Parallel Design Patterns: Exercise 3

Lawrence Mitchell

January 16, 2012

Generic fractal set computation

In this practical, you'll extend your mandelbrot set code to carry out computation of multiple different fractal sets. All the scaffolding code should be the same, the only change will be to the code used to check if a complex number is in a given set.

To achieve this generic behaviour we will use function pointers to generalise the `compute_set` function.

Function pointers in C

C allows you to define a variable which points to the address of a function. Such a variable is (for obvious reasons) termed a *function pointer*. The syntax is somewhat different from normal variable declarations, and is best described with an example.

```
 1:  int fn(float a, int b)
 2:  {
 3:      return (int)a + b;
 4:  }
 5:
 6:  int main(void)
 7:  {
 8:      int (*fn_pointer)(float, int);
 9:
10:      fn_pointer = &fn;
11:
12:      return (*fn_pointer)(1.0, 2);
13:  }
```

In lines 1-4 we define a function taking two arguments, a `float` and an `int` and returning an `int`. In line 8 we define a variable `fn_pointer` to hold a reference to this function. On line 10 we assign the address of the defined function to the function pointer, finally on line 12 is an example of how to call a function through a function pointer.

It can become slightly confusing to use function pointers in argument lists to functions. One solution is to use `typedef` to wrap the variable declaration as demonstrated in the following example.

```
typedef int (*fn_pointer)(float, int);

int fn(float a, int b)
{
    return (int)a + b;
}

int main(void)
{
    fn_pointer fp;

    fp = &fn;

    return fn(1.0, 2);
}
```

Now you can write a function which takes a function pointer as an argument more easily.

```
typedef int (*fn_pointer)(float, int);

int fn(float a, int b)
{
    return (int)a + b;
}

int fn_with_fp_arg(fn_pointer f, float a, int b)
{
    return (*f)(a, b);
}
```

Without the `typedef` this code is more complicated.

```
int fn(float a, int b)
{
    return (int)a + b;
}

int fn_with_fp_arg(int (*f)(float, int), float a, int b)
{
    return (*f)(a, b);
}
```

For a much more complete introduction to function pointers, including description of the syntax for C++ as well as C, have a read of The Function Pointer Tutorial [1].

Procedure pointers in Fortran

Fortran also has a way of specifying that an argument to a function or subroutine is a procedure, rather than a variable. The argument to the function should be specified as an interface block, rather than a variable declaration.

```
real function minimum(a, b, fn)
  ! Return the minimum of fn(x) in the interval (a,b)
  real, intent(in) :: a, b

  interface
     real function fn(x)
       real, intent(in) :: x
     end function fn
  end interface

  ...
  f = fn(x)
  ...
end function minimum
```

[1] www.newty.de/fpt/index.html

When invoking the function `minimum` the third argument should name a function with an interface that matches that in the dummy argument list. If we want to pass the name of the function around in variable, like C function pointers, we need to work a bit harder. Here's a complete example.

```
 1:  module fnptr
 2:    implicit none
 3:    abstract interface
 4:       real function fn(x)
 5:         real, intent(in) :: x
 6:       end function fn
 7:    end interface
 8:  contains
 9:    real function a(x)
10:      real, intent(in) :: x
11:      a = x*x
12:    end function a
13:  end module fnptr
14:
15:  program main
16:    use fnptr
17:    implicit none
18:    procedure(fn), pointer :: fp
19:    fp => a
20:    call sub(fp, 3.0)
21:    call sub(a, 2.0)
22:  contains
23:    subroutine sub(f, x)
24:      procedure(fn) :: f
25:      real, intent(in) :: x
26:      write(*,*)f(x)
27:    end subroutine sub
28:  end program main
```

In this example, we define a module (lines 1-13) that contains an interface for a real function, note that the `abstract` keyword only arrived in Fortran 2003. In the main program we define a procedure pointer that matches this interface (line 18). We then point the function pointer at the function `a`. In line 20 we call a subroutine with this function pointer, in the next line we call the function directly. Note that compiler support for this feature is

patchy. Things will improve in the coming years as Fortran 2003 support becomes more widespread.

The Julia set

The additional fractal set computation you should implement is computation of the Julia set. Consider a rational function $f(z)$ mapping from the complex plane to itself $f : \mathbb{C} \to \mathbb{C}$. The Julia set is the set of complex numbers z for which the repeated application of $f(z)$ is repelling. To draw the set, we will only consider entire functions, specifically, we will restrict ourselves to functions of the form $f(z) = z^2 + c$. In this case the Julia set is the boundary of the set of points which converge to infinity under repeated application of the function. Like the Mandelbrot set, we use a heuristic to decide if a point z is in the set.

1. Let $z_0 = z$
2. If $|z_0| > 2$, return 0
3. Compute $z_{n+1} = z_n^2 + c$
   (a) If $|z_{n+1}| > 2$, return $n + 1$
   (b) If $n + 1 = n_{\max}$, return $n + 1$
   (c) Else, set $n \leftarrow n + 1$ and go to 3.

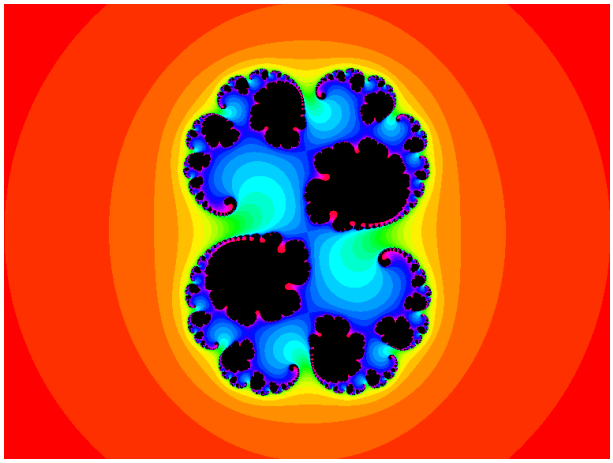

Figure 1: The Julia set for $f(z) = z^2 + 0.285 + \mathrm{i}0.01$

We colour the returned iteration numbers the same as for the Mandelbrot set (so you don't need to change the image drawing routines at all). Figure 1 shows the Julia set for $f(z) = z^2 + 0.285 + \mathrm{i}0.01$ generated with the commandline arguments `-x -2 -X 2 -y -1.5 -Y 1.5 -i 32`

Debugging problems

The most likely issue you will stumble across in this exercise is that your code for generating the set considers the image array to store the complex plane in a different order from that expected by the image writing routines. The data should be ordered slightly differently depending on whether you are in using the C or Fortran code:

C

- `image[0][0]` is the bottom left corner
- `image[max_y-1][0]` is the top left corner
- `image[max_y-1][max_x-1]` is the top right corner
- `image[0][max_x-1]` is the bottom right corner

Fortran

- `image(1,1)` is the bottom left corner
- `image(1, max_y)` is the top left corner
- `image(max_x, max_y)` is the top right corner
- `image(max_x, 1)` is the bottom right corner

The example Julia set shown in figure 1 should pick up on these problems because it is not symmetric about either the X or Y axes.