Scientific Python

Milind Kumar V
EE16B025

02-02-2018

Abstract

This report presents an elementary study of the two primary modules used in scientific python, namely Scipy and Numpy. The various ways to determine running integrals using the definition of $tan^{-1}x$ as a definite integral are explored. This is further done using the Trapezoidal Rule setting a tolerance of 10^{-8} and subsequent error estimates are plotted.

1 Introduction

This report presents three ways of obtaining the values of the function $tan^{-1}x$ at various values of x one of which is using the *quad* function in the scipy module. The following relation is used to obtain the value of $tan^{-1}x$ over at any x

$$tan^{-1}x = \int_0^x \frac{du}{1+u^2}$$

The error in the values of $tan^{-1}x$ is measured by comparing the values returned by the *arctan* function *arctan* in numpy and is plotted. Subsequently, a strategy using the trapezoid rule is devised to compute the aforementioned integral. A comparision of the results of vectorisation of the code and the use of a for loop is also made.

2 Methods and results

The functions necessary for this exercise have been wriiten in a separate file and thus necessicate making making the following imports.

```
1
2     from __future__ import division
3     import numpy as np
4     from matplotlib import pyplot as plt
5     from functions import function1, taninv, trapezoid_integrate,
      trapezoid_integrate_for_loop
6
7
```

This first section deals with the necessary imports with the *functions* referring to the document in which the functions necessary for this assignement are stored. The following are the imports in the functions file.

```
1
2     from __future__ import division
3
4     import numpy as np
5     from scipy.integrate import quad
6
```

The second line of code above helps remove the ambiguity around the / operator in Python.

1. A function is made to compute the values $1/(1+u^2)$ and given a vector, to return this value for every element of the vector.

```
1
2     def function1(a):
3     a= np.array(a)
4     return 1.0/(1.0+a**2)
5
6
```

2. A vector of values is made for the above function to be tested upon using the following code.

```
1
2     # Step 2
3
4
5     start= 0.0
6     stop= 5.0
7     num= 51
8     # to include the last element and avoid obo error
9     vector= np.linspace(start, stop, num) # calling x vector
10
11
12    values= function1(vector)
13
14
```

3. This vector is passed to the above function and the returned values for $1/(1+u^2)$ are subsequently plotted using the matplotlib module.

```
1
2
3       ##########
4       # Step 3
5
6       plt.plot(vector, values, "bo")
7       plt.xlabel("x")
8       plt.ylabel("$f(x)$")
9       plt.title("""Plot of $f(x) = 1/(1+x^2) $""")
10      plt.show()
11      plt.close()
12
13
14
```

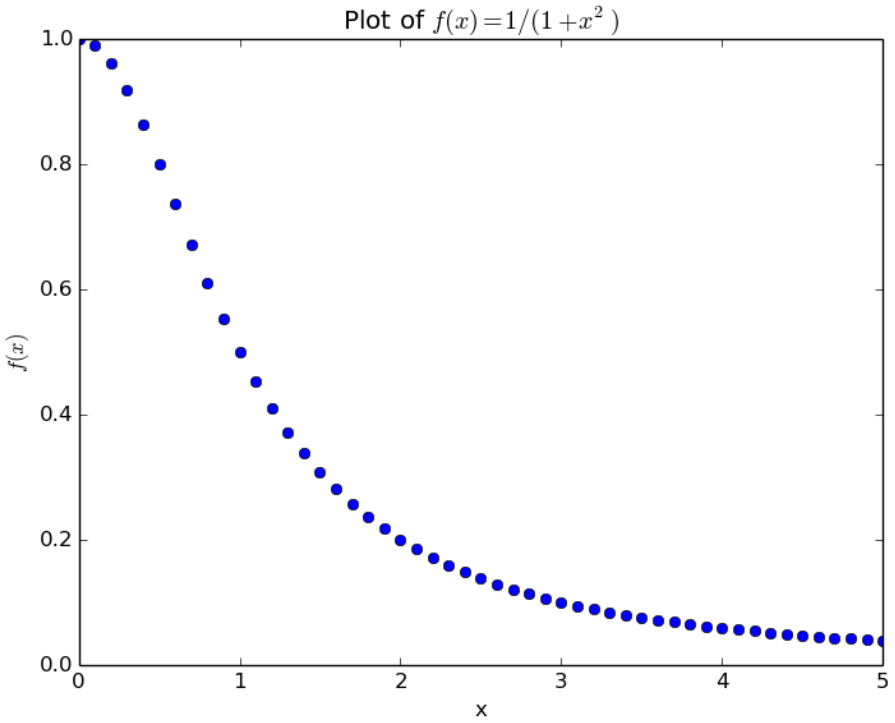

Figure 1: A plot of $f(x)$ vs x

Figure 3 shows the plot of $f(x) = 1/(1+x^2)$ which decays with x as is

expected. We now define $I(x) = tan^{-1}x$ as follows

$$I(x) = tan^{-1}x = \int_0^x f(x)dx = \int_0^x dx/(1+x^2) \tag{1}$$

4. The values of $I(x)$ are computed for vector which we have defined beforehand using the *quad* function from scipy.integrate. This function returns a tuple which is subsequently converted to a numpy array and returned to the calling function *taninv*. We consider the first column vector for the returned values of $tan^{-1}x$. The following code describes the *taninv* function from functions.py.

```
1
2     def taninv(a):
3     a= np.array(a)
4     array= []
5     for i in a:
6       array.append(quad(function1, 0.0, i))
7     array= np.array(array)
8   # array[:,0] = [round(i,5) for i in array[:,0]]
9     return (array)
10
11
```

The following code computes the error and makes a plot of $I(x)$ and the *np.arctan* function from the numpy library and the error between the two.

```
1
2     ############
3     # Step 4
4
5     taninv_calculated= np.array(taninv(vector)[:,0])
6     # remember that the quad function
7     # returns (integral, error)
8     taninv_actual= np.arctan(vector)
9     error= taninv_calculated- taninv_actual
10
11
12
13
14    plt.figure(1)
15
16    plt.subplot(211)
17    plt.plot( vector, taninv_calculated, "ro", label= "quad"
     )
```

```
18      plt.plot( vector, taninv_actual, "k-", label="$tan^{-1}(
        x)$", linewidth=2)
19      plt.xlabel("x")
20      plt.ylabel("$\int^x_0 du/(1+u^2) $")
21      plt.title("A comparision of quad and np.arctan()")
22      plt.legend(loc= "lower right")
23
24      plt.subplot(212)
25      plt.semilogy(vector, error, "ro")
26      plt.xlabel("x")
27      plt.ylabel("Error")
28      plt.title("Error in $\int^x_0 dx/(1+x^2) $")
29      plt.tight_layout()
30      plt.show()
31      plt.close()
32
33
34
35      plt.semilogy(vector, (taninv(vector))[:,1], "ro")
36      plt.xlabel("x")
37      plt.ylabel("Error from quad")
38      plt.title("Error returned by the quad function")
39
40      plt.tight_layout()  # ensures reasonable gaps between
41                  # the first subplot and the second
42      plt.show()
43
44      plt.close()
45
46
```

This results in the following plot.

The error returned by the quad function is plotted which looks as follows.

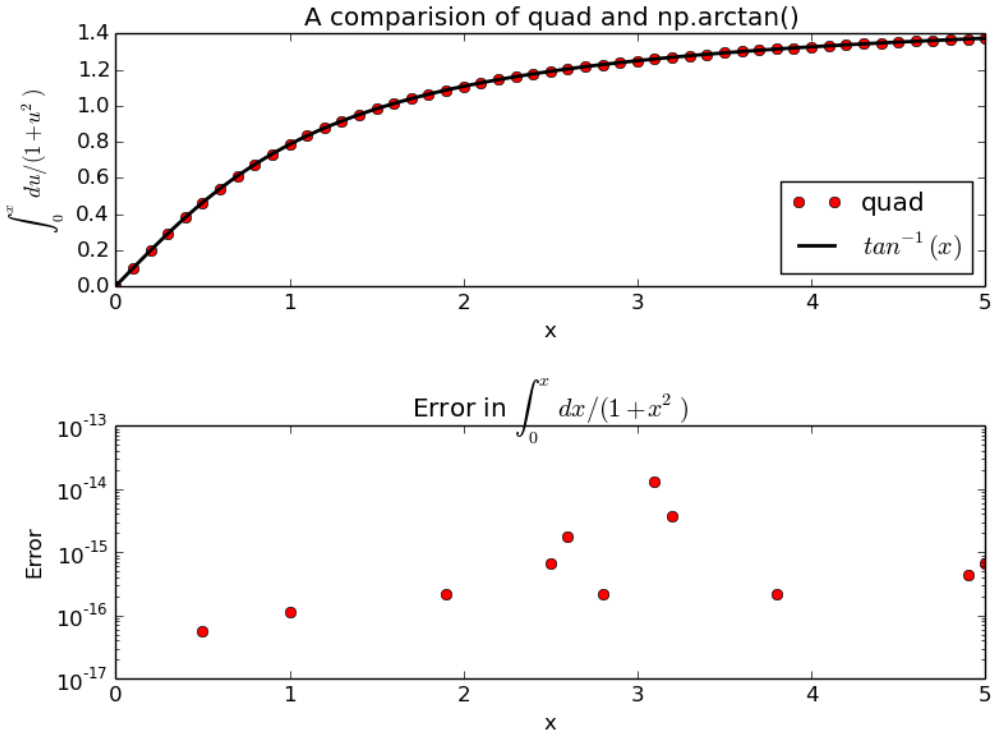

Figure 2: A plot of $I(x)$ and error

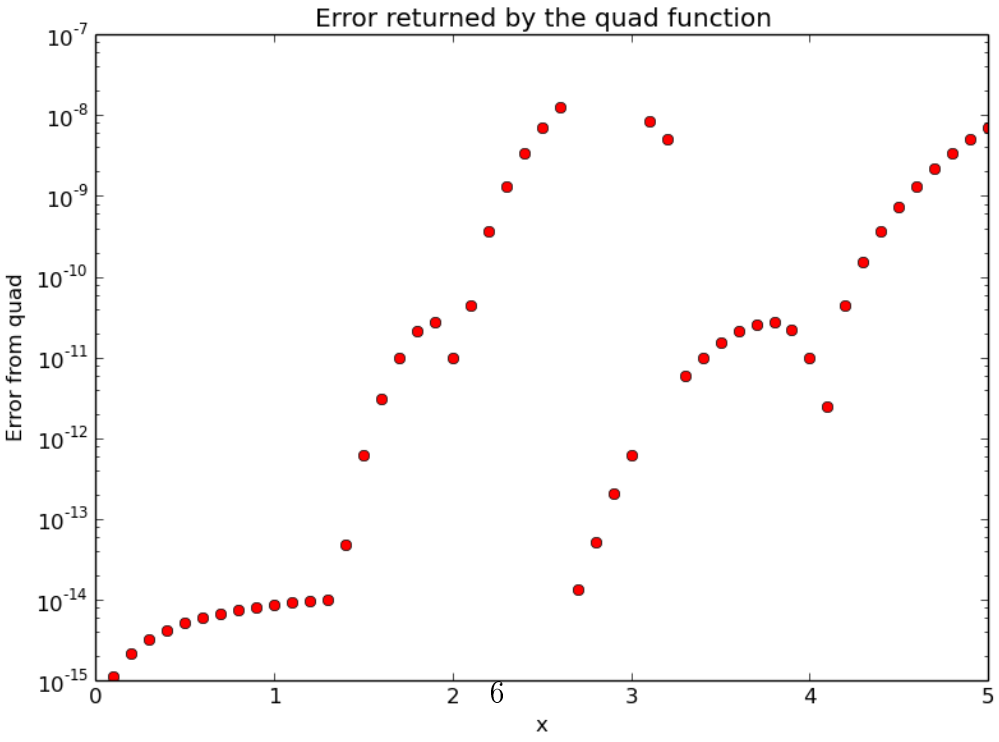

Figure 3: Error returned by the quad function

The table shows values of $tan^{-1}x$ returned by the *quad* function and the *np.arctan* function.

Value of x	**Returned value from quad**	**Value of** *np.arctan*
0.0	0.0	0.0
0.1	0.09967	0.09967
0.2	0.1974	0.1974
0.3	0.29146	0.29146
0.4	0.38051	0.38051
0.5	0.46365	0.46365
0.6	0.54042	0.54042
0.7	0.61073	0.61073
0.8	0.67474	0.67474
0.9	0.73282	0.73282
1.0	0.7854	0.7854
1.1	0.83298	0.83298
1.2	0.87606	0.87606
1.3	0.9151	0.9151
1.4	0.95055	0.95055
1.5	0.98279	0.98279
1.6	1.0122	1.0122
1.7	1.03907	1.03907
1.8	1.0637	1.0637
1.9	1.08632	1.08632
2.0	1.10715	1.10715
2.1	1.12638	1.12638
2.2	1.14417	1.14417
2.3	1.16067	1.16067
2.4	1.17601	1.17601
2.5	1.19029	1.19029
2.6	1.20362	1.20362
2.7	1.21609	1.21609
2.8	1.22777	1.22777
2.9	1.23874	1.23874
3.0	1.24905	1.24905
3.1	1.25875	1.25875
3.2	1.26791	1.26791
3.3	1.27656	1.27656
3.4	1.28474	1.28474
3.5	1.2925	1.2925

3.6	1.29985	1.29985
3.7	1.30683	1.30683
3.8	1.31347	1.31347
3.9	1.31979	1.31979
4.0	1.32582	1.32582
4.1	1.33156	1.33156
4.2	1.33705	1.33705
4.3	1.3423	1.3423
4.4	1.34732	1.34732
4.5	1.35213	1.35213
4.6	1.35674	1.35674
4.7	1.36116	1.36116
4.8	1.3654	1.3654
4.9	1.36948	1.36948
5.0	1.3734	1.3734

Table 1: Values of $tan^{-1}x$

5. The trapezoid rule is used to achieve results similar to the quad functions. The following code implements this algorithm:

$$I = \begin{cases} 0 & x = a \\ 0.5(f(a) + f(x_i)) + \sum_{j=2}^{i-1} f(x_j) & x = a + ih \end{cases}$$

where $f(x)$ is the function to be integrated, a the lower limit, b the upper limit and h the step size. Alternately, we can write out the above algorithm as follows,

$$I_i = h\left(\sum_{j=1}^{i} f(x_j) - \frac{1}{2}(f(x_1) + f(x_i))\right)$$

```
1
2       def trapezoid_integrate(a,b,h,f):
3         # making the assumption that b-a/h is integral
4         avbl_points= np.linspace(a,b,((b-a)/h)+1)
5         values= f(avbl_points)
6         integral= []
7         integral= h*(np.cumsum(values)- 0.5*( [values[0]]*len(
    values) + values ))
```

```
8       return np.array(integral)
9
10
11      def trapezoid_integrate_for_loop(a,b,h,f):
12        # making the assumption that b-a/h is integral
13        avbl_points= np.linspace(a,b,((b-a)/h)+1)
14        values= f(avbl_points)
15        integral= []
16        for i in range(0,len(values)):
17          integral.append(h*(np.sum(values[:i+1])- 0.5*(values
    [0]+values[len(values[:i+1])-1])))
18        return np.array(integral)
19
20
```

The first of these functions vectorises the code whereas the second function iterates through the values separated by h using a for loop. These functions are called in order and print the same output.

```
1
2       ###########
3       # Step 5
4
5       import time as t
6
7       # vectorised code
8       t1= t.time()
9       print trapezoid_integrate(0,5,0.1,function1)
10      t2= t.time()
11      print t2-t1
12      # code with the for trapezoid_integrate_for_loop
13      t1= t.time()
14      print trapezoid_integrate_for_loop(0,5,0.1,function1)
15      t2= t.time()
16      print t2-t1
17
18      # vectorised code
19      t1= t.time()
20      print trapezoid_integrate(0,5,0.0001,function1)
21      t2= t.time()
22      print t2-t1
23      # code with the for trapezoid_integrate_for_loop
24      t1= t.time()
25      print trapezoid_integrate_for_loop(0,5,0.0001,function1)
26      t2= t.time()
27      print t2-t1
28
```

```
29
```

This is a comparision of the performance of the vectorised code against the for loop and produces the following output.

```
milind@milind-VPCSB26FG:~/EE2703/assignment 2$ python lab2.py
[ 0.          0.09950495  0.19708682  0.29103531  0.38001031  0.46311376
  0.53987847  0.61020022  0.67424507  0.73235719  0.7849815   0.83260593
  0.87572217  0.91480133  0.95028059  0.98255709  1.01198665  1.03888507
  1.06353099  1.08616943  1.10701542  1.12625756  1.14406135  1.16057212
  1.17591769  1.19021069  1.20355055  1.21602521  1.22771268  1.23868228
  1.24899578  1.25870832  1.26786925  1.27652286  1.28470897  1.29246345
  1.29981869  1.30680403  1.31344605  1.31976891  1.3257946   1.33154319
  1.337033    1.34228082  1.34730204  1.35211077  1.35672003  1.36114179
  1.3653871   1.36946616  1.37338844]
0.00339794158936
[ 0.          0.09950495  0.19708682  0.29103531  0.38001031  0.46311376
  0.53987847  0.61020022  0.67424507  0.73235719  0.7849815   0.83260593
  0.87572217  0.91480133  0.95028059  0.98255709  1.01198665  1.03888507
  1.06353099  1.08616943  1.10701542  1.12625756  1.14406135  1.16057212
  1.17591769  1.19021069  1.20355055  1.21602521  1.22771268  1.23868228
  1.24899578  1.25870832  1.26786925  1.27652286  1.28470897  1.29246345
  1.29981869  1.30680403  1.31344605  1.31976891  1.3257946   1.33154319
  1.337033    1.34228082  1.34730204  1.35211077  1.35672003  1.36114179
  1.3653871   1.36946616  1.37338844]
0.00480914115906
[  0.00000000e+00   9.99999995e-05   1.99999997e-04 ...,   1.37339307e+00
   1.37339692e+00   1.37340077e+00]
0.015280008316
[  0.00000000e+00   9.99999995e-05   1.99999997e-04 ...,   1.37339307e+00
   1.37339692e+00   1.37340077e+00]
2.28464102745
```

Figure 4: Vectorised code vs for-loop as the input size goes up

Finally, the function $1/(1+x^2)$ is integrated over the range $(0, 1)$ several times halving the value of h successively and determining the value of h at which the tolerance value drops below 10^{-8}. The tolerance is defined as the maximum of the absolute value of the difference between the common points over consecutive values of h. This error and the deviation of the function from np.arctan are also plotted as shown in Figure 5.

```
1
2       #to determine h
3       interval_start= 0
```

```
    interval_end= 1
    h= 0.1
    hlist =[]

    tolerance= 1
    integrals =[]
    i=1
    error =[]

    integrals.append(trapezoid_integrate( interval_start, interval_end, h, function1))

    while tolerance > 10**(-8):
      h= h/2
      hlist.append(h)
      integrals.append(trapezoid_integrate( interval_start, interval_end, h, function1))
      feed= np.linspace(interval_start, interval_end,((interval_end- interval_start)/h)+1)
      actual_error= max(abs(integrals[i]- np.arctan(feed)))
      error.append( [max( abs( integrals[i][::2]- integrals[i-1])), actual_error])
      i+=1
      tolerance= error[len(error)-1][0]

    error= np.array(error)
    plt.loglog(hlist, error[:,0],"ro",label="estimated error")
    plt.loglog(hlist, error[:,1],"b+",label="actual error")
    plt.legend(loc="lower right")
    plt.show()
    plt.close()

```

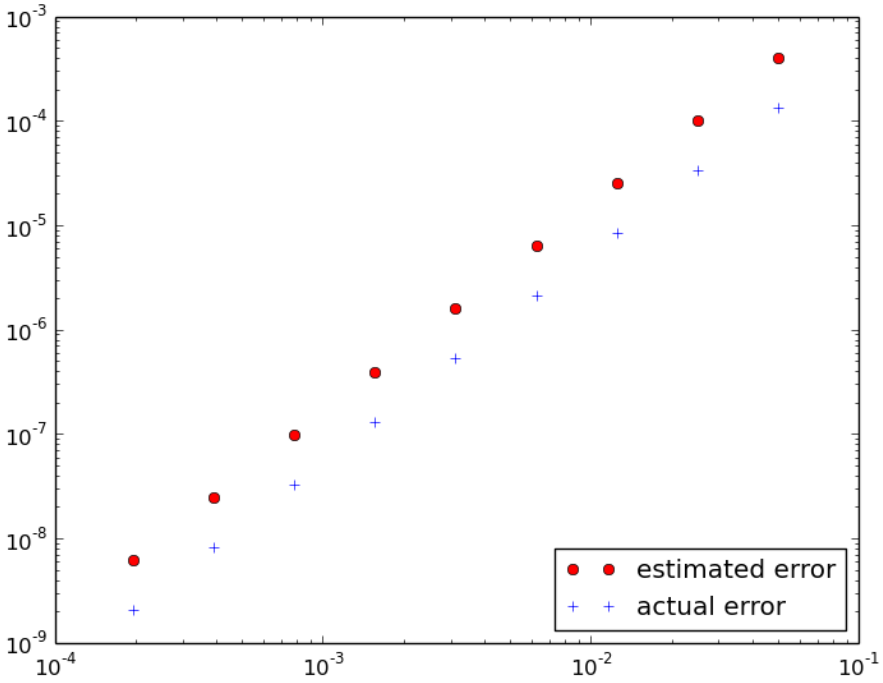

Figure 5: Estimated error and Exact error

3 Conclusion

The *quad* function and the np.arctan function have marginal difference and this can be seen from Figure 2. The matplotlib library further proves to be invaluable in plotting and visualization of data. We also notice that vectorization of code significantly speeds up code as computation becomes more and more intensive. This can be seen from Figure 4 When the size of h is sufficiently large at 0.1, the code with loops performs nearly as well as the vectorized code, however this is not the case with a much smaller h. The trapezoidal rule further serves as a good method to integrate $f(x)$ as Figure 5 indicates. Error declines quite significantly with decrease in h.

In conclusion, methods necessary for scientific computing using python, essentially the scipy and numpy libraries have been outlined with emphasis on the significant advantage of vector methods.