PYTHON FOR M.Sc STUDENTS

V.N.Purushothaman
Department of Physics
Sree Kerala Varma College, Trichur

31-03-2013

Contents

1 Unit-1: Basics of Python language 1

1.1 Inputs 1

1.2 Outputs 1

1.3 Variables and data types 2

1.4 operators 3

1.5 Strings 4

1.6 Mutable and Immutable Types 5

1.7 Conditional Execution 14

1.8 Iteration and looping 14

1.9 Functions and Modules 15

1.10 File input and Output 16

1.11 Pickling 16

1.12 Problems 17

2 Advanced Python Programming 29

2.1 NumPy 29

2.1.1 Introduction 29

2.1.2 Array creation 29

2.1.3 Array modification 33

2.1.4 Printing arrays 34

2.1.5 Saving and restoring arrays 35

2.1.6 Indexing, Slicing and Iterating 39

2.1.7 Arrays as matrices 39

2.1.8 Arrays as polynomial coefficients 44

2.1.9 Linear Algebra 45

3 Plotting and visualization 49

3.1 Matplotlib 49

3.1.1 The Matplotlib Module 49

3.1.2 Plotting mathematical functions 55

4 Numerical Analysis 61

4.1 Numerical methods 61

4.1.1 Inverse of a function 61

4.1.2 Interpolation with Cubic Spline 66

4.1.3 Zeros of polynomials 70

4.1.4 Monte Carlo Methods 77

4.1.5 Sampled Data 81

4.1.6 Discrete Fourier Transform 82

4.1.7 Fast Fourier Transform(FFT) 84

4.1.8 Shooting method 85

4.1.9 Relaxation method: 87

5 Simulations 89

5.1 A computational approach to physics 89

5.1.1 Simple harmonic oscillator 90

5.1.2 Central field motion 92

5.1.3 Monte-Carlo simulations- value of π 95

5.1.4 Logistic map 97

5.1.5 Driven LCR circuit 100

5.1.6 Standing waves 104

5.1.7 Simulation of radioactivity 105

Preface

This is a rough collection of the lecture notes I had prepared to teach the course PHY2C08: COMPUTATIONAL PHYSICS prescribed for M.Sc physics students of colleges affiliated to Calicut University. Due to lack of time, it is done at a terrific pace which might have caused a few mistakes here and there. Shortage of explanations and examples is another casualty of such hurry. I hope that it may be useful in some small way to students and teachers. Please be kind enough to inform me when you come across mistakes in the ideas or language used in this monograph. *Purushothaman. V.N,*
Department of Physics,
Sree Kerala Varma College, Trichur
email:vadakkedam@rediffmail.com
Mob: 9446723810

Chapter 1

Unit-1: Basics of Python language

1.1 Inputs

A program is a set of statements used to produce an output from the input data. Numbers (real and complex), are read from terminal using the function *input('Prompt')* . Strings are read using the function *raw_input('prompt').* For example

```
>>> b=input('Give a number: ')
Give a number: 10
 >>>> b
10
>>> a=raw_input('Give a text: ')
Give a text: hopeless
>>> a
'hopeless'
```

1.2 Outputs

The output of a program can be a number, text or graphics. For text and numbers *print* statement is employed. For graphic output functions like *show(),savefig(),imshow()* etc are defined in relevant modules.

```
>>> x=5
>>> print x
```

```
5
>>> y=[2,5,7,9]
>>> print y
[2, 5, 7, 9]
>>> z='beamer'
>>> print z
beamer
>>> print x,y,z
5 [2, 5, 7, 9] beamer
```

Formatted output is possible just as in c-language. The general form of format string is $\%m.nx$ where m is an integer showing the total width to be used for printing, n is an integer representing the number of decimal places to be used while printing floating point numbers so that $|m| \geq 1+n$(decimal point+decimal places) and x is c for single character, f for float, e for float in scientific format, s for string, x for hexadecimal, o for octal, d or i for integer and $0d$ for integer with zeros on the left to fill the width.

```
>>> print '1)%5d  2)%5i 3)%05d '%(23,23,23)
1)   23  2)    23 3)00023
>>> print '1)%4c'%('z')
1)   z
>>> print '1)%12s, 2)%-12s'%('zoology','zoology')
1)     zoology, 2)zoology
>>> print '1)%12.5f, 2)%-12.5f, 3)%12.7f'%(24.5,24.5,24.5)
1)    24.50000, 2)24.50000    , 3)  24.5000000
>>> print '1)%12.5x, 2)%12.5o'%(24.5,24.5)
1)       00018, 2)       00030
```

1.3 Variables and data types

A computer program to solve a problem is designed using variables belonging to the supported data types. Python supports numeric data types like integers, floating point numbers and complex numbers. To handle character strings, it uses the String data type. Python also supports other compound data types likelists, tuples, dictionaries. In the previous example, x is numeric, y is a list and z is a string.

1.4 operators

Operators are functionality that do something and can be represented by symbols such as + or by special keywords. Operators require some data to operate on and such data are called operands. In $x = 2 + 3$, 2 and 3 are the *operands* and '=' and '+' are *operators.* The other operators are

```
or,and,not(Boolean OR,AND,NOT) : returns True or Fals
in(Membership): returns True or Fals,
not in(Non-membership): returns True or False,
<, <=, >, >=, !=, == (Comparisons): returns True or False
|,^,&(Bitwise OR, XOR,AND),
<<, >>(Bitwise Shifting left and right)
+,-,* (Add, Subtract, Multiply)
/,%,**(divide, reminder, Exponentiation)
+x (Positive), -x(Negative),
~(Bitwise NOT),
x[index](Subscription)
```

Bitwise operator works on bits and perform bit by bit operation. Assume a = 60 and b = 13. Now in binary format they will be as follows:

```
a = 0011 1100,
b = 0000 1101
Binary AND Operator copies a bit to the result if it exists in both operands.
a&b = 0000 1100
Binary OR Operator copies a bit if it exists in either operand.
a|b = 0011 1101
Binary XOR Operator copies the bit if it is set in one operand but not both.
a^b = 0011 0001
Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.
~a  = 1100 0011
Binary Left Shift Operator moves left operand by the number of bits specified by the
a << 2 will give 1111 0000 which is 240
Binary Right Shift Operator moves left operand by the number of bits specified by th
a >> 2 will give 0000 1111which is 15
```

Precedence Rules of operators: These rules give the sequence of executions of an expression containing more than one operator.

1. ( ) (anything in brackets is done first. Highest precedence)
2. ** (exponentiation is done next)
3. -x, +x (unary ±)
4. *, /, %, // (multiplication, division, remainder after division, successive division)
5. +, - (addition, subtraction)
6. relational operators: <, >, <=, >=, !=, ==
7. logical *not*
8. logical *and*
9. logical *or* (Lowest precedence)

1.5 Strings

String is a collection of same kind of elements (characters). It is a compound or collection data type. The individual elements of a string can be accessed by indexing.

```
>>> s = 'hello world'
>>> s[0]
'h'
>>> s[7]
'o'
>>> s[5]
' '
>>> s[-1]  # will print the last character
'd'
```

Strings can be added and multiplied by integers.

```
>>> p,q,r='Eating ','troubles',' meeting '
>>> p+q+r
'Eating troubles meeting'
>>> 2*p+q
'Eating Eating troubles'
```

1.6 Mutable and Immutable Types

There is one major difference between String, tuple, list,dictionary types. List and dictionary are mutable but *string and tuple* are immutable. We can change the value of an element in a list , add new elements to them and remove any existing element. This is not possible with String and tuple types.In the case of sets one variety called *frozenset* is immutable while *set* is mutable.

List

List is much more flexible than String. The individual elements can be of any type, even another list. Lists are defined by enclosing the elements inside a pair of square brackets and separated by commas.

```
>>> l=[2.3,'A',3,'khan']
>>> type(l)
<type 'list'>
>>> 2*l
[2.29, 'A', 3, 'khan', 2.29, 'A',3, 'khan']
>>> l[3]=28
>>> l
[2.299, 'A', 3, 28]
```

Lists respond to the + (concatenation) and * (repetition) operators like strings.The result is a new list.

Python Expression	Results	Description
len([1, 2, 3])	3	Length
[1, 2, 3] + [4, 5, 6]	[1, 2, 3, 4, 5, 6]	Concatenation
['Hi!'] * 4	['Hi!', 'Hi!', 'Hi!', 'Hi!']	Repetition
3 in [1, 2, 3]	True	Membership
for x in [1, 2, 3]: print x,	1 2 3	Iteration

List Methods

A method is a function that is coupled to some object, be it a list, a number, a string, or whatever. In general, a method is called like this: *object.method(arguments)*. If a is list object and max() is a method defined on list class to find the maximum value in the list, then *a.max()* returns maximum of list *a*. A method call is like a function call, except that the object is put before the name of the method method with a dot separating them. Lists have several methods that can be used to examine or modify their contents.

1. **append:** The append method is used to append an object to the end of a list:

   ```
   >>>lst = [1, 2, 3]
   >>>lst.append(4)
   >>>lst
   >>>[1,2, 3, 4]

   The same can be achieved using + operator.
   >>> x=[3,5,4]
   >>> x+=[2]
   >>> x
   [3, 5, 4, 2]
   ```

2. **count:** The count method counts the occurrences of an element in a list:

   ```
   >>>['to', 'be', 'or', 'not', 'to', 'be'].count('to')
   >>>2
   >>>x = [[1, 2], 1, 1, [2, 1, [1, 2]]]
   >>>x.count(1)
   >>>x.count([1, 2])
   >>>2.
   ```

3. **extend:** The extend method allows you to append several values at once by supplying a sequence of the values you want to append. In other words, your original list has been extended by the other one:

   ```
   >>>a = [1, 2, 3]
   >>>b = [4, 5, 6]
   >>>a.extend(b)
   >>>a
   [1,2, 3, 4, 5, 6]
   ```

   This may seem similar to concatenation, but the important difference is that the extend method modifies a list without creating a new one. In ordinary concatenation, a completely new list is returned:

   ```
   a = [1, 2, 3]
   b = [4, 5, 6]
   a + b
   [1, 2, 3, 4, 5, 6]
   a
   [1,2, 3]
   ```

4. **index:** The index method is used for searching lists to find the index of the first occurrence of a value:

   ```
   >>> knights = ['We', 'are', 'the', 'knights', 'who', 'say', 'ni']
   >>> knights.index('who')
   4
   >>> knights.index('herring')
   >>> ValueError: list.index(x): x not in list
   ```

   When you search for the word 'who', you find that it's located at index 4:

   ```
   >>> knights[4]
   'who'
   ```

5. **insert:** The insert method is used to insert an object into a list: »> numbers = [1, 2, 3, 5, 6, 7] »> numbers.insert(3, 'four') »> numbers »> [1, 2, 3, 'four', 5, 6, 7] As with extend, you can implement insert with slice assignments:

   ```
   numbers = [1, 2, 3, 5, 6, 7]
   numbers[3:3] = ['four']
   numbers
   [1, 2, 3, 'four', 5, 6, 7]
   ```

6. **pop:** The pop method removes an element (by default, the last one) from the list and returns it:

   ```
   >>> x = [1, 2, 3]
   >>> x.pop()
   >>> x
   >>> [1, 2]
   >>> x.pop(0)
   >>> x
   >>> [2]
   ```

   The pop method is the only list method that both modifies the list and returns a value.

   ```
   >>> x = [1, 2, 3]
   >>> x.append(x.pop())
   >>> x
   >>> [1, 2, 3]
   ```

7. **remove:** The remove method is used to remove the first occurrence of a value:

```
>>> x = ['to', 'be', 'or', 'not', 'to', 'be']
>>> x.remove('be')
>>> x
['to', 'or', 'not', 'to', 'be']
>>> x.remove('bee')
>>> ValueError: list.remove(x): x not in list
```

8. **del:** To remove a list element, you can use either the *del* statement if you know exactly which element(s) you are deleting or the *remove()* method if not known. The reverse method reverses the elements in the list.

   »> x = [1, 2, 3] »> x.reverse() »> x »> [3, 2, 1] Note that reverse changes the list and and saves under the same name. »> x = [1, 2, 3] »> list(reversed(x)) [3, 2, 1]

9. **sort:** The sort method is used to sort lists in place. Sorting 'in place' means changing the original list so its elements are in sorted order, rather than simply returning a sorted copy of the list:

```
>>> x = [4, 6, 2, 1, 7, 9]
>>> x.sort()
>>> x
>>> [1, 2, 4, 6, 7, 9]
>>> x = [4, 6, 2, 1, 7, 9]
>>> x.sort(reverse=True)
>>> x
>>> [9, 7, 6, 4, 2, 1]
```

10. **sorted:** Another way of getting a sorted copy of a list is using the sorted *function*:

```
>>> x = [4, 6, 2, 1, 7, 9]
>>> y=sorted(x)
>>> x,y
([4, 6, 2, 1, 7, 9], [1, 2, 4, 6, 7, 9])
```

    This function can actually be used on any sequence, but will always return a list:

```
>>> sorted('Python')
>>> ['P', 'h', 'n', 'o', 't', 'y']
```

11. **len:** Gives the number of elements in a list.

12. **max:** Gives the element having maximum ASCII value in a list.

13. **min:** Gives the element having minimum ASCII value in a list.
14. **cmp:** This function is the basis for sorting. *cmp(a, b)* returns -1 if a $<$ b, 0 if a $==$ b and 1 if a $>$ b.
15. **list:** Converts a string or tuple into a list.
16. **sum:** Returns the sum of elements in a numeric list.

For example

```
>>> a=[2,5,'A','a','ab']
>>> max(a)
'ab'
>>> a.append('z')
>>> a
[2,5,'A','a','ab','z']
>>> max(a)
'z'
>>> min(a)
2
>>> x=[4, 7, 8, 2, 3, 12]
>>> cmp(a,x)
-1
>>> cmp(x,a)
1
>>> y=x
>>> y
[4, 7, 8, 2, 3, 12]
>>> cmp(x,y)
0
>>> c='design'
>>> list(c)
['d', 'e', 's', 'i', 'g', 'n']
>>> d=('j','k',3,5)
>>> list(d)
['j', 'k', 3, 5]
>>> sum(x)
36
```

Slicing: Elements from a list can be selected using slicing operator ':'. If x is a list, then $x[m : n : p]$ represents the set of elements of x with indices $[m^{th}, (m+p)^{th}, (m+2p)^{th}, ..]$ excluding n^{th} element.

```
>>> x=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> x[0:-1:2]
[0, 2, 4, 6, 8]
>>> x[1:-1:2]
[1, 3, 5, 7]
>>> sum(x[1:-1:2])
16
>>> x[0:5:2]
[0, 2, 4]
>>> sum(x[0:5:2])
6
```

set:

A *set* object is an unordered collection of immutable values. They cannot be indexed by any subscript. The built-in function *len()* returns the number of items in a set. There are currently two intrinsic set types:

(1)Sets: These represent a mutable set. They are created by the built-in *set()* constructor and can be modified afterwards by several methods, such as *add(), clear(), discard()*.

Frozen sets: These represent an immutable set. They are created by the built-in *frozenset()* constructor. As a frozenset is immutable, it can be used as an element of another set, or as a dictionary key. Common uses of sets include membership testing, removing duplicates from a sequence, and computing mathematical operations such as intersection, union, difference, and symmetric difference. The main functions are *len(), union(), intersection(), difference(), symmetric_ difference(), issubset() and issuperset()*. There is also an operator equivalent for many of these functions. Let s and t be two sets. Then

1. x *in* s: test element x for membership in s (True/False)
2. x *not in* s: test element x for non-membership in s (True/False)
3. s.issubset(t): [s <= t]: test whether every element in s is in t
4. s.issuperset(t): [s >= t]: test whether every element in t is in s
5. s.union(t): [s | t]: new set with elements from both s and t
6. s.intersection(t): [s & t]: new set with elements common to s and t
7. s.difference(t): [s - t]: new set with elements in s but not in t

8. s.symmetric_difference(t) : [s t̂]: new set with elements in either s or t but not both

9. s.copy(): new set with a copy of s

10. s.clear(): Remove all elements from the set s.

11. s.discard(x): Remove element x from set s if it is a member. If x is not a member, nothing happens.

12. s.update():Update s with the union of itself and others.

```
>>> a=[1,2.0,6.1,'l']
>>> b=set(a)
>>> b
set([1, 2.0, 'l', 6.0999999999999996])
>>> c=set([1,3,6.1,'k'])
>>> c
set([1, 'k', 3, 6.0999999999999996])
>>> d=set('domain')
>>> d
 set(['a', 'd', 'i', 'm', 'o', 'n'])
>>> f=set((1,3,5,9.1))
>>> f
set([1, 3, 9.0999999999999996, 5])
>>> b|c
set([1, 2.0, 3, 'k', 'l', 6.0999999999999996])
>>> b&c
set([1, 6.0999999999999996])
>>> b.difference(c)
set([2.0, 'l'])
>>> c.difference(b)
set(['k', 3])
>>> c.symmetric_difference(b)
set(['k', 2.0, 3, 'l'])
>>> s=set([1,3,5,7,9])
>>> s.add(11)
>>> s
set([1, 3, 5, 7, 9, 11])
>>> s.remove(5)
>>> s
set([1, 3, 7, 9, 11])
```

Tuples

Tuples are data structures that are very similar to lists, but they cannot be modified (immutable). They can only be created. Tuples have important roles as keys for dictionaries. A tuple is a sequence that is enclosed by parentheses ( ). The following line of code creates a three-element tuple

```
>>> x = ('a', 'b', 'c')
```

Interconversion between lists and tuples is possible using *list()* and *tuple()* functions.

```
>>> list((1, 2, 3, 4))
[1,2, 3, 4]
>>> tuple([1, 2, 3, 4])
(1,2, 3, 4)
```

Dictionaries

Dictionaries are associative arrays. It is a group of $\{key : value\}$ pairs. The elements in a dictionary are indexed by keys. Keys in a dictionary are required to be unique. Keys can be almost any Python type, but are usually numbers or strings. Values, on the other hand, can be any arbitrary Python object. Dictionaries are enclosed by curly braces - { } and values can be assigned and accessed using square braces []. They are different from sequence type containers like *lists* and *tuples* in the method of storing data. There is no concept of order among elements. They are unordered.Their main use include storing time of modification of files as values and file name as keys, telephone directory with name as value and phone number as key, address book with name as key and address as value, the coordinate of a point(tuple) as key and its colour as value in a graphic screen etc. Example for a dictionary is given below.

```
>>> dct={} #Creates an empty dictionary
>>> dct['host']='Earth'
#'host' is the key and 'earth' is the value.
>>> dct
{'host': 'Earth'}
>>> dct['port']=80
>>> dct
{'host': 'Earth', 'port': 80}
>>> dct.keys()
```

```
['host', 'port']
>>> dct.values()
['Earth', 80]
>>> print dct['host']
```

Dictionary functions and methods:

1. cmp(dict1, dict2): Compares elements of both dictionaries.
2. len(dict): Gives the total length of the dictionary. This would be equal to the number of Key-value pairs in the dictionary.
3. str(dict): Produces a printable string representation of a dictionary
4. type(variable): Returns the type of the passed variable. If passed variable is dictionary then it would return a dictionary type.

Python includes following dictionary methods

1. dict.clear(): Removes all elements of dictionary dict
2. dict.copy(): Returns a shallow copy of dictionary dict
3. dict.fromkeys(): Create a new dictionary with keys from seq and values set to value.
4. dict.get(key, default=None): For key key, returns value or default if key not in dictionary
5. dict.has_key(key): Returns true if key in dictionary dict, false otherwise
6. dict.items(): Returns a list of dict's (key, value) tuple pairs
7. dict.keys(): Returns list of dictionary dict's keys
8. dict.setdefault(key, default=None): Similar to get(), but will set dict[key]=default if key is not already in dict
9. dict.update(dict2): Adds dictionary dict2's key-values pairs to dict
10. dict.values(): Returns list of dictionary dict's values

1.7 Conditional Execution

The most fundamental aspect of a programming language is the ability to control the sequence of operations. One of this control is the ability to select one action from a set of specified alternatives. The other one is the facility to repeat a series of actions any number of times or till some condition becomes false. To execute some section of the code only if certain conditions are true python uses *if, elif,...,else* construct.

```
>>> x = input('Enter a number ')
10
>>> if x>10:
          print 'x>10' # Note the Colon and indentation.
elif x<10:
          print 'x<10'
else:
     print 'x=10'
>>>x=10
```

1.8 Iteration and looping

when a condition remains true, if a set of statements are to be repeated, the *while* and *for* constructs are employed. The general syntax of a while loop may be given as follows.
*while*condition:
set of statements to be repeated
for elements *in* list or tuple :
set of statements to be repeated

```
>>> x=10
>>> while x>0:
              print x,
              x=x-1
>>>10 9 8 7 6 5 4 3 2 1
>>> for i in range(10,0,-1):
                       print i,
>>>10 9 8 7 6 5 4 3 2 1
```

1.9 Functions and Modules

A function is a block of code that performs a specific task. Using a function in a program is called 'calling' the function. Python has two tools for building functions: *def* and *lambda*. For example, we can build a function that returns the square root of a number as follows:
(1) *def* squareroot(x): return *math.sqrt(x)*
(2) squareroot = *lambda* x: *math.sqrt(x)*
(3) g = *lambda* x: x*2
g(3)=6
(4) (*lambda* x: x*2)(3)= 6
If a function is used only once (called from only one place in your program) Lambda functions are useful and convenient for two reasons: (1)There is no need to give the function a name.(2) It can be defined where it is used.

The next method of defining functions is illustrated below. For finding the largest of x,y,z

```
>>> def large(x,y,z):
                    if y>x:
                          x,y=y,x
                    if z>x:
                          z,x=x,z
                    return(x)
>>> large(3,4,2)
4
```

In Python, the definitions of functions, variables, constants may be saved in a file and use them in a script or in an interpreter just like header files in C-language. Such a file is called a *module*. The file name is the module name with the suffix *.py* appended. Within a module, the name of the module is assigned to the global variable _*name*_. Definitions from a module can be imported into other modules or into the main module. Examples of some standard modules are math, os,random, pylab, numpy etc. The main advantage of creating and using modules is that longer programs can be split into several files so that maintenance of code is easy and can be reused in several programs by including the file with the keyword *import* at the beginning of the program.

1.10 File input and Output

Files are used to store data and program for later use. This program creates a new file named 't.txt' (any existing file with the same name will be deleted) and writes a String to it. The file is closed and then reopened for reading data. The relevant functions are open, write, read and close.

```
>>> f=open('t.txt','w')
>>> f.write('breaking into the file')
>>> f.close()
>>> f=open('t.txt','r')
>>> f.read()
'breaking into the file'
```

1.11 Pickling

Strings can easily be written to and read from a file. Numbers take more effort, since the read() method only returns strings, which will have to be converted into a number explicitly. However, it is very complicated when trying to save and restore data types like lists, dictionaries etc. Rather than constantly writing and debugging code to save complicated data types, Python provides a standard module called pickle. Functions dump and load are used in pickle. *pickle.dump(a,f)* will save object *a* to file *f*. *a=pickle.load(f)* retrieves data from file *f*.

Pickling- Examples

```
>>>import pickle
>>>a=10.1
>>>b='sh'
>>>c=[5,3,2]
>>>f = open("state", 'w')
>>>pickle.dump(a, f)
pickle.dump(b, f)
pickle.dump(c, f)
file.close()

>>>file = open("state", 'r')
Reading and writing files
```

```
a = pickle.load(file)
b = pickle.load(file)
c = pickle.load(file)
file.close()
```

Any data that was previously in the variables a, b, or c is restored to them by pickle.load.

1.12 Problems

1. To sort a set of numbers

   ```
   a=[]
   n=input('Give the count of numbers n')
   for i in range(n):a.append(input('Type the numbers'))
   a.sort() #Ascending order
   print a
   a.reverse()  # Descending order
   print a
   ```

2. Simultaneous arrays: Construct two 100-element arrays such that i^{th} element of one array is $\sin(2\pi i/100)$ and the other $\cos(2\pi i/100)$.

   Program:

   ```
   from math import *
   x=[sin(2*pi*i/100) for i in range(1,101)]
   y=[cos(2*pi*i/100) for i in range(1,101)]
   print x,y
   ```

3. To create a triangle of equispaced stars(*)

   Program:

   ```
   n=input('Howmany rows ? ')
   for i in range(n):
       print
       for j in range(i+1): print '*',
   ```

   When the program is run

```
Howmany rows ? 7

*
* *
* * *
* * * *
* * * * *
* * * * * *
* * * * * * *
```

4. Fibonacci series:

```
n=input('Number below which series is required: ')
a, b = 0, 1
while b < n:
       print b,
       a, b = b, a+b
```

5. To read a $m \times n$ matrix using list-methods only

```
m,n=input('order of matrix m,n = ')
a=[]
for i in range(m):
    b=[]
    for j in range(n):
        b.append(input('give elements a(i,j)'))
    a.append(b)
print 'output: ',a
```

When the program is run

```
order of matrix m,n = 2,3
give elements a(i,j)1
give elements a(i,j)2
give elements a(i,j)3
give elements a(i,j)4
give elements a(i,j)5
give elements a(i,j)6
output: [[1, 2, 3], [4, 5, 6]]
```

6. To generate values of a function, say, $x \sin x$

```
import math
x=[0.1*i for i in range(10)]
```

```
y=[i*math.sin(i) for i in x]
for i in range(10):
        print '(%0.2f,%0.5f)'%(x[i],y[i]),
```

The (x, y) values obtained are given below

```
(0.00,0.00000), (0.10,0.00998), (0.20,0.03973), (0.30,0.08866),
(0.40,0.15577), (0.50,0.23971), (0.60,0.33879), (0.70,0.45095),
(0.80,0.57388), (0.90,0.70499),
```

7. To find factorial of a number

```
n=input('Give the number whose factorial is required')
f=1
for i in range(1,n+1):f*=i
print f
```

8. Permutations

```
n,r=input('Give n and r in nPr: ')
p=1
for i in range(n,n-r,-1):p*=i
print p
```

9. combinations The definition is

$$^{n}C_{r} = \frac{n!}{(n-r)!r!} = \frac{n(n-1)(n-2)....(n-r+1)}{r(r-1)(r-2)....3.2.1} = \frac{n}{r}\ {}^{(n-1)}C_{(r-1)}$$

```
def combination(n,r):
    if r==0:return 1
    else:return n*combination(n-1,r-1)/r
n,r=input('Give n and r in nCr: ')
print combination(n,r)
```

When program is run

```
Give n and r in nCr: 6,3
20
>>>
Give n and r in nCr: 7,4
35
```

Aliter:

```
def f(a):
    if a==1:return(1)
    else: return(a*f(a-1))
n,r=input('Give n and r in nCr: ')
print f(n)/(f(n-r)*f(r))
```

10. Pascal's triangle

Pinciple: It is the set of binomial coefficients arranged in rows. The, n^{th} row corresponds to the coefficients of the expansion $(a+b)^n$

```
def f(a):
    if a==0:return(1)
    else: return(a*f(a-1))
n=input('Howmany rows ? ')
for i in range(n):
    print
    for j in range(i+1): print f(i)/(f(i-j)*f(j)),
```

Aliter:

```
def combination(n,r):
    if r==0:return 1
    else:return n*combination(n-1,r-1)/r
m=input('Howmany rows ? ')
for i in range(m+1):
    print
    for j in range(i+1):print combination(i,j),
```

when the program is run

```
Howmany rows ? 7

1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
```

11. Generate the triangular sequence 0,1,3,6,10,15,21,...n

Principle: Obviously the numbers are combinations

$$\binom{n}{2} = \frac{n(n-1)}{1 \times 2}$$

where $n = 0, 1, 2, 3...$

Program:

```
m=input('Give maximum number upto which the series is required: ')
print [i*(i+1)/2 for i in range(m) if i*(i+1)/2 < m]
```

When the program is run

```
Give maximum number upto which the series is required: 150
[0, 1, 3, 6, 10, 15, 21, 28, 36, 45, 55, 66, 78, 91, 105, 120, 136]
```

Aliter:
The relation between adjacent elements is

$$a[0] = 0, n > 0, a[n] = a[n-1] + (n-1)$$

```
n,a=input('n: '),0
for i in range(n):
    a+=i
    print a
```

12. Hailstorm numbers

Principle: Pick any whole number. If it's odd, multiply the number by 3, then add 1. If it's even, divide it by 2. Now, apply the same rules to the answer that you just obtained. Do this over and over again, applying the rules to each new answer. Hence these are the set of numbers obtained by the following rule of iteration.
If s_i is even, $s_{i+1} = s_i/2$, else $s_{i+1} = 3s_i + 1$

Program:

```
s=input('Seed number for Hailstorm series: ')
while s!=1:
        if s%2==0:s/=2
        else:s=(3*s+1)
        print s,
```

When the program is run

```
Seed number for Hailstorm series: 17
52 26 13 40 20 10 5 16 8 4 2 1
```

13. To find largest and smallest in a set of numbers

```
x=[]
n=input('Give the count of numbers')
for i in range(n): x.append(input('Type numbers one at a time '))
print 'Largest of the series is',max(x)
print 'Smallest of the series is',min(x)
```

14. To solve quadratic equation

```
from cmath import *
a,b,c=input("Give coefficients in the order a,b,c seperated by comma.  "
d=sqrt(b*b-4*a*c)
print "Root 1 = ",(-b+d)/(2*a),"Root 2 = ",(-b-d)/(2*a)
```

When program is run

```
Give coefficients in the order a,b,c seperated by comma.  1,2,3
Root 1 =  (-1+1.41421356237j) Root 2 =  (-1-1.41421356237j)
```

15. To verify orthogonality of sine and cosine functions
Principle: If the functions are orthogonal, then

$$\int_0^\pi \sin\theta\cos\theta d\theta \approx \sum_{i=0}^{\pi} \sin\theta_i\cos\theta_i \approx 0$$

```
from math import*
print sum([sin(pi*i/180)*cos(pi*i/180) for i in range(180)]),'is negligi
```

Output: -4.85722573274e-15 is negligible

16. To check whether a given number is prime

Principle: n is a prime number if it is exactly divisible only by 1 and n. To check this, see whether it is divisible by any number between 2 and $n/2$ (integer division).

```
n=input("Give the number n: ")
m=1+n/2
for i in range(2,m):
    if n%i!=0: continue
    else:
        print '%d is not a prime. It is divisible by %d'%(n,i)
        break
if i==n/2: print '%d is a prime'%n
```

When the program is run

```
Give the number n: 83
83 is a prime
>>>
Give the number n: 245791
245791 is not a prime. It is divisible by 7
```

17. To find the count of prime numbers in a given range.

Principle: Even numbers cannot be prime. Check in the set of odd numbers for prime numbers.

```
j,k=input("Give the range between which prime numbers are required: ")
x=range(j,k,1)
p=[]
for n in x:
    m=n/2
    if n>0 and n<4:p.append(n)
    for i in range(2,m+1):
        if n%i==0: break
        if i==m: p.append(n)
print 'There are %d prime numbers in the range(%d, %d). They are'%(len(p),j,k)
print p
```

When the program is run

```
Give the range between which prime numbers are required: 49,150
There are 20 prime numbers in the range(49, 150). They are
[53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101,
 103, 107, 109, 113, 127, 131, 137, 139, 149]
```

18. To check whether a given word is a palindrome.

Principle: A sequence is a palindrome if it is the same when read from left or right. For example, xyzzyx,123321 are palindromes.

Program:

```
x=raw_input('Give the word')
y=[x[-i-1] for i in range(len(x))]
if list(x)==y:print 'palindrome'
else: print 'Not a palindrome'
```

When the program is run

```
Give the word: abc121cba
palindrome
>>>
Give the word: election
Not a palindrome
```

19. To find square root of a number

Principle: Let x be the square root of n. Then $x^2 - n = 0$. Now x is the root of this equation. It can be calculated using Newton-Raphson method.

$$f(x) = x^2 - n,\ f'(x) = 2x$$

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} = \frac{1}{2}\left(x_k + \frac{n}{x_k}\right)$$

Program:

```
n,e=input('Give the number n and accuracy required e: ')
x0,x=0,1
while abs(x-x0)>e:
    x0=x
    x=(x0+n/x0)/2.0
print 'Square root of %0.3f =%0.5f '%(n,x0)
```

When the program is run

```
Give the number n and accuracy required e: 24,.001
Square root of 24.000 =4.89900
```

20. To find m^{th} root of a number

Principle: Let x be the m^{th} root of n. Then $x^m - n = 0$. Now x is the m^{th} root of this equation. It can be calculated using Newton-Raphson method.

$$f(x) = x^m - n,\ f'(x) = mx^{m-1}$$

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} = \frac{1}{m}\left[(m-1)x_k + \frac{n}{x_k^{m-1}}\right]$$

Program:

```
n,m,e=input('Give the number n, order of root m and accuracy e required : ')
x0,x=0.0,1.0
while abs(x-x0)>e:
    x0=x
    x=((m-1)*x0+n/x0**(m-1))/m
print '%0.2f (th) root of %0.3f = %0.5f '%(m,n,x0)
```

When the program is run

```
Give the number n, order of root m and accuracy e required : 65,3,.01
3.00 th root of 65.000 =4.03005
>>>
Give the number n, order of root m and accuracy e required : 65,4,.01
4.00 th root of 65.000 =2.84527
>>>
Give the number n, order of root m and accuracy e required : 2.88,1.5,.01
1.50 th root of 2.880 =2.03016
```

Aliter: Using bisection method.

```
def f(x,m,n):return x**m-n
a,b,m,n,k=input('Intervel (a,b),order of root m, number n and no. of iterations
i=0
while i<k:
        c=(a+b)/2.0
        if f(a,m,n)*f(c,m,n)<0:b=c
        else:a=c
        i+=1
print 'The %d root of %f after %d iterations is %20.15f'%(m,n,i,c)
```

21. To convert temperature in Fahrenheit into centigrade

Principle: $32^oF = 0^0C$, $212^oF = 100^oC$. Therefore the conversion formulae are

$$C = \frac{5}{9}(F - 32)$$

$$F = \frac{9}{5}C + 32$$

Program:

```
t=input('Give the temperature: ')
c=input('Is the given temperature is in 1.centigrade or 2.farenheit (1/2
if c==1: print '%0.2f C= %0.3f F'%(t,t*9.0/5+32.0)
else: print '%0.2f F= %0.3f C'%(t,(t-32.0)*5.0/9)
```

When the program is run

```
Give the temperature: 25
Is the given temperature is in 1.centigrade or 2.farenheit (1/2): 1
25.00 C= 77.000 F
>>>
Give the temperature: 180
Is the given temperature is in 1.centigrade or 2.farenheit (1/2): 2
180.00 F= 82.222 C
```

22. To find value of π

Principle: The value of π can be calculated using $\tan 45^o = \tan(\pi/4) = 1$ as follows.

$$\frac{\pi}{4} = \tan^{-1} 1 = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + ... = \sum_{i=0}^{\infty} \frac{(-1)^i}{2i+1}$$

Program:

```
x=[(-1)**i/(2.0*i+1) for i in range(100000)]
pi=4*sum(x)
print 'The value of pi=%0.10f'%pi
```

When the program is run, the following output is obtained.

```
The value of pi=3.1415826536
```

Python keywords

Core Python has 30 keywords:
(1)and (2)as (3)break (4)class (5)continue (6)def (7)del (8)elif (9)else (10)except (11)finally (12)for (13)from (14)global (15)if (16)import (17)in (18)is (19)lambda (20)nonlocal (21)not (22)or (23)pass (24)raise (25)return (26)assert (27)try (28)while (29)with (30)yield

Chapter 2

Advanced Python Programming

2.1 NumPy

2.1.1 Introduction

NumPy's main class is the homogeneous multidimensional array called *ndarray.* This is a table of elements (usually numbers), all of the same data type. Each element is indexed by a tuple of positive integers. Examples of multidimensional array objects include vectors, matrices, spreadsheets etc. The term *multidimensional* refers to arrays having several dimensions or axes. The number of axes is often called rank (not a tensor rank).

For example, the coordinates of a point in 3-D space (x, y, z) is an array of rank 1. This also gives the position vector of that point. The array $([1., 0., 0.], [0., 1., 2.])$ is one of rank 2. It is equivalent to $\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 2 \end{pmatrix}$ (it is 2-dimensional). The first dimension (rows) has a length of 2, the second dimension(column) has a length of 3. The array $([[1., 0.], 0.], [[0., 1.], 2.])$ is one of rank 3. It is equivalent to $\begin{pmatrix} (1 & 0) & 0 \\ (0 & 1) & 2 \end{pmatrix}$ (it is 3-dimensional).

2.1.2 Array creation

There are many ways to create arrays. For example, you can create an array from a regular Python list or tuple using the array function.

```
>>> a = array( [2,3,4] )
>>> a
array([2, 3, 4])
>>> type(a)      # a is an object of the ndarray class
<type 'numpy.ndarray'>
```

The function *array()* transforms sequences of sequences into two-dimensional arrays, and it transforms sequences of sequences of sequences into three-dimensional arrays, and so on. The type of the resulting array is deduced from the type of the elements in the sequences.

```
>>> b = array( [ (1.5,2,3), (4,5,6) ] )
>>> b
array([[ 1.5,  2. ,  3. ],
       [ 4. ,  5. ,  6. ]])
```

To create an array whose elements are sequences of numbers, *NumPy* provides a function $arange(x_1, x_2, dx)$ and returns $x_1, x_1 + dx, ..., x_2 - dx$. It is analogous to *range function* but accepts floating point numbers also.

```
>>> arange( 10, 30, 5 )
array([10, 15, 20, 25])
>>> arange( 0, 2, 0.3 )
array([ 0. ,  0.3,  0.6,  0.9,  1.2,  1.5,  1.8])
```

array and *arange* are not the only functions that create arrays. Usually the elements of the array are not known from the beginning, and a placeholder array(empty array) is needed. There are some functions to create arrays with some initial content. By default, the type of the created array is *float64*. The function *zeros((m,n))* creates a 2-D array of m rows and n columns with *zeros* as elements. Similarly the function *ones((m,n))* creates an array full of *ones*, the function *empty((m,n))* creates an array without filling it in and the function *random((m,n))* creates an array filling it with random numbers between 0 and 1. *identity (n)* creates an n-dimensional unit matrix. Then the initial content is random and it depends on the state of the memory. In these functions the arguments m, n specifies the size along each axis of the array.

Using *arange* with floating point arguments, it is generally not possible to predict the number of elements obtained because of the floating point precision. Hence it is better to use the function $linspace(x_1, x_2, nx)$ which returns equispaced nx numbers from x_1 to x_2.

The general syntax of these functions are *empty (shape=, dtype=int)* Return an uninitialized array of data type, *dtype,* and given *shape.*

An array of zeros can be created with a specified shape using *zeros()* function. *zeros(shape=, dtype=)*:Return an array of data type *dtype* and given shape filled with zeros. An array of ones can be created with a specified shape using *ones()* function. *zeros(shape=, dtype=)*: Return an array of data type *dtype* and given shape filled with zeros. an identity matrix can be created using *identity()* function
identity (n, dtype=int): Return a 2-d square array of *shape (n,n)* and data type, *dtype* with ones along the main diagonal.

```
>>> empty( (2,3) )
array([[  3.73603959e-262,   6.02658058e-154,   6.55490914e-260],
       [  5.30498948e-313,   3.14673309e-307,   1.00000000e+000]])

>>> empty( (2,3) )    # the content may change in different invocations
array([[  3.14678735e-307,   6.02658058e-154,   6.55490914e-260],
       [  5.30498948e-313,   3.73603967e-262,   8.70018275e-313]])

>>> zeros( (3,4) )
array([[0.,  0.,  0.,  0.],
       [0.,  0.,  0.,  0.],
       [0.,  0.,  0.,  0.]])

>>> ones( (2,3,4), dtype=int16 )
array([[[ 1, 1, 1, 1],
        [ 1, 1, 1, 1],
        [ 1, 1, 1, 1]],
       [[ 1, 1, 1, 1],
        [ 1, 1, 1, 1],
        [ 1, 1, 1, 1]]], dtype=int16)

>>> a=identity(4,dtype=float)
>>> a
array([[ 1.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.],
       [ 0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  1.]])

>>> linspace( 0, 2, 9 )
array([ 0.  ,  0.25,  0.5 ,  0.75,  1.  ,  1.25,  1.5 ,  1.75,  2.  ])
>>> x = linspace( 0, 2*pi, 10 )
>>> x
```

```
array([ 0.        ,  0.6981317 ,  1.3962634 ,  2.0943951 ,  2.7925268 ,
        3.4906585 ,  4.1887902 ,  4.88692191,  5.58505361,  6.28318531])
>>> f = sin(x)
array([  0.00000000e+00,   6.42787610e-01,   9.84807753e-01,
         8.66025404e-01,   3.42020143e-01,  -3.42020143e-01,
        -8.66025404e-01,  -9.84807753e-01,  -6.42787610e-01,
        -2.44921271e-16])
```

Array attributes

The important attributes of any *ndarray* object are:

1. *b.ndim* :It gives the rank of the array.
2. *b.shape*:It returns a tuple of integers indicating the size of the array in each dimension. For a matrix with m rows and n columns, *shape* returns (m, n).
3. *b.size*. Returns the total number of elements in all dimensions of the array. This is equal to the product of the elements of *shape* command.
4. *b.dtype* Returns the data type of the elements in the array. *NumPy* provides the following datatypes: $bool_, character, int_, int8, int16, int32, int64, float_, float8, float16, f$
5. *b.itemsize*: Returns the size in bytes of each element of the array.For example, an array of elements of type *float64* has *itemsize* 8 (=64/8), while one of type *complex32* has *itemsize* 4 (=32/8).
6. *b.data*: Returns the buffer containing the actual elements of the array.

Example for these methods : We define the following array:

```
>>> from numpy import *
>>> a = array([(0, 1, 2),(3, 2, 1)],)
>>>a.shape, a.ndim, a.size, a.itemsize, a.dtype
(  (2, 3),   2,      6,      4,     'dtype('int32') )
```

The type of the array can also be explicitly specified at creation time:

```
>>> c = array( [ [1,2], [3,4] ], dtype=complex )
>>> c
```

```
array([[ 1.+0.j,  2.+0.j],
       [ 3.+0.j,  4.+0.j]])
>>> c.dtype
dtype('complex128')
```

A frequent error consists in calling array with multiple numeric arguments, rather than providing a single list of numbers as an argument.

```
>>> a = array(1,2,3,4)   # WRONG because numbers within () are taken as arguements.
>>> a = array([1,2,3,4])  \# RIGHT because [1,2,3,4] is a single list.
```

2.1.3 Array modification

The shape of an array can be changed with various commands:
ravel(),transpose(),reshape(m,n,...),resize(m,n,...)
Here $(m, n, ...)$ is the size of the multidimensional array. For example

```
>>> from numpy import*
>>> a=array([[ 0,  1,  2,  3],
         [ 4,  5,  6,  7],
         [ 8,  9, 10, 11]])
>>> a.ravel()
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
>>> a
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]]) # No permanent change to shape
>>> a.reshape(4,3)
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
>>> a
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]]) # No permanent change to shape
>>> a.resize(4,3)
>>> a
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
>>> a
 array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])      # Permanent change to shape
>>> a.transpose()
array([[ 0,  3,  6,  9],
       [ 1,  4,  7, 10],
       [ 2,  5,  8, 11]]) # No permanent change to shape
```

The reshape function returns its argument with a modified shape, whereas the resize method modifies the array itself:

2.1.4 Printing arrays

When you print an array, *NumPy* displays it in a similar way to nested lists, but with the following layout:

1. the last dimension is printed from left to right,
2. the last but one, from top to bottom,
3. and the rest, also from top to bottom, separating each slice by an empty line.

One dimensional arrays are then printed as rows, two dimensional as matrices and three dimensional as lists of matrices.

```
>>> a = arange(6)
>>> print a
[0 1 2 3 4 5]
>>>
>>> b = arange(12).reshape(4,3)
>>> print b
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
>>>
```

```
>>> c = arange(24).reshape(2,3,4)
>>> print c
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]
 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
```

If an array is too large to be printed, *NumPy* automatically skips the central part of the array and only prints the corners:

```
>>> print arange(10000)
[   0    1    2 ..., 9997 9998 9999]
>>>
>>> print arange(10000).reshape(100,100)
[[   0    1    2 ...,   97   98   99]
 [ 100  101  102 ...,  197  198  199]
 [ 200  201  202 ...,  297  298  299]
 ...,
 [9700 9701 9702 ..., 9797 9798 9799]
 [9800 9801 9802 ..., 9897 9898 9899]
 [9900 9901 9902 ..., 9997 9998 9999]]
```

2.1.5 Saving and restoring arrays

The simplest way to store arrays is to write it to a text file as text using the *numpy* function *savetxt()*. The array can be retrieved using the function *genfromtxt()*. The syntax of these functions are
savetxt(fname,array,fmt= ,delimiter=)
Here fname is the name of the file to be created and opened for writing, array is the name of the array, fmt is the format specification of the data to be stored, delimiter is the character used to distinguish elements of the array.

genfromtxt(fname,dtype=,comments=# ,delimiter= ,skiprows=). Here *dtype* is the datatype of array elements and *skiprows* accepts a number which refers to the number of rows to skip from 0^{th} row.

```
>>> b
```

```
array([[  0.,   1.,   2.,   3.],
       [  4.,   5.,   6.,   7.],
       [  8.,   9.,  10.,  11.],
       [ 12.,  13.,  14.,  15.]])
>>> savetxt('f1.txt',b,fmt='%8.6f',delimiter='&')
>>> savetxt('f2.txt',b,fmt='%8.4f',delimiter=' ')

When f1.txt and f2.txt are opened in a text editor, the contents of the file
f1.txt
0.000000 &1.000000 &2.000000 &3.000000
4.000000 &5.000000 &6.000000 &7.000000
8.000000 &9.000000 &10.000000&11.000000
12.000000&13.000000&14.000000&15.000000

f2.txt
 0.0000  1.0000   2.0000   3.0000
 4.0000  5.0000   6.0000   7.0000
 8.0000  9.0000  10.0000  11.0000
12.0000 13.0000  14.0000  15.0000

>>> genfromtxt('f1.txt',dtype='float')
array([[  0.,   1.,   2.,   3.],
       [  4.,   5.,   6.,   7.],
       [  8.,   9.,  10.,  11.],
       [ 12.,  13.,  14.,  15.]])

>>> genfromtxt('f1.txt',skiprows=2)
array([[  8.,   9.,  10.,  11.],
       [ 12.,  13.,  14.,  15.]])
```

If the arrays are too large, saving them in text format consumes large volume of memory. In that case they can be saved in binary format.

```
>>> from numpy import *
>>> a=genfromtxt('f1.txt')
>>> save('f3.npy',a)
When the file f3.npy is opened in a word processor, the following output is c
ï¿½NUMPY##F#{'descr': '<f8', 'fortran_order': False, 'shape': (4, 4), }
##############ï¿½?#######@#######@#######@#######@#######@#######@####### @###

>>> b=load('f3.npy')
>>> b
```

```
array([[  0.,   1.,   2.,   3.],
       [  4.,   5.,   6.,   7.],
       [  8.,   9.,  10.,  11.],
       [ 12.,  13.,  14.,  15.]])
```

Basic Arithmetic Operations on arrays

Arithmetic operators apply elementwise on arrays. A new array is created and filled with the result.

```
>>> a = array( [20,30,40,50] )
>>> b = arange( 4 )
>>>b
array([0,1,2,3])
>>> c = a-b
>>> c
array([20, 29, 38, 47])
>>> b**2
array([0, 1, 4, 9])
>>> 10*sin(a)
array([ 9.12945251, -9.88031624,  7.4511316 , -2.62374854])
>>> a<35
array([True, True, False, False], dtype=bool)
>>> i = identity( 3 )
>>> i
array([[1, 0, 0],
       [0, 1, 0],
       [0, 0, 1]])
>>> i + i  # add element to element
array([[2, 0, 0],
       [0, 2, 0],
       [0, 0, 2]])
>>> i + 4  # add a scalar to every entry
array([[5, 4, 4],
       [4, 5, 4],
       [4, 4, 5]])
>>> a = array( range(1,10)).reshape(3,3)
>>> a
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

```
>>> i * a  # element to element
array([[1, 0, 0],
       [0, 5, 0],
       [0, 0, 9]])
>>> x = array( [1,2,3])
>>> x
array([1, 2, 3])
>>> y = array( [ [4], [5], [6] ])
>>> y
array([[4],
       [5],
       [6]])
>>> x + y
array([[5, 6, 7],
       [6, 7, 8],
       [7, 8, 9]])
       #This is equivalent to ([[1,2,3],[1,2,3],[1,2,3]]+
          #([[4,4,4],[5,5,5],[6,6,6]])
>>> x*y
array([[ 4,  8, 12],
       [ 5, 10, 15],
       [ 6, 12, 18]])
>>> x/y
array([[0, 0, 0],
       [0, 0, 0],
       [0, 0, 0]])
>>> y/x
array([[4, 2, 1],
       [5, 2, 1],
       [6, 3, 2]])
>>> x%y
array([[1, 2, 3],
       [1, 2, 3],
       [1, 2, 3]])
>>> y%x
array([[0, 0, 1],
       [0, 1, 2],
       [0, 0, 0]])
>>> x**y
array([[  1,  16,  81],
       [  1,  32, 243],
       [  1,  64, 729]])
>>> s=arange(1,6,1)
>>> s
array([1, 2, 3, 4, 5])
>>> s.sum() #sum of all elements
15
>>> s.prod()  # product of all elements
120
>>> s.mean() # Mean of all elements
3.0
>>> s.var() #Variance
2.0
>>> s.std() #Standard deviation
1.4142135623730951
```

2.1.6 Indexing, Slicing and Iterating

One dimensional arrays can be indexed, sliced and iterated over like lists and other Python sequences.

```
>>> a = arange(10)**3
>>> a
array([  0,   1,   8,  27,  64, 125, 216, 343, 512, 729])
>>> a[2]
8
>>> a[2:5]
array([ 8, 27, 64])
>>> a[:6:2] = -1000                          \# modify elements in a
>>> a[::-1]                                  \# reversed a
array([  729,   512,   343,   216,   125, -1000,    27, -1000,     1, -1000])
>>> for i in a: print i**(1/3.),
nan 1.0 nan 3.0 nan 5.0 6.0 7.0 8.0 9.0
```

2.1.7 Arrays as matrices

As the product operator '*' operates elementwise (product of corresponding elements) in *NumPy* arrays, the matrix product ($c_{ij} = \sum_k a_{ik} b_{kj}$) can be found using the *dot* function. It also gives the dot product of two vectors.The function *inner (x,y)* computes the inner product ($z_{ij} = \sum_k x_{ik}.y_{jk}$) between two arrays. For 1-D arrays *dot* and *inner* functions give the same result. Similarly a *cross* function is defined which returns the cross product of two vectors. *outer (x, y)* computes an outer product of two vectors ($z_{ij} = x_i.y_j$). In matrix class to create matrices *mat* method and *matrix* method are defined.*mat(data, dtype=)*,*matrix(data, dtype=)*. This *data* can be any *list, tuple, string* or *array* . This function interprets the input as a matrix.

```
>>> from numpy import*
>>> mat(range(2,7))
matrix([[2, 3, 4, 5, 6]])
>>> a,b=mat([[1,2],[3,4]]),mat('1,2;3,4')
>>> a
matrix([[1, 2],
        [3, 4]]),
>>> b
matrix([[1, 2],
        [3, 4]])
>>> r=mat('1,2,3,4')
>>> r
matrix([[1, 2, 3, 4]]) # Row matrix
>>> c=mat('1;2;3;4')
>>> c
matrix([[1],
        [2],
        [3],
        [4]]) # column matrix

>>> k=arange(15).reshape(3,5)
>>> k
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])

>>> k=mat(k) #array k becomes 3X5 matrix k
>>> t=transpose(k) # transposed 5X3 matrix
>>>l
matrix([[ 0,  5, 10],
        [ 1,  6, 11],
        [ 2,  7, 12],
        [ 3,  8, 13],
        [ 4,  9, 14]]))
>>> k*t   #It must be a 3X3 matrix
matrix([[ 30,  80, 130],
        [ 80, 255, 430],
        [130, 430, 730]])
```

```
>>> dot(k,t)
matrix([[ 30,  80, 130],
        [ 80, 255, 430],
        [130, 430, 730]])
>>> t.fill(3)
>>> t
array([[3, 3, 3],
       [3, 3, 3],
       [3, 3, 3],
       [3, 3, 3],
       [3, 3, 3]])

>>> s=range(1,6)
>>> m= mat(s)
>>> m
matrix([[1, 2, 3, 4, 5]]) #The two square brackets are there as most of the matrices
>>> n=matrix(range(1,6))
>>> n
matrix([[1, 2, 3, 4, 5]])
>>> x=[1,2,3]
>>> y=[3,2,1]
>>> dot(x,y) # like dot product of vectors
10
>>> cross(x,y) # like cross product of vectors
array([-4,  8, -4])
>>> inner([1,2,3],[10,100,1000])
3210  # 1.10+2.100+3.1000
>>> a=arange(9).reshape(3,3)
>>> b=a.T # another method to find transpose.
>>> a
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> b
array([[0, 3, 6],
       [1, 4, 7],
       [2, 5, 8]])
>>> inner(a,b) #Ordinary inner product of vectors for 1-D arrays (without complex co
product over the last axes.

array([[ 15,  18,  21],
       [ 42,  54,  66],
       [ 69,  90, 111]])
```

```
It is possible to perform increment and decrement operations without creating
>>> a = ones((2,3), dtype=int)  #integer array
>>> b = random.random((2,3)) #float array
>>> a *= 3
>>> a
array([[3, 3, 3],
       [3, 3, 3]])
>>> b += a
>>> b
array([[ 3.69092703,  3.8324276 ,  3.0114541 ],
       [ 3.18679111,  3.3039349 ,  3.37600289]])
>>> a += b                                  \# b is converted to integer type
>>> a
array([[6, 6, 6],
       [6, 6, 6]])
```

When operating with arrays of different numeric data types, the type of the resulting array corresponds to the more general or precise one.

```
>>> a = ones(3, dtype=int32)
>>> b = linspace(0,pi,3)
>>> b.dtype.name
float64
>>> c = a+b
>>> c
array([ 1.        ,  2.57079633,  4.14159265])
>>> c.dtype.name
'float64'
>>> d = exp(c*1j)
>>> d
array([ 0.54030231+0.84147098j, -0.84147098+0.54030231j,
       -0.54030231-0.84147098j])
>>> d.dtype.name
'complex128'
```

Many unary operations, like computing the sum of all the elements in the array, are implemented as methods of the *ndarray* class.

```
>>> a = random.random((2,3))
>>> a
array([[ 0.6903007 ,  0.39168346,  0.16524769],
       [ 0.48819875,  0.77188505,  0.94792155]])
>>> a.sum()
3.4552372100521485 #sum of all elements
>>> a.min()
0.16524768654743593
>>> a.max()
0.9479215542670073
```

By default, these operations apply to the array as if it were a list of numbers, regardless of its shape. However, by specifying the axis parameter you can apply an operation along the specified axis(dimension) of an array:

```
>>> b = arange(12).reshape(3,4)
>>> b
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>>
>>> b.sum(axis=0)                         # Give sum as a 1-D array(sum of each colu
array([12, 15, 18, 21])
>>> b.sum(axis=1)                         # Give sum as a 1-D array(sum of each row)
array([6, 22, 38])
>>> b.min(axis=1)                         # minimum of each row
array([0, 4, 8])
>>> b.min(axis=0)     # minimum of each column
array([0,1,2,3])
>>> p.max(axis=0)
matrix([[12, 13, 14, 15]])

>>> b.cumsum(axis=1)                         # cumulative sum along the rows
array([[ 0,  1,  3,  6],
       [ 4,  9, 15, 22],
       [ 8, 17, 27, 38]])
>>> p=arange(16).reshape(4,4)
>>> p
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
>>> trace(p)
30
 >>> p=mat(p)
>>> p
matrix([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11],
        [12, 13, 14, 15]])
>>> trace(p)
30
```

2.1.8 Arrays as polynomial coefficients

There are four methods defined in polynomial class to create and manipulate polynomials.

1. *poly1d(c, r, v)*: creates a one-dimensional polynomial. Here c represents an array or list. If r is *True*, c represents roots of the polynomial. If r is *False* (which is the default), polynomial coefficients zeroth element corresponding to the highest power of variable. v is the character to be used as polynomial variable.

   ```
   >>> c=[3,1,-1,-3]
   >>> p=poly1d(c)
   >>> print p
      3     2
   3 x + 1 x - 1 x - 3

   >>> p1=poly1d(c,True)
   >>> print p1
      4      2
   1 x - 10 x + 9
   >>> p2=poly1d(c,False,'y')
   >>> print p2
      3     2
   3 y + 1 y - 1 y - 3
   ```

2. *polyval(p, x)*: Here p is the polynomial and x is the value at which p is to be evaluated. The polynomial value at x is returned. For polynomials defined above

   ```
   >>> polyval(p2,2)
   23
   >>> polyval(p1,2)
   -15
   ```

```
>>> polyval(p,2)
23
```

3. *poly(s)*: This function returns the coefficients of the polynomial with the s as the set of roots in the form of an array.

```
>>> d=[0,0]
>>> poly(d)
array([1, 0, 0]) #x*x=0
>>> d=[1,-1,2]
>>> f=poly(d) #(x-1)(x+1)(x-2)
>>> f
array([ 1, -2, -1,  2])
>>> print poly1d(f)
   3     2
1 x - 2 x - 1 x + 2
```

4. roots(p): Returns the roots of polynomial p

```
>>> g=[1,-2,-1,2]
>>> p=poly1d(g)
>>> roots(p)
array([-1.,  2.,  1.])
Roots can also be found using the following command.
>>> p.r
array([-1.,  2.,  1.])
```

2.1.9 Linear Algebra

The linear algebra module is a subclass of *numpy*. It is called *linalg*. A few functions are defined in the *NumPy.linalg* sub-package. The important functions are

1. *norm(x)*: Returns norm of a vector x, $norm = \sqrt{\sum_i x_i^2}$
2. *det(a)*: Returns determinant of a square matrix
3. *inv(a)*: Returns inverse of a non-singular square matrix
4. *pinv(a)*: Returns pseudoinverse of a singular square matrix. For invertible matrices, this is the same as the inverse.
5. *solve(a,y)*: Returns the solution vector x to the linear equation $ax = y$

6. *eig(a)*:Return all solutions (λ, x) to the equation $ax = \lambda x$. The first element of the return tuple contains all the eigenvalues. The second element of the return tuple contains the eigenvectors (i^{th} eigenvector as i^{th} column).
7. *eigvals(a)*: Returns all eigenvalues of square matrix a as an array
8. *eigh(h)*: Return all solutions (λ, x) to the equation $hx = \lambda x$ where h is a hermitian matrix.
9. *eigvalsh(h)*:Returns all eigenvalues of hermitian matrix h as an array

These are also included in the sub-package *numpy.dual.*

Let a be a square matrix $\begin{pmatrix} 1 & 1 & 2 \\ -1 & 0 & 1 \\ 2 & 3 & 0 \end{pmatrix}$. It is created as

```
>>> from numpy import*
>>> from numpy.linalg import*
>>> a=array([[1,0,1],
 [2,1,0],
 [0,2,4]])
>>> det(a)
8.0
>>> inv(a)
array([[ 0.5   ,  0.25  , -0.125],
       [-1.    ,  0.5   ,  0.25 ],
       [ 0.5   , -0.25  ,  0.125]])
>>> eigvals(a)
array([ 0.8223493+1.07730381j,  0.8223493-1.07730381j,  4.3553014+0.j])
>>> eig(a)
(array([ 0.8223493+1.07730381j,   0.8223493-1.07730381j,    4.3553014+0.j   ]
 #eigenvalues

array([[-0.06908062+0.41891651j, -0.06908062-0.41891651j,  0.28156897+0.j ],
       [ 0.77771286+0.j        ,  0.77771286+0.j        ,  0.16783528+0.j ],
       [-0.43902814-0.14884162j, -0.43902814+0.14884162j,  0.94474877+0.j ]])
# Three eigenvectors
```

To solve the simultaneous equations

$$2x + 3y + 4z = 8,\ 3x + 4y + 5z = 10,\ 4x - 5y + 6z = 32$$

```
>>> a=mat('2,3,4;3,4,5;4,-5,6') # The coefficient matrix
>>> a
matrix([[ 2,  3,  4],
        [ 3,  4,  5],
        [ 4, -5,  6]])
>>> s=mat('8;10;32') # Constant vector S in matrix equation AX=S
>>> s
matrix([[ 8],
        [10],
        [32]])
>>> b=solve(a,s)
>>> b
matrix([[ 1.],
        [-2.],
        [ 3.]])
```

Chapter 3

Plotting and visualization

3.1 Matplotlib

Graphs, charts, surface plots etc are visual presentations of numerical data. It is useful to compare, contrast, detect trends, predict and comprehend huge amounts of data. Different Python modules are used for generating two and three dimensional graphs.

3.1.1 The Matplotlib Module

The python package *Matplotlib* produces graphs and figures in different hardcopy formats like *jpeg, bmp, eps, png* etc. Most of the functions of *NumPy* and *matplotlib.pyplot* are defined in the module *pylab* also. It also provides many functions for matrix manipulation. The data for plotting are supplied as Python *lists* or Numpy *arrays.* This module contains a lot of methods for plotting and annotating graphs. Some of these methods are used frequently in scientific computing. They are listed below with examples.

1. *plot()Function*: The general format is

   $$plot(x, y, color =, linestyle =, marker =, markerfacecolor =, markersize =)$$

   x,y are lists or arrays, *color* color of graph, *linestyle* specifies dashedline('–'),solid line('-'), dotted line (':') etc., *marker*marking data points on the graph ('.','*'), *markerfacecolor* the color of marker and *markersize* is the size of marker.

2. *show()*: Sends graphic output to the screen.

3. *xlabel(")*: " is name of variable plotted along x-axis.

4. *ylabel(")*: " is name of variable plotted along y-axis.

5. *title(")*: " is name of Graph.

6. *legend(names,loc=)*: The *names* of different curves as a list or tuple of strings, *loc* is the location in the graph where the legend must appear. *Upper right, lower left* etc. *loc=0* fits the legend at the most convenient location.

7. *grid(True)*: Shows grid lines if *True*

8. *axis(z)*: Used to set or get the axis properties.
   axis() Returns the current axes limits [xmin, xmax, ymin, ymax].
   axis(z):Sets the min and max of the x and y axes, with z = [xmin, xmax, ymin, ymax].
   axis('off'): Removes the axis lines and labels.
   axis('equal'): Changes limits of x or y axis so that equal increments of x and y have the same length so that a circle will appear circular.

9. *figure()*

   $$figure(num, figsize = (w, h), dpi = N, facecolor =' b', edgecolor =' k')$$

   num is an integer variable. This function creates a new figure if *figure(num)* does not exist. If it already exists, it becomes active. *figsize=(w,h)* is a tuple of width and height in inches, *dpi=N* creates figure with resolution N dots per square inch, *facecolor* sets the backgroundcolor *edgecolor* sets the border color. All arguments except the first are optional.

10. *subplot(m, n, N)*: Where m is the number of rows, n number of columns and $N = 1$ is the first plot number and increasing N fill rows first. $N_{max} = m \times n$

11. *text(x,y,'string')* Writes 'string' at (x, y) with respect to bottom-left corner as origin and scaled as in the figure.

12. *bar(left, height, width=0.8, bottom=0,color=None, edgecolor=None, linewidth=None,ye xerr=None, ecolor=None, capsize=3,align='edge', orientation='vertical', log=False)* Make a bar plot with rectangles representing the two arrays *left*, *height* bounded by left, left + width, bottom, bottom + height with optional arguments whose default values are given.

13. *barh()*Like bar() except that the bars are horizontal.

14. *contour()*For plotting implicit functions.

15. *contourf()*For color-filled plotting of implicit functions.

16. *loglog()*Make a plot with log scaling on the x and y axis. It supports all the keyword arguments of *plot()* function.

17. *semilogx()* Graph with x-axis plotted in log scale. It supports all the keyword arguments of *plot()* function.

18. *semilogy()*Graph with y-axis plotted in log scale.

19. *ogrid[minx:maxx:nxj,miny:maxy:nyj]*Creates a grid of *nx* x-vlues in the range *(xmin,xmax)*and *ny* y-values in the range *(ymin,ymax)*.eg. *x,y=ogrid[1:10:100j,5:15:150j]*

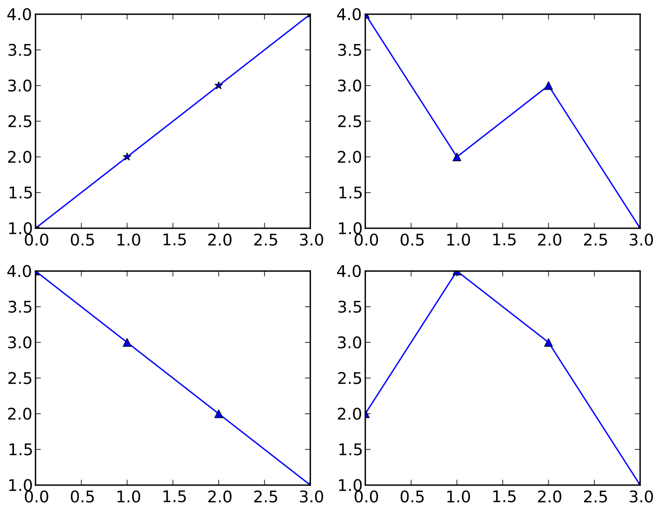
20. *pie()pie(x, explode=None, labels=None, colors=None, autopct=None, pctdistance=0.6,shadow= labeldistance=1.1, hold=None)* Makes a pie chart of array x. The fractional area of each wedge is given by *x/sum(x)*. If sum(x) <= 1, then the values of x give the fractional area directly and the array will not be normalized.

21. *polar(theta, r, args)* Make a polar plot. *theta* and *r* are lists or arrays. Multiple theta, r arguments are supported, with format strings, as in plot().

22. *scatter(x, y, s=20, c='b', marker='o', cmap=None, norm=None,vmin=None, vmax=None, alpha=1.0, linewidths=None,verts=None, **kwargs)* Make a scatter plot of x versus y, where x, y are converted to 1-D sequences which must be of the same length,N.

Examples

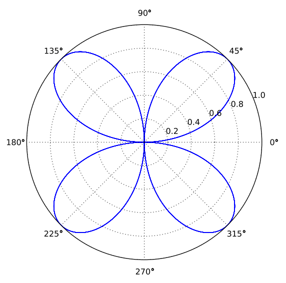
```
from matplotlib.pyplot import*
from numpy import*
subplot(2,2,1)
plot([1,2,3,4],'*-')
subplot(2,2,2)
plot([4,2,3,1],'^-')
subplot(2,2,3)
plot([4,3,2,1],'^-')
subplot(2,2,4)
plot([2,4,3,1],'^-')
show()
```



Polar plots: Polar coordinates locate a point on a plane with one distance and one angle. The distance ‘r’ is measured from the origin. The angle θ is measured from positive direction of x-axis in the anti-clockwise sense. Plotting is done using $polar(theta, radius, format\ string)$ function. An example is given below.

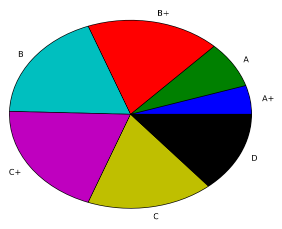
Polar rhodonea A *rhodonea* or *rose curve* is a sinusoid $r = \sin(n\theta)$ where n is a constant. If n is an integer the curve will have $2n$ 'petals' and n 'petals' if n is odd. If n is a rational number(=p/q, p,q integers), then the curve is closed and has finite length. If n is an irrational number, then it is closed and has infinite length.

```
from matplotlib.pyplot import*
from numpy import*
n=2
th = linspace(0, 10*pi,1000)
r = sin(n*th)
polar(th,r)
show()
```



Pie Charts: A pie chart is a circular chart in which a circle is divided into sectors. Each sector visually represents an item in a data set to match the percentage or fraction of the item in the total data set. Pie charts are useful to compare different parts of a whole amount. They are often used to present financial information. The function *pie(list of percentages or fractions , labels=list of labels)* produces a pie chart. Both the lists must have the same length.

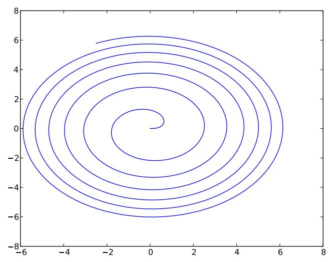
```
from matplotlib.pyplot import*
from numpy import*
labs = ['A+', 'A', 'B+', 'B', 'C+', 'C','D']
fracs = [5,8,18, 19, 20,17,14]
pie(fracs, labels=labs)
show()
```



Parametric plots: A parametric plot is a visual description of a set of parametric equations. If x and y are both functions of a variable t, then they create a set of parametric equations. For example, the two equations $y = t\sin t^2$ and $x = t\cos t^2$ form a set of parametric equations in which y and x are functions of t, the graph of which will be in this form.

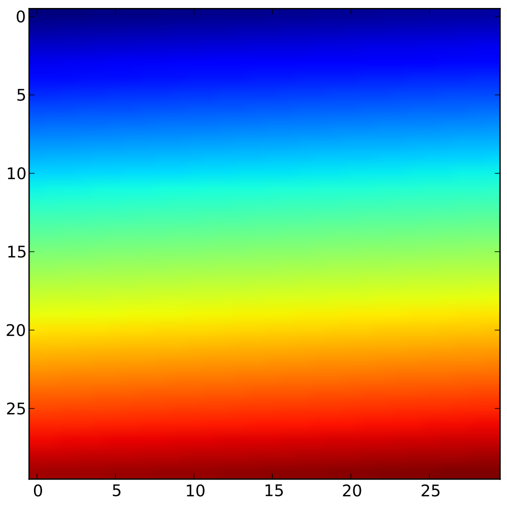
```
from matplotlib.pyplot import*
from numpy import*
```

```
t=arange(0,6.3,0.001)
x=t*cos(t*t)
y=t*sin(t*t)
plot(x,y)
show()
```



2-D plots in colours: Two dimensional matrix can be represented graphically by assigning a color to each point proportional to the value of that element. The function *imshow(matrix)* is employed to create such plots.

```
from matplotlib.pyplot import*
from numpy import*
m=linspace(0,1,900).reshape(30,30)
imshow(m)
show()
```



3.1.2 Plotting mathematical functions

sine function:

```
from pylab import*
x=linspace(0,2*pi,200)
y=sin(x)
plot(x,y)
xlabel('x')
ylabel('sin(x)')
title('Plot of sine function')
grid(True)
show()
```

Logarithm function: Logarithm function *log(x)* gives logarithm of a variable to the base exponential e.

```
from pylab import*
x=linspace(0,200,200)
y=log(x)
plot(x,y)
xlabel('x')
ylabel('log(x)')
title('Plot of log function')
grid(True)
show()
```

Exponential function: Exponential function *exp(x)* gives e^x of a variable x.

```
from pylab import*
x=linspace(0,5,200)
y=exp(x)
plot(x,y)
xlabel('x')
ylabel('exp(x)')
title('Plot of exponential function')
grid(True)
show()
```

Gaussian function: Gaussian function is given by $y = exp(-x^2)$ gives e^x of a variable x.

```
from pylab import*
x=linspace(-5,5,200)
y=exp(-x**2)
plot(x,y)
xlabel('x')
ylabel('gaussian(x)')
title('Plot of Gaussian function')
grid(True)
show()
```

Gamma function: The gamma function is defined by the integral

$$\Gamma(x) = \int_0^\infty t^{x-1}e^{-t}dt$$

and satisfies the recurrence relation

$$\Gamma(x+1) = x\Gamma(x)$$

and reflection formula

$$\Gamma(x)\Gamma(1-x) = \frac{\pi}{\sin \pi x}$$

$$\Gamma(1+x)\Gamma(1-x) = \frac{\pi x}{\sin(\pi x)}$$

$$\Gamma(1-x) = \frac{\pi x}{\Gamma(1+x)\sin(\pi x)}$$

It can be used to calculate Γ-functions less than 1. Since $\sin(\pi x)$ is zero for integer x, Γ-function is unbounded for negative integers but not for negative fractions. The following approximation method,derived by Lanczos, is employed for calculating the Γ-function numerically. For $x > 0$,

$$\Gamma(1+x) = k^{k-5}e^{-k}\sqrt{2\pi}\left(a_0 + \frac{a_1}{x+1} + \frac{a_2}{x+2} + ... + \frac{a_n}{x+n} + \epsilon\right)$$

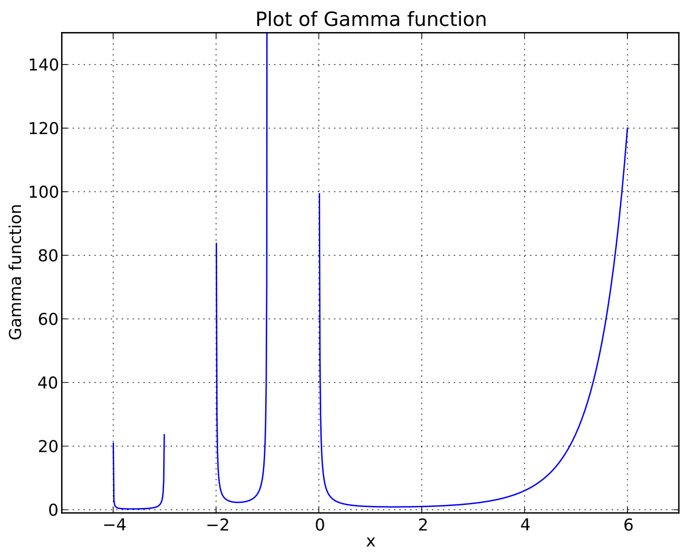
where $k = x + 5.5$, ϵ the error term and a_i expansion coefficients. For $|\epsilon| < 2 \times 10^{-10}$, $n = 6$ is sufficient. The coefficients are given by
$a_0 = 1.00002746310005$, $a_1 = 76.18009172947146$
$a_2 = -86.50532032941677$, $a_3 = 24.01409824083091$
$a_4 = -1.231739572450155$, $a_5 = 1.208650973866179 \times 10^{-3}$,
$a_6 = -5.395239384953 \times 10^{-6}$

$$\ln\Gamma(1+x) = ln(k^{k-5}) - k + ln(\sqrt{2\pi}) + \ln\left(a_0 + \frac{a_1}{x+1} + \frac{a_2}{x+2} + ... + \frac{a_n}{x+n}\right)$$

$$\ln\Gamma(x) = \ln\left(k^{k-5}\right) - k + \ln\left[\frac{\sqrt{2\pi}}{x}\left(a_0 + \frac{a_1}{x+1} + \frac{a_2}{x+2} + ... + \frac{a_n}{x+n}\right)\right]$$

But reasonable accuracy can be achieved by just taking the first five terms and approximating elements of a to 2 decimal places.

```
from pylab import*
def gamma(x0):
    a=[1,76.18,-86.505,24.014,-1.232]
    k=x0+5.5
    s=(k-5)*log(k)-k
    s1=a[0]
    for i in range(1,5):
        s1+=a[i]/(x0+i)
    return exp(s+log(sqrt(2*pi)*s1/x0))
p=linspace(-5,6,1000)
z=gamma(p)
plot(p,z)
axis([-5,7,-1,150])
xlabel('x')
ylabel('Gamma function')
title('Plot of Gamma function')
grid(True)
show()
```



Polynomial Evaluation: Legendre Function Legendre functions $P_n(x)$ are defined through a generating function as

$$\frac{1}{\sqrt{1-2xt+t^2}} = \sum_{n=0}^{\infty} P_n(x)t^n$$

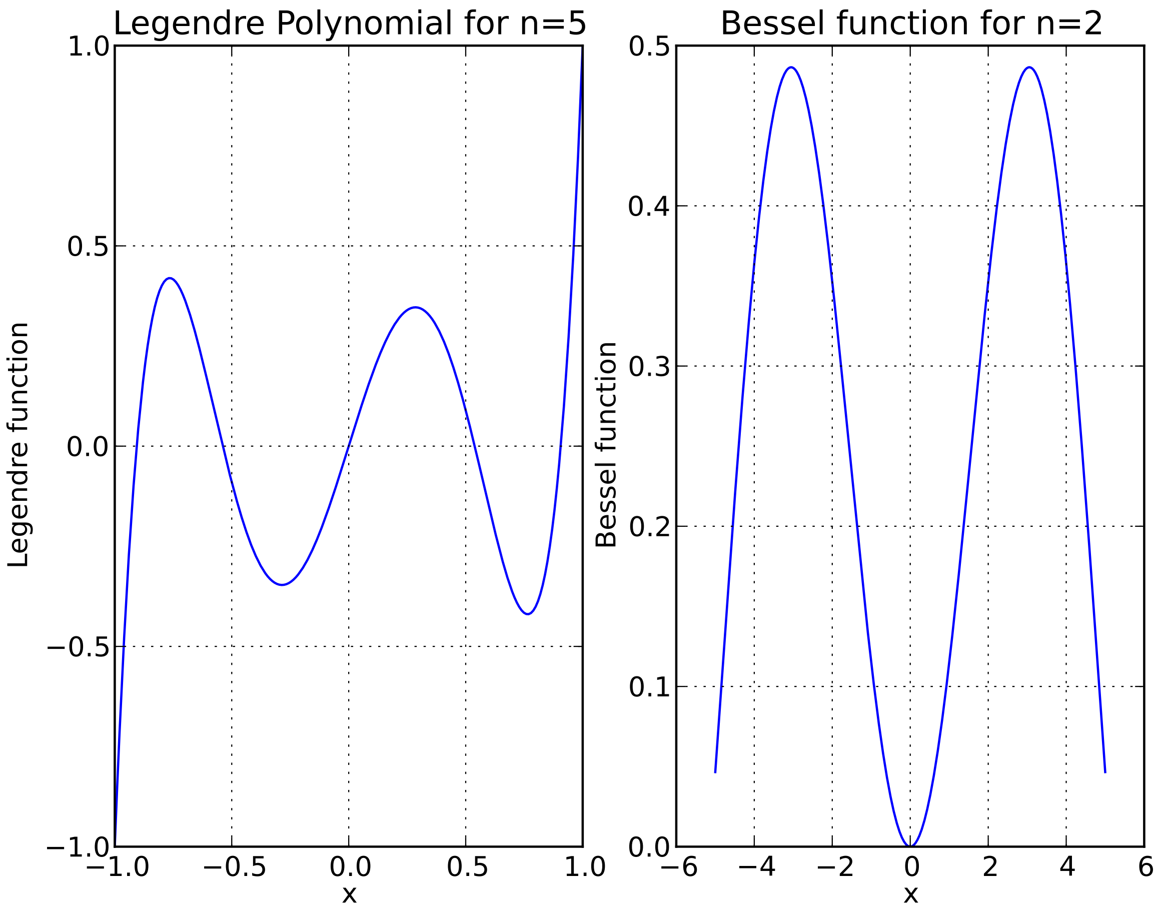
where $|t| \leq 1$ The zeroth term $(n = 0)$ of the expansion is $1 = P_0(x)t^0$ Hence $P_0(x) = 1$ for all x. Expanding left side as a binomial series and equating coefficient of t on both sides, $xt = P_1(x)t$ or $P_1(x) = x$ for all x. Differentiating and equating coefficients of t^n on both sides one gets

$$P_{n+1}(x) = \frac{(2n+1)xP_n(x) - nP_{n-1}(x)}{n+1}$$

This is a recurrence relation which may be used for calculating Legendre polynomials of any order.

```
from pylab import*
def legendre(m,z):
p0,p1=1,z
if m==0:p=p0
if m==1:p=p1
if m>1:
    for i in range(1,m):
        p=((2*i+1)*z*p1-i*p0)/(i+1)
        p0,p1=p1,p
return(p)

n=input('Give Order n of Legendre function')
x=linspace(-1,1,200)
y=legendre(n,x)
plot(x,y)
xlabel('x')
ylabel('Legendre function')
title('Plot of Legendre Polynomial for n=5')
grid(True)
show()
```



Polynomial Evaluation: Bessel's Function The Bessel function of order n for a variable x is given by the series

$$J_n(x) = \sum_{s=0}^{\infty} \frac{(-1)^s}{s!(n+s)!} \left(\frac{x}{2}\right)^{2s+n}$$

The zeroth term ($s = 0$) of the expansion is $\frac{1}{n!}\left(\frac{x}{2}\right)^n$. The ratio of the p^{th} and $(p-1)^{th}$ term of the expansion is $\frac{-1}{p(n+p)}\left(\frac{x}{2}\right)^2$. Using these results Bessel function of any order can be calculated for any desired accuracy.

```
from pylab import*
def bessel(m,z):
    z/=2.0
    factorial=1
    for i in range(1,n+1):factorial*=i
    term=[z**n/factorial]
    for i in range(1,10):
        term.append(-term[i-1]*z*z/(i*(i+n)))
    return sum(term)

n=input('Give Order n of Bessel function : ')
x=linspace(-5.0,5.,1000)
y=[bessel(n,i) for i in x]
plot(x,y)
xlabel('x')
ylabel('Bessel function')
title('Plot of Bessel function for n=2')
grid(True)
show()
```

Chapter 4

Numerical Analysis

4.1 Numerical methods

4.1.1 Inverse of a function

Inverse functions are often used in physics. For example, consider the length L of the mercury column in a capillary tube as function of temperature T. $L = f(T)$. When it is used as a thermometer, the length of the mercury pellet is measured and the temperature is inferred from it using the formula $T = f^{-1}(L)$. Similarly, in a piezoelectric crystal, the voltage V developed is a function of stress S applied. $V = f(S)$. In a strain gauge, this voltage is measured to estimate the load W placed on it. $W = Af^{-1}(V)$ where A is the surface area of the crystal on which the load applies stress. If a function represents a principle, in general, its application employs the inverse function.

Definition 1 (Function) *For every x in a set X, if an object f maps exactly one element y in set Y then f is called a function with domain X, and co-domain or range Y.*

It is represented as

$$\forall x \in X,\ y \in Y,\ f : x \rightarrow y$$

y is called the image and x, the preimage. If there is more than one preimage for a given image, then f is called an *onto mapping*. For example, $f : x \rightarrow y$ has the explicit form $y = x^2$, then image y is the same for all $\pm x$ in the domain X. If for every image, there is a unique preimage, f is a *one-to-one mapping*. For example, $f : x \rightarrow y$ has the explicit form $y = x^3$, then image y is the unique for every x in the domain X. In terms of domain and co-domain sets $f : X \to Y$.

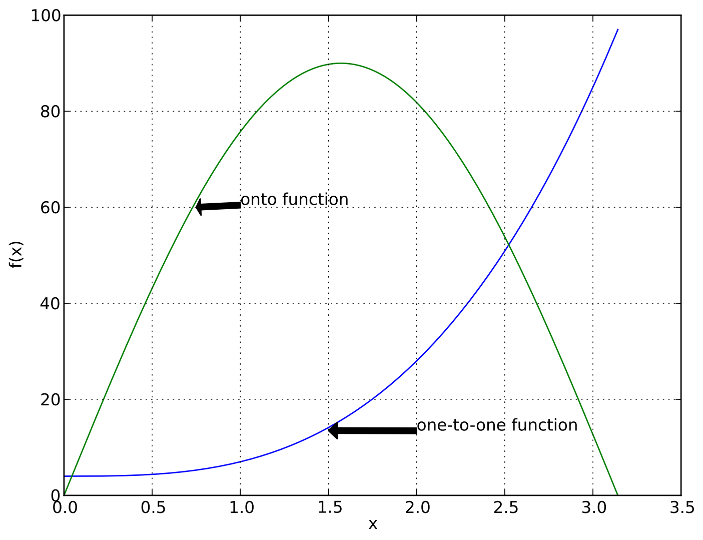
Definition 2 (Inverse Function) *If f is a function whose domain is the set X, and range is the set Y and there exists a function g with domain Y and range X, with the property*

$$\forall x \in X, f(x) = y \in Y \textit{ if and only if } \forall y \in Y,\ g(y) = x$$

then g is called inverse of f.

The function f sends all elements of the the domain X to the range Y. If f is invertible, the function g is unique. There is exactly one function g satisfies this property. Function g is called the inverse of f, denoted formally as f^{-1}. $(\neq 1/f)$. Since f implies unique y for each x and g implies unique x for each y, the two mappings f and g must be one-to-one. This is a necessary condition for f to have an inverse.

Test to check whether a function is one-to-one: It is called *the horizontal line test.* If no line parallel to x-axis intersects the graph of the function $y = f(x)$ at more than one point, that function is one-to-one.



Method of finding inverse

There are different methods of finding inverse function.

Algebraic method: The algorithm for finding an inverse function g for $f(x)$ algebraically involves the following steps.

1. Check whether f is a one-to-one mapping.
2. Put $f(x) = y$
3. Swap the x and y variables
4. Solve for y. It gives $f^{-1}(x)$
5. Verify that $f^{-1}(f(x)) = x$ or $f(f^{-1}(x)) = x$

Example 1 *Find inverse of* $f(x) = 3x + 4$

1. f is a one-to-one function.
2. Put $3x + 4 = y$
3. Swap variables: $3y + 4 = x$
4. solve for y: $y = (x - 4)/3 = f^{-1}(x)$
5. $f(f^{-1}(x)) = 3 * [(x - 4)/3] + 4 = x$

Hence inverse function is $f^{-1}(x) = (x - 4)/3$

Problem 1 *Find inverse of* $f(x) = (x + 1)/x$

Answer:$f^{-1}(x) = 1/(x - 1)$

Problem 2 *Find inverse of* $f(x) = \log x$

Answer:$f^{-1}(x) = e^x$

Problem 3 *Find inverse of* $f(x) = (x + 1)/x$

Answer:$f^{-1}(x) = 1/(x - 1)$

Swapping method: Often the experimental data related through some function will be in the form of ordered pairs formed from tables.

x	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
$y = f(x)$	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8	y_9

Then $f(x)$ may be expressed as ordered pairs $f(x) : (x_i, y_i))$, $i = 1, 2, 3...$ Here

$$f(x) : (x_1, y_1), (x_2, y_2), (x_3, y_3), (x_4, y_4), (x_5, y_5), (x_6, y_6), (x_7, y_7), (x_8, y_8)(x_9, y_9)$$

If $\forall i, j,\ x_i \neq x_j$, and $y_i \neq y_j$, then the function $f(x)$ is one-to-one. Its inverse will exist and can be obtained by simply swapping x and y values.

$$f^{-1}(x) : (y_1, x_1), (y_2, x_2), (y_3, x_3), (y_4, x_4), (y_5, x_5), (y_6, x_6), (y_7, x_7), (y_8, x_8)(y_9, x_9)$$

In tabular form, it will appear as

x	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8	y_9
$f^{-1}(x)$	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9

Example 2 *Find the inverse of the function*

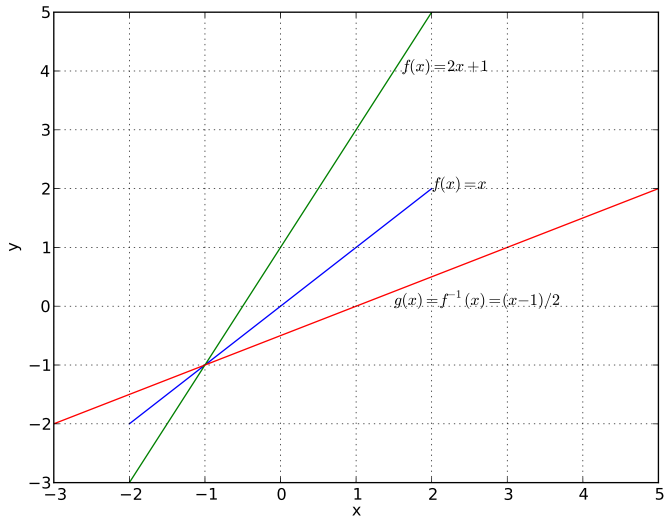
x	1	-2	-1	0	2	3	4	-3
$f(x)$	2	0	3	-1	1	-2	5	1

Swapping x and y, we get the inverse function as x and y never repeats among themselves.

$$f^{-1}(x) = (2, 1), (0, -2), (3, -1), (-1, 0), (1, 2), (-2, 3), (5, 4), (1, -3)$$

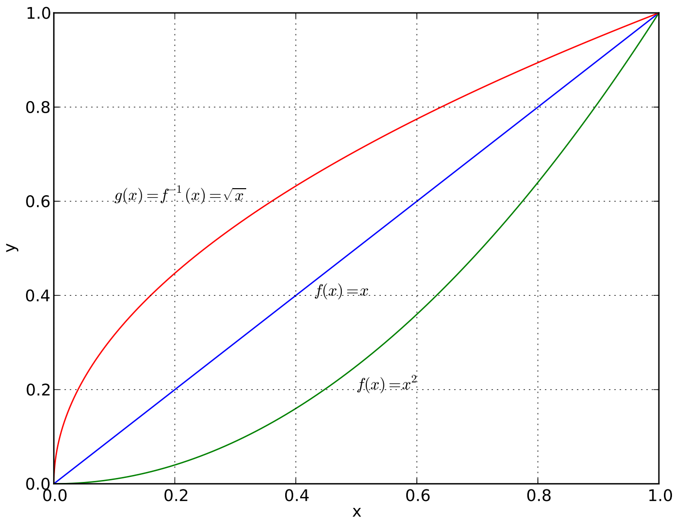
Graphical method: The basic principle is that the graph of an inverse relation is the *reflection* of the graph of original relation on the *identity line* (slope=1),$y = x$.

Example 3 *The function $f(x) = 2x + 1$ has $f^{-1}(x) = (x - 1)/2$. Plots are given below*



Often it is necessary to restrict the domain on certain functions to guarantee that the inverse relation is also a function.

Example 4 *For example if $y = ax^2$ is the function, then it is one-to-one only for $x > 0$.*

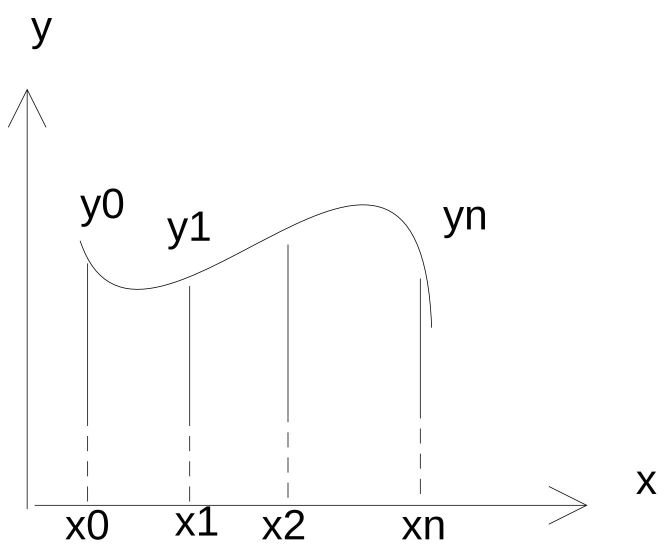


[1]

[1]Note that all graphs will not produce an inverse relation which is also a function.

4.1.2 Interpolation with Cubic Spline

Real world numerical data is usually difficult to analyse. Any function which would effectively correlate the data would be difficult to obtain and highly unwieldy. To this end, the idea of the cubic spline was developed. Using this process, a series of unique cubic polynomials are fitted between each of the data points, with the stipulation that the curve obtained be continuous and appear smooth. These cubic splines can then be used to determine rates of change and cumulative change over an interval.



Theory:
Let there be $(n+1)$ data points $(x_i, y_i), i = 0, 1, 2, ...n$. The essential idea is to fit a piecewise function of the form

$$\mathfrak{P}(x) = \begin{cases} p_0(x) & x_0 \leq x < x_1 \\ p_1(x) & x_1 \leq x < x_2 \\ ... & ... \\ p_{n-1}(x) & x_{n-1} \leq x < x_n \end{cases}$$

where each $p_i(x)$ is a cubic polynomial of the form

$$p(x) = a_3(x - x_i)^3 + a_2(x - x_i)^2 + a_1(x - x_i) + a_0$$

. To make the interpolation continuous, smooth and well-behaved, we impose the following conditions

1. The piecewise function $\mathfrak{P}(x)$ will interpolate all data points. That is $\mathfrak{P}(x_i) = p_i(x_i) = y_i$

2. $\mathfrak{P}(x)$ will be continuous on the interval (x_0, x_n). That is $p_i(x_i) = p_{i-1}(x_i)$
3. The first derivative $\mathfrak{P}'(x)$ will be continuous on the interval (x_0, x_n). that is, $p_i'(x_i) = p_{i-1}'(x_i)$
4. The second derivative$\mathfrak{P}$"(x) will be continuous on the interval (x_0, x_n). That is, $p_i''(x_i) = p_{i-1}''(x_i)$

The second derivative of a cubic polynomial is of degree one (a straight line). For the first pair of points $(x_0, y_0), (x_1, y_1)$ the following form may be chosen.

$$p_0''(x) = a_0 \frac{x - x_1}{x_0 - x_1} + a_1 \frac{x - x_0}{x_1 - x_0}$$

a_0 and a_1 are then given by

$$p_0''(x_0) = a_0,\, p_0''(x_1) = a_1$$

which are the values of the second derivative of $p_0(x)$ at $x = x_0$ and $x = x_1$. Integrating $p_0''(x)$ with respect to x

$$p_0'(x) = a_0 \frac{(x - x_1)^2}{2(x_0 - x_1)} + A + a_1 \frac{(x - x_0)^2}{2(x_1 - x_0)} + B$$

where $A.B$ are constants of integration. The two constants are different because the first term is integrated with respect to $(x - x_1)$ while the second term is integrated with respect to $(x - x_0)$ Integrating again

$$p_0(x) = a_0 \frac{(x - x_1)^3}{6(x_0 - x_1)} + A(x - x_1) + a_1 \frac{(x - x_0)^3}{6(x_1 - x_0)} + B(x - x_0)$$

To find A, B we use the two given points (x_0, y_0) and (x_1, y_1). Substituting $x = x_0$

$$p_0(x_0) = a_0 \frac{(x_0 - x_1)^3}{6(x_0 - x_1)} + A(x_0 - x_1) + a_1 \frac{(x_0 - x_0)^3}{6(x_1 - x_0)} + B(x_0 - x_0)$$

$$y_0 = a_0 \frac{(x_0 - x_1)^3}{6(x_0 - x_1)} + A(x_0 - x_1)$$

$$A = \frac{y_0}{x_0 - x_1} - \frac{a_0(x_0 - x_1)}{6}$$

Similarly using $p_0(x_1) = y_1$

$$y_1 = p_0(x_1) = a_0 \frac{(x_1 - x_1)^3}{6(x_1 - x_1)} + A(x_1 - x_1) + a_1 \frac{(x_1 - x_0)^3}{6(x_1 - x_0)} + B(x_1 - x_0)$$

$$B = \frac{y_1}{x_1 - x_0} - \frac{a_1(x_1 - x_0)}{6}$$

The polynomial has the form

$$\begin{aligned} p_0(x) &= \frac{a_0}{6}\frac{(x-x_1)^3}{(x_0-x_1)} + \left[\frac{y_0}{x_0-x_1} - \frac{a_0}{6}(x_0-x_1)\right](x-x_1) \\ &+ \frac{a_1}{6}\frac{(x-x_0)^3}{(x_1-x_0)} + \left[\frac{y_1}{x_1-x_0} - \frac{a_1}{6}(x_1-x_0)\right](x-x_0) \end{aligned}$$

Similarly the polynomial between (x_1, y_1) and (x_2, y_2) will have the form

$$\begin{aligned} p_1(x) &= \frac{a_1}{6}\frac{(x-x_2)^3}{(x_1-x_2)} + \left[\frac{y_1}{x_1-x_2} - \frac{a_1}{6}(x_1-x_2)\right](x-x_2) \\ &+ \frac{a_2}{6}\frac{(x-x_1)^3}{(x_2-x_1)} + \left[\frac{y_2}{x_2-x_1} - \frac{a_2}{6}(x_2-x_1)\right](x-x_1) \end{aligned}$$

To evaluate the second derivatives $a_i = p_i''(x_i)$, we use the condition that the first derivative be continuous at the knots. That is, $p_0'(x_1) = p_1'(x_1)$.

$$\begin{aligned} p_0'(x) &= \frac{3a_0}{6}\frac{(x-x_1)^2}{(x_0-x_1)} + \frac{y_0}{x_0-x_1} - \frac{a_0}{6}(x_0-x_1) \\ &+ \frac{3a_1}{6}\frac{(x-x_0)^2}{(x_1-x_0)} + \frac{y_1}{x_1-x_0} - \frac{a_1}{6}(x_1-x_0) \end{aligned}$$

$$p_0'(x_1) = \frac{y_0}{x_0-x_1} - \frac{a_0}{6}(x_0-x_1) + \frac{a_1}{2}(x_1-x_0) + \frac{y_1}{x_1-x_0} - \frac{a_1}{6}(x_1-x_0)$$

Rearranging and simplifying

$$\begin{aligned} p_0'(x_1) &= \frac{y_0}{x_0-x_1} - \frac{a_0}{6}(x_0-x_1) + \frac{a_1}{2}(x_1-x_0) + \frac{y_1}{x_1-x_0} - \frac{a_1}{6}(x_1-x_0) \\ &= \frac{y_1-y_0}{x_1-x_0} + \frac{a_0}{6}(x_1-x_0) + \frac{2a_1}{6}(x_1-x_0) \end{aligned}$$

Similarly

$$\begin{aligned} p_1'(x) &= \frac{3a_1}{6}\frac{(x-x_2)^2}{(x_1-x_2)} + \frac{y_1}{x_1-x_2} - \frac{a_1}{6}(x_1-x_2) \\ &+ \frac{3a_2}{6}\frac{(x-x_1)^2}{(x_2-x_1)} + \frac{y_2}{x_2-x_1} - \frac{a_2}{6}(x_2-x_1) \end{aligned}$$

At $x = x_1$,

$$p_1'(x_1) = \frac{3a_1}{6}\frac{(x_1-x_2)^2}{(x_1-x_2)} + \frac{y_1}{x_1-x_2} - \frac{a_1}{6}(x_1-x_2) + \frac{y_2}{x_2-x_1} - \frac{a_2}{6}(x_2-x_1)$$

Which can be simplified as

$$p_1'(x_1) = \frac{y_2-y_1}{x_2-x_1} - \frac{2a_1}{6}(x_2-x_1) - \frac{a_2}{6}(x_2-x_1)$$

Therefore continuity of first derivative requires

$$\frac{y_1-y_0}{x_1-x_0}+\frac{a_0}{6}(x_1-x_0)+\frac{2a_1}{6}(x_1-x_0)=\frac{y_2-y_1}{x_2-x_1}-\frac{2a_1}{6}(x_2-x_1)-\frac{a_2}{6}(x_2-x_1)$$

$$\frac{a_0}{6}(x_1-x_0)+\frac{2a_1}{6}(x_1-x_0)+\frac{2a_1}{6}(x_2-x_1)+\frac{a_2}{6}(x_2-x_1)=\frac{y_2-y_1}{x_2-x_1}-\frac{y_1-y_0}{x_1-x_0}$$

The second and third terms on the left hand side may be combined

$$\frac{a_0}{6}(x_1-x_0)+\frac{2a_1}{6}(x_2-x_0)+\frac{a_2}{6}(x_2-x_1)=\frac{y_2-y_1}{x_2-x_1}-\frac{y_1-y_0}{x_1-x_0}$$

For the polynomials $p_1(x)$ between (x_1,x_2) and $p_2(x)$ between (x_2,x_3),

$$\frac{a_1}{6}(x_2-x_1)+\frac{2a_2}{6}(x_3-x_1)+\frac{a_3}{6}(x_3-x_2)=\frac{y_3-y_2}{x_3-x_2}-\frac{y_2-y_1}{x_2-x_1}$$

There are four unknowns a_0,a_1,a_2,a_3 and two equations. So the values of any two of them must be assumed to get the other two. As two of the variables repeat in every successive equations (a_1, a_2 here), if we compute all the n-polynomials between the $(n+1)$ data points, we will get $(n-1)$ equations and $(n+1)$ variables. Usually it is assumed that $a_0=a_n=0$ so that there are $(n-1)$ variables and $(n-1)$ equations which can be exactly solved. Cubic spline with $a_0=a_n=0$ is called natural splin4e

If the x-values are equispaced, let $x_{i+1}-x_i=h$, then the $(n-1)$ equations may be written as

$$\begin{pmatrix} 4 & 1 & 0 & ... & 0 & 0 & 0\\ 1 & 4 & 1 & ... & 0 & 0 & 0\\ 1 & 4 & 1 & ... & 0 & 0 & 0\\ .. & .. & .. & ... & .. & .. & ..\\ 0 & 0 & 0 & ... & 1 & 4 & 1\\ 0 & 0 & 0 & ... & 0 & 1 & 4 \end{pmatrix}\begin{pmatrix} a_1\\ a_2\\ a_3\\ ..\\ a_{n-2}\\ a_{n-1}\end{pmatrix}=\frac{6}{h^2}\begin{pmatrix} y_0-2y_1+y_2\\ y_1-2y_2+y_3\\ y_2-2y_3+y_4\\ ....\\ y_{n-3}-2y_{n-2}+y_{n-1}\\ y_{n-2}-2y_{n-1}+y_n\end{pmatrix}$$

The general formula for the cubic polynomial between x_i and x_{i+1} is given by

$$\begin{aligned} p_i(x)=&\frac{a_i}{6}\frac{(x-x_{i+1})^3}{(x_i-x_{i+1})}+\left[\frac{y_i}{x_i-x_{i+1}}-\frac{a_i}{6}(x_i-x_{i+1})\right](x-x_{i+1})\\ &+\frac{a_{i+1}}{6}\frac{(x-x_i)^3}{(x_{i+1}-x_i)}+\left[\frac{y_{i+1}}{x_{i+1}-x_i}-\frac{a_{i+1}}{6}(x_{i+1}-x_i)\right](x-x_i)\end{aligned}$$

Problem 4 -*Suppose*

$$s(x)=\begin{cases} x^3+ax^2-4x+c, & 0\le x\le 2\\ -x^3+9x^2+bx+34, & 2\le x\le 4\end{cases}$$

Find constants a,b,c such the $s(x)$ is twice continuously differentiable on the interval $[0,4]$.

Problem 5 *Suppose*

$$s(x) = \begin{cases} ax^3 + x & -2 \le x \le 0 \\ x^3 + bx, & 0 \le x \le 2 \end{cases}$$

Find constants a, b such the $s(x)$ is twice continuously differentiable on the interval $[-2, 2]$.

Problem 6 *Suppose*

$$s(x) = \begin{cases} 0 & x \le 2 \\ (x-2)^3 & 2 < x \end{cases}$$

Is $s(x)$ a cubic spline? Justify your answer.

Problem 7 *Using cubic spline interpolation technique, find* $y(x = 0.6)$ *from the following data.*

x	*0.1*	*0.2*	*0.4*	*0.7*	*1.1*
y	*0.5754*	*0.6796*	*0.8026*	*0.9179*	*1.0231*

Ans: $y(0.6) = 0.8846$

4.1.3 Zeros of polynomials

Polynomials are used in physics to describe the trajectory of projectiles. Polynomial integrals can be used to express energy, inertia and voltage difference. in quantum mechanics, orthogonal polynomials appear as energy and momentum eigenfunctions. The zero or root of these polynomials gives positions and instants of zero probability for a physical system. They are also used for interpolation of experimental data.

Definition 3 *If $p(x) = \sum_{i=0}^{n} a_i x^i$ is a polynomial of degree n and $p(b) = 0$, then b is called a zero or root of the polynomial $p(x)$.*

There is an important theorem that relates the factors and zeros (roots) of a polynomial. There are 5 theorems about roots of Polynomials. They are n-Zero theorem, Remainder theorem, Factor theorem, Rational Root Theorem, Irrational Root Theorem, Complex Root Theorem, Descartes Rule.

Theorem 1 (n-Zero's Theorem) *If $p(x)$ is of Degree n, then it has at most n zeros.*

Theorem 2 (The Remainder Theorem) *If $\dfrac{p(x)}{x-b} = q(x)$and r, the remainder, then $p(b)$*
r

Proof:

$$p(x) = q(x)(x-b) + r,\ p(b) = r$$

To verify remainder theorem: If $p(x) = x^3 - x^2 - 17x - 16$. Let us divide $p(x)$ by $(x-5)$. The quotient is $q(x) = x^2 + 4x + 3$ and the remainder is $r = -1$. Now $p(5) = 5^3 - 5^2 - 17 \times 5 - 16 = -1 = r$. Hence verified.

If remainder $r = 0$ wehave the factor theorem.

Theorem 3 (Factor Theorem) *If $p(x) = \sum_{i=0}^{n} a_i x^i$ is a polynomial of degree n and $(x-b)$ is a factor of the polynomial $p(x)$, then b is a zero of $p(x)$.*

Proof:

$$p(x) = q(x)(x-b),\ p(b) = 0$$

To verify factor theorem: If $p(x) = x^3 + (a-1)x^2 - (a+6)x - 6a$. Factoring the polynomial we get $p(x) = (x+a)(x+2)(x-3)$. Hence $x = (-a, -2, +3)$ are the set of roots. $p(-a) = a^3 + (a-1)a^2 - (a+6)a - 6a = 0$. Similarly $p(-2) = p(3) = 0$

Theorem 4 (Bolzano's Theorem) *If $p(x)$ is a continuous function in the interval $x \in (a,b)$ and $p(a)p(b) < 0$, then there exists at least one $x = c \in (a,b)$ such that $p(c) = 0$.*

This is a special case of Intermediate Value Theorem. To verify Bolzano's theorem: If $p(x) = 6x^3 - 20x^2 - 14x + 60$. $(x-2)$ is a zero of this function.

$$p(1.5) = +14.25,\ p(2.5) = -6.25,\ p(1.5) \times p(2.5) = -89.0625 < 0$$

The root 2 lies between 1.5 and 2.5

Theorem 5 (Rational Roots Theorem) *Rational Roots of $p(x) = \sum_{i=0}^{n} a_i x^i$ will be of the form of (factors of a_0 divided by factors of a_n.*

A rational root is one which is expressible as a quotient of two integers. In the polynomial $p(x) = 6x^3 - 13x^2 + x + 2$, $a_0 = 2$, $a_n = a_3 = 6$. Hence the possible rational roots are $factors\,of\,a_0/factors\,of\,a_3 = factors\,of\,2/factors\,of\,6$. That is

$$\frac{\pm 1, \pm 2}{\pm 1, \pm 2, \pm 3, \pm 6} = \pm 1, \pm 1/2, \pm 1/3, \pm 1/6, \pm 2, \pm 2/3$$

The theorem states that if there is a rational root , it must be one of these. In fact, the roots are $(2, 1/2, -1/3)$ which belong to this set.

Theorem 6 (Irrational Root Theorem) *If $p(x)$ is a polynomial with rational coefficients and $a+\sqrt{b}$ where a and b are rational and $\sqrt{b}$ is irrational is a root, then the conjugate $a-\sqrt{b}$ is also a root.*

For example, if $p(x) = 2x^3 - x^2 - 9x - 4$, its roots are $x_0 = 1/2, x_\pm = (1 \pm \sqrt{17})/2$. Irrational roots occur in pairs.

Theorem 7 (Complex Root Theorem) *If $p(x)$ is a polynomial function with real coefficients and $a+ib$ is a root, then $a-ib$ must also be a root of $p(x)$.*

This theorem states that if the Coefficients are real, all Complex Roots occur in Conjugate Pairs. For example, the roots of $p(x) = 3x^3+7x^2+11x+3$ are $x_\pm = -1 \pm i\sqrt{2}$, $x_0 = 1/3$
An obvious corollary is <u>for an odd-degree polynomial there exist at least one real root.</u>

A method of determining the maximum number of positive and negative real roots of a polynomial is given by **Descartes Rule**.
For positive roots, start with the sign of the coefficient of the lowest (or highest) power of x. Count the number of sign changes n as you proceed from the lowest to the highest power (ignoring powers which do not appear). Then n is the maximum number of positive roots. Consider $p(x) = x^7 + x^6 - x^4 - x^3 - x^2 + x - 1$. Since there are three sign changes, there are a maximum of three possible real positive roots. Actually, there is only one real positive root $x = 1.1147$ and three pairs of complex roots $-1.2 \pm 0.6i, -0.3 \pm i, 0.4 \pm 0.5i$ for this polynomial.

Theorem 8 *Every polynomial p of degree n with complex coefficients a_i and $a_n \neq 0$ can be represented as*

$$p(x) = \prod_{i=1}^{n} a_i(x - b_i)$$

where b_i are the roots of p .

Sum and product of roots of polynomials Any polynomial $p(x) = \sum_{i=0}^{n} a_i x^i$ of degree n has n roots b_i, $i = 1, 2, 3..n$. If a_i are real, the sum and product of roots are given by

$$\sum_{i=1}^{n} b_i = -\frac{a_{n-1}}{a_n}, \ \prod_{i=1}^{n} b_i = (-1)^n \frac{a_0}{a_n}$$

Numerical procedure: Deflation

Consider a polynomials of degree $n > 2$. Any one root b_1 is found by any one of the methods - bisection, Newton-Raphson, secant, Laguerre's etc. The polynomial can be written as a product $p(x) = (x - b_1)q_1(x)$ where $q_1(x)$ is a reduced or deflated polynomial of degree $n - 1$. Also the roots of q_1 are exactly the remaining roots of p. Any one root of q_1 is then determined as b_2. Deflating q_1, $q_1(x) = (x - b_2)q_2(x)$. The same procedure may be repeated till the degree of q_i is 2. Then quadratic formula gives the last two roots. This method of successive deflation has the following advantages.

1. Deflation is just polynomial division.
2. The effort of finding a root generally decreases in each step.
3. The method cannot converge twice to the same non-multiple root.
4. Successive Deflation is numerically stable, if the root of smallest absolute value is divided out in each step.

Laguerre's Method of finding polynomial roots: Polynomial roots can be rational, irrational, real or complex, this method will converge to all types of roots.

The basic principle of this iterative method is given below.
Let $p(x)$ be a n^{th} degree polynomial.

$$p(x) = a_0 + a_1x + a_2x^2 + ... + a_nx^n = \sum_{i=0}^{n} a_ix^i$$

If the n roots are b_i, $i = 1, 2, 3, ...n$.

$$p(x) = (x - b_1)(x - b_2).....(x - b_n) = \prod_{i=1}^{n}(x - b_i)$$

Taking logarithm of modulus on both sides

$$\ln|p(x)| = \ln|(x - b_1)| + \ln|(x - b_2)|..... + \ln|(x - b_n)| = \sum_{i=1}^{n} \ln|(x - b_i)|$$

The modulus is taken since logarithm of negative real numbers and complex numbers are not defined. Differentiating with respect to x

$$\frac{d\ln|p(x)|}{dx} = +\frac{1}{(x - b_1)} + \frac{1}{(x - b_2)}..... + \frac{1}{(x - b_n)}$$

$$\frac{p'(x)}{p(x)} = \sum_{i=1}^{n} \frac{1}{(x-b_i)} = c(say)$$

Differentiating again

$$\frac{d^2 \ln|p(x)|}{dx^2} = -\frac{1}{(x-b_1)^2} - \frac{1}{(x-b_2)^2} \ldots\ldots - \frac{1}{(x-b_n)^2}$$

$$\frac{p''(x)p(x) - p'(x)p'(x)}{p(x).p(x)} = \frac{p''(x)}{p(x)} - \left[\frac{p'(x)}{p(x)}\right]^2 = -\sum_{i=1}^{n} \frac{1}{(x-b_i)^2}$$

$$\left[\frac{p'(x)}{p(x)}\right]^2 - \frac{p''(x)}{p(x)} = \sum_{i=1}^{n} \frac{1}{(x-b_i)^2} = d(say)$$

Let $x_j - b_j = e_j$ where x_j is the j^{th} trial root and b_j, the actual root. To find e_j, we assume that all other roots b_i are equidistant from x_j. Let $x_j - b_i = s$, $i = 1, 2, .., n$ and $i \neq j$. Then

$$c = +\frac{1}{s} + \frac{1}{s} \ldots\ldots + \frac{1}{e_j} + \ldots\ldots + \frac{1}{s} = \frac{1}{e_j} + \frac{n-1}{s}$$

Similarly

$$d = \frac{1}{e_j^2} + \frac{n-1}{s^2}$$

Eliminating s and solving for e_j

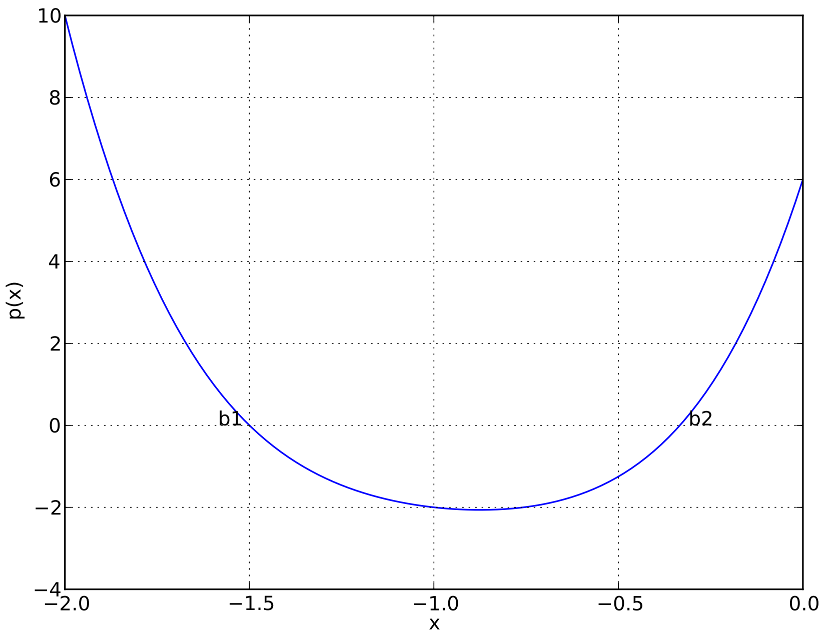
$$e_j = \frac{n}{c \pm \sqrt{(n-1)(n\,d - c^2)}}$$

For $|e_j|$ to be small, the modulus of denominator $|c \pm \sqrt{(n-1)(n\,d - c^2)}|$ must be large. If $c < 0$, denominator is $c - \sqrt{(n-1)(n\,d - c^2)}$. If $c > 0$, the denominator must be $c + \sqrt{(n-1)(n\,d - c^2)}$. To reduce e_j, the process is repeated with $x_j \rightarrow x_j - e_j$ till reasonable accuracy is obtained.

The polynomial is then divided by $(x - b_j)$ to get a lower degree polynomial $q(x)$. The root of $q(x)$ is found using the same method. This procedure is repeated, till all the roots are obtained.

Example-1 : To find the roots of the polynomial $p(x) = 6x^4 + 23x^3 + 37x^2 + 28x + 6$ we find that all coefficients are real and positive and rational. Hence complex and irrational roots occur in pairs. No change of sign means no real positive root. $p'(x) = 24x^3 + 69x^2 + 74x + 28$ and $p''(x) = 72x^2 + 138x + 74$. Let us assume that $x_j = -1.0$

in Legurre's procedure



```
Trial value: x=-1.0000
(1),p(-1.0000)=-2.0000,p'(-1.0000)=-1.0000,
   p''(-1.0000)= 8.0000,c=0.5000,d=4.2500, x=-1.5271,
(2)p(-1.5271)=0.2480,p'(-1.5271)=-9.5650,
   p''(-1.5271)=31.1665,c=-38.5659,d=1361.6680, x=-1.5000
(3) p(-1.5000)=-0.0000 .Hence it is one root.
```

Hence $x = -1.5$ is a root. $(x + 1.5) = (2x + 3)/2$ is a factor of given polynomial.

$$q(x) = \frac{p(x)}{2x + 3} = 3x^3 + 7x^2 + 8x + 2$$

```
Trial value: x=0.0000
(1) q(0)=2.0000  q'(0)=8.0000
 q''(0)=14.0000  c=4.0000  d=9.0000, x=-0.3405
(2)q(0.3405)=-0.0310  q'(0.3405)=4.2761
   q''(0.3405)=7.8702  c=-137.7945  d=19240.9417,  x=-0.3333
(3)q(0.3333)=0.0000 Hence it is another root.
```

Hence $x = -0.333$ is a root. $(x + 0.3333) = (x + 1/3) = (3x + 1)/3$ is a factor of given polynomial.

$$q1(x) = \frac{q(x)}{3x + 1} = x^2 + 2x + 2$$

This quadratic equation has roots

$$x_{\pm} = \frac{-2 \pm \sqrt{2^2 - 4.1.2}}{2} = -1 \pm i$$

Hence the set of roots are $(x = (1/3, 3/2, 1+i, 1-i)$

Example-2 : To find the roots of the polynomial $p(x) = x^4 + 1x^3 + 6x^2 + 4x + 16$ we find that all coefficients are real and positive. Hence complex and irrational roots occur in pairs. No change of sign means no real positive root. $p'(x) = 4x^3 + 3x^2 + 12x + 4$ and $p''(x) = 12x^2 + 6x + 12$. We start with a small trial root $x_j = -1.0$. The iterations give following values. $(1)x_j = (-0.8 - 1.2499j)$, $(2)x_j = (-0.9851 - 1.7613j)$, $(3)x_j = (-1.0000 - 1.7320j) = -1 - \sqrt{3}j$ Since $-1 - \sqrt{3}j$ is a root its complex conjugate $-1 + \sqrt{3}j$ is also a root. Hence by factor theorem

$$p(x) = q(x)[x + 1 + \sqrt{3}j][x + 1 + \sqrt{3}j] = q(x)(x+1)^2 + 3 = q(x)(x^2 + 2x + 4)$$

$$q(x) = \frac{x^4 + 1x^3 + 6x^2 + 4x + 16}{x^2 + 2x + 4} = x^2 - x + 4$$

The roots of this quadratic equation are given by $x_{\pm} = 0.5 \pm 1.9365j$. Hence the set of roots are $(-1 \pm \sqrt{3}j, 0.5 \pm 1.9365j)$

Problem 8 *Gas tank that is 10 meters in length (end to end) consists of a right-cylinder and is capped at either end by a hemisphere. What is the radius of the tank if the volume is 50 cubic meters?*

Answer: $Volume = 2\pi r^3/3 + \pi r^2(L - 2r) + 2\pi r^3/3$ simplify to the form $\pi r^3 - 15\pi r^2 + 75 = 0$ The three roots are $r = 14.89235732, 1.32108215, -1.21343947$. The first is not possible as the total length is only $10m$. Third value is unacceptable as radius cannot be negative. So $r = 1.32108215$. Also $2\pi r^3/3 + \pi r^2(L - 2r) + 2\pi r^3/3 = 50.00104m^3$

Example-3 : To find the roots of a cubic polynomial $p(x) = x^3 + x - 10$, we observe that there is one sign change so that there may be a positive root. Since coefficients are real, complex and/or irrational roots occur in pairs. Since there are only 3 roots one of them must be real. We start with a trial $x_j = 0$ in Legurre's procedure. The two iterations give (1) $x_j = 1.4293$ and (2) $x_j = 2.0068$ so that $p(x) = 0.0$ Hence $(x - 2)$ is a factor. Dividing with this factor

$$q(x) = \frac{x^3 + x - 10}{x - 2} = x^2 + 2x + 5$$

Using quadratic formula

$$x_{\pm} = \frac{-2 \pm \sqrt{4-20}}{2} = -1 \pm 2j$$

Hence the roots are $(2, -1 \pm 2j)$

Example-4 : To find the roots of a cubic polynomial $p(x) = 6x^3 - 11x^2 - 14x + 24$, we observe that there are 2 sign changes so that there may be 2 positive roots. As before we start with $x_j = 0$ in Legurre's procedure. Successive iterations give $x_j =$ 0.9101, 1.3001, 1.3333(= 4/3) Hence $x - 4/3$ or $(3x - 4)$ is a factor.

$$q(x) = \frac{6x^3 - 11x^2 - 14x + 24}{3x - 4} = 2x^2 - x - 6 = 2x^2 - 4x + 3x - 6 = 2x(x-2) + 3(x-2) = (x-2)(2x+3)$$

The roots are $x = 2$ and $x = -3/2$. The set of 3 roots are $x = 2, 4/3, -3/2$.

Problem 9 *Find roots of following polynomials using Legurre's method.*

No.	polynomial $p(x)$	Answer$(j = \sqrt{-1})$
1	$x^4 - 10x^3 + 35x^2 - 50x + 24$	$1, 2, 3, 4$
2	$x^4 - 2x^3 + 2x - 1$	$1, -2, -3, 4$
3	$x^4 - 3x^3 - 3x^2 + 11x - 6$	$2, -1/3, 3/2, 2$
4	$x^4 - 4x^3 + 6x^2 - 4x + 1$	$1, 3, (2 \pm 1j)$
5	$x^4 + 4x^3 + 6x^2 + 4x + 5$	$2 \pm j, 0 \pm j$
6	$x^3 - 3x^2 + 3x - 1$	$(2 \pm 3j), 4$
7	$x^3 - 6x^2 + 11x - 6$	$2, 3, 4$
8	$x^3 - 4x^2 + 5x - 2$	$(1 \pm 5j), 3$
9	$x^3 - 8x^2 + 20x - 16$	$3, 3, 0.5$

4.1.4 Monte Carlo Methods

Monte Carlo method is an iterative computational method used to calculate multi-dimensional integrals and investigate the behaviour of physical systems using stochastic methods. It is used as a statistical tool in studying situations which are not amenable to compute using deterministic algorithms. With the development of powerful computers and efficient algorithms, the Monte Carlo method is found to be very useful in finding numerical solutions to quantitative problems which are nonlinear and involving uncertain parameters. In physics, Monte Carlo methods are used in nuclear physics(nuclear model), molecular dynamics, crystal physics, statistical physics(Ising model), X-ray Imaging, Electron Dynamics in Doped Semiconductors, Quantum chromodynamics etc. This method is based on the following concepts and principles.

Definition 4 (Random variable) *A random variable is an assignment of numbers to possible outcomes of random events.*

For example, consider tossing a pair of coins. The number of heads n showing when the coins land is a random variable. It can be assigned the number 0 to the outcome [Tail, Tail]. That is $n(T,T) = 0$. Similarly $n(T,H) = n(H,T) = 1$ and $n(H,H) = 2$.

Definition 5 (Expectation or Expected Value) *The expected value of a random variable is the long-term limiting average of its values in independent repeated events.*

The expected value of the random variable X is denoted E[X]. For example, while tossing a coin, the fraction of times the coin lands with head up is half when the number of tosses are very large. It is expressed as $E[n(H)] = 0.5$. This idea can be expressed mathematically as follows. Let a random variable X takes a values x_i with probability p_i. If the sum of products converges absolutely to some value Y, that is,

$$\lim_{n\to\infty} \sum_{i=1}^{n} p_i.x_i = Y$$

, then Y is the expectation of X. For a continuous random variable X, the expectation may be defined as

$$E[X] = \int xP(x)dx$$

where $P(x)$ is the probability of random variable X having a value x.

Taking the expected value is a linear operation: if X and Y are two random variables,$E[X+Y] = E[X] + E[Y]$, and for any constant a, $E[aX] = aE[X]$

Theorem 9 (Law of large numbers) *As the number of trials of a random process increases, the percentage difference between the expected values and actual values (mean measured values) of all random variables goes to zero.*

The mean of values of x obtained from n independent trials is given by

$$\langle x\rangle = \frac{1}{N}\sum_{i=1}^{N} x_i$$

The law of large numbers states that

$$\lim_{N\to\infty} E[x] - \langle x\rangle = 0$$

Simple integration

Basic Principle: Consider the definite integral $\int_a^b f(x)dx$ where $f(x)$ is differentiable in the interval (a, b). By definition,

$$\int_a^b f(x)dx = \lim_{N\to\infty} \sum_{i=0}^{N} f(x_i)\delta x_i$$

where $a \leq x_i \leq b$. If δx_i is a constant for all i, $\delta x_i = (b-a)/N$.

$$\int_a^b f(x)dx = \lim_{N\to\infty} \frac{b-a}{N} \sum_{i=0}^{N} f(x_i)$$

if x_i are chosen at random between a and b. But

$$\frac{1}{N}\sum_{i=0}^{N} f(x_i) = \langle f \rangle$$

$$\lim_{N\to\infty} \frac{1}{N}\sum_{i=0}^{N} f(x_i) = \lim_{N\to\infty} \langle f(x_i) \rangle = E[f]$$

where $E[f]$ is the expectation by law of large numbers. If $b-a) = L$,

$$\int_a^b f(x)dx = L\langle f \rangle$$

For surface and volume integrals, the above relation becomes

$$\int_s f(x)dx = S\langle f \rangle, \int_v f(x)dx = V\langle f \rangle$$

The error e in the above calculation is the difference between the expected value and the mean value which is given by their standard deviation. For a k-dimensional integral, it is given by

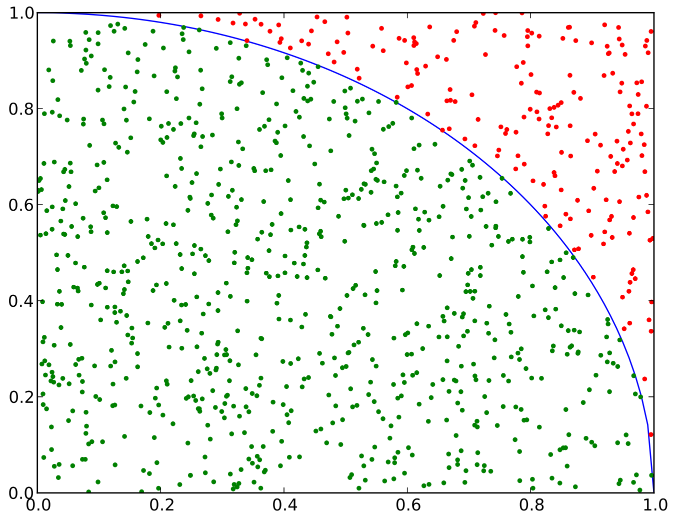
$$e = \pm\tau\sqrt{\frac{\langle f^2 \rangle - \langle f \rangle^2}{N}}$$

where τ is k-dimensional volume and $\langle f^2 \rangle = \sum_{i=0}^{N} f^2(x_i)$. It is only a rough estimate of the error.

Numerical procedure: In order to integrate a function over a complicated domain D, Monte Carlo integration uses random points over some simple domain D', which encloses D. Each random number generated is then checked to see whether it is within

D. Out of the n random numbers, if m are within D, then the ratio (m/n) gives the ratio of n-dimensional volumes D/D'. For example, to calculate the area D of one quadrant of a circle of radius r, we enclose it within a square (D') of side r. Pairs of random numbers (x_i, y_i) in the interval $(0, r)$ are generated and checked to see if $x^2 + y^2 \leq r^2$ so that it is within D. The ratio m/n of the number of random pairs to the total number of pairs gives the ratio of their areas D/D' . Then

$$D = D'(m/n)$$



.

For example, the integral $\int_0^{2\pi} \frac{d\theta}{2+\cos\theta}$ which is $2\pi/\sqrt{3} = 3.62759872847$, gives the following values.

No.of random points in $(0, 2\pi)$	Value of integral
10000	3.63293517
20000	3.63805025
30000	3.63569129
40000	3.63152102
50000	3.62770939

Integration by Importance Sampling

To reduce the variance(standard deviation) in the calculation of a definite integral using Monte Carlo method, sampling of the data points is introduced. Sampling is the process of selecting data from a domain of interest. By studying the sample, one can generalize the results back to the data set from which sample chosen. Sampling reduces volume of data to be processed, thereby reducing computing time.

Importance sampling: The term importance sampling refers to the process of classifying values of the input random variables in a simulation according to the impact on the quantity being estimated. If the large impact values (important values) are emphasized by sampling more frequently, then the variance of the estimated quantity can be reduced. Hence, the basic methodology in importance sampling is to choose a distribution which gives more weight to the important values. In a simulation, outputs are again weighted to correct for the use of the biased distribution. This ensures that the new importance sampling is unbiased. Thus the fundamental problem in importance sampling is to determine the distribution that is properly biased. Such a distribution saves large amounts of computing time.

Integration using importance sampling Consider the integral $I = \int f\, d\tau$ where f is a continuous function in some n-dimensional volume τ. Let $f(x) = g(x)h(x)$ where $g(x)$ is a positive function satisfying the condition $\int g(x)d\tau = 1$. Then the integral becomes $I = \int h(x)g(x)\, d\tau$. This can be interpreted as follows.

One can integrate f by sampling it with uniform probability density $d\tau$ in simple Monte Carlo method. But the same can be done by sampling $h(x)$ with non-uniform probability density $g(x)d\tau$. The generalized fundamental theorem is that the integral of any function f is estimated, using N sample points $x_i, ..., x_N$, by

$$\int_a^b f(x)d\tau = \tau\langle\frac{f}{g}\rangle$$

with an error estimate

$$error = \pm\tau\sqrt{\frac{\langle f^2/g^2\rangle - \langle f/g\rangle^2}{N}}$$

To choose the probability distribution $g(x)$, one proceeds by searching a function which minimises the absolute value of error (variational method). This may be taken as a constraint so that Legrange's multiplier method can also be used for finding $g(x)$.

4.1.5 Sampled Data

Sampled data refers to a subset of a large data system that is discrete or continuous and which has all the characteristics of the complete data system. The size of the sample required for this purpose is given by sampling theorem.

Theorem 10 (Nyquist-Shannon Sampling Theorem) *In order for a band-limited signal of maximum frequency ν_m to be reconstructed fully, it must be sampled at a rate $f \geq 2\nu_m$.*

Here band-limited signal refers to a signal with a zero power for frequencies $\nu > \nu_m$. A signal sampled at $f = 2\nu_m$ is said to be Nyquist sampled, and f_c is called the Nyquist critical frequency. No information is lost if a signal is sampled at f_c, and no additional information is gained by sampling faster than this rate. For example, to sample a sine wave of frequency ν, the minimum sampling rate is $f_c = 2\nu$. That is the time interval Δ between samples and period of sine wave T are related as $Delta = T/2$. Hence a convenient choice is to sample at positive and negative peak of the wave. If $h(t)$ contains frequencies f outside of the range $(-fc, fc)$ where f_c is the Nyquist frequency, then the power content of these frequencies is moved into that range $-f_c < f < f_c$ so that power spectrum is modified. This phenomenon is called *aliasing*. Any frequency component outside of the range $(-fc, fc)$ is translated into that range as a result of discrete sampling.

4.1.6 Discrete Fourier Transform

Consider a function $h(t)$ of a continuous variable t transformed into a function H of another independent variable ω having range $(-\infty, \infty)$ by the equation

$$H(\omega) = \int_{-\infty}^{\infty} h(t)e^{i\omega t}dt$$

where $\omega = 2\pi f$. $h(t)$ can be retrieved using the inverse transform

$$h(t) = \int_{-\infty}^{\infty} H(\omega)e^{-i\omega t}d\omega$$

The integration in these transforms can be replaced with a summation over the function values $h(t)$ corresponding to properly sampled values of t. Consider N consecutive sampled values at constant separation Δ. Let

$$t_k = k\Delta,\, h_k \equiv h(t_k),\, k = 0, 1, 2, ..., N-1$$

If the function $h(t)$ is continuous and not periodic, then we assume that the sampled points are such that $h(t)$ is similar in structure at all times t. Even though all frequencies in the Nyquist frequency range $(-f_c, f_c)$ are possible, we can get Fourier transforms $H(f_k)$ only for N-frequencies as there are only N-input samples. Consider N values

$$f_n = \frac{n}{N\Delta}, n = [N/2, (N/2)-1, (N/2)-2, ...(-N/2)+1, -N/2]$$

The extreme values of n, that is $\pm N/2$ exactly correspond to the lower and upper limits $\pm f_c$. If $\omega_n = 2\pi f_n$, then the discrete Fourier transform is given by

$$H(\omega_n) = \int_{-\infty}^{\infty} h(t)e^{i\omega_n t}dt \approx \sum_{k=0}^{N-!} h_k e^{(i\omega_n t_k)}\Delta = \Delta \sum_{k=0}^{N-!} h_k e^{(i\omega_n k\Delta)}$$

Substituting for $\omega_n\Delta = 2\pi f_n\Delta = 2\pi n/N$ in the exponent

$$H(\omega_n) = \Delta \sum_{k=0}^{N-!} h_k \exp\left(\frac{2\pi ikn}{N}\right)$$

The quantity

$$\sum_{k=0}^{N-!} h_k \exp\left(\frac{2\pi ikn}{N}\right) = H_n$$

where H_n is called the discrete Fourier transform of the N points h_k.

$$H(f_n) \approx H_n\Delta$$

The inverse discrete Fourier transform is then given by

$$h_k = \frac{1}{N} \sum_{n=0}^{N-!} H_n \exp\left(-\frac{2\pi ikn}{N}\right)$$

Algorithm For n data points

```
1: Read input list $x$ of length $n$
2: $\omega \leftarrow 2\pi/n$
3: for $p = 0$ to $n-1$ do
4:     $s \leftarrow 0$
5:     for $q = 0$ to $n-1$ do
6:         $s \leftarrow s + x_q e^{i\omega pq}$
7:     end for
8:     $y_p \leftarrow s$
9: end for
```

Program: The following function will calculate the discrete Fourier transform of list of values x.

```
#input x is list, y is output list
from cmath import*
def dft(x):
    n = len(x)
    omega, y = 2*pi/n,[0]*n
    for p in xrange(n):
        s = 0
        for q in xrange(n):
            s +=x[q]*exp(omega*q*p*1j)
        y[p] = s
    return y
print dft([1,3,5,3,1,-1,-3])
Output
[(9+0j), (-3.49+11.41j), (-0.11+1.681j), (2.60-0.14j),
 (2.60+0.14j), (-0.11-1.68j), (-3.49-11.41j)]
```

Thus the calculation of discrete Fourier transform of N-sampled points requires $N \times N = N^2$ computations. There is an algorithm to reduce the number of computations $N \log_2 N$ called Fast Fourier Transform.

4.1.7 Fast Fourier Transform(FFT)

This algorithm was first conceived by Gauss in the 18^{th} centuary. With the advent of modern computers, Tukey and Cooley developed an algorithm to implement it on a digital computer. There are other algorithms similar to this one developed recently.

The basic principle is the divide and conquer strategy just like any other large data systems. The discrete Fourier transform of n-sampled points requires $N \times N = N^2$ computations. If the data set is divided into two equal parts, each part requires $N^2/4$ computations for its Fourier transform. Hence total number of computations is only $N^2/2 = N^2/2^1$ which shows a reduction by $N^2/2$. If each half is still divided, the number of computations gets reduced to $4 \times (N/4)^2 = N^2/4 = N^2/2^2$ In general if the $N = 2^p$-points are divided into $M = 2^q$ equal parts, the number of computations becomes $N^2/2^M = 2^{2p-q}$.

Procedure: There are different algorithms for FFT. One of the simplest is the Sande-Tukey algorithm. It is given below.

Let $N = 2^p$ an integer power of 2. If the length of the data set is not a power of two, zeros may be added as data elements up to the next power of two. If f_n are n-data points separated by N equal intervals, then its discrete Fourier transform (DFT) is given by

$$F_k = \sum_{n=0}^{N-1} f_n W^{nk},\ k = 0, 1, 2, 3..N-1$$

where $W = e^{2\pi i/N}$. Let us divide the entire data into two sets-odd and even indices-each of length $N/2$ with

$$\begin{aligned}
F_k &= \sum_{n=0}^{(N/2)-1} f_{2n} W^{2nk} + \sum_{n=0}^{(N/2)-1} f_{2n+1} W^{(2n+1)k} \\
&= \sum_{n=0}^{(N/2)-1} W^{2nk} \left[f_{2n} + f_{2n+1} W^k \right] \\
&= F_k^{(e)} + W^k F_k^{(o)}
\end{aligned}$$

where $F_k^{(e)} = \sum_{n=0}^{N/2-1} W^{2nk} f_{2n}$ and $F_k^{(o)} = \sum_{n=0}^{N/2-1} W^{2nk} f_{2n+1}$, superscript 'o' and 'e' stands for odd and even number4ed data. The crux of the solution is to consider these odd and even sets of $N/2$ numbers as transforms of sequences of length $N/2$. This is called Danielson-Lanczos Lemma. It is found that this Lemma can be used recursively. Having reduced the problem of computing F_k to that of computing $F_k^{(e)}$ and $F_k^{(o)}$, the same reduction of $F_k^{(e)}$ to the problem of computing the transform of its $N/4$ even-numbered input data (even k in f_{2k}) as $F_k^{(ee)}$ and $N/4$ odd-numbered data (odd k in f_{2k}) $F_k^{(eo)}$. Similarly division of $F_k^{(o)}$ can also be done into $F_k^{(oe)}$ and $F_k^{(oo)}$. In other words, one can define discrete Fourier transforms of the points which are respectively even-even, even-odd, odd-even and odd-odd on the successive subdivisions of the data. Since N is a power of 2, it is evident that one can continue applying the Danielson-Lanczos Lemma until we have subdivided the data all the way down to transforms of length 1. The Fourier transform of length one is just the identity operation that copies its one input number into its one output slot! In other words, for every pattern of $\log_2 N$ there is a one-point transform that is just one of the input numbers f_n for some n.

4.1.8 Shooting method

Shooting method is employed to solve ordinary second order differential equations with a pair of boundary conditions . It is a two-point boundary value problem. The boundary conditions at the starting point do not determine a unique solution to start with. Starting boundary conditions is almost certain not to satisfy the boundary conditions at the other boundary point. In general, iteration is required to correlate boundary conditions into a single global solution of the differential equation. Differential equations are to be integrated over the interval of interest several times. Only for linear differential equations, the number of iterations can be predicted. Consider the general form of a second order linear differential equation for the function $y(x)$

$$y''(x) = p(x)y'(x) + q(x)y(x) + r(x)$$

with boundary condition $y(a) = \alpha, y(b) = \beta$. Suppose $u(x)$ is a function which satisfies the above equation with initial conditions $u(a) = \alpha$, $u'(a) = 0$

$$u''(x) = p(x)u'(x) + q(x)u(x) + r(x)$$

Let $v(x)$ be a function which satisfies the equation

$$v''(x) = p(x)v'(x) + q(x)v(x),\ v(a) = 0, v'(a) = 1$$

Then the linear combination $y = u + cv$ is a solution.

$$\begin{aligned} y''(x) &= u''(x) + cv''(x) \\ &= p(x)u'(x) + q(x)u(x) + r(x) + c[p(x)v'(x) + q(x)v(x)] \\ &= p(x)[u'(x) + cv'(x)] + q(x)[u(x) + cv(x)] + r(x) \\ &= p(x)y'(x) + q(x)y + r(x) \end{aligned}$$

To evaluate c, we use the boundary condition at $x = b$

$$u(b) + cv(b) = \beta,\ c = \frac{\beta - u(b)}{v(b)}$$

Hence the solution is

$$y(x) = u(x) + \frac{\beta - u(b)}{v(b)}v(x)$$

Procedure: Any second order linear differential equation can be split into two coupled first order equations and solved by Runge-Kutta method if two initial conditions are known. In linear shooting method, the following procedure is followed.

1. First solve
$$u'(x) = s(x),\ u(a) = \alpha$$
$$s'(x) = p(x)s(x) + q(x)u(x) + r(x),\ s(a) = 0$$

2. Then solve
$$v'(x) = t(x),\ v(a) = 0$$
$$t'(x) = p(x)t(x) + q(x)v(x),\ t(a) = 1$$

3. Finally, the desired solution $y(x)$ is the linear combination
$$y(x) = u(x) + \frac{\beta - u(b)}{v(b)}v(x)$$

Eigenvalue problems: Consider boundary value problem

$$y''(x) = p(x)y'(x) + q(x)y(x) + r(x),\ y(a) = \alpha,\ y(b) = \beta$$

. If $q(x) = \lambda$ a constant and $r(x) = 0$, it becomes an eigenvalue problem in differential equations.

$$y''(x) - p(x)y'(x) = \lambda y(x),\ y(a) = \alpha,\ y(b) = \beta$$

. Here λ is an eigenvalue and $y(x)$ is an eigenfunction corresponding to λ. The method of solution involves the computation $y(b)$ for different λ and find the one for which $y(b) = \beta$. This can be done efficiently if we find the roots of the logarithmic derivative as it will cancel all multiplicative constants from y and y'.

$$\left[\frac{y'(x,\lambda}{y(x,\lambda)}\right]_{x=\beta} = 0$$

4.1.9 Relaxation method:

It is an approach different from shooting method. The differential equations are replaced by finite-difference equations between a set of points in the range of integration. The conversion of a differential operator to a difference operator is done as follows. Let y_0, y_1, y_2 be three points corresponding to the x-values $x_0, x_0+\delta x, x_0+2\delta x$ respectively

$$y'(x_0) = \left[\frac{dy}{dx}\right]_{x=0} \approx \frac{y_1 - y_0}{x_0 + \delta x - x_0} = \frac{y_1 - y_0}{\delta x}$$

$$y''(x_0) = \left[\frac{dy'}{dx}\right]_{x=0} \approx \frac{y_1' - y_0'}{\delta x} \approx \left[\frac{y_2 - y_1}{\delta x} - \frac{y_1 - y_0}{\delta x}\right]\frac{1}{\delta x} = \frac{y_2 - 2y_1 + y_0}{(\delta x)^2}$$

Procedure: The method of solution involves the following steps. Let $y(x)$ be the unknown function

1. The differential equation is converted into a difference equation.
2. A trial solution is assumed which consists of values for the dependent variables at each mesh point. It may not satisfy the desired finite-difference equation, nor the required boundary conditions.
3. It is then substituted in the difference equation and a solution is obtained.
4. This solution is substituted back to the difference equation and solution is again found.

5. This iterative process is continued till the solution is in close agreement with the true solution. This is indicated by the agreement with difference equation and boundary condition. Also further iteration will not change the solution significantly.

Relaxation method is preferred over shooting method in the following situations.

- If the boundary conditions are subtle, or involve complicated algebraic relations.
- If the solution is smooth and not highly oscillatory.
- If the differential equations have extraneous solutions which disappears during iteration. They will not appear in the final solution satisfying all boundary conditions.
- If a good initial guess is possible, relaxation methods are very efficient. Often it may be necessary to solve a problem many times, each time using a slightly different value of some parameter like an eigenvalue. In that case, the previous solution is usually a good initial guess when the parameter is changed.

Chapter 5

Simulations

5.1 A computational approach to physics

Simulation in physics refers to the imitation of the behaviour of physical systems with time (temporal evolution). Time evolution of a system follow deterministic laws. These laws are invariably differential equations. If the differential equations are non-linear or they are sets of coupled equations, analytic solutions are either too difficult or impossible without heavy approximations. Numerical solutions are the only alternative in those cases. To reduce the errors in the results, often one has to use a large number of time steps. A computer becomes an absolute necessity in such cases. By changing variables in the simulation, the behaviour of the system under different circumstances can be studied virtually. It thus leads to the concept of a theoretical lab.

Steps involved in Simulation

The number of steps involved depends generally on the complexity of the phenomena to be simulated. But the following 7 steps are mandatory.

1. Identify the property or phenomena of interest to be studied.
   Eg. Motion of masses under mutual attraction.

2. Choose the field of force which describes how the atoms or other particles within the system interact with each other and also with the external world.
   In the above case, it can be an inverse-square law force.

$$f = \frac{k}{r^2}$$

3. Create a set of variables that you may need to construct a simulation.
   In the above case, initial and current positions $(x_0, y_0), (x, y)$, initial and current velocities $(vx_0, vy_0), (vx, vy)$, acceleration $a = f/m$ and time step dt

4. Derive an equation relating different variables. In most of the cases of interest in physics, it may be a differential equation.
   In the above case, as acceleration is radial $a_x = a\cos\theta,\ a_y = a\sin\theta = ky/r^3$

$$a_x = \frac{d^2x}{dt^2} = k.x/r^3$$

$$a_y = \frac{d^2y}{dt^2} = k.y/r^3$$

5. Choose a suitable method of solution of the equation. For ordinary differential equations any of these methods- Runge-Kutta, predictor-corrector, Monte-Carlo, Euler etc.-may be used. In Euler method.

$$v_x = v_{x0} + a_x dt$$

$$v_y = v_{y0} + a_y dt$$

$$x = x_0 + v_x dt$$

$$y = y_0 + v_y dt$$

6. Solve the equation for different values of the variable starting from initial values and incrementing in steps of proper size.
   Calculate $[x(t + n\ dt), y(t + n\ dt)]$ from $[x\{t + (n-1)dt\}, y\{t + (n-1)dt\}]$.

7. Either print the output as a table of values or plot the output as a graph.

5.1.1 Simple harmonic oscillator

Principle

A simple harmonic oscillator is described by the equation

$$\frac{d^2x}{dt^2} + \omega^2 x = 0$$

This can be split into 3 equations using acceleration a, velocity v and displacement x as follows.

$$a = -\omega^2 x,\ \delta v = a\,\delta t,\ \delta x = v\,\delta t$$

As before, the last two relations can be expressed as

$$v_i = v_{i-1} + a\,\delta t$$

Similarly the displacement is given by

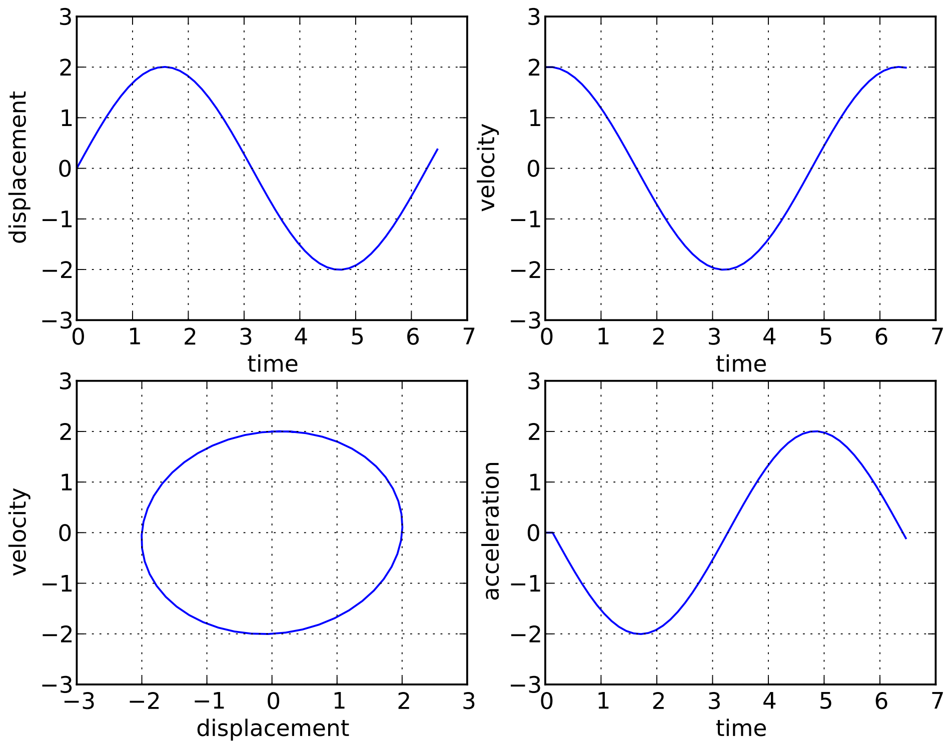
$$x_i = x_{i-1} + v_i \delta t$$

The $x - t$ graph and $x - v$ graph (Phase curve) are drawn for the oscillator.

Program

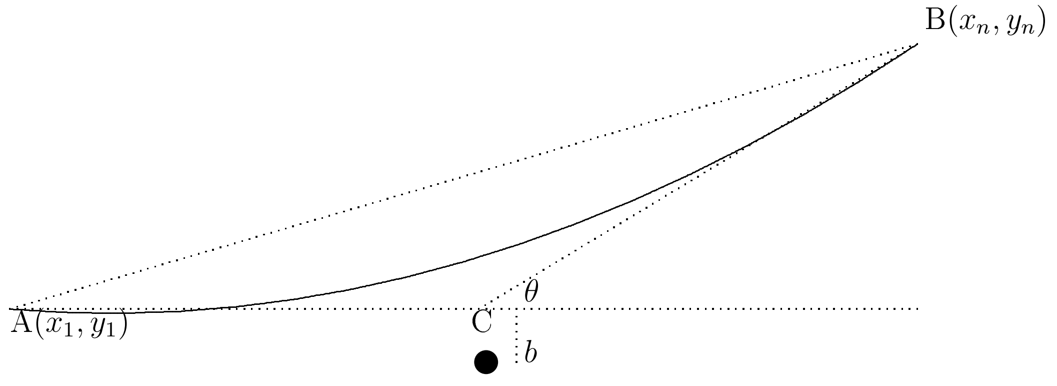
```
from pylab import*
n=50
x=zeros(n,dtype=float)
t=zeros(n,dtype=float)
v=zeros(n,dtype=float)
a=zeros(n,dtype=float)
x[0],v[0],omega=0,2,1 #input('initial displacement, velocity and angular frequency')
dt=2.1*pi/(n*omega)
a[0]=-x[0]*omega**2
for i in range(1,n):
    a[i]=-x[i-1]*omega**2
    v[i]=v[i-1]+a[i]*dt
    x[i]=x[i-1]+v[i]*dt
    t[i]=t[i-1]+dt
subplot(2,2,1)
xlabel("time")
ylabel("displacement")
grid(True)
plot(t,x)
subplot(2,2,2)
xlabel("time")
ylabel("velocity")
grid(True)
plot(t,v)
subplot(2,2,3)
xlabel("displacement")
ylabel("velocity")
grid(True)
plot(x,v)
subplot(2,2,4)
xlabel("time")
```

```
ylabel("acceleration")
grid(True)
plot(t,a)
show()
```



5.1.2 Central field motion

A force field having a potential function $V(r, \theta, \phi) = V(r)$ is called a central field. As it depends only on r it has spherical symmetry and consequently angular momentum is conserved. Gravitational field and Coulomb field are examples of central fields. The general form of the potential is $V(r) = kr^n$ where k is a constant and n a real number. Rutherford scattering is an example of motion in a repulsive central field.



Principle

Rutherford's experiment is to measure the deflection of a beam of $\alpha-$ particles by gold nuclii due to Coulomb repulsion. The electrostatic force is given by

$$\vec{F} = \frac{Ze.2e}{4\pi\epsilon_0 r^2}\hat{r}$$

Resolving and putting $r^2 = x^2 + y^2$, the components of acceleration are

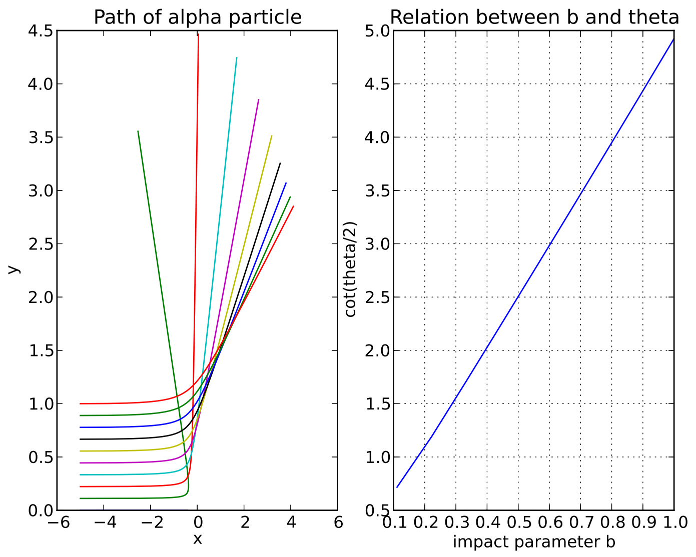
$$a_x = \frac{2ze^2x}{4m\pi\epsilon_0 r^3} \qquad (1)$$
$$a_y = \frac{2ze^2y}{4m\pi\epsilon_0 r^3}$$

Putting $c = \frac{2ze^2}{4m\pi\epsilon_0}$, $a_x = \frac{d^2x}{dt^2}$ and $a_y = \frac{d^2y}{dt^2}$ one gets

$$\frac{d^2x}{dt^2} = \frac{cx}{(x^2+y^2)^{3/2}} \qquad (2)$$
$$\frac{d^2y}{dt^2} = \frac{cy}{(x^2+y^2)^{3/2}}$$

The velocity and position of every $\alpha-$particle at different instants of time are then determined by solving the above differential equations numerically. From figure, if (x_1, y_1) are asymptotic points , $\triangle ABC$ is isosceles. If θ is the angle of deflection,

$$\cot(\theta/2) = \frac{x_n - x_0}{y_n - y_0}$$



Algorithm

```
1:  Read the initial values of velocities and positions of particles in the beam, the
    impact parameter and the time step $dt$.
2:  for $b$=0 to 20 step 1 do
3:      while $\|\mathbf{y}\|$ is below a fixed value $y_n$ do
4:          Calculate $a_x$ and $a_y$ using formulae 3 and 5.1.2
5:          Calculate $x$ and $y$ using $4^{th}$-order Runge-Kutta Method.
6:          plot $(x, y)$
7:      end while
8:      Calculate $\cot(\theta/2)$ and $b/\cot(\theta/2)$
9:      plot $\cot(\theta/2)$ against $b$
10: end for
11: End
```

Program

```
from pylab import*
c=21.82743562;
dt=0.0001
x=zeros(10001,'float')
y=zeros(10001,'float')
b=linspace(0,1,10)
cotthetaby2=zeros(10,'float')
```

```
for j in range(10):
    x[0],y[0],t,vx,vy=-5,b[j],0,10,0
    for i in range(10000):
        vx+=x[i]*c*dt/(x[i]*x[i]+y[i]*y[i])**1.5
        vy+=y[i]*c*dt/(x[i]*x[i]+y[i]*y[i])**1.5
        x[i+1]=x[i]+vx*dt
        y[i+1]=y[i]+vy*dt
    subplot(1,2,1)
    xlabel('x')
    ylabel('y')
    title("Path of alpha particle")
    plot(x,y)
    cotthetaby2[j]=(x[i]-x[0])/(y[i]-y[0])
subplot(1,2,2)
xlabel('impact parameter b')
ylabel('cot(theta/2)')
title("Relation between b and theta")
grid(True)
plot(b,cotthetaby2)
show()
```

5.1.3 Monte-Carlo simulations- value of π

Principle

Points in the first quadrant of a unit circle centered at origin satisfies the following inequalities.

1. $0 \leq x \leq +1$
2. $0 \leq y \leq +1$
3. $x^2 + y^2 \leq 1$.

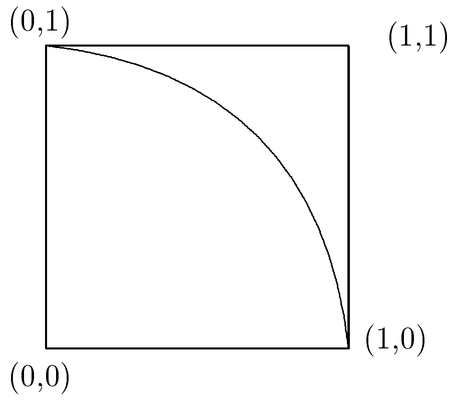
.The first two conditions are satisfied by all points in a unit square in the first quadrant as shown in figure. The ratio of area of the sector$(= \pi/4)$ to the area of the square$(= 1^2)$ is $\pi/4$. Four times this ratio gives the value of π.

To determine this ratio in Monte Carlo Method, pairs of pseudo-random numbers in the range $(0, 1)$ are generated for coordinates (x, y). All these pairs fall within the square, but only those points belong to the circle for which $\sqrt{x^2 + y^2} \leq 1$. The number of such points N are counted. The ratio of number N to the total number of points gives the ratio of their areas.

In PYTHON there is one in-built random number generator using the multiplicative congruential recursive method developed by D.Lehmer. The i^{th} and $(i+1)_{th}$ random numbers are related as

$$x_{i+1} = (ax_i + c)\text{mod } m$$

where the multiplier $a = 7^5 - 1 = 16,806$, the increment $c = 0$ and the modulus $m = 2^{31} - 1 = 2,14,74,83,647$ are called magic numbers. The recurrence relation suggests that the random numbers will repeat with a period less than m. The magic numbers for the set (a, m, c) are chosen so that period is $\approx m$ and every number between 0 and $m-1$ occur at some point. The role of initial seed x_0 has only little effect.



Algorithm

```
1: Read $N$,the count of random numbers to be generated
2: $j \leftarrow 0$
3: for $i = 1$ to $N$ do
4:     $x \leftarrow rand()$
5:     $y \leftarrow rand()$
6:     if $(x^2 + y^2) \leq 1$ then
7:         $j \leftarrow j + 1$
8:     end if
9: end for
10: $\pi \leftarrow 4.j/i$
11: Print $\pi$
12: End
```

Program

```
from random import random
j=0
for i in range(1000000):
    if (random()**2+random()**2)<=1:j+=1
print "Value of pi = ",4.0*j/i
```

5.1.4 Logistic map

Principle

A map is a function which relates the coordinates of a point P_{n+1} in terms of those of the previous point P_n. A map is always discrete as it uses the previous value of the dependent variable as the present value of independent variable. There is thus no question of differentiability for a map.

The logistic map is developed by Robert May in 1876 as a mathematical model of population growth whose generations do not overlap with a fixed environment. It is given by

$$x_{n+1} = cx_n(1 - x_n)$$

This is called logistic map. where $0 < x < 1$ and $c > 1$. A continuous form of this map is the logistic equation $f(x) = cx(1 - x)$

Characteristics of logistic equation and map:

1. The roots of logistic equation are obtained by setting $f(x) = 0$. They are x=0 and x=1.

2. $$\frac{df(x)}{dx} = c(1 - 2x)$$

   Extremum occurs at df(x)/dx=0 which is, at x = 1/2. This is a maximum because $d^2f(x)/dx^2 = -c$ which is negative. This point x=1/2 is called the critical point of the function possessing only a single maximum in a given intervel $(\ 0 < x < 1)$ in this case.

3. After some iterations of the map , it often converges to some fixed value called an 'attractor'. Any further iteration of will yield the same value. If x_n^* is such a value,

$$\begin{aligned} x_n^* &= c x_n^*(1 - x_n^*) \qquad (3) \\ x_n^* &= 1 - 1/c \end{aligned}$$

Condition for stability of attractor

In the map if $x_n < 0$, then iterations will lead $x_{n+1} to - \infty$. If $x_n = 0$ then x_{n+1} is zero always. For $x = 1/c, x_{n+1} = 1 - 1/c = x_n^*$. The range $0 < x_n \leq 1/c$ is called the 'basin of attraction' of x_n^*. A value x_n approaches x_n^* if successive iterations bring it closer to x_n^* .

$$\begin{aligned} \left| \frac{x_{n+1} - x_n^*}{x_n - x_n^*} \right| &< 1 \\ \left| \frac{f(x_n) - x_n^*}{f(x_{n-1}) - x_n^*} \right| &< 1 \end{aligned}$$

In the limit $f(x_{n-1}) - x_n^* \to 0$

$$\begin{aligned} \left| \frac{df(x_n)}{dx_n} \right|_{x_n = x_n^*} &< 1 \\ |c(1 - 2x_n^*)| = |2 - c| &< 1 \end{aligned}$$

This is possible only if $1 < c < 3$.

When $c = 3$ the attractor bifurcates to two fixed points x_1^* and x_2^* in such a way that

$$\begin{aligned} x_2^* &= f(x_1^*) \\ x_1^* &= f(x_2^*) \\ x_2^* &= f[f(x_2^*)] \\ &= c^2 x_2^*(1 - x_2^*)[1 - c x_2^*(1 - x_2^*)] \end{aligned}$$

Each x_2 is said to be a fixed point of period 2. In general, if x_p is a fixed point of period p , x_p repeats after a set of p iterations of f. That is

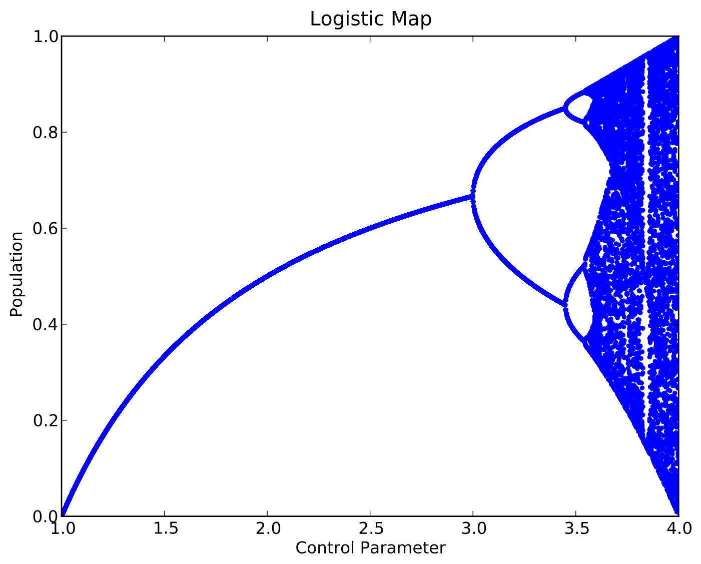
$$f^{(p)}(x_p) = f(f(f ... p - times ... (x_p) = x_p$$

This bifurcation of the attractor at $c = 3$ is called pitchfork bifurcation due to its shape.

$$|f(f(x_n))| \leq 1$$

This requires $c \geq 1 + \sqrt{6} = 3.449489743$. Then each branch of fixed points bifurcates into two separate branches. The points on these branches will be of period 4.

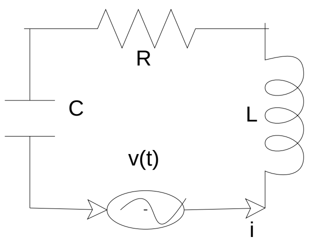
If c is further increased, further branching occurs. Fixed points of period p give rise to 2^p branches. It is found that for $c = 3.5699.....$, an infinite number of bifurcations occur. In logistic map, fixed points never repeat. The band of fixed points forms a continuum. Complete chaos begins from this point. Thus bifurcation is the route to chaos for logistic equation.



Program

```
from pylab import*
def f(c, x):  return c * x * (1 - x)
ci,cf,x0,n,g =2.9,3.655, 0.1,50,1000
ci,cf=input("range of control parameter = ")
cs=(cf-ci)/1000.0
Lc,Lx = [],[]
for c in arange(ci, cf, cs):
        x = x0
        for i in range(g): x = f(c, x)
        p = 0
        while p < n:
            x = f(c, x)
            Lc.append(c)
            Lx.append(x)
            p += 1
plot(Lc, Lx, ".")
xlabel("Control Parameter")
ylabel("Population")
show()
```

5.1.5 Driven LCR circuit



Principle:

If a voltage $v(t)$ is given to a series LCR- circuit, Kirchoff's voltage law gives

$$L\frac{di}{dt} + Ri + \frac{q}{C} = V(t)$$

Since current is rate of flow of charge

$$L\frac{d^2q}{dt^2} + R\frac{dq}{dt} + \frac{q}{C} - V(t) = 0$$

This second order differential equation can be split into two first order coupled equations and solved simultaneously.

$$\frac{dq(t)}{dt} = i(t),\ \frac{di(t)}{dt} = -\frac{R}{L}i(t) - \frac{q}{LC} + \frac{V(t)}{L}$$

These coupled equations can be solved numerically using Runge-Kutta fourth order method. The general formula for coupled differential equations is given below.

Let

$$\frac{dy}{dx} = f(x,y,z),\ \frac{dz}{dx} = g(x,y,z)$$

be two coupled equations with initial conditions $y(x0) = y0$, $z(x0) = z0$. Then the values $y(x + \delta x)$, $z(x + \delta x)$ are given by a 4-step determination of slopes at $x0, x0 + \delta x/2, x0+\delta x/2, x0+\delta x$ successively as follows. let $\delta x = h$ which is the usual convention.

```
k1=h f(x0,y0,z0)
m1=h g(x0,y0,z0)
k2=h f(x0+h/2,y0+k1/2,z0+m1/2)
m2=h g(x0+h/2,y0+k1/2,z0+m1/2)
k3=h f(x0+h/2,y0+k2/2,z0+m2/2)
m3=h g(x0+h/2,y0+k2/2,z0+m2/2)
k4=h f(x0+h,y0+k3,z0+m3)
m4=h g(x0+h,y0+k3,z0+m3)
y(x0+h)=y0+(k1+2 k2+2 k3+k4)/6
z(x0+h)=z0+(m1+2 m2+2 m3+m4)/6
```

Using the pair $(y(x0 + h), z(x0 + h)$ the values at $x0 + 2h$ $(y(x0 + 2h), z(x0 + 2h)$ are found using the above formula. The process is repeated till we get the value of (y, z) at the desired x.

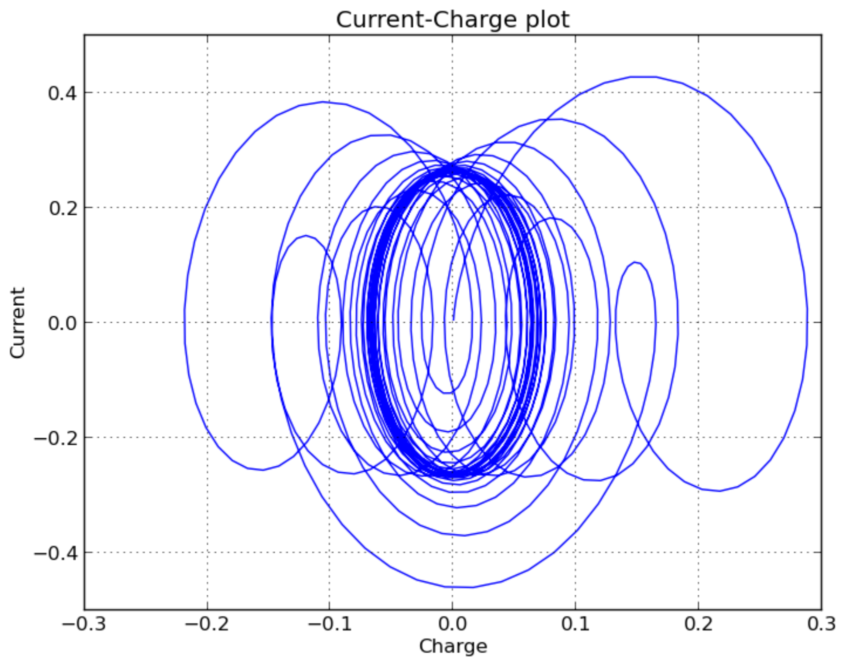
Program:

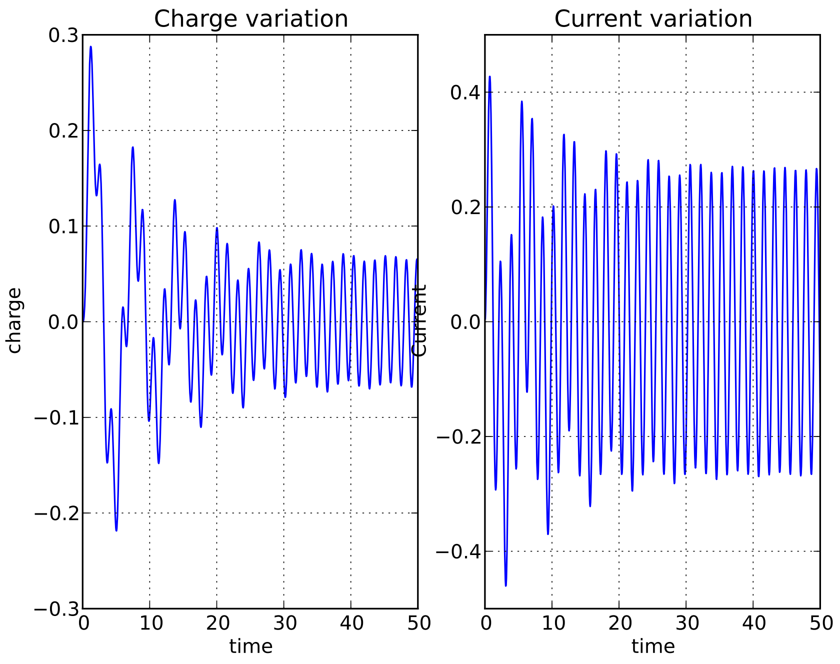
```
#x=time, y=charge, z=current
#The following function solves the
#coupled equations y'=f(x,y,z) and z'=g(x,y,z)
from pylab import*
```

```
def f(x,y,z):return z
def g(x,y,z):return (-0.2*z-y+sin(4*x))#L=1H,C=1F, omega=4/s,R=0.2ohm.

#Runge-Kutta fourth order function
def rk4solution(f,g,x,y,z,h,n):
  result=[[],[],[]]
  for i in range(n):
    k1=h*f(x,y,z)
    m1=h*g(x,y,z)
    k2=h*f(x+h/2,y+k1/2,z+m1/2)
    m2=h*g(x+h/2,y+k1/2,z+m1/2)
    k3=h*f(x+h/2,y+k2/2,z+m2/2)
    m3=h*g(x+h/2,y+k2/2,z+m2/2)
    k4=h*f(x+h,y+k3,z+m3)
    m4=h*g(x+h,y+k3,z+m3)
    x,y,z=x+h,y+(k1+2*k2+2*k3+k4)/6,z+(m1+2*m2+2*m3+m4)/6
    result[0].append(x)
    result[1].append(y)
    result[2].append(z)
  return result

s=rk4solution(f,g,0.,0.,0.,0.05,1000)
figure(1)
subplot(1,2,1)
xlabel('time')
ylabel('charge')
title('Charge variation')
grid(True)
plot(s[0],s[1])
subplot(1,2,2)
xlabel('time')
ylabel('Current')
title('Current variation')
grid(True)
plot(s[0],s[2])
figure(2)
xlabel('Charge')
ylabel('Current')
title('Current-Charge plot')
grid(True)
plot(s[1],s[2])
show()
```





The graphs show some initial distortions but later starts oscillating with the impressed frequency ω. It can be explained if we look at the analytic solution of the differential equation

$$L\frac{d^2q}{dt^2} + R\frac{dq}{dt} + \frac{q}{C} - V_0 \sin{(\omega t)}(t) = 0$$

$$\frac{d^2q}{t^2} + 2\gamma\frac{dq}{dt} + \omega_0^2 q - v_0 \sin{(\omega t)}(t) = 0$$

where $2\gamma = R/L, \omega_0^2 = 1/LC, v_0 = V(0)/L$

$$q(t) = Ae^{-\gamma t}\sin{(\omega' t + \phi)} + B\sin{(\omega t + \psi)}$$

where A is real and positive amplitude, ϕ, ψ are additional phases $\omega' = \sqrt{\omega_0^2 - \gamma^2}$ and

$$B = \frac{v_0}{\sqrt{(\omega_0^2 - \omega^2)^2 + 4\gamma^2\omega^2}}$$

The first term of the solution represents damped oscillations while the second term gives the forced oscillations. They interfere to give some distorted waveforms. Gradually the amplitude of damped oscillation reduces to zero and the LCR-circuit starts oscillating with impressed frequency.

5.1.6 Standing waves

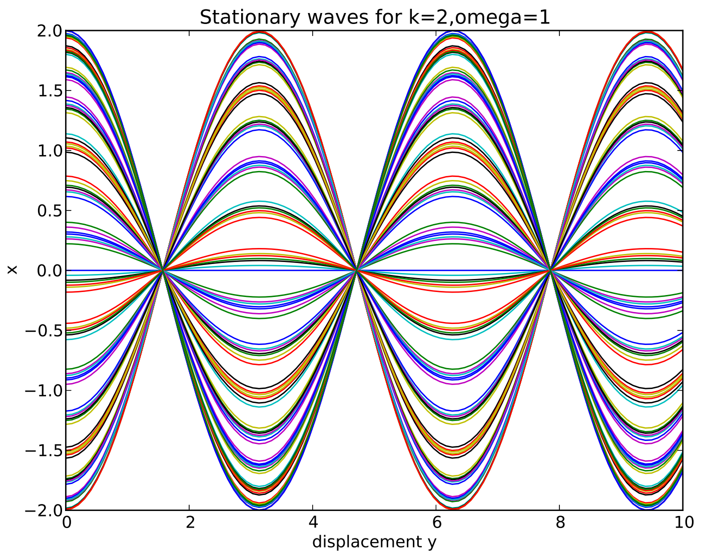
Standing waves are produced when a travelling wave gets reflected and superimpose with the original wave.

Principle

Let $y_1 = f(x - vt)$ be the forward wave and $y_2 = f(x + vt)$ be the reflected wave. Then $y = y_1 + y_2$ will be the composite wave. If it is a harmonic wave of spatial frequency $k = 2\pi/\lambda$ and angular frequency $\omega = 2\pi\nu$, it can be represented as $y1 = A\sin{(kx - \omega t)}$ and $y_2 = A\sin{(kx + \omega t)}$. Hence superposed wave is given by

$$y = A\sin{(kx - \omega t)} + A\sin{(kx + \omega t)}$$

It can be seen that the superposed pattern consists of points of zero amplitude called nodes . Energy is not transmitted through nodes. Hence the name stationary waves.



Program

```
from pylab import*
k,omega,=2.,1.
t=x=linspace(0,10,100)
y=[sin(k*x-omega*i)+sin(k*x+omega*i) for i in t]
xlabel('displacement y')
ylabel('x')
title('Stationary waves for k=2,omega=1')
plot(x,y)
show()
```

5.1.7 Simulation of radioactivity

Principle

Radioactive decay is an inherently non-deterministic process that can be simulated very naturally using the Monte Carlo method. The observation that the mean-life is a characteristic of the nucleus leads to the assumption that *the probability P of any one particle decaying per unit time in a radioactive sample is a constant.* Suppose that the probability of any given atom decaying over a time interval Δt is given by λ, where $0 < \lambda < 1$. Then the history of a single atom can be simulated by choosing a sequence of random numbers $x_k, k = 1, ..$ uniformly distributed on $(0,1)$. The atom survives until the first occurance of $x_k < \lambda$. This approach can be used to simulate an ensemble of N atoms. Let ΔN be the number of particles that decay in some small time interval Δt. Then the decay probability per particle, $\Delta N/N$, is proportional to the length of the time interval over which we observe the particle.

$$\frac{\Delta N(t)}{N(t)} = -\lambda \Delta t, \; \Delta N(t) = -\lambda \Delta t N(t)$$

This is a finite-difference equation in which $\Delta N(t)$ and Δt are experimental observables. Hence it cannot be integrated the way one solves a differential equation. But numerical or algebraic solutions are possible. Because the decay process is random, an exact value for $\Delta N(t)$ cannot be predicted. $\Delta N(t)$ may be taken as the average number of decays when observations are made of many identical systems of N radioactive particles.

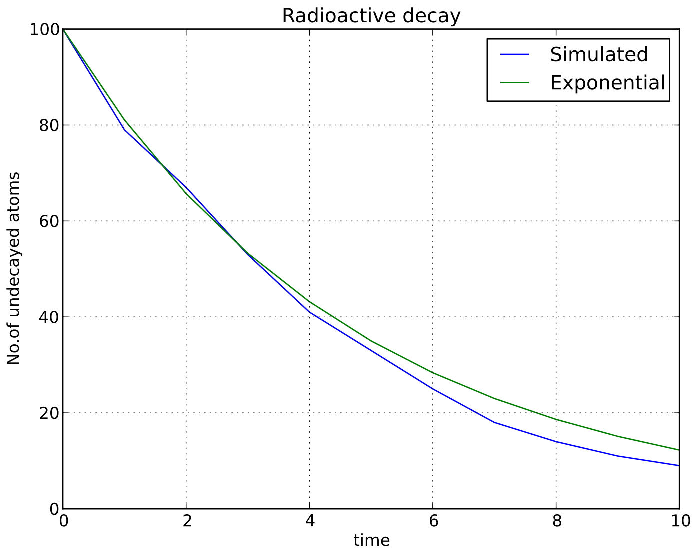
Algorithm

```
1: Read $N$ ,$D$,(initial number of parent and daughter atoms), maximum no of time
   intervals $m$ and decay constant $\lambda$
2: $T \leftarrow 0$
3: while $N > 0$ and $T < m$ do
4:     $NU \leftarrow N$
5:     for $i = 1$ to $NU$ do
6:         $x \leftarrow rand()$
7:         if $0 < x \leq lambda$ then
8:             $N \leftarrow N - 1$
9:         end if
10:    end for
11:    $T \leftarrow T + 1$
12: end while
13: End
```

Program:

```
from pylab import*
N,M,Lambda,T=100,10,0.21,0
NU=[N]
while N>0 and T<M:
    for i in range(N):
        if random()<= Lambda:N-=1
    NU+=[N]
    T+=1
T=arange(T+1)
plot(T,NU)
plot(T,Y)
Y=NU[0]*exp(-Lambda*T)
legend(['Simulated','Exponential'])
xlabel('time')
ylabel('No.of undecayed atoms')
title('Radioactive decay')
grid(True)
```

It can be seen that as N increases, the decay graph coincides with the exponential curve. Hence the differential equation $\frac{dN}{N} = -\lambda dt$ is only a large-number approximation of the difference equation $\frac{\Delta N}{N} = -\lambda \Delta t$



Index

m^{th} root, 25

Factor theorem, 70
Irrational Root Theorem, 70
Rational Root Theorem, 70

algorithm, 63
aliasing, 82
angular frequency, 104
append, 6
array, 29
axis(), 50

bar, 50
barh, 50
Bessel's Function, 59
binomial coefficients, 20
bisection method, 25
boundary conditions, 85

centigrade, 25
close, 16
cmath, 22
co-domain, 61
combination, 19
Complex Root Theorem, 70
concatenation, 6
constructor, 10
contour, 50
contourf, 50
count, 6
cross, 39
cross product, 40
cubic polynomial, 76
cubic spline, 66

D.Lehmer, 96
Danielson-Lanczos Lemma, 85
deflation, 73
Descartes Rule, 70
det, 45
determinant, 45
dictionaries, 2
dictionary, 5, 12
difference, 10
difference operator, 87
differential operator, 87
discrete transform, 82
domain, 61
dump, 16

eig, 46
eigenvalue, 46
eigenvector, 46
eigh, 46
eigvals, 46
eigvalsh, 46
elif, 14
else, 14
expectation, 78
expected value, 78
extend, 6

factorial, 19
Fahrenheit, 25
Fast Fourier Transform, 84
Fibonacci series, 18
figure(), 50
file, 16
Fourier transforms, 82
frozenset, 5
function, 15

Gamma function, 56
Gaussian function, 56

genfromtxt, 35
grid(), 50

Hailstorm number, 21
harmonic wave, 104
hermitian matrix, 46
homogeneous, 29

if, 14
image, 61
immutable, 5, 10, 12
importance sampling, 81
imshow, 54
imshow(), 1
index, 7, 39
inner product, 39
input, 1
insert, 7
interpolate, 66
intersection, 10
inv, 45
inverse function, 61
invertible matrices, 45
irrational roots, 74

key, 12

Laguerre's Method, 73
law of large numbers, 78
legend(), 50
Legendre Function, 57
Legurre's procedure, 76
linalg, 45
linear algebra, 45
linear equation, 45
list, 2, 5
load, 16
logistic map, 97
loglog plot, 51

magic numbers, 96
math, 15
Matplotlib, 49
module, 15
Monte Carlo, 77, 95
multidimensional, 29
multiplicative congruential recursive method, 96
mutable, 5

n-Zero theorem, 70
ndarray, 29
Newton-Raphson method, 24, 25
node, 104
non-singular, 45
norm, 45
numeric, 2
Numpy, 49
numpy, 15
Nyquist critical frequency, 82
Nyquist sampled, 82
Nyquist-Shannon Sampling Theorem, 81

ogrid[], 51
one-to-one, 62
one-to-one mapping, 61
onto mapping, 61
open, 16
operand, 3
operator, 3
orthogonality, 22
os, 15
outer, 40
outer product, 40

palindrome, 23
parametric plot, 53
Pascal's triangle, 20
Permutation, 19
pickle, 16
Pie charts, 53
pinv, 45
plot(), 49
Polar coordinate, 52
polar plot, 51
pop, 7
preimage, 61
prime number, 23
pseudo-random numbers, 95
pseudoinverse, 45

pylab, 15

quadratic equation, 22
quadratic formula, 77

random, 15
random variable, 78
rational number, 52
ravel, 33
raw_input, 1
read, 16
relaxation method, 87
Remainder theorem, 70
remove, 7
reshape, 33
reshape function, 34
resize, 33
reverse, 8
rhodonea, 52
Robert May, 97
rose curve, 52

sampling, 80
sampling theorem, 81
Sande-Tukey algorithm, 84
savefig(), 1
savetxt, 35
scatter plot, 51
semilogx, 51
semilogy, 51
shooting method, 85
show(), 1, 49
simple harmonic oscillator, 90
simulation, 81
sinusoid, 52
skiprows, 35
slice, 39
solve, 45
sort, 8
sorted, 8
spatial frequency, 104
square matrix, 46
square root, 24
standard deviation, 79
standing waves, 104
String, 4, 5
string, 2
swapping method, 64
symmetric difference, 10

text(), 50
the horizontal line test, 62
title(), 50
transpose, 33
triangular sequence, 20
tuple, 5, 12
tuples, 2

unary operation, 42
union, 10
unit matrix, 30

variance, 80

write, 16

xlabel(), 50

ylabel(), 50