Optimization: Gradient Methods

Code

Ivan Yi-Fan Chen

2023 Fall

Hide

```
library(tidyverse)
library(gridExtra)
```

1 Optimization

- Optimization and root finding are the same to some extent:
  - Root finding solves $f(x) = 0$.
  - Optimization solves $g'(x) = 0$ if the interior optimal point exists.
- Similar algorithms can thus be used for both optimization and root finding.
- BUT the **second-order derivative** matters for optimization
  - Solving the first-order condition is required, but we also need to know whether the solution is a minimum or a maximum.
  - Some revisions to root finding algorithms are needed.
- Optimization following the above approach is referred to as the **Gradient Methods**.
- Optimization can also be done using **Non-gradient Methods** where we don't need to do numerical calculus but still be able to get the optimal solutions.
- Optimization methods:

Type	Category	Methods
Gradient	Gradient only	Gradient Descent
Gradient	Gradient and Hessian	Newton-Raphson
Gradient	Gradient and approximated Hessian	Quasi-Newton
Non-gradient	Grid Search	Grid search
Non-gradient	Simplex Method	Nelder-Mead
Non-gradient	Monte Carlo and others	Genetic Algorithm, Simulated Annealing, Particle Swarm

1.1 Some talks about algorithms before moving on

- Machine Learning is essentially an optimization problem:
  1. Collect huge amount of data and construct a **model** of pattern.
  2. Develop a "scoring system" that matches the model results to empirical data.
  3. Minimize the difference between model results and the data using the score system. This is essentially an optimization.
- Illustrative with Optical Character Recognition (OCR):
  1. Want to use computer to digitalize loads of printed documents and reads for you.
  2. First, scan all the documents and obtain all alphabets in all fonts from the scanned files.

3. Construct a model to match the correct alphabet with the scanned alphabet. Have the computer tell us what alphabets it *sees* using the model.
4. Score the performance of the model. Score is low if the performance is bad.
5. Have the computer adjust the parameters of the model until a setting such that the score is highest is found. This is essentially an optimization problem.
6. Now the computer knows the alphabets, move to the next stage by teaching computer the meaning of words, context and sentiments in the documents, etc. All come in a similar fashion: construct models and let the computer pick the parameters that yield the highest scores.

- SES algorithms are also similar: construct a model behind to find the behaving patterns of the users (say preferences and ideologies), construct a scoring system and match between users to genereate the highest score.
- Learning optimization lets you know what algorithms are really doing out in the field.

2 Initial Search

- Optimization algorithms need initial guesses to start with.
- How to pick a good initial guess so that we get to the results faster and more correctly? There are some quick ways to do this.
- We cover both Grid Search and Simple Bracketing.

2.1 Grid Search

- Simple and brutal:
  1. Form a grid composed of equidistant points like what we have seen back in Ch. 4.
  2. Evaluate the function at each of the points, and pick the minimal one.
- Nearly impossible to give you the minimum as there are **infinitely many** points in the space.
- Good for picking a set of initial points for more delicate optimization algorithms.
- Demonstrate with a unimodal function. Note that it is easy to be extended to a higher-dimensional world.
  - Meet our old friend

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

    where we are interested on the interval $[-4, 3.5]$.
  - Perform a grid search by cutting the interval into 1000 equidistant points (including both endpoints).

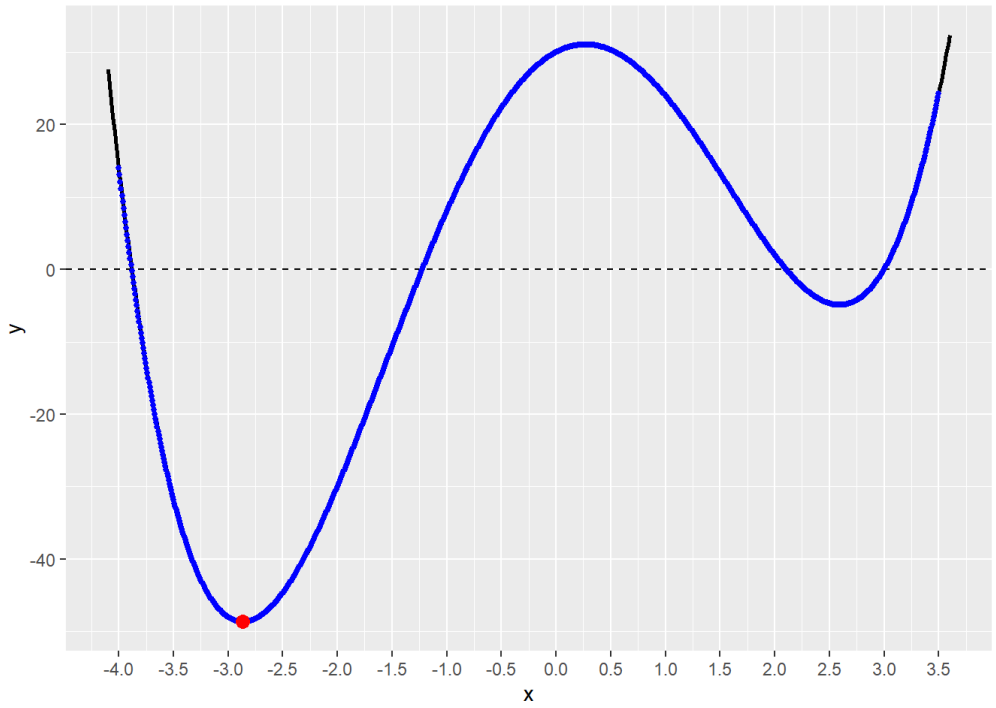
Hide

```
#Let's look at our old friend in the previous chapter...
pf<-function(x) {x^4-15*x^2+8*x+30}
grd<-seq(from=-4,to=3.5,length.out=1000)
grdval<-pf(grd)
pts<-data.frame(x=grd,y=grdval)
grdmin<-pts[which.min(pts$y),] #Pick the minimum out of the 1000 trials.
print(grdmin)
```

```
##             x         y
## 152 -2.866366 -48.66819
```

Hide

```
ggplot(data=data.frame(x=c(-4.1,3.6)),aes(x))+
  stat_function(fun=pf,n=500,size=1,color="black")+
  geom_hline(yintercept=0,color="black",linetype=2)+
  scale_x_continuous(breaks=seq(-4,3.5,0.5))+
  geom_point(data=pts,aes(x=x,y=y),size=1,color="blue")+
  geom_point(data=grdmin,aes(x=x,y=y),size=3,color="red")
```



- Looks right, but it is easily checked that the “minimum” is in fact not the true minimum.

Hide

```
grdmin$x
```

```
## [1] -2.866366
```

Hide

```
pf(grdmin$x)
```

```
## [1] -48.66819
```

Hide

```
pf(-2.866365)<pf(grdmin$x)
```

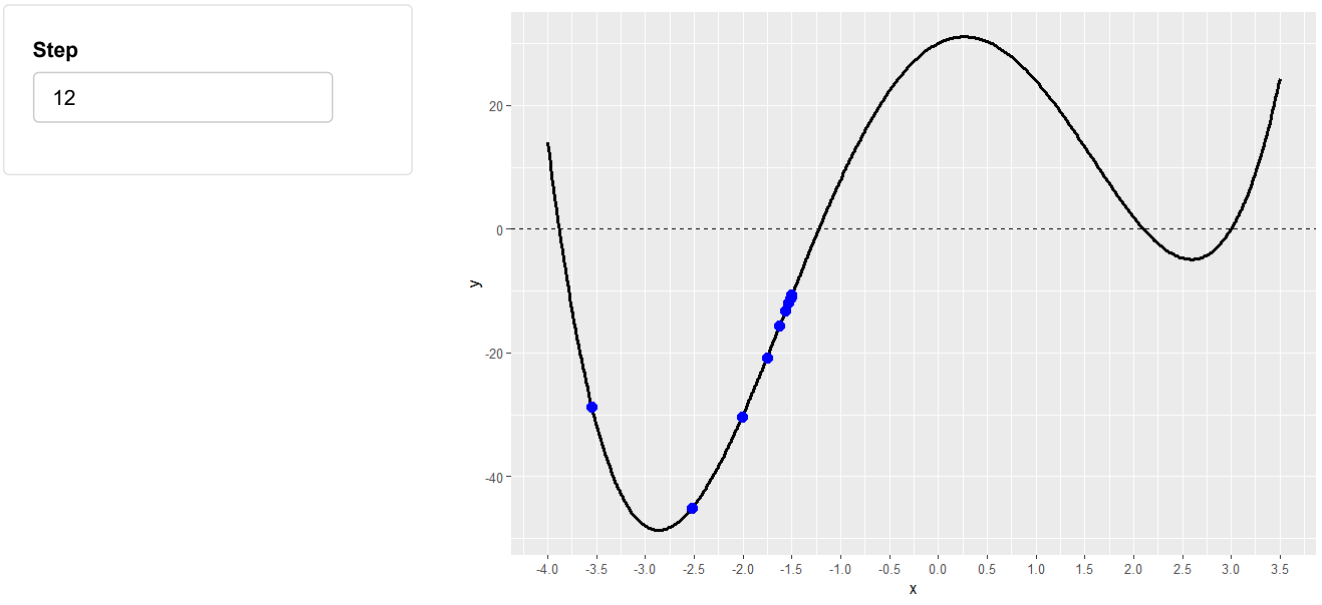
```
## [1] TRUE
```

2.1.1 Remarks

- In a one dimensional case, could be quite efficient if the interval is small enough.
- Becomes extremely costly as we have more dimensions.
  - If we have 8 variables, and for each variable we break into 10 equidistant nodes. Then we have 10^8 nodes in our grid to be computed. Can easily take two hours if the model is really complicated!
  - Think about what we have encountered back in Quasi-Monte Carlo integration!
- The result found are almost always **NOT** the maximum / minimum.
- Good for finding out candidates of initial points used in other optimization algorithms.

2.2 Bracketing Method

- The objective is to find the **interval** that contains the optimal point for an **unimodal** function.
  - **Unimodality:** a function that has a **unique** maximum / minimum.
  - Examples: $y = x^2$ on the whole real line, $y = x^4 - 15x^2 + 8x + 30$ on $x \in [1, 3.5]$, $y = x$ defined on $x \in [1, 2]$.
  - Functions such as $y = e^x$ and $y = x$ defined on the whole real line are NOT unimodal since they don't have a maximum / minimum.
  - Mind the **opennes** of interval! For example, $y = e^x$ is unimodal on $x \in [1, 2]$, but NOT on $(1, 2)$. In the later case, any for any x close to 2 there always exists an $2 - \epsilon$ such that $e^{2-\epsilon} > e^x$!
- If a function defined on $[a, b]$ is unimodal, then there exists a $c \in [a, b]$ such that the extrema occurs in either $[a, c]$ or $[c, b]$.
- Remember the bisection method? The spirit is similar: we can narrow down $[a, b]$ by changing a and b with an algorithm such that eventually the resulting subinterval contains the extrema.
- We focus on the **Simple Bracketing Method**
  - Pick an arbitrary point a, our goal is to find a c such that $[a, c]$ contains the minimum of the function $f(x)$.
  - Pick a and setup a step size s, say $s = 1e - 2$.
  - Compute $f(a)$ and $f(a + s)$. Check which one is larger so that we determine to which direction we should go so that we go **downhills**.
    - If $f(a) > f(a + s)$, then we search to the **right** of a afterwards.
    - If $f(a) < f(a + s)$, then we search to the **left** of a afterwards. In this case we reverse the sign of s.
  - Once direction is determined, we move toward the direction for each iteration i by $2^{i-1}s$ and evaluate at the points. That is, we evaluate $f(x)$ at $\{a + s,\ a + 2s,\ a + 4s, \dots\}$ and so on.
  - Stop the iteration upon seeing $f(x)$ goes up. Namely, we stop at i-th iteration upon seeing $f(x_i) > f(x_{i-1})$.



- Why increase the step size for each iteration?
  - The objective is to go downhills.
  - If we know that the hill is indeed going down, then we should **confidently** march forward until we flying over the valley, instead of walking slowly.
  - No need to worry about overshoot as the purpose is just **bracketing** instead of pinning down.
- For bracketing maximum, simply reverse the algorithm to climb uphills.

2.2.1 Implementation on R

- Algorithm (downhills):

  Step 0: Determine the interval to search, x_{lb} and x_{ub}.

  Step 1: Pick an arbitrary x_0 and choose a step size $s > 0$.

  Step 2: Evaluate $f(x_0)$ and $f(x_0 + s)$, and determine the sign of s

  a. If $f(x_0) > f(x_0 + s)$, let s be it.
  b. If $f(x_0) < f(x_0 + s)$, then replace s with $-s < 0$.

  Step 3: For each iteration i, evaluate $f(x)$ at $x_i \equiv x_0 + 2^{i-1}s$ and compare $f(x_i)$ with $f(x_{i-1})$. Stop the iteration upon the seeing $f(x_i) > f(x_{i-1})$. The minimum falls between $[x_{i-1}, x_i]$.

- The following code include an additional feature: what if x_i goes **beyond** the boundary of interval?

- A home-made simple bracketing function `bkt(init,lb,ub,s,r,imax)`

  - `init` is the initial point (x_0).
  - `lb` and `ub` are lower and upper bounds of the function of interest (a and b in the math part).
  - `s` is step size, and `imax` is the maximum number of iteration.
  - Explain `r<1` later.

Hide

```
#Simple bracketing method
bkt<-function(init,lb,ub,s=1e-3,r=1e-2,imax=1000) {
  #Check if the initial point is beyond the search interval. If yes, force it to be on the endpoint.
  if(init>ub) {init<-ub}
  if(init<lb) {init<-lb}

  #Determine the direction to search. We check to the right of the initial point
  x<-init+s
  if(x>ub) {x<-ub}
  val0<-pf(init)
  val1<-pf(x) #Check to the right of initial point.
  if (val0<=val1) {s<--s} #Search to the left if to the right seems to be an uphill.

  #Iterate until we can't descent any further.
  i<-1
  history<-data.frame(i=c(0,1:imax),x=c(init,rep(NA,imax)),val=c(pf(init),rep(NA,imax)))
  while (i<=imax) {
    x1<-init+(2^(i-1))*s #Step size is increasing as we iterate.
    #Deal with boundary issue: take the boundary if hit, and reverse searching direction and reduce s
tep size by a factor of r.
    if(x1>ub) {
      x1<-ub
      s<--1*s*r}
    if(x1<lb) {
      x1<-lb
      s<--1*s*r}
    val1<-pf(x1)
    val0<-pf(init)

    if(val1<val0) { #If f(x1)<f(x0) then minimum must be falling beyond x1. So use x1 as new init.
      history$x[i+1]<-x1
      history$val[i+1]<-val1
      init<-x1
      i<-i+1
    } else { #If f(x1)>=f(x0) then minimum must be between x0 and x1. Stop and report
      history$x[i+1]<-x1
      history$val[i+1]<-val1
      rs.lb<-min(c(init,x1))
      rs.ub<-max(c(init,x1))
      cat("Minimum occurs between",rs.lb,"and",rs.ub,".\n")
      return(list(bound=c("lb"=rs.lb,"ub"=rs.ub),history=na.omit(history)))
      break
    }

  }

}

rs<-bkt(init=-1.5,lb=-4,ub=-1)
```

```
## Minimum occurs between -3.547 and -2.523 .
```

Hide

```
print(rs$history)
```

```
##     i      x       val
## 1   0 -1.500 -10.68750
## 2   1 -1.501 -10.72700
## 3   2 -1.503 -10.80601
## 4   3 -1.507 -10.96407
## 5   4 -1.515 -11.28032
## 6   5 -1.531 -11.91326
## 7   6 -1.563 -13.18044
## 8   7 -1.627 -15.71564
## 9   8 -1.755 -20.75382
## 10  9 -2.011 -30.39490
## 11 10 -2.523 -45.14698
## 12 11 -3.547 -28.80732
```

- Initial point matters when there are multiple local extrema.

Hide

```
bkt(init=0,lb=-4,ub=3)
```

```
## Minimum occurs between -4 and -2.047 .
```

```
## $bound
##     lb     ub
## -4.000 -2.047
##
## $history
##     i      x        val
## 1   0  0.000  30.000000
## 2   1 -0.001  29.991985
## 3   2 -0.003  29.975865
## 4   3 -0.007  29.943265
## 5   4 -0.015  29.876625
## 6   5 -0.031  29.737586
## 7   6 -0.063  29.436481
## 8   7 -0.127  28.742325
## 9   8 -0.255  26.988853
## 10  9 -0.511  22.063369
## 11 10 -1.023   7.213288
## 12 11 -2.047 -31.671284
## 13 12 -4.000  14.000000
```

- What if we hit the boundary? This is where `r` kicks in.
- If an iteration x_i goes beyond the boundary, do the following:
    1. Force x_i to be the boundary point a or b depending on which side you bump into.
    2. Reverse the searching direction by replacing `s` with `-s`.
    3. Finally, scale down the step size by multiply with `r<1`.
- If the boundary is indeed a minimum, then the algorithm stops at $i+1$ since $f(x_{i+1}) > f(x_i)$ by construction. If not then the search continues.
- Why using `r` to scale down the step size?
    - The effective step size $2^{i-1}s$ gets large if we iterate many times, then most likely we will be flying to the other end of the interval upon reversing the direction!
    - Multiplying with `r` is like we admit that we have gone too far and restart the search carefully.

Hide

```
rs<-bkt(init=1,lb=-4,ub=3)
```

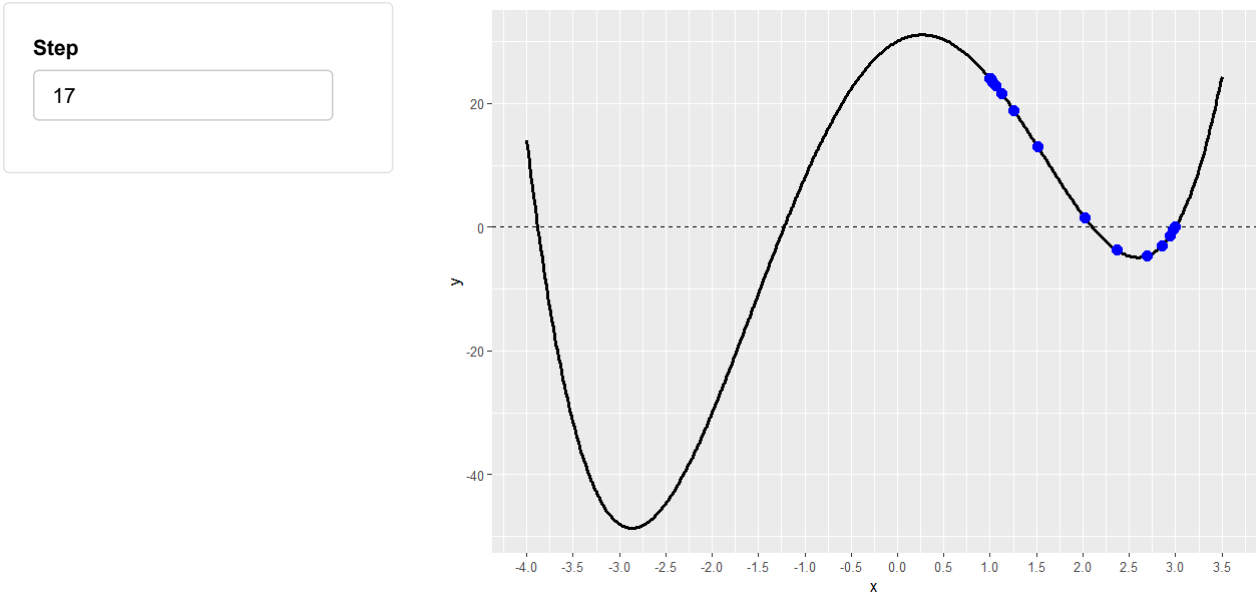
```
## Minimum occurs between 2.36512 and 2.6928 .
```

Hide

```
print(rs$history)
```

```
##     i       x        val
## 1   0 1.00000 24.0000000
## 2   1 1.00100 23.9819910
## 3   2 1.00300 23.9459191
## 4   3 1.00700 23.8735604
## 5   4 1.01500 23.7279886
## 6   5 1.03100 23.4334711
## 7   6 1.06300 22.8312949
## 8   7 1.12700 21.5772927
## 9   8 1.25500 18.8953288
## 10  9 1.51100 13.0538265
## 11 10 2.02300  1.5448586
## 12 11 3.00000  0.0000000
## 13 12 2.97952 -0.5162251
## 14 13 2.93856 -1.4529888
## 15 14 2.85664 -2.9607623
## 16 15 2.69280 -4.6456845
## 17 16 2.36512 -3.6954134
```

- Check it visually.



2.2.2 Remark

- Doesn't look easy to be implemented into a higher dimensional case.
- There are more ways for bracketing, such as Fibonacci search, golden section search, quadratic fit search, and Shubert-Piyavskii method.

3 Gradient Based Methods

- Uses gradients (first-derivatives) to find the minimum / maximum of a function.
- Based on the conditions for optimal solutions learned in undergraduate calculus.

- We will cover both Newton-Raphson and Gradient Descent. Also briefly introduce Quasi-Newton algorithms.

3.1 Newton-Raphson Method

- Consider a minimization problem

$$\min_x g(x)$$

  An interior solution x^* is such that the first- and second-order conditions hold:

$$g'(x^*) = 0$$

$$g''(x^*) > 0$$

  - For maximization, things are all the same except that we require $g''(x^*) < 0$ instead.
- The approach is essentially the same as what we have learned about Newton-Raphson algorithm in root finding.
- In root finding, we use **linear approximation / first-order Taylor expansion** to approximate the function to our interest.
- Here we use **second-order Taylor expansion** to approximate the function to our interest at an initial point $x0$:

$$g(x) \approx g(x_0) + g'(x_0)(x - x_0) + g''(x_0)\frac{(x - x_0)^2}{2} \equiv h(x)$$

- Now that we approximate $g(x)$ with $h(x)$. Then the minimum of $h(x)$ should be the same as that of $g(x)$ if x_0 is close enough. Then $h'(x) = 0$ needs to hold:

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

- The above gives us the x that **minimizes** $h(x)$.

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

- If our initial guess x_0 is close to the minimal of $g()$, x, then we have $x_0 \approx x$ such that $g'(x_0) = 0$ and $g''(x_0) > 0$ hold.
- What if we guessed it wrong? Then we update our guess following the following updating equation:

$$x_{i+1} = x_i - \frac{g'(x_i)}{g''(x_i)}$$

- If $g(x)$ is well-behaved in the sense that $g''(x) > 0$ throughout its domain, then:
  1. If $g'(x_i) > 0$, we should reduce x_i to go **downhills** if we are searching for a minimal. The above updating equation is consistent since in this case we have $x_i > x_{i+1}$.
  2. If $g'(x_i) < 0$, then we should increase x further so that $g(x)$ gets smaller. The updating equation is consistent since in this case $x_i < x_{i+1}$.
- As we have learned in root finding, we can use the notion of contraction mapping to improve the algorithm by introducing a convergence factor s:

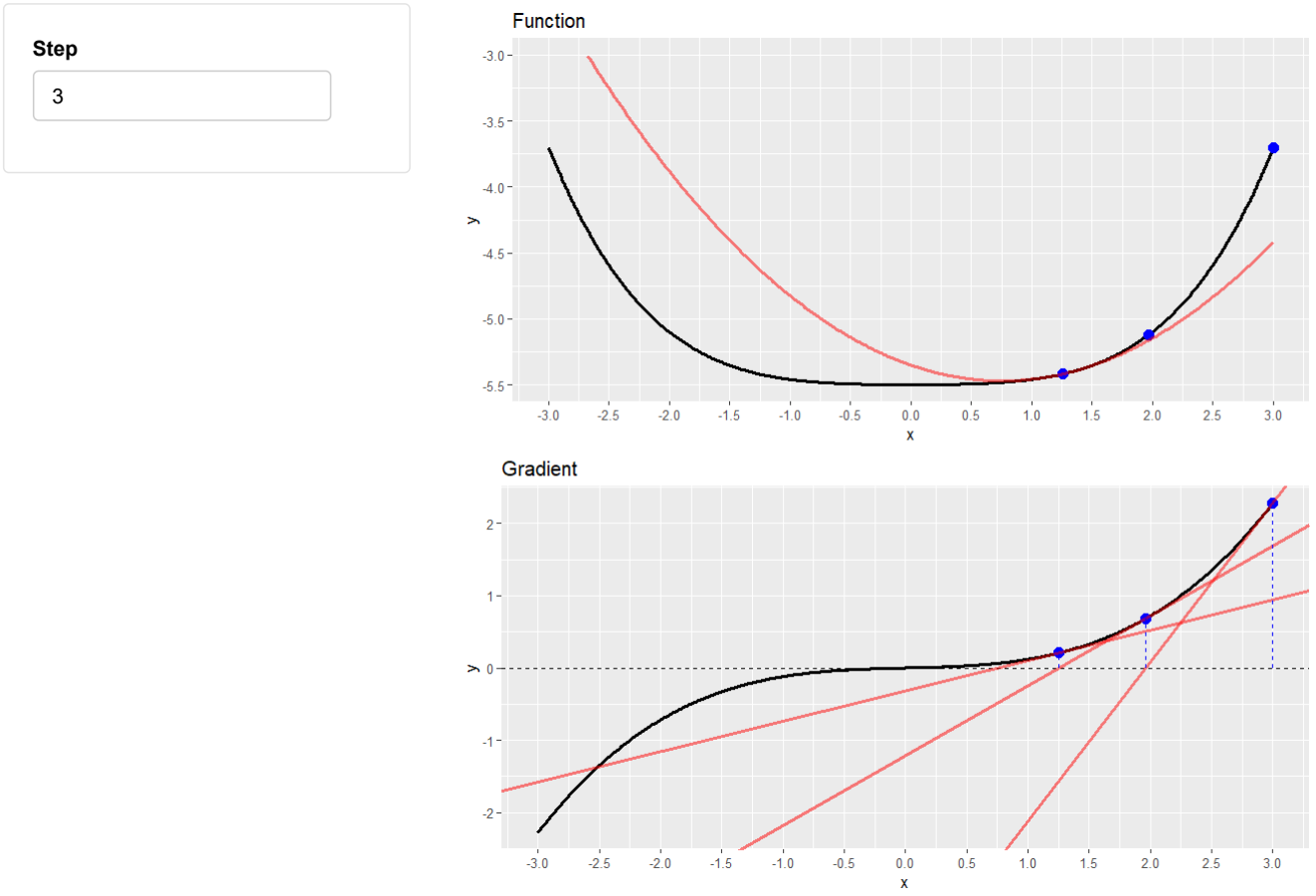
$$x_{i+1} = x_i - s\frac{g'(x_i)}{g''(x_i)}$$

- **Quick Summary:**
  1. A function with minimal $g(x)$ must be U-shaped in the neighborhood. We approximate $g(x)$ in this neighborhood with respect to x_0 using its second-order Taylor expansion $h(x)$. Note that $h(x)$ is a U-shaped quadratic function.

2. We use the minimal of $h(x)$ to approximate the minimal of $g(x)$. If we do not get it right, then we do the approximation again to get a new minimal of $h(x)$ until we get what we want.
3. The updating equation features climbing **downhills** if $g(x)$ is strictly convex as required by minimization problems.
4. Can include a convergence factor to the updating equation to utilize contraction mapping.
5. If $g''(x) < 0$, then the algorithm climbs uphill and gives us a maximum.

- Illustration:

```
## Minimum found upon 9 -th iteration at x= -8.268553e-09 .
```



3.1.1 More Intuitive Interpretation

- The first-order condition is essentially a **root finding** problem, $f(x) = 0$. It is attempting that we use the Newton-Raphson algorithm as we have taught as

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

  - Review the previous chapter if you have difficulty comprehend this approach.
  - See the lower-half of the interactive widget for an illustration.
- Now note that $f(x) \equiv g'(x)$, the Newton-Raphson updating equation is exactly

$$x_{i+1} = x_i - \frac{g'(x_i)}{g''(x_i)}$$

- Is that all? NO! **Root finding DO NOT concern about second-order derivative!!** Just by solving this root finding problem gives you either a maximum **OR** minimum! We require $g''(x) > 0$ for the "root" to entail a minimum, and vice versa.

3.1.2 Implementation on R

- Algorithm:

Step 0: Choose tolerance level ϵ, step size for numerical derivatives h and maximum amount of iteration. Also a convergence factor s if needed.

Step 1: Pick an initial guess x_0, compute the first- and second-derivative at x_0, and form the updating equation

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

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

a. If yes, then end the iteration and x_1 is what we need.

b. If no, use x_1 as the new initial guess and repeat Steps 1 and 2 until $|x_i - x_{i-1}| < \epsilon$. Then x_i is what we want.

- One can compute the derivative in the process with several approaches:
  - Numerical: finite difference, automatic differentiation.

    Finite difference can be very inaccurate; automatic differentiation can be complicated to implement since we also need the second-order derivatives.
  - Mathematical: hand solve the derivatives of the function and supply into your algorithm. Obviously only works when the function is not too complicated to solve.
- Demonstrate with

$$y = \frac{x^4 + x^2}{50} - \frac{11}{2}$$

Hide

```
#Newton-Raphson
fn<-function(x){((x^4+x^2)/50)-(11/2)}

Newton<-function(fun,x0,s=1,tol=1e-6,h=1e-6,imax=1000) {
  history<-data.frame(i=rep(NA,imax),x0=rep(NA,imax),val0=rep(NA,imax),
                      x1=rep(NA,imax),val1=rep(NA,imax),diff=rep(NA,imax))
  i<-1

  while (i<=imax) {
    #Derivatives using central rule
    fx0<-(fun(x0+(h/2))-fun(x0-(h/2)))/h
    fxx0<-(fun(x0+h)-2*fun(x0)+fun(x0-h))/(h^2)

      #Updating rule and check differences
    x1<-x0-s*(fx0/fxx0)
    diff<-abs(x0-x1)
    history[i,]<-c(i,x0,fun(x0),x1,fun(x1),diff)

      if (diff<tol) {
        cat("Minimum found upon",i,"-th iteration at x=",x1,".\n")
        return(list(x=x1,val=history$val1[i],history=na.omit(history)))
        break
      }

      if (i==imax) {
        cat("Iteration ended before convergence. Returning results from the last iteration. \n")
        return(list(x=x1,val=history$val1[i],history=na.omit(history)))
        break
      }

      x0<-x1
      i<-i+1

  }
}

Newton(fun=fn,x0=3)
```

```
## Minimum found upon 9 -th iteration at x= -8.268553e-09 .
```

```
## $x
## [1] -8.268553e-09
##
## $val
## [1] -5.5
##
## $history
##   i            x0      val0            x1      val1         diff
## 1 1  3.000000e+00 -3.700000  1.964063e+00 -5.125236 1.035937e+00
## 2 2  1.964063e+00 -5.125236  1.255533e+00 -5.418774 7.085297e-01
## 3 3  1.255533e+00 -5.418774  7.569927e-01 -5.481972 4.985407e-01
## 4 4  7.569927e-01 -5.481972  3.893352e-01 -5.496509 3.676575e-01
## 5 5  3.893352e-01 -5.496509  1.266938e-01 -5.499674 2.626414e-01
## 6 6  1.266938e-01 -5.499674  6.511099e-03 -5.499999 1.201827e-01
## 7 7  6.511099e-03 -5.499999  1.359251e-04 -5.500000 6.375174e-03
## 8 8  1.359251e-04 -5.500000 -9.715744e-08 -5.500000 1.360222e-04
## 9 9 -9.715744e-08 -5.500000 -8.268553e-09 -5.500000 8.888889e-08
```

3.1.3 Issues with Newton-Raphson

- The algorithm runs into problems more often then not. Demonstrate with our old friend $y = x^4 - 15x^2 + 8x + 30$.
    - **Issue 1:** Use $x = -0.5$ as the initial point and get an extrema at $x \approx 0.27$. But it is the **maximum** of the function, instead of the **minimum** that we are for!

Hide

```
#Getting a maximum instead
Newton(fun=pf,x0=-0.5)
```

```
## Minimum found upon 4 -th iteration at x= 0.2692698 .
```

```
## $x
## [1] 0.2692698
##
## $val
## [1] 31.07182
##
## $history
##   i         x0     val0        x1     val1         diff
## 1 1 -0.5000000 22.31250 0.3333141 31.01238 8.333141e-01
## 2 2  0.3333141 31.01238 0.2687343 31.07182 6.457982e-02
## 3 3  0.2687343 31.07182 0.2692698 31.07182 5.355659e-04
## 4 4  0.2692698 31.07182 0.2692698 31.07182 1.085498e-08
```

- **Issue 2:** Algorithm cannot proceed if we happen to run into a point with 0 second-order derivative, i.e., inflection point. Let's pick $x = -\sqrt{10}/2$ as the initial point.

Hide

```
#Cannot proceed if we run into inflection points.
Newton(fun=pf,x0=-sqrt(10)/2)
```

```
## Error in if (diff < tol) {: 需要 TRUE/FALSE 值的地方有缺值
```

- **Issue 3:** Get an local minimum instead of a global one. Try out $x = 3$ and $x = -4$ as initial points.

Hide

```
#Might found just a local optimal.
Newton(fun=pf,x0=3) #This gives you the local minimum
```

```
## Minimum found upon 5 -th iteration at x= 2.594031 .
```

```
## $x
## [1] 2.594031
##
## $val
## [1] -4.903306
##
## $history
##   i       x0      val0       x1      val1         diff
## 1 1 3.000000  0.000000 2.666742 -4.765142 3.332582e-01
## 2 2 2.666742 -4.765142 2.597033 -4.903077 6.970910e-02
## 3 3 2.597033 -4.903077 2.594037 -4.903306 2.995933e-03
## 4 4 2.594037 -4.903306 2.594031 -4.903306 5.317654e-06
## 5 5 2.594031 -4.903306 2.594031 -4.903306 4.200502e-10
```

Hide

```
Newton(fun=pf,x0=-4) #This gives you the global minimum.
```

```
## Minimum found upon 5 -th iteration at x= -2.863301 .
```

```
## $x
## [1] -2.863301
##
## $val
## [1] -48.66852
##
## $history
##   i        x0      val0        x1      val1         diff
## 1 1 -4.000000  14.00000 -3.210033 -44.06609 7.899666e-01
## 2 2 -3.210033 -44.06609 -2.911099 -48.58915 2.989340e-01
## 3 3 -2.911099 -48.58915 -2.864408 -48.66847 4.669114e-02
## 4 4 -2.864408 -48.66847 -2.863302 -48.66852 1.106188e-03
## 5 5 -2.863302 -48.66852 -2.863301 -48.66852 7.746157e-07
```

- Issues 1 and 2 can be easily fixed. I define a revised version of the algorithm `NewtonFix` and the following shows that it is free from issues 1 and 2. Your assignment asks you to figure out how to fix these issues.

Hide

```
#Fixing 0-Hessian and getting maximum.

## Idea is very simple: when finding the minimum of f(x) we require fxx(x)>0. So we just check it bef
ore performing any further steps. The criteria is as follows:
## 1. For any x0 passed in, compute fx(x0) and fxx(x0).
## 2. If sign(fxx(x0)) > 0, then we are fine.
## 3. If sign(fxx(x0)) < 0, then the updating equation now finds maximum for you. That is,
##    x1 = x0 - s(fx(x0)/fxx(x0)), and a negative fxx(x0) makes the updating equation to become
##    x1 = x0 + s(fx(x0)/|fxx(x0|).
##    In this case fx>0 leads to x1>x0 so you are climbing up, and vice versa!
##    To deal with this, just assign s = -s so that you reverse the searching direction!
## 4. If sign(fxx(x0)) = 0, then we need to use sign of fx(x0) to determine whether we "shift" x0
##    to the left or to the right by a small amount.
###    4a. If sign(fx(x0)) > 0, then we want to assign x0.new = x0 - epsilon because we are at the up
hill part while we want to climb DOWN.
###    4b. If sign(fx(x0)) < 0, then we do x0.new = x0 + epsilon for the opposite reason as above.
###    4c. If sign(fx(x0)) = 0 then we are on the saddle point. We could just compute a SECANT line a
nd use its slope for judment if we really want to be serious. But here I just do a random step by hav
ing x0.new = sample(x0+epsilon,x0-epsilon) w.p. 0.5.


NewtonFix<-function(fun,x0,s=1,tol=1e-6,h=1e-6,imax=1000) {

  #Define functions for 1st and 2nd derivative since we might need to call them repeatedely.
  dev1<-function(x){(fun(x+(h/2))-fun(x-(h/2)))/h}
  dev2<-function(x){(fun(x+h)-2*fun(x)+fun(x-h))/(h^2)}

  history<-data.frame(i=rep(NA,imax),x0=rep(NA,imax),val0=rep(NA,imax),
                      x1=rep(NA,imax),val1=rep(NA,imax),diff=rep(NA,imax))
  x1<-NA
  i<-1

  while (i<=imax) {
    #Derivatives using central rule
    fx0<-dev1(x0)
    fxx0<-dev2(x0)

    if(sign(fxx0)==0&&sign(fx0)>0) {

      x0<-x0-1e-1
      i<-i+1

      } else if(sign(fxx0)==0&&sign(fx0)<0) {

      x0<-x0+1e-1
      i<-i+1

      } else if(sign(fxx0)==0&&sign(fx0)==0) {

      x0<-sample(c(x0+1e-1,x0-1e-1),1)
      i<-i+1

      } else {

        if(sign(fxx0)<0) {s<--s}
          #Updating rule and check differences
          x1<-x0-s*(fx0/fxx0)
```

```
          diff<-abs(x0-x1)
          history[i,]<-c(i,x0,fun(x0),x1,fun(x1),diff)

        if (diff<tol) {
          cat("Minimum found upon",i,"-th iteration at x=",x1,".\n")
          return(list(x=x1,val=history$val1[i],history=na.omit(history)))
          break
        }

          x0<-x1
          i<-i+1
      }

    if (i==imax) {
      cat("Iteration ended before convergence. Returning results from the last iteration. \n")
      history[i,]<-c(i,x0,fun(x0),x1,fun(x1),diff)
      return(list(x=x1,val=history$val1[i],history=na.omit(history)))
      break
      }

  }
}
```

Hide

```
NewtonFix(fun=pf,x0=-0.5) #The same setting now gives us a minimum, at least a local one near 2.6.
```

```
## Minimum found upon 7 -th iteration at x= 2.594031 .
```

```
## $x
## [1] 2.594031
##
## $val
## [1] -4.903306
##
## $history
##   i        x0      val0        x1      val1         diff
## 1 1 -0.500000 22.312500 -1.333314 -4.172098 8.333141e-01
## 2 2 -1.333314 -4.172098  3.110109  3.351834 4.443423e+00
## 3 3  3.110109  3.351834  2.703203 -4.587247 4.069063e-01
## 4 4  2.703203 -4.587247  2.600671 -4.902185 1.025320e-01
## 5 5  2.600671 -4.902185  2.594058 -4.903306 6.612683e-03
## 6 6  2.594058 -4.903306  2.594031 -4.903306 2.650644e-05
## 7 7  2.594031 -4.903306  2.594031 -4.903306 9.803922e-10
```

Hide

```
NewtonFix(fun=pf,x0=-sqrt(10)/2) #Converged to the global minimum near -2.86.
```

```
## Minimum found upon 11 -th iteration at x= -2.863301 .
```

```
## $x
## [1] -2.863301
##
## $val
## [1] -48.66852
##
## $history
##     i         x0        val0         x1        val1         diff
## 2   2  -1.681139   -17.85496 -11.770492 17052.24015 1.008935e+01
## 3   3 -11.770492 17052.24015  -7.989797  3083.67236 3.780695e+00
## 4   4  -7.989797  3083.67236  -5.553638   474.21154 2.436159e+00
## 5   5  -5.553638   474.21154  -4.051426    20.79894 1.502212e+00
## 6   6  -4.051426    20.79894  -3.233921   -43.37015 8.175050e-01
## 7   7  -3.233921   -43.37015  -2.916885   -48.56857 3.170357e-01
## 8   8  -2.916885   -48.56857  -2.864692   -48.66845 5.219341e-02
## 9   9  -2.864692   -48.66845  -2.863303   -48.66852 1.389290e-03
## 10 10  -2.863303   -48.66852  -2.863301   -48.66852 1.352587e-06
## 11 11  -2.863301   -48.66852  -2.863301   -48.66852 4.152825e-10
```

3.1.4 In-Class Exercise

- How to fix Issues 1 and 2? Figure it out and implement.

  Hint:

  For Issue 1, you can adjust the direction of search by changing the sign of the convergence factor s. But then you need to figure out whether you should step forwards or backwards so that your new guess is more likely to get closer to a minimum. To figure it out, recall the first- and second-order conditions for finding minimum / maximum.

  For Issue 2, you need to make a local shift at your initial point. But shift to which direction? The reasoning behind is similar to how you deal with Issue 1.

3.1.5 Newton-Raphson in Multivariate Optimization

- Start with a function of 2 variables $f(x, y)$.
- From undergraduate calculus, the interior solution to the minimization problem $\min_{x,y} f(x, y)$ requires the following conditions to hold:

$$\frac{\partial f(x, y)}{\partial x} = 0$$

$$\frac{\partial f(x, y)}{\partial y} = 0$$

$$\frac{\partial^2 f(x, y, z)}{\partial x^2} > 0$$

$$\frac{\partial^2 f(x, y, z)}{\partial x^2} \frac{\partial^2 f(x, y, z)}{\partial y^2} - \left(\frac{\partial^2 f(x, y, z)}{\partial x \partial y} \right)^2 > 0$$

- Can be restated in terms of matrix algebra: The first order conditions needs to hold

$$\begin{bmatrix} \frac{\partial g(x,y,z)}{\partial x} \\ \frac{\partial g(x,y,z)}{\partial y} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

  and that the Hessian matrix

$$H = \begin{bmatrix} \frac{\partial^2 g(x,y,z)}{\partial x^2} & \frac{\partial^2 g(x,y,z)}{\partial y \partial x} \\ \frac{\partial^2 g(x,y,z)}{\partial x \partial y} & \frac{\partial^2 g(x,y,z)}{\partial y^2} \end{bmatrix}$$

  needs to be **positive definite**.

- More generally:
  - Let $\mathbf{x} \equiv \begin{bmatrix} x_1 & \dots & x_N \end{bmatrix}$ be the vector of N variables.
  - Let $\nabla g(\mathbf{x}) \equiv \begin{bmatrix} \frac{\partial g(\mathbf{x})}{\partial x_1} & \dots & \frac{\partial g(\mathbf{x})}{\partial x_N} \end{bmatrix}$ be the **gradient vector**.
  - Let

$$H(\mathbf{x}) \equiv \begin{bmatrix} \frac{\partial^2 g(\mathbf{x})}{\partial x_1^2} & \frac{\partial^2 g(\mathbf{x})}{\partial x_2 \partial x_1} & \dots & \frac{\partial^2 g(\mathbf{x})}{\partial x_N \partial x_1} \\ \frac{\partial^2 g(\mathbf{x})}{\partial x_2 \partial x_1} & \frac{\partial^2 g(\mathbf{x})}{\partial x_2^2} & \dots & \frac{\partial^2 g(\mathbf{x})}{\partial x_N \partial x_2} \\ \dots & \dots & \dots & \dots \\ \frac{\partial^2 g(\mathbf{x})}{\partial x_N \partial x_1} & \frac{\partial^2 g(\mathbf{x})}{\partial x_N \partial x_2} & \dots & \frac{\partial^2 g(\mathbf{x})}{\partial x_N^2} \end{bmatrix}$$

    be the **Hessian matrix**.
  - We require the gradient vector to be equal to a zero vector, and the Hessian matrix to be **positive definite** for a minimization problem.
  - For a maximization problem, the conditions are the same except that the Hessian matrix now needs to be **negative definite**.

- Nevertheless the updating equation is surprisingly similar to the one-variable case:

$$\mathbf{x}_{i+1} = \mathbf{x}_i - s[H(\mathbf{x_i})]^{-1} \nabla g(\mathbf{x}_i)$$

  where $[H(\mathbf{x_i})]^{-1}$ is the **inverse Hessian matrix**.

- Totally okay if you don't know how to test the definiteness of matrix and solve for inverse matrix.
  - Inverse matrix is obtained by `solve()` command in R. Specifically, `solve(A)` gives us the inverse matrix of matrix `A` if it is **non-singular**. You get an error message if it is singular.
  - One can think of inverse matrix as putting a number to the denominator. Just like that a number needs to be non-zero to be qualified as a denominator, the matrix needs to be non-singular to be invertible.
  - The Hessian matrix is **symmetric** if $g(x)$ is **continuously differentiable**. Then we can use `isposdef` in the `pracma` package to test if the Hessian matrix is **strictly positive definite**.
  - Testing positive definiteness is sufficient even for a maximization problem. This is because a strictly convex function entails a positive definite Hessian matrix, and that the function becomes strictly concave if we multiply by -1. A maximization and a minimization problem is thus interchangeable for the same function.
- Minimize $u = x^2 + y^2$. Easily checked that its minimal occurs at $x = 0$ and $y = 0$.

Hide

```
#Multivariate Minimization
ufn<-function(x,y){x^2+y^2}

grad<-function(x,y,h=1e-6) {
  xf<-x+(h/2)
  xb<-x-(h/2)
  yf<-y+(h/2)
  yb<-y-(h/2)

  fx<-(ufn(x=xf,y=y)-ufn(x=xb,y=y))/h
  fy<-(ufn(x=x,y=yf)-ufn(x=x,y=yb))/h

  return(matrix(c(fx,fy),ncol=1,nrow=2))
}

Hess<-function(x,y,h=1e-6){

  fxx<-(ufn(x=x+h,y=y)-2*ufn(x=x,y=y)+ufn(x=x-h,y=y))/(h^2)
  fyy<-(ufn(x=x,y=y+h)-2*ufn(x=x,y=y)+ufn(x=x,y=y-h))/(h^2)
  fxy<-((ufn(x=x+(h/2),y=y+(h/2))-ufn(x=x-(h/2),y=y+(h/2)))/h-
        (ufn(x=x+(h/2),y=y-(h/2))-ufn(x=x-(h/2),y=y-(h/2)))/h)/h

  return(matrix(c(fxx,fxy,fxy,fyy),ncol=2,nrow=2))
}

imax<-1000
tol<-1e-6
s<-1
x0<-5
y0<-7
h<-1e-6
i<-1
history<-data.frame(i=c(0:imax),x=c(x0,rep(NA,imax)),y=c(y0,rep(NA,imax)),diff=c(Inf,rep(NA,imax)))

while(i<=imax) {

  fxv<-grad(x0,y0,h)
  fxxv<-Hess(x0,y0,h)
  invfxxv<-solve(fxxv)
  x0v<-c(x0,y0)
  x1v<-x0v-s*invfxxv%*%fxv

  diff<-max(abs(x1v-x0v))

  history[i+1,]<-c(i,x1v[1],x1v[2],diff)

  if(diff<tol) {
    cat("Minimum found upon",i,"-th iteration.\n")
    rst<-list(x=c(x=x1v[1],y=x1v[2]),val=ufn(x=x1v[1],y=x1v[1]),history=na.omit(history))
    return(rst)
  }

  if(i==imax) {
    cat("Iteration ended before convergence. Returning the latest result. \n")
    rst<-list(x=c(x=x1v[1],y=x1v[2]),val=ufn(x=x1v[1],y=x1v[1]),history=na.omit(history))
    return(rst)
  }
```

```
  x0<-x1v[1]
  y0<-x1v[2]
  i<-i+1

}
```

```
## Minimum found upon 3 -th iteration.
```

```
## $x
##             x             y
## -8.478601e-24 -7.031035e-24
##
## $val
## [1] 1.437734e-46
##
## $history
##   i             x             y         diff
## 1 0  5.000000e+00  7.000000e+00          Inf
## 2 1 -2.633886e-02  1.303250e-02 6.986968e+00
## 3 2 -6.160803e-10 -1.479928e-09 2.633886e-02
## 4 3 -8.478601e-24 -7.031035e-24 1.479928e-09
```

Hide

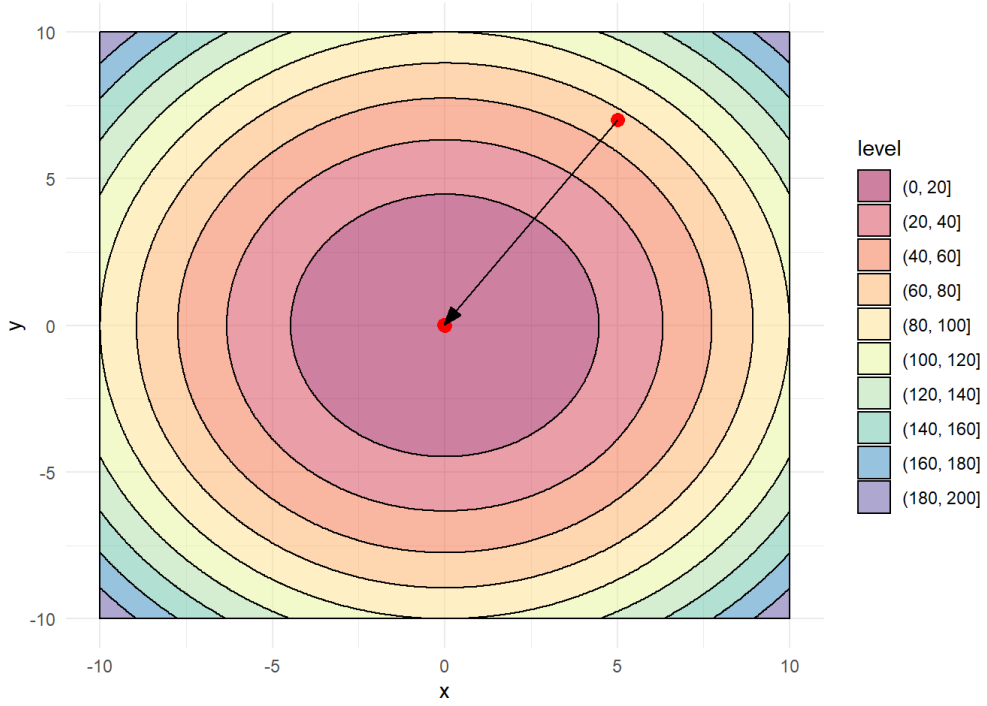
```
rst
```

```
## $x
##             x             y
## -8.478601e-24 -7.031035e-24
##
## $val
## [1] 1.437734e-46
##
## $history
##   i             x             y         diff
## 1 0  5.000000e+00  7.000000e+00          Inf
## 2 1 -2.633886e-02  1.303250e-02 6.986968e+00
## 3 2 -6.160803e-10 -1.479928e-09 2.633886e-02
## 4 3 -8.478601e-24 -7.031035e-24 1.479928e-09
```

Hide

```
#Make a contour plot to show the searching procedure
gx<-seq(-10,10,length.out=500)
unit<-rep(1,500)
grid<-cbind(x=gx%x%unit,y=rep(gx,500))%>%data.frame()
grid<-cbind(grid,f=ufn(x=grid$x,y=grid$y))
df<-rst$history

ggplot(grid,aes(x=x, y=y))+
  geom_contour_filled(aes(z=f), color="black", alpha=0.5)+
  scale_fill_brewer(palette = "Spectral")+
  theme_minimal() +
  geom_point(data=df,aes(x=x,y=y),color="red",size=3)+
  geom_segment(data=df,aes(x=x[1],y=y[1],xend=x[2],yend=y[2]),
               arrow=arrow(type="closed",angle=20,length=unit(x=0.03,units="npc")))
```



3.1.6 Remarks

- Essentially the same technique that we have used for root finding.
- But unlike in root finding, now we concern about whether the second derivative is positive or negative because not all of the roots of the first-order condition are what we want.
- Can be extended to a multivariate world. That is, you can solve things like utility maximization across different products using this technique.
- Can include a convergence factor to improve computation efficiency.
- However there are problems:
    1. NOT guaranteed to converge to a solution. For example, can overshoot and bouncing.
    2. Gives you a **local** instead of global extrema more often then not.
    3. Based on differentiability of the function. Needs to be at least **twice differentiable**.
    4. Hessian matrix is **very expensive** to compute!
        - In a 2 variable case we need to compute 3 second-order derivatives (thanks to continuous differentiability.)
        - In a 3 variable case, there are 6 in total.
        - N variables, then $N(N+1)/2$ in total!
        - Finite difference is not very accurate, hence the error can keep accumulating and eventually causing your Hessian matrix to be seemingly singular.

3.2 Gradient Descent

- Newton-Raphson is expensive because of the second-derivative / Hessian matrix.
- How about we just use the gradient vector / first derivative since is sufficient to tell us if we are going up- or downhills?
- Exactly what **Gradient Descent Algorithm** does.
- **Gradient Descent Algorithm:**
    - Implemented **exactly the same** as Newton-Raphson, except that the updating equation becomes

$$x_{i+1} = x_i - sg'(x_i)$$

- s is the convergence factor. But here it is more frequently referred to as the **learning rate** because Gradient Descent is widely applied in **Machine Learning**.
- Uses only the gradient, i.e., it concerns about up- and downhill. The curvature of $g(x)$ does not matter.
- If $g'(x_i) > 0$, then we want to move to the left so that we go downhill. Hence $x_i > x_{i+1}$. Conversely we want to move to the right to go downhill, hence $x_i < x_{i+1}$.
- The learning rate s determines how aggressive the iteration is. Higher s makes the algorithm to be more aggressive as it enlarges the step size $g'(x)$.

- In a more general case, we may want to include an invertible matrix P so that the updating equation becomes

$$\mathbf{x}_{i+1} = \mathbf{x}_i - s[P]^{-1}\nabla g\left(\mathbf{x}_i\right)$$

- Note that P matrix is composed of constants hence is **invariant** in $\mathbf{x}$. This allows us to **weight** each elements in the gradient vector differently.
  - Some of the variables are more important than the others. We emphasize the importance by using P to weight them differently.
  - The Hessian matrix can also be think of as a weighting matrix. But it weights by curvature and is variable across different points. That's why it is expensive and why we want to use Gradient Descent instead.
  - In contrast, the learning rate can be think of as a weight that applies identically to all elements in the gradient vector.
  - Similar concepts are used in choosing the optimal weighting matrix in **Generalized Method of Moment** and **Simulated Method of Moment**. Take an advanced Econometric course for details.

3.2.1 Implementation on R

- No more talking, just do it:

Hide

```
#Gradient Descent
#Differs from Newton-Raphson by that we no longer have the 2nd order derivative!

GradDesc<-function(fun,x0,s=1,tol=1e-6,h=1e-6,imax=1000) {
  history<-data.frame(i=rep(NA,imax),x0=rep(NA,imax),val0=rep(NA,imax),
                      x1=rep(NA,imax),val1=rep(NA,imax),diff=rep(NA,imax))
  i<-1

  while (i<=imax) {
    #Derivatives using central rule
    fx0<-(fun(x0+(h/2))-fun(x0-(h/2)))/h

    #Updating rule and check differences
    x1<-x0-s*fx0
    diff<-abs(x0-x1)
    history[i,]<-c(i,x0,fun(x0),x1,fun(x1),diff)

      if (diff<tol) {
        cat("Minimum found upon",i,"-th iteration at x=",x1,".\n")
        return(list(x=x1,val=history$val1[i],history=na.omit(history)))
        break
      }

      if (i==imax) {
        cat("Iteration ended before convergence. Returning results from the last iteration. \n")
        return(list(x=x1,val=history$val1[i],history=na.omit(history)))
        break
      }

      x0<-x1
      i<-i+1

  }
}
```

- Gradient descent on our good old friend $y = x^4 - 15x^2 + 8x + 30$. Now we no longer getting stuck by inflection and saddle points, and no longer climbing uphills. But we can still run into local solutions.

Hide

```
#No longer got stuck at inflection and saddle points, no longer climbing uphills.
rst<-GradDesc(fun=pf,x0=-0.5,s=0.015)
```

```
## Minimum found upon 9 -th iteration at x= -2.863301 .
```

Hide

```
rst<-GradDesc(fun=pf,x0=-sqrt(10)/2,s=0.015)
```

```
## Minimum found upon 7 -th iteration at x= -2.863301 .
```

Hide

```
rst<-GradDesc(fun=pf,x0=3,s=0.015)
```

```
## Minimum found upon 9 -th iteration at x= 2.594032 .
```

Hide

```
rst<-GradDesc(fun=pf,x0=-4,s=0.015)
```

```
## Minimum found upon 7 -th iteration at x= -2.863301 .
```

- The learning rate s is crucial: it easily go overshooting because we no longer use the information of curvature to **properly weight** the gradients.

Hide

```
#Role of convergence factor / learning rate
#s=0.015, converged quite well
GradDesc(fun=pf,x0=-0.5,s=0.015)
```

```
## Minimum found upon 9 -th iteration at x= -2.863301 .
```

```
## $x
## [1] -2.863301
##
## $val
## [1] -48.66852
##
## $history
##   i        x0       val0        x1       val1         diff
## 1 1 -0.500000  22.312500 -0.837500  13.270877 3.375000e-01
## 2 2 -0.837500  13.270877 -1.299129  -2.860634 4.616293e-01
## 3 3 -1.299129  -2.860634 -1.872182 -25.267962 5.730529e-01
## 4 4 -1.872182 -25.267962 -2.440937 -43.400263 5.687546e-01
## 5 5 -2.440937 -43.400263 -2.786747 -48.473242 3.458102e-01
## 6 6 -2.786747 -48.473242 -2.862277 -48.668480 7.553037e-02
## 7 7 -2.862277 -48.668480 -2.863327 -48.668516 1.049718e-03
## 8 8 -2.863327 -48.668516 -2.863301 -48.668516 2.646779e-05
## 9 9 -2.863301 -48.668516 -2.863301 -48.668516 6.812684e-07
```

Hide

```
#s=0.03, does not converge. Actually it bounces back and forth
rst<-GradDesc(fun=pf,x0=-0.5,s=0.03)
```

```
## Iteration ended before convergence. Returning results from the last iteration.
```

Hide

```
print(rst$history[-c(1:990),])
```

```
##         i        x0      val0        x1      val1      diff
## 991   991 -2.597812 -46.46794 -3.072043 -47.07264 0.4742307
## 992   992 -3.072043 -47.07264 -2.597812 -46.46794 0.4742307
## 993   993 -2.597812 -46.46794 -3.072043 -47.07264 0.4742307
## 994   994 -3.072043 -47.07264 -2.597812 -46.46794 0.4742307
## 995   995 -2.597812 -46.46794 -3.072043 -47.07264 0.4742307
## 996   996 -3.072043 -47.07264 -2.597812 -46.46794 0.4742307
## 997   997 -2.597812 -46.46794 -3.072043 -47.07264 0.4742307
## 998   998 -3.072043 -47.07264 -2.597812 -46.46794 0.4742307
## 999   999 -2.597812 -46.46794 -3.072043 -47.07264 0.4742307
## 1000 1000 -3.072043 -47.07264 -2.597812 -46.46794 0.4742307
```

Hide

```
#s=1, goes wild! This is actually a divergence. But it stopped just because the numerical derivative is treated as 0 by the computer due to roundoff error!
GradDesc(fun=pf,x0=-0.5,s=1)
```

```
## Minimum found upon 4 -th iteration at x= -4.409006e+14 .
```

```
## $x
## [1] -4.409006e+14
##
## $val
## [1] 3.778877e+58
##
## $history
##   i            x0         val0            x1         val1         diff
## 1 1 -5.000000e-01 2.231250e+01 -2.300000e+01 2.717520e+05 2.250000e+01
## 2 2 -2.300000e+01 2.717520e+05  4.794700e+04 5.285009e+18 4.797000e+04
## 3 3  4.794700e+04 5.285009e+18 -4.409006e+14 3.778877e+58 4.409006e+14
## 4 4 -4.409006e+14 3.778877e+58 -4.409006e+14 3.778877e+58 0.000000e+00
```

- Illustrate Gradient Descent in a multivariable context with $u = x^2 + y^2$.
    - Again we see that the choice of learning rate matters a lot: it might converge directly, converging in zigzag manner, or simply bouncing and never converge.

Hide

```
#Multivariate Minimization
ufn<-function(x,y){x^2+y^2}

grad<-function(x,y,h=1e-6) {
  xf<-x+(h/2)
  xb<-x-(h/2)
  yf<-y+(h/2)
  yb<-y-(h/2)

  fx<-(ufn(x=xf,y=y)-ufn(x=xb,y=y))/h
  fy<-(ufn(x=x,y=yf)-ufn(x=x,y=yb))/h

  return(matrix(c(fx,fy),ncol=1,nrow=2))
}

GradDescXY<-function(x0,y0,s=1,imax=50,h=1e-6) {

  i<-1
  history<-data.frame(i=c(0:imax),x=c(x0,rep(NA,imax)),y=c(y0,rep(NA,imax)),
                      diff=c(Inf,rep(NA,imax)))

  while(i<=imax) {
    fxv<-grad(x0,y0,h)
    x0v<-c(x0,y0)
    x1v<-x0v-s*fxv

    diff<-max(abs(x1v-x0v))
    history[i+1,]<-c(i,x1v[1],x1v[2],diff)

    if(diff<tol) {
      cat("Minimum found upon",i,"-th iteration.\n")
      rst<-list(x=c(x=x1v[1],y=x1v[2]),val=ufn(x=x1v[1],y=x1v[1]),history=na.omit(history))
      return(rst)
      }

    if(i==imax) {
      cat("Iteration ended before convergence. Returning the latest result. \n")
      rst<-list(x=c(x=x1v[1],y=x1v[2]),val=ufn(x=x1v[1],y=x1v[1]),history=na.omit(history))
      return(rst)
      }

      x0<-x1v[1]
      y0<-x1v[2]
      i<-i+1

  }

}

rs05<-GradDescXY(x0=1,y0=2,s=0.5)
```

```
## Minimum found upon 2 -th iteration.
```

Hide

```
rs07<-GradDescXY(x0=1,y0=2,s=0.7)
```

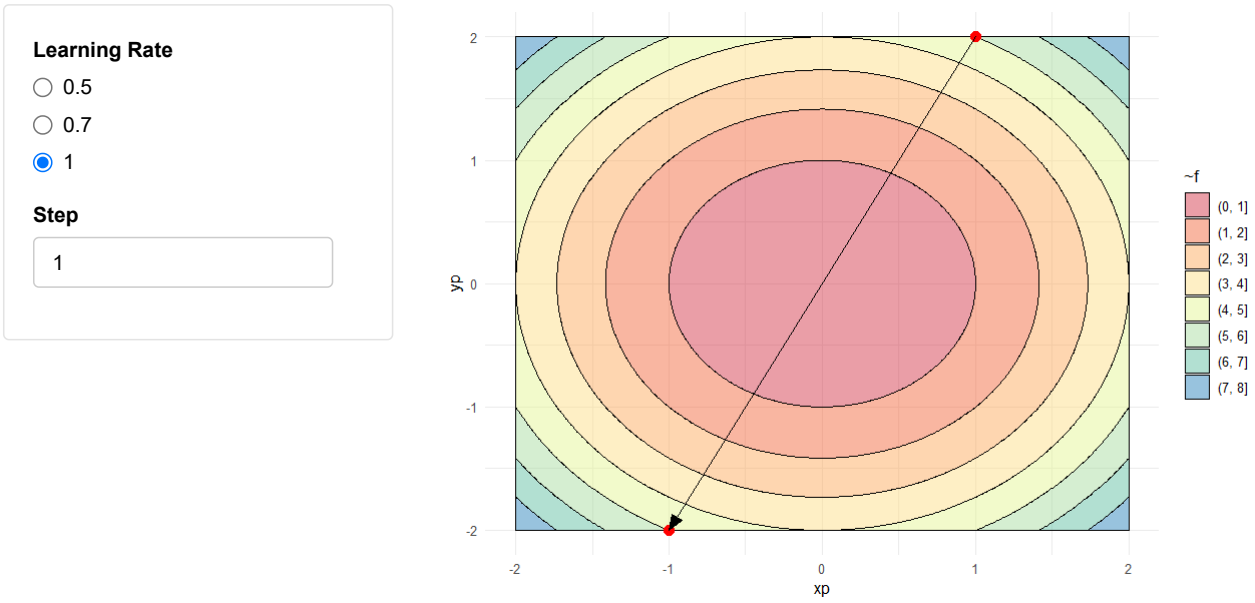
```
## Minimum found upon 18 -th iteration.
```

Hide

```
rs1<-GradDescXY(x0=1,y0=2,s=1)
```

```
## Iteration ended before convergence. Returning the latest result.
```

- Role of learning rate:



3.2.2 Remarks

- Gradient Descent do not use the second-derivative to weight the gradients, hence can go really wild easily.
- The choice of learning rate s matters. The choice of the weighting matrix P is also important.
- Learning rate can be determined using **line search algorithms**. But this is beyond our scope.
- Easy to implement, not very costly to compute. Probably why it is popular in machine learning, where the problems tend to be high-dimension and computation intensive already.

3.3 Quasi-Newton Methods (just to mention)

- The key idea is to **substitute** the Hessian matrix with other matrices.
  - The substitution matrix needs to capture the curvature of the function to some extent, and needs to be less expensive to compute.
  - Gradient Descent is different since it simply assign an arbitrary matrix.
- Frequently used Quasi-Newton Algorithm
  - BHHH algorithm:
    - Use the **outer product** of gradient vector as the substitute of the negative Hessian matrix:

$$-H\left(\mathbf{x}\right) \approx \begin{bmatrix} \left(\frac{\partial g(\mathbf{x})}{\partial x_1}\right)^2 & \frac{\partial g(\mathbf{x})}{\partial x_1}\frac{\partial g(\mathbf{x})}{\partial x_2} & \cdots & \frac{\partial g(\mathbf{x})}{\partial x_1}\frac{\partial g(\mathbf{x})}{\partial x_N} \\ \frac{\partial g(\mathbf{x})}{\partial x_2}\frac{\partial g(\mathbf{x})}{\partial x_1} & \left(\frac{\partial g(\mathbf{x})}{\partial x_2}\right)^2 & \cdots & \frac{\partial g(\mathbf{x})}{\partial x_2}\frac{\partial g(\mathbf{x})}{\partial x_N} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial g(\mathbf{x})}{\partial x_N}\frac{\partial g(\mathbf{x})}{\partial x_1} & \frac{\partial g(\mathbf{x})}{\partial x_N}\frac{\partial g(\mathbf{x})}{\partial x_2} & \cdots & \left(\frac{\partial g(\mathbf{x})}{\partial x_N}\right)^2 \end{bmatrix}$$

    - From the view of econometrics, this is similar to using the arithmetic definition of variance to construct the variance-covariance matrix (also the Hessian matrix in the context).

- BFGS and DFP algorithms:
  - Use secant updates to approximate the inverse Hessian matrix.
  - Generally start by using an identity matrix, then recursively update this matrix using changes in the variable and gradient.
  - Very similar to how we get the optimal weighting matrix in GMM. Take a course of advanced econometrics for details.
  - BFGS is more often used since it is more sophisticated than DFP.
- L-BFGS algorithm:
  - Based on BFGS, but it use "Limited-memory" hence is faster. Essentially it uses only the last m values of the changes in variable and gradient vectors instead of using the full history for iterating out the substituted Hessian matrix.

4 Concluding Remarks

- Methods taught here are similar in the sense that they are all related to **contraction mapping** and **fixed point** algorithms.
  - Both tackles optimization by reformulating the problem into a root finding problem.
  - Newton-Raphson cares about second-derivative, and this is consistent with what we learned in undergraduate calculus.
  - Quasi-Newton also cares about second-derivative, but it uses other gradient information to avoid doing the second-derivatives actually.
  - Gradient Descent cares about up- and downhills. Second-derivative is not used at all.
- Still there are problems:
  - Differentiability is required. Even Gradient Descent requires first-order differentiability.
  - Does not guarantee convergence.
  - Stuck at local extrema easily because we have no other mechanisms to escape from peaks and valleys once getting there!

5 Assignments

1. Consider a function $y = -3x^3 + 9x^2 + 2x$. Your missions are as follows:
   a. Design an R program that finds the optima of y. Your program needs to tell us whether the optima found is a maximum or minimum. It also needs to report the optimal points and values, the step sizes used, convergence factor used, whether the iteration converges or ended before convergence, and history of iteration.
   b. Use $x = 1.01$ and $x = 0.99$ as your initial points and find the optima.
   c. Use $x = 1$ as your initial point. You will run into a problem, figure out what is the problem and how to fix your optimization program.
2. Consider a function $z = -(1.5 - x)^2 - 100(y - x^2)^2$. The objective is to find (x, y) that **maximizes** this multivariate function. Do the following:
   a. Solve this optimization problem using Newton-Raphson. Solve the Hessian matrix by hands and supply it into the algorithm.
   b. Also use Newton-Raphson, but this time use Forward Rule of finite difference for the Hessian matrix.
   c. Do the same, but this time `hessian()` in `pracma` package for the Hessian matrix. You need to take a look at its documentation for how it works.
   d. Use Gradient Descent instead of Newton Raphson.
   e. Compare the results from a to d. Explain as detailed as possible.

I suggest that you design a program of Newton-Raphson algorithm that allows users to choose how the Hessian matrix is computed. Then you can use just one R program to do the first three questions by specifying the method of Hessian computation.