

# CTAC 2006 Workshop: Python in Scientific Computing

## Exercise 3 - Achieving Sequential Performance with Python

Ole Nielsen, Geoscience Australia and Australian National University

In this exercise we will optimise the Python program from exercise 2 by rewriting the innermost loop in C and link it into Python.

### 1 Background

Even though most of Python's primitives are powerful and fast, it is generally the case that Python loops are slow compared to their compiled C or Fortran equivalents. This is due to the interpreted nature of Python and does not present a problem unless loops are very long and/or nested. In such cases one will benefit from rewriting the innermost loop in C or Fortran and provide its functionality as a new Python function. Modules available in the Python language are often implemented as Fortran and C functions either for performance reasons or to provide a Python interface to legacy software.

#### 1.1 Identify the bottleneck

Step number one when optimising code is to identify the areas where most time is spent. It is often the case that a vast majority of time is spent within a fairly small textual segment of the code. This is also known as the 'bottleneck' and this is where the gain from optimisation will be the greatest.

Modify your program `mandel_sequential.py` from exercise 2 to have

```
kmax = 2**15  
M = N = 50
```

The larger value of `kmax` will give a finer resolution of colours but will also require a lot more work in the inner loop. Try to run the program. How long does it take?

**Question:** How long would it take to compute the set for  $M = N = 700$ , all other things being equal?

In the following we will rewrite the routine `calculate_point`, being the one where most computations occur, in C programming language and link it into the Python program.

## 1.2 Mandelbrot iteration in C

We will first write and test the routine in pure C. Create a file called `temp.c` with the following contents

```
// Define data type for complex numbers
typedef struct {
    double real;
    double imag;
} Py_complex;

// Computational function
int _calculate_point(Py_complex c, int kmax) {
    int count;
    double temp, lengthsq = 0.0;
    Py_complex z;
    z.real = 0.0; z.imag = 0.0;
    count = 0;

    while (count < kmax && lengthsq <= 4) {

        //-----
        // Your code goes here !
        //-----

        count++;
    }

    return count;
}

int main() {
    Py_complex c;
    int k, kmax = 256;

    c.real = 0.5;
    c.imag = 0.5;

    k = _calculate_point(c, kmax);

    printf ("Number of iterations for (%.2f + %.2fi) is %d\n", c.real, c.imag, k);
}
```

Explanation of the program

- The program starts with a declaration of a structure, `Py_complex`, which is used to represent a complex number: It consists of two double precision numbers, `real` and `imag` accessed through the dot notation as in `c.imag`. The name `Py_complex` is arbitrary but used here to make it easy to plug into Python in a moment.
- The function `_calculate_point` is designed to compute the Mandelbrot iteration for *one* point in the complex plane. It is incomplete in its present state, though, and will always return `kmax`.
- The main program calls the function `_calculate_point` for one point to verify that it can compile and give the correct result in one instance. Notice how the complex numbers are assigned values and how they are printed.

Compile this program with `cc temp.c -o temp.x` and run it . You should get the output

```
Number of iterations for (0.50 + 0.50i) is 256
```

which is clearly *wrong*, of course, because nothing is being computed. Recall from the test routine of exercise 2 that the number of iterations should be 5.

**Exercise:** Complete the C routine, `_calculate_point`, and verify the result.

*Hints:*

- Only modify the code in the spot indicated. It should take about four lines to complete.
- You will need to explicitly program the mathematics for computing  $z^2$ . Recall that a complex number  $z = \Re(z) + \Im(z)i$  and that  $i^2 = -1$ .
- Your code should assign the appropriate values to `z.real`, `z.imag`.
- Instead of computing the length of `z` we will compute its square ( $z\bar{z}$ ), `lengthsq`. This is why the loop uses 4 instead of 2 in the stop criterion.
- You will need to use the temporary variable `temp` for temporary storage when computing  $z * z$ .

### **1.3 Writing the Python extension**

A Python C-extension is a C program with some special 'wrapper' functions that translate external data representations back and forth from the Python representations.

Type or paste in the following simple C extension and call it `mandel_ext.c`

```
#include "Python.h"

// Computational function

//-----
// Your function _calculate_point goes here !
//-----

// Interface to Python
PyObject *calculate_point(PyObject *self, PyObject *args) {
    PyComplexObject *C; // Python Complex object
    Py_complex c;      // C Complex structure
    int kmax, count;

    // Convert Python arguments to C
    if (!PyArg_ParseTuple(args, "Oi", &C, &kmax))
        return NULL;
    c = PyComplex_AsCComplex((PyObject*) C);

    // Call underlying routine
    count = _calculate_point(c, kmax);

    // Return results as a Python object
    return Py_BuildValue("i", count);
}

// Method table for python module
static struct PyMethodDef MethodTable[] = {
    {"calculate_point", calculate_point, METH_VARARGS},
    {NULL, NULL}
};

// Module initialisation
void initmandel_ext(void){
    Py_InitModule("mandel_ext", MethodTable);
}
```

The C-extension consists of five parts

1. Inclusion of the Python header file. This gives access to a large number of functions that manipulate Python objects within C.
2. The function you wish to make available to Python. In this case the function `_calculate_point` you developed in Section 1.2.
3. Interface to Python. Here called `calculate_point` (without the trailing underscore) which is the name of the function Python will be able to call. This function reads arguments passed by Python, converts them into C variables,

calls the C function `_calculate_point` and finally converts the result into a Python object and returns it.

4. Method table. Lists the interfaces that should be available to Python. In this case there is only one, namely `calculate_point`.
5. Module initialisation. Needed to get everything started.

Part 1, 4 and 5 tend to be the same in virtually all C-extensions so the main efforts lie with the underlying computational routines and the Python interfaces.

**Exercise:** Paste in your own `_calculate_point` routine in the C-extension where indicated.

## 1.4 Compiling the C extension

### 1.4.1 Create a compiled object file

To compile, we need to include the Python libraries, set various compiler flags, link the *object file* (here `mandel_ext.o`) into what is known as a *shared object file* (`mandel_ext.so`) which can be dynamically linked into executables — in this case the Python interpreter.

A script that does this on most platforms has been included so the compilation and linking is accomplished by issuing the command

```
python compile.py mandel_ext.c
```

Feel free to look at the internal workings of this script and use it elsewhere if you wish.

## 1.5 Testing the C extension

The new module should pass the same test suite as the Python version: Modify the test program `test_calculate_point.py` and replace the line

```
from mandelbrot import calculate_point
```

with

```
from mandel_ext import calculate_point
```

signifying that we are now testing the C version instead of the Python version. Run the test suite as in exercise 2 and verify that all tests pass!

## 1.6 Using the C extension

We are now ready to use the C-extension within our Python program. To use it, simply import it like any other Python module. Modify your file `mandelbrot.py` by adding the following statement

```
from mandel_ext import calculate_point #Faster C implementation
```

at the top of your function `calculate_region` just after the inclusion of `Numeric`.

This will make the function use the C extension *instead of* the Python version of `calculate_point`.

**Exercise:** Run the unit test `test_calculate_region` and verify that it passes now that `calculate_region` is using the new computational routine.

**Exercise:** Run your program now using the C-extension and verify that it still computes the desired result by looking at the plot.

## 1.7 Measuring the speed

With any optimisation, be it parallel or sequential, there is little point in going through the work unless there is a distinct payoff in terms of speed or space savings. In this case we are looking for speed.

**Question:** How much faster is the C-extension for the problem with  $M=N=50$  stated in Section 1.1?

## 1.8 Reaping the benefits

We are now ready to reap the benefits of our efforts. Modify your program `mandel_sequential.py` to deal with the much larger problem size `kmax = 2**15` and  $M = N = 700$ . This should yield a large Mandelbrot set with fine colour resolution in about 48 seconds.

**Question:** How much faster is this compared to a pure Python implementation with  $M=N=700$  ?