Numerical Differentiation

Ivan Yi-Fan Chen

2023 Fall

Finite Difference Approach

Univariate Finite Difference

- Recall the definition of derivative in undergraduate calculus:

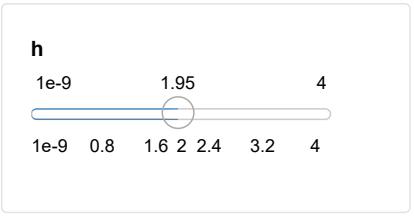
$$f'(x) \equiv \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

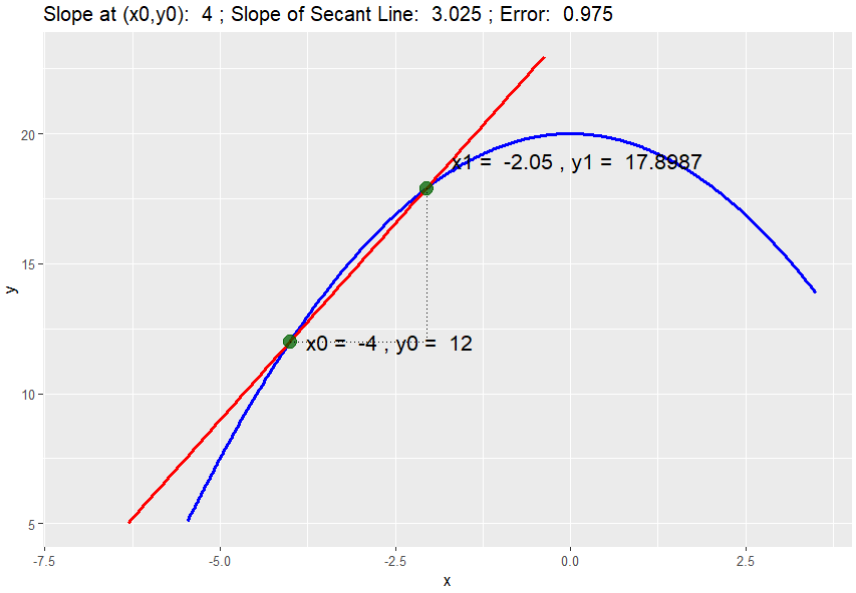
The derivative of $f(x)$ at point x is defined by the slope of its **secant line** when the change in x, i.e., h, is **infinitesimal**. In this limit, the secant line is essentially the **tangent line** at x. Hence the derivative of $f(x)$ is essentially the slope of the tangent line at x.

- Demonstrate the definition using the example

$$\begin{aligned} f(x) &= -\frac{1}{2}x^2 + 20 \\ \Rightarrow f'(x) &= -x \end{aligned}$$

We demonstrate how we approximate the slope of tangent line at $x = -4$.





- It is fairly intuitive to implement derivative: the definition of derivative gives us the *Forward Difference Rule*.
- **Forward Difference Rule:** We approximate the derivative at x by considering a forward move by an arbitrarily small step size h:

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

We start from x and step ahead by h to get the new value $f(x+h)$. This is why we refer to this rule as *Forward*.

- What really defines a derivative is an arbitrarily small deviation near the point of interest. So we don't need to restrict ourselves to start from x and move ahead by h. It is totally fine to move *back* by h, and also OK to center at x and move left and right by $h/2$. They are **essentially the same** when h is small enough. These are what we referred to as the *Backward* and *Central Difference Rule*.

- **Backward Difference Rule:** We approximate the derivative at x by moving backwards by an arbitrarily small step size h:

$$f'(x) \approx \frac{f(x-h)-f(x)}{h}$$

- **Central Difference Rule:** We approximate the derivative at x by using x as the center and looks at a forward and a backward move by $h/2$, where the (total) step size h is arbitrarily small

$$f'(x) \approx \frac{f(x+\frac{h}{2})-f(x-\frac{h}{2})}{h}$$

- Implementation of the above rules are simple: just find an h and plug things into the formula above!
- Demonstrate the derivatives of the 3 rules with $y = 1/(1+x)$ at $x = 4$. We choose $h = 10^{-6}$ as the step size. They are all very similar.

```
#Demonstrate Rules for Finite Difference Approach
#Forward Difference Rule
func<-function(x){1/(1+x)} #Define function to be differentiated
h<-10^-6 #Define step size
x<-4 #Set initial point
diff1.fwd<-(func(x+h)-func(x))/h #Take 1st derivative
diff1.fwd #Report result. True value is -0.04.
```

```
## [1] -0.03999999
```

```
#Backward Difference Rule
diff1.bwk<-(func(x)-func(x-h))/h #Take note of the numerator.
diff1.bwk
```

```
## [1] -0.04000001
```

```
#Central Difference Rule
diff1.cent<-(func(x+(h/2))-func(x-(h/2)))/h #Take note of the numerator.
diff1.cent
```

```
## [1] -0.04
```

Second- and Higher-order Derivatives

- We focusing on the Central Difference Rule to derive the Second-order Derivative. The same derivation applies to other two rules.
- It is fairly intuitive to approximate the Second-order Derivative by its definition:

$$f''(x) \approx \frac{f'(x+\frac{h}{2})-f'(x-\frac{h}{2})}{h} = \frac{\frac{f(x+\frac{h}{2}+\frac{h}{2})-f(x+\frac{h}{2}-\frac{h}{2})}{h} - \frac{f(x-\frac{h}{2}+\frac{h}{2})-f(x-\frac{h}{2}-\frac{h}{2})}{h}}{h}$$

  Note that the $x+\frac{h}{2}$ in $f'(x+\frac{h}{2})$ should be treated as x in the formula of Central Difference Rule for first derivative, and that $x-\frac{h}{2}$ in $f'(x-\frac{h}{2})$ should be treated as x in the formula of Central Difference Rule for first derivative.
- It is totally fine to write down the formula above in R directly for implementation. But with some algebraic manipulation we can make it simple:

$$f''(x) \approx \frac{f'(x+\frac{h}{2})-f'(x-\frac{h}{2})}{h} = \frac{f(x+\frac{h}{2})-2f(x)+f(x-\frac{h}{2})}{h^2}$$

- Demonstrate Second-order derivative of $y = 1/(1+x)$ at $x = 4$.

```
#Second Derivative using Central Difference Rule
diff2.cent<-(func(x+h)-2*func(x)+func(x-h))/(h^2) #Just follow the formula...
diff2.cent #True value is 0.016.
```

```
## [1] 0.01598721
```

- Under the Central Difference Rule, the derivative of order n is approximated by

$$f^{(n)}(x) \approx \frac{\sum_{i=0}^{n}(-1)^i C_i^n f(x+(\frac{n}{2}-i)h)}{h^n}$$

- Find Wikipedia for the formula for both Forward and Backward Rules: https://en.wikipedia.org/wiki/Finite_difference (https://en.wikipedia.org/wiki/Finite_difference)
- **In-Class Exercise:** Let x be the last digit of your student ID. Find the third derivative for $y = 1/(1+x)$ at x using the Central Difference Rule with $h = 10^-6$. You can do the math from the second-order derivative on your own. Or you can use the formula above. In this case, you need to use `factorial(n)` to compute $n!$. Note that $C_i^n = \frac{n!}{i!(n-i)!}$.

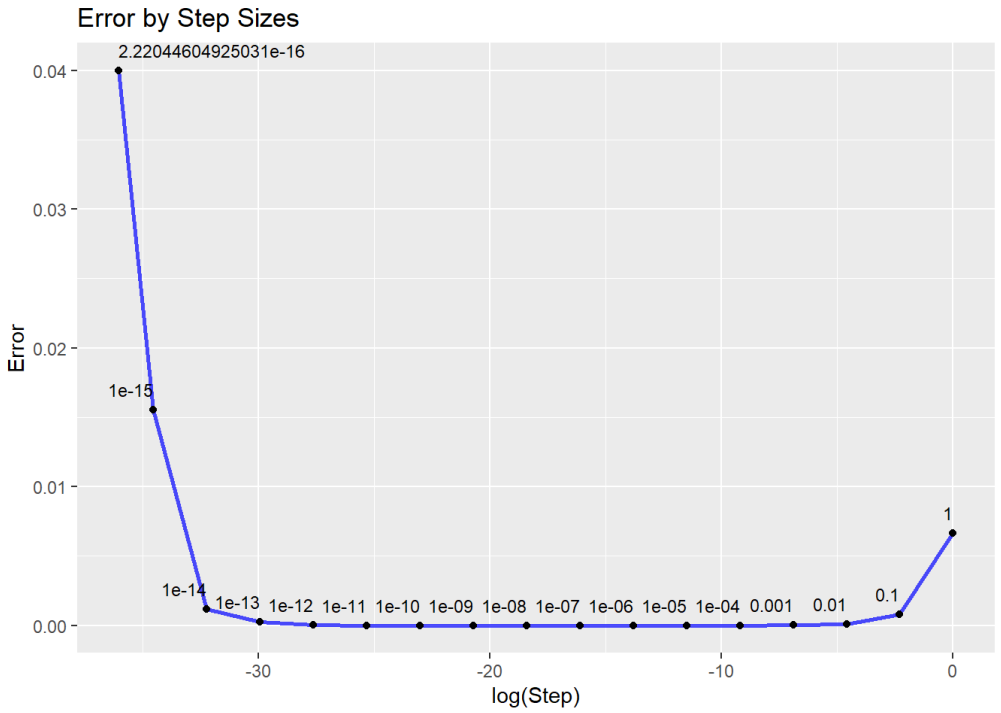
The Smaller the Step Size, the Better?

- By definition of derivative, h needs to be arbitrarily small. Intuitively you might want to try out setting $h = 10^{-100}$.
- But we are working on a computer, on which *real numbers* are approximated by *float numbers*. Thus we have a precision issue. The **machine epsilon** on R (and most of the programming languages) is 2^{-52} (falls somewhere between 10^{-16} and 10^{-15}).
  - Machine epsilon is a number, such that all numbers x less than it are treated as 0 in the sense that $1 + x = 1$. In a sloppy way, you can think that machine epsilon determines whether a number is too small to be meaningful hence can be treated as 0.
  - We can access R's machine epsilon using `.Machine$double.eps` .
  - The command `.Machine` returns a very long list that shows that precision and rounding setting on the current system. Use `?.Machine` for details.
- Demonstrate how the step size h affects accuracy. We start from $h = 1$ and stop at the machine epsilon.

```
#Try different levels of h, and plot out the results.
#Get machine epsilon
me<-.Machine$double.eps #About 2^-52, which falls between (10^-16,10^-15).
s<-c(0:15)
h<-c(10^(-s),me) #The sequence is 10^-i, i={0,...,15} plus machine epsilon.
rst<-data.frame(h,
                rep(NA,length(h)))%>%
  'colnames<-'(c("Step","Value"))

#Loop to get derivatives using different step sizes, and compute error.
#Here we use Forward Difference Rule for demonstration.
for (i in 1:length(h)) {
  rst$Value[i]<-(func(x+h[i])-func(x))/h[i]
}
rst<-mutate(rst,Error=abs(Value+0.04))

#Plot the error. We take log to the step sizes for better exposition.
ggplot(data=rst,aes(x=log(Step),y=Error))+
  geom_line(color="blue",size=1,alpha=0.7)+
  geom_point(aes(x=log(h)))+
  annotate(geom="text",x=log(rst$Step[-length(h)]),y=rst$Error[-length(h)],
           label=as.character(h)[-length(h)],
           size=3,vjust=-1,hjust=1)+
  annotate(geom="text",x=log(h[length(h)]),y=rst$Error[length(h)],
           label=as.character(h)[length(h)],size=3,vjust=-1,hjust=0)+
  labs(title="Error by Step Sizes")
```



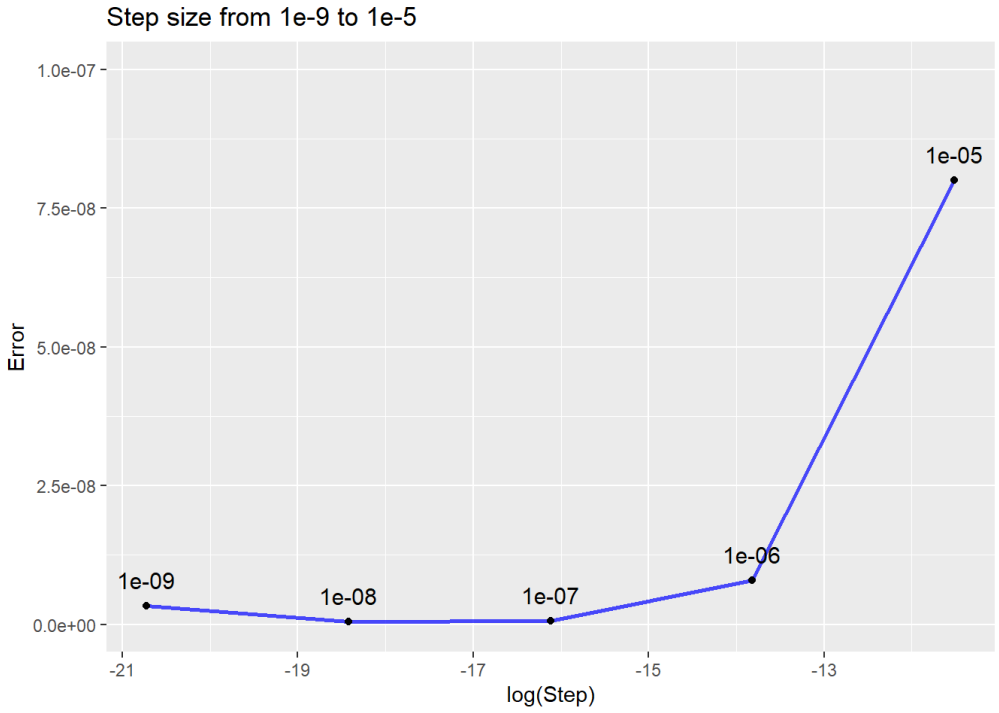
- Machine epsilon is the ***worst***, followed by 10^{-15}. Setting $h = 1$ is way more accurate than them!
- The error plot is, by and large, U-shaped. But it in fact fluctuates along the way if we add more points.
- Zoom in to check how 10^{-5} to 10^{-9} perform.

```
#Zoom to (1e-9,1e-5)
ggplot(data=rst,aes(x=log(Step),y=Error))+
  geom_line(color="blue",size=1,alpha=0.7)+
  geom_point(aes(x=log(h)))+
  annotate(geom="text",x=log(rst$Step[-length(h)]),y=rst$Error[-length(h)],
           label=as.character(h)[-length(h)],
           size=4,vjust=-1,hjust=0.5)+
  labs(title="Step size from 1e-9 to 1e-5")+
  ylim(c(0,1e-7))+xlim(c(log(1e-9),log(1e-5)))
```

```
## Warning: Removed 12 row(s) containing missing values (geom_path).
```

```
## Warning: Removed 12 rows containing missing values (geom_point).
```

```
## Warning: Removed 11 rows containing missing values (geom_text).
```



- Why is small h tend to be bad? There are two reasons.

1. The numerical derivative is by nature ill-behaved.

   For whatever rule, the approximator always takes the form $\frac{\Delta f(x)}{h}$, where $\Delta f(x)$ is obtained due to a perturbation at x by h. Since h is small, the denominator is close to 0. Since $f(x)$ must be a continuous function to be differentiable, a small change in h also leads to a small $\Delta f(x)$. Therefore the numerator is also close to 0. Then what we look at is similar to a $\frac{0}{0}$ indeterminate form! If h is too small, it is possible that we think of the approximator to be explosively large while in fact it is not.

2. Rounding issue due to the float number system.

   Computers *simulate* real numbers using the float system. The **real number** system is by construction an approximation in the sense that each real number is defined as a **least upper bound** of a sequence of numbers. For example, $\sqrt{2}$ is defined by the sequence containing infinitely many elements {1, 1.4, 1.41, 1.414,...}. In practice it is impossible to find infinitely many numbers. Instead, we need to define a small number as the “threshold”, such that we stop the

approximation below it and call it an accurate representation to the number we want. This is how float number system work conceptually. The threshold is machine epsilon. The following chunk shows how numbers above or below machine epsilon are treated.

```
1+10^-15 #Just above machine epsilon.
```

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

```
1+10^-16 #Just below machine epsilon
```

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

```
1+10^-17 #Below machine epsilon
```

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

```
1+10^-15==1+10^-16 #Different.
```

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

```
1+10^-16==1+10^-17 #Same!!
```

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

```
format(2^-53,nsmall=20) #Half of machine epsilon, hence is treated as 0.
```

```
## [1] "1.110223e-16"
```

```
format(1+2^-53,nsmall=20) #As if 1+0=1
```

```
## [1] "1.00000000000000000000"
```

```
format(2^-53+1,nsmall=20) #As if 0+1=1
```

```
## [1] "1.00000000000000000000"
```

But machine epsilon is only a threshold to stop approximating a number. We can still do arithmetic operations to a number below machine epsilon and making it big enough to be reported. For example, machine epsilon on R is 2^{-52}, hence 2^{-53} is treated as 0. But the following chunk shows that this small number can become meaningful due to the order of arithmetic operation. Specifically, if we do $2^{-53}+2^{-53}$ first then we make small numbers into machine epsilon so that doing further arithmetic operations with it becomes meaningful.

```
#Now we see how rounding is at play.
format(2^-53+2^-53,nsmall=20) #This gives us the machine epsilon, the smallest number on the system.
```

```
## [1] "2.220446e-16"
```

```
format(1+2^-53+2^-53,nsmall=20) #Being treated as 0 so we get 1.
```

```
## [1] "1.00000000000000000000"
```

```
format(2^-53+2^-53+1,nsmall=20) #But here it is NOT treated as 0 since we are adding with 2^-52!
```

```
## [1] "1.00000000000000022204"
```

```
#Since summation works from left to right!
```

The above nature of float number system leads to an rounding issue. There are small numbers in the expressions already, and they can accumulate during arithmetic operations and eventually becomes large enough to be meaningful. This leads to errors in the results.

- How to improve the accuracy?

1. Use Central Difference Rule. It is proven that the error term of Central Rule converges to 0 at order $O(h^2)$, while for Forward Rule it is at order $O(h)$.
    - $O(.)$ is referred to as the "Big O notation" in computer science and econometrics. We say that $f(x)$ is $O(g(x))$ if for a a function that $g(x) > 0$ there exists a M and a x_0 such that we have $|f(x)| \leq Mg(x)$ for all $x > x_0$. If we think of $f(x)$ as an error term, then $O(g(x))$ says that the error is no larger than $g(x)$ when x is large enough.

      See Wikipedia: https://en.wikipedia.org/wiki/Big_O_notation (https://en.wikipedia.org/wiki/Big_O_notation)
    - In a nutshell, h^2 is smaller than h when h is less than 1 as required by numerical derivative. Therefore the Central Rule is more accurate than the Forward Rule.
2. A rule of thumb to improve the precision issue is to simply use the **square root of machine epsilon**. For R, it is exactly $\sqrt{2^{-52}} = 2^{-26}$. We demonstrate with the same example. It out performs all h in the previous example except 10^{-7} and 10^{-8}.

```
#Use square root of machine epsilon as our step size.
oh<-sqrt(me) #About 1.5e-8, so is between 1e-8 and 1e-7.
val.oh<-(func(x+oh)-func(x))/oh #Use Forward Difference Rule for comparison.
error.oh<-abs(val.oh+0.04)
error.oh #Report Error
```

```
## [1] 9.685755e-10
```

```
#Find the step sizes in the previous example that are more accurate than using `oh`.
rst[which(rst$Error<error.oh),]%>%print()
```

```
##    Step Value        Error
## 8 1e-07 -0.04 5.761657e-10
## 9 1e-08 -0.04 5.340573e-10
```

3. In practice, say IT companies, the step size is oftentimes fixed at 10^{-6} or 10^{-5}.

In-Class Exercises

1. Divide the interval $x \in [1, 2]$ into 100 equidistant points with `seq()`.
2. Compute the *second-derivative* of $y = 3x^2 + \ln(x^2)$ with the Central Rule at the 100 points you just obtained using the following step sizes: $h \in \{2^{-26}, 10^{-7}, 10^{-6}, 10^{-5}\}$.
3. Note that the true value is $y' = 6x + \frac{2}{x}$. Find the error for each point, and make an error plot with error on y-axis, x on x-axis, and use different lines for different levels of h.

4. Nash (1990) proposes to use a location-dependent step size. Specifically, he suggests that we use $h = \epsilon \times (|x| + \epsilon)$ where ϵ is a small number. Repeat 2 and 3 using Nash's approach, and we compare diffferent settings of ϵ by trying out $\epsilon \in \{2^{-26}, 10^{-7}, 10^{-6}, 10^{-5}\}$.

Multivariate Finite Difference

- This is done similarly to the univariate case. The partial derivatives are exactly the same as what we have learned before, except that we treat all other variables as parameters. For example, for $f(x, y)$ the Central Rule gives us

$$\frac{\partial f(x,y)}{\partial x} \approx \frac{f(x + \frac{h}{2}, y) - f(x - \frac{h}{2}, y)}{h}$$

$$\frac{\partial^2 f(x,y)}{\partial x^2} \approx \frac{f(x + \frac{h}{2}, y) - 2f\left(x, y\right) + f(x - \frac{h}{2}, y)}{h^2}$$

- Cross derivative is a bit complicated, but it still follows the calculus definition. Use $f(x, y)$ as an example, we have

$$\frac{\partial^2 f(x,y)}{\partial y \partial x} = \lim_{k \to 0} \frac{\frac{\partial f(x, y+k)}{\partial x} - \frac{\partial f(x,y)}{\partial x}}{k} = \lim_{k \to 0} \lim_{h \to 0} \frac{\frac{f(x+h, y+k) - f(x, y+k)}{h} - \frac{f(x+h, y) - f(x,y)}{h}}{k}$$

  Therefore we can approximate by Central Rule as

$$\frac{\partial^2 f(x,y)}{\partial y \partial x} \approx \frac{f\left(x + \frac{h}{2}, y + \frac{k}{2}\right) - f\left(x, y + \frac{k}{2}\right) - f\left(x + \frac{h}{2}, y\right) + f\left(x, y\right)}{hk}$$

  - The cross derivative $\partial^2 f(x,y) / \partial x \partial y$ is analogously obtained.

- The following demonstrates how we perform partial derivatives to $U = x^\alpha y^{1-\alpha}$ at $(x, y) = (1, 1)$ given that $\alpha = 0.7$. For simplicity we set $h = k$ and try out different levels of step sizes.

```
U<-function(x,y,a) {(x^a)*(y^(1-a))} #Cobb-Douglas.
#Suppose that a = 0.7, and we look at the derivative at x=y=1.

#Use Central Rule to construct the derivative.
#Pack it into a function so that we can try different steps.
dU12<-function(h) {
  #Partial derivative of x
fx<-(U(x=1+(h/2),y=1,a=0.7)-U(x=1-(h/2),y=1,a=0.7))/h

#Partial derivative of y
fy<-(U(x=1,y=1+(h/2),a=0.7)-U(x=1,y=1-(h/2),a=0.7))/h

#Second derivative of x
fxx<-(U(x=1+h,y=1,a=0.7)-2*U(x=1,y=1,a=0.7)+U(x=1-h,y=1,a=0.7))/(h^2)

#Second derivative of y
fyy<-(U(x=1,y=1+h,a=0.7)-2*U(x=1,y=1,a=0.7)+U(x=1,y=1-h,a=0.7))/(h^2)

#Cross deriviative: x first then y
fxy<-((U(x=1+(h/2),y=1+(h/2),a=0.7)-U(x=1-(h/2),y=1+(h/2),a=0.7))/h-
      (U(x=1+(h/2),y=1-(h/2),a=0.7)-U(x=1-(h/2),y=1-(h/2),a=0.7))/h)/h

#Cross derivative: y first then x
fyx<-((U(x=1+(h/2),y=1+(h/2),a=0.7)-U(x=1+(h/2),y=1-(h/2),a=0.7))/h-
      (U(x=1-(h/2),y=1+(h/2),a=0.7)-U(x=1-(h/2),y=1-(h/2),a=0.7))/h)/h

#Report the output
out<-c(h,fx,fy,fxx,fyy,fxy,fyx)%>%'names<-'(c("h","fx","fy","fxx","fyy","fxy","fyx"))
return(out)
}

#The true values are that fx=0.7, fy=0.3, fxx=fyy=-0.21, fxy=fyx=0.21

rst<-lapply(c(1e-9,1e-8,sqrt(me),1e-7,1e-6,1e-5,1e-4),dU12)
rst<-do.call(rbind,rst)
rst #1e-7 and higher are way more reasonable!
```

```
##                 h        fx  fy         fxx        fyy       fxy       fyx
## [1,] 1.000000e-09 0.7000001 0.3 111.0223025  0.0000000 0.0000000 0.0000000
## [2,] 1.000000e-08 0.7000000 0.3  -1.1102230  0.0000000 1.1102230 1.1102230
## [3,] 1.490116e-08 0.7000000 0.3   0.0000000 -0.5000000 0.5000000 0.5000000
## [4,] 1.000000e-07 0.7000000 0.3  -0.2109424 -0.2109424 0.2109424 0.2109424
## [5,] 1.000000e-06 0.7000000 0.3  -0.2100542 -0.2100542 0.2099432 0.2099432
## [6,] 1.000000e-05 0.7000000 0.3  -0.2099998 -0.2099987 0.2100009 0.2100009
## [7,] 1.000000e-04 0.7000000 0.3  -0.2100000 -0.2100000 0.2100000 0.2100000
```

```
#Try out Nash's way but we revise it to fit into multidimension.
#We use Euclidean Norm for |x|, i.e., |(x-a,y-b)| = sqrt((x-a)^2+(y-b)^2).
#We compute the distance to the origin so a=b=0.
nashh<-function(t) {t*(sqrt(1^2+1^2)+t)}
h<-nashh(c(1e-9,1e-8,sqrt(me),1e-7,1e-6,1e-5,1e-4))
rst<-lapply(h,dU12)
rst<-do.call(rbind,rst)
rst #Similar to above: those computed from 1e-7 or above are way better. But yes, error reduced.
```

```
##                 h  fx        fy        fxx        fyy       fxy       fyx
## [1,] 1.414214e-09 0.7 0.3000001  0.0000000 55.5111512 0.0000000 0.0000000
## [2,] 1.414214e-08 0.7 0.3000000  0.0000000  0.0000000 0.5551115 0.5551115
## [3,] 2.107342e-08 0.7 0.3000000  0.0000000  0.0000000 0.0000000 0.0000000
## [4,] 1.414214e-07 0.7 0.3000000 -0.2109423 -0.2109423 0.2109423 0.2109423
## [5,] 1.414215e-06 0.7 0.3000000 -0.2100539 -0.2099984 0.2100539 0.2100539
## [6,] 1.414224e-05 0.7 0.3000000 -0.2100002 -0.2100002 0.2100002 0.2100002
## [7,] 1.414314e-04 0.7 0.3000000 -0.2100000 -0.2100000 0.2100000 0.2100000
```

- Introduce some terminologies before ending this section. Consider a multivariate function $f(\mathbf{x})$ where $\mathbf{x} \equiv (x_1, x_2, \ldots, x_N)$.
  - **Gradient vector:** the vector of partial derivatives of the multivariate function with respect to each of its variable.

$$\nabla f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f(\mathbf{x})}{\partial x_1} & \frac{\partial f(\mathbf{x})}{\partial x_2} & \cdots & \frac{\partial f(\mathbf{x})}{\partial x_N} \end{bmatrix}_{1 \times N}$$

  - **Hessian matrix:** the matrix of second-order partial derivatives and cross derivatives between all variables of the multivariate function

$$\nabla^2 f(\mathbf{x}) = \begin{bmatrix} \frac{\partial^2 f(\mathbf{x})}{\partial x_1^2} & \frac{\partial^2 f(\mathbf{x})}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f(\mathbf{x})}{\partial x_1 \partial x_N} \\ \frac{\partial^2 f(\mathbf{x})}{\partial x_2 \partial x_1} & \frac{\partial^2 f(\mathbf{x})}{\partial x_2^2} & \cdots & \frac{\partial^2 f(\mathbf{x})}{\partial x_2 \partial x_2} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial^2 f(\mathbf{x})}{\partial x_N \partial x_1} & \frac{\partial^2 f(\mathbf{x})}{\partial x_N \partial x_2} & \cdots & \frac{\partial^2 f(\mathbf{x})}{\partial x_N^2} \end{bmatrix}_{N \times N}$$

  - The Hessian matrix is always a square matrix with second-order partial derivatives falling on its diagonal. The cross derivatives always fall off the diagonal.
  - The Hessian matrix is symmetric if the function is **continuously differentiable**.
- Not to confuse Hessian matrix with **Jacobian matrix**. A Jacobian matrix is a matrix of first-order partial derivatives for M multivariate functions with respect to all of their variables:

$$\mathbf{J} = \begin{bmatrix} \frac{\partial f_1(\mathbf{x})}{\partial x_1} & \frac{\partial f_1(\mathbf{x})}{\partial x_2} & \cdots & \frac{\partial f_1(\mathbf{x})}{\partial x_N} \\ \frac{\partial f_2(\mathbf{x})}{\partial x_1} & \frac{\partial f_2(\mathbf{x})}{\partial x_2} & \cdots & \frac{\partial f_2(\mathbf{x})}{\partial x_N} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial f_M(\mathbf{x})}{\partial x_1} & \frac{\partial f_M(\mathbf{x})}{\partial x_2} & \cdots & \frac{\partial f_M(\mathbf{x})}{\partial x_N} \end{bmatrix}_{M \times N}$$

Remarks

- After all of those sufferings, actually there is a short cut on R. The `pracma` package offers `fderiv()` that performs finite difference rule for up to 8-degree of derivatives. Try it on your own, and check `?fderiv` for details.
- Finite difference rule in nature is an approximation, and suffers from approximation errors stemming from the choice of h, rounding issue of float number system, and the behavior of the function $f(x)$. The numerical results can go really wild.
- Do the derivative manually and then evaluate it in R should be the most accurate way. But most of the time the functions are just way too complicated to manually taking derivatives. Sometimes it is even impossible to do so when the model becomes highly non-linear in a high dimension.
- The way out is **Automatic Differentiation**. It gives you the **exact** derivative.

Automatic Differentiation Approach

- Automatic Differentiation is an algorithm that yields the derivatives **exactly**. Features:

1. The results are **exact** since no approximations are involved.
2. It handles **chain rules** automatically.

- Idea:
  1. To get exact results, you should just provide the functional form of derivative to R for evaluation. This is not a problem for basic functions. But for complicated functions this is impractical.
  2. Functions are complicated because they are composed of many small functions. Doing it manually is difficult because a lot of chain rules will be applied. But computers evaluate a function following a similar manner to how we perform chain rule. What if we just provide the derivatives of basic formula and then let computer handle the chain rule part?
- The idea above is implemented by the **dual number** system.

Dual Number

- A dual number takes the form $a + b\epsilon$, such that
  - a and b are real numbers, where a is the *real part* and b is the *dual part*.
  - ϵ is *nilpotent*, i.e., $\epsilon^2 = 0$ for all ϵ.
- Consider the dual numbers $x = a_1 + b_1\epsilon$ and $y = a_2 + b_2\epsilon$. Their arithmetic operation are given as follows, and we see that ϵ being nilpotent matters for both multiplication and division.
  - Addition and Subtraction:

$$x \pm y = (a_1 + b_1\epsilon) \pm (a_2 + b_2\epsilon) = (a_1 \pm a_2) + (b_1 \pm b_2)\epsilon$$

  - Multiplication: Note that $\epsilon^2 = 0$, we have

$$xy = (a_1 + b_1\epsilon)(a_2 + b_2\epsilon) = a_1a_2 + a_1b_2\epsilon + a_2b_1\epsilon$$

  - Division: Since $epsilon^2 = 0$, we have

$$\frac{x}{y} = \frac{a_1 + b_1\epsilon}{a_2 + b_2\epsilon} = \frac{(a_1 + b_1\epsilon)(a_2 - b_2\epsilon)}{(a_2 + b_2\epsilon)(a_2 - b_2\epsilon)} = \frac{a_1a_2 - a_1b_2\epsilon + a_2b_1\epsilon}{a_2^2} = \frac{a_1}{a_2} + \frac{a_2b_1 - a_1b_2}{a_2^2}\epsilon$$

  - Multiplication with a constant c:

$$cx = ca_1 + cb_1\epsilon$$

- Recall Taylor expansion of function $f(x)$ at $x = a$:

$$f(x) = f(a) + f'(a)(x - a) + f''(a)\frac{(x - a)^2}{2} + f'''(a)\frac{(x - a)^3}{3!} + \dots$$

- If we perform Taylor expansion to $f(x)$ at $x = a + b\epsilon$, then we have $x - a = b\epsilon$ hence

$$f(a + b\epsilon) = f(a) + f'(a)b\epsilon + f''(a)\frac{(b\epsilon)^2}{2} + f'''(a)\frac{(b\epsilon)^3}{3!} + \dots$$

  Because $\epsilon^2 = 0$, all terms involving in third- or higher-order derivatives directly becomes 0. Thus we get

$$f(a + b\epsilon) = f(a) + f'(a)b\epsilon$$

  Note that this is an **exact** relation, not an approximation.

- The Taylor expansion implies that, if we plug a dual number into a function, the returned value is a dual number such that its real part is the function evaluated at the real part $f(a)$, and the dual part is proportional to **first-derivative evaluated at the real part** $f'(a)$.
- Let $b = 1$, we always have

$$f(a + b\epsilon) = f(a) + f'(a)\epsilon$$

meaning that the dual part is exactly the derivative evaluated at a.

- Look at some more examples to convince yourselves:

1. Power function $f(x) = x^k$

$$f(a + \epsilon) = a^k + ka^{k-1}\epsilon$$

2. Exponential function $f(x) = e^x$

$$f(a + \epsilon) = e^a + e^a\epsilon$$

3. Natural Log $f(x) = \ln x$

$$f(a + \epsilon) = \ln a + \frac{1}{a}\epsilon$$

- **Chain Rule is Built-in in dual numbers**. To see this, consider evaluating $f(g(x))$ at $x = a + \epsilon$. The Taylor expansion above leads to the following:

$$f(g(a + \epsilon)) = f(g(a) + g'(a)\epsilon)$$

Let $\widehat{a} \equiv g(a)$ and $\hat{b} \equiv g'(a)$, we can further rewrite the above as

$$f(g(a + \epsilon)) = f(\widehat{a} + \hat{b}\epsilon) = f(\widehat{a}) + f'(\widehat{a})\hat{b}\epsilon = f(g(a)) + f'(g(a))g'(a)\epsilon$$

Obviously, **the dual part of the composite function is exactly the chain rule**.

- The above implies two key points when implementing dual number on a programming language:

1. Need to revise both multiplication and division so that the computer knows that $\epsilon^k = 0$ for $k \geq 2$. Conceptually we need to declare that the object is a dual number. Then we modify the arithmetic operators so that they check if the object we feed in is a dual number. If yes, then the operators use the arithmetic rule for dual numbers.
2. Need to revise math functions. We need to modify these functions so that they check if the object they get is a dual number or not. If yes, then evaluate following the Taylor's expansion above. For this process, we need to **hand code** the derivative and supply it into the R-function in this process.

- In other words, we need to define dual number and supply the derivatives of basic functions to the computer. Then the algorithm of computer will takeover the evaluation. If we are able to do so, then chain rule is automatically applied no matter how complicated the function we really want to look at is.

Further Remarks on Automatic Differentiation

- If dual number is properly introduced, then functions are evaluated at dual numbers in the way that we have introduced.
- But we didn't "code" the chain rule in the process. All we have done is to introduce the dual part, revise the arithmetic and functions and provide the derivatives if needed.

- The chain rule part is based on how computers evaluate a function. In general, the computer evaluates the function layer-by-layer with one of the two algorithms: the **forward mode / forward accumulation** and the **reverse mode / reverse accumulation**.
- For a function $y = f(x)$, applying chain rule to it leads to

$$\frac{dy}{dx} = \frac{dy}{dw_1}\frac{dw_1}{dx} = \frac{dy}{dw_2}\frac{dw_2}{dw_1}\frac{dw_1}{dx} = \ldots = \frac{dy}{dw_N}\frac{dw_N}{dw_{N-1}}\cdots\frac{dw_1}{dx}$$

  where we can think of each w_i as a "node". The two algorithms differ by the order of evaluation.

  - Forward mode evaluates the derivative inside out, i.e., dw_1/dw, then dw_2/dw_1, and so on.
  - Reverse mode evaluates the derivative starting from the outside, i.e., dy/dw_N, then dw_N/dw_{N-1} and so on.
- Consider an example from Wikipedia: $y = x_1x_2 + \sin x_1$ such that we are interested in solving for $\partial y/\partial x_1$. We can break this function into 5 nodes:

1. $w_1 \equiv x_1$, hence $dw_1 = dx_1$.
2. $w_2 \equiv x_2$, hence $dw_2 = dx_2$.
3. $w_3 \equiv w_1w_2$, hence $dw_3 = w_1(dw_2) + (dw_1)w_2$
4. $w_4 \equiv \sin w_1$, hence $dw_4 = \cos w_1 dw_1$
5. $w_5 \equiv w_3 + w_4$, hence $dw_5 = dw_3 + dw_4$

- Forward mode evaluates inside out. Therefore, it assigns $dx_1 = 1$, $dx_2 = 0$ and x_1 and x_2 as the values we assigned. Then it plug them into the nodes for function value and derivative value following the order above (from 1 to 5).
- Backward mode evaluates outside in, and is a bit counter intuitive (but said to be computationally more efficient):

1. Since $y = w_5$, we have $\partial y/\partial w_5 = 1$.
2. Since $w_5 = w_3 + w_4$, we have $\partial y/\partial w_4 = (\partial y/\partial w_5)(\partial w_5/\partial w_4)$, where $\partial w_5/\partial w_4 = 1$.
3. Since $w_5 = w_3 + w_4$, we have $\partial y/\partial w_3 = (\partial y/\partial w_5)(\partial w_5/\partial w_3)$, where $\partial w_5/\partial w_3 = 1$.
4. Since $w_3 = w_1w_2$, we have $\partial y/\partial w_2 = (\partial y/\partial w_3)(\partial w_3/\partial w_2)$, where $(\partial w_3/\partial w_2) = w_1$.
5. $\partial y/\partial w_1$ is a bit complicated since w_1 affects y via both w_3 and w_4. We distinguish the effects from the two margins as follows:

   5a. $\partial y/\partial w_{1,3} = (\partial y/\partial w_3)(\partial w_3/\partial w_{1,3})$, where $\partial w_3/\partial w_{1,3} = w_2$.

   5b. $\partial y/\partial w_{1,4} = (\partial y/\partial w_4)(\partial w_4/\partial w_{1,4})$, where $\partial w_4/\partial w_{1,4} = \cos w_{1,4}$.

   5c. Together we have $\partial y/\partial w_1 = \partial y/\partial w_{1,3} + \partial y/\partial w_{1,4}$.

   Following the order above, the computer combines the results from each stage in order and yields both the functional value and derivatives.

Implementation on R

- Unfortunately R does not have built-in dual number. You either program it on your own, or use a dedicated package.
- For programming, in general this involves in the following steps:

  1. Need to declare a new `class` (equivalent to `struct` in C, Matlab and Julia) to introduce dual number following its definition.
  2. Need to modify the basic arithmetic in order to adopt dual number. In short, you need to teach R how arithmetic operations is performed for the dual number class.

3. Similar to Step 2, you need to modify the math functions like `exp` and `log` and other special functions like `gamma` so that they know what to do with dual numbers.
    - These are somewhat easy to perform on Julia, but for R you will need to have knowledge on *functional programming*. Hadley Wickham's book is your best friend if you really want to dive into it: http://adv-r.had.co.nz/OO-essentials.html (http://adv-r.had.co.nz/OO-essentials.html)

- For dedicated packages, there are some…
    - `dual` introduces dual number and have adopted it to basic arithmetic and frequently used functions. See `?dual` for details.
    - `madness` is a more recent package that implements higher order and higher dimension automatic differentiation on R.
    - `autodiffr` is a wrapper that allows R users to run packages of automatic differentiation built for Julia. Need to install Julia, and this package needs to be downloaded from GitHub: https://github.com/Non-Contradiction/autodiffr (https://github.com/Non-Contradiction/autodiffr)
- We will be demonstrate with `dual` package. Explore its documentation on your own.

```
library(dual)
x<-dual(f=1, grad=3) #Define a dual number with real part 1 and dual part 3.
class(x) #A dual class
```

```
## [1] "dual"
## attr(,"package")
## [1] "dual"
```

```
typeof(x) #A S4 functional call. This is beyond our scope. If you are interested in programming a new structure and package in R. Check Hadley's book for it.
```

```
## [1] "S4"
```

```
str(x) #Shows the strucutre of a dual object
```

```
## Formal class 'dual' [package "dual"] with 2 slots
##   ..@ f   : num 1
##   ..@ grad: num 3
```

```
attributes(x) #Returns a named list containing items in x.
```

```
## $f
## [1] 1
##
## $grad
## [1] 3
##
## $class
## [1] "dual"
## attr(,"package")
## [1] "dual"
```

```
#Need to use attributes if we want to retrieve items from x
attributes(x)$f
```

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

- Arithmetic of dual numbers

```
x<-dual(f=3, grad=1)
y<-dual(f=2, grad=2)
x+y
```

```
## Real: 5.000000
## Duals: 3.000000
```

```
x-y
```

```
## Real: 1.000000
## Duals: -1.000000
```

```
x*y
```

```
## Real: 6.000000
## Duals: 8.000000
```

```
x/y
```

```
## Real: 1.500000
## Duals: -1.000000
```

- Derivatives for basic functions. Plugging `x` into the functions yields the derivatives evaluated at $x = 3$ since the real part of `x` is set to 3.

```
#Power function, should return 6 since y' = 2x
x^2
```

```
## Real: 9.000000
## Duals: 6.000000
```

```
#Exponential function. Can be checked by the fact that y'=y=exp(x)
exp(x)
```

```
## Real: 20.085537
## Duals: 20.085537
```

```
exp(x)%>%attributes()%>%'$'("grad")%>%'=='(exp(3))
```

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

```
#Logarithmics, can be checked by y'=1/x
log(x)
```

```
## Real: 1.098612
## Duals: 0.333333
```

```
log(x)%>%attributes()%>%'$'("grad")%>%'=='(1/3)
```

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

```
#Trignometrics.
sin(x)
```

```
## Real: 0.141120
## Duals: -0.989992
```

```
sin(x)%>%attributes()%>%'$'("grad")%>%'=='(cos(3))
```

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

```
cos(x)
```

```
## Real: -0.989992
## Duals: -0.141120
```

```
cos(x)%>%attributes()%>%'$'("grad")%>%'=='(-sin(3))
```

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

```
tan(x)
```

```
## Real: -0.142547
## Duals: 1.020320
```

```
tan(x)%>%attributes()%>%'$'("grad")
```

```
## [1] 1.02032
```

```
(1/cos(3))^2 #Direct comparison do not go through because of rounding error. But similar enough!
```

```
## [1] 1.02032
```

- Check the basic rules of derivatives using `x`

```
#Summation rule
x^2+3*x^3 #y' = 2x+9x^2 = 87 at x=3. Yes correct.
```

```
## Real: 90.000000
## Duals: 87.000000
```

```
#Subtraction rule
x^2-3*x^3 #y' = 2x-9x^2 = -75. Correct.
```

```
## Real: -72.000000
## Duals: -75.000000
```

```
#Product rule
(x^2)*log(x) #y' = 2x*log(x)+x^2*x^-1 = 9.591674. Correct.
```

```
## Real: 9.887511
## Duals: 9.591674
```

```
#Quotient rule
x/(1+x) #y' = 1/((1+x)^2)=1/16=0.0625. Correct.
```

```
## Real: 0.750000
## Duals: 0.062500
```

- Complicated chain rule

```
exp(-exp(-x)) #y' = exp(-(x+exp(-x))) = 0.04737. Correct.
```

```
## Real: 0.951432
## Duals: 0.047369
```

```
#Take derivative to the CDF of standard normal distribution at x=3. It involves in the "Error Functio
n", a special function that is based on the Gamma function. Fortunately this is supported.
#But we need to call it by `dual::erf()` since the same function is used by other active packages.
(1/2)*(1+dual::erf(x/sqrt(2))) #Returns 0.004432.`
```

```
## Real: 0.998650
## Duals: 0.004432
```

```
dnorm(x=3,mean=0,sd=1) #Check PDF of standard normal at x=3. Similar!
```

```
## [1] 0.004431848
```

- All sounds good. But there are drawbacks:

1. Need to "redefine" the math functions to take dual number. For example, `dnorm(x,0,1)` returns an error since `dual` does not adopt the `dual` class to `dnorm`. One will need to use the definition of normal pdf for its derivative, instead of calling directly from `dnorm`.

   This means that one need to be familiar with the derivative properties of the functions as well as programming if we are to bring functions of interest to automatic differentiation!

2. Need specialized algorithms to further bring automatic differentiation to higher order derivatives.

- Nevertheless it is still good to learn about it. At least you know what you are doing if you will be using a full-fledged program dedicated for automatic differentiation.

Assignment

- Wrap the formula of Central Difference Rule into an R function that handles **one-dimensional** derivatives to *all* orders. Try to make this function as flexible as possible. For example, you may want to allow for additional parameters to be passed to the expression that we are going to differentiate.
- Use the function you just built to compute the **first-order** derivative to the function $y = \frac{\ln(1-x^2)}{\ln(x+1)}$ **near** $x = 0$. Let's set $h = 10^{-5}$. Since this function is undefined at $x = 0$, you should start from $x = 0.1$ and gradually reduce x so that it gets closer to 0. Then for each x you compute the numerical derivative, and repeat until the numerical derivative becomes stable at 10^{-6} level.

- Repeat the above. But this time use Nash's approach with $\epsilon = 10^{-5}$.