Lecture 8: Root Finding Methods

Cheng Lu

Overview

- Root Finding Methods
    - Bisection Method
    - Newton-Raphson Method
- Optimization Methods
    - Gradient Descent
    - Newton's Method for Optimization

Bisection Method

Bisection method is a root finding method for continuous functions.

Intermediate Value Theorem

If function f is a continuous on $[a, b]$, then it takes any value between $f(a)$ and $f(b)$ in the interval.

In other words, for continuous function f, if $f(a)$ and $f(b)$ has opposite sign, then there exists $r \in [a, b]$ such that $f(r) = 0$.

Example

Let $f(x) = x^2 - 2$, since $f(-2) = 2$ and $f(-1) = -1$, then there is $r \in [-2, -1]$, such that $f(r) = 0$.

Using the given theorem, people construct bisection method to find root for continuous functions, where the root is the r given above.

Bisection Method

- For a given continuous function f, if we know $f(a)$ and $f(b)$ have opposite sign, then one of them is positive, another is negative
- If we let $c = \frac{a+b}{2}$, then $f(c)$ can be positive, negative, or 0
  - If $f(c) = 0$, then c is the root, otherwise either $f(a)$ and $f(c)$ have opposite sign, or $f(b)$ and $f(c)$ have opposite sign
  - If $f(a)$ and $f(c)$ have opposite sign, then a root is inside $[a, c]$, we can let the upper bound b equals to c, then a root is inside the new $[a, b]$
  - If $f(b)$ and $f(c)$ have opposite sign, then a root is inside $[c, b]$, we can let the lower bound a equals to c, then a root is inside the new $[a, b]$
- Idea: repeat the above steps to decrease the length of the interval $[a, b]$ until the length is small enough

Bisection Method

Algorithm

Input: function f, real numbers a and b.

- Step 1: Calculate $c \leftarrow \frac{a+b}{2}$
- Step 2: Calculate $f(c)$
- Step 3: If $f(c) = 0$ or $b - a$ is small enough, then return c as the root and stop the algorithm
- Step 4: If $f(a)$ and $f(c)$ have opposite sign, then let $b \leftarrow c$ and go to step 1. Otherwise $f(b)$ and $f(c)$ must have opposite sign, let $a \leftarrow c$ and go to step 1

Bisection Method

Pseudocode in Scratch Paper

```
f(x) <- function of x: x^2 - 2
a <- -2
b <- -1
tol <- 0.001 # tolerance, a small number
while true
    c <- (a+b)/2
    if f(c) = 0 or b - a < tol
        return c and stop
    end if
    if f(a)*f(c) < 0
        b <- c
    else
        a <- c
    end
end while
```

Bisection Method

Implementation in R

```
> f <- function(x){
+   x^2 - 2
+ }
> a <- -2
> b <- -1
> tol <- 0.001 # tolerence, a small number
> while(TRUE){
+   c <- (a+b)/2
+   if(f(c) == 0 || b - a < tol){
+     break
+   }
+   if(f(a)*f(c) < 0){
+     b <- c
+   }else{
+     a <- c
+   }
+ }
> print(paste("Root of f(x) is",  c))
[1] "Root of f(x) is -1.41455078125"
```

Bisection Method

After we implement the algorithm in R, we can write it into a function: Let the input variables as the inputs of the function.

Example

```
> bisection <- function(f, a, b, tol = 0.001){
+   while(TRUE){
+     c <- (a+b)/2
+     if(f(c) == 0 || b - a < tol){
+       break
+     }
+     if(f(a)*f(c) < 0){
+       b <- c
+     }else{
+       a <- c
+     }
+   }
+   print(paste("Root of f(x) is",  c))
+ }
> bisection(f, -2, -1)
[1] "Root of f(x) is -1.41455078125"
```

Bisection Method

The algorithm may have problems in practice, for example

- 1. The "while" loop may takes too many iterations to converge when the tolerance is too small
- 2. If we input $a = -1, b = -2$, then the algorithm converge early, since $b - a = -1 < 0.001$
- 3. If $f(a)$ and $f(b)$ have the same sign or equals to **NA**, the function may not return a root of f
- 4. Function f was evaluated multiple times, not efficient
- 5. The function doesn't return the number

Then we need to modify the code to solve the problems

- 1. Set maximum number of iterations $N.max$, and let the algorithm stop when $N.max$ is reached
- 2. Change $b - a < tol$ to $|b - a| < tol$
- 3. Use **warning()** or **stop()** function to notify the user when $f(a)$ and $f(b)$ have the same sign, or just return a **NA**
- 4. Save $f(a)$ and $f(c)$ in variables
- 5. Let the function return the number rather than print it

Bisection Method

Example

```
> bisection.new <- function(f, a, b, tol = 0.001, N.max = 100){
+   f.a <- f(a)
+   f.b <- f(b)
+   if(f.a*f.b > 0){
+     warning("f(a) and f(b) have same sign, output may not be a root")
+   }else if(f.a == 0){
+     return(a)
+   }else if(f.b == 0){
+     return(b)
+   }else if(is.na(f.a*f.b)){
+     return(NA)
+   }
```

Bisection Method

Example Continued

```
+   for(n in 1:N.max){
+     c <- (a+b)/2
+     f.c <- f(c)
+     if(f.c == 0 || abs(b - a) < tol){
+       break
+     }
+     if(f.a*f.c < 0){
+       b <- c
+       f.b <- f.c
+     }else{
+       a <- c
+       f.a <- f.c
+     }
+   }
+   return(c)
+ }
```

Bisection Method

Example

```
> bisection.new(f, -1, -2)
[1] -1.414551
> bisection.new(f, -1, 1)
[1] 0.9995117
Warning message:
In bisection.new(f, -1, 1) :
  f(a) and f(b) have same sign, output may not be a root
```

With the modification, the algorithm becomes more robust.

Bisection Method

We can review how we design the bisection method:

- 1. Define the Problem: Find the root for a function
- 2. Build the model from methodologies: Intermediate Value Theorem
- 3. Specifies the steps for the algorithm
- 4. Write down pseudocode in scratch paper
- 5. Implement the algorithm in R
- 6. Write it into a function and test it with different inputs
- 7. Modify the code to handle bad inputs and extreme cases, and improve efficiency

These steps can be used to design other algorithms. But some of the steps may not necessary when you are familiar with it.

Newton-Raphson Method

Suppose we want to find the root of f, we can try to find the root of its first order approximation at each step. Suppose f is differentiable, then

$$f(x_0 + h) = f(x_0) + hf'(x_0) + o(h)$$

where $o(h)$ denotes the higher order infinitesimal term of h, i.e. $\lim_{h\to 0} \frac{o(h)}{h} = 0$.
If we let

$$f(x_0 + h) \approx f(x_0) + hf'(x_0) = 0$$

Then we can solve for h that

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

So the new point is given by

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

Newton-Raphson Method

The idea is simple, given a initial point x_0, then iteratively compute the new point $x_1 \leftarrow x_0 - \frac{f(x_0)}{f'(x_0)}$ and let the old point $x_0 \leftarrow x_1$ until the value doesn't change too much.

Algorithm

Input: function f, derivative f', initial value x_0

- Step 1: Calculate $x_1 \leftarrow x_0 - \frac{f(x_0)}{f'(x_0)}$
- Step 2: If $|x_1 - x_0|$ is small enough, return x_1 as the root and stop the algorithm
- Step 3: Let $x_0 \leftarrow x_1$ and go to step 1

With the experience of bisection method, we may get familiar with the procedure for constructing the algorithm, then we can directly write down the function.

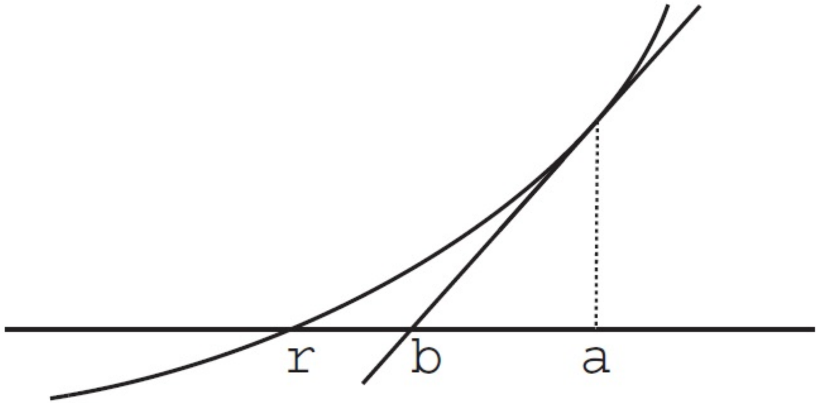
Newton-Raphson Method

Example

```
> Newton_Raphson <- function(f, df, x0, tol = 0.001, N.max = 100){
+   for (n in 1:N.max) {
+     x1 <- x0 - f(x0)/df(x0)
+     if(abs(x1 - x0) < tol){
+       break
+     }
+     x0 <- x1
+   }
+   return(x1)
+ }
> f <- function(x){x^2 - 2}
> df <- function(x) 2*x
> Newton_Raphson(f, df, -1)
[1] -1.414214
```

Newton-Raphson Method

Newton-Raphson method has geometric interpretation: The new point x_1 (b below) is the intersection of the tangent line at the old point x_0 (a below) and the x axis.



Newton-Raphson Method

Example

Suppose that a 2-year bond with a principal of \$100 provides coupons at the rate of 6% per annum semiannually. If the bond price is \$98.39, what is the yield of the bond? (See Hull (2003)[a])

[a]Hull, John C. Options futures and other derivatives. Pearson Education India, 2003.

The yield of bond is implicitly defined by equation

$$3e^{-0.5y} + 3e^{-y} + 3e^{-1.5y} + 103e^{-2y} = 98.39$$

Then we can define

- $f(y) = 3e^{-0.5y} + 3e^{-y} + 3e^{-1.5y} + 103e^{-2y} - 98.39$
- $f'(y) = -1.5e^{-0.5y} - 3e^{-y} - 4.5e^{-1.5y} - 206e^{-2y}$

And the yield y is just the root of f.

Newton-Raphson Method

Then we can calculate the yield y using the root finding methods

Example

```
> f <- function(y) 3*exp(-0.5*y) + 3*exp(-1*y) +
+   3*exp(-1.5*y) + 103*exp(-2*y) - 98.39
> df <- function(y) -1.5*exp(-0.5*y) + -3*exp(-1*y) +
+   -4.5*exp(-1.5*y) + -206*exp(-2*y)
> Newton_Raphson(f, df, 0.02)
[1] 0.06759816
> bisection.new(f, 0, 1) # find root of "f" in [0,1]
[1] 0.06787109
```

Newton-Raphson Method

We can also use **uniroot()** function to find the root as well, and **str()** function gives a compactly display of the output.

Example

```
> uniroot(f, c(0,1))$root
[1] 0.06762546
> str(uniroot(f, c(0, 1))) #find root of "f" in [0,1]
List of 5
 $ root      : num 0.0676
 $ f.root    : num -0.00514
 $ iter      : int 4
 $ init.it   : int NA
 $ estim.prec: num 6.1e-05
```

Gradient Descent

Gradient

For a univariate differentiable function f, we know its derivative is given by $f'(x) = \frac{df}{dx}$. Let f be multivariate differntiable function, let $x = (x_1, x_2, \ldots, x_n)$, then $f(x) = f(x_1, x_2, \ldots, x_n)$, and the gradient of f is given by

$$\nabla f(x) = \nabla_x f(x) = \frac{\partial f}{\partial x} = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n} \right)$$

Suppose we want to find the minimum of function f, we can try to find the minimum of its first order approximation at each step. Suppose f is differentiable, then

$$f(x_0 + h) = f(x_0) + \langle \nabla f(x_0), h \rangle + o(\|h\|)$$

where the inner product $\langle \nabla f(x_0), h \rangle = \nabla^T f(x_0) h = \sum_{i=1}^{n} \frac{\partial f}{\partial x_i} h_i$, and $\|h\| = \sqrt{\langle h, h \rangle}$.

Gradient Descent

The first order approximation is thus given by

$$f(x_0 + h) \approx f(x_0) + \langle \nabla f(x_0), h \rangle$$

But if we try to minimize it with respect to h, the result may not bounded, so we need to add a constant that the norm of h is less than or equal to some constant C.

$$\min_h \langle \nabla f(x_0), h \rangle$$
$$\text{subject to} : \|h\| \leq C$$

We eliminate $f(x_0)$ since it is a constant.

Gradient Descent

Apply Cauchy-Schwartz inequality, we have

$$|\langle \nabla f(x_0), h\rangle| \leq \|\nabla f(x_0)\|\|h\| \leq C\|\nabla f(x_0)\| = C\left\langle \nabla f(x_0), \frac{\nabla f(x_0)}{\|\nabla f(x_0)\|}\right\rangle$$

then

$$\langle \nabla f(x_0), h\rangle \geq \left\langle \nabla f(x_0), -C\frac{\nabla f(x_0)}{\|\nabla f(x_0)\|}\right\rangle$$

We find the direction of steepest descent of f at x_0, which is along to its negative gradient $h^* = -\alpha\nabla f(x_0), \alpha > 0$. So, in each step, we should update

$$x_1 \leftarrow x_0 - \alpha\nabla f(x_0)$$

until the difference $\|x_1 - x_0\|$ is small enough, where the coefficient α is called "step size" or "learning rate". If α is too large, the algorithm may not converge, if α is too small, the algorithm may converge too slow. So, choosing α is a very difficult task.

Gradient Descent

Suppose we are going to minimize the function $f(x) = x^2 - 10x + 3$, then $f'(x) = 2x - 10$. If we choose $\alpha = 1$, then for any start point x_0, we have

$$
\begin{gathered}
x_1 = x_0 - 1 * (2x_0 - 10) = -x_0 + 10 \\
x_2 = x_1 - 1 * (2x_1 - 10) = (-x_0 + 10) - (2(-x_0 + 10) - 10) = x_0 \\
x_3 = -x_0 + 10 \\
\cdots
\end{gathered}
$$

Then the new point will oscillate between x_0 and $-x_0 + 10$ no matter what initial point x_0 we choose, and the algorithm will not converge. This means $\alpha = 1$ is too large for the current function f.

Skip "specifying the steps for algorithm", "writing down the pseudocode", and "implementing the algorithm in R", we directly write the function.

Gradient Descent

Example

```
> gradient_descent <- function(df, x0, alpha = 0.2,
+                               tol = 0.0001, N.max = 100){
+   for (n in 1:N.max) {
+     x1 <- x0 - alpha*df(x0)
+     if(sqrt(sum((x1 - x0)^2)) < tol){
+       break
+     }
+     x0 <- x1
+   }
+   return(x1)
+ }
> df <- function(x) 2*x - 10
> gradient_descent(df, 6)
[1] 5.000102
```

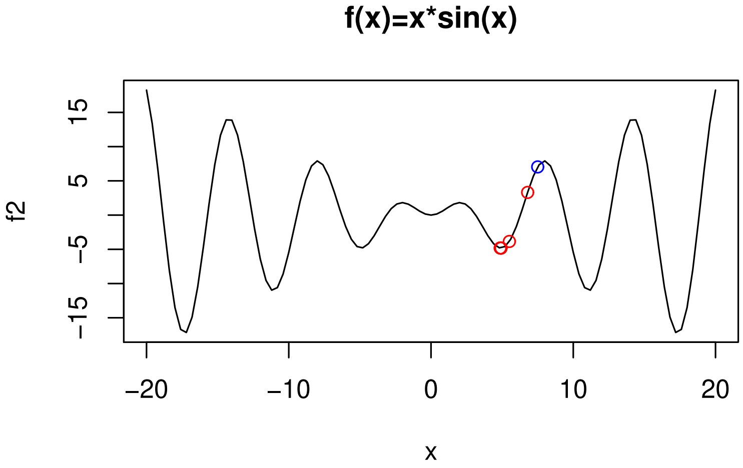
Gradient Descent

Example: Local minimum

```
> f2 <- function(x) x*sin(x)
> df2 <- function(x) sin(x) + x*cos(x)
> plot(f2, xlim = c(-20,20), main = "f(x)=x*sin(x)") # draw function
> x0 <- 7.5 # initial point
> alpha <- 0.2 # learning rate
> N.max <- 100 # max number of iteration
> tol <- 0.0001 # tolerance
> points(x0, f2(x0), col = "blue") # draw initial point with blue color
> for (n in 1:N.max) {
+   x1 <- x0 - alpha*df2(x0)
+   points(x1, f2(x1), col = "red") # draw new points with red color
+   if(abs(x1 - x0) < tol){
+     break
+   }
+   x0 <- x1
+ }
```

Gradient Descent

When we start at $x_0 = 7.5$ (the blue point), the algorithm converge to a local minimum point.



Newton's Method for Optimization

In each step of gradient descent method, we try to minimize the first order approximation of objective function f, how about the second order approximation of f?

$$f(x_0 + h) \approx f(x_0) + \langle \nabla f(x_0), h \rangle + \frac{1}{2} \langle h, \nabla^2 f(x_0) h \rangle$$

where $\nabla^2 f(x_0)$ is the Hessian of f at x_0. Then we can define the minimization problem

$$\min_h \langle \nabla f(x_0), h \rangle + \frac{1}{2} \langle h, \nabla^2 f(x_0) h \rangle$$

When $\nabla^2 f(x_0)$ is positive definite, the solution is given by

$$h = -[\nabla^2 f(x_0)]^{-1} \nabla f(x_0)$$

Newton's Method for Optimization

So, in each step, we need to update

$$x_1 \leftarrow x_0 - \alpha[\nabla^2 f(x_0)]^{-1}\nabla f(x_0)$$

where α is usually set to be 1.
Since it minimize the second order approximation of f, when f itself is a quadratic function (with positive definite Hessian), Newton's method for optimization will converge in one step no matter what initial point x_0 we choose.

Newton's Method for Optimization

Example

```
> Newton_optim <- function(df, d2f, x0, alpha = 1,
+                          tol = 0.0001, N.max = 100){
+   for (n in 1:N.max) {
+     x1 <- x0 - alpha*solve(d2f(x0), df(x0))
+     if(sqrt(sum((x1 - x0)^2)) < tol){
+       break
+     }
+     x0 <- x1
+   }
+   return(x1)
+ }
> df <- function(x) 2*x - 10
> d2f <- function(x) 2
> Newton_optim(df, d2f, 6)
[1] 5
```