lags to see what is inside it.

```
>>> lags = c[0] # lags
>>> auto_corr = c[1] # autocorrelation
>>> print(auto_corr)
[-50 -49 -48 -47 -46 -45 -44 -43 -42 -41 -40 -39 -38 -37 -36 -35 -34 -33
 -32 -31 -30 -29 -28 -27 -26 -25 -24 -23 -22 -21 -20 -19 -18 -17 -16 -15
 -14 -13 -12 -11 -10  -9  -8  -7  -6  -5  -4  -3  -2  -1   0   1   2   3
   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19  20  21
  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39
  40  41  42  43  44  45  46  47  48  49  50]
```

The `acorr` provides positive and negative lags. We don't need both, we can get rid of it by giving a index which is in boolean format and is obtained by using the statement `lags>=0`. We also remove the autocorrelation array which corresponds to negative lags.

```
>>> auto_corr = auto_corr[lags>=0]
>>> lags = lags[lags>=0]
```

Now, we need the autocorrelation at two point, one just above the threshold, and one just below the threshold. We get their indices by counting how many times the auto correlation is above threshold.

```
>>> n = sum(auto_corr>np.exp(-1))
>>> print(n)
9
```

One point is at 8th indices, and another is at 9th indices. Now, we can use `interp1d` to get the value of exact lag when autocorrelation is equal to threshold. This provides the correlation length which is 8.65.

```
>>> f = interp1d([auto_corr[n], auto_corr[n-1]], [lags[n], lags[n-1]])
```

```
>>> corr_len = f(np.exp(-1))
array(8.64725236367028)
```

5.11 Uncertainty Intervals

Many times, we get a number of ensembles of the data. And we want to see the behaviour of these ensemble. Let us first begin by generating ensemble, and then plotting them. In this example, first we are generating a signal with sin behaviour. Then, we are using `vstack` to stack the data i.e. to make 2 dimensional array using many one dimensional arrays. After this, we mix some noise in the data so that ensemble looks slightly different. We are transposing the data using `T` attributes for plotting, otherwise it will plot ensemble on $x-axis$ instead of time.

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> import scipy.stats as st
>>>
>>> # generate some data
>>> x = 100*np.sin(np.linspace(0,10,100))
>>> X = np.vstack([x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x])
>>> e = 10*np.random.randn(20,100)
>>>
>>> X_err = X+e
>>>
>>> plt.plot(X_err.T, 'k')
>>> plt.xlabel('Time')
>>> plt.ylabel('X')
>>> plt.savefig('/home/tomer/my_books/python_in_hydrology/images/X_err.png')
```

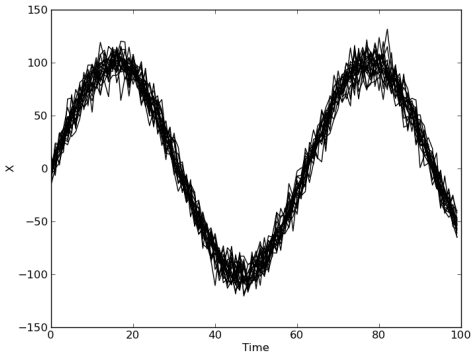

Figure 5.19: Various ensemble of data.

Fig. 5.19 shows the plot of these ensembles. We see that all the ensemble are behaving similar to each one. But we can not infer anything more on the behaviour of ensemble using this plot. A better way to visualize ensemble is by using the various uncertainty intervals along with the mean. We can compute the uncertainty interval at various percentile using the `st.scoreatpercentile` function.

We are computing 10^th, 50^th, and 90^th percentile. 50^th percentile is the median. We are plotting median than the mean, because if there are some outliers in the data, median provided better insight into the behaviour of ensemble. Fig. 5.20 shows the median, 10^th, and 90^th of ensemble. Using this plot, we can make out that the spreads of the ensemble is not same everywhere; it is relatively more in the peak and valley and less elsewhere.

```
>>> ll = st.scoreatpercentile(X_err, 10) # 10th percentile
>>> ml = st.scoreatpercentile(X_err, 50) # 50th percentile
>>> ul = st.scoreatpercentile(X_err, 90) # 90th percentile
>>>
>>> plt.plot(ml, 'g', lw=2, label='Median')
>>> plt.plot(ul, '--m', label='90%')
>>> plt.plot(ll, '--b', label='10%')
>>> plt.xlabel('Time')
>>> plt.ylabel('X')
>>> plt.legend(loc='best')
>>> plt.savefig('/home/tomer/my_books/python_in_hydrology/images/X_uncer.png')
```

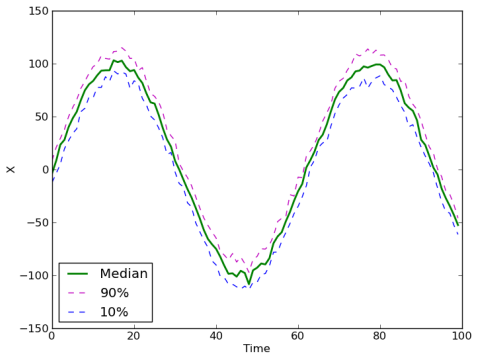

Figure 5.20: The median, and uncertainty intervals of data estimated from ensemble.

The uncertainty intervals could be plotted by shaded regions. The `plt.fill_between` provides the option of filling color in between two array, and can be used to make a shaded regions.

```
>>> plt.plot(ml, 'g', lw=2, label='Median')
>>> plt.fill_between(range(100), ul, ll, color='k', alpha=0.4, label='90%')
>>> plt.xlabel('Time')
>>> plt.ylabel('X')
>>> plt.legend(loc='best')
>>> plt.savefig('/home/tomer/my_books/python_in_hydrology/images/X_uncer_shade.png')
```

Fig. 5.21 shows the plot of uncertainty using the shaded region.

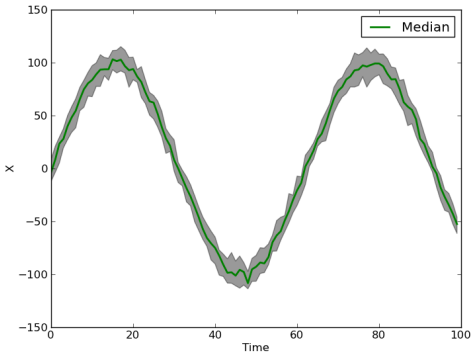

Figure 5.21: The median, and uncertainty intervals of data estimated from ensemble. The uncertainty intervals are shown using shaded region.

Chapter 6

Spatial Data

6.1 Types of spatial data

Raster and vector are the two basic data structures for storing and manipulating images and graphics data in GIS (Geographic Information Systems). Raster image comes in the form of individual pixels, and each spatial location or resolution element has a pixel associated where the pixel value indicates the attribute, such as color, elevation, or an ID number. Vector data comes in the form of points and lines, that are geometrically and mathematically associated. Points are stored using the coordinates, for example, a two-dimensional point is stored as (x, y). Lines are stored as a series of point pairs, where each pair represents a straight line segment, for example, (x1, y1) and (x2, y2) indicating a line from (x1, y1) to (x2, y2).

We will create some raster data using some mathematical function, and then also add noise into it. We will keep both the data (with noise, and without noise) for future use. `np.mgrid` is used to create gridded points. The data is plotted using `plt.matshow` function, which is a simple function to visualize a two dimensional array. Fig. 6.1 shows the data without noise, data corrupted with noise is shown in Fig. 6.2. The data without noise shows a systematic behaviour, while it is blurred in the data added with noise.

```
>>> import numpy as np
>>> # generate some synthetic data
>>> X, Y = np.mgrid[0:101, 0:101]
>>> data = np.sin((X**2 + Y**2)/25)
>>> data_noisy = data + np.random.random(X.shape)
>>>
>>> # plot the data
>>> plt.matshow(data)
>>> plt.colorbar()
>>> plt.savefig('/home/tomer/my_books/python_in_hydrology/images/spatial_data.png')
>>>
>>> plt.matshow(data_noisy)
>>> plt.colorbar(shrink=0.5)
>>> plt.savefig('/home/tomer/my_books/python_in_hydrology/images/spatial_data_noisy.png')
```

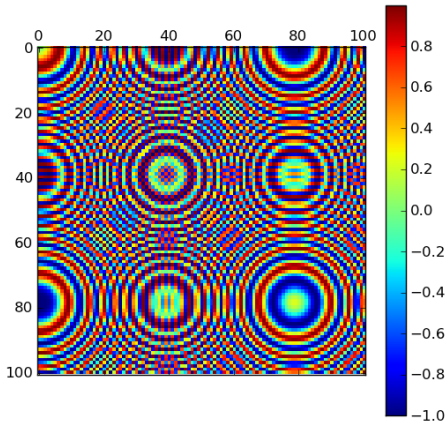

Figure 6.1: Synthetic data created.

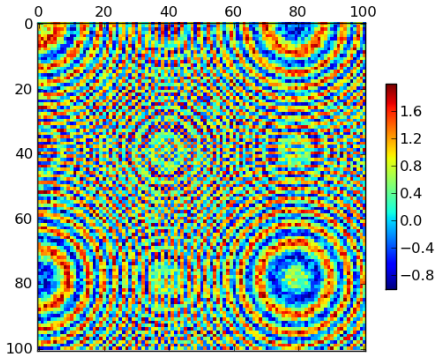

Figure 6.2: Synthetic data perturbed with noise.

We can also generate a vector data with using some points. Fig. 6.3 shows the vector data.

```
>>> # vector data
>>> vector_x = [10,7,24,16,15,10]
>>> vector_y = [10,23,20,14,7,10]
>>>
>>> #plot vector data
>>> plt.clf()
>>> plt.plot(vector_x, vector_y)
>>> plt.axis((5,25,5,25))
>>> plt.savefig('/home/tomer/my_books/python_in_hydrology/images/vect.png')
```

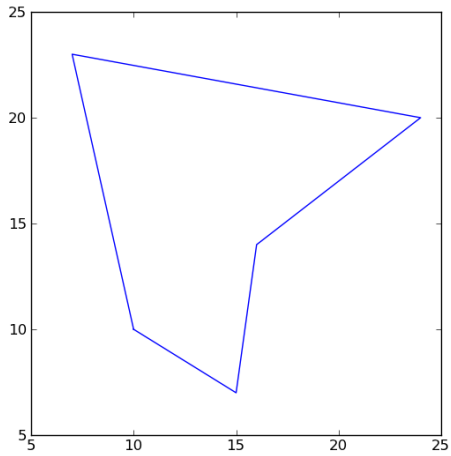

Figure 6.3: Vector data.

The geospatial data can be classified into two major parts. In the first part we have information about some feature, like a two dimensional array showing the spatial variation in elevation etc. In the second part, we have information about the co-ordinates of the data. A typical processing chain for geo-spatial data is given in flow chart 6.4. We have the geospatial data, and we extract the feature information and co-ordinate information separately, then we process them separately, and finally after processing we again combine them. The processing for feature information could be some mathematical operation, and for co-ordinate information, it could some co-ordinate transformation etc.

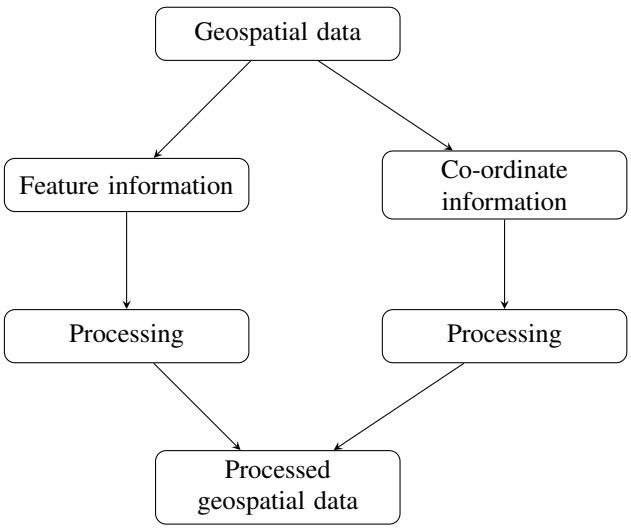

Figure 6.4: Flowchart showing a typical chain for processing the geospatial data.

6.2 Geoinformation

The raster data can be geo-referenced either by specifying the `GeoTransform` variable, or by specifying the GCPs for the image. The GeoTransform (GT) is related to the geographical co-ordinates in the following way,

$$X_{geo} = GT[0] + X_{image} * GT[1] + Y_{image} * GT[2], \tag{6.1}$$

$$Y_{geo} = GT[3] + X_{image} * GT[4] + Y_{image} * GT[5], \tag{6.2}$$

where, subscript *geo* refers to the global co-ordinates, and *image* refers to the image co-ordinates, i.e. pixel and line number of image. The number in the square bracket ([]) represents the indices of the GT variables. X and Y are co-ordinates. $GT[2]$ and $GT[4]$ represents the orientation of the image, with respect to the X and Y of *geo*, and becomes zero if the image is north up. $GT[1]$ represents pixel width, and $GT[5]$ represents the pixel height. $GT[0]$ and $GT[3]$ is the position of the top left corner of the top left pixel of the raster. It should be noted that the pixel/line coordinates in the above are from (0.0,0.0) at the top left corner of the top left pixel to at the bottom right corner of the bottom right pixel. The pixel/line location of the center of the top left pixel would therefore be (0.5,0.5).

The information related to the geo-referencing can be specified by specifying the control points. Control points should contains minimally the GCP_{Pixel} (pixel number of image), GCP_{Line} (line number of image), GCP_X (X co-ordinate), GCP_Y (Y co-ordinate), and GCP_Z (Z co-ordinate). The (Pixel,Line) position is the GCP location on the raster. The (X,Y,Z) position is the associated georeferenced location with the Z often being zero. Usually polynomials are used to transform the image co-ordinate (Pixel,Line) into geo-referenced co-ordinates.

Normally a dataset will contain either an affine geotransform or GCPs. In addition to the information about how co-ordinates are related to the geo-referenced co-ordinates, the name of co-ordinate system is also assigned to the image.

6.3 Writing Raster

There are various format for storing the raster data; Geotiff is the most commonly used. We will use `gdal` library to read and write the raster data. Check if you have installed `gdal` by issuing the command `import gdal`. If you get no error, then things are under control, otherwise go to `http://trac.osgeo.org/gdal/wiki/DownloadSource`, download and install the latest version. Let us first write the the data in GeoTIFF format. First we will write the data without noise.

```
>>> import gdal
>>> driver = gdal.GetDriverByName('GTiff')
>>> file_name = "/home/tomer/my_books/python_in_hydrology/datas/data.tif"
>>> dataset = driver.Create(file_name, data.shape[1], data.shape[0], 1,
... gdal.GDT_Float32)
>>> dataset.SetGeoTransform((664000.0, 100.0, 0.0, 1309000.0, 0.0, -100.0))
>>> dataset.GetRasterBand(1).WriteArray(data, 0, 0)
>>> dataset = None
```

First we created the driver, and asked it to create GTIFF file. Other types of format can also be created. A list of the format supported by gdal, and their code for creating the driver are

listed at http://www.gdal.org/formats_list.html, e.g. code for portable network graphics is PNG. Then we create the database i.e. we create the file in computer, by issuing command dirver.Create. The inputs required to Create are name of the file, size of the data, number of band in the data, format of the data. Then we define the geoinformation by issuing the SetGeoTransform command. And finally we write the data using the method GetRasterBand. It is good practice to close the data with defining the dataset as None. If path for the specified file name does not exist, it returns None, and will give error if other operations are performed over it.

In the similar way, we can write the data corrupted with noise.

```
>>> driver = gdal.GetDriverByName('GTiff')
>>> file_name = "/home/tomer/my_books/python_in_hydrology/datas/data_noisy.tif"
>>> dataset = driver.Create(file_name, data_noisy.shape[1], data_noisy.shape[0], 1, gdal.GDT_Fl
>>> dataset.SetGeoTransform((664000.0, 100.0, 0.0, 1309000.0, 0.0, -100.0))
>>> dataset.GetRasterBand(1).WriteArray(data_noisy, 0, 0)
>>> dataset = None
```

6.4 Writing Vector

Shapefile (.shp) format is quiet commonly used type of vector data. Let us write one shapefile. To write shapefile, we will be using the package ogr. OGR is a part of the GDAL library. OGR deals with the vector formats, while GDAL main library is for raster formats. A list of format supported by OGR along with their code name to be used while creating driver, is given at http://www.gdal.org/ogr/ogr_formats.html. Let us say, we want to make a shapefile having location of the four cities and their name. The details of cities are as:

Name	Latitude	Longitude
Bijnor	29.4	78.1
Delhi	28.6	77.2
Bangalore	13.0	77.8
Berambadi	11.8	76.6

We begin with importing ogr library and defining the location and names of the cities.

```
>>> import ogr
>>> lat = [29.4,28.6,13.0,11.8]
>>> lon = [78.1,77.2,77.8,76.6]
>>> name = ['Bijnor', 'Delhi', 'Bangalore', 'Berambadi']
```

Now, we define the name of driver (ESRI Shapefile), and create the data source. driver.CrateDataSource defines the name of the folder where data will be saved. ds.CreateLayer defines the name of the shapefile along with the geometry type (point in this case). Then we define, field name as 'Name' and say that it is a string type having a maximum width of 16.

```
>>> driver = ogr.GetDriverByName("ESRI Shapefile")
>>> ds = driver.CreateDataSource('/home/tomer/my_books/python_in_hydrology/datas/')
>>> layer = ds.CreateLayer('location', geom_type=ogr.wkbPoint)
>>> field_defn = ogr.FieldDefn('Name', ogr.OFTString )
>>> field_defn.SetWidth(16)
>>> layer.CreateField(field_defn)
```

Now, we have the basic information ready, and we can start adding the information about cities (name and location). First we create a feature to store the information about city. Then, we add the name of the city in the 'Name' field. After this, we say that it is point type, and we add its longitude and latitude. At last, we destroy the feature and data source, so that nothing else can be done with them, and our data is saved properly.

```
>>> i = 0
>>> for i in range(len(name)):
>>>     feature = ogr.Feature(layer.GetLayerDefn())
>>>     feature.SetField('Name', name[i])
>>>     pt = ogr.Geometry(ogr.wkbPoint)
>>>     pt.SetPoint_2D(0, lon[i], lat[i])
>>>     feature.SetGeometry(pt)
>>>     layer.CreateFeature(feature)
>>>     feature.Destroy()
>>> ds.Destroy()
```

We can see this shapefile in any GIS viewere. Fig. 6.5 shows the location of cities which was generated using QGIS.

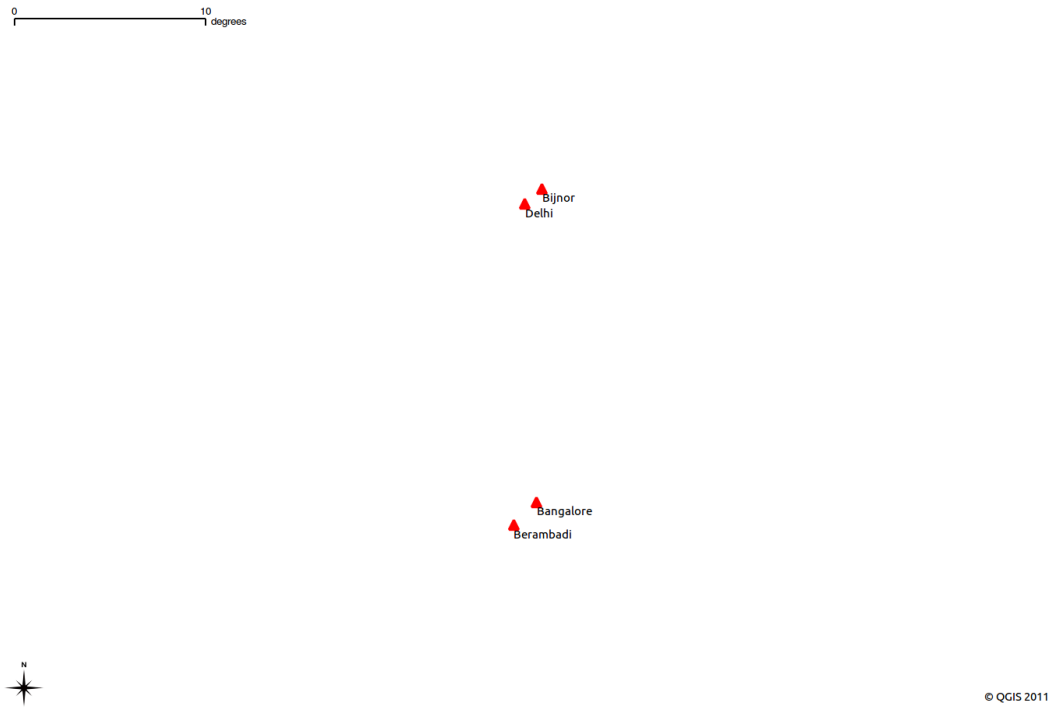

Figure 6.5: Location of the cities.

6.5 Reading the raster

In this section, we will read the data that we wrote in earlier section. We will read the raster data that has noise in it.

```
>>> driver = gdal.GetDriverByName('GTiff')
>>> file_name = "/home/tomer/my_books/python_in_hydrology/datas/data_noisy.tif"
>>> dataset = gdal.Open(file_name, GA_ReadOnly)
>>> geotransform = dataset.GetGeoTransform()
>>> data = dataset.GetRasterBand(1).ReadAsArray()
>>> dataset = None
```

6.6 Read the vector

In this section, we will read the vector data, that we wrote earlier. First, we need to import `ogr` library. Then we open data source by specifying the directory of the shapefile. Then, we use `GetLayerByName` to read the shapefile by specifying the name of shpaefile without .shp extension. After this, we are printing `\n` which means print a blank line. `\n` represents a new line, and `\t` represents the tab. Now, we are printing header information (SI., Name, Latitude, Longitude). We are using `.format` to format the output. The number inside `'{}'` after the colon (:) represents the length of the output. The we read the feature in the shapefile one by one using the `for` loop. From each feature, we extract the name using `GetFieldAsString`, Longitude using `GetX` and Latitude using `GetY`. At the end to avoid corrupting the database, we close it safely by specifying data source as `None`.

```
>>> import ogr
>>> ds = ogr.Open( '/home/tomer/my_books/python_in_hydrology/datas/' )
>>> lyr = ds.GetLayerByName('location' )
>>>
>>> print("\n")
>>> print("{} \t {:10s} \t {} \t {}".format('SI', 'Name', 'Longitude', 'Latitude'))
>>> for feat in lyr:
>>>     geom = feat.GetGeometryRef()
>>>     name = feat.GetFieldAsString(0)
>>>     lat = geom.GetX()
>>>     lon = geom.GetY()
>>>     print('{0} \t {1:10s} \t {2:.3f} \t \t {3:.3f}'.format(0, name, lat, lon ))
>>>
>>> ds = None

SI  Name         Latitude   Longitude
0   Bijnor       78.100     29.400
0   Delhi        77.200     28.600
0   Bangalore    77.800     13.000
0   Berambadi    76.600     11.800
```

6.7 Filtering

The active RADAR data is affected by speckle/noise. Before we extract some useful information from the satellite data, we need to remove/minimize these speckle from the data. These filters essentially try to remove the high frequency information. In reality this high frequency information need not to be noise. So we need to specify how much filtering we want. The types of filter can be divided into two categories: adaptive and non adaptive filters. Adaptive filters adapt their

weightings across the image to the speckle level, and non-adaptive filters apply the same weightings uniformly across the entire image. In this section, we will be using one example of both category. We will use median filter from the non-adaptive filter category, and Wiener filter from the adaptive category.

We can import the `medfilt2d` function from the `scipy.signal` library. First, we read the noisy data that we saved in tif format from hard disk. Then we apply a filter to it having window size of 3×3. Fig. 6.6 shows the filtered image. When, we compare this image with the original image, and with the image in which we mixed some noise, we see that filtered images showed more smooth variation, but is not showing no where near to the original image.

```
>>> from osgeo import gdal
>>> from scipy.signal import medfilt2d
>>> from osgeo.gdalconst import *
>>> import matplotlib.pyplot as plt
>>>
>>> # read the raster data
>>> driver = gdal.GetDriverByName('GTiff')
>>> file_name = "/home/tomer/my_books/python_in_hydrology/datas/data_noisy.tif"
>>> dataset = gdal.Open(file_name, GA_ReadOnly)
>>> geotransform = dataset.GetGeoTransform()
>>> data = dataset.GetRasterBand(1).ReadAsArray()
>>> dataset = None
>>>
>>> data_median = medfilt2d(data, kernel_size=3) # median filter of 3X3 window
>>> # plot the data
>>> plt.matshow(data_median)
>>> plt.colorbar()
>>> plt.savefig('/home/tomer/my_books/python_in_hydrology/images/median.png')
```

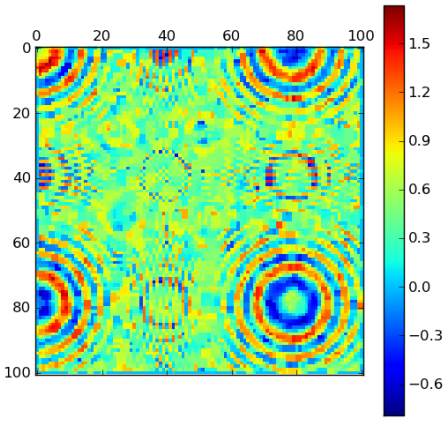

Figure 6.6: Noisy data after filtering with median filter.

Now, we apply Wiener filter on the same dataset. We keep the same window size for Wiener filter also. After doing the filtering, we are also saving the data in tif format. This tell use, how can read some geospatial data, process the feature information (matrix in this case), and then save the feature information with the co-ordinate information.

```
>>> from scipy.signal import wiener
>>> data_wiener = wiener(data, mysize=(3,3)) # Wiener filter
>>> # plot
>>> plt.matshow(data_wiener)
>>> plt.colorbar()
>>> plt.savefig('/home/tomer/my_books/python_in_hydrology/images/wiener.png')
>>>
>>> # save the data into tif format
>>> driver = gdal.GetDriverByName('GTiff')
>>> file_name = "/home/tomer/my_books/python_in_hydrology/datas/data_filtered.tif"
>>> dataset = driver.Create(file_name, data_wiener.shape[1], data_wiener.shape[0], 1, gdal.GDT_
>>> dataset.SetGeoTransform(geotransform)
>>> dataset.GetRasterBand(1).WriteArray(data_wiener, 0, 0)
>>> dataset = None
```

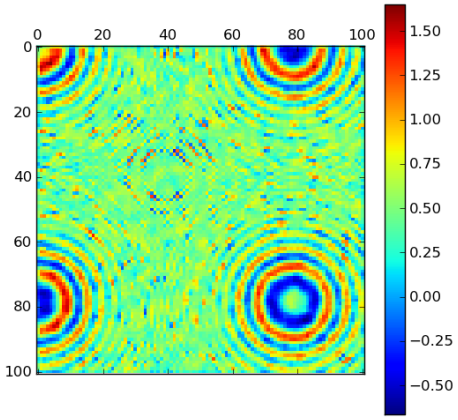

Figure 6.7: Noisy data after filtering with Wiener filtering.

6.8 NDVI

Normalized Difference Vegetation Index (NDVI) is a index to analyse variation in the vegetation. The formula to compute NDVI is as:

$$NDVI = \frac{NIR - RED}{NIR + RED} \tag{6.3}$$

We begin with importing libraries, and do not forget to import `division` from `__future__` library. This is to avoid the integer division. Then we read the data that is in tiff format. To compute NDVI, we need data for NIR (band4) and RED (band3).

```
>>> # import the required library
>>> from __future__ import division
>>> from osgeo import gdal
>>> from osgeo.gdalconst import *
>>> import matplotlib.pyplot as plt
>>>
>>> # read the banda 3 raster data
>>> driver = gdal.GetDriverByName('GTiff')
>>> file_name = "/home/tomer/my_books/python_in_hydrology/datas/band3.tif"
>>> dataset = gdal.Open(file_name, GA_ReadOnly)
>>> geotransform = dataset.GetGeoTransform()
>>> projection = dataset.GetProjection()
>>> band3 = dataset.GetRasterBand(1).ReadAsArray()
>>> dataset = None
>>> # read the band 4 raster data
>>> file_name = "/home/tomer/my_books/python_in_hydrology/datas/band4.tif"
>>> dataset = gdal.Open(file_name, GA_ReadOnly)
>>> band4 = dataset.GetRasterBand(1).ReadAsArray()
>>> dataset = None
```

Apart from the data, we are also retrieving the `geotransform` and `projection` information. Let us print them one by one.

```
>>> print(geotransform)
(76.5, 0.001, 0.0, 11.85, 0.0, -0.001)
```

The first entry in this tell us, that latitude and longitude for the north-west corner are 11.85 and 76.5 respectively. Resolution of data is 0.001 in both x and y direction, and the image has no rotation. Let us print now, projection information.

```
>>> print(projection)
GEOGCS["WGS 84",DATUM["unknown",SPHEROID["WGS84",6378137,298.257223563]],
PRIMEM["Greenwich",0],UNIT["degree",0.0174532925199433]]
```

This tells us that the datum of data is WGS84, and our data is in geographic co-ordinates (latitude and longitude). More details on the projections and their parameters can be found at `http://spatialreference.org`.

We can check the data type using `dtype` attributes. We see that data type integer. That is why we have imported `division` from `__future__` library, to avoid integer division.

```
>>> print(band3.dtype)
uint8
```

Now we compute NDVI, and plot the result using `matshow`. In `matshow`, we are also specifying `vmin` and `vmax` to control the extent of `colorbar`.

```
>>> ndvi = (band4-band3)/(band4+band3)
>>>
>>> plt.matshow(ndvi,cmap=plt.cm.jet, vmin=-1, vmax=1)
>>> plt.colorbar(shrink=0.8)
>>> plt.savefig('/home/tomer/my_books/python_in_hydrology/images/ndvi.png')
```

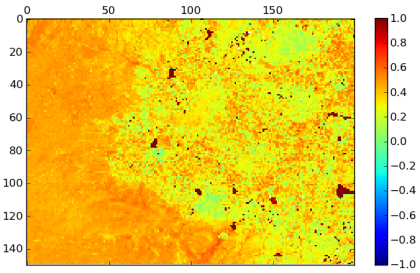

Figure 6.8: Normalized difference vegetation index.

Chapter 7

Plotting

First problem that we had during the plotting was the font size. We can change the font size by specifying the fontsize method of pyplot, but for this we need to specify everywhere the fontsize. This becomes cumbersome task, especially when you want to make many plots. There is another way to come out of this problem, by updating the `prcParams` method of plt. The updating is done in the following manner.

```
>>> import matplotlib.pyplot as plt
>>> params = {'axes.labelsize': 17,
>>>           'text.fontsize': 17,
>>>           'legend.fontsize': 17,
>>>           'xtick.labelsize': 17,
>>>           'ytick.labelsize': 17,
>>>           'text.usetex': False,
>>>           'font.size':17}
>>>
>>> plt.rcParams.update(params)
```

The name of the parameters are self explaining. If we want to keep the same font size for all kind of text (e.g. ticks, labels, legend etc.), we can simply update `text.fontsize`. It is a good practice to write this before making plots and define the value which suits you. If later you want to change font size for some attribute of plot, you can define that particular value there. We will be using these fontsizes in all plots henceforth, but will not be adding this text into each code for brevity.

7.1 Date axis

Most of the data in hydrology is time series data. To plot such data, it is required that $x-axis$ should be time axis, and on $y-axis$ the data should be plotted. We will be using `timeseries` library of `scikits` package to deal with time series data, which also provides functions to plot time series data. We will generate some data at regular (daily) interval which is like having data simulated using some model, and some data at random time which is like having measurements at irregular interval. The measurement data also contains corresponding vectors for time i.e. year, month and day.

```
>>> import scikits.timeseries as ts
>>> x = np.arange(500)
>>> y_sim = np.sin(x/25.0)
>>> year = [2009, 2010, 2010, 2010, 2011]  # year for measurement
>>> month = [10, 2, 5, 9, 1]   # month for measurement
>>> day = [20, 15, 17, 22, 15]  # day for measurement
>>> y_meas = [0.4, -0.5, 0, 1, 0]
```

We begin with creating the time series for regular data. To do so, first we define start date, then we use this start date and simulated data to make a timeseries object using `ts.timeseries`.

```
>>> first_date = ts.Date(freq='D',year=2009,month=10,day=05)
>>> data_series = ts.time_series(y_sim, start_date=first_date)
```

As measured data is not at regular frequency, we can not define it so easily. We need to make date objects for each date, and then we can create time series object.

```
>>> date = []
>>> for i in range(len(year)):
>>>     date.append(datetime.date(year[i], month[i], day[i]))
>>> meas_series = ts.time_series(y_meas, dates=date,freq='D')
```

`scikits.timeseries` provides `lib.plotlib` to make plots of time series. The library has similar functions to make plots and hence easy to adopt. Fig. 7.1 shows the plot having time axis.

```
>>> import scikits.timeseries.lib.plotlib as tpl
>>> fig = tpl.tsfigure()
>>> fsp = fig.add_tsplot(111)
>>> fsp.tsplot(data_series, 'r', lw=3, label='simulated')
>>> fsp.tsplot(meas_series, 'g*', ms=20, label='measured')
>>> fsp.set_ylim(-2, 2)
>>> fsp.grid()
>>> plt.ylabel('Data')
>>> fsp.legend(loc='best')
>>> plt.savefig('/home/tomer/my_books/python_in_hydrology/images/date.png')
>>> plt.close()
```

7.2 Bar charts

Let us first create some data. We will crate two variables (rainfall and runoff) of length 5.

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> n = 5
>>> rainfall_mean = 500+300*np.random.rand(n)
>>> runoff_mean = 0.75*np.random.rand(n)*rainfall_mean
```

The `bar` requires the corresponding value for x, and it does not calculate by itself. We are also specifying the width of bars. The `plt.bar` returns rectangle patches which can be used to modify

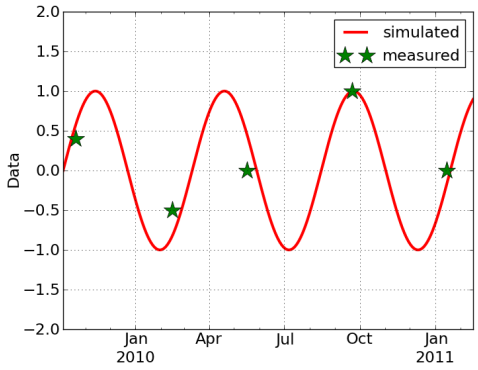

Figure 7.1: Plot showing dates for x-axis.

patches or to extract some information from them. In this example these rectangle patches are used to extract their height, and then to place text at the top of bars. Fig. 7.2 shows the bar plot.

```
>>> ind = np.arange(n)  # the x locations for the groups
>>> width = 0.35       # the width of the bars
>>>
>>> rects1 = plt.bar(ind, rainfall_mean, width, color='g', label='Rainfall')
>>> rects2 = plt.bar(ind+width, runoff_mean, width, color='m', label='Runoff')
>>>
>>> plt.ylabel('Annual sum (mm)')
>>> plt.title('Water balance')
>>> plt.xticks(ind+width, ('2001', '2002', '2003', '2004', '2005') )
>>>
>>> def autolabel(rects):
>>>     # attach some text labels
>>>     for rect in rects:
>>>         height = rect.get_height()
>>>         plt.text(rect.get_x()+rect.get_width()/2., 1.05*height, '%d'%int(height),
>>>                 ha='center', va='bottom')
>>>
>>> plt.legend()
>>> autolabel(rects1)
>>> autolabel(rects2)
>>>
>>> plt.ylim(ymax=1000)
>>> plt.savefig('/home/tomer/my_books/python_in_hydrology/images/bar.png')
>>> plt.close()
```

7.3 Pie charts

First we generate three variables, runoff, recharge and evapotranspiration by assuming some rainfall.

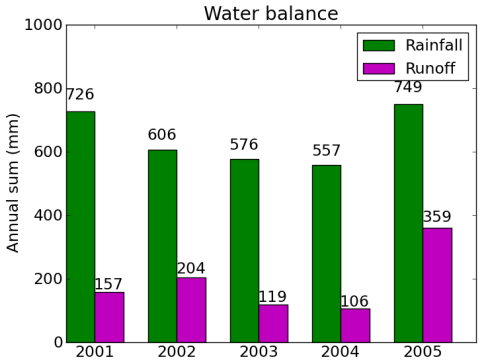

Figure 7.2: Plot showing an example of bar.

```
>>> rainfall = 1000
>>> runoff = 0.5*np.random.uniform()*rainfall
>>> recharge = 0.2*np.random.uniform()*rainfall
>>> evapotranspiration = rainfall - runoff - recharge
```

We modify the figure size by its default size to make the plot square. Then, we define the are for pie chart by specifying the `plt.axis`.

```
>>> plt.figure(figsize=(8,8))
>>> plt.axis([0.2, 0.2, 0.8, 0.8])
```

We make a list of the variables, and tuple for the name of variables.

```
>>> labels = 'Runoff', 'Recharge', 'Evapotranspiration'
>>> fracs = [runoff, recharge, evapotranspiration]
```

We can use explode to highlight some of the variable, in this case 'Recharge'. The amount of the explode is controlled by its parameters. The `autopct` is used to define the format of the number inside pie. Fig. 7.3 shows the pie chart.

```
>>> explode=(0, 0.1, 0)
>>> plt.pie(fracs, explode=explode, labels=labels, autopct='%1.1f%%', shadow=True)
>>> plt.title('Annual water balance', bbox={'facecolor':'0.6', 'pad':10})
>>> plt.savefig('/home/tomer/my_books/python_in_hydrology/images/pie.png')
```

7.4 2 D plots

The images (JPEG, TIFF, PNG etc.) comes in two formats: greyscale or RGB. `plt.imshow` can be used to show both type of images. First we use `imread` to read the data from figures, then we make a three dimensional array data by first creating an empty array, and then specifying the each band data in RGB (red, green blue) order. If we are reading an image having all the three bands in a single image, the `imread` will provide the data as three dimensional array and can be used directly.

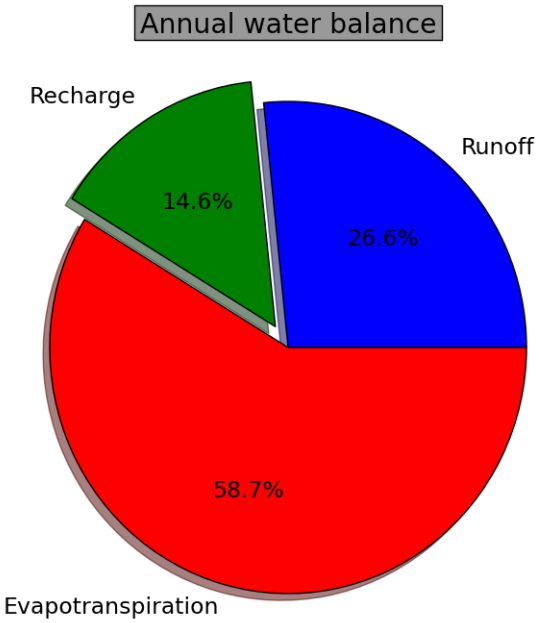

Figure 7.3: Plot showing an example of pie with explode.

The `imshow` is used to make the 2 dimensional plot with interpolation algorithm type to change its default interpolation type. Fig. 7.7 shows two dimensional map generated using the `imshow`.

```
>>> band2 = plt.imread('/home/tomer/my_books/python_in_hydrology/datas/band2.tif')
>>> band3 = plt.imread('/home/tomer/my_books/python_in_hydrology/datas/band3.tif')
>>> band4 = plt.imread('/home/tomer/my_books/python_in_hydrology/datas/band4.tif')
>>>
>>> foo = np.empty((band2.shape[0], band2.shape[1], 3))
>>> foo[:,:,2] = band2
>>> foo[:,:,1] = band3
>>> foo[:,:,0] = band4
>>>
>>> plt.imshow(foo, interpolation='hanning')
>>> plt.savefig('/home/tomer/my_books/python_in_hydrology/images/imshow.png')
```

`pcolor` stands for pseudo color, and is used to increase the contrast in the data while making plots. The `colorbar` is used to show the colormap. The type of colormap is controlled by using the `cmap` as input to `pcolor`. Fig. 7.5 shows the pseudo color plot.

```
>>> plt.pcolor(band2, cmap=plt.cm.Paired)
>>> plt.colorbar()
>>> plt.ylim(ymax=band2.shape[0])
>>> plt.savefig('/home/tomer/my_books/python_in_hydrology/images/pcolor.png')
```

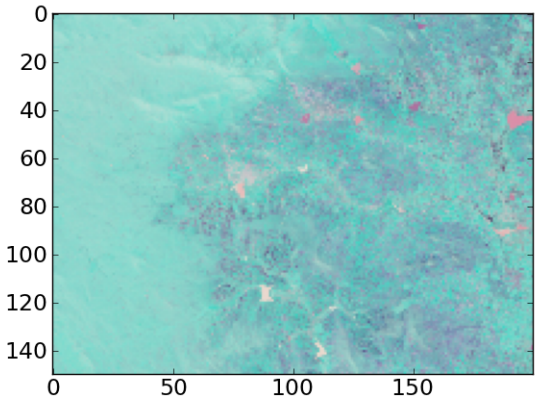

Figure 7.4: Two dimensional plot using the imshow.

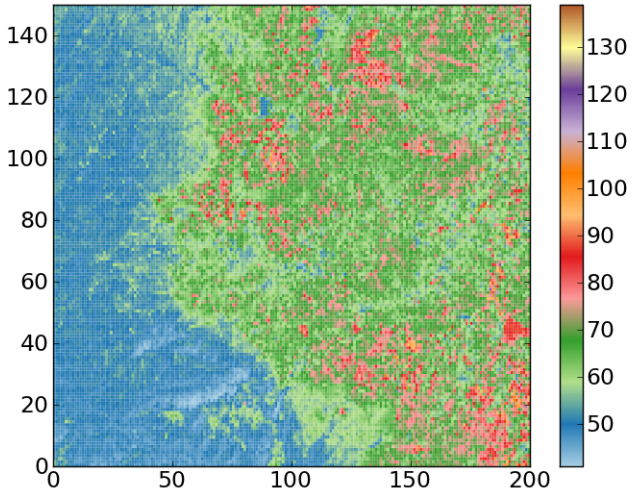

Figure 7.5: Two dimensional plot using the pcolor.

We will be using the band2 data to make the contours. Since the 'band2' data has very hight spatial variability, we will be first filtering it using median filter.

```
>>> from scipy.signal import medfilt2d
>>> data = medfilt2d(band2, kernel_size=7)
```

`plt.contour` is used to make contours. By default it does not show the contour values, to show the contour labels, we use the `plt.clabel`. Fig. 7.6 shows the contour plot along with contour labels.

```
>>> CS = plt.contour(data,10)
>>> plt.clabel(CS, inline=1, fontsize=10)
>>> plt.savefig('/home/tomer/my_books/python_in_hydrology/images/contour.png')
```

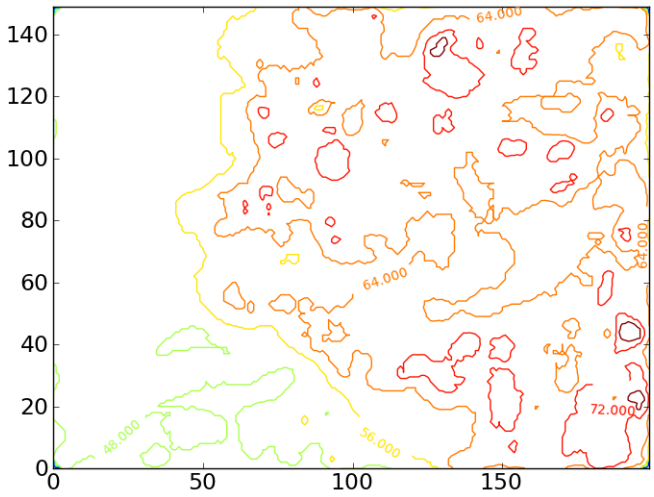

Figure 7.6: Plot showing the contour along with contour labels.

`plt.contour` provides the empty contour, i.e. there is no color between successive contours. We can use `contourf` to make filled contour plots. Fig. 7.7 shows the filled contour plot.

```
>>> plt.contourf(data,10)
>>> plt.colorbar()
>>> plt.savefig('/home/tomer/my_books/python_in_hydrology/images/contourf.png')
```

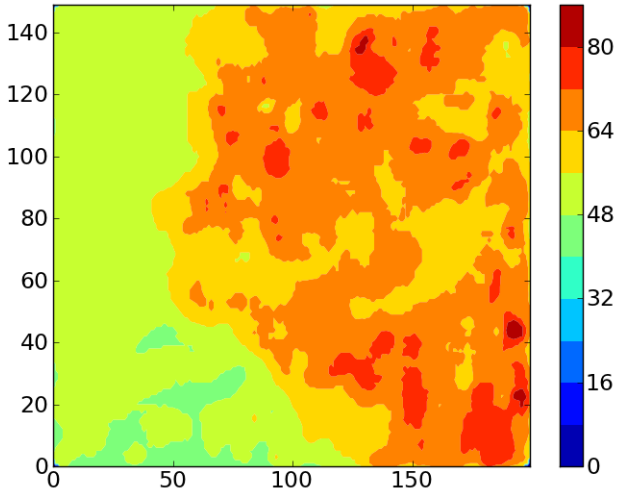

Figure 7.7: Filled contour plotted using the contourf.

7.5 3 D plots

To make three dimensional plot, we need to import `Axes3D` library from the `mpl_toolkits.mplot3d`. The scatter or line plot in three dimension is made in the way similar to two dimension. We will generate three variables, and make the three dimensional scatter plot. Fig. 7.8 shows the three dimensional scatter plot.

```
>>> import numpy as np
>>> from mpl_toolkits.mplot3d import Axes3D
>>>
>>> x = np.random.randn(100)
>>> y = np.random.randn(100)
>>> z = np.random.randn(100)
>>>
>>> fig = plt.figure()
>>> ax = fig.add_subplot(111, projection='3d')
>>> ax.scatter(x, y, z, color='k', marker='s')
>>>
>>> ax.set_xlabel('x')
>>> ax.set_ylabel('y')
>>> ax.set_zlabel('z')
>>> plt.savefig('/home/tomer/my_books/python_in_hydrology/images/3dscatter.png')
>>> plt.close()
```

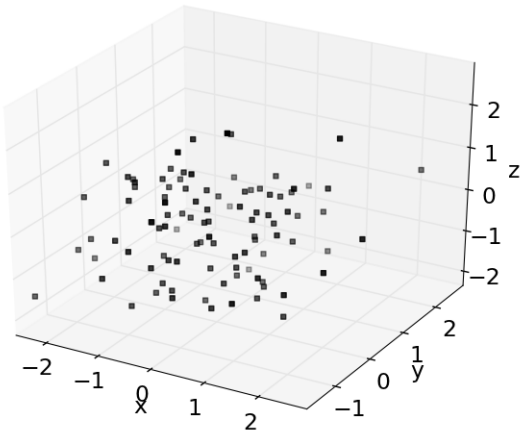

Figure 7.8: Date axis

7.6 Box-plot

Box-plot is a way to graphically visualize the statistical properties of the data. It provides information about the minimum, first quartile (Q1), median (Q2), upper quartile (Q3), maximum, and

outliers if present. `boxplot` is used to make boxplot. Fig. **??** shows the box plot.

```
>>> n = 4
>>> x = range(n)
>>> y = 5+np.random.randn(100,4)
>>>
>>> plt.boxplot(y,'gD')
>>> plt.savefig('/home/tomer/my_books/python_in_hydrology/images/boxplot.png')
```

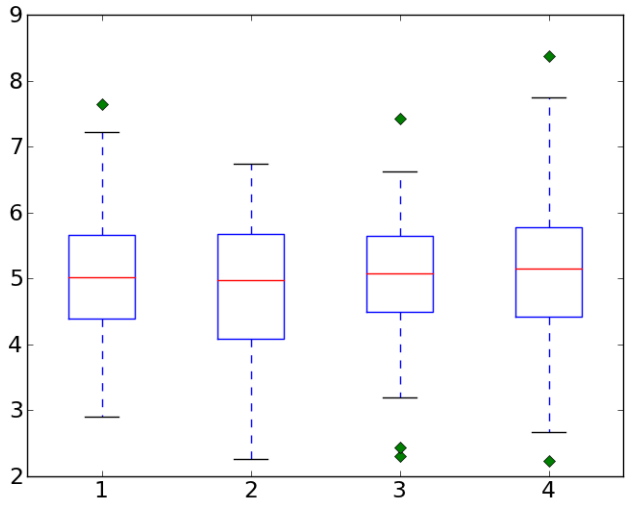

Figure 7.9: Box plot of data.

7.7 Q-Q plot

The Q-Q (quantile-quantile) plot is a graphical method of comparing two probability distributions by plotting their quantiles against each other. We will use `statistics` library to calculate the quantiles. We will generate three random variable, two having same (normal) distributions, and one having different (uniform) distribution, and will compare their behaviour on the Q-Q plot. Fig. 7.10 shows the Q-Q plot. On the x-axis we have the quantiles for normal distribution, on y-axis we are plotting quantiles of uniform and normal distributions. We see that when the distributions are same, they fall on 1:1 line, otherwise they depart from it.

```
>>> import statistics as st
>>> from scipy.interpolate import interp1d
>>>
>>> def Q(data):
>>>     F, data1 = st.cpdf(data, n=1000)
>>>     f = interp1d(F, data1)
>>>     return f(np.linspace(0,1))
>>>
```

```
>>> x = np.random.randn(1000)
>>> y = 5*np.random.rand(1000)
>>> z = np.random.randn(1000)
>>>
>>> Qx = Q(x)
>>> Qy = Q(y)
>>> Qz = Q(z)
>>>
>>> plt.plot([-5,5] , [-5,5], 'r', lw=1.5, label='1:1 line')
>>> plt.plot(Qx, Qy, 'gd', label='Uniform')
>>> plt.plot(Qx, Qz,  'm*', label='Normal')
>>> plt.axis((-5, 5, -5, 5))
>>> plt.legend(loc=2)
>>> plt.xlabel('Normal')
>>> plt.ylabel('observed')
>>> plt.savefig('/home/tomer/my_books/python_in_hydrology/images/q_q.png')
```

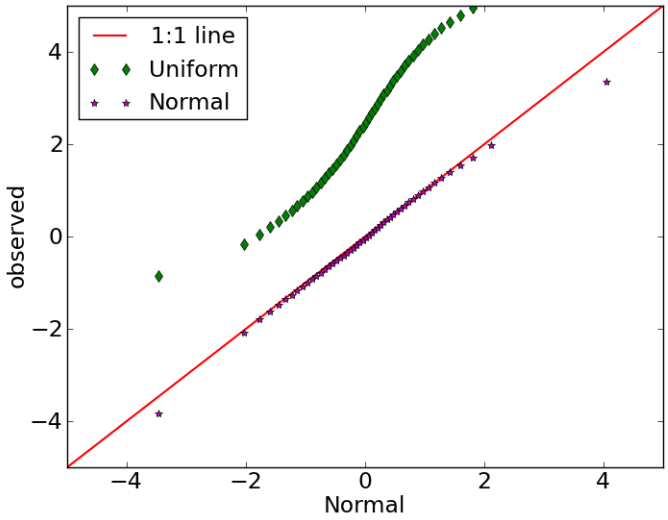

Figure 7.10: Quantile-quantile plot.

7.8 plotyy

If we want to plot two variables in the same plot which have different range (minimum and maximum), then we should not plot them using same axis. If we do so, we will not be able to see the variation in one variable. Fig. 7.11 shows the plot having two y axis.

```
>>> fig = plt.figure()
>>> plt.subplots_adjust(top=0.9,bottom=0.15,left=0.15,right=0.85)
>>> ax1 = fig.add_subplot(111)
>>>
```

```
>>> t = np.linspace(0,10)
>>> y1 = 5*np.sin(t)
>>> y2 = 10*np.cos(t)
>>>
>>> ax1.plot(t, y1, 'g', label='sin')
>>> ax1.set_xlabel('time (s)')
>>>
>>> #Make the y-axis label and tick labels match the line color.
>>> ax1.set_ylabel('sin', color='b')
>>> for tl in ax1.get_yticklabels():
>>>     tl.set_color('b')
>>> ax1.set_ylim(-6,6)
>>> plt.legend(loc=3)
>>>
>>> ax2 = ax1.twinx()
>>> ax2.plot(t, y2, 'r', label='cos')
>>> ax2.set_ylabel('cos', color='r')
>>> for tl in ax2.get_yticklabels():
>>>     tl.set_color('r')
>>> ax2.set_ylim(-15,15)
>>>
>>> plt.legend(loc=4)
>>> plt.savefig('/home/tomer/my_books/python_in_hydrology/images/multiple_y.png')
```

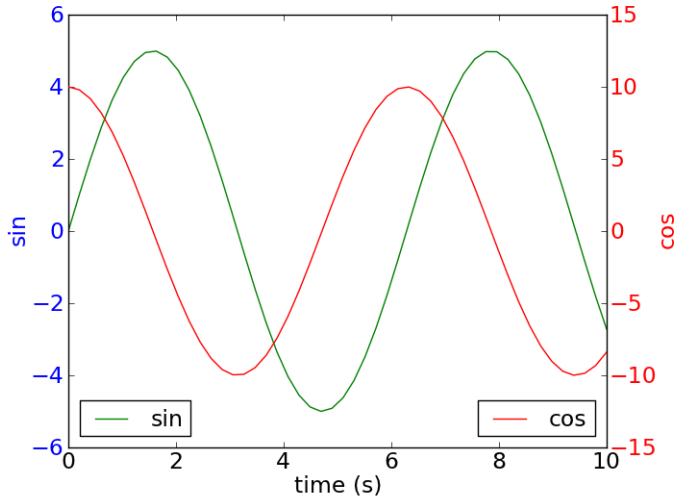

Figure 7.11: Plot showing multiple y-axis.

7.9 Annotation

Apart from plotting lines, dot, legends etc., we may need to put additional information on the plot. This is called annotation. We can put different arrow, texts etc. to make our graph more clear. Fig. 7.12 shows one such figure having few arrows, texts to improve the readability of the graph.

```
>>> t = np.linspace(0,10)
>>> y = 5*np.sin(t)
>>>
>>> plt.plot(t, y, lw=3, color='m')
>>> plt.xlabel('x')
>>> plt.ylabel('y')
>>> plt.annotate('ridge', xy=(np.pi/2, 5),  xycoords='data',
>>>             xytext=(-20, -75), textcoords='offset points',
>>>             arrowprops=dict(arrowstyle="->")
>>>             )
>>>
>>> plt.annotate('valley', xy=(1.5*np.pi, -5),  xycoords='data',
>>>             xytext=(-30, 80), textcoords='offset points',
>>>             size=20,
>>>             bbox=dict(boxstyle="round,pad=.5", fc="0.8"),
>>>             arrowprops=dict(arrowstyle="->",),
>>>             )
>>>
>>> plt.annotate('Annotation', xy=(8, -5),  xycoords='data',
>>>                 xytext=(-20, 0), textcoords='offset points',
>>>                 bbox=dict(boxstyle="round", fc="green"), fontsize=15)
>>>
>>> plt.text(3.0, 0, "Down", color = "w", ha="center", va="center", rotation=90,
>>>             size=15, bbox=dict(boxstyle="larrow,pad=0.3", fc="r", ec="r", lw=2))
>>>
>>> plt.text(np.pi*2, 0, "UP", color = "w", ha="center", va="center", rotation=90,
>>>             size=15, bbox=dict(boxstyle="rarrow,pad=0.3", fc="g", ec="g", lw=2))
>>>
>>> plt.savefig('/home/tomer/my_books/python_in_hydrology/images/annotate.png')
```

7.10 Basemap

The `Basemap` library of `mpl_toolkits.basemap` provides options for showing some variable on the globe, showing boundaries (hydrological, political etc.) in most commonly used projections. We will use plot the band1 data with the boundary of the Berambadi watershed. Fig. 7.13 shows the band1 data with boundary of watershed marked in white.

```
>>> from mpl_toolkits.basemap import Basemap
>>> import gdal
>>> from gdalconst import *
>>>
>>> # read the data
```

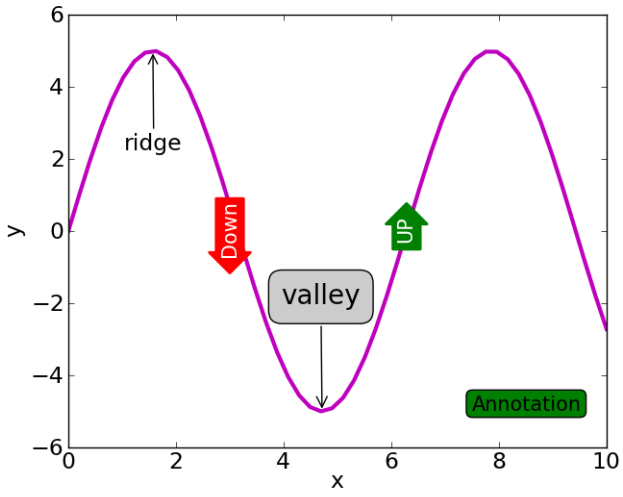

Figure 7.12: Plot showing various annotations.

```
>>> dataset = gdal.Open("/home/tomer/my_books/python_in_hydrology/datas/band1.tif",GA_ReadOnly)
>>> band1 = dataset.GetRasterBand(1).ReadAsArray()
>>> GT = dataset.GetGeoTransform()
>>>
>>> dataset = None
>>>
>>> # make the co ordinate for the berambadi
>>> lon = np.linspace(GT[0]+GT[1]/2, GT[0]+GT[1]*(band1.shape[1]-0.5), band1.shape[1])
>>> lat = np.linspace(GT[3]+GT[5]/2, GT[3]+GT[5]*(band1.shape[0]-0.5), band1.shape[0])
>>> Lon, Lat = np.meshgrid(lon, lat)
>>>
>>> # make the base map
>>> m = Basemap(projection='merc',llcrnrlat=11.72,urcrnrlat=11.825,\
>>>             llcrnrlon=76.51,urcrnrlon=76.67,lat_ts=20,resolution=None)
>>>
>>> # draw parallels and meridians.
>>> m.drawparallels(np.arange(11.7,11.9,.05),labels=[1,0,0,0])
>>> m.drawmeridians(np.arange(76.4,76.8,.05),labels=[0,0,0,1])
>>>
>>> # read the shapefile archive
>>> s = m.readshapefile('/home/tomer/my_books/python_in_hydrology/datas/berambadi','berambadi'
>>>     color='w',linewidth=2.5)
>>>
>>> # compute native map projection coordinates of lat/lon grid.
>>> x, y = m(Lon,Lat)
>>>
>>> # contour data over the map
>>> cs = m.pcolor(x,y,band1,cmap=plt.cm.jet)
>>> cb = plt.colorbar(cs, shrink=0.6, extend='both')
```

```
>>>
>>> plt.title(" Band 1")
>>> plt.savefig('/home/tomer/my_books/python_in_hydrology/images/basemap.png')
```

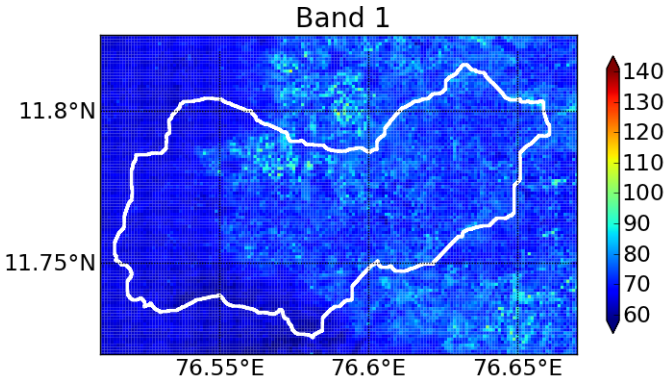

Figure 7.13: Spatial variation of band1 along with boundary of Berambadi watershed.

7.11 Shared axis

Often it is required to make two or more plots having the same axis (x or y or both). The `plt.subplots` provides an easy way to make the common (shared) axis. First, we will generate the synthetic data having different range. Then plot using the `plt.subplots`. The other options in `plt.subplots` are similar to `plt.subplot`.

```
>>> x1 = range(100)
>>> x2 = range(125)
>>>
>>> y1 = np.random.rand(100)
>>> y2 = 2.0*np.random.rand(125)
>>> y3 = np.random.rand(125)
>>> y4 = 1.5*np.random.rand(100)
>>>
>>> fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, sharex=True, sharey=True)
>>> ax1.plot(x1,y1, 'ro')
>>> ax2.plot(x2,y2, 'go')
>>> ax3.plot(x2,y3, 'bs')
>>> ax4.plot(x1,y4, 'mp')
>>> plt.tight_layout()
>>> plt.savefig('/home/tomer/my_books/python_in_hydrology/images/shared_xy.png')
```

Fig. 7.14 shows the resulted plot.

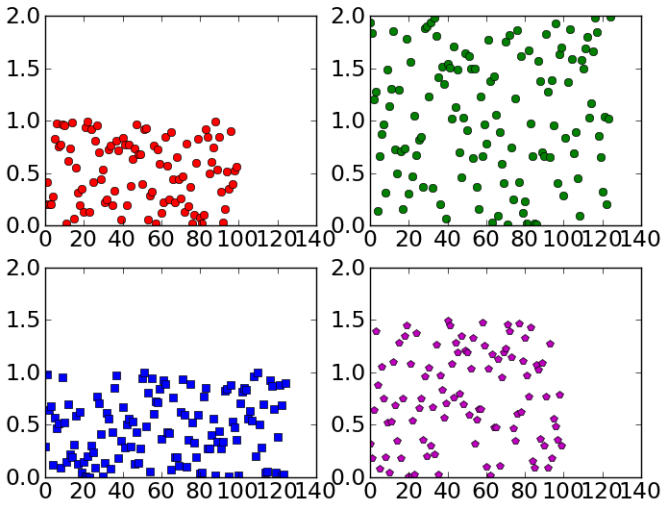

Figure 7.14: Plots having the common (shared) axis.

7.12 Subplot

So far, we have used subplot having same width and height. The situation might arise, when we need to increase the size for some subplot. In the following section, we will try to plot such in such case. Firs, we will use `plt.subplot`, itself to make some particular subplot.

```
>>> x = np.random.rand(25)
>>> y = np.arccos(x)
>>>
>>> plt.close('all')
>>> plt.subplot(221)
>>> plt.scatter(x,y)
>>>
>>> plt.subplot(223)
>>> plt.scatter(x,y)
>>>
>>> plt.subplot(122)
>>> plt.scatter(x,y)
>>> plt.tight_layout()
>>> plt.savefig('/home/tomer/my_books/python_in_hydrology/images/sub_plot1.png')
```

Fig. 7.15 shows the resulted plot. The `plt.tight_layout()` increase the readability of the ticks labels. If this option is not used, then you might have got figures with overlapping labels etc. This options prevents overlapping of axis, title etc.

Now, we will make these kinds of subplot using `subplot2grid`.

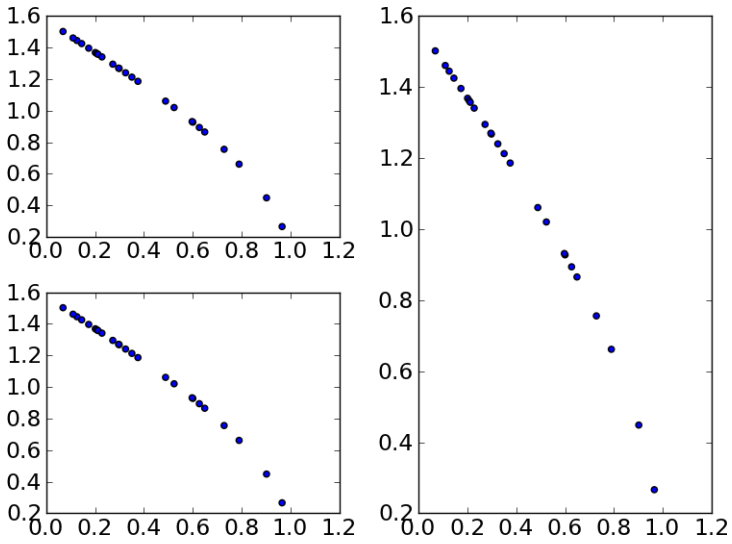

Figure 7.15: Plot with varying span of subplots.

```
>>> fig = plt.figure()
>>> fig.subplots_adjust(wspace=0.5, hspace=0.4)
>>>
>>> ax1 = plt.subplot2grid((3, 3), (0, 0))
>>> ax2 = plt.subplot2grid((3, 3), (0, 1), colspan=2)
>>> ax3 = plt.subplot2grid((3, 3), (1, 0), colspan=2, rowspan=2)
>>> ax4 = plt.subplot2grid((3, 3), (1, 2), rowspan=2)
>>>
>>> ax1.scatter(10*x,y)
>>> ax2.scatter(10*x,y)
>>> ax3.scatter(10*x,y)
>>> ax4.scatter(10*x,y)
>>> plt.savefig('/home/tomer/my_books/python_in_hydrology/images/sub_plot2.png')
```

Fig. 7.16 shows the resulted plot.

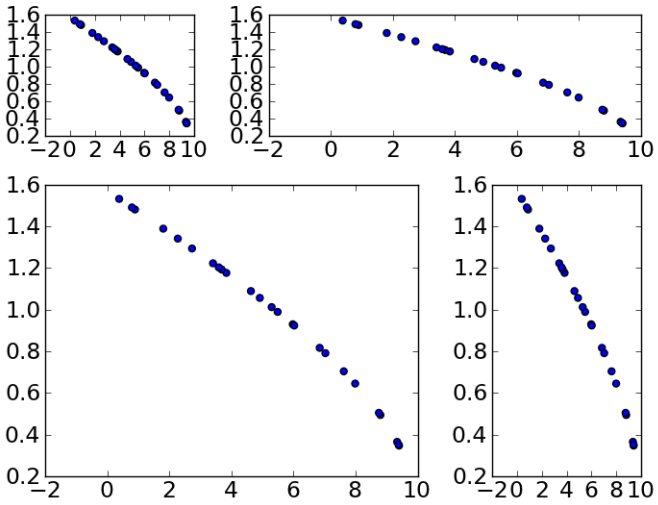

Figure 7.16: Plot with varying span of subplots plotted using subplot2grid.

Chapter 8

Input-Output

8.1 xls

The `data.xls` file contains the data of soil moisture estimated from the AMSR-E platform. You can open the xls file and have a look at its content. In this file we have data in two sheets, Ascending and Descending which corresponds to satellite direction. Each sheet contains the time series data for various grids point. Missing data is assigned a number of 999.9. In this section we will read data of one station for all the time, modify the data which is missing, and write in another xls file. We will be using `xlsrd` library to read data from xls file, and `xlwt` to write the data to xls file. The `xlrd` does not read `xlsx` data file, you should convert the `xlsx` type of file into `xls` before reading.

```
>>> import xlrd
>>> import numpy as np
```

We create book object by passing the name of xls file to `xlrd.open_workbook`. The sheet from which we need to read the data is specified using the `sheet_by_name`.

```
>>> book = xlrd.open_workbook('/home/tomer/my_books/python_in_hydrology/datas/data.xls')
>>> sheet = book.sheet_by_name('Ascending')
```

The number of columns and rows in sheets can be checked by using the `nrows` and `ncols` attributes respectively.

```
>>> sheet.nrows
1100
>>> sheet.ncols
39
```

Our sheet's first two rows are heading of table and latitude and longitude, and hence the length of time series data is two lesser than the number of rows. First we create an empty array to store the data, and then we read the data cell by cell using the `cell_value`. We will be reading the data of grid having latitude equal to 12.4958 and longitude equal to 75.7484, which is in fourth column (indices start with zero).

```
>>> sm = np.empty(sheet.nrows-2)
```

```
>>> year = np.empty(sheet.nrows-2, int)
>>> month = np.empty(sheet.nrows-2, int)
>>> day = np.empty(sheet.nrows-2, int)
>>> for i in range(sm.shape[0]):
>>>    sm[i] = sheet.cell_value(i+2,27)
>>>    year[i] = sheet.cell_value(i+2,0)
>>>    month[i] = sheet.cell_value(i+2,1)
>>>    day[i] = sheet.cell_value(i+2,2)
```

We can check the data of some variable e.g. sm.

```
>>> sm
array([  16.6,  999.9,   15.3, ...,   17.4,  999.9,   18.2])
```

We can define all the missing data as nan.

```
>>> sm[sm==999.9] = np.nan
>>> sm
array([ 16.6,   nan,  15.3, ...,  17.4,   nan,  18.2])
```

Now the soil moisture data has nan instead of 999.9 to denote missing values. We will write this soil moisture data into xls file using `xlwt` library. First we open a workbook, then we add a sheet by name using `add_sheet`. After this we start writing entries cell by cell. Finally, we save the worksheet using `book.save`.

```
>>> import xlwt
>>> book = xlwt.Workbook()
>>> sheet = book.add_sheet('Ascending')
>>> sheet.write(0,0, 'Year')
>>> sheet.write(0,1, 'Month')
>>> sheet.write(0,2, 'Day')
>>> sheet.write(0,3, 'Latitude')
>>> sheet.write(1,3, 'Longitude')
>>>
>>> for i in range(len(sm)):
>>>     sheet.write(i+2, 4, sm[i])
>>>     sheet.write(i+2, 0, year[i])
>>>     sheet.write(i+2, 1, month[i])
>>>     sheet.write(i+2, 2, day[i])
>>>
>>> book.save('/home/tomer/my_books/python_in_hydrology/datas/data1.xls')
```

I have written a library `ambhas.xls` which provides relatively easy way to read and write the xls data. The data can be read in the following way.

```
>>> from ambhas.xls import xlsread
>>> fname = '/home/tomer/my_books/python_in_hydrology/datas/data.xls'
>>> foo = xlsread(fname)
>>> data = foo.get_cells('a3:a5', 'Ascending')
```

The data to `xls` file is written in the following way. The data which is written should be a numpy array.

```
>>> from ambhas.xls import xlswrite
>>> fname = '/home/tomer/my_books/python_in_hydrology/datas/data.xls'
>>> foo = xlswrite(data, 'a3', 'Ascending')
>>> foo.save(fname)
```

As this library depends upon the `xlrd`, it also does not read `xlsx` data file, and you should convert the `xlsx` type of file into `xls` before reading.

8.2 Text file

Some of the software/tools take the input form text files and then write to text files. If we want to do bath processing (process many files) using these tools, then we need to modify the input text files and extract the information from the file written by the tool. In this section we will read a file, and then change some parameter of the file, and then write it.

Before jumping into many thing, first we will just read the text file. The 'r' is for reading, 'w' is for 'writing' and 'a' is for appending into existing file.

```
>>> fname_read = '/home/tomer/my_books/python_in_hydrology/datas/Albedo.prm'
>>> f_read = open(fname_read, 'r')
>>> for line in f_read:
>>>     print line
>>> f_read.close()
NUM_RUNS = 1



BEGIN

INPUT_FILENAME = /home/tomer/data/input.hdf

OBJECT_NAME = MOD_Grid_BRDF|

FIELD_NAME = Albedo_BSA_Band1

BAND_NUMBER = 1

SPATIAL_SUBSET_UL_CORNER = ( 13.0 75.0 )

SPATIAL_SUBSET_LR_CORNER = ( 11.0 78.0 )

RESAMPLING_TYPE = BI

OUTPUT_PROJECTION_TYPE = UTM

ELLIPSOID_CODE = WGS84

UTM_ZONE = 0
```

```
OUTPUT_PROJECTION_PARAMETERS = ( 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0  )

OUTPUT_FILENAME = /home/tomer/data/output.tif

OUTPUT_TYPE = GEO

END
```

We see that file has many parameters defined, which will be used by some tool to process input hdf images. Let us say, we have four input files, and we want to batch process them.

```
>>> fname_input = ['input0', 'input1', 'input2', 'input3']
>>> fname_output = ['output0', 'output1', 'ouput2', 'output3']
>>>
>>> for i in range(len(fname_input)):
>>>
>>>     fname_read = '/home/tomer/my_books/python_in_hydrology/datas/Albedo.prm'
>>>     f_read = open(fname_read, 'r')
>>>
>>>     fname_write = '/home/tomer/my_books/python_in_hydrology/datas/Albedo_ii.prm'.replace('ii',str(i
>>>     f_write = open(fname_write, 'w')
>>>     for line in f_read:
>>>         if 'INPUT_FILENAME' in line:
>>>             line = line.replace('input',fname_input[i])
>>>             print line
>>>         if 'OUTPUT_FILENAME' in line:
>>>             line = line.replace('output',fname_output[i])
>>>             print line
>>>         f_write.write(line)
>>>
>>>     f_write.close()
>>>
>>> f_read.close()
```

8.3 NetCDF

In this section, we will read a NetCDF file, and write in the same format. I am using the file, *rhum.2003.nc* which can be downloaded from `http://www.unidata.ucar.edu/software/netcdf/examples/files.html`. We will be using `NetCDF` library from `Scientific.IO` to read and write the NetCDF data, so lets first import it.

```
>>> import numpy as np
>>> from Scientific.IO import NetCDF as nc
```

First, we open the file.

```
>>> file = nc.NetCDFFile('/home/tomer/my_books/python_in_hydrology/datas/rhum.2003.nc', 'r')
```

We can look at its attributes by using `dir`.

```
>>> dir(file)
['Conventions', 'base_date', 'close', 'createDimension', 'createVariable', 'description',
'flush', 'history', 'platform', 'sync', 'title']
```

The `title` tells about the title of dataset, description provides the description of the content of file.

```
>>> file.title
'mean daily NMC reanalysis (2003)'
>>> file.description
'Data is from NMC initialized reanalysis\n(4x/day).  It consists of most variables interpolated
```

We can look at the dimension of the data.

```
>>> file.dimensions
{'lat': 73, 'time': None, 'lon': 144, 'level': 8}
```

We see that, the data has four dimensions: lat, time, lon, and level. The size of each dimensions is also given. Now, we can look at the variables in the data.

```
>>> file.variables
{'lat': <NetCDFVariable object at 0x1d33270>, 'rhum': <NetCDFVariable object at 0x1d33030>, 't
```

This provides, the name of variables and a reference of the variable to the data. This means that this does not load the data into memory, in fact just provide a reference in the file, and we can retrieve only the variable that we want. We shall get the value of 'rhum' variable. First we the reference to some variable name. Then we can see its unit, data type, and get its value.

```
>>> foo = file.variables['level']
>>> foo.units
'%'
>>> foo.typecode
'h'
>>> rhum = foo.getValue
```

Now, we can look at the shape of the variable 'rhum'.

```
>>> rhum.shape
(365, 8, 73, 144)
```

The first dimension represent the time, second represent the various pressure levels, and third represent the latitude, and the last one is longitude.

We can write the file in the same way. First we open the file for writing.

```
>>> file = nc.NetCDFFile('/home/tomer/my_books/python_in_hydrology/datas/test.nc', 'w')
```

Then we can define some global attributes like title, description etc.

```
>>> setattr(file, 'title', 'trial')
>>> setattr(file, 'description', 'File generated while tesing to write in NetCDF')
```

Now, we can create some dimensions. We need to define the name of dimension and their size.

```
>>> file.createDimension('lat', 73)
>>> file.createDimension('lon', 144)
>>> file.createDimension('level', 8)
>>> file.createDimension('time', 365)
```

Now, we can save the variables. First, we need to define the dimension from the list created above. The dimension should be tuple, notice the comma after the `'lat',` . After this, we can create variable using `createVariable`, we need to specify the name of variable, format and dimension. We see that it has created a variable named 'lat' and is referring to it.

```
>>> varDims = 'lat',
>>> lat = file.createVariable('lat', 'f', varDims)
>>> print(file.variables)
{'lat': <NetCDFVariable object at 0x2c39078>}
```

Finally, we can assign our data to this variable.

```
>>> lat = np.random.rand(73)
```

Now, we can close the file.

```
>>> file.close()
```

8.4 Pickle

`Pickle` format is very fast to read and write. But it is only useful when you want to keep data for yourself, e.g. write data from one program, and read the same data into another program. First we import `cPicle` and call it `pickle`.

```
>>> import cPickle as pickle
```

We define one variable e.g. a list and first save it and then read it.

```
>>> var = [2, 5, 8, 'foo']
```

We use `pickle.dump` to save the data.

```
>>> var = [2, 5, 8, 'foo']
>>> pickle.dump(var, open( "/home/tomer/my_books/python_in_hydrology.pkl", "wb" ) )
>>>
>>> var1 = pickle.load( open( "/home/tomer/my_books/python_in_hydrology.pkl", "rb" ) )
>>> print(var1)
[2, 5, 8, 'foo']
```

Chapter 9

Numerical Modelling

9.1 Integration

Suppose we want to integrate some function and the function can not be integrated analyticall, then we go for numerical integration. To check if our numerical integration shcems is working properly or not, we integrate the function for which analytical solution is available and then we compare our solution. So let us begin with function of one variable. Suppose, we have function $f(x) = x$ to integrate over limit 0 to 10. We know from mathematics that the answer is 50. Let us now try numerical methods about the solution. We will use `integrate` library of `scipy`. For simple function we an use `lambad` function instead of `def` to define the function. The function `integrate.quad` perform our task. It returns the integration and error in integration.

```
>>> from scipy import integrate
>>> y_fun = lambda x: x
>>> y,err = integrate.quad(x2,0,10)
>>> print(y,err)
(50.0, 5.551115123125783e-13)
```

We get the 50.0 as answer which is exactly the analytical solution, and also it says that error is the solution is very low. It could be by chance, that we get accurate solution from the numerical scheme. So let us try one more function, this time exponential. We will integrate $f(x) = exp(-x)$ over 0 to ∞.

```
>>> y_func = lambda x: np.exp(-x)
>>> y = integrate.quad(y_func, 0, np.inf)
>>> print(y)
(1.0000000000000002, 5.842606742906004e-11)
```

We see that the solution is very near to the analytical solution (1.0). These function had only the variable as input, but we may want to have additional parameter in the function, e.g. $f(x) = exp(-ax$, and assume $a = 0.5$ for this example. The `integrate.quad` provides options for providing additional argument also.

```
f = lambda x,a : np.exp(-a*x)
y, err = integrate.quad(f, 0, 1, args=(0.5,))
```

```
>>> y
0.786938680574733
```

9.2 ODE

Let us solve the ordinary differential equation, given as:

$$\frac{dy}{dt} = -xt, \tag{9.1}$$

with,

$$y_0 = 10. \tag{9.2}$$

First, we can import required libraries.

```
>>> import numpy as np
>>> from scipy import integrate
>>> import matplotlib.pyplot as plt
```

Now, we define our function, and the timer at which we want the solution.

```
>>> y1 = lambda x,t : -x*t
>>> t = np.linspace(0,10)
```

Now, we can use `integrate.odeint` to solve the ordinary differential equation. Then, we can make a plot of the solution with respect to the time.

```
>>> y = integrate.odeint(y1, 10, t)
>>> # plot
>>> plt.plot(t,y)
>>> plt.xlabel('x')
>>> plt.ylabel('y')
>>> plt.savefig('/home/tomer/my_books/python_in_hydrology/images/ode.png')
```

Fig. 9.2 shows the variation of y over time.

Let us, solve a system of ordinary differential equation given as,

$$\frac{dx}{dt} = Ax, \tag{9.3}$$

where,

$$A = \begin{bmatrix} -D_1 & D_1 & 0 \\ D_1 & -D_1 - D_2 & D_2 \\ 0 & D_2 & -D_3 \end{bmatrix} \tag{9.4}$$

We begin with defining the parameters and A matrix.

```
>>> D = [0.2, 0.1, 0.3]
>>> A = np.array([[D[0], -D[0], 0],
```

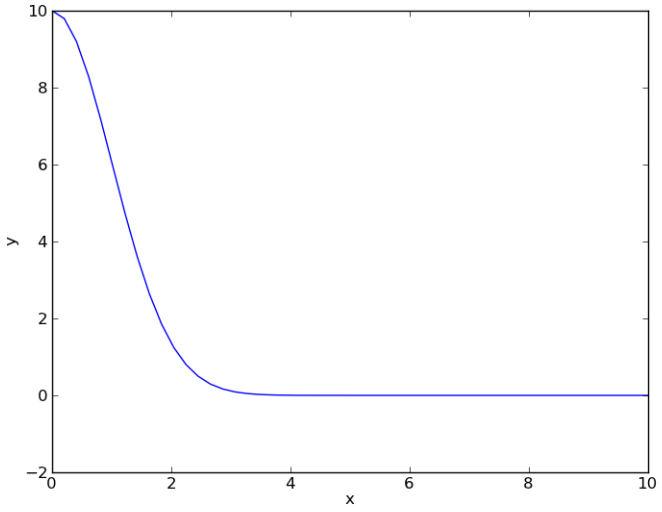

Figure 9.1: Variation of y over time.

```
>>>              [D[0], -D[0]-D[1], D[1]],
>>>              [0, D[2], -D[2]]])
```

Now, we can define our function dx/dt.

```
>>> def dX_dt(sm, t=0):
>>>     return np.dot(A,sm)
```

Finally, we define time, initial condition, use `integrate.odeint` to solve, and then plot.

```
>>> t = np.linspace(0, 10, 100)
>>> X0 = np.array([10, 5, 20])
>>> X, infodict = integrate.odeint(dX_dt, X0, t, full_output=True)
>>>
>>> plt.plot(t,X)
>>> plt.xlabel('Time')
>>> plt.ylabel('X')
>>> plt.legend(['X1','X2','X3'])
>>> plt.savefig('/home/tomer/my_books/python_in_hydrology/images/ode_system.png')
```

9.3 Parameter Estimation

```
>>> from scipy import optimize, special
>>> x = np.arange(0,10,0.01)
>>> for k in np.arange(0.5,5.5):
>>>     y = special.jv(k,x)
```

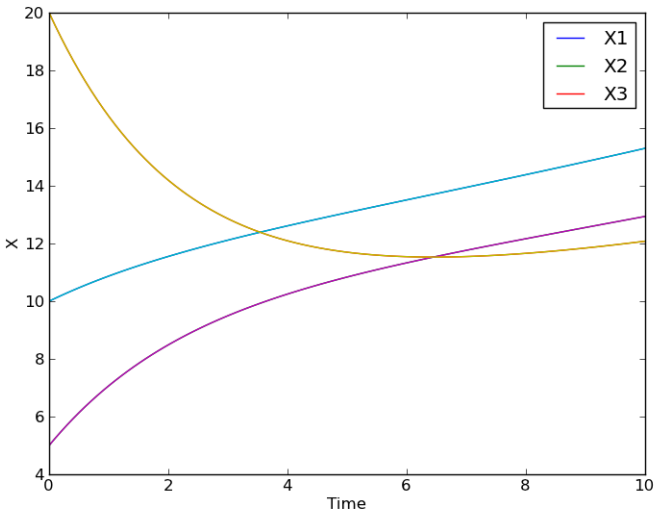

Figure 9.2: Variation of x over time.

```
>>>
>>> f = lambda x: -special.jv(k,x)
>>> x_max = optimize.fminbound(f,0,6)
>>>
>>> plt.plot(x,y, lw=3)
>>> plt.plot([x_max], [special.jv(k,x_max)],'rs', ms=12)
>>> plt.title('Different Bessel functions and their local maxima')
>>> plt.savefig('/home/tomer/my_books/python_in_hydrology/images/inverse.png')
>>> plt.close()
```

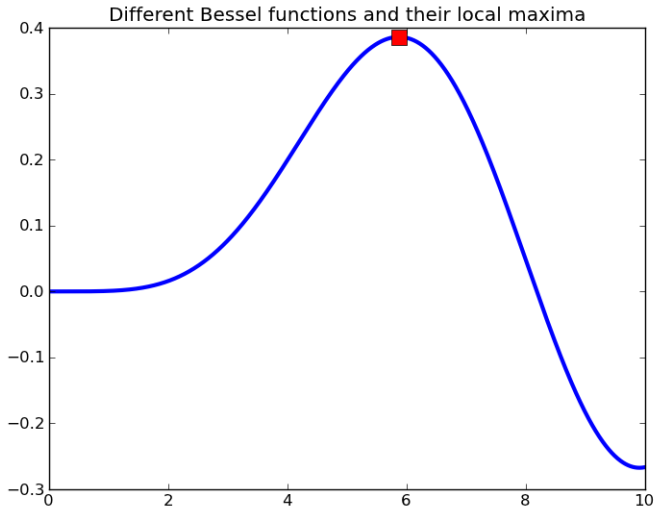

Figure 9.3: Demonstration of inverse modelling.

Chapter 10

Advance statistics

10.1 copula

Copulas are used to describe the dependence between random variables. Copula means coupling two CDFs. Let us generate two random variables; one having normal distribution, another combination of first one and uniform distribution.

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> from matplotlib.ticker import NullFormatter
>>>
>>> # synthetic data
>>> x = np.random.randn(1000)
>>> y = np.random.randn(1000)
```

First we would like to how is our data related by using scatter plot, and also we would like to see how is the distribution of x and y. We can do this in three separate plots, or using subplots. In the present case we will trying this in one plot by specifying different axis for these 3 plots. We begin with defining the axis limits for our three plots. The input to `axis` are x and y for the lower left corner, width and height of the plot. in the following example we are specifying axis in such as way so that plots are aligned properly.

```
>>> plt.clf()
>>> axScatter = plt.axes([0.1, 0.1, 0.5, 0.5])
>>> axHistx = plt.axes([0.1, 0.65, 0.5, 0.3])
>>> axHisty = plt.axes([0.65, 0.1, 0.3, 0.5])
```

Now, we use this axis to make plots.

```
>>> # the plots
>>> axScatter.scatter(x, y)
>>> axHistx.hist(x)
>>> axHisty.hist(y, orientation='horizontal')
>>>
>>> # set the limit of histogram plots
>>> axHistx.set_xlim( axScatter.get_xlim() )
```

```
>>> axHisty.set_ylim( axScatter.get_ylim() )
>>>
>>> plt.savefig('/home/tomer/my_books/python_in_hydrology/images/copula_1.png')
```

Fig. 10.1 shows the resulted plot.

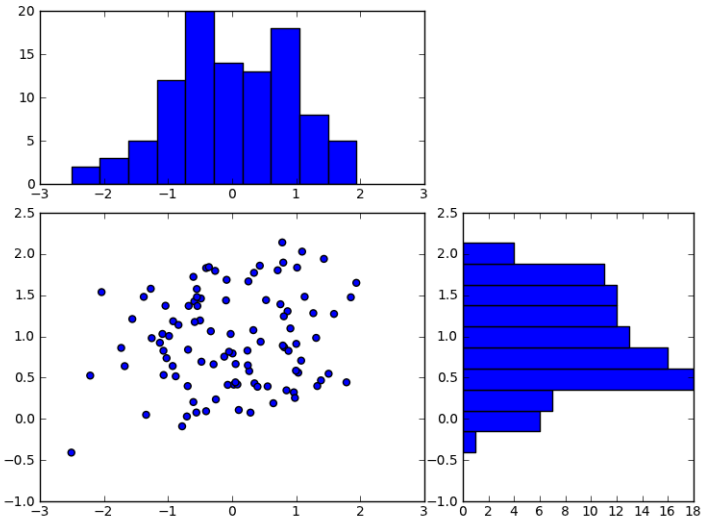

Figure 10.1: Scatter plot along with marginal histograms.

Now, let us try to simulate ensemble of data using copula. I have written a library, `ambhas.copula` to deal with copulas. This library has three copulas (Frank, Clayton, and Gumbel) in it. First we import the library, then we initialize the class.

```
>>> from ambhas.copula import Copula
>>> Copula(x, y, 'frank')
```

We can get the value of Kendall's tau, and the parameter of Frank copula by attributes `tau` and `theta` respectively.

```
>>> print(foo.tau)
0.179797979798
>>> print(foo.theta)
1.66204833984
```

We can generate the ensemble using Frank copula.

```
>>> x1,y1 = foo.generate_xy()
```

Now, we can plot the simulated data with original data.

```
>>> plt.scatter(x1,y1, color='g')
>>> plt.scatter(x,y, color='r')
```

```
>>> plt.xlabel('x')
>>> plt.ylabel('y')
>>> plt.savefig('/home/tomer/my_books/python_in_hydrology/images/copula_2.png')
```

Fig. 10.2 shows the resulted plot.

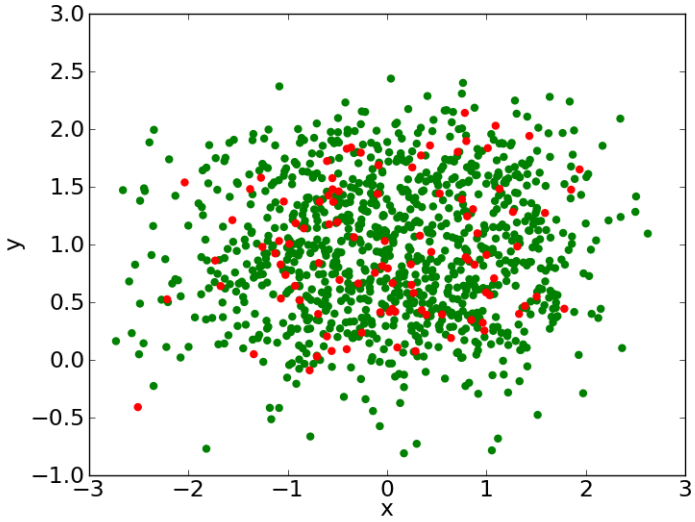

Figure 10.2: Simulated ensemble along with original data.

10.2 Multivariate distribution

So far, we have generated random variables with only univariate distribution. In this section we will be generating multivariate nominally distributed random variable by specifying the mean and covariance matrix to `np.random.multivariate_normal`.

```
>>> mean = [0,5]
>>> cov = [[1,0.4],[0.4,1]]
>>> data = np.random.multivariate_normal(mean,cov,5000)
```

We can check its mean and covariance.

```
>>> print(data.mean(axis=0))
[ 0.00814047  5.00406784]
>>> print(np.corrcoef(data.T))
[[ 1.          0.40707918]
 [ 0.40707918  1.        ]]
```

We see that generated random variable had mean and covariance close to the specified input.

Appendix A

Install library

A.1 Basemap

The installation of new library looks very easy by use of `pip`, but in reality it is not true. Each library has its own dependency, and the way to get those dependency, and hence needs special way to install them. We used `Basemap` in making maps. So lets install this library. This library is available for download from `http://sourceforge.net/projects/matplotlib/files/matplotlib-toolkits/`. Download the latest version of basemap from this weblink. There are some *.exe files meant to be install in Windows. I have not tired them, if you are window you can try them first, if these does work for some reason, then you can download source (*.tar.gz) file.

After downloading any new library, first you should try,

```
>>> sudo pip install /path/to/*.tar.gz
```

if this works, then installation is really easy.

Now, unzip/untar the downloaded *.tar.gz file. If there is `setup.py` file in the directory, then you should run the following command (after going into the folder),

```
>>> sudo python setup.py install
```

If this fails, or there is no file, `setup.py` then, you should read either `readme` or `install` file. Atleast one of them will tell, how to install the library. In the case of `basemap` library, we see that some instructions are given in the section, 'install' in the file *README*. It says the first we need to install the `geos` library, and says that we can install by going to `geos` sub directory in the `basemap` directory, and issuing the following commands:

```
>>> sudo make
>>> sudo make install
```

Now, the `geos` library is installed, you can go back to the basemap directory, and installed it by issuing the following command:

```
>>> sudo python setup.py install
```

Index