- 
- 
- 
- 
- 
- 
- 

PerformancePython

Contents

A beginners guide to using Python for performance computing

A comparison of weave with NumPy, Pyrex, Psyco, Fortran (77 and 90) and C++ for solving Laplace's equation. This article was originally written by Prabhu Ramachandran.

laplace.py is the complete Python code discussed below. The source tarball (perfpy_2.tgz) contains in addition the Fortran code, the pure C++ code, the Pyrex sources and a setup.py script to build the f2py and Pyrex module.

Introduction

This is a simple introductory document to using Python for performance computing. We'll use NumPy, SciPy's weave (using both weave.blitz and weave.inline) and Pyrex. We will also show how to use f2py to wrap a Fortran subroutine and call it from within Python.

We will also use this opportunity to benchmark the various ways to solve a particular numerical problem in Python and compare them to an implementation of the algorithm in C++.

Problem description

The example we will consider is a very simple (read, trivial) case of solving the 2D Laplace

equation using an iterative finite difference scheme (four point averaging, Gauss-Seidel or Gauss-Jordan). The formal specification of the problem is as follows. We are required to solve for some unknown function u(x,y) such that $\nabla^2 u = 0$ with a boundary condition specified. For convenience the domain of interest is considered to be a rectangle and the boundary values at the sides of this rectangle are given.

It can be shown that this problem can be solved using a simple four point averaging scheme as follows. Discretise the domain into an (nx x ny) grid of points. Then the function u can be represented as a 2 dimensional array - u(nx, ny). The values of u along the sides of the rectangle are given. The solution can be obtained by iterating in the following manner.

Toggle line numbers

```
1 for i in range(1, nx-1):
2     for j in range(1, ny-1):
3         u[i,j] = ((u[i-1, j] + u[i+1, j])*dy**2 +
4                   (u[i, j-1] + u[i, j+1])*dx**2)/(2.0*(dx**2 +
dy**2))
```

Where dx and dy are the lengths along the x and y axis of the discretised domain.

Numerical Solution

Implementing a solver for this is straight forward in Pure Python. Use a simple NumPy array to store the solution matrix u. The following code demonstrates a simple solver.

Toggle line numbers

```
1  import numpy
2
3  class Grid:
4      """A simple grid class that stores the details and solution of
the
5      computational grid."""
6      def __init__(self, nx=10, ny=10, xmin=0.0, xmax=1.0,
7                   ymin=0.0, ymax=1.0):
8          self.xmin, self.xmax, self.ymin, self.ymax = xmin, xmax,
ymin, ymax
9          self.dx = float(xmax-xmin)/(nx-1)
10         self.dy = float(ymax-ymin)/(ny-1)
11         self.u = numpy.zeros((nx, ny), 'd')
12         # used to compute the change in solution in some of the
methods.
13         self.old_u = self.u.copy()
14
15     def setBCFunc(self, func):
16         """Sets the BC given a function of two variables."""
17         xmin, ymin = self.xmin, self.ymin
18         xmax, ymax = self.xmax, self.ymax
19         x = numpy.arange(xmin, xmax + self.dx*0.5, self.dx)
20         y = numpy.arange(ymin, ymax + self.dy*0.5, self.dy)
21         self.u[0  ,:] = func(xmin,y)
22         self.u[-1,:] = func(xmax,y)
```

```
23          self.u[:,0] = func(x,ymin)
24          self.u[:,-1] = func(x,ymax)
25
26      def computeError(self):
27          """Computes absolute error using an L2 norm for the solution.
28          This requires that self.u and self.old_u must be
appropriately
29          setup."""
30          v = (self.u - self.old_u).flat
31          return numpy.sqrt(numpy.dot(v,v))
32
33
34  class LaplaceSolver:
35      """A simple Laplacian solver that can use different schemes to
36      solve the problem."""
37      def __init__(self, grid, stepper='numeric'):
38          self.grid = grid
39          self.setTimeStepper(stepper)
40
41      def slowTimeStep(self, dt=0.0):
42          """Takes a time step using straight forward Python
loops."""
43          g = self.grid
44          nx, ny = g.u.shape
45          dx2, dy2 = g.dx**2, g.dy**2
46          dnr_inv = 0.5/(dx2 + dy2)
47          u = g.u
48
49          err = 0.0
50          for i in range(1, nx-1):
51              for j in range(1, ny-1):
52                  tmp = u[i,j]
53                  u[i,j] = ((u[i-1, j] + u[i+1, j])*dy2 +
54                            (u[i, j-1] + u[i, j+1])*dx2)*dnr_inv
55                  diff = u[i,j] - tmp
56                  err += diff*diff
57
58          return numpy.sqrt(err)
59
60      def setTimeStepper(self, stepper='numeric'):
61          """Sets the time step scheme to be used while solving given
a
62          string which should be one of ['slow', 'numeric', 'blitz',
63          'inline', 'fastinline', 'fortran']."""
64          if stepper == 'slow':
65              self.timeStep = self.slowTimeStep
66          #  ...
67          else:
68              self.timeStep = self.numericTimeStep
69
70      def solve(self, n_iter=0, eps=1.0e-16):
71          err = self.timeStep()
```

```
        count = 1

        while err > eps:
            if n_iter and count >= n_iter:
                return err
            err = self.timeStep()
            count = count + 1

        return count
```

The code is pretty simple and very easy to write but if we run it for any sizeable problem (say a 500 x 500 grid of points), we'll see that it takes *forever* to run. The CPU hog in this case is the `slowTimeStep` method. In the next section we will speed it up using NumPy.

Using NumPy

It turns out that the innermost loop of the `LaplaceSolver.slowTimeStep` method can be readily expressed by a much simpler NumPy expression. Here is a re-written timeStep method.

Toggle line numbers

```
    def numericTimeStep(self, dt=0.0):
        """Takes a time step using a NumPy expression."""
        g = self.grid
        dx2, dy2 = g.dx**2, g.dy**2
        dnr_inv = 0.5/(dx2 + dy2)
        u = g.u
        g.old_u = u.copy() # needed to compute the error.

        # The actual iteration
        u[1:-1, 1:-1] = ((u[0:-2, 1:-1] + u[2:, 1:-1])*dy2 +
                         (u[1:-1,0:-2] + u[1:-1,
2:])*dx2)*dnr_inv

        return g.computeError()
```

The entire for i and j loops have been replaced by a single NumPy expression. NumPy expressions operate elementwise and hence the above expression works. It basically computes the four point average. If you have gone through the NumPy tutorial and played with NumPy a bit you should be able to understand how this works. The beauty of the expression is that its completely done in C. This makes the computation *much* faster. For a quick comparison here are some numbers for a single iteration on a 500x500 grid. On a PIII 450Mhz with 192 MB RAM, the above takes about 0.3 seconds whereas the previous one takes around 15 seconds. This is close to a 50 fold speed increase. You will also note a few things.

1. We cannot compute the error the way we did earlier inside the for loop. We need to make a copy of the data and then use the computeError function to do this. This costs us memory and is not very pretty. This is certainly a limitation but is worth a 50 fold speed increase.
2. The expression will use temporaries. Hence, during one iteration, the computed values at an already computed location will not be used during the iteration. For

instance, in the original for loop, once the value of u[1,1] is computed, the next value for u[1,2] will use the newly computed u[1,1] and not the old one. However, since the NumPy expression uses temporaries internally, only the old value of u[1,1] will be used. This is not a serious issue in this case because it is known that even when this happens the algorithm will converge (but in twice as much time, which reduces the benefit by a factor of 2, which still leaves us with a 25 fold increase).

Apart from these two issues its clear that using NumPy boosts speed tremendously. We will now use the amazing weave package to speed this up further.

Using weave.blitz

The NumPy expression can be speeded up quite a bit if we use weave.blitz. Here is the new function.

Toggle line numbers

```
 1 #  import necessary modules and functions
 2 from scipy import weave
 3 #  ...
 4     def blitzTimeStep(self, dt=0.0):
 5         """Takes a time step using a NumPy expression that has been
 6         blitzed using weave."""
 7         g = self.grid
 8         dx2, dy2 = g.dx**2, g.dy**2
 9         dnr_inv = 0.5/(dx2 + dy2)
10         u = g.u
11         g.old_u = u.copy()
12
13         #  The actual iteration
14         expr = "u[1:-1,1:-1] = ((u[0:-2,1:-1] + u[2:,
1:-1])*dy2 + "\
15           "(u[1:-1,0:-2] + u[1:-1,2:])*dx2)*dnr_inv"
16         weave.blitz(expr, check_size=0)
17
18         return g.computeError()
```

If you notice, the only thing that has changed is that we put quotes around the original numeric expression and call this string 'expr' and then invoke weave.blitz. The 'check_size' keyword when set to 1 does a few sanity checks and is to be used when you are debugging your code. However, for pure speed it is wise to set it to 0. This time when we time the code for a 500x500 array for a single iteration it takes only about 0.1 seconds which is about a three fold increase! There are again a few things to note.

1. The first time this method is called, it will take a long while to do some magic behind your back. The next time it is called, it will run immediately. More details on this are in the weave documentation. Basically, weave.blitz converts the NumPy expression into C++ code and uses blitz++ for the array expression, builds a Python module, stores it in a special place and invokes that the next time the function call is made.
2. Again we need to use a temporary array to compute the error.
3. blitz does *not* use temporaries for the computation and therefore behaves more like the original (slow) for loop in that the computed values are re-used immediately.

Apart from these points, the results are identical as compared to the original for loop. It's

only about 170 times faster than the original code! We will now look at yet another way to speed up our original for loop. Enter weave.inline!

Using weave.inline

Inline allows one to embed C or C++ code directly into your Python code. Here is a simple version of an inlined version of the code.

Toggle line numbers

```
 1 from scipy.weave import converters
 2 #  ...
 3     def inlineTimeStep(self, dt=0.0):
 4         """Takes a time step using inlined C code -- this version
uses
 5         blitz arrays."""
 6         g = self.grid
 7         nx, ny = g.u.shape
 8         dx2, dy2 = g.dx**2, g.dy**2
 9         dnr_inv = 0.5/(dx2 + dy2)
10         u = g.u
11
12         code = """
13                #line 120 "laplace.py" (This is only useful for
debugging)
14                double tmp, err, diff;
15                err = 0.0;
16                for (int i=1; i<nx-1; ++i) {
17                    for (int j=1; j<ny-1; ++j) {
18                        tmp = u(i,j);
19                        u(i,j) = ((u(i-1,j) + u(i+1,j))*dy2 +
20                                  (u(i,j-1) +
u(i,j+1))*dx2)*dnr_inv;
21                        diff = u(i,j) - tmp;
22                        err += diff*diff;
23                    }
24                }
25                return_val = sqrt(err);
26                """
27         #  compiler keyword only needed on windows with MSVC
installed
28         err = weave.inline(code,
29                            ['u','dx2','dy2','dnr_inv','nx',
'ny'],
30                            type_converters=converters.blitz,
31                            compiler = 'gcc')
32         return err
```

The code itself looks very straightforward (which is what makes inline so cool). The inline function call arguments are all self explanatory. The line with '#line 120 ...' is only used, for debugging and doesn't affect the speed in anyway. Again the first time you run this function it takes a long while to do something behind the scenes and the next time it blazes away. This time notice that we have far more flexibility inside our loop and can easily compute an error term without a need for temporary arrays. Timing this version results in a

time for a 500x500 array of a mere 0.04 seconds per iteration! This corresponds to a whopping 375 fold speed increase over the plain old for loop. And remember we haven't sacrificed any of Python's incredible flexibility! This loop contains code that looks very nice but if we want to we can speed things up further by writing a little dirty code. We won't get into that here but it suffices to say that its possible to get a further factor of two speed up by using a different approach. The code for this basically does pointer arithmetic on the NumPy array data instead of using blitz++ arrays. This code was contributed by Eric Jones. The source code accompanying this article contains this code.

Next, we look at how it is possible to easily implement the loop inside Fortran and call it from Python by using f2py.

Using f2py

f2py is an amazing utility that lets you easily call Fortran functions from Python. First we will write a small Fortran77 subroutine to do our calculation. Here is the code.

```
c File flaplace.f
      subroutine timestep(u,n,m,dx,dy,error)
      double precision u(n,m)
      double precision dx,dy,dx2,dy2,dnr_inv,tmp,diff
      integer n,m,i,j
cf2py intent(in) :: dx,dy
cf2py intent(in,out) :: u
cf2py intent(out) :: error
cf2py intent(hide) :: n,m
      dx2 = dx*dx
      dy2 = dy*dy
      dnr_inv = 0.5d0 / (dx2+dy2)
      error = 0
      do 200 j=2,m-1
         do 100 i=2,n-1
            tmp = u(i,j)
            u(i,j) = ((u(i-1,j) + u(i+1,j))*dy2 +
     &           (u(i,j-1) + u(i,j+1))*dx2)*dnr_inv
            diff = u(i,j) - tmp
            error = error + diff*diff
 100     continue
 200  continue
      error = sqrt(error)
      end
```

The lines starting with cf2py are special f2py directives and are documented in f2py. The rest of the code is straightforward for those who know some Fortran. We trivially create a Python module for this using the following command.

```
% f2py -c flaplace.f -m flaplace
```

Here is how the Python side of things looks.

Toggle line numbers

```
1 import flaplace
2
```

```
    def fortranTimeStep(self, dt=0.0):
        """Takes a time step using a simple fortran module that
        implements the loop in Fortran.  """
        g = self.grid
        g.u, err = flaplace.timestep(g.u, g.dx, g.dy)
        return err
```

Thats it! Hopefully someday scipy.weave will let us do this inline and not require us to write a separate Fortran file. The Fortran code and f2py example were contributed by Pearu Peterson, the author of f2py. Anyway, using this module it takes about 0.029 seconds for a 500x500 grid per iteration! This is about a 500 fold speed increase over the original code.

f2py can also work with more modern Fortran versions. This is useful because Fortran90 has special array features that allow a more compact, nicer and faster code. Here is the same subroutine in Fortran90:

```
! File flaplace90_arrays.f90
subroutine timestep(un,m,dx,dy,error)
implicit none
!Pythonic array indices, from 0 to n-1.
real (kind=8), dimension(0:n-1,0:m-1), intent(inout) :: u
real (kind=8), intent(in) :: dx,dy
real (kind=8), intent(out) :: error
integer, intent(in) :: n,m
real (kind=8), dimension(0:n-1,0:m-1) :: diff
real (kind=8) :: dx2,dy2,dnr_inv
!f2py intent(in) :: dx,dy
!f2py intent(in,out) :: u
!f2py intent(out) :: error
!f2py intent(hide) :: n,m
dx2 = dx*dx
dy2 = dy*dy
dnr_inv = 0.5d0 / (dx2+dy2)
diff=u
u(1:n-2,1:m-2) = ((u(0:n-3,1:m-2) + u(2:n-1,1:m-2))*dy2 + &
                          (u(1:n-2,0:m-3) + u(1:n-2,
2:m-1))*dx2)*dnr_inv
error=sqrt(sum((u-diff)**2))
end subroutine
```

Remark that the operations are performed in a single line, very similar to Numpy. The compilation step is exactly the same. In a 1000x1000 grid, this code is 2.2 times faster than the fortran77 loops.

The source tarball (perfpy_2.tgz) also contains a Fortran95 subroutine using the forall construct, which has a very similar performance.

Using Pyrex

We also implemented the timeStep function in Pyrex using the code from the fast inline version. The Pyrex sources are a little longer than the weave, blitz or Fortran code since we have to expose the NumPy array structure. The basic function looks like this.

Toggle line numbers

```
def pyrexTimeStep(ndarray u, double dx, double dy):
    if chr(u.descr.type) <> "d":
        raise TypeError("Double array required")
    if u.nd <> 2:
        raise ValueError("2 dimensional array required")
    cdef int nx, ny
    cdef double dx2, dy2, dnr_inv, err
    cdef double *elem

    nx = u.dimensions[0]
    ny = u.dimensions[1]
    dx2, dy2 = dx**2, dy**2
    dnr_inv = 0.5/(dx2 + dy2)
    elem = u.data

    err = 0.0
    cdef int i, j
    cdef double *uc, *uu, *ud, *ul, *ur
    cdef double diff, tmp
    for i from 1 <= i < nx-1:
        uc = elem + i*ny + 1
        ur = elem + i*ny + 2
        ul = elem + i*ny
        uu = elem + (i+1)*ny + 1
        ud = elem + (i-1)*ny + 1

        for j from 1 <= j < ny-1:
            tmp = uc[0]
            uc[0] = ((ul[0] + ur[0])*dy2 +
                     (uu[0] + ud[0])*dx2)*dnr_inv
            diff = uc[0] - tmp
            err = err + diff*diff
            uc = uc + 1; ur = ur + 1;  ul = ul + 1
            uu = uu + 1; ud = ud + 1

    return sqrt(err)
```

The function looks long but is not too hard to write. It is also possible to write without doing the pointer arithmetic by providing convenient functions to access the array. However, the code shown above is fast. The sources provided with this article contains the complete Pyrex file and also a setup.py script to build it. Timing this version, we find that this version is as fast as the fast inlined version and takes only 0.025 seconds.

Using Matlab and Octave

We have implemented the Numeric version in Matlab and Octave (laplace.m) and run the tests on a different computer (hence the "estimate" values in the table below). We have found that no significant speed-up is obtained in Matlab, while Octave runs twice slower than NumPy. Detailed graphs can be found here.

An implementation in C++

Finally, for comparison we implemented this in simple C++ (nothing fancy) without any Python. One would expect that the C++ code would be faster but surprisingly, not by much! Given the fact that it's so easy to develop with Python, this speed reduction is not very significant.

A final comparison

Here are some timing results for a 500x500 grid for 100 iterations. Note that we also have a comparison of results of using the slow Python version along with Psyco.

Type of solution	Time taken (sec)
Python (estimate)	1500.0
Python + Psyco (estimate)	1138.0
Python + NumPy Expression	29.3
Blitz	9.5
Inline	4.3
Fast Inline	2.3
Python/Fortran	2.9
Pyrex	2.5
Matlab (estimate)	29.0
Octave (estimate)	60.0
Pure C++	2.16

This is pretty amazing considering the flexibility and power of Python.

Download the source code for this guide here: perfpy_2.tgz

View the complete Python code for the example: laplace.py

View the complete Matlab/Octave code for the example: laplace.m

PerformancePython (last edited 2009-11-20 11:09:54 by RamonCrehuet)