MAT2630

Boyan Kostadinov

August 19, 2014

Contents

Floating-Point Anomalies

The finite precision arithmetic that digital computers are using leads to numerical errors and anomalies simply because computers can store real numbers with finitely many digits, and irrational numbers require infinitely many digits to be represented exactly. Only integers and rational numbers whose denominator is a power of 2 have a finite binary representation and can therefore be stored in the computer memory exactly. All other numbers are stored with some finite precision that leads to numerical errors.

For example, after simplifying, we get the following identity:

$$\frac{(x+y)^2 - 2xy - y^2}{x^2} = 1$$

However, the numerical result could be quite different from 1 when x is very small and y is very large, because then we have to divide two very small numbers and since they both come with numerical errors, the result is a complete nonsense:

```
x=1e-4
y=1e4
((x+y)^2 - 2*x*y - y^2)/(x^2) # dividing two very small numbers leads to anomalies
```

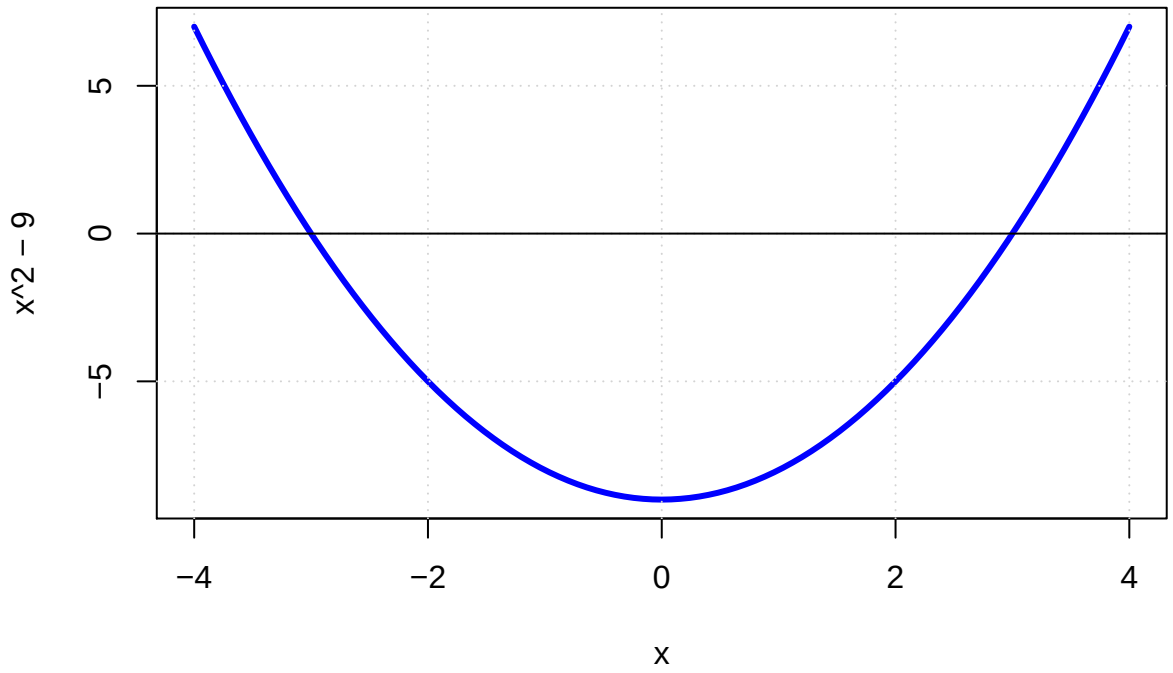
```
## [1] 0
```

Root-finding using the Bisection Method

The bisection method is the first numerical algorithm we'll consider for finding roots of arbitrary nonlinear functions.

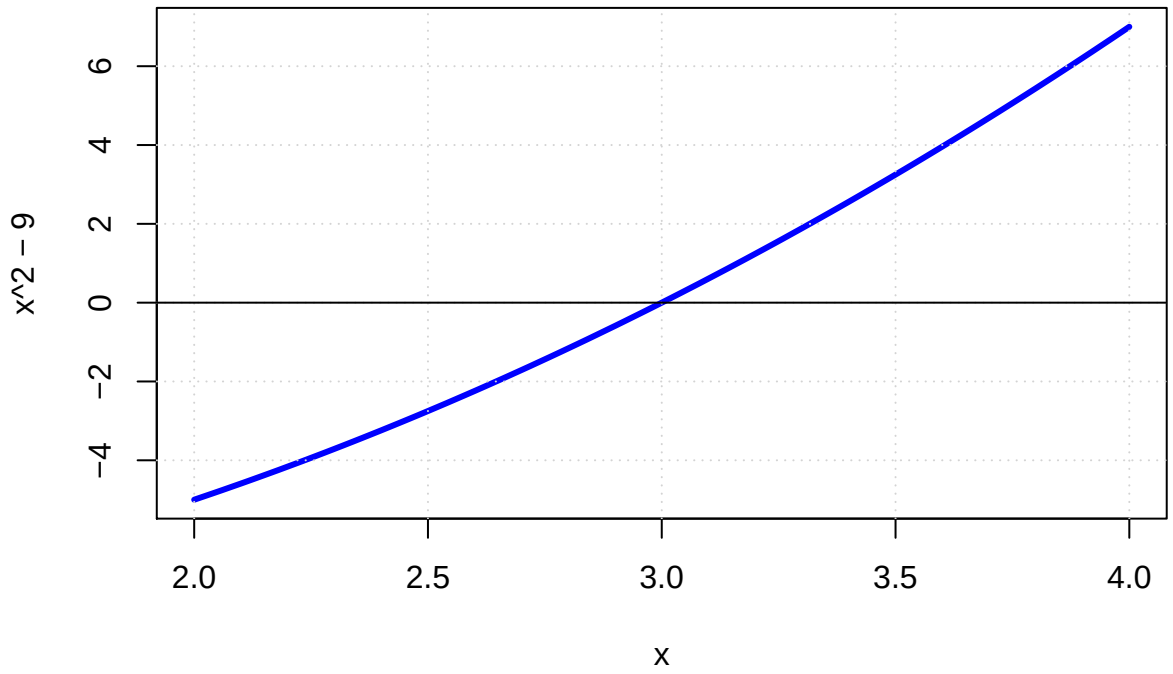
Suppose that $f : R \to R$ is a continuous function. A root of f is a solution to the equation $f(x) = 0$. That is, a root is a number $x_0 \in R$ such that $f(x_0) = 0$. If we draw the graph of our function, say $y = f(x)$, which is a curve in the plane, a solution of $f(x) = 0$ is the x-coordinate of a point at which the curve crosses the x-axis.

```
curve(x^2-9,-4,4,col="blue",lwd=3)
grid()
abline(h=0)
```



We can zoom in on the interval $[2, 4]$ to find one of the roots and then we can look inside the interval $[-4, -2]$ to find the second root.

```
curve(x^2-9,2,4,col="blue",lwd=3)
grid()
abline(h=0)
```



The bisection method works by first isolating an interval in which the root must lie, and then successively refining the bounding interval in such a way that the root is guaranteed to always lie inside the interval. More precisely, the width of the bounding interval is successively halved.

Suppose that f is a continuous function, then it is easy to see that f has a root in the interval (a, b) if either

$f(a) < 0$ and $f(b) > 0$ or if $f(a) > 0$ and $f(b) < 0$. That is, the function must have opposite signs at the endpoints of the interval, because only then the continuity of the function guarantees a root inside this interval since the graph of the function must cross the x-axis at some point inside this interval.

A convenient way to verify this condition is to check if $f(a)f(b) < 0$ - this is equivalent to f having opposite signs at a and b. The bisection method works by taking an interval (a, b) that contains a root, then successively refining a and $b > a$ until $b - a \leq \delta$, where δ is some predefined tolerance. Here is the bisection algorithm:

Bisection Method: Start with $a < b$ such that $f(a)f(b) < 0$.

1. if $b - a \leq \delta$ then stop, if not go to step 2.
2. let $c = (a + b)/2$, if $f(c) = 0$ stop, otherwise go to step 3.
3. if $f(a)f(c) < 0$ then put $b \leftarrow c$, otherwise put $a \leftarrow c$.
4. go back to step 1.

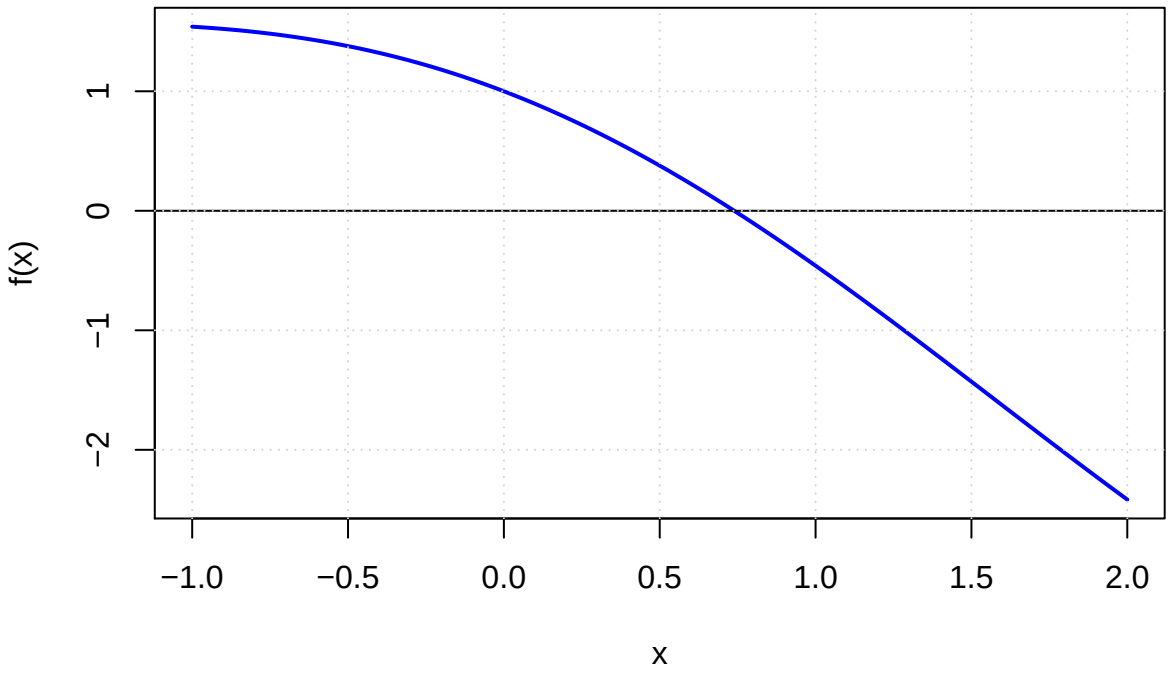
Note that at every iteration of the algorithm, we know that there is root in the interval (a, b) and with every iteration the interval is halved. Provided we start with $f(a)f(b) < 0$, the algorithm is guaranteed to converge. If we stop when $b - a \leq \delta$, then we know that both a and b are within distance δ of a root.

For the purpose of quick development and computation, one can remove all the extra checks in the bisection algorithm and simply work with a bare bone algorithm that does the job, provided the assumptions we make about the function are satisfied. We do simplify the algorithm by running it only for a certain number of steps (given a default value of 20) rather than specifying tolerance:

```
# bisect1 computes a root approximation of f(x) in [a,b] using bisection.
# we must have f(a)f(b)<0 and only a specified number of steps are executed.
bisect1<-function(f,a,b,steps=20){
  for (n in 1:steps){
    c<-(a+b)/2
    if (f(a)*f(c)<0){ # if true, a and c make the new interval
      b<-c
      } else {
        a<-c  # now, c and b make the new interval
        }
    }
  return((a+b)/2) # the midpoint is the best estimate
  }
```

We are interested in finding a root of $f(x) = \cos(x) - x$. We can first plot the function to locate visually where the roots maybe hiding. Once we find an interval that contains a root, then we can apply the `bisect1` function to find the root using the bisection algorithm.

```
f<-function(x) cos(x)-x
curve(f(x),-1,2,col="blue",lwd=2)
abline(h=0)
grid()
```



We can see that a root must live in the interval $[0, 1]$ since the graph of f appears to cross the x-axis in this interval. We can now apply the bisection algorithm to find the root in $[0, 1]$ with 20 steps.

```
## [1] 0.7391
```

```
## Check: f(root)= 6.905e-07
```

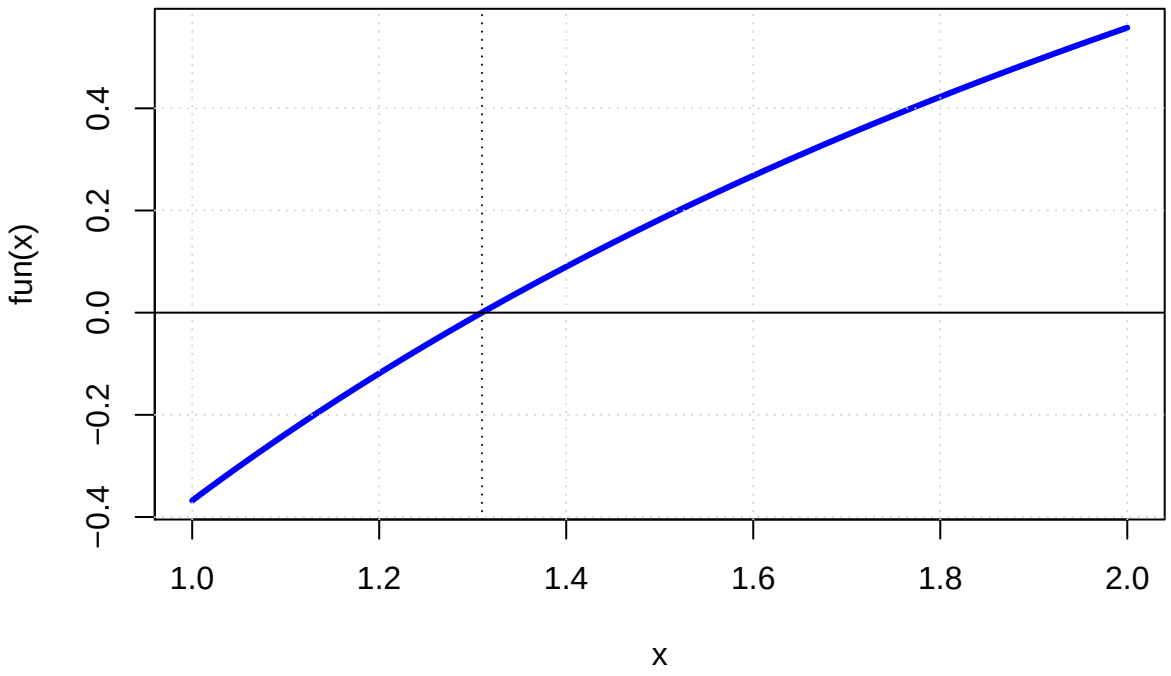
We can create a more sophisticated bisection algorithm with checks on the inputs and rather than specifying a certain number of steps for the algorithm to run, we can use a while loop to stop only when a condition is met, which will guarantee that the root was computed within certain tolerance.

```
bisect<- function(fun,a,b,tol = 1e-9) {
  # applies the bisection algorithm to find x such that fun(x)=0 within the tolerance
  # a and b must bracket a root, that is a < b and fun(a)*fun(b)< 0
  # the algorithm iteratively refines a and b and terminates when b - a <= tol
  # check inputs
  if (a >= b) {
    cat("error: left bound >= right bound \n")
    return(NULL)
    }
  fa <- fun(a)
  fb <- fun(b)
  if (fa == 0) {
    return(a)
    } else if (fb == 0) {
      return(b)
      } else if (fa*fb > 0) {
        cat("error: the function has the same sign at both ends \n")
        return(NULL)
        }
  # successively refine a and b
  n <- 0
```

```
  while ((b - a) > tol) {
    c <- (a + b)/2
    fc <- fun(c)
    if (fc == 0) {
      return(c)
      } else if (fa*fc < 0) {
        b <- c
        fb <- fc
        } else {
          a <- c
          fa <- fc
          }
    n <- n + 1
    cat("at iteration", n, "the root lies between",a, "and",b, "\n")
    }
  # return (approximate) root
  return((a + b)/2)
  }
```

Here is this bisection algorithm in action.

```
# to find a root of fun in [1,2]
fun <- function(x) return(log(x) - exp(-x))
curve(fun,1,2,col="blue",lwd=3)
grid()
abline(h=0,lty=1)
abline(v=1.31,lty=3)
```




```
options(digits=6)
(root<-bisect(fun, 1, 2, tol = 1e-06))
```

```
## at iteration 1 the root lies between 1 and 1.5
## at iteration 2 the root lies between 1.25 and 1.5
## at iteration 3 the root lies between 1.25 and 1.375
## at iteration 4 the root lies between 1.25 and 1.3125
## at iteration 5 the root lies between 1.28125 and 1.3125
## at iteration 6 the root lies between 1.29688 and 1.3125
## at iteration 7 the root lies between 1.30469 and 1.3125
## at iteration 8 the root lies between 1.30859 and 1.3125
## at iteration 9 the root lies between 1.30859 and 1.31055
## at iteration 10 the root lies between 1.30957 and 1.31055
## at iteration 11 the root lies between 1.30957 and 1.31006
## at iteration 12 the root lies between 1.30957 and 1.30981
## at iteration 13 the root lies between 1.30969 and 1.30981
## at iteration 14 the root lies between 1.30975 and 1.30981
## at iteration 15 the root lies between 1.30978 and 1.30981
## at iteration 16 the root lies between 1.3098 and 1.30981
## at iteration 17 the root lies between 1.3098 and 1.30981
## at iteration 18 the root lies between 1.3098 and 1.3098
## at iteration 19 the root lies between 1.3098 and 1.3098
## at iteration 20 the root lies between 1.3098 and 1.3098
```

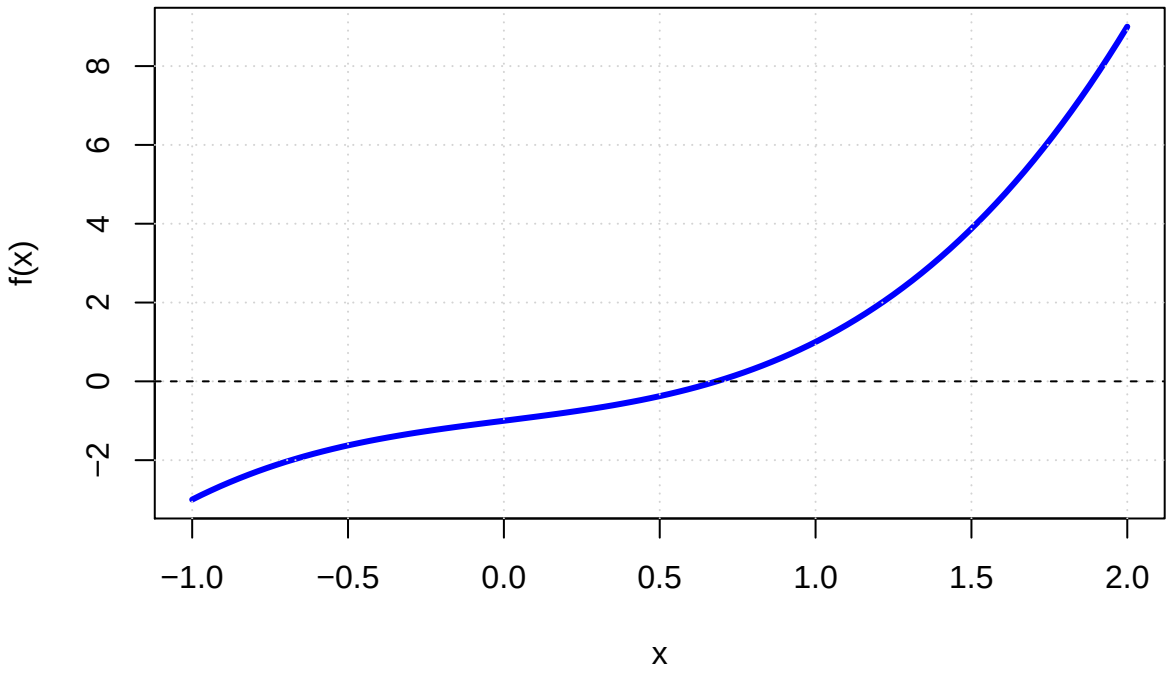
```
## [1] 1.3098
```

```
fun(root) # check the numerical root
```

```
## [1] 8.8216e-08
```

Examples Let's find the root(s) of the cubic polynomial $f(x) = x^3 + x - 1$ using bisection. We can first plot the function to locate an interval that contains only one root.

```
f<-function(x) x^3+x-1
curve(f(x),-1,2,col="blue",lwd=3)
grid()
abline(h=0,lty=2)
```



It appears the function has only one real root, inside the interval $[0.5, 1]$. Any polynomial of degree n must have exactly n roots, but they can be either real or complex. In this case, the other two roots are complex. For polynomials with real coefficients, complex roots come in complex conjugate pairs.

```
options(digits=4)
bisect(f,0.5,1,tol=1e-6) # compute a real root of f in [0.5,1] using bisection
```

```
## at iteration 1 the root lies between 0.5 and 0.75
## at iteration 2 the root lies between 0.625 and 0.75
## at iteration 3 the root lies between 0.625 and 0.6875
## at iteration 4 the root lies between 0.6562 and 0.6875
## at iteration 5 the root lies between 0.6719 and 0.6875
## at iteration 6 the root lies between 0.6797 and 0.6875
## at iteration 7 the root lies between 0.6797 and 0.6836
## at iteration 8 the root lies between 0.6816 and 0.6836
## at iteration 9 the root lies between 0.6816 and 0.6826
## at iteration 10 the root lies between 0.6821 and 0.6826
## at iteration 11 the root lies between 0.6821 and 0.6824
## at iteration 12 the root lies between 0.6823 and 0.6824
## at iteration 13 the root lies between 0.6823 and 0.6824
## at iteration 14 the root lies between 0.6823 and 0.6823
## at iteration 15 the root lies between 0.6823 and 0.6823
## at iteration 16 the root lies between 0.6823 and 0.6823
## at iteration 17 the root lies between 0.6823 and 0.6823
## at iteration 18 the root lies between 0.6823 and 0.6823
## at iteration 19 the root lies between 0.6823 and 0.6823

## [1] 0.6823
```

Numerical Accuracy of the Bisection Algorithm **DEFINITION:** An approximate root found by the bisection algorithm is **correct within p decimal places** if the error between the actual root and the approximate root is less than 0.5×10^{-p}.

If $[a, b]$ is the starting unit length interval, then after n bisection steps, the resulting interval $[a_n, b_n]$ has length $b_n - a_n = (b-a)/2^n$ since with every step the interval is halved. Choosing the midpoint of the final interval $x_c = (a_n + b_n)/2$ gives the best estimate of the true root r, and the estimate for the root being x_c is within half the interval length of the true root r.

This means that after n steps of the Bisection Method, we have that

$$\text{solution error} = |x_c - r| < \frac{b-a}{2^{n+1}}$$

Using the formual for the solution error, we can decide how many steps of bisection are required to achieve a certain accuracy for the approximate root, based on the definition of *correct to p decimal places*.

For example, in order to achieve an accuracy of $p = 6$ decimal places, assuming that we started with a unit interval $b - a = 1$, we must have:

$$\text{solution error} = \frac{b-a}{2^{n+1}} = \frac{1}{2^{n+1}} \leq 0.5 \times 10^{-6},$$

which gives an estimate for n (compute it yourselves):

$$n \geq 19.9316,$$

so we can take $n = 20$ steps to achieve an accuracy of 6 decimal places.

QUESTION: Assuming we start with a unit interval, compute how many steps are needed for the bisection method to get a root correct within 10 decimal places. Answer: 34

Fixed-Point Iteration

Let us apply the cos() function repeatedly to an arbitrary starting number. That is, apply the cos() function to the starting number, then apply cos() to the result, then to the new result, and so forth. Continue until the digits no longer change:

```
options(digits=5)
out <- runif(1,0,10) # 1 random number from (0,10)
for(i in 2:40) {
  out[i] <- cos(out[i-1]) # repeated application of cos
  }
tail(out,8)
```

```
## [1] 0.73908 0.73909 0.73908 0.73909 0.73908 0.73909 0.73909 0.73909
```

The resulting sequence of numbers converges to 0.73909.

In this section, our goal is to explain why this calculation, an instance of Fixed-Point Iteration (FPI), converges.

Fixed points of a function The sequence of numbers produced by iterating the cosine function appears to converge to a fixed number r. Subsequent applications of cosine do not change the number. For this input, the output of the cosine function is equal to the input, or $\cos(r) = r$.

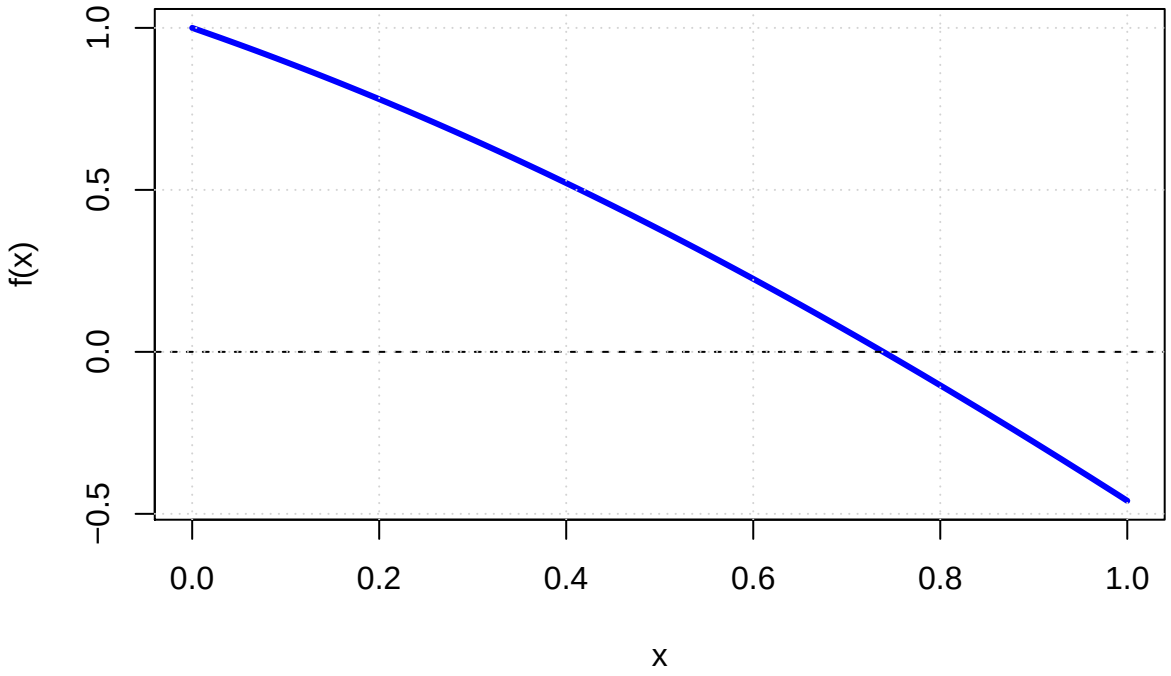
DEFINITION The real number r is a **fixed point** of the function g if $g(r) = r$.

The number $r = 0.73909$ is an approximate fixed point for the function $g(x) = \cos(x)$.

The function $g(x) = x^3$ has three fixed points, $r = -1, 0, 1$.

Let's use the Bisection Method to solve the equation $\cos(x) - x = 0$.

```
f<-function(x)cos(x)-x
curve(f,0,1,col="blue",lwd=3)
abline(h=0,lty=2)
grid()
```




```
bisect1(f,0,1,steps=30) # for loop implementation with 20 steps default
```

```
## [1] 0.73909
```

We can rewrite the equation $\cos(x) - x = 0$ into a new equation $\cos(x) = x$. A solution r of this equation is the same thing as the fixed-point of $\cos(x)$. Solving the fixed-point equation $\cos(r) = r$ is the same problem but from a different point of view.

When the output equals the input, that number is a fixed point of $\cos(x)$, by definition, and simultaneously a solution of the equation $\cos(x) - x = 0$.

Once the equation is written as $g(x) = x$, Fixed-Point Iteration proceeds by starting with an initial guess x_0 and a repeated application of the function g.

Fixed-Point Iteration:

$$x_0 = \text{initial guess}$$
$$x_{i+1} = g(x_i) \text{ for } i = 0, 1, 2, \ldots$$

Therefore,

$$x_1 = g(x_0)$$

$$x_2 = g(x_1)$$
$$x_3 = g(x_2)$$
$$\dots$$

and so forth. The sequence of numbers x_i may or may not converge as the number of steps goes to infinity. However, if g is continuous and the x_i converge, say, to a number r, $\lim_{i\to\infty} x_i = r$, then r is a fixed point of g. In fact, the continuity of g implies:

$$g(r) = g(\lim x_i) = \lim_{i\to\infty} g(x_i) = \lim_{i\to\infty} x_{i+1} = r$$

The Fixed-Point Iteration algorithm applied to a function g is easily written as a repeated function composition:

```
# Fixed-Point Iteration function
# Input: a function g, starting point init, and number of times=steps to apply the function
# Output: a vector of size steps, with all applications of the function
fpi<-function(g,init,steps){
out <- init # some initial number
for(i in 2:steps) {
  out[i] <- g(out[i-1]) # repeated application of g
  }
return(out)
}
```

Let's apply the fixed-point iteration to $g(x) = \cos(x)$:

```
(out<-fpi(cos,init=0,steps=32))
```

```
##  [1] 0.00000 1.00000 0.54030 0.85755 0.65429 0.79348 0.70137 0.76396
##  [9] 0.72210 0.75042 0.73140 0.74424 0.73560 0.74143 0.73751 0.74015
## [17] 0.73837 0.73957 0.73876 0.73930 0.73894 0.73918 0.73902 0.73913
## [25] 0.73905 0.73911 0.73907 0.73909 0.73908 0.73909 0.73908 0.73909
```

We see the convergence here to the number 0.73909, which therefore is the fixed-point of the cosine function.

Fixed-Point Iteration solves the fixed-point problem $g(x) = x$, but we are interested in solving equations.

Can every equation $f(x) = 0$ be turned into a fixed-point problem $g(x) = x$?

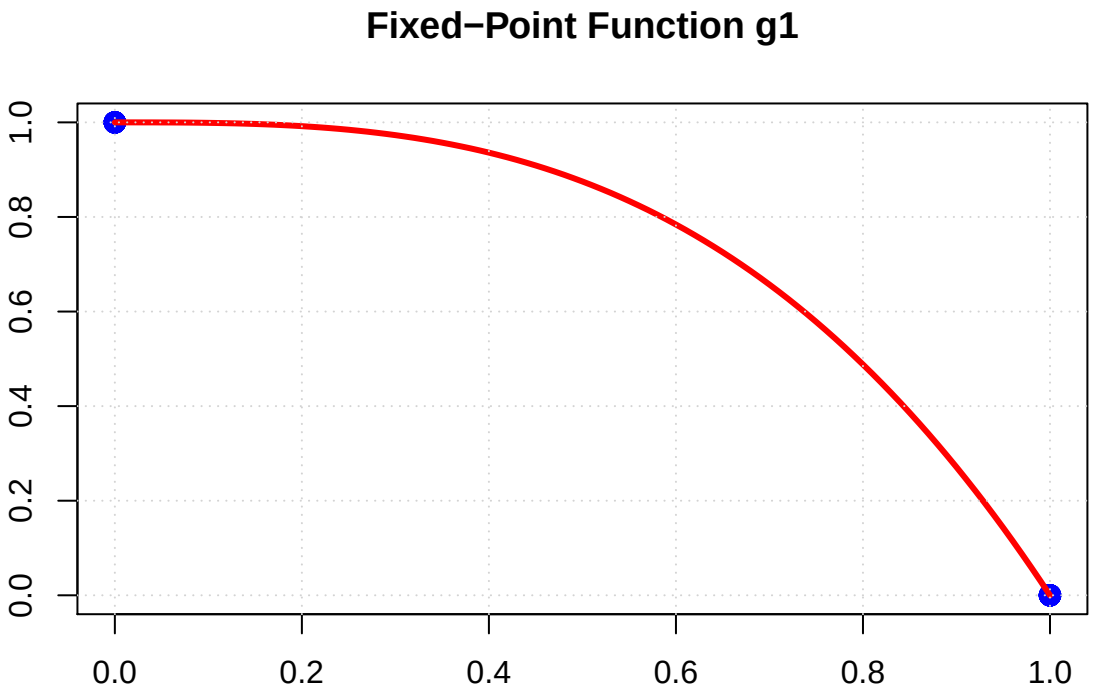
The answer is yes, and in many different ways. However, some ways will not lead to convergence of the fixed-point iteration, some ways will be slower than others, and in general it is up to us to make a good choice.

For example, we can rewrite the equation $x^3 + x - 1 = 0$ as a fixed-point equation in many different ways:

1. $x = 1 - x^3$
2. $x^3 = 1 - x \implies x = \sqrt[3]{1-x}$
3. Add to both sides $2x^3$ to get $3x^3 + x = 1 + 2x^3 \implies x(3x^2+1) = 1 + 2x^3 \implies x = \frac{1+2x^3}{1+3x^2}$

So, we get three different fixed-point equations $x = g(x)$ that represent the same equation $f(x) = 0$. Let's try to find the fixed-points in the three cases. In the case of the fixed-point function $g_1(x) = 1 - x^3$, we get:

```
g1<-function(x) 1-x^3
out1<-fpi(g1,init=0,steps=24)
plot(head(out1,-1),tail(out1,-1),col="blue",lwd=4,ylab="",xlab="")
curve(g1,0,1,col="red",lwd=3,add=T)
title(main="Fixed-Point Function g1")
grid()
```

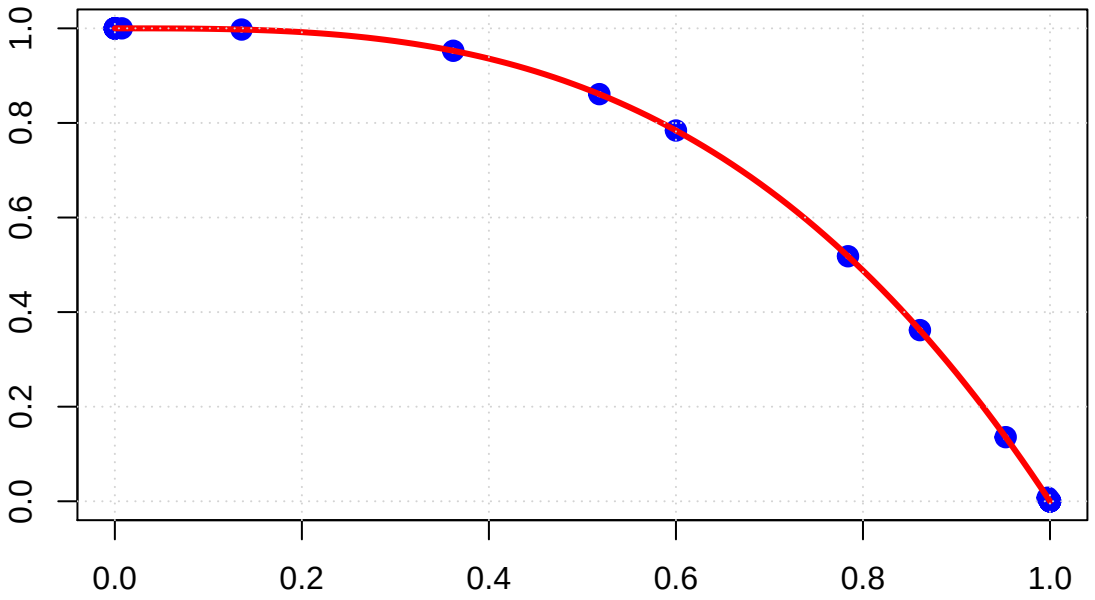



```
tail(out1,15)
```

```
##  [1] 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
```

We don't get a convergence. Let's change the initial guess to $x_0 = 0.6$, which is very close to the root:

```
g1<-function(x) 1-x^3
out1<-fpi(g1,init=0.6,steps=24)
plot(head(out1,-1),tail(out1,-1),col="blue",lwd=4,ylab="",xlab="")
curve(g1,0,1,col="red",lwd=3,add=T)
title(main="Fixed-Point Function g1")
grid()
```

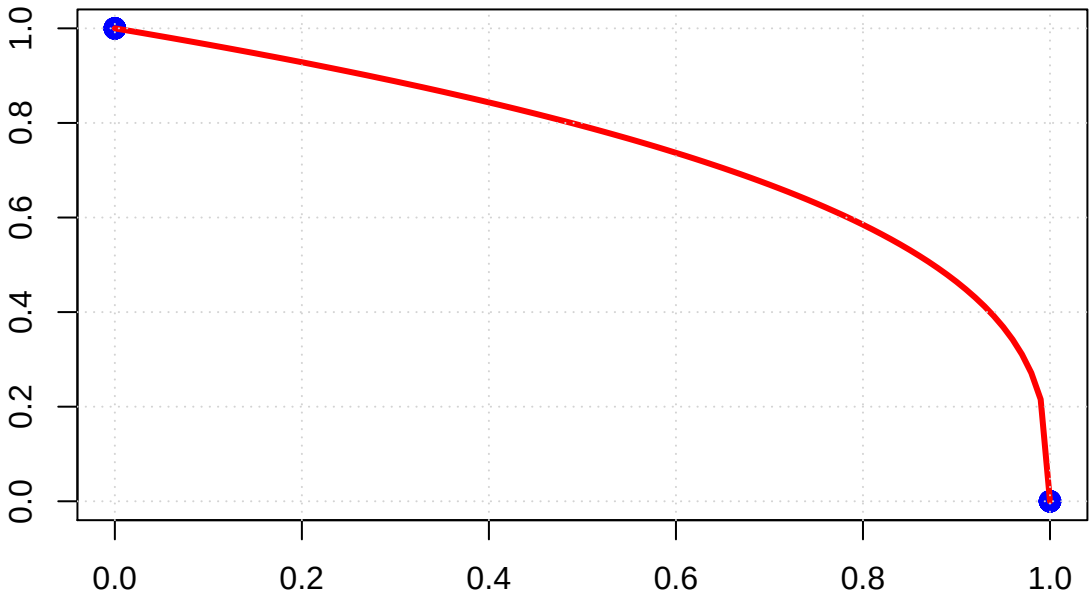



```
out1
```

```
##  [1] 6.0000e-01 7.8400e-01 5.1811e-01 8.6092e-01 3.6190e-01 9.5260e-01
##  [7] 1.3556e-01 9.9751e-01 7.4553e-03 1.0000e+00 1.2431e-06 1.0000e+00
## [13] 0.0000e+00 1.0000e+00 0.0000e+00 1.0000e+00 0.0000e+00 1.0000e+00
## [19] 0.0000e+00 1.0000e+00 0.0000e+00 1.0000e+00 0.0000e+00 1.0000e+00
```

We still don't get a convergence, so this fixed-point function is useless for us. Now, let's investigate the second fixed point function $g_2(x) = (1-x)^{1/3}$:

```
g2<-function(x) (1-x)^(1/3)
out2<-fpi(g2,init=0,steps=24)
plot(head(out2,-1),tail(out2,-1),col="blue",lwd=4,ylab="",xlab="")
curve(g2,0,1,col="red",lwd=3,add=T)
title(main="Fixed-Point Function g2")
grid()
```

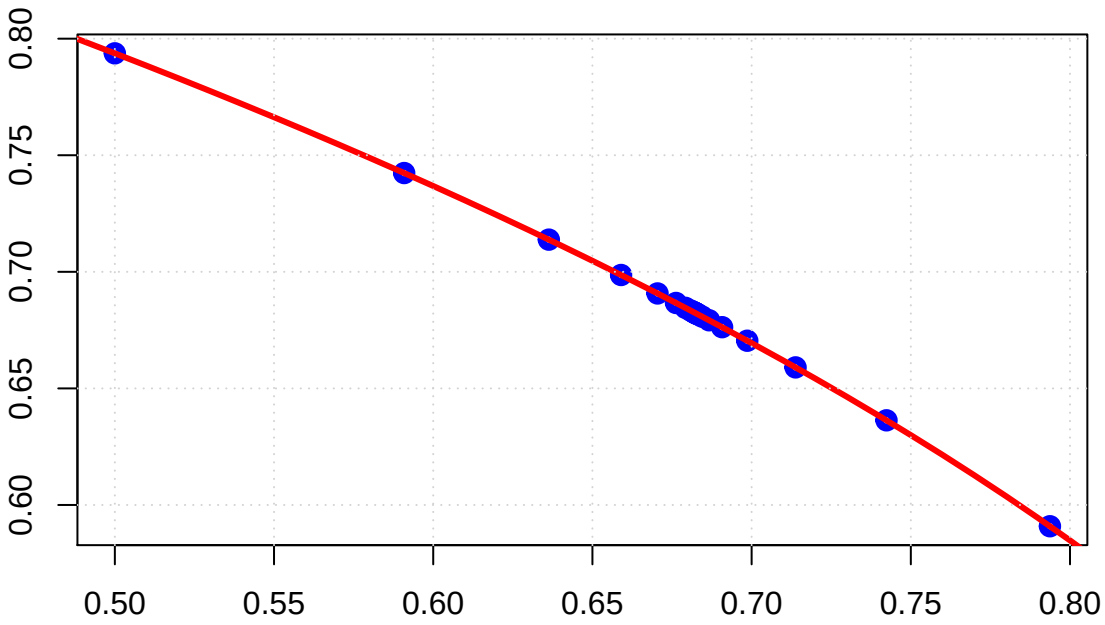



```
tail(out2,15)
```

```
##  [1] 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
```

We don't get a convergence with this initial guess, so let's change it to $x_0 = 0.5$:

```
g2<-function(x) (1-x)^(1/3)
out2<-fpi(g2,init=0.5,steps=24)
plot(head(out2,-1),tail(out2,-1),col="blue",lwd=4,ylab="",xlab="")
curve(g2,0,1,col="red",lwd=3,add=T)
title(main="Fixed-Point Function g2")
grid()
```

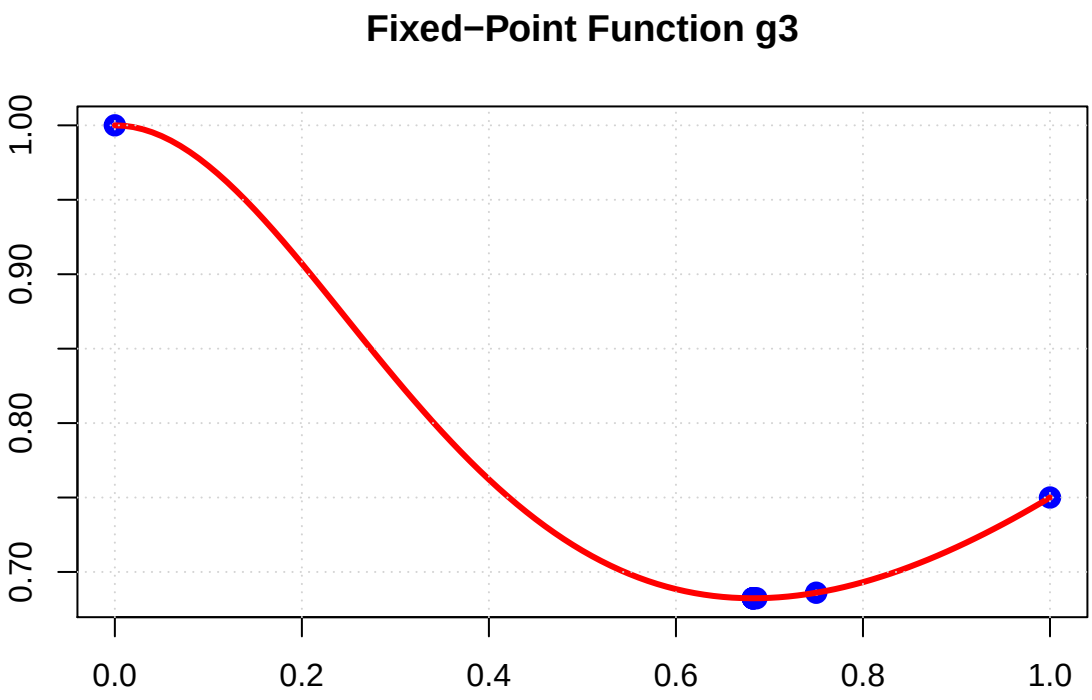



```
tail(out2,15)
```

```
##  [1] 0.69073 0.67626 0.68665 0.67922 0.68454 0.68074 0.68346 0.68151
##  [9] 0.68291 0.68191 0.68263 0.68211 0.68248 0.68222 0.68241
```

Now, we get a convergence. Finally, let's investigate the third fixed-point function $g_3(x) = \frac{1+2x^3}{1+3x^2}$:

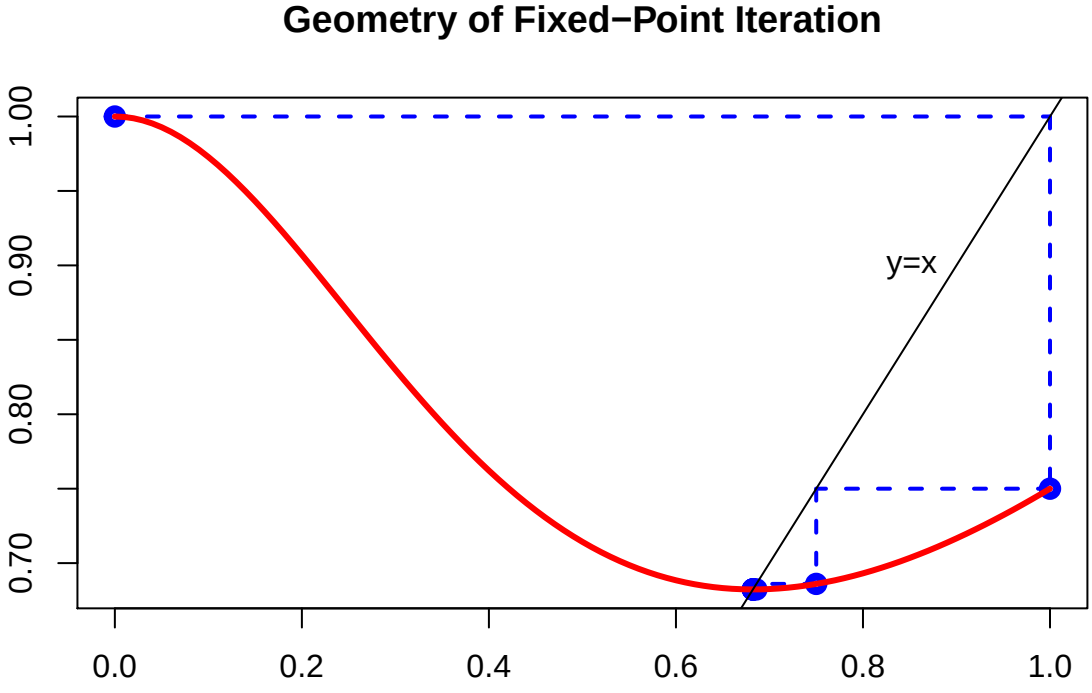
```
g3<-function(x) (1+2*x^3)/(1+3*x^2)
out3<-fpi(g3,init=0,steps=10)
plot(head(out3,-1),tail(out3,-1),col="blue",lwd=4,ylab="",xlab="")
curve(g3,0,1,col="red",lwd=3,add=T)
title(main="Fixed-Point Function g3")
grid()
```




```
out3
```

```
##  [1] 0.00000 1.00000 0.75000 0.68605 0.68234 0.68233 0.68233 0.68233
##  [9] 0.68233 0.68233
```

In this case, we get a very fast convergence to the fixed-point 0.68233.



Computing $\sqrt{2}$ by Fixed-Point Iteration Consider the function $g(x) = \frac{1}{2}\left(x + \frac{2}{x}\right)$.

QUESTION: Compute by hand the fixed points of g, that is, solve the equation $g(x) = x$.

The fixed points of g are $\pm\sqrt{2}$. Thus, we can use fixed-point iteration to compute approximately the fixed point $\sqrt{2}$, by starting with a guess of 1, close to the fixed point.

```
options(digits=12)
g<-function(x) 1/2*(x+2/x)
(out<-fpi(g,init=1,steps=5))
```

```
## [1] 1.00000000000 1.50000000000 1.41666666667 1.41421568627 1.41421356237
```

After only 5 steps, the fixed-point iteration gives an approximation for $\sqrt{2}$ equal to 1.414213562375, and compared with the exact value $\sqrt{2} = 1.414213562373$, we can check using `all.equal()` that the approximation is correct within 11 digits, check: TRUE (R code behind the check).

The Babylonians knew more than 4000 years ago about this iteration procedure for getting an approximate value of $\sqrt{2}$.

Newton's Method

Newton's Method, also called the Newton–Raphson Method, usually converges much faster than the bisection and the fixed-point iteration methods. To find a root of $f(x) = 0$, a starting guess x_0 is given, and the tangent line to the function f at $(x_0, f(x_0))$ is drawn. The tangent line will approximately follow the function down to the x-axis toward the root. The intersection point of the line with the x-axis is an approximate root, but probably not exact if f curves. Therefore, this step is iterated.

From this geometric picture, we can develop an algebraic formula for Newton's Method. The tangent line at x_0 has slope given by the derivative $f'(x_0)$. One point on the tangent line is $(x_0, f(x_0))$. The point-slope formula for the equation of the tangent line is

$$y = f(x_0) + f'(x_0)(x - x_0)$$

so that looking for the intersection point x_1 of the tangent line with the x-axis is the same as substituting $y = 0$ in the line:

$$0 = f(x_0) + f'(x_0)(x_1 - x_0) \implies x_1 = x_0 - \frac{f(x_0)}{f'(x_0)},$$

assuming $f'(x_0) \neq 0$. This gives an approximation for the root, which we call x_1. Next, the entire process is repeated, beginning this time with x_1, to produce x_2, and so on, yielding the following iterative formula:

Newton's Method:

$$x_0 = \text{initial guess}$$
$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \text{ for } i = 0, 1, 2, \ldots$$

Examples:

1. Let's find the Newton's Method formula for the equation $x^3 + x - 1 = 0$. Here, $f(x) = x^3 + x - 1$, so $f'(x) = 3x^2 + 1$. Thus, the recursive formula is:

$$x_{i+1} = x_i - \frac{x_i^3 + x_i - 1}{3x_i^2 + 1} \text{ for } i = 0, 1, 2, \ldots$$

2. Let's find the Newton's Method formula for the equation $x^2 = 2$. Here, $f(x) = x^2 - 2$ since the equation $f(x) = 0$ is equivalent to the original equation $x^2 = 2$. Thus, Newton's recursive formula is:

$$x_{i+1} = x_i - \frac{x_i^2 - 2}{2x_i} = \frac{1}{2}\left(x_i + \frac{2}{x_i}\right) \text{ for } i = 0, 1, 2, \ldots,$$

which is nothing but the Babylonian formula we used to compute $\sqrt{2}$ approximately with fixed-point iteration.

We can easily implement Newton's Method as a function, which takes the function f, the derivative function df, an initial guess init and the number of steps for Newton's iteration.

```
newton<-function(f,df,init,steps=10){
  x<-init
  for (i in 1:(steps-1)){
    x[i+1]<-x[i]-f(x[i])/df(x[i])
    }
  return(x)
  }
```

Example: Let's apply Newton's method to the function $f(x) = x^2 - 2$.

```
f<-function(x)x^2-2
df<-function(x)2*x
(out<-newton(f,df,1,steps=5))
```

```
## [1] 1.00000000000 1.50000000000 1.41666666667 1.41421568627 1.41421356237
```

A good approximation to $\sqrt{2}$ is then 1.414213562375.

Since we are dividing by the derivative of f evaluated at the current approximation, it is advisable to check whether the derivative is close to 0 or not. We can just add a special check for this:

```
newton1<-function(f,df,init,steps=10){
  x<-init
  for (i in 1:(steps-1)){
    if (isTRUE(all.equal(df(x[i]),0,1e-6))){
      return("Singular Derivative")
      } else {
        x[i+1]<-x[i]-f(x[i])/df(x[i])
        }
    }
  return(x)
  }
```

```
f<-function(x)x^2-2
df<-function(x)2*x
newton1(f,df,0,steps=5)
```

```
## [1] "Singular Derivative"
```