week02-03–root-finding

January 21, 2024

1 George McNinch Math 87 - Spring 2024

2 Week 2: Root Finding

3 Overview

We are often interesting in finding solutions to (non-linear) equations $f(x) = 0$.

Here we describe various methods for finding such solutions under assumptions and requirements.

By a root of f, we just mean a real number x_0 such that $f(x_0) = 0$.

Of course, for some very special functions f, we have formulas for roots. For example, if f is a quadratic polynomial, say $f(x) = ax^2 + bx + c$ for real numbers a, b, c, then there are in general two roots, given by the *quadratic formula*

$$x_0 = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

(Of course, these roots are only real numbers in case of $b^2 - 4ac \geq 0$).

But such a formula is far too much to ask for, in general!

We describe here some algorithmic methods for approximating roots of "nice enough" functions. These methods are less precise than, say, the quadratic formula, but they are more generally applicable.

4 Bisection - overview

The bisection algorithm permits one to approximate a root of a continuous function f, provided that one knows points $x_L < x_R$ in the domain of f for which the function values $f(x_L)$ and $f(x_R)$ are non-zero and have opposite signs. The algorithm then returns an approximate root in the interval (x_L, x_R).

Of course, for a continuous f the *intermediate value theorem* implies that there is at least one root x_0 of f in the interval (xL, xR).

To find a root, the algorithm iteratively divides the interval $[x_L, x_R]$ into two sub-intervals by introducing the midpoint $x_C = \dfrac{x_L + x_R}{2}$. It examines the signs of the values $f(x_L)$, $f(x_C)$ and $f(x_R)$ and discards the interval on which the sign doesn't change. (Of course, if $f(x_C)$ happens to be zero, that is the root!)

So for example, if $f(x_L)$ and $f(x_C)$ differ in sign, the procedure is repeated on this smaller interval $[x_L, x_C]$.

One way of looking at the "theory" underlying the use of this algorithm is the following: writing x_N for the approximate solution returned by the algorithm after N iterations, one knows that the limit

$$\lim_{N\to\infty} x_N$$

exists and is a solution to $f(x) = 0$ – in words: the estimates converge to a solution.

The `python` library `scipy` has an implementation of the bisection algorithm, which we can use.

This implementation is found in the `scipy.optimization` library, and the function has the following specification:

`{python} scipy.optimize.bisect(f,a,b,args=(), xtol=2e-12, rtol=8.881784197001252e-16, maxiter=100, full_output=False, disp=True)`

Here `f` is the function in question, and `a` and `b` are values bracketing some root of `f`.

Morally, the argument `rtol` indicates the desired tolerance – thus the function should return a value `x` for which `|f(x)| < rtol`. In practice, things are a bit more complicated (read the docs when required…!)

Also:

> If convergence is not achieved in `maxiter` iterations, an error is raised. Must be >= 0.

5 Example

For example, we can use `bisect` to approximate the roots of

$$f(x) = x^2 - x - 1.$$

Recall that we actually know already - from the quadratic formula - that those roots are

$$\frac{1 \pm \sqrt{5}}{2}.$$

Let's try to find them using bisection.

We first bracket by the interval $[1, 2]$ and then by the interval $[-2, 0]$:

[5]:
```python
import numpy as np
from scipy.optimize import bisect

def f(x):
    return x**2 - x - 1

## lets make a list of the solutions

approx_sol = np.array([bisect(f,1,2),
```

```
                              bisect(f,-2,0)])

approx_sol
```

[5]:
```
array([ 1.61803399, -0.61803399])
```

[3]:
```
sol_via_radicals = np.array([(1+np.sqrt(5))/2,
                             (1-np.sqrt(5))/2 ])

report = "\n".join([f"bisection solutions: {approx_sol}",
                    f"via radicals:        {sol_via_radicals}",
                    f"difference:          {approx_sol-sol_via_radicals}"])

print(report)
```

```
bisection solutions: [ 1.61803399 -0.61803399]
via radicals:        [ 1.61803399 -0.61803399]
difference:          [-1.17417187e-12  1.17417187e-12]
```

Question: what does this `bisect`function do if `f(a)` and `f(b)` have the same sign?

6 Example

We can estimate zeros of the sin function - here we get an approximation to π, since we happen to know that $\sin(1) > 0$, $\sin(4) < 0$, and π is the unique root of $\sin(x) = 0$ between 1 and 4:

[6]:
```
def g(x): return np.sin(x)

bisect(g,1,4)
```

[6]:
```
3.1415926535887593
```

Question: How does this solution compare with the value of pi stored by `numpy`?

(Compare with `np.pi`)

7 Example

And we can estimate the transcendental number $e = \exp(1)$ e.g. by finding roots of the function $f(x) = 1 - \ln(x)$:

(**Question**: try comparing the answer with `np.exp(1)`).

[7]:
```
def h(x):
    return 1 - np.log(x)

bisect(h,1,3)
```

[7]:
```
2.7182818284582027
```

Here are some slightly more sophisticated methods of approximating roots:

8 Secant Method

You can read the wikipedia description of the secant method here.

> The secant method is a root-finding algorithm that uses a succession of roots of secant lines to better approximate a root of a function f.

9 Newton's method

And here is the wikipedia description of Newton's method.

> it is a root-finding algorithm which produces successively better approximations to the roots (or zeroes) of a real-valued function. The most basic version starts with a single-variable function f defined for a real variable x, the function's derivative f', and an initial guess x_0 for a root of f.

Let's quickly summarize the simplest form of Newton's method:

If the function is sufficiently "nice" and if the initial guess x_0 is close enough to a root, then

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

is a better approximation of the root than x_0. Notice that x_1 is the x-coordinate of the point of intersection of the x-axis with the tangent line to f at $(x_0, f(x_0))$.

The process is then iterated: for $n \geq 2$, we set

$$x_n = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})}.$$

Under favorable circumstances, $\lim_{n\to\infty} x_n$ is a root of f.

The `scipy` library makes both the secant method and Newton's method available via scipy.optimize.newton

```
{python} scipy.optimize.newton(func, x0,                    fprime=None,
args=(),                       tol=1.48e-08,                    maxiter=50,
fprime2=None,                  x1=None,                         rtol=0.0,
full_output=False,             disp=True)
```

The mandatory arguments to this function are `func` and the initial guess `x0`. If the derivative `fprime` is given, this function uses Newton's method to approximate a root.

If the value of `fprime2` is `None` - the default value – then this function uses either Newton's method or the secant method to approximate a root of f. (If a second derivative `fprime2` is given, then Halley's method is used).

Assuming `fprime2 = None`, whether to use Newton's method or the secant method is determined by the value of `fprime`.

If the value of fprime is None (the default value), then this function uses the secant method to approximate a root of f. It then requires a value other than None for the x1 argument (since the secant method requires *two* initial values).

If fprime is given, this function uses Newton's method to approximate a root.

Let's repeat the preceding examples:

9.1 Example

- $f(x) = x^2 - x - 1$.

[8]:
```
from scipy.optimize import newton

def f(x):
    return x**2 - x - 1

## secant method
sec=[newton(f,1,x1=2),newton(f,-1,x1=-2)]

## Newton's method
def fprime(x):
    return 2*x - 1

newt=[newton(f,1,fprime),newton(f,-1,fprime)]

report = "\n".join([f"secant {sec}",
                    f"newton {newt}",])
print(report)
```

```
secant [1.618033988749909, -0.6180339887498949]
newton [1.618033988749895, -0.6180339887498948]
```

10 Example

π via sin

[9]:
```
## use the secant method with x0 = 1 and x1 = 4
sec_pi = newton(np.sin,1.0,x1=4.0)

## use newton's method with x0=1
newt_pi = newton(np.sin,2,fprime=np.cos)

report = "\n".join([f"secant: {sec_pi}",
                    f"newton: {newt_pi}"])

print(report)
```

```
secant: 3.141592653589793
newton: 3.141592653589793
```

11 Example:

e via $h(x) = 1 - \ln(x)$.

[10]:
```
def h(x):
    return 1 - np.log(x)

def hprime(x):
    return -1/x

e_secant = newton(h,2,x1=3)
e_newt   = newton(h,3,fprime=hprime)


report = "\n".join([f"secant: {e_secant}",
                    f"newton: {e_newt}"])

print(report)
```

```
secant: 2.718281828459045
newton: 2.718281828459045
```

Question: what was the role of x_0 and x_1 in the above secant method examples? and what was the role of x_0 in the above newton-method examples?

See what happens when you vary x_0 in the computation of `newt_pi` above.

See what happens when you give `newton` an incorrect first derivative.

12 Modeling exaample

A large population of N people need to be tested for a disease. In order to reduce the costs of testing, a grouping strategy is proposed: Take blood samples from each person in a group of x people. Divide each sample in half and mix one-half of each person's sample into one mixture. Test the mixture. If it is negative, then we know that all x people in the group are negative. If it is positive, then at least one person in that group is positive, so test the other half of each person's sample. What value of x minimizes the total number of tests that needs to be done?

Variables:

- N = total population
- x = group size
- q = probability of one individual testing negative
- T = total number of tests
- T_g = total number of group tests
- T_i = expected number of individual tests
- $T = T_g + T_i$

The number of group tests is just the population/group size, $T_g = N/x$.

For T_i we have N/x groups of x people and the probability of all people in the group being negative is q^x. Thus, the probability of one person in the group testing positive is $1 - q^x$. If this happens, we have to do x tests! So...

$$T_i = \frac{N}{x}\left[(1-q^x)x\right] = N\left(1-q^x\right).$$

and thus

$$T = T_i + T_g = \frac{N}{x} + N(1-q^x) = N(\frac{1}{x} + 1 - q^x).$$

To find the value of x that yields the minimum number of required tests, we need to solve the equation $\dfrac{dT}{dx} = 0$.

Well,

$$\frac{dT}{dx} = N\left(\frac{-1}{x^2} - q^x \ln q\right).$$

Since q represents a *probability*, we have $0 < q < 1$. In particular, $\ln(q) < 0$. Thus in order that $\dfrac{dT}{dx} = 0$. we must have

$$g(x) = \frac{-1}{x^2} - (\ln q)q^x = 0.$$

It is not easy to directly solve the equation $g(x) = 0$. So we will apply Newton's method. For this, we need to know $g'(x)$ as well; it is

$$g'(x) = \frac{2}{x^3} - (\ln q)^2 q^x.$$

[11]:
```
import numpy as np
from functools import partial

def g(q,x):
    return (-1/x**2) - np.log(q)* q**x

def gprime(q,x):
    return 2/x**3 - (np.log(q))**2 * q**x

q_values = [0.7, 0.8, 0.9, 0.95, 0.99, 0.999, 0.9999]

## note that partial(g,q) returns the function given by h(x) = g(q,x)
## in other words, we "partially evaluate" the function g(q,x) to get a␣
↪function
## only of x.

def newt(q):
    return newton(partial(g,q),2,fprime=partial(gprime,q))
```

```python
# the following code returns a list of pairs (q,newt(q))
# where q runs through the list q_values.
# Here, newt(q) is the solution to g(q,x) = 0 obtained from Newton's method
# (with x0 = 2).

list(map(lambda x: (x,newt(x)),q_values))
```

[11]:
```
[(0.7, 2.719531322598942),
 (0.8, 2.9381695580526563),
 (0.9, 3.7545775568830564),
 (0.95, 5.02238523178711),
 (0.99, 10.516237295014893),
 (0.999, 32.12707425945638),
 (0.9999, 100.5012836847976)]
```