Root Finding

Ivan Yi-Fan Chen

2023 Fall

Preliminaries for R

`do.call`

- `do.call(what,args)`
  - `what` is a R function.
  - `args` is a **named list** such that contains the arguments of `what` .
- `do.call` evaluates `what` using `args` .

```
fx<-function(x,y) {x+y} #An arbitrary function taking x and y.
fx(x=1,y=2)
```

```
## [1] 3
```

```
do.call(what=fx, args=list(x=1,y=2))
```

```
## [1] 3
```

- Usage 1: Some operations generate a list of object, but you want to combine them into a matrix or data.frame. Then use `do.call` for this.

```
chr<-c("This is a pen","This is an apple")
#Want to break the two sentences into single words, and store each word into a cell of a data.frame.
#Try to use `strsplit` to break the sentence by " " (space). But this gives you a LIST.
chrl<-strsplit(x=chr, split=" ")
chrl
```

```
## [[1]]
## [1] "This" "is"   "a"    "pen"
##
## [[2]]
## [1] "This"  "is"    "an"    "apple"
```

```
#Use do.call plus rbind.
do.call(what=rbind, args=chrl)
```

```
##      [,1]   [,2] [,3] [,4]
## [1,] "This" "is" "a"  "pen"
## [2,] "This" "is" "an" "apple"
```

- Usage 2: Define a function `f` that takes function `g` as its input. Moreover, we want `f` to perform certain manipulation to **specified** input in `g` and pass back into it. But there are problems: `function` evaluates according to either the position or the name of the input, e.g., `sample(size=2,x=c(10:100))` and `sample(c(10:100),2)` are the same. Assume that `f` detects the input of `sample` , make changes to the specified argument and rerun `sample` . Then how do we feed the changed argument into `sample` via `f` ? Use a self-defined function to illustrate this problem:

```r
fx<-function(x=1,y=2) {x+y} #An arbitrary function taking x and y.

#Want to define a function `fntmp` that multiplies the specified argument of fx by 2, and re-evaluat
e.
#What we want to do:
fx(x=2*1,y=2) #Multiply 2 to x and run fx.
```

```
## [1] 4
```

```r
fx(x=1,y=2*2) #Multiply 2 to y and run fx.
```

```
## [1] 5
```

- Intuitively we want to write. But this works only for multiplying x!

```r
fntmp<-function(fun=fx,var=1,...) {
  var<-2*var
  fx(var,...)
}
fntmp(fx,1)
```

```
## [1] 4
```

- How about use something like var="x=1" and var="y=2"? It won't work since what we pass in is a character

```r
fntmp<-function(fun=fx,var="x=1",...) {
  var<-2*var
  fx(var,...)
}
fntmp(fx,var="x=1")
```

```
## Error in 2 * var: 二元運算子中有非數值引數
```

- How about we use two arguments, one is for the name of the variable of interest and the other is its current value? No it won't work because it is again a string…

```r
fntmp<-function(fun=fx,var="x",val=1,...) {
  val<-2*val
  arg<-paste(var,"=",val,sep="")
  fx(arg,...)
}
fntmp(fx,var="x",val=1,y=2)
```

```
## Error in x + y: 二元運算子中有非數值引數
```

```r
fntmp(fx,var="y",val=2,x=1)
```

```
## Error in x + y: 二元運算子中有非數值引數
```

- For the previous one, there is still a hope called **non-standard evaluation**.
  - Takes a string and parse it as an R syntax.
  - Inefficient and hard to manage. You won't be able to know what you did 2 days later!

    - If you really want to learn this, check out Hadley’s book: http://adv-r.had.co.nz/Computing-on-the-language.html (http://adv-r.had.co.nz/Computing-on-the-language.html)
- What we want to do can be done with `do.call`.

```
fx<-function(x,y) {x+y} #An arbitrary function taking x and y.

fntmp<-function(f,var,...) {

  if(!is.list(var)||length(var)!=2) {stop("var needs to be a list of length 2.")}
  if(!is.character(var[[1]])||!is.numeric(var[[2]])) {
    stop("First and second elements of var must be character and numeric respectively.")}
  if(!is.function(f)) {stop("f must be a function.")}

  ar<-var[[2]]*2 #Multiplies the variable of interest by 2
  ar<-c(ar)%>%'names<-'(var[[1]]) #Supply the name of variable
  ar<-as.list(c(ar,...)) #Bind all the variables, including ... as a named list
  rst<-do.call(f,ar) #Evaluate with the modified variables using do.call
  return(list(variables=ar,result=rst)) #Return output
}
fx(x=1,y=2)
```

```
## [1] 3
```

```
fntmp(var=list("x",1),fx,y=2)
```

```
## $variables
## $variables$x
## [1] 2
##
## $variables$y
## [1] 2
##
##
## $result
## [1] 4
```

```
fntmp(var=list("y",2),fx,x=1)
```

```
## $variables
## $variables$y
## [1] 4
##
## $variables$x
## [1] 1
##
##
## $result
## [1] 5
```

sign

- `sign(x)` tests whether the numeric object `x` is positive, negative, or zero.
    - `x` is a vector.
    - For each element in `x`, returns 1 if positive, -1 if negative, and 0 if 0.

```
#Single numeric object
sign(21)
```

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

```
sign(-53)
```

```
## [1] -1
```

```
sign(0)
```

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

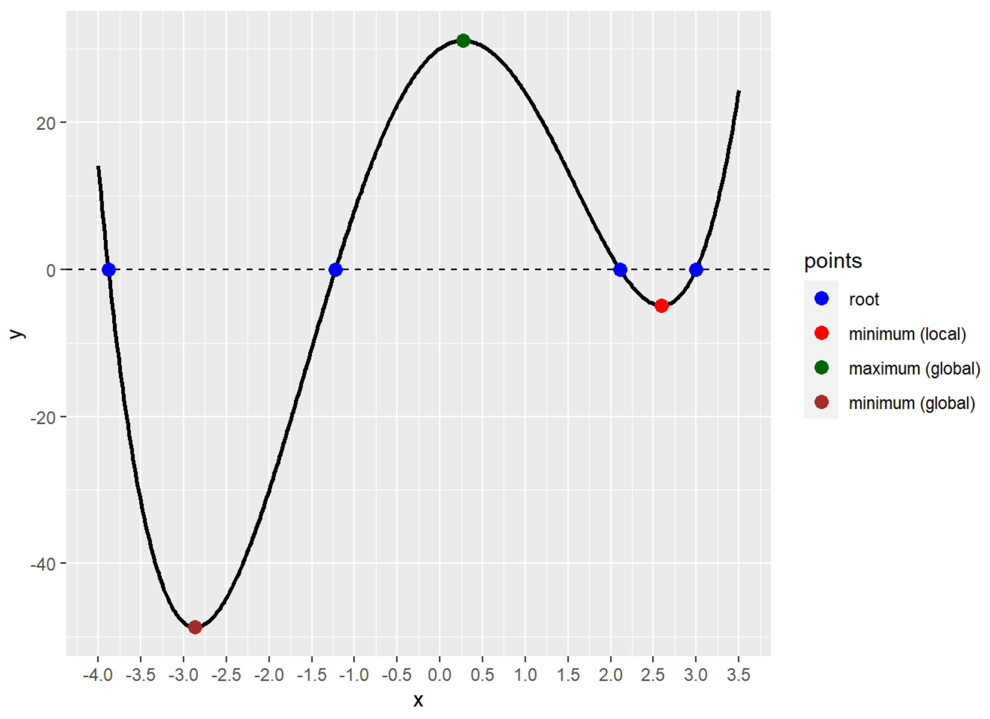
```
#A vector of numeric object
sign(c(21,-53,0))
```

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

Root Finding v.s. Optimization

- **Root Finding** problem is about finding an x^* such that $g(x^*) = 0$, i.e., solving $g(x)$.
    - We refer x^* to as the root of the function.
    - Can have multiple solutions.
    - Example 1: $g(x) = f(x) - b$, then the root is defined by that $f(x^*) = b$.
    - Example 2: $g(x) = f(x) - h(x)$, then the root is such that $f(x^*) = h(x^*)$.
- **Optimization Problem** is about finding the **minimum** of $g(x)$. That is, we find an x^* such that $\min_x g(x)$.
    - How about **maximum**? This is equivalent to minimizing $-g(x)$.
- Consider the following function for illustration.

$$y = x^4 - 15x^2 + 8x + 30$$



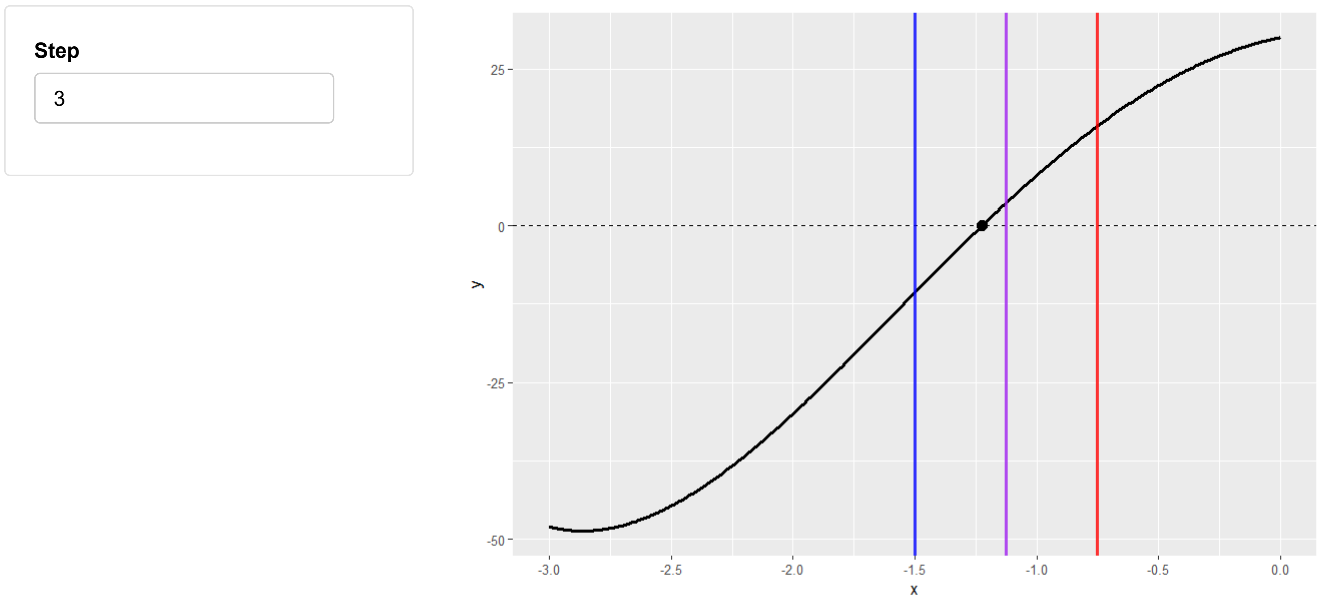
- For a function $g(x)$ that is at least twice differnetiable, the optimization problem can be reformulated into a root finding problem using first order derivative as $g'(x^*) = 0$. The second order derivative determines whether x^* entails a minimum or maximum.
- A root finding problem can also be reformulated into an optimization problem:
    1. **Convert to unconstrained optimization:** If $g(x) = 0$ holds for some x^*, then x^* are such that $|g(x)|$ is minimized. Oftentimes we work with minimizing $\sqrt{g(x)^2}$.
    2. **Convert to constrained optimization:** If $g(x) = 0$ holds for some x^*, then x^* are such that $\min_x 1$ subject to $g(x) = 0$. Therefore we can solve by Lagrangian method

    $$\min_x L = 1 + \lambda(g(x))$$

    where λ is the Lagrangian multiplier.
- Converting a root finding problem into an optimization problem is the last resort, as algorithm for optimization frequently need more computations to obtain information that are **not** required by root finding. Less efficient to do the same compared with root finding algorithms.
    - Do this only when you have no idea how the underlying equation can be solved.
- We cover 3 methods for root finding. All of the methods are quite fundamental, and can be hand-coded on one's own.

Bisection Method

- Based on **Intermediate Value Theorem** of continuous functions:
    1. Function $g(x)$ is continuous on the interval $x \in [a, b]$.
    2. Signs of the function are opposite on both ends, i.e., $g(a)g(b) < 0$.
    3. If 1 and 2 holds, then there exists some $x^* \in [a, b]$ such that $g(x^*) = 0$.
- Some caveats when applying Intermediate Value Theorem:
    - x^* might not be unique.
    - Having the same sign on both ends doesn't imply that x^* does not exist.
- Implication on root finding: we can approximate out an x^* by narrowing down the interval $[a, b]$ following a certain rule.
    1. If $g(a)g(b) < 0$, then find a midpoint $m_1 \equiv (b - a)/2$. WOLG assume that $g(a) < 0 < g(b)$.
    2. Check the sign of $g(m_1)$. If $g(m_1) > 0$ then use m_1 as the upper bound and keep the interval $[a, m_1]$. Otherwise use m_1 as the lower bound and keep the interval $[m_1, b]$. The root x^* must be falling in the new interval that we just have obtained here.
    3. Apply Steps 1 and 2 with respect to the new interval obtained in Step 2. Eventually we will have an interval $[a_c, b_c]$ such that the length is nearly 0. This interval **must** contain the root such that the root is approximated by the midpoint, i.e., $x^* \approx (b_c - a_c)/2$.


Step

3

Implementation on R

- Settings before computation:

    1. Determine the function to solve $g(x)$.
    2. Determine the interval to search for the root $I_0 \equiv [a, b]$.
    3. Determine the tolerance level ϵ to define "close enough / convergence".
    4. Determine the maximum amount of iteration $imax$ since the function might be ill-behaved such that the iteration never converges.

- Algorithm:

  Step 1. Check the signs of $g(a)$ and $g(b)$. Stop iteration if the signs are the same.

  Step 2. Find the midpoint $p \equiv (b - a)/2$. Check the sign of $g(p)$, and pick a new upper or lower bound for a new interval I_1:

    a. If $g(a)$ and $g(p)$ have different signs (hence $g(p)$ and $g(b)$ have the same sign), then use p as the new upper bound. That is, $I_1 \equiv [a_1, b_1]$ with $a_1 = a$ and $b_1 = p$.
    b. If $g(b)$ and $g(p)$ have different signs (hence $g(p)$ and $g(a)$ have the same sign), then use p as the new lower bound. That is, $I_1 \equiv [a_1, b_1]$ with $a_1 = p$ and $b_1 = b$.

  Step 3. Check the length of I_1 by $err \equiv |b_1 - a_1|$. If $err < tol$, the root is approximated by the midpoint $x^* \approx p$ hence we stop iteration (converged). Otherwise keep repeating Steps 2 and 3 until convergence or running out of iteration.

- To implement, you will need to do loops. Oftentimes we use `while`. But also see people using `for` or `repeat`.

- Need to use `if...else` to test if the conditions are meet, and `break` to leave the loop if converged.

- Remember to update the boundaries and looping counter.

- I have written a function `bisec` to perform one-dimensional bisection root finding for a general function. But I'm not showing you the code here. This is left as an in-class exercise.

- `bisec(fun, bnd, tol=1e-6, imax=1000, ...)`

    - `fun` is the function that you want to find the root. Potentially it takes not only the variable of interest, but also other parameters.

    - `bnd` is a list of length 3. The first element must be the name of the variable in interest (hence is a character). The 2nd and 3rd elements are endpoints of the interval to search for the root. For example, `bnd = list("x", -5, 5)`.
    - `tol` is a number that determines whether the solution is converged, defaulted to 10^{-6}.
    - `imax` is the maximum number of iteration, and is defaulted to 1000.
    - `...` is the parameter inputs in `fun`.
- `bisec` returns a named list of length 3 along with a message if either the algorithm converged at a root, or if `imax` is reached before converging to a root.
- The named list returned by `bisec` constains the following names and information:
    - `root` is the root that solves the function if converged, is the latest midpoint if convergence is not reached in `imax` iterations.
    - `val` is the value of the function evaluated at `root`. If the algorithm converges, we expect that `val` to be very close to 0.
    - `history` is a `data.frame` with 4 columns that documents the history of iterations.
        - Column 1 `i` is a vector starting from 0 to the number of iterations that ends the algorithm. `i=0` refers to the initial setting, and `i=1` and onward indicates the i-th time of iteration.
        - Columns 2 and 3 are `a` and `b`. Each are the vectors of the endpoints of the interval. Note that the first row is the initial setting.
        - Column 4 `m` is the midpoint between `a` and `b` in the same iteration. Note that the first row is the initial setting.
- `bisec` checks if the inputs are eligible, and report and error message if not.
- `bisec` gives an error message if the signs of values on the endpoints are the same.
- Try it out with $y = x + k$, where k is a parameter.

```r
funs<-function(x,k) {x+k} #Solution is x=-3
bisec(fun=funs, bnd=list("x",-5,5),k=3)
```

```
## Root occured at  -3  upon  25 -th iteration.
## $root
## [1] -3
##
## $val
## [1] 1.788139e-07
##
## $history
##     i         a         b         m
## 1   0 -5.000000  5.000000  0.000000
## 2   1 -5.000000  5.000000  0.000000
## 3   2 -5.000000  0.000000 -2.500000
## 4   3 -5.000000 -2.500000 -3.750000
## 5   4 -3.750000 -2.500000 -3.125000
## 6   5 -3.125000 -2.500000 -2.812500
## 7   6 -3.125000 -2.812500 -2.968750
## 8   7 -3.125000 -2.968750 -3.046875
## 9   8 -3.046875 -2.968750 -3.007812
## 10  9 -3.007812 -2.968750 -2.988281
## 11 10 -3.007812 -2.988281 -2.998047
## 12 11 -3.007812 -2.998047 -3.002930
## 13 12 -3.002930 -2.998047 -3.000488
## 14 13 -3.000488 -2.998047 -2.999268
## 15 14 -3.000488 -2.999268 -2.999878
## 16 15 -3.000488 -2.999878 -3.000183
## 17 16 -3.000183 -2.999878 -3.000031
## 18 17 -3.000031 -2.999878 -2.999954
## 19 18 -3.000031 -2.999954 -2.999992
## 20 19 -3.000031 -2.999992 -3.000011
## 21 20 -3.000011 -2.999992 -3.000002
## 22 21 -3.000002 -2.999992 -2.999997
## 23 22 -3.000002 -2.999997 -3.000000
## 24 23 -3.000002 -3.000000 -3.000001
## 25 24 -3.000001 -3.000000 -3.000000
## 26 25 -3.000000 -3.000000 -3.000000
```

```
#Solution right on endpoint. Still it goes through.
bisec(fun=funs, bnd=list("x",-4,-3),k=3)$root
```

```
## Root occured at  -3  upon  21 -th iteration.
```

```
## [1] -3
```

```
#Try out using a small number of imax so that the iteration ends before getting the actual root.
bisec(fun=funs, bnd=list("x",-5,5),k=3,imax=5)
```

```
## Maximum iteration reached, but not converging to a root.
## Returning the result of the last iteration as root along with history.
## $root
## [1] -2.8125
##
## $val
## [1] 0.1875
##
## $history
##   i      a    b       m
## 1 0 -5.000  5.0  0.0000
## 2 1 -5.000  5.0  0.0000
## 3 2 -5.000  0.0 -2.5000
## 4 3 -5.000 -2.5 -3.7500
## 5 4 -3.750 -2.5 -3.1250
## 6 5 -3.125 -2.5 -2.8125
```

- Consider the function $y = x^4 - 15x^2 + 8x + 30$ again. It has 4 roots:

$$x = \{3,\ 2.1055,\ -1.2235,\ -3.882\}$$

- Can we find the positive roots if we search between $[-1.5, 3.5]$? If not, where should we search?

```
#Can run into problems if the function has multiple solutions
#Interval contains 2 positive roots and a negative root but we only find the negative one. But why?
bisec(pf,bnd=list("x",3.5,-1.5))$root
```

```
## Root occured at  -1.223462  upon  29 -th iteration.
```

```
## [1] -1.223462
```

```
#Interval contains 2 positive roots, but can't start root finding since the endpoints have the same s
ign!
bisec(pf,bnd=list("x",1.5,3.5))$root
```

```
## [1] "Values of fun take the same sign on both ends of the interval."
```

```
## NULL
```

```
#Need to narrow down the interval so that the interval contains exactly one root.
bisec(pf,bnd=list("x",1.5,2.5))$root
```

```
## Root occured at  2.105483  upon  26 -th iteration.
```

```
## [1] 2.105483
```

```
bisec(pf,bnd=list("x",2.5,3.5))$root
```

```
## Root occured at  3  upon  1 -th iteration.
```

```
## [1] 3
```

- Debugging experiment: What if one of the endpoint contains solution, but there is another solution in the interval? Use the same function with $x = 3$ as the upper bound for experiment.

```
#Search in [0,3]. There are two roots within this interval, and 3 is one of it.
bisec(pf,bnd=list("x",0,3))$root
```

```
## Root occured at  2.105483  upon  27 -th iteration.
```

```
## [1] 2.105483
```

- The root $x = 3$ is missed out, meaning that my code does not check the value on the endpoint before starting the iteration. Still the other root is found.
- What if `m` also leads to `pf(m)=0` ? We could possibly run into a situation that the new interval to search is $x \in [m, 3]$ plus that `sign(m)=sign(3)=0` . If this happens the algorithm stops with an error and reports nothing. But this is certainly wrong since we actually have found the roots!
- Not an issue in my code because I check whether `pf(m)=0` upon getting `m` before constructing the new interval! If `pf(m)=0` happens, my code stops the iteration and report it as a root. This preclude the possibility of the previous point. Still a bit disappointing that we only get one out of the two roots.

Suggested Workflow and Tips

1. Plot out the function first to check where the roots are, and visually find the interval that contains the root to your interest. Deal with one and only one root at a time!

2. Do bisection to the interval you just visually constructed to get the root.

3. If possible, keep the iteration history so that you can figure out the problem if you don't get the root.

   For example, for some reasons you might run into complex number if the function to your interest has an $\sqrt{x}$, or run into evaluation error if the denominator of the function is very close to 0. You will need to check the history and the functional form to determine how to avoid the problem (say writing an `ifelse` to replace the problematic x with a random number).

- You might run bisection to the same function multiple times because you are also trying out different parameters to the function. Then most likely you use a loop for this. It is very unlikely that you make a plot each time you try out a new parameter: you probably have 100 different settings, with each setting can take up to `imax` of iterations! It is unlikely that you can fine-tune the 100 bisections!
- No really good answers for the above point. But the suggested workflow helps you to familiarize with the function you are working on, hence minimizes the possibility that you encounter problems after changing the parameters.
- Isn't it redundant if we are able to spot the location of the root before getting its number? No! You still need a number to proceed with your computation and follow-up analysis! Just by spotting the root visually is not enough.

Remarks

- There are also ready-to-use bisection functions in other programming languages. Usually these functions are smarter as they combine with other algorithms to speed up convergence or to refine the initial guess.
  - We require the function to take different signs at both endpoints, hence we might think of a U-shaped function like $y = x^2$ to have no roots while it actually has one. There are algorithms that search around the endpoints to refine the initial guess and prevent this problem to some extent.
  - The `fzero()` function in *Matlab* is based on bisection method, but one can provide only one initial point to it. The function will apply an algorithm to construct an interval including this initial guess with different signs at the endpoints.
- The `pracma` package ports `fzero()` on *Octave* to *R*. *Octave* can be think of a free version of *Matlab* as it tries to mimic the base functions of it. For `fzero()` , it is very similar to what we have in *Matlab*.
- Bisection method can be extended to a multivariable world, but it is beyond our scope.

    - The concept is simple: it is a multidimension version of the **Theorem of Nested Interval**. But implementing this idea for multidimensional root finding on computers is a big topic that worth for researches.

In-Class Exercise

- Build your own bisection algorithm. It doesn't need to be as delicate as mine, and there is no need to wrap it into a function. A one-time code composed of `while` and `if...else` suffices if the minimum requirements are meet:
    1. Can take any function with `x` being the name of the interested variable.
    2. Leaves flexibility for functions taking inputs other than `x`.
    3. Users can supply their preferred `tol` and `imax`.
    4. Checks if the signs of function value on the endpoints of the interested interval are the same, and stops execution if yes.
    5. Returns both the (possible) root and the function value evaluated at it. Also tells if the algorithm is converged when returning the results.

Newton-Raphson Method

- Before diving into root finding, recall that for a differentiable function $g(x)$ we can always approximate its behavior using a Taylor expansion at a point x_0. The most simple one is a linear approximation:

$$g(x) \approx g(x_0) + g'(x_0)(x - x_0)$$

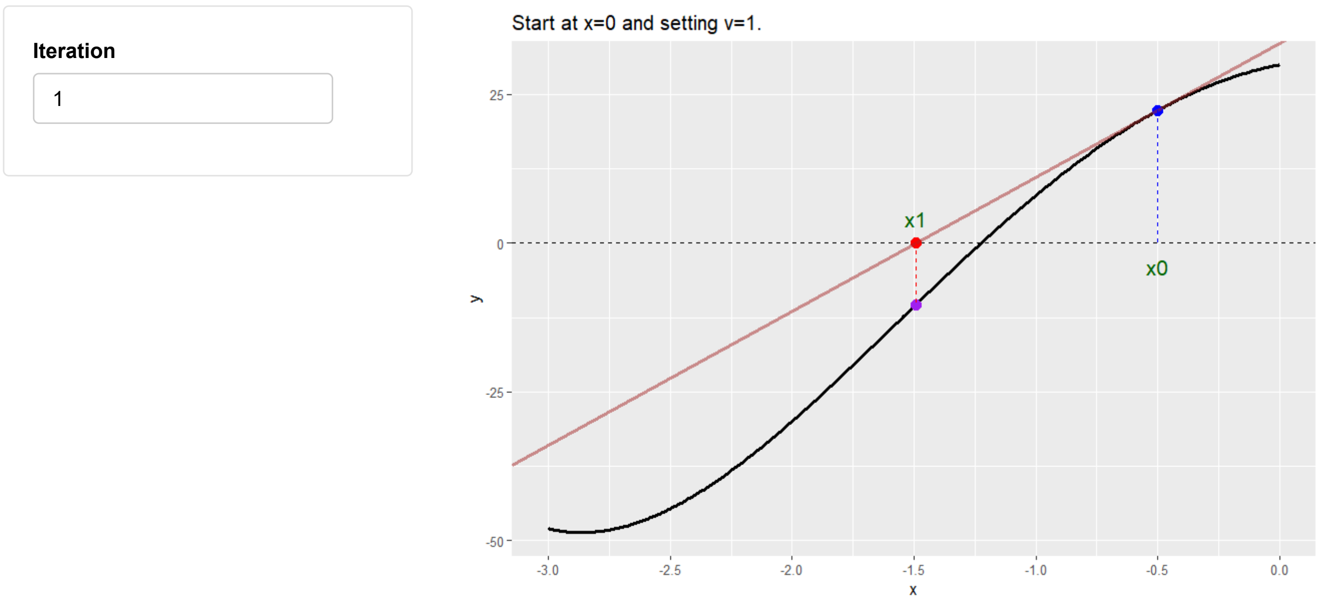
- If x happens to be a root, x^*, then $g(x) = 0$. But then this also means that the above linear approximation becomes

$$0 = g(x_0) + g'(x_0)(x^* - x_0)$$

  As a result, the root x^* is found by rearranging the equation above as

$$x^* = x_0 - \frac{g(x_0)}{g'(x_0)}$$

- **Newton-Raphson** algorithm is based on the idea above:
    - Guess an arbitrary x_0, use the above equation to compute a new point x_1.
    - Check if x_1 is the root. If yes, congratulations. If no, then use x_1 as the **new** guess and apply the same equation to get another new point x_2, and so on.
    - Keep repeating this process until $x_i \approx x_{i+1}$, meaning that $g(x_i) \approx 0 \approx g(x_{i+1})$ from the updating equation above. That is, we find the root if x barely changes any more.


Iteration

1

- Issue of Newton-Raphson:
    - It **does not** ensure convergence of solution! You may simply diverge, or bouncing back and forth between two points.
    - If the function has multiple roots, you might get an unexpected root even if you guess near the root you have in mind.
    - There is a simple refinement. We will come back to this soon.

Implementation on R

- Algorithm:

  Step 1. Pick any arbitrary x_0, and compute

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

  Step 2. Check if $|x_0 - x_1| < \epsilon$, where ϵ is the tolerance level.

    a. If yes, then we have the root.
    b. If not, then use x_1 as the new initial guess and keep repeating Steps 1 and 2 until convergence. The converged x is the root we are for.

- Again, you need to choose an initial point, set up tolerance level and maximum iteration.
- Also need to use flow control, i.e., `for/while/repeat` and `if...else`.
- Implementation below. There we have an additional parameter `v`, which is related to the refinement of Newton-Raphson algorithm.

```r
#Newton-Raphson

newtonraph<-function(fun,init,h=1e-6,tol=1e-6,imax=1000,v=1,...) {
  #Check eligibility of inputs
  if(!is.function(fun)) {stop("fun must be a function.")}
  if(!is.list(init)||length(init)!=2) {stop("init must be a list of length 2.")}
  if(!is.numeric(init[[2]])||length(init[[2]])!=1) {
    stop("Second element of init must be a numeric of length 1.")}
  if(!is.numeric(h)||length(h)!=1) {stop("h must be a numeric of length 1.")}
  if(!is.numeric(tol)||length(tol)!=1) {stop("tol must be a numeric of length 1.")}
  if(!is.numeric(imax)||length(imax)!=1) {stop("imax must be a numeric of length 1.")}
  if(!is.numeric(v)||length(v)!=1) {stop("v must be a numeric of length 1.")}
  if(!init[[1]]%in%formalArgs(fun)) {stop("Variable to evaluate does not exist.")}

  #Set up input to be passed to fun.
  arg<-c(init[[2]],...)%>%as.list()
  names(arg)[[1]]<-init[[1]]
  #Prepare for 1st derivative with Central Rule
  argf<-arg
  argf[[1]]<-argf[[1]]+(h/2)
  argb<-arg
  argb[[1]]<-argb[[1]]-(h/2)

  #Initialize Iteration
  i<-1
  hist<-data.frame(i=c(1:imax),x0=rep(NA,imax),val0=rep(NA,imax),slope=rep(NA,imax),
                   x1=rep(NA,imax),diff=rep(NA,imax),val1=rep(NA,imax),jump=rep(0,imax))

  while (i<=imax) {
    val<-do.call(fun,arg) #Evaluate at initial level x_i.
    slp<-(do.call(fun,argf)-do.call(fun,argb))/h #Evaluate slope at initial point.

    #Handles 0-slope issue by making a random shift in an arbitrarily small interval around the initi
al point. We use h as the radius of interval, but one might want to make it adjustable instead of har
d-coded.
    #Note that we still use the point computed above, just that the we use a new slope so it is a sec
ant line instead of a tangent line.
    if (slp==0) {
      cat("0 slope encountered, compute a new slope by making a random shift by radius ",h, ".\n")
      arg[[1]]<-sample(c(arg[[1]]-h,arg[[1]]+h),1)
      argf[[1]]<-arg[[1]]+(h/2)
      argb[[1]]<-arg[[1]]-(h/2)
      slp<-(do.call(fun,argf)-do.call(fun,argb))/h

      hist$jump[i]<-1 #Record this event in history.
    }

    arg.new<-arg[[1]]-v*val/slp #Get new level x_(i+1) with a convergence factor v.

    diff<-abs(arg[[1]]-arg.new) #Compute error.

    #Record the results
    hist$x0[i]<-arg[[1]]
    hist$val0[i]<-val
    hist$slope[i]<-slp
    hist$x1[i]<-arg.new[[1]]
    hist$diff[i]<-diff
    if (i>1) {hist$val1[i-1]<-val} #Treat the value evaluated at the new point in the previous round
as backlog.

    if (diff<tol) {
      cat("Iteration converged upon ",i,"-th iteration, with a root at ",names(arg[[1]]),"= ",arg.ne
w,".\n")
      hist$val1[i]<-val
      return(list(root=arg.new,val=val,history=na.omit(hist)))
    }

    #Update initial point
    arg[[1]]<-arg.new
    argf[[1]]<-arg.new+(h/2)
    argb[[1]]<-arg.new-(h/2)

    if (i==imax) {
      cat("Iteration ended before convergence. Returning the last evaluation as root.\n")
      hist$val1[i]<-do.call(fun,arg)
      return(list(root=arg.new,val=hist$val1[i],history=na.omit(hist)))
    }

    i<-i+1

  }

}

#Guess at x=-0.5. Get the solution near -1.2
newtonraph(pf,init=list("x",-0.5))
```

```
## Iteration converged upon  5 -th iteration, with a root at  =  -1.223462 .
```

```
## $root
## [1] -1.223462
##
## $val
## [1] -1.716387e-10
##
## $history
##   i        x0          val0    slope        x1         diff          val1 jump
## 1 1 -0.500000  2.231250e+01 22.50000 -1.491667 9.916667e-01 -1.035844e+01    0
## 2 2 -1.491667 -1.035844e+01 39.47375 -1.229253 2.624134e-01 -2.166674e-01    0
## 3 3 -1.229253 -2.166674e-01 37.44768 -1.223467 5.785870e-03 -1.995868e-04    0
## 4 4 -1.223467 -1.995868e-04 37.37852 -1.223462 5.339612e-06 -1.716387e-10    0
## 5 5 -1.223462 -1.716387e-10 37.37846 -1.223462 4.591882e-12 -1.716387e-10    0
```

```
#Guess at x=0, and we run into the root -3.88 instead of the one near -1.2
rs.out<-newtonraph(pf,init=list("x",0))
```

```
## Iteration converged upon  5 -th iteration, with a root at  =  -3.882021 .
```

```
rs.out
```

```
## $root
## [1] -3.882021
##
## $val
## [1] 1.246272e-06
##
## $history
##   i        x0          val0     slope        x1         diff          val1 jump
## 1 1  0.000000  3.000000e+01    8.0000 -3.750000 3.750000e+00 -1.318359e+01    0
## 2 2 -3.750000 -1.318359e+01  -90.4375 -3.895776 1.457757e-01  1.521177e+00    0
## 3 3 -3.895776  1.521177e+00 -111.6326 -3.882149 1.362664e-02  1.408428e-02    0
## 4 4 -3.882149  1.408428e-02 -109.5683 -3.882021 1.285435e-04  1.246272e-06    0
## 5 5 -3.882021  1.246272e-06 -109.5489 -3.882021 1.137640e-08  1.246272e-06    0
```

```
#Use a new iteration factor. Choose v=0.7 to make the adjusted slope flatter.
#Get the solution near -1.2 this time
rs.conv<-newtonraph(pf,init=list("x",0),v=0.7)
```

```
## Iteration converged upon  14 -th iteration, with a root at  =  -1.223462 .
```

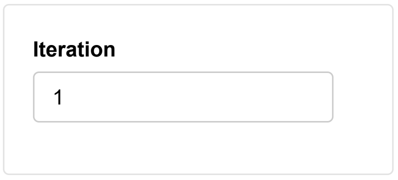
```
rs.conv[[1]]
```

```
## [1] -1.223462
```

```
rs.conv[[2]]
```

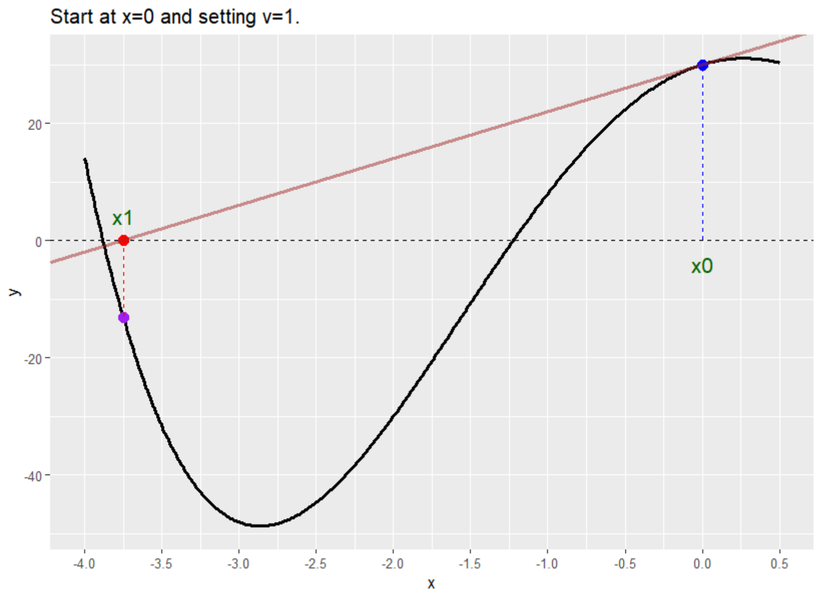
```
## [1] -2.327973e-05
```

- What happened to the case that `init=0` ? The slope is too flat such that the iteration leads to a new x that is too far away from the root we want! We refer to this situation as **overshoot**.

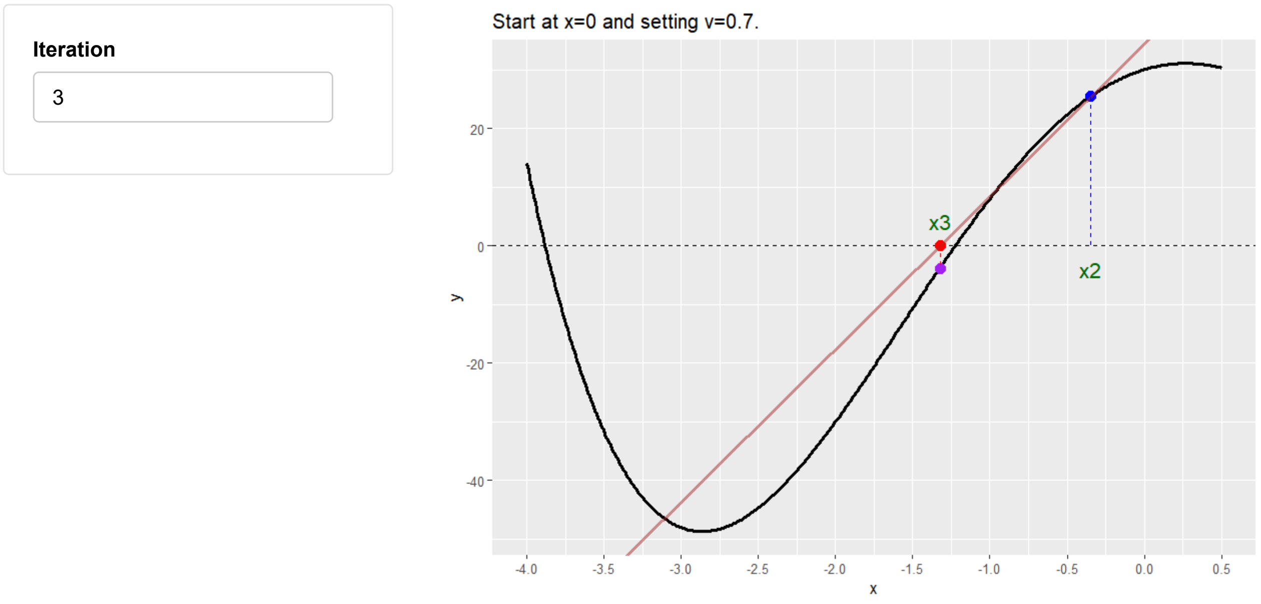


Iteration

1



- Then what is the role of `v` ? The line we use to search for the root becomes steeper, hence the new guess gets closer and closer to the desired root.



Iteration

3

- Setting `v=0.7` gives us what we want by making the slope steeper, hence a smaller step size of iteration. But why can we do that?

- The updating equation of Newton-Raphson comes from linear approximation:

$$g(x) = g(x_0) + g'(x_0)(x - x_0)$$

- If we perform iteration to updtae x_0 and converge to x^*, then $x_0 \approx x^*$ hence the above becomes $0 = g(x^*) \approx g(x_0)$. The second term **does not** matter.

- If the second term doesn't matter if we guess it right, then we can modify Newton-Raphson by adding an additional multiplier v, so that

$$g(x) = g(x_0) + \frac{g'(x_0)}{v}(x - x_0)$$

  If x^* is the root so that $g(x^*) = 0$, we have

$$x^* = x_0 - v\frac{g(x_0)}{g'(x_0)}$$

  Therefore we can use an alternative updating equation:

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

  instead of the original one.

- The $v > 0$ here is what we refer to as the **convergence factor**.

- Setting $v = 1$ is exactly to the original Newton-Raphson method that uses the **tangent line** at point x_0 for iteration.

- Setting $v \neq 1$ means that we use **secant lines** instead of tangent lines, as we have shown visually.

- Higher v leads to a **steeper** secant line, hence a **smaller** step size from the initial guess since $|x_1 - x_0| = |v\frac{g(x_0)}{g'(x_0)}|$. The searching becomes more **aggressive**.

- Smaller v entails a **flatter** secant line and a **larger** step size. Root searching becomes more **conservative**.

- Adjust v to adjust the step size. Helps you breaking out from non-convergence.

```
#Use a new iteration factor. Choose v=2 to make the adjusted slope steeper.
rs.div<-newtonraph(pf,init=list("x",0),v=2)
```

```
## Iteration ended before convergence. Returning the last evaluation as root.
```

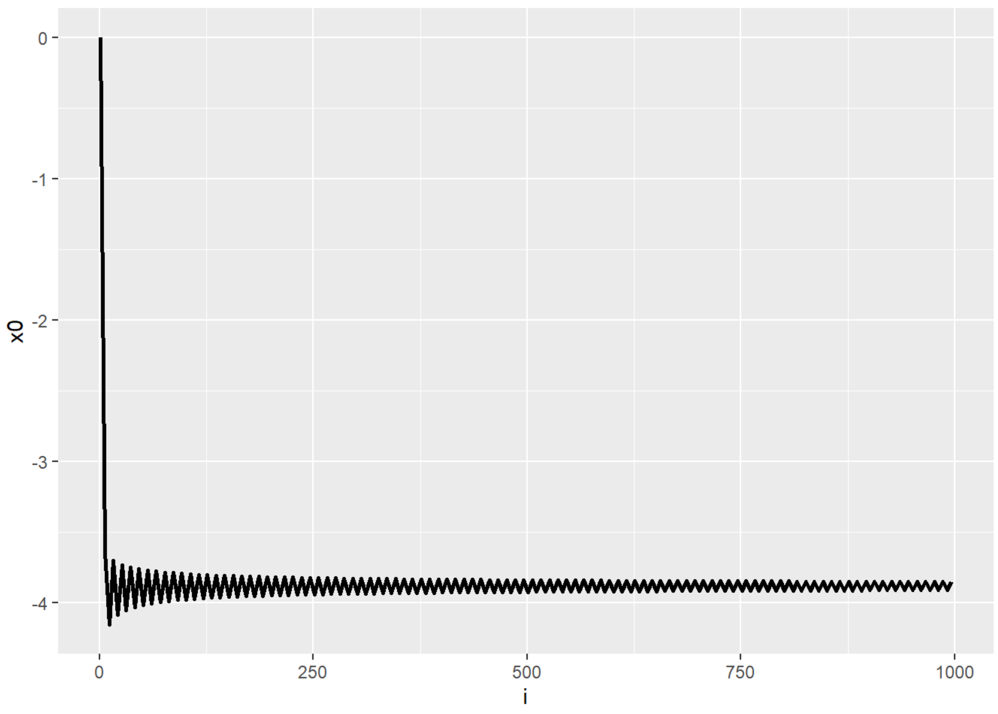
```
rs.div[[1]]
```

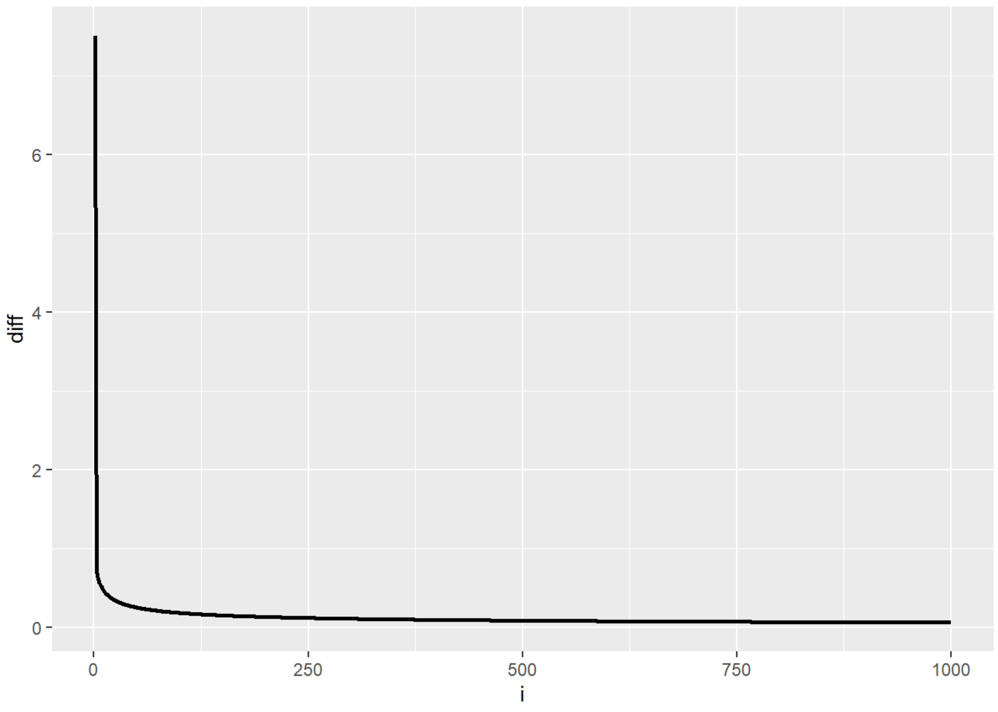
```
## [1] -3.912149
```

```
rs.div[[2]]
```

```
## [1] 3.369444
```

- What happened to the case using `v=2` ?





- The algorithm seems to be **trapped**. You can think of this as a result of overshooting: At first your new x goes beyond the root, say to the left of it. So you do a new iteration to pull it back. But again you pull back too much so that x is now to the right of the root with a long distance. All of your iterations are overshooting to the opposite directions.
- After a certain amount of iteration, the algorithm starts to jump back and forth between two xs. The pattern thus looks similar to a wave like $sinx$: it goes up and down within an interval but its limit is undefined hence no convergence at all.
- Think of one of the situation in the Cobweb Theory! The computation of its equilibrium is characterised by the case here.
- Seeing a back-and-forth pattern graphically does NOT mean that we never have a convergence. My hunch is that in this particular case the iteration can still converge to the root near -3.88. But the rate of convergence is so slow hence it takes "forever" to reach the root.

Suggested Workflow and Tips

1. Plot the function of interest out to get some idea about where to initialize the algorithm.
2. First initialize the algorithm with `v=1` as in vanilla Newton-Raphson and `imax=1000`. If it converges to the desired root, good.
3. What if the algorithm reports no convergence? Then we check the path of `x` and `diff`.
    - If it seems to be converging or just jumping back-and-forth, then it probably means that your algorithm is overshooting the solutions. In this case, try to use a smaller convergence factor `v` to lessen the overshooting problem.

      A **higher** `v` flattens the line by pivoting it around the initial point, thus the algorithm searches in a more **aggressive** manner that every iteration finds a new x with a large step size from the current point. This is desired if we want to search for a long range. In contrast, a **lower** `v` makes the algorithm to be more **conservative**, hence we are searching near the initial point and move **gradually**. This is desired if we suspect that the root we want is near the initial guess, or that the algorithm is overshooting.

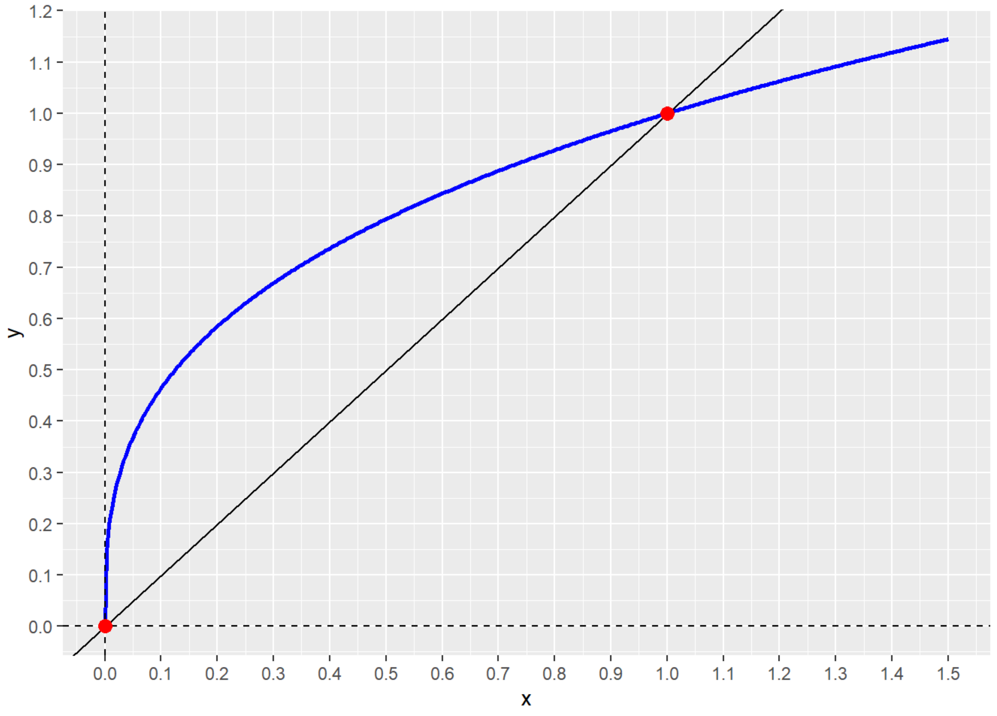
      Note that you should NOT try to increase `imax` here. Most likely it is just a waste of time since the rate of convergence is nearly 0 or doesn't even exist.
    - If the pattern suggest convergence and shows no signs of being trapped, then either increase `imax` and be patient, or use a higher `v` to speed up the algorithm.
    - Seeing `diff` getting higher and / or `x` being highly unstable such that it doesn't even fluctuate within an interval. This probably means **divergence**. One can try to use a smaller `v` or simply change a initial point.
    - In all the cases, changing an initial point is always an option. But I'd suggest to adjust `v` first since you choose the initial point according to the plot in the first place.

Remarks

- Not very difficult to hand code from scratch.
- Flexible, can be extended to multidimensional root finding.
- Differentiability is required.
- Might not converge to what you want, e.g., overshooting, divergence, unexpected roots.
- Use convergence factor to control the step size.

Fixed Point Iteration (Contraction Mapping)

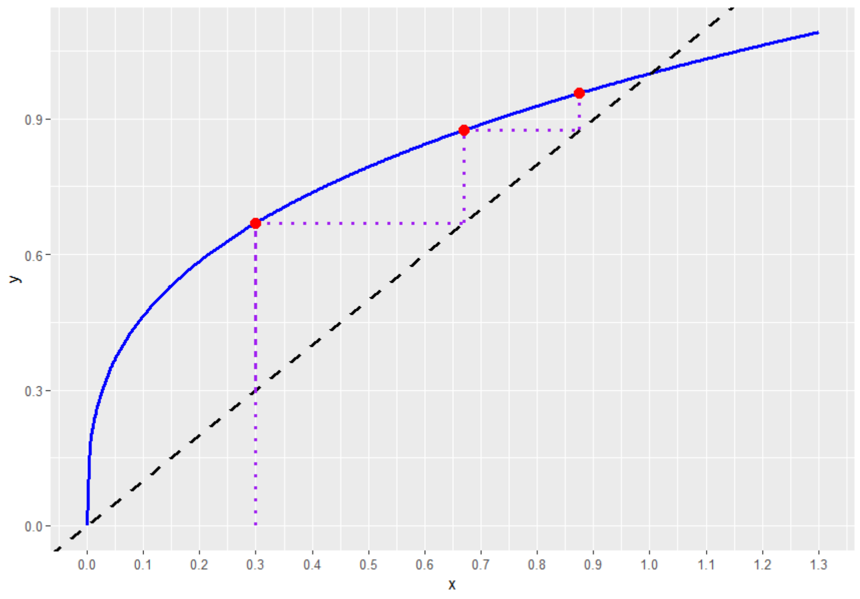
- **Fixed Point:** A point of a function that maps to itself. More formally, a fixed point x^* of a function $h(x)$ is such that $h(x^*) = x^*$.
- The following draws $y = x(1/3$ in the $[0,1]^2$ space. Points $(0,0)$ and $(1,1)$ are fixed point since their self-mappings are themselves.
  - Plug $x = 1$ into $x^{1/3}$ gives you $y = 1$, the same goes for $x = 0$. But NOT other points!



- A fixed point problem is to solves the problem $h(x) = x$.
- Root finding problem is in fact a special case of fixed point problem:
  - Root finding solves $g(x) = 0$.
  - Define $h(x) \equiv g(x) - x$, hence the the root finding problem is equivalent to solving $x = g(x) - x = h(x)$.
- Fixed point iteration:
  1. Pick an arbitrary starting point x_0, plug into $h(x)$ to get y_0.
  2. Check if $x_0 = y_0$. If yes, good. If no, then use y_0 as a new guess x_1 and plug into $h(x)$ to get y_1 and so on.
  3. Repeat until we find a point x^* such that $x^* = h(x^*)$, i.e., the fixed point.

Iteration

3



- Why can we do this? Because of **Contraction Mapping Theorem**.
    - **Contraction Mapping:** Consider a mapping $f(.)$ and two points x and y. The mapping $f(.)$ is called a contraction mapping if there exists a $k \in [(]0, 1)$ such that $|f(x) - f(y)| \leq k|x - y|$.

  If $f(.)$ is a continuous function, then this is equivalent to that $|f'| < 1$.

    - **Banach Fixed Point Theorem:** Also referred to as Contraction Mapping Theorem. In a simplified manner, this theorem says that in a non-empty complete space with a contraction mapping, there always exists a fixed point x^*. Moreover, sequence generated from the contraction mapping $\{x_n\}$ converges to this fixed point.
    - For details, take a Real Analysis course.
- The fixed point iteration, also known as contraction mapping iteration, is an implementation to Banach Fixed Point Theorem.

  If $h(x)$ is a contraction mapping ($h'(x) < 0$), then the initial guess and updates are the sequences generated from this contraction mapping, and Banach Fixed Point Theorem guarantees that eventually we get the fixed point if we do enough of iterations.

- In short:
    1. One can convert a root finding problem into a fixed point problem.
    2. **Require $h'(x) < 0$ for fixed point iteration to work.**
    3. Because of 2, one can use a convergence factor when reformulating the root finding problem to ensure that the mapping becomes a contraction mapping.

Implementation on R

- Algorithm is very very similar to Newton-Raphson, but there are lots of details that can very by a lot between cases.
- First need to set up an initial guess, tolerance level ϵ, maximum number of iteration, and supply the convergence factor if needed.
- General algorithm:

  Step 1. Plug the initial guess x_0 into the function to get a new guess as $x_1 = h(x_0)$.

  Step 2. Check if $|x_0 - x_1| < \epsilon$:

    a. If yes, then x_1 is the fixed point x^*.
    b. If no, then compute $x_2 = h(x_1)$ and repeat Steps 1 and 2 until x converges. The converged x is the fixed point we are for.
- Look into 3 different cases to see how we can apply contraction mapping out in the field.

Simple Root Finding

- We use contraction mapping to solve the problem $x^{\frac{1}{3}} - x = 0$. The code is not provided for this simple example, and is left as your assignment back home.

```
## Algorithm converged upon 14 -th iteration at 0.9999992
## $root
## [1] 0.9999992
##
## $val
## [1] 0.9999997
##
## $history
##     i        x0      val0        x1         diff
## 1   1 0.3000000 0.6694330 0.6694330 3.694330e-01
## 2   2 0.6694330 0.8747871 0.8747871 2.053541e-01
## 3   3 0.8747871 0.9563880 0.9563880 8.160092e-02
## 4   4 0.9563880 0.9852461 0.9852461 2.885806e-02
## 5   5 0.9852461 0.9950576 0.9950576 9.811573e-03
## 6   6 0.9950576 0.9983498 0.9983498 3.292190e-03
## 7   7 0.9983498 0.9994496 0.9994496 1.099815e-03
## 8   8 0.9994496 0.9998165 0.9998165 3.668743e-04
## 9   9 0.9998165 0.9999388 0.9999388 1.223213e-04
## 10 10 0.9999388 0.9999796 0.9999796 4.077711e-05
## 11 11 0.9999796 0.9999932 0.9999932 1.359274e-05
## 12 12 0.9999932 0.9999977 0.9999977 4.530954e-06
## 13 13 0.9999977 0.9999992 0.9999992 1.510323e-06
## 14 14 0.9999992 0.9999997 0.9999997 5.034414e-07
```

Solving a Partial Equilibrium

- The demand and supply function for cupcakes are respectively given as follows:

$$q_D = 10 - 2p$$

$$q_S = 2 + 5p$$

  where the subscript D and S refers to demand and supply respectively.

- The excess demand of this market is defined by the difference between quantity demanded and quantity supplied $ED \equiv q_D + q_S$ as

$$ED = 8 - 7p$$

- Econ 101 says that in a market equilibrium the price p^* is such that no excess demand $ED = 0$.

- We can solve this system by contraction mapping using the following updating equation

$$p^{(i+1)} = p^{(i)} + vED^{(i)} = p^{(i)} + v(8 - 7p^{(i)})$$

  where v is the convergence factor, and i denotes for the i-th iteration. That is, $p^{(i)}$ is our guess to p^* in the i-th iteration and we get $p^{(i+1)}$ for our next iteration. This approach is similar to that in Lucas and Alvarez (2007) and has been widely used in quantitative trade models.

- If we do reach the equilibrium, the the above updating equation automatically becomes $p^* = p^*$ since $ED = 0$ must hold.

- This time we wrap it into a function so that we can demonstrate how the convergence factor matters here.

```r
#Define the function
ED<-function(p) {8-7*p}

#By default we guess at p=1.2, setting v=1, tol=1e-6 and iterate for 15 times at most
#Note that ED is already hard coded into this solver.
PEsolve<-function(p0=1.2,v=1,tol=1e-6,imax=15) {
  #Initialize algorithm
  i<-1
  hist.con<-data.frame(i=c(1:imax),p0=rep(NA,imax),ED0=rep(NA,imax),
                     p1=rep(NA,imax),diff=rep(NA,imax))

  while (i<=imax) {

  ed0<-ED(p0) #Evaluate the function at x0.
  p1<-p0+v*ed0 #By the algorithm, convergence is such that f(x0)=x0==x1.
  diff<-abs(p0-p1) #Define the error.

  #Document history
  hist.con$p0[i]<-p0
  hist.con$ED0[i]<-ed0
  hist.con$p1[i]<-p1
  hist.con$diff[i]<-diff

  if (diff<tol) {
    cat("Algorithm converged upon",i,"-th iteration at",p0,"\n")
    return(list(root=p0,val=ed0,history=na.omit(hist.con))) #Output
    break #Break from the loop if convergence is achieved.
  }

  if (i==imax) {
    cat("Algorithm ended before convergence. Returning the results.\n")
    return(list(root=p0,val=ed0,history=na.omit(hist.con))) #Output but not converged.
  }

  p0<-p1 #Update initial guess
  i<-i+1
}

}

#Run the solver using the default settings.
pev1<-PEsolve()
```

```
## Algorithm ended before convergence. Returning the results.
```

```r
print(pev1$history)
```

```
##     i           p0           ED0            p1         diff
## 1   1          1.2 -4.000000e-01  8.000000e-01 4.000000e-01
## 2   2          0.8  2.400000e+00  3.200000e+00 2.400000e+00
## 3   3          3.2 -1.440000e+01 -1.120000e+01 1.440000e+01
## 4   4        -11.2  8.640000e+01  7.520000e+01 8.640000e+01
## 5   5         75.2 -5.184000e+02 -4.432000e+02 5.184000e+02
## 6   6       -443.2  3.110400e+03  2.667200e+03 3.110400e+03
## 7   7       2667.2 -1.866240e+04 -1.599520e+04 1.866240e+04
## 8   8     -15995.2  1.119744e+05  9.597920e+04 1.119744e+05
## 9   9      95979.2 -6.718464e+05 -5.758672e+05 6.718464e+05
## 10 10    -575867.2  4.031078e+06  3.455211e+06 4.031078e+06
## 11 11    3455211.2 -2.418647e+07 -2.073126e+07 2.418647e+07
## 12 12  -20731259.2  1.451188e+08  1.243876e+08 1.451188e+08
## 13 13  124387563.2 -8.707129e+08 -7.463254e+08 8.707129e+08
## 14 14 -746325371.2  5.224278e+09  4.477952e+09 5.224278e+09
## 15 15 4477952235.2 -3.134567e+10 -2.686771e+10 3.134567e+10
```

- This time we use the same setting except that we adjust the convergence factor to 0.1.

```
pev01<-PEsolve(v=0.1)
```

```
## Algorithm converged upon 10 -th iteration at 1.142858
```

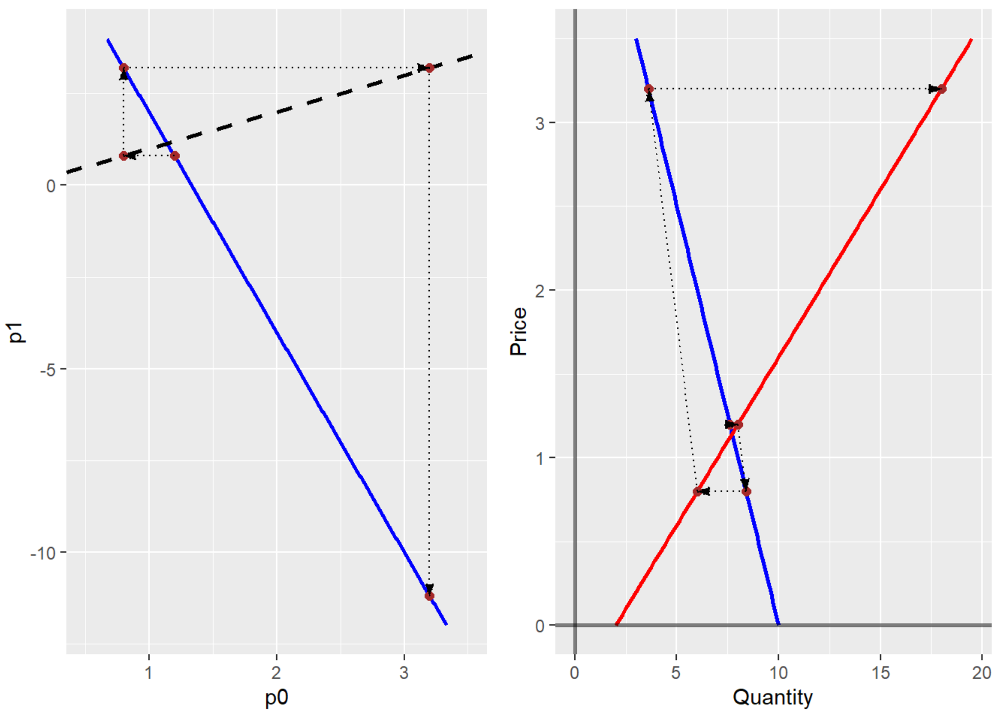
```
pev01
```

```
## $root
## [1] 1.142858
##
## $val
## [1] -7.8732e-06
##
## $history
##     i       p0         ED0       p1       diff
## 1   1 1.200000 -4.0000e-01 1.160000 4.0000e-02
## 2   2 1.160000 -1.2000e-01 1.148000 1.2000e-02
## 3   3 1.148000 -3.6000e-02 1.144400 3.6000e-03
## 4   4 1.144400 -1.0800e-02 1.143320 1.0800e-03
## 5   5 1.143320 -3.2400e-03 1.142996 3.2400e-04
## 6   6 1.142996 -9.7200e-04 1.142899 9.7200e-05
## 7   7 1.142899 -2.9160e-04 1.142870 2.9160e-05
## 8   8 1.142870 -8.7480e-05 1.142861 8.7480e-06
## 9   9 1.142861 -2.6244e-05 1.142858 2.6244e-06
## 10 10 1.142858 -7.8732e-06 1.142857 7.8732e-07
```

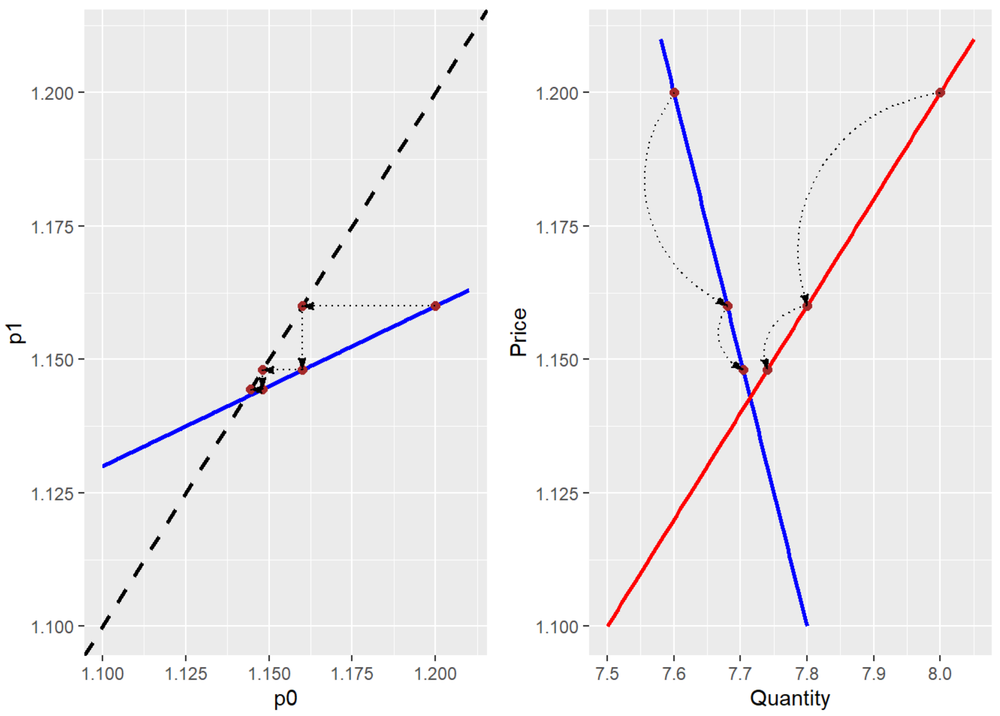
- Why does the first attempt with `v=1` fails but the second attempt with `v=0.1` ? Recall the Cobweb Theory you have learned back in Econ 101. The condition for contraction mapping fails to hold when `v=1` .

```
## Warning: Removed 56 row(s) containing missing values (geom_path).
```

```
## Warning: Removed 1 rows containing missing values (geom_point).
```



- By changing `v=0.1`, we find a new updating rule such that contraction mapping holds.



General Equilibrium Model

- The purpose of this very advanced example is to show you how flexible that contraction mapping is in solving a multidimensional model.
- Perfectly fine that you can't comprehend most of the details. What you really need to take note is how I apply a two-layer contraction mapping algorithm, and the rationale behind forming the updating equations. The idea of coding is exactly the same as the previous example.

Basic Settings

- Two countries indexed by $k = \{A, B\}$.
- Each country has a monopolistic firm producing a country-specific goods.
- Each country has exactly one representative agent, who consume goods produced by both countries and inelastically supplies one unit of labor in the domestic labor market.

- In country k, the demand for its domestic product and the imported foreign products are

$$q_{kk} = I_k p_k^{-\sigma};\ q_{kj} = I_k p_j^{-\sigma}$$

  where I_k is the income in country k, p_k is the price of goods produced in country k, and $\sigma > 1$ is demand elasticity.
    - Note that the subscript kj refers to country k's import from country j. If $k = j$ then it means domestic import (domestically produced and consumed).
- Production technology is linear, such that the labor demand

$$l_k = \frac{Q_k}{\varphi_k}$$

  where φ_k is the productivity in country k and Q_k is the total output in country k.
- Specifically, $Q_k = q_{kk} + q_{jk}$. That is, some of the outputs sold to local agents while some other outputs are exported.
- The price of the country-specific product is the same in both countries, and the firm determines this price as a monopolist. The profit of the firm in country k is thus

$$\pi_k = p_k\left(q_{kk} + q_{jk}\right) - \frac{w_k}{\varphi_k}(q_{kk} + q_{jk})$$

  where p_k is the price of the product produced by country k, and w_k is the wage level in country k.

Equilibrium Conditions

- It is readily checked that the equilibrium prices and quantities are

$$p_k = (\frac{\sigma}{\sigma-1})\frac{w_k}{\varphi_k};\ q_{kk} = I_k(\frac{\sigma}{\sigma-1}\frac{w_k}{\varphi_k})^{-\sigma};\ q_{jk} = I_j(\frac{\sigma}{\sigma-1}\frac{w_k}{\varphi_k})^{-\sigma}$$

- Let X_k be the total **sales revenue** in country k, and let $p_k q_{kk} \equiv X_{kk}$ and $p_k q_{jk} \equiv X_{jk}$ be country k's sales in both country k and j. It is readily verified that

$$X_k = X_{kk} + X_{jk};\ X_{kk} = \left(\frac{\sigma}{\sigma-1}\frac{w_k}{\varphi_k}\right)^{1-\sigma} I_k;\ X_{jk} = \left(\frac{\sigma}{\sigma-1}\frac{w_k}{\varphi_k}\right)^{1-\sigma} I_j$$

- Since the agent is representative, his income equals to labor income $w_k(q_{kk} + q_{jk})/\varphi_k$ and profit π_k. This implies that the income of the agent in country k is exactly the total sales of the firm:

$$I_k = X_k$$

- Agent in country k consumes both products. Let E_k be the total **expenditure**, it must equal to the sum of **expenditure** on both domestic and imported products as

$$E_k = X_{kk} + X_{kj}$$

- Goods Market Clearing must hold:

$$X_k = E_k$$

- Labor supply is inelastic as 1, hence labor market clearing becomes

$$\frac{\sigma-1}{\sigma}X_k = w_k$$

  where $\frac{\sigma-1}{\sigma}X_k$ is the wage bill paid by the the firm to the agent and w_k is the labor income.
- The above conditions imply that trade is balanced: $X_{jk} = X_{kj}$.

Model Solving

- We have 4 variables to solve: $\{I_A, I_B, w_A, w_B\}$.
- Idea of computation: We solve by a two-layer contraction mapping where the outer loop solves for wages with labor market clearing, and the inner loop solves for income using goods market clearing.

1. Guess w_A and w_B as $w_A^{(i)}$ and $w_B^{(i)}$. Then go to the inner loop.
   a. Given the guesses to wages, guess income $I_A^{(j)}$ and $I_B^{(j)}$ so that we can solve for X_A, X_B, E_A and E_B.
   b. Use goods market clearing conditions $E_A - X_A$ and $E_B - X_B$ to update income until convergence. Denote the converged income by $I_A^{(i)}$ and $I_B^{(i)}$. We pass them back to the outer loop.
2. Use the income obtained in the inner loop to update wages by labor market clearing conditions for both countries. Specifically, we update until convergence by the Lucas-Alvarez approach as

$$w_k^{(i+1)} = w_k^{(i)} + v(\frac{\sigma-1}{\sigma}X_k^{(i)} - w_k^{(i)})$$

   where the second term is the excess demand in the labor market.

- Since we have 2 countries, the error is determined by the **maximum difference** among the two countries. That is, the model converges if the least convergent variable is converged.

- Assume that $\sigma = 2$, $\varphi_A = 1$ and $\varphi_B = 2$ for simplicity. We can thus boil down the goods market clearing condition and the excess demand of the labor market EDL as

$$I_i = \frac{\left(\frac{w_i}{\varphi_i}\right)^{1-\sigma}}{\left(\frac{w_i}{\varphi_i}\right)^{1-\sigma} + \left(\frac{w_j}{\varphi_j}\right)^{1-\sigma}}(I_i + I_j)$$

  and

$$EDL_k = \frac{I_k}{\varphi_k}\left(\frac{\sigma}{\sigma-1}\right)^{-\sigma}\left[\left(\frac{w_k}{\varphi_k}\right)^{-\sigma} + \left(\frac{w_j}{\varphi_j}\right)^{-\sigma}\right] - 1$$

- We can construct the updating rules for the inner loop as

$$I_k^{(l+1)} = \frac{\frac{\varphi_k}{w_k^{(i)}}}{\frac{\varphi_k}{w_k^{(i)}} + \frac{\varphi_j}{w_j^{(i)}}}\left(I_k^{(l)} + I_j^{(l)}\right)$$

  and for the outer loop as

$$w_k^{(i+1)} = w_k^{(i)} + v\left(\frac{I_k^{(i)}}{\varphi_k}\frac{1}{4}\left[\left(\frac{\varphi_k}{w_k^{(i)}}\right)^2 + \left(\frac{\varphi_j}{w_j^{(i)}}\right)^2\right] - 1\right)$$

Implementation

- The goods market clearing condition shows that the share of a country's income to the whole economy is exactly the country's share of productivity adjusted wage to the economy. Since $\varphi_A = 1$ and $\varphi_B = 2$, it is reasonable to initialize the computation with a guess such that $I_B = 2I_A$ and $w_B = 2w_A$.

```r
#Modified Krugman (1980) Model
#Initialize Algorithm
tol<-1e-6
imax<-1e+5

i<-1 #For outter loop
v<-0.5
#Guess income and wages for both countries in vectors.
I.init<-c(1,2)
w.init<-c(1/3,2/3)
phi<-c(1,2) #The functions are symmetric, so plugging in the numbers here is easier for coding.
pw<-phi/w.init

while (i<=imax) {
  j<-1 #For inner loop.
  #cat("i: ",i,"\n.")
  #cat("I.init:",I.init,"\n.")
  #cat("w.init:",w.init,"\n.")

  #Inner loop to iterate for income.
  while (j<=imax) {

    I.new<-pw*(sum(I.init)/sum(pw)) #Iterate for new income.
    diff.I<-max(abs(I.init-I.new)) #Pick the largest one as the error.

    if (diff.I<tol) {
      #cat("Income converged.\n")
      I.out<-I.new
      break
      }

    if (j>imax) { #Warn if not converged.
      cat("Income does not converge.\n")
      I.out<-c(NA)
      break
      }

    I.init<-I.new #Update
    j<-j+1

    }

  #Report if the inner loop fails.
  if (sum(is.na(I.out))!=0) {
    cat("Model does not converge since the inner loop fails to converge.\n")
    break
  }

  w.new<-w.init+v*((I.out/phi)*(1/4)*sum(pw^2)-1) #Iterate for new wage.
  diff.w<-max(abs(w.init-w.new)) #Use the largest difference as distance


  if (diff.w<tol) { #Return converged income to outer loop as I.out
    cat("Wage converged upon",i,"-th iteration.\n")
    w.out<-w.new
    break
    }
  if (i>imax) { #Warn if not converged.
    cat("Wage does not converge.\n")
    w.out<-c(NA)
    break
    }

  w.init<-w.new #Update initial guess on income
  pw<-phi/w.init
  i<-i+1

}
```

```
## Wage converged upon 30 -th iteration.
```

```
w.out
```

```
## [1] 1.118035 1.118034
```

```
I.out
```

```
## [1] 0.999999 2.000001
```

Suggested Workflow and Tips

- Very similar to Newton-Raphson method in terms of choosing initial points and convergence factor.
- Forming the updating equation is the main challenge:
  - If the equation $g(x) = 0$ can be easily written as $x = h(x)$ with $|h'(x)| < 1$, one could use $x^{(i+1)} = h(x^{(i)})$ as the updating equation. This is exactly the approach in the first example and the income updating rule in the last example.
  - If $|h'(x)| >= 1$, then one could introduce the convergence factor into the updating equation as

    $$x^{(i+1)} = x^{(i)} + v(x^{(i)} - h(x^{(i)}))$$

    and then use the convergence factor to "force out" a contraction mapping. If the solution do exist, then the above updating rule becomes $x = x$ as desired.
  - If the equation does not have an obvious contraction mapping pattern, then construct the updating rule as

    $$x^{(i+1)} = x^{(i)} + v(g(x^{(i)}))$$

    This is similar to the above one. The way we deal with the excess demand in the second example and in the last example belongs to this approach.
  - Construct the updating rule with economics thinking. For example, we say that the excess demand is defined as $q_d - q_s$ and vanishes as we reduce the price starting from a high level. But what if we write $q_s - q_d$ in the updating equation instead? You will get a **negative price** with a positive convergence factor. Mathematically it is easy to fix. But this way of writing by itself is unreasonable from an economic view. We should not run into this problem (and confusion to the results) if we have think through the ideas behind the equations of models.

Remarks

- Fixed point iteration / contraction mapping is easy to hand code from scratch.

- Accommodates root finding problems. Use convergence factor to “force out” a contraction mapping from a root finding problem.
- Can be extended to higher dimensions to solve a system of equations as we have seen.

Assignment

- Complete your contraction mapping algorithm to solve find the root of $x^{\frac{1}{3}} - x = 0$. Should be a cake walk if you are able to follow the other more complicated examples of contraction mapping.