03_21_modules_n_finding_roots

Unknown Author

April 1, 2014

Part I

More on modules. Finding roots.

In [1]:
```
# Imports
import random
import math
```

1 Modules

Since we shall be writing a lot of functions now which we might use later, it is better to write them as modules. I'll write most of them on ipython notebooks so that you can see them online, but to use them as modules, you've to export them as python files (or just grab the .py file from the webpage.) Reference for this is *section 4.5 : Making modules* of *Langtangen*'s *A primer on Scientific Programming with Python.*Last time when we imported gaussian as a module, it resulted in a lot of garbage being printed because of all the script hanging around in the python file. We don't want that to happen everytime. Still we might want to run the module as a stand alone python file as we were doing till now.

The trick is to use the __`name`__ variable which equals to the module name (in this case the filename) if the file is imported as a module. If the file is run as a python code then __`name`__ takes the value __`main`__. Also function names starting with an underscore(_) are not imported. We can use them to write functions which we want to be only part of the python code in the file.We want to code this file as a module which does algorithms to find roots. Today we'll do bisection method, Newton's method and secant method. We shall write functions for them. We shall also write *hidden* functions which test these methods when the code is run as a program.

2 Finding roots

The mathematical aim of this lecture is to find roots of equations of the type

$$f(x) = 0 \tag{1}$$

for some function f. In other words, given f our aim is to find (at least) one x such that $f(x) = 0$. We did this for systems of linear equation using Gaussian elimination and you know how to do it for quadratic equations. There are a lot of algorithms which offer different degrees of efficiency. In today's lecture we'll concentrate on three easy to understand methods.

2.1 Bisection method

Assumption :

- The function is continuous.
- It is easy to find two values x_1 and x_2 such that $f(x_1)$ and $f(x_2)$ have opposite signs.

We shall input x_1 and x_2 from the user. This will also tell us that there is at least one root in the interval (x_1, x_2). The method to find the root is the following.

1. Find the mid point of x_1 and x_2; call it m.
2. If $f(m)$ is zero, we are done.
3. Otherwise look at sign of $f(m)$.
    1. If it is opposite of that of x_1, repeat the procedure for x_1 and m.
    2. Otherwise repeat the procedure with m and x_2.

Some very important remarks.

- When we compute using a computer, there will be round of errors. So while testing if $f(m) = 0$, if we write `f(m) == 0`, it might be false as computational error might cause the code for `f` to return something like 10^{-10} instead of 0. The solution is to fix an acceptible error, which we shall denote by `err` and check if `f(m) < err`.
- It is tempting to use recursion here, and we will write one version which uses recursion. But it is more efficient to not use it.

In [2]:

```python
def find_opp_signs(f, err=1E-7, max_pass=10) :
    """It helps the user to try different values of x and y
    till one find x and y such that f(x) and f(y) have different
    signs. It returns (x, y). If the user finds a root, say r, it returns (r, r)"""

    x = input("Please input a number to be used as x : ")
    fx = f(x)
    if abs(fx) < err :
        print "You found a solution."
        retpair = (x, x)
    else :
        y = input("Please input a number to be used as y : ")
        fy = f(y)
        if abs(fy) < err :
            print "You found a solution."
            retpair = (y, y)
        else :
            exit = False
            no_of_passes = 0
            while fx * fy > 0 and not exit:
                # This means they have the same sign!
                no_of_passes += 1
                y = input("Your y did not work. Try again : ")
                fy = f(y)
                if abs(fy) < err :
                    print "You found a solution."
                    retpair = (y, y)
                    exit = True
                elif no_of_passes > max_pass :
                    print "Maximum no of trials exceeded."
                    retpair = None
                    exit = True
            if not exit :
                retpair = (x, y)
    return retpair
```

In [3]:
```
def _test_find_opp_signs() :
    print "-"*40
    print "Testing find_opp_signs."
    r = find_opp_signs(lambda x : x*x - 4)
    if r == None :
        print "Nothing found."
    else :
        (x, y) = r
        if x == y :
            print "You found a solution,", x, "."
        else :
            print "(x, y) =", (x, y), ", f(x) =", x*x - 4, ", f(y) =", y*y - 4, "."
```

In [4]:
```
def auto_find_opp_signs(f, err, no_of_loops) :
    random.seed()
    x = random.random()
    fx = f(x)
    y = x
    fy = f(y)
    exitloop = False
    no_passes = 0
    if abs(fy) < err :
        print "Found a zero."
        retpair = (y, y)
    else :
        while fx*fy > 0 and not exitloop :
            N = int(1/random.random())
            N += 2
            y = 2 * N * random.random() - N
            fy = f(y)
            no_passes += 1
            if abs(fy) < err :
                print "Found a zero."
                retpair = (y, y)
                exitloop = True
            elif no_passes > no_of_loops :
                print "Maximum no of loops exceeded."
                retpair = None
                exitloop = True
        if not exitloop :
            retpair = (x, y)
    return retpair
```

In [5]:
```
def _test_auto_find_opp_signs() :
    print "-"*40
    print "Testing auto_find_opp_signs"
    f = lambda x : x*x + 2*x - 3
    (x, y) = auto_find_opp_signs(f, 1E-07, 1000)
    print "(x, y) = ", (x, y), ", f(x) = ", f(x), ", f(y) = ", f(y),"."
```

Now we wrote enough codes to find points whose images have opposite signs. Now let us code the bisection method.

In [6]:
```
def bisection(f, err=1E-7, trials=10000) :
    # First find x_1 and x_2. We use auto first, and then if necessary manual.
    pair = auto_find_opp_signs(f, err, trials)
    if pair == None :
        # Try manual
        can_exit = False
        while not can_exit :
            pair = find_opp_signs(f, err, 10)
            if pair == None :
                ask = raw_input("Do you want to continue? (y/n)")
                if ask == "n" :
                    can_exit = True
            else :
                can_exit = True
    if pair == None :
        print "Sorry, I can't help you."
```

```
            m = None
        else :
            (x1, x2) = pair
            m = (x1 + x2)/2.0
            while abs(f(m)) >= err/10 :
                if f(x1) * f(m) < 0 :
                    x2 = m
                else :
                    x1 = m
                m = (x1 + x2) / 2.0
                # print m, f(m)
        return m
```

In [7]:

```
def _test_bisection() :
    print "-"*40
    print "Testing bisection."
    f = lambda x : x*x*x - 3*x*x + 3*x - 1
    soln = bisection(f, 1e-30, 10000)
    if soln == None :
        print "OK! No solutions."
    else :
        print "Soln of x^3 - 3x^2 + 3x - 1 = 0 is", soln
        print "and f(x) is", f(soln), "."
```

In [8]:

```
if __name__ == '__main__' :
    _test_find_opp_signs()
    _test_auto_find_opp_signs()
    _test_bisection()
```

```
----------------------------------------
Testing find_opp_signs.
Please input a number to be used as x : 9
Please input a number to be used as y : 3
Your y did not work. Try again : 5
Your y did not work. Try again : 9
Your y did not work. Try again : 12
Your y did not work. Try again : 100000
Your y did not work. Try again : 0
(x, y) = (9, 0) , f(x) = 77 , f(y) = -4 .
----------------------------------------
Testing auto_find_opp_signs
(x, y) =  (0.42050763018381, 1.9360129377637882) , f(x) =
-1.98215807259 , f(y) =  4.62017197072 .
----------------------------------------
Testing bisection.
Soln of xˆ3 - 3xˆ2 + 3x - 1 = 0 is 1.00000351881
and f(x) is 0.0 .
```

3 Newton's method

The methood

To use Newton's method we should know the following :

1. The function, f, whose root we want to find.
2. The derivative of the function f'.
3. An initial point x_0.
4. An error limit, `err`, such that we can consider x to be a zero whenever $f(x) <$`err`.

5. Note that sometimes this algorithm just diverges leading to a possibly infinite loop. To avoid that we also set an upper limit on the number of iterations `max_iterations`.

Now the algorithm goes like this

1. Compute $f(x_0)$ and $f'(x_0)$.
2. If $f(x_0) <$ `err` , return x_0 to be the root.
3. If $f'(x_0) = 0$ return an error saying that the algorithim cannot continue.
4. Otherwise, compute x_1 using the formula $x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$.
5. Iterate with x_1 playing the role of x_0.
6. Continue till some x_n satisfies $f(x_n) <$ `err` **or** $n \geq$ `max_iterations`.

In [9]:
```
def newton(f, fp, x, err=1.0E-07, max_iterations=10000) :
    """This code tries to find a root of f using Newton's method.
    fp is the derivative of f
    x is the starting point.
    err is the error below which we consider numbers to be zero.
    max_iterations is the total number of iterations to do before
    giving up."""

    iterations_done = 0
    fx = f(x)
    fpx = fp(x)
    if abs(fpx) < err :
        print "Value of f' is too small at %g. Cannot continue." % x
        retval = None
    while abs(fx) >= err and abs(fpx) >= err and iterations_done < max_iterations :
        iterations_done += 1
        x = x - fx/fpx
        fx = f(x)
        fpx = fp(x)
        # print x, fx, fpx
    if abs(fx) < err :
        retval = x
    elif abs(fpx) < err :
        print "Value of f' is too small at %g. Cannot continue." % x
        retval = None
    else :
        print "Maximum iterations reached : %d" % iterations_done
        retval = None
    return retval
```

In [10]:
```
def _myquad(x) :
    return x*x - 4*x + 3

def _dermyquad(x) :
    return 2*x - 4

def _test_newton() :
    soln1 = newton(_myquad, _dermyquad, 0)
    if soln1 == None :
        print "Could not find solution to the quadratic equation."
    else :
        print "Root of the quadratic equation is %g." % soln1

    soln2 = newton(lambda x : math.exp(- x*x/1E50) + 1, lambda x : -2 * x * math.exp(-
    if soln2 == None :
        print "Could not find solution for e^(-x^2/10^50) + 1 = 0."
    else :
        print "Root of the e^(-x^2/10^50) + 1 map is %g." % soln2

    soln3 = newton(lambda x : x*x + 1, lambda x : 2*x, 10)
    if soln3 == None :
        print "Cound not find solution to x*x + 1 = 0."
    else :
        print "Solution for x*x + 1 = 0 is %g." % soln3
```

In [11]:
```
if __name__ == '__main__' :
    _test_newton()
```
```
Root of the quadratic equation is 1.
Maximum iterations reached : 10000
Could not find solution for e^(-x^2/10^50) + 1 = 0.
Value of f' is too small at 0. Cannot continue.
Cound not find solution to x*x + 1 = 0.
```