Lab 8: change a python program to a C++ code

Due date: April 14 , 11:59pm
(1) take the shown python program and create a C++ code (the code is available in a zip file)
(2) then compare the run time and
(3) describe the difficulties you encountered.

The C++ does not need to do the plotting, use the supplied plotting code to plot the data from the python code (julia.py) and the C++ code. Turn in your C++ program and a PDF file that contains the python graph and the C++ graph and your description about the problems/exciting things/nuisances of the translation to C++.

```
#/usr/bin/env python
#
# shows the Julia set (see wikipedia)
import sys
import math
import cmath
import random
import numpy as np

def calcz(z,c,zabsmax):
    '''
    calculates the recurrsive function z=z^2 + c; and counts how many iterations are
        needed to
    reach the maximum nitmax or reach the maximum for z (zabsmax).
    The ratio (iterations/maxiterations) [range 0..1] times 255 is returned; this
    allows to use these return values as a color scale between 0...255.
    '''
    nit = 0
    nitmax = 1000
    while abs(z) < zabsmax and nit < nitmax:
        z = z**2 + c
        nit += 1
        ratio = (float(nit) / nitmax) * 255.0
    return ratio


def julia_loop(im_width, im_height, xwidth, yheight, xmin, ymin, nitmax):
    '''
    main loop that calculates recursive function for every pixel
    '''
    print "Calculate the 2D plane..."
    zabsmax = 10.0
    c = complex(-0.1,0.65)
    julia = np.zeros((im_width, im_height))
    for ix in range(im_width):
        for iy in range(im_height):
            nit = 0
            # Map pixel position to a point in the complex plane
            z = complex(float(ix) / im_width * xwidth + xmin,
                        float(iy) / im_height * yheight + ymin)
```

```
            # Do the iterations
            julia[ix][iy] = calcz(z,c,zabsmax)
    return julia


if __name__ == '__main__':
    if len(sys.argv)>1:
        file = sys.argv[1]
    else:
        file = 'juliadata.txt'

    print("Julia set fractal generator\n")
    im_width = 1000
    im_height = 1000
    xmin,xmax = -0.5, 0.5
    xwidth = xmax-xmin
    ymin, ymax = -0.5, 0.5
    yheight = ymax - ymin
    nitmax = 1000
    zabsmax = 10.0
    title="Julia set fractal generator"
    julia = julia_loop(im_width, im_height, xwidth, yheight, xmin, ymin, nitmax)
    with open(file,'w') as f:
        f.write(str(im_width)+'\n')
        f.write(str(im_height)+'\n')
        f.write(str(xmin)+'\n')
        f.write(str(xmax)+'\n')
        f.write(str(xwidth)+'\n')
        f.write(str(ymin)+'\n')
        f.write(str(ymax)+'\n')
        f.write(str(yheight)+'\n')
        #f.write(str(nitmax)+'\n')
        #f.write(str(zabsmax)+'\n')
        for i in julia:
            for j in i:
                f.write(str(j)+'\t')
        f.write('\n')
```

Translating python to C++ is not that difficult because many constructs like for or while or if can be used very similarly. The only problem are the typing of variables and memory allocation. Write the C++ code into a file called `julia.cpp`; to compile I suggest the commandline and then use `g++ -g julia.cpp -o julia` and once it is working fo rtime testing use `g++ -O2 julia.cpp -o julia`. To test timing, compare:

```
time python julia.cpp
time julia
# to create the picture (test.pdf) [assumes the same filename as used in julia.py]
python juliaplot.py
```

Here is the fragment C++ code, that specifies the function and their arguments:

```
#import <iostream>
#include <fstream>
#include <cmath>
#include <complex>
#include <cstring>

double calcz(std::complex<double> z, std::complex<double> c, double zabsmax)
{
   // write your code here: this is very similar to the python function,
   // but every variable needs to be typed
   return ratio;
}


void julia_loop(double ***julia, long im_width, long im_height, double xwidth, double
    yheight, double xmin, double ymin, long nitmax)
{
   // use the (*julia) to allocate memory using new to get a im_width pointers to *julia
       array of doubles
   (*julia) = new double * [im_width];
  // now assign each column
  //
  // now walk through everry pixel and apply calcz()
}

int main(int argc, char **argv)
{
   // define the variables then call the function over all pixels
   julia_loop(&julia, im_width, im_height, xwidth, yheight, xmin, ymin, nitmax);
   // write out julia to a file that then is read by juliaplot.py (see zip package)
}
```