EE5471: Numerical Integration

Harishankar Ramachandran
EE Dept, IIT Madras

August 19, 2019

1 Reading Portion

Chapter on integration, both from Numerical Recipes and from Stoer's book.

2 Codes

Last week I gave a C code that could be used inside weave. This week I show a different technique where we take a Fortran function and convert it into a built in Python function. (Doing the same in C is more difficult, as the tools are not so convenient).

We will create a python module called romberg that contains three functions, namely polint, trapzd and qromb. These are basically the numerical recipes versions of the algorithms with small changes.

To get the module working, download romberg.f from the site. Then run 'f2py -c -m romberg romberg.f' to generate romberg.so, the module. The '-c' switch says compile the fortran file, and the '-m romberg' says create a module called romberg.

We add directives to a Fortran code to make f2py automatically generate the python module (see the code). However, there are limits to this. f2py does not know how to handle functions that are to be passed from python. Here is where we need such a capability:

We write a routine in C to integrate a function. The calling sequence of the function would be:

```
double intg(double *func(double),double a,double b)
```

Python should call this function as

```
I=intg(fun,a,b)
```

where `intg` is a python function. Now there is the tricky part. How do C or Fortran know how to call a python function? This is known as "call back" meaning that we wrote a C routine and linked it to python, but that routine, in turn, is calling back to python. How is this function to be written and how is it to be linked?

f2py is happiest with Fortran code, so I show the technique there. Consider the routine `trapzd`. It is part of romberg.f. Here is the calling syntax and the directives for this function:

```
      SUBROUTINE trapzd(func,a,b,sin,sout,n)
c
cf2py intent(out) :: sout
cf2py integer :: n
cf2py real*8 :: a
cf2py real*8 :: b
cf2py real*8 :: sin
cf2py real*8 :: sout
```

These lines just define the variables of the function and say that `sout` is the return variable. Note that the recipes code had a single variable 's' that both passed the old integral value to the function and returned the new value. It is easier for f2py (and python) to deal with variables that are either only input or only output. So I split it into two variables.

```
     EXTERNAL func
cf2py a = func(b)
```

This is where f2py learns that func is a function. The "EXTERNAL" statement in Fortran says that. And the line below gives a directive telling f2py what sort of interface to create. The rest is done by f2py and it says that func is a call back function. **Note that the function is a Python function being called from Fortran.**

Now let us look at how polint is handled:

```
      SUBROUTINE polint(xx,yy,n,x,y,err,c,d)
c
cf2py intent(out) :: y
cf2py intent(out) :: err
cf2py intent(hide) :: n
cf2py intent(hide) :: c
cf2py intent(hide) :: d
cf2py integer :: n
cf2py real*8,dimension(n) :: xx
cf2py real*8,dimension(n) :: yy
cf2py real*8,dimension(n) :: c
cf2py real*8,dimension(n) :: d
cf2py real*8 :: x
cf2py real*8 :: y
cf2py real*8 :: err
```

We see the same type of information about the arguments. But there are some new features.

- xx, yy , c and d are arrays of length n. Now, "n" is required by Fortran. But Python knows "n" from len(xx). So we ask f2py to "hide" the variable. i.e., pass the variable to Fortan, but the calling sequence on Python's side will not include it.
- Similarly, c and d are work arrays. While they are created and sent from Python, they need not clutter the Python code. So we ask f2py to "hide" both arrays.
- Arrays are declared using the dimension(n) directive.

With this in place we create the module using:

```
f2py -c -m romberg romberg.f
```

and then, within ipython I execute the following commands:

```
# script to test the romberg module
from scipy import *
from matplotlib.pyplot import *
import scipy.special as sp
import romberg as r
# integrate sin(x) as a test
print r"Testing Trapzd ..."
s=0.0                 # initialize the integral to zero
I0=cos(0.0)-cos(1.0)  # the exact value to compare with
print "i\tI[i]\tError"# print a heading
for i in range(1,10):
    s=r.trapzd(lambda x:sin(x),0,1,s,i)
    print "%1d %.8f %.2e" % (i,s,s-I0)
# end for
```

The import command loads the romberg module with the alias 'r'. Look at the line calling the trapzd function:

```
s=r.trapzd(lambda x:sin(x),0,1,s,i)
```

The bounds of the integral are zero and one, and the input value is 's'. The output value is returned, so not an argument. 'i' is the iteration number. But the first argument is 'fun', a call back function. And it is defined as

```
lambda x:sin(x)
```

This is as powerful as it gets. I am creating and passing a function to this Fortran function and it understands how to use the python function! The 'lambda' syntax is used to create "anonymous" functions. The syntax is:

```
lambda dummy_vars: expression
```

So this created a function without a name that returned the value of $\sin x$ when passed the value of x. I could also have defined the function normally using

```
def f(dummy_vars):
  return expression
```

These are exactly equivalent, except that for a function that is used exactly once, and is so simple, the lambda approach is very convenient. The output of this script is as follows:

```
Testing Trapzd ...
i I[i] Error
1 0.42073549 -3.90e-02
2 0.45008052 -9.62e-03
3 0.45730094 -2.40e-03
```

```
4 0.45909897 -5.99e-04
5 0.45954804 -1.50e-04
6 0.45966028 -3.74e-05
7 0.45968834 -9.35e-06
8 0.45969536 -2.34e-06
9 0.45969711 -5.85e-07
```

What is worth appreciating is how *convenient* the module has become. And with very little effort on our part.

Now that we are in ipython, we can also look at the usage of trapzd and polint above:

```
In [4]: r.trapzd?
a : input float
b : input float
sin : input float
n : input int
Other Parameters
----------------
func_extra_args : input tuple, optional
    Default: ()
Returns
-------
sout : float
Notes
-----
Call-back functions::
  def func(b): return a
  Required arguments:
    b : input float
  Return objects:
    a : float
```

It tells us the input parameters and also tells us how the function should be used.

This is nice. But polint is even nicer:

```
In [9]: r.polint?
Type:       fortran
String Form:<fortran object>
Docstring:
y,err = polint(xx,yy,x)
Wrapper for "polint".
Parameters
----------
xx : input rank-1 array('d') with bounds (n)
yy : input rank-1 array('d') with bounds (n)
x : input float
Returns
-------
y : float
err : float
```

All the extra inputs are gone! It is trivial to use.

Now let us look at qromb.

```
SUBROUTINE qromb(func,a,b,ss,dss,numcalls,EPS,K,c,d)
c
cf2py intent(out) :: ss
cf2py intent(out) :: dss
cf2py intent(out) :: numcalls
cf2py real*8,optional :: EPS /1.e-6/
cf2py integer,optional :: K /5/
cf2py intent(hide) :: c
cf2py intent(hide) :: d
cf2py real*8,dimension(K) :: c
cf2py real*8,dimension(K) :: d
cf2py real*8 :: a
cf2py real*8 :: b
cf2py real*8 :: ss
      EXTERNAL func
cf2py real*8 :: y1
cf2py real*8 :: y2
cf2py y1 = func(y2)
```

We have the same things as before, with an extra feature. EPS and K are defined to be optional with a default value. Here is what the help says in ipython after importing the module:

```
func_extra_args : input tuple, optional
    Default: ()
eps : input float, optional
    Default: 1e-06
k : input int, optional
    Default: 5
Returns
-------
ss : float
dss : float
numcalls : int
Notes
-----
Call-back functions::
  def func(y2): return y1
  Required arguments:
    y2 : input float
  Return objects:
    y1 : float
```

It tells us that eps and k (made lower case by default, since FORTRAN treats the code as case insensitive) are optional inputs.

Module Usage

polint: This is a restricted polint meant for Romberg. It implements the minimal NR algorithm.

```
y,err = polint(xx,yy,x)
```

where xx and yy define the table and x is the point at which the value is desired. The value and the estimated error are returned. If two xx values are the same, we have an error condition. This can be checked by seeing if err is negative.

trapzd: This implements the incremental trapzd algorithm defined in NR. It only does the extra function calls and modifies the old estimate and returns the new.

```
s_out = trapzd(func,a,b,s_in,n)
```

where func is a python function, a and b are bounds of the integral, s_in is the old value of the integral (ignored for $n = 1$ and n is the iteration index. Returns the improved estimate of the integral in s_out.

qromb: This also implements qromb as defined in NR. NR hardcoded the order and accuracy which are here optional arguments.

```
ss,dss,numcalls = qromb(func,a,b,[eps,k])
```

where func is a python function, a and b are the bounds of the integral, eps is the desired accuracy (default of 10^{-6}) and k is the order to use (default is 5). The routine returns the estimate in ss, an estimate of the absolute value of the error in dss and the number of function calls in numcalls. If the error is negative it means that qromb did not converge.

3 Programming Portion

The total power in an Electromagnetic mode has to be computed in order to properly normalize the fields. The integral involved is

$$\frac{1}{2}\int\int \left(E_x H_y^* - E_y H_x^*\right) dxdy$$

For dielectric fibres, this calculation involves the following integral.

$$I = \frac{2}{a^2}\int_0^a J_\nu^2(\kappa r)\, rdr + \frac{2}{a^2}\left|\frac{J_\nu(\kappa a)}{K_\nu(\gamma a)}\right|^2 \int_a^\infty K_\nu^2(\gamma r)\, rdr \tag{1}$$

where J_ν is the Bessel function of the first kind and K_ν is the modified Bessel function of the second kind (standard special functions).

The exact solution to Eq. (1) is known to be

$$\left[J_\nu^2(\kappa a) - J_{\nu+1}(\kappa a) J_{\nu-1}(\kappa a)\right] + \left|\frac{J_\nu(\kappa a)}{K_\nu(\gamma a)}\right|^2 \left[K_{\nu+1}(\gamma a) K_{\nu-1}(\gamma a) - K_\nu^2(\gamma a)\right] \tag{2}$$

In this assignment, we are going to explore the numerical evaluation of this integral for the case of $\nu = 3$, $\kappa a = 2.7$ and $\gamma a = 1.2$.

1. Transform the integral to the dimensionless variable $u = r/a$. You should obtain:

$$I = 2\int_0^1 J_\nu^2(ku)udu + 2\left|\frac{J_\nu(k)}{K_\nu(g)}\right|^2 \int_1^\infty K_\nu^2(gu)udu$$

   where $k = \kappa a = 2.7$ and $g = \gamma a = 1.2$.

2. Graph the integrand in Python from 0 to ∞ (use a semi-log plot, since the function amplitudes vary widely, but the function is positive everywhere). Study its characteristics, since those will be required to design your algorithm. Is the function continuous? Is it smooth everywhere?

3. The integral goes to ∞. So we have to discard part of the integral and numerically compute

$$\int_0^a f(x)dx$$

   What value of a will a good choice? Can you analytically bound

$$\int_a^\infty f(x)dx$$

   for large a? **Hint:** The Bessel functions all have approximations in terms of complex exponentials for large arguments.

4. Use Python's built-in integrator, `quad`, found in `scipy.integrate` to do the integration. Verify the given solution. How many calls were required? (Integrate from 0 to a).

   - You need to define the integrand as a Python function.
   - The function form depends on whether $u < 1$ or $u > 1$.
   - Ask for full output, and look at `scipy.integrate.quad_explain()` for details. One of the outputs is the number of calls made to the function.

5. Use the trapezoidal method and obtain the integral. How does the error scale with h? (See the above example code for how to use the `trapzd` routine in module romberg. Does it matter if $r = 1$ is one of the r_j values or if it lies between some r_j and r_{j+1}?

   - Define a global variable 'count' to keep track of how many calls have been made (I discussed global variables in python, in class).
   - Plot the error (defined as difference between the numerical integral and the theoretical value) on the y axis and number of calls on the x axis. What sort of plot should you use?
   - What is the trend? How good is the trapzoidal algorithm?

6. Use the romberg module's `qromb` to investigate Romberg integration. Integrate the entire integral from 0 to ∞. How does the error scale with number of calls? Plot the trend.

7. Split the romberg integrals into $(0,1)$ and $(1,\infty)$. Repeat the study. Explain the difference in the number of calls required to achieve a given accuracy.

8. Write a python program that implements qromb using `trapzd` and `polint` from the romberg module.

9. Vary the order of qromb and see how the number of calls scales for a fixed error of 10^{-8}.

10. Use spline integration to improve over trapezoidal integration. What scaling do you get (plot it!)? Can you explain why? **Hint:** Look at the spline interpolated function near $r = 1$.

    Note that simple spline integration is a trivial use of the B-Spline routines in scipy. To get the trivial splines, use

    ```
    import scipy.interpolate as si
    tck=si.splrep(x,y) # x,y contains table of data
    ```

    This generates cubic splines that correspond to the curve that passes exactly through the points. To interpolate, use

    ```
    yy=si.splev(xx,tck) ## xx contains points at which to interpolate
    ```

    To integrate, use

    ```
    I=si.splint(a,b,tck)
    ```

    These all effectively use the not-a-knot boundary condition.

    B-Splines can do much more. They can give *smooth* curves that give more importance to certain points and minimize the least-squares error of the fit.

11. Break up the spline integration into two separate parts, from 0 to 1 and from 1 to $u_{\max}$. How do the scalings change (plot ...)? What does this tell us about spline fitting and spline integration?

12. Rewrite the python to implement qromb where h is reduced to $h/3$ on each step and interface it to Python. You cannot use the `trapzd` routine in module `Romberg` since it implements a divide by two algorithm. Implement even that routine in python but use polint from the module . Numerically compare its performance with the standard Romberg. Analytically predict the performance difference.

13. Plot all the scalings on a single graph, and compare the various approaches used to understand which method works best.

Note: When is Romberg optimal?

Any scheme that samples a function to obtain its integral is optimal if it uses each and every sample equally well. So each sample must give the same amount of useful information. A function that is nearly zero everywhere except in a very small region is not usefully integrated by a uniformly sampled scheme - I would be much better off breaking the domain into multiple regions. A better approach would be to tranform the *function* to make it uniform:

Suppose I am given a function $f(x) = 0.01x^2$ that I want to integrate from 0 to 10. I define a transformation $u(x)$ such that $f(x)dx = g(u)du$, i.e.,

$$g(u) = f(x)\frac{dx}{du}$$

My goal is to make $g(u)$ as close to uniform as possible. So I guess $u = x^\alpha$ which means $x = u^{1/\alpha}$. Then,

$$\begin{aligned} g(u) &= u^{2/\alpha}\left(\frac{1}{\alpha}\right)\frac{u^{1/\alpha}}{u} \\ &= \frac{1}{\alpha}u^{\frac{3-\alpha}{\alpha}} \end{aligned}$$

That gives me the integral

$$I = 0.01\int_0^{10} x^2 dx = \frac{0.01}{3}\int_0^{1000} du = \frac{10}{3} = 0.01\frac{10^3}{3}$$

For a complex integral, obviously I cannot make $g(u)$ exactly a constant. But the more constant I make it, the better Romberg will perform. (I had given a problem on this earlier, but removed it as the assignment is already too long.)