Numerical Integration: Quadrature Approach

Ivan Yi-Fan Chen

2023 Fall

Preliminaries for R

Customized Function

- Use `function` to implement an equation or something even more complicated.
- Consider an equation $1/(1+y)$, what happens if we just write this in R?

```
1/(1+y) #Gives you an error message because we haven't defined `y` yet!
```

```
## Error in eval(expr, envir, enclos): 找不到物件 'y'
```

```
y<-1 #Define y as a number
1/(1+y) #Gives you a number.
```

```
## [1] 0.5
```

```
y<-c(1,2,3) #Define a vector
1/(1+y) #Gives you a vector such that each element is evaluated.
```

```
## [1] 0.5000000 0.3333333 0.2500000
```

- But all you have are just *numbers*. A number is just a number, you can't do manipulations such as integration or derivatives.
- Define a single variable function

```
fun1<-function(x) {1/(1+x)} #Define a function
fun1 #Gives you the definition of the function
```

```
## function(x) {1/(1+x)}
## <environment: 0x0000000026e07c10>
```

```
fun1(1) #Gives you a number by evaluating at x = 1.
```

```
## [1] 0.5
```

```
k<-1 #Assign a number to a variable.
fun1(k) #Directly evaluate with the variable.
```

```
## [1] 0.5
```

```
fun1(x = k) #The same as the above one.
```

```
## [1] 0.5
```

```
fun1() #Gives you an error message saying that arguments are not provided.
```

```
## Error in fun1(): 缺少引數 "x"，也沒有預設值
```

- You really have a function instead of numbers, with its argument taking the value you feed in.
- Define a multivariable function

```
fun2<-function(x, y) {1/(y+x)} #Define a function
fun2 #Gives you the definition of the function
```

```
## function(x, y) {1/(y+x)}
## <environment: 0x0000000026e07c10>
```

- Role of order of inputs:
  - The **order** of feeding in arguments matter **if** the exact name of argument is NOT specified. The first object provided will be treated as the first argument value, i.e., in `fun2(1,2)` we will see below that 1 is treated as `x` and so on.
- Order doesn't matter if the name of argument is also supplied, i.e., `fun2(x=1,y=2)` and `fun2(y=2,x=1)` are the same.

```
fun2(1,2) #Gives you a number by evaluating at x = 1 and y = 2
```

```
## [1] 0.3333333
```

```
fun2(x=1, y=2) #Same as above.
```

```
## [1] 0.3333333
```

```
fun2(y=2, x=1) #Same as above.
```

```
## [1] 0.3333333
```

- The following gives you an error message since `y` is not provided, and a default value fro `y` is not specified. You'll run into the same situation by having `fun2(y=2)`, `fun2(1)`, or `fun2()`.

```
fun2(x=1) #Gives you an error message since y is
```

```
## Error in fun2(x = 1): 缺少引數 "y"，也沒有預設值
```

- We can specify default values

```
fun2<-function(x = 1, y = 2) {1/(y+x)} #Define a function with default values provided.
fun2() #Gives you a number at x=1 and y=2
```

```
## [1] 0.3333333
```

```
fun2(3) #Gives you a number at x=3 and y=2
```

```
## [1] 0.2
```

```
fun2(x=3) #Same as above.
```

```
## [1] 0.2
```

```
fun2(y=5) #Gives you a number at y=5 and x=1
```

```
## [1] 0.1666667
```

- In R, functions take *global values*. In other softwares like *Matlab* you will need to declare an object to be global in the environment, and then you need to plug in a syntax in your function so that it *reads* the global parameter.

```
fun2<-function(x = 1, y = 2) {z/(y+x)} #Define a function with a parameter z.
fun2() #Gives you an error since z is NOT defined.
```

```
## Error in fun2(): 找不到物件 'z'
```

- No way to supply `z` via `fun2()` since it takes only `x` and `y`. But we can define `z` in the environment, and then the `fun2()` automatically takes it as a *global parameter*.

```
z<-3
fun2() #Evaluates at x=1, y=2 and z=3
```

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

```
fun2(x=3, y=3) #Evaluates at x=y=z=3
```

```
## [1] 0.5
```

```
z<-9 #Change z
fun2() #Evaluates at x=1, y=2, and z=9.
```

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

- The approach above are frequently used if we want to treat `z` as a parameter that is controlled from the global environment as it is not concerned by the underlying function. For example, the exponents α of a Cobb-Douglas utility function $U = x^{\alpha}y^{1-\alpha}$ can be coded in this way since we usually treat α to be fixed and focus on consumer's choices on x and y.

Generating Sequences of Identical Step Sizes

- `:` is probably the most intuitive way to get a sequence of numbers with step size 1.

```
c(1:5) #Gives you a sequence {1, 2, 3, 4, 5}
```

```
## [1] 1 2 3 4 5
```

```
c(2.718:6.3) #Gives you a sequence {2.718, 3.718, 4.718, 5.718}. 6.3 is omitted since 6.718 is above 6.3 already.
```

```
## [1] 2.718 3.718 4.718 5.718
```

- But what if we want sequences with *arbitrary* step sizes? Use `seq`.
- `seq(from, to, by, length.out)`, where
  - `from` and `to` are starting and ending values of the sequence.
  - `by` is the step size. See `length.out`.
  - `length.out` determines the length (number of elements) of the sequence. If specified, then `by` is calculated *uniformly* with `from`, `to`, and `length.out`

Well, easier shown then said

```
#Generate a sequence from 1 to 5. Step size is defaulted to 1.
seq(from=1, to=5)
```

```
## [1] 1 2 3 4 5
```

```
#Generate a sequence from 1 to 5 with step size 0.5.
seq(from=1, to=5, by=0.5)
```

```
## [1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

- Oftentimes all we need is `seq(from, to, by)` to generate whatever sequence we need. But one need to be cautious about the step size as incompatible elements will NOT be included in the result. Consider the following example:

```
#Generate a sequence from 1 to 3.6. But you will get {1, 2, 3} since step size is defaulted to 1.
seq(from=1, to=3.6)
```

```
## [1] 1 2 3
```

```
#Generate a sequence from 1 to 3.6 with step size 0.5. You'll see that 3.6 is missed and the last ele
ment is 3.5 since 3.5+0.5 = 4 is beyond 3.6
seq(from=1, to=3.6, by=0.5)
```

```
## [1] 1.0 1.5 2.0 2.5 3.0 3.5
```

- Using `by` can cause missing out of endpoints. In the previous example the endpoint 3.6 is missed out if we set `by=0.5`.
- Use `length.out` instead of `by` to prevent this issue, provided that `length.out >= 2`.

```
#Generate a sequence from 1 to 3.6, but with a vector of length 1 (1 element only)
#Gives you 1.
seq(from=1, to=3.6, length.out = 1)
```

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

```
#Do the same, but this time we need the result to contain 2 elements.
#Gives you {1, 3.6} so the step size is 2.6
seq(from=1, to=3.6, length.out = 2)
```

```
## [1] 1.0 3.6
```

```
#Do the same again, but we want 3 elements in the output.
#Step size is now (3.6-1)/2 = 1.3
seq(from=1, to=3.6, length.out = 3)
```

```
## [1] 1.0 2.3 3.6
```

- One can think of `length.out` as the number of elements / nodes on a number line, thus `length.out - 1` is exactly the number of segments / intervals.

Basic Rules for Numerical Integration

- Definite integral

$$I = \int_a^b g(x)dx$$

- For function $g(x)$, it can be re-expressed as $g(x) = p(x)f(x)$, where $p(x)$ can be think of as a **weighting function** to $f(x)$. Therefore a definite integral can be rewritten as

$$I = \int_a^b p(x)f(x)dx$$

  - A trivial case: $p(x) = 1$ hence $g(x) = f(x)$.
  - We can **choose** $p(x)$ to serve our needs. We will see this when it comes to Gaussian quadrature.
- Typically, *numerical* integration involves in choosing x_is over the domain, finding weightage ω_i for each x_i, and then evaluate by summing things up. Namely, we choose x_is and ω_is such that

$$I = \int_a^b p(x)f(x)dx \approx \sum_{i=1}^{n} \omega_i f(x_i)$$

- Sounds familiar? Think about how integrals are defined back in undergraduate calculus!
- Riemann Sum:

$$\int_b^a f(x)dx \equiv \lim_{\Delta x \to 0} \sum_{i=1}^{n} f(x_i)\Delta x$$

  In this case, $\omega_i = \Delta x$ for all i, i.e., equal weight / *step size* (although arbitrarily small).
- We will be introducing numerical approaches based on Riemann Sum, which you have learned back in undergraduate calculus if you still remember…
- Focus on a simple example: integrating $g(x) = \frac{1}{1+x}$ from 2 to 20. Specifically, we let $p(x) = 1$ hence $g(x) = f(x) = \frac{1}{1+x}$. The problem we want to solve is thus

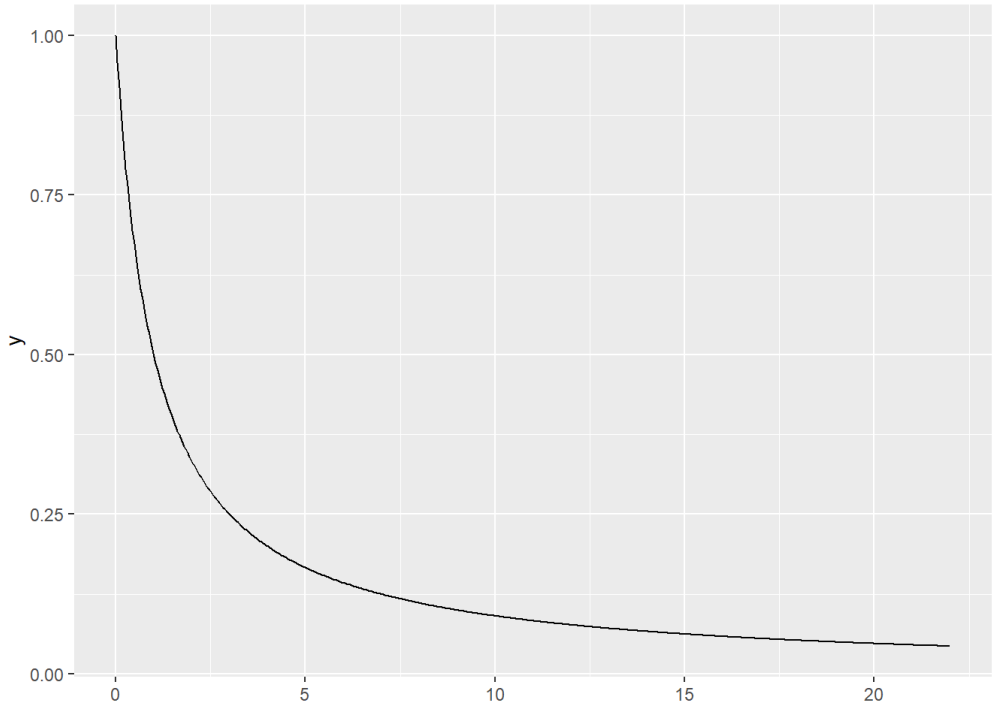
$$\int_2^{20} g(x)dx = \int_2^{20} f(x)dx = \int_2^{20} \frac{1}{1+x}dx$$

  Simple calculus shows that

$$\int_2^{20} \frac{1}{1+x}dx = \ln(7) \approx 1.9459$$

- Let's plot it out, and note down the *true value* when integrated on $[2, 20]$.

```
library(tidyverse)
#Simple Function to be integrated
intfun<-function(x) {1/(1+x)}
ggplot(data=data.frame(x=c(0,22)))+
  stat_function(fun=intfun, n=500)+xlim(c(0,22))
```




```
#Objective is to integrate on the interval (2,20)
#The true value:
TV<-log(7)
TV
```

```
## [1] 1.94591
```

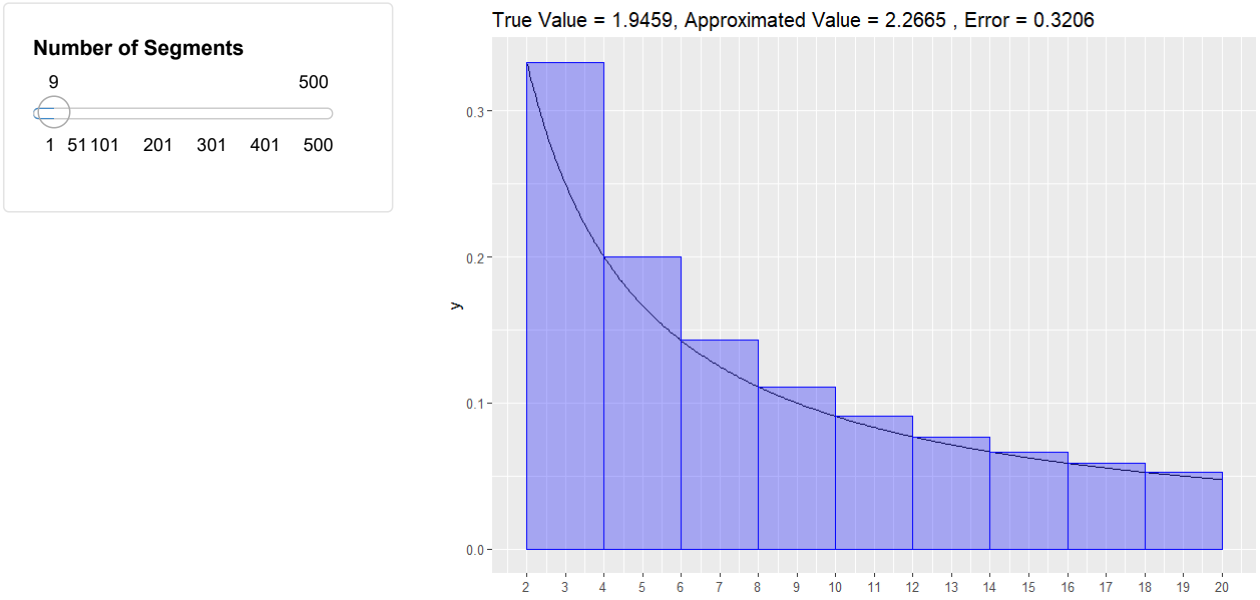
Forward Rectangular Rule

- Consider a definite integral $\int_a^b f(x)dx$ on the interval $[a,b]$ of length $b-a$.
- We can separate the interval into n segments hence $n+1$ points, where $x_1 = a$ and $x_{n+1} = b$.
- The *step size* is equal, such that $s \equiv x_{i+1} - x_i = (b-a)/n$.
- Forward Rectangular Rule says that we pick a sufficiently large n so that

$$\int_a^b f(x)dx \approx \frac{b-a}{n} \sum_{i=1}^{n} f(x_i)$$

- Think of it as $p(x) = 1$, and $\omega_i = (b-a)/n$ is equal for all i.
- Note that we only use points $\{x_1 = a, x_2 = a+s, x_3 = a+2s, \ldots, x_n = b-s\}$ to evaluate the function. The point $x_n = b$ is NOT used.

- Easier to see it visually with the simple function we considered above.



- Implementation with R. Note that we use `length.out` instead of `by` . Why so?

```
n.fr<-20 #number of nodes, needs to be integer greater than 2.
x.fr<-seq(2,20,length.out=n.fr) #nodes x_i
sp.fr<-x.fr[2]-x.fr[1] #Step
y.fr<-intfun(x.fr)[-n.fr] #Evaluate nodes. The last element is the endpoint and is not needed.
val.fr<-sum(sp.fr*y.fr) #Do Riemann Sum.
err.fr<-abs(val.fr-TV) #error. Note that `abs()` takes the absolute value.
val.fr
```

```
## [1] 2.08931
```

```
err.fr
```

```
## [1] 0.1433997
```

- I specify the number of nodes with `n.fr` and use `length.out` in `seq` to get exactly `n.fr` nodes of equal step sizes, hence `n.fr-1` intervals of equal length. Then I recover step size by taking first difference.
- Alternatively, one can also specify **the number of intervals**, say `ni` . Then compute the step size as `(20-2)/ni` . Then we get the nodes using `seq(from=2, to=20, by= (20-2)/ni)` . The rest of the coding are then exactly the same.
- The above is just a one time coding. We may want to wrap it into a **function** that can do definite integral for an **arbitrary equation** on an **arbitrary interval**. This is easily done as follows:

```
#Denote a function by `funct`, lower and upper bounds by `lb` and `ub`, and number of intervals `n``
myFwdRec<-function(funct, lb, ub, n, ...) { #use ... to pass other inputs needed by funct
  if (is.function(funct)==F) {
    stop("funct needs to be a function") #Check if funct is a function. If not, show error and quit.
  } else if (!is.finite(lb)||!is.finite(ub)) { #Note that need to use `||` instead of `|` for logical
purpose.
    stop("Infinite interval.") #Error and quit if boundaries are not finite.
  } else if (n<=0) {
    stop("Invalid interval number") #n must be strictly positive.
  } else {
    n<-as.integer(n) #Force n into an integer if a real number is provided.
    x<-seq(from=lb, to=ub, length.out=(n+1)) #Get nodes
    s<-x[2]-x[1] #Get step size.
    y<-funct(x, ...)[1:n] #The last node will not be used. Also pass other inputs needed.
    if (sum(!is.finite(y))!=0) { #If so, then funct(x) involves in Inf at some x, e.g., 1/x at x=0.
      stop("Singularity encountered.") #We cannot handle this case, so error and quit.
    } else {
      val<-sum(y*s) #Compute by Forward Rectangular Rule
      return(val) #Return the result
    }
  }
}

#Try a single input function
exfunc<-function(x) {5-x}
myFwdRec(funct=exfunc, lb=0, ub=6, n=100) #True value is 12.
```

```
## [1] 12.18
```

```
#Can also write functions in myFwdRec directly. Come in handy when funct is not too complicated.
myFwdRec(funct = function(x) {5-x}, lb=0, ub=6, n=100)
```

```
## [1] 12.18
```

```
#Try a function with multiple inputs
exfunc<-function(x,y,z) {5*y-x+z}
myFwdRec(funct=exfunc, lb=0, ub=6, n=100, y=1, z=2) #True value is 24.
```

```
## [1] 24.18
```

```
#Try a non-function object
t<-3
myFwdRec(funct=t, lb=0, ub=6, n=100)
```

```
## Error in myFwdRec(funct = t, lb = 0, ub = 6, n = 100): funct needs to be a function
```

```
#Try infinite boundary
myFwdRec(funct=function(x) {1+x}, lb=0, ub=Inf, n=100)
```

```
## Error in myFwdRec(funct = function(x) {: Infinite interval.
```

```
#Try invalid interval number
myFwdRec(funct=function(x){1+x}, lb=0, ub=3, n=-1)
```

```
## Error in myFwdRec(funct = function(x) {: Invalid interval number
```

```
#Try singularity, say 1/x at x=0
myFwdRec(funct=function(x){1/x}, lb=0, ub=1, n=100)
```

```
## Error in myFwdRec(funct = function(x) {: Singularity encountered.
```

- You should try to make your own customized function.
    - When doing large projects you will need to recycle certain code chunks over and over. If this happens, wrapping them into functions saves your time from finding your code chunks and doing copy-paste.
    - Functions are easier to manage. For large scale projects doing this and that, wrapping each part into functions make it a lot easier for debugging. When running into problems performing certain tasks, you just need to examine the functions for these tasks instead of going through a lot of lines.
    - Make a function as simple as possible. Ideally one function handles one specific task.
    - This kind of simple functions might have been implemented in R packages by big guys. But it is still a good practice for befriending with R and other programming languages.
    - Try to handle exceptions as possible (like those `if else` I do in the example above). It helps you figuring out problems and ensures reliability of your function.

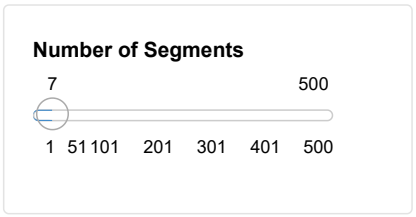
Midpoint Rule

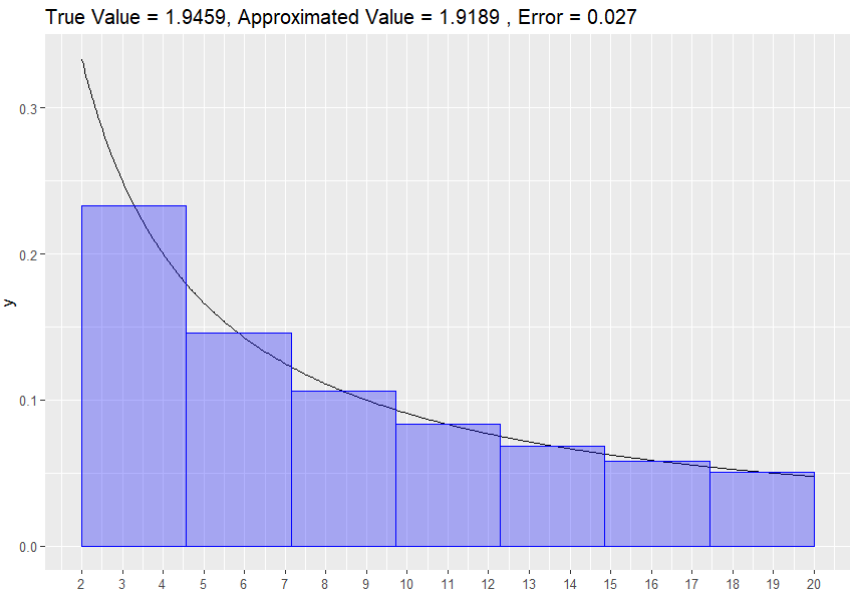
- Very similar to Forward Rectangular Rule, but this time midpoints of segments are used instead:

$$\int_a^b f(x)dx \approx \frac{b-a}{n}\sum_{i=1}^{n} f(\frac{x_i + x_{i+1}}{2})$$

- If we have n segments, then we have $n+1$ nodes. Now all of the nodes are used to compute the midpoints for each segments. Still the sum involves in only n terms.

Visualizing Midpoint Rule





In-class Exercise

Implement the Midpoint with R. Refer to the code for Forward Rectangular Rule. The key idea is that you need to figure out how to get all the n midpoints. If possible, wrap it into a function.

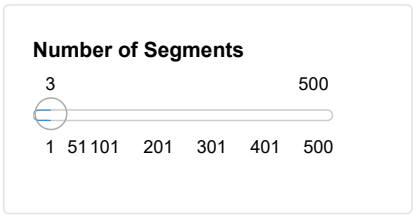
Trapezoidal Rule

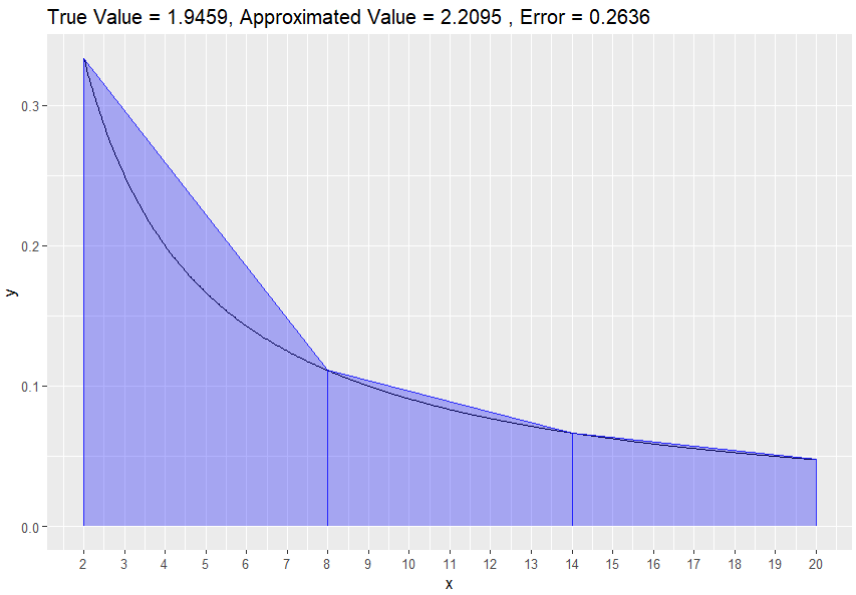
- An extension to the approaches above. We use trapezoids instead of rectangles:

$$\int_a^b f(x)dx \approx \frac{b-a}{n}\sum_{i=1}^{n}\frac{f(x_i)+f(x_{i+1})}{2}$$

- Now we have n segments and $n+1$ nodes. All of the nodes are used, and nodes except for the very first and the very last ones are used *twice*.

Visualizing Trapezoidal Rule





In-class Exercise

Implement the Trapezoidal with R. Refer to the code for Forward Rectangular Rule. The key is that you need to *shift* the evaluated function. For example, if you have 2 segments hence 3 nodes `x<-c(x1,x2,x3)`, then `y<-intfun(x)` gives you `c(y1,y2,y3)`. You need to compute `c(y1+y2,y2+y3)` for the trapezoidal regions, which can be done with vector indices `[]`.

Simpson's Rule

- In our example:
  - The Midpoint Rule is actually **more** accurate than the Trapezoidal Rule. Almost twice as accurate. Actually this can be proven by math when working with general functions.
  - The direction of error are opposite for both rules: Midpoint *underestimates* while Trapezoidal *overestimates*. It can be proven that even for a general function the directions of errors are opposite for the two rules.
- What if we combine both rules? Use Midpoint plus *some* Trapezoidal so that the *over-* and *underestimates* are balanced. This gives us the **Simpson's Rule**.
- Simpson's Rule states that

$$I=\int_a^b f(x)\,dx \approx \frac{b-a}{6n}\sum_{i=1}^{n}\left[f(x_i)+4f\left(\frac{x_i+x_{i+1}}{2}\right)+f(x_{i+1})\right]$$

  where n is the number of segments (hence we have $n+1$ nodes in total).

- With simple algebraic manipulations, the RHS is in fact

$$I = \frac{1}{3}\frac{b-a}{n}\sum_{i=1}^{n}\frac{f(x_i)+f(x_{i+1})}{2} + \frac{2}{3}\frac{b-a}{n}\sum_{i=1}^{n} f\left(\frac{x_i+x_{i+1}}{2}\right)$$

  which is exactly an *weighted average* between the Trapezoidal Rule and the Midpoint Rule where the weights are $\frac{1}{3}$ and $\frac{2}{3}$ respectively.

Implementation is easy once you know how to implement Midpoint and Trapezoidal Rules.

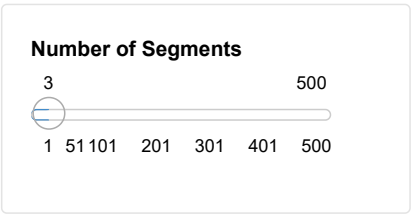
```
n.simp<-10 #number of nodes, needs to be integer greater than 2.
x.simp<-seq(2,20,length.out=n.simp) #nodes x_i
sp.simp<-x.simp[2]-x.simp[1] #Step

y.simp.tr<-intfun(x.simp) #Evaluate for the trapezoidal part
y.simp.mid<-intfun((x.simp+(sp.simp/2))) #Evaluate for the midpoint part

#1/3 of trapezoidal plus 2/3 of midpoint
val.simp<-(1/3)*(sum((y.simp.tr[-n.simp]+y.simp.tr[-1])*(sp.simp/2)))+
          (2/3)*(sum((y.simp.mid[-n.simp]*sp.simp)))
err.simp<-abs(val.simp-TV) #error
val.simp
```

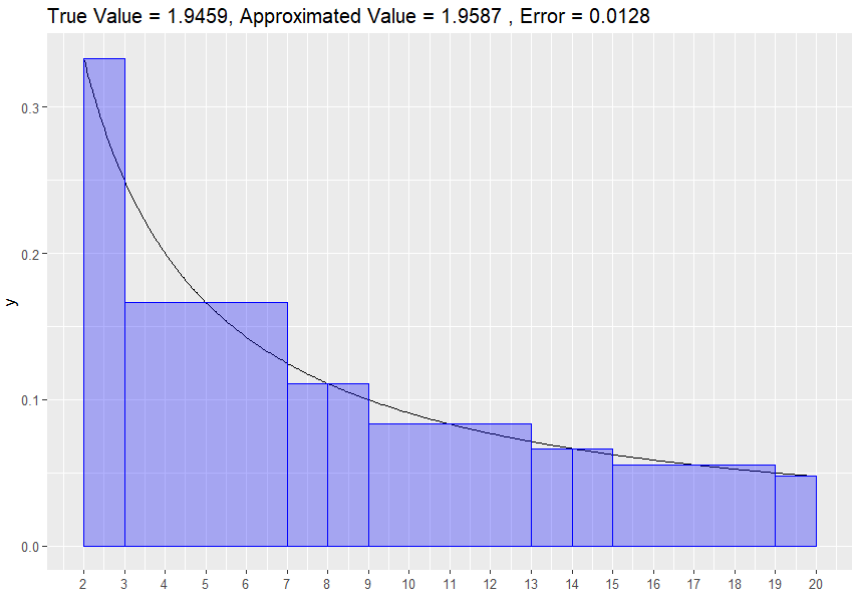
```
## [1] 1.946244
```

```
err.simp
```

```
## [1] 0.0003342785
```

Visualizing Simpson's Rule

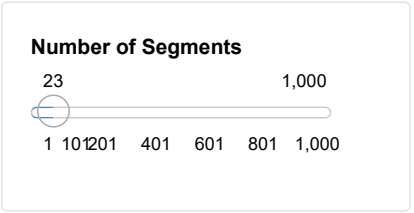


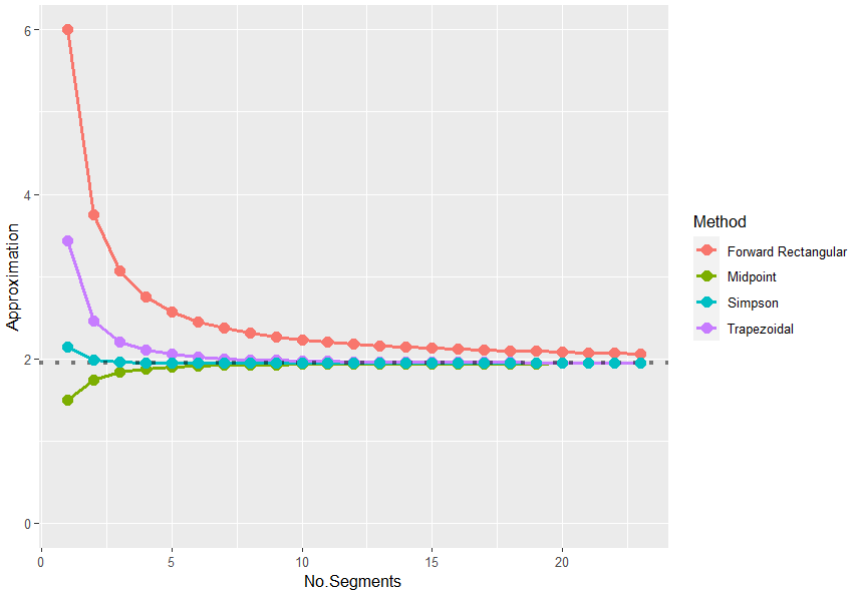


Performance Comparison… at least for the underlying example

- Number of segments / nodes determines the *accuracy* of numerical approximation.
- Fewer numbers needed is desired: you do less of computation hence saves time.
  - A big issue when the functions / models you face come at large scale and highly non-linear.

Visualizing the performances of the rules we have discussed so far. Simpson's Rule seems to be the best one, followed by Midpoint and Trapezoidal. Forward Rectangular is the least efficient one as it has difficulty to converge to 10^{-5} level even after 500 iterations.





- So how do we define **convergence**? Undergraduate calculus says

$$\lim_{n\to\infty} |f(x_n) - L| < \epsilon$$

- Literally, for any arbitrarily picked small number ϵ, there exists a number of iteration N, such that for all $n \geq N$ we always find the distance $|f(x_n) - L|$ to be less than ϵ. Namely, when we iterate for many many times the results should be almost similar.
- Here in our example, $L = \ln 7$ and n is the number of intervals for computing numerical integration. Note that more intervals means smaller step sizes. We call it a convergence if there exists an N such that using more intervals than N (hence using smaller step sizes) do not give us results too different from L.
- Practically, we usually let $\epsilon = 10^{-6}$ and find N following the criteria above.
- Implementation process:

  Step 0: Choose a tolerance level ϵ, and a tolerance number N_{tol}. Set up a *counter* as $cont = 0$. Choose an initial number of intervals n, and compute the difference between the true value L and the approximated value $L^{(n)}$ as $err^{(n)} \equiv |L^{(n)} - L|$.

  Step 1: Increase n by 1, and compute $err^{(n+1)} \equiv |L^{(n+1)} - L|$.

  Step 2: Check if $err^{(n+1)} < err^{(n)} < \epsilon$ holds. If yes, increase $cont$ by 1. If not, reset $cont$ to 0.

  Step 3: Repeat Steps 1 and 2, i.e., check $err^{(n+2)} < err^{(n+1)} < \epsilon$ and determine whether $cont$ is increased or reset to 0. Repeat until $cont = N_{tol}$.

- Literally speaking, we call it a **convergence** if the error becomes less than ϵ for a continued number of iterations N_{tol}. This is exactly the numerical counterpart of the definition of convergence you learned in undergraduate calculus. The N in the definition of convergence is exactly $n - N_{tol}$ where n is the number of iteration such that the above procedure stops at.

- What if the numerical integration does not converge for the function we work with? For example, think about integrating $1/x$ near $x = 0$. Then we need to setup a maximum number of iteration N_{max} so that we break the iteration if it does not converge after N_{max} times of iterations.
    - This idea is exactly the numerical counter part of **divergence**, i.e., $\lim_{n\to\infty} f(x) = \infty$.
- Demonstrate using the Forward Rectangular Rule because we have a function for it already. (See? No need to copy-paste. Just call it!)

```
tol<-10^(-4) #use 1e-4 as the tolerance level (what we called to be small).
#This is rather large, but I have spent a whole night trying to have it converge at 1e-6 level but to
no avail..... Anyway this is just for demonstration so we are fine.
err<-abs(myFwdRec(funct=intfun, lb=2, ub=20, n=1)-log(7)) #Define error size.
#Set number of interval `n=1` for illustration.

cont<-0 #counting for consecutively "small" (err less than tol in a row).
Ntol<-1000 #Criteria to count as consecutively small.

i<-1 #Initialize number of iteration to 1.
NStop<-10000 #Need to set up a maximum amount of iteration to avoid infinite loops.

#Keep going if the errors are not consecutively small in a row. See how we construct cont later.
#Also, the convergence of this rule is very slow. So we add 10 new intervals for each iteration.
#Namely, we have 10 intervals in the first iteration and 20 in the second iteration and so on.
while (cont<=Ntol) {
  #Convert number of iteration to interval numbers
  n<-10*i
    #Compute new error with a new iteration (with one more interval of course)
  err.new<-abs(myFwdRec(funct=intfun, lb=2, ub=20, n=n)-log(7))

  #Determine if consecutively small happens. Reset to 0 if becomes "large" during the process.
  if (err.new<tol&&err<tol) {
    cont<-cont+1
  } else {
      cont<-0
  }

  #Stop iteration if consecutively small happen in a row.
  if (cont==Ntol) {
    cat("Converged with err < tol for consecutively ",Ntol," times.")
    break
    }

  if(i==NStop) {
    cat("Do not converge to true value after ",NStop," times of iteration.")
    break
  }

  #Update error and number of iterations if the iteration continues.
  err<-err.new #Store current error as the "old" error for the next iteration.
  i<-i+1 #Update the number of iteration
}
```

```
## Converged with err < tol for consecutively  1000  times.
```

```
#n is the number of iteration, and the N we want is exactly n-Ntol.
if (cont==Ntol) {
  cat("Number of iterations = ",i,"; Converged when n = ",i-Ntol)
}
```

```
## Number of iterations =  3572 ; Converged when n =  2572
```

- In our example we know the true value by doing simple calculations. But in general we don't know about it (otherwise why bother doing numerical integration?).
- In practice, we use the notion of **Cauchy Sequence**
- **Cauchy Sequence:** Consider a sequence $\{x_n\}$. This sequence is a Cauchy Sequence if for all arbitrary small numbers $\epsilon > 0$ there exists an integer N such that $|x_m - x_n| < \epsilon$ for all $m > n \geq N$.
  - Literally speaking, the elements in the sequence are going to be similar to each other as the sequence goes.
- **Theorem:** On the whole real line, Cauchy Sequence implies Convergent Sequence **and vice versa.**
  - Convergent sequence is always Cauchy, but for Cauchy to imply convergence we need the metric space to be **complete**. Fortunately the space composed of the whole real line is complete so we have the theorem above. This is beyond the scope of the course. One should just go to Advanced Calculus or Real Analysis for details.
- Implementation:
  - In most of the situations we face, the outcome from a numerical integration is a real number. Then the iteration will be giving us a sequence composed of real numbers. Then the theorem above implies that we can inspect if $|L^{(n+1)} - L^n| < \epsilon$ holds for N_{tol} times starting from some n.

Assignment

- Choose one numerical integration methods out of Midpoint, Trapezoidal, and Simpson's Rule. Find the numbers needed for the rule you choose to converge by the **Cauchy Approach** for the problem

$$\int_0^{10} e^{-(x+e^{-x})} dx$$

  You might be able to findout the true value by hand. But **do pretend that you have absolutely no idea about the true value!**

Remarks

- The approaches we have introduced are special cases of the **Newton-Cotes Quadrature**.
- Newton-Cotes Quadrature work in the following manner:
  - Pick a series of nodes ξ_i on the interval $[a, b]$ such that the step sizes are **equal**, i.e., $|\xi_{i+1} - \xi_i|$ is a constant for all i.
  - Given the series of nodes ξ_i, pick **optimal weights** for each node ω_i.
  - The approximation is done by computing

$$\int_a^b f(x)dx \approx \sum_{i=1}^{n} \omega_i f(\xi_i)$$

- For Forward Rectangular, we have ξ_i to be the first n nodes if we break the interval into n intervals of equal distances. In Midpoint, ξ_i is the midpoints of each interval. The weight is simply the step size.

- For Trapezoidal, nodes are xi_i and weights are either full or half of the step size depending on whether the nodes are endpoints.
- For Simpson's, nodes and weights are generated from the roots of a **quadratic Lagrange interpolating polynomial**. This polynomial is designed to generate the best approximation to a curve with 3 nodes.

Gaussian Quadrature Rules

- **Gaussian Quadrature Rule** focuses on finding the integral of an function $g(x) = p(x)f(x)$ with a specific form for its weighting function $p(x)$ on an interval $[a, b]$ given the number of nodes n.
- Once n is determined, it generates the nodes and weights (x_i, ω_i) from a polynomial (depending on which of the Gaussian rule used). Hence the numerical integration is

$$\int_a^b g(x)dx = \int_a^b p(x)f(x)dx \approx \sum_{i=1}^{n} \omega_i f(x_i)$$

- These polynomials are designed such that n nodes can **exactly** fit the integrals of polynomials of degree $2n - 1$. Namely, Gauss Quadrature Rules compute the exact values of integration to general polynomial of degree $2n - 1$ using just n nodes.
- For non-polynomial integrands, Gauss Quadrature Rules **approximate** the integrals.
- Which (specifically-designed) polynomial is used to generate the weights and nodes? Depends on which rule we use, and is beyond our scope. Refer to the Wikipedia pages of the rules introduced in this course for more details…
- Why do we need Gaussian Quadrature? Because:

1. Deals with *improper integral*, e.g., integrating on unbounded interval such as $(0, \infty)$ and $(-\infty, \infty)$, integrand involving in singularity such as $1/x$ around 0.
    - Nodes **never** fall on the boundaries as opposed to rules previously introduced, so unbounded interval or singularities on boundaries are not an issue.
    - Nodes **might** fall on singular points if they are in the interior of the interval, e.g., integrating $1/x$ over $[-1,1]. But we can always use the singular points to divide the interval into subintervals, perform numerical integration on them and sum them up (**additivity of integration**).
2. Flexible: simple changes of variable allows us to do numerical integration to most, if not all, of the functions on various intervals.

- We cover 4 different Gaussian Quadrature Rules (and discuss 3 of them). Which rule to use? Depends on the type of interval $[a, b]$ and the form of weighting function $p(x)$:

Quadrature	$p(x)$	x interval
Gauss-Legendre	$p(x) = 1$	$[-1, 1]$
Gauss-Chebyshev	$p(x) = (1 - x^2)^{-\frac{1}{2}}$	$[-1, 1]$
Gauss-Hermite	$p(x) = e^{-(x^2)}$	$(-\infty, \infty)$
Gauss-Laguerre	$p(x) = e^{-x}$	$[0, \infty)$

- But what if the intervals and the form of $p(x)$ are not aligned with ANY of the rules above? Two solutions:

1. If $p(x)$ does not align, we can supply it by force.

- Example: We want to integrate $k(x)$ on $[0, \infty)$ using Gauss-Laguerre but we cannot decompose $k(x)$ into $k(x) = e^{-x}f(x)$. Then we can simply write

$$k(x) = e^{x}k(x)e^{-x} \equiv f(x)e^{-x}$$

and use Gauss-Laguerre to compute

$$\int_0^\infty k(x)dx = \int_0^\infty f(x)e^{-x}dx \approx \sum_{i=1}^n \omega_i f(x_i)$$

2. If the interval does not align, say trying to integrate on $[-3, 10]$ using Gauss-Legendre, we can do a *change of variable* to reformulate the problem such that the interval is aligned.

- *Change of Variable:* We want to integrate $f(x)$ on an interval $[a, b]$. Suppose that y and x takes a relationship by $x = h(y)$ where $h(y)$ is differentiable (at least to the first degree) and is invertible ($h^{-1}(x)$ exists) on the interval. Then we can rewrite our integration problem in terms of y by the following manner:

$$\int_a^b f(x)dx = \int_{h^{-1}(a)}^{h^{-1}(b)} f(h(y))h'(y)dy$$

  - Example 1: Attempting to integrate $f(x) = x^2$ on $[1, 4]$, and we have $x = e^{2y}$ hence $dx/dy = 2e^{2y}$ and $y = \ln(x)/2$. Then

$$\int_1^4 x^2dx = \int_{\frac{\ln(1)}{2}}^{\frac{\ln(4)}{2}} (e^{2y})^2 2e^{2y}dy = \int_0^{\ln(2)} 2e^{6y}dy$$

  - Example 2: Think about the problem we have been looking at $\int_2^{20} \frac{1}{1+x}dx$. We can do a change of variable to transform the interval into $[-1, 1]$. This can be done by fitting a linear function $y = ax + b$ that solves the following system

$$1 = 20a + b; \ \ -1 = 2a + b$$

The solutions to this system are $a = 1/9$ and $b = -11/9$. Hence, the linear transformation we go for is $y = (x - 11)/9$, $x = 9y + 11$, and $dx = 9dy$. We can thus restate the problem in terms of y such that the interval of integration is $[-1, 1]$.

$$\int_2^{20} \frac{1}{1+x}dx = \int_{-1}^1 \frac{1}{1 + (9y + 11)}9dy = \int_{-1}^1 \frac{3}{3y + 4}dy$$

  - Example 3: We want to integrate $g(x) = x^{-2}$ on $[1, \infty)$. We can do a change of variable so that the interval to be integrated becomes $[0, \infty)$. This is simply done by letting $y = x - 1$, hence $y \in [0, \infty)$ and $dy = dx$. Then the problem becomes

$$\int_1^\infty x^{-2}dx = \int_0^\infty (y + 1)^{-2}dy$$

  - Example 4: Continued from the previous example. We can also do a change of variable so that the interval of integration becomes $(0, 1]$. This is simply done by letting $y = x^{-1}$, hence $y \in (0, 1]$ and $dx = -y^{-2}dy$. Therefore,

$$\int_1^\infty x^{-2}dx = \int_0^1 y^2y^{-2}dy = \int_0^1 dy$$

- Example 5: Continued from Example 3, we can also convert the interval of integration as $[-1,1]$. We first follow the approach in Example 4 so that we integrate on $y \in (0,1]$. Then we let $z = 2y - 1$ so that $z \in (-1,1]$ and $dz = 2dy$. Then we have

$$\int_1^\infty x^{-2}dx = \int_0^1 dy = \int_{-1}^1 \frac{1}{2}dz$$

- Example 6: Consider the problem $\int_{-\infty}^0 x^2 dx$. We can convert the interval of integration to become $[0,\infty)$ by letting $y = -x$ hence $dy = -dx$. Then we have

$$\int_{-\infty}^0 x^2 dx = \int_0^\infty y^2 dy$$

- The transformation $x = h(y)$ are in general NOT unique. You might come up with a more complicated $h(y)$ but still convert the interval into the one you need. Nevertheless, no need to go for complication. Oftentimes a **linear transformation** is the most brainless way to serve our purpose. The following are frequently used linear transformations that convert an interval into $[-1,1]$:

x **interval**	$x = h(y)$	$dx = h'(y)dy$
$x \in [a,b]$	$x = \frac{a+b}{2} + \frac{b-a}{2}y$	$dx = \frac{b-a}{2}dy$
$x \in (-\infty,\infty)$	$x = \frac{y}{1-y^2}$	$dx = \frac{y^2+1}{(y^2-1)^2}dy$
$x \in [a,\infty)$	$x = a + \frac{1+y}{1-y}$	$dx = \frac{2}{(y-1)^2}dy$
$x \in (-\infty,b]$	$x = b - \frac{1-y}{1+y}$	$dx = \frac{2}{(1+y)^2}dy$

Frequently Used Gaussian Quadratures and Implementation on R

We will discuss Gauss-Legendre, Gauss-Hermite, and Gauss-Laguerre Rules here. These rules are implemented on R by the `pracma` package. There are many other quadrature rules in this package. See its documentation for more details.

```
library(pracma)
```

Gauss-Legendre

- Recall that Gauss-Legendre handles the problem of integrating $g(x) = p(x)f(x)$ with $p(x) = 1$ on $[-1,1]$.
- Once the number of nodes n is determined, Gauss-Legendre Quadrature determines nodes x_i and weights ω_i by utilising the Legendre Polynomial. See Wikipedia: https://en.wikipedia.org/wiki/Gauss%E2%80%93Legendre_quadrature (https://en.wikipedia.org/wiki/Gauss%E2%80%93Legendre_quadrature)
- Implemented on R by `gaussLegendre(n, a, b)`
  - `` `n `` is the number of nodes and must be an integer greater than 2.
  - `a` and `b` are lower and upper bounds of the integration. Both `a` and `b` must be **finte**. This design exempts us from doing change of variable manually if the original problem involves in only finite interval.
  - `gaussLegendre()` returns a named list with names `x` and `w`, each corresponds to nodes and weights respectively. Both are vectors of lengths `n`.
- Workflow:
  - Choose `n`, `a` and `b` and run `gaussLegendre()`.
  - Retrieve `x` and `w`.

- Plug `x` into the function of interest, multiply with `w` and then sum up the whole vector to get the numerical approximation.
- Since Gauss-Legendre asks for $p(x) = 1$, we can literally apply Gauss-Legendre for any non-linear functions as long as the interval of integration is finite after possible change of variables.

- Try the example $\int_2^{20} \frac{1}{1+x} dx$

```
#Generate the Gauss-Legendre Quadrature on the interval [2,20]. 10 nodes should be enough.
gl<-gaussLegendre(n=10, a=2, b=20)
gl #Inspect the returned nodes and weights.
```

```
## $x
##  [1]  2.234841  3.214430  4.885314  7.099441  9.660131 12.339869 14.900559
##  [8] 17.114686 18.785570 19.765159
##
## $w
##  [1] 0.6000421 1.3450621 1.9717773 2.4234005 2.6597180 2.6597180 2.4234005
##  [8] 1.9717773 1.3450621 0.6000421
```

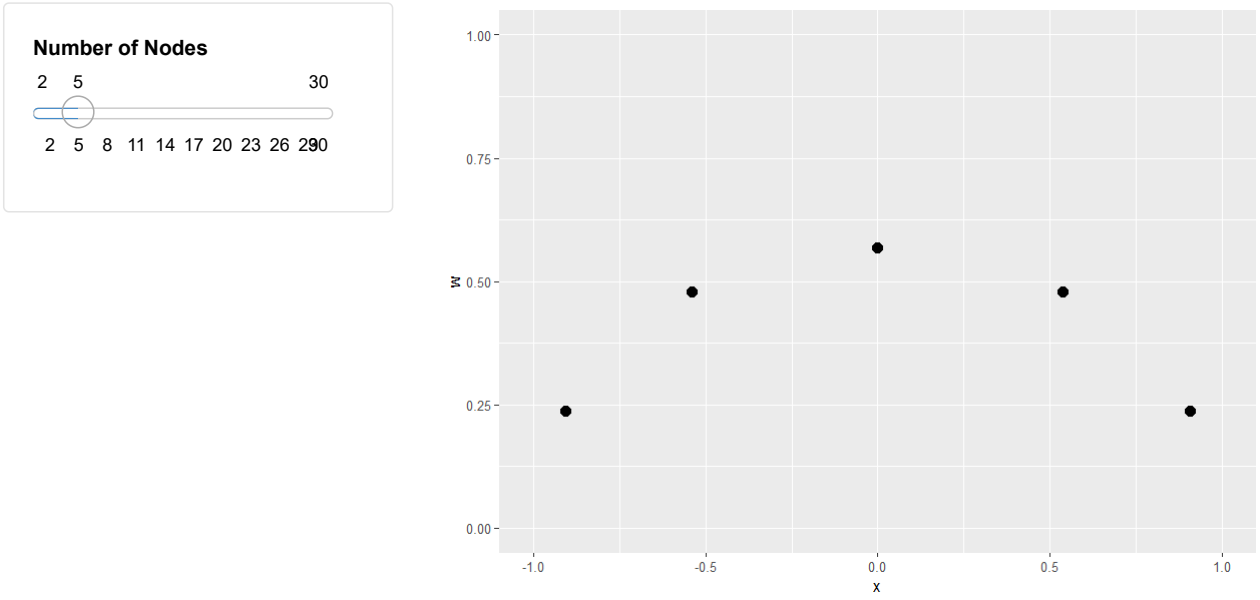
```
sum(intfun(gl$x)*gl$w) #Get an exact solution: ln(7)
```

```
## [1] 1.94591
```

```
#How about we use fewer nodes, say n=2?
gl<-gaussLegendre(n=2, a=2, b=20)
sum(intfun(gl$x)*gl$w) #Not very accurate now, only get 1.8462 < ln(7)
```

```
## [1] 1.846154
```

- How do the nodes and weights look like? It is something like “discrete uniform”, but the weights gradually decrease as the nodes get closer to the boundaries.
  - Tries to treat $f(x)$ at all nodes as equal as possible, but still $f(x)$ near the center becomes more important as we have more nodes.
- See it interactively using the default $[-1, 1]$ setting



Gauss-Hermite

- Gauss-Hermite deals with integrating $g(x) = p(x)f(x)$ with $p(x) = e^{-x^2}$ on $(-\infty, \infty)$.
- Once the nubmer of nodes are determined, it finds the nodes and weights with Hermite Polynomial. See https://en.wikipedia.org/wiki/Gauss%E2%80%93Hermite_quadrature (https://en.wikipedia.org/wiki/Gauss%E2%80%93Hermite_quadrature)
- Normal distribution is one of the cases that can be integrated with this approach after a minimal extent of change of variable

$$f(x) = f\left(x\right) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$

- Implemented on R by `gaussHermite(n)`.
  - `n` is number of nodes, and needs to be an integer greater than 2.
  - Returns a named list containing vectors of nodes `x` and weights `w` of length `n`.
- Workflow.
  - Manually decompose the function as $g(x) = e^{-x^2} f(x)$ first. If $g(x)$ by itself does not contain any parts similar to e^{-x^2}, then supply it artificially so that we have $g(x) = e^{-x^2} f(x)$ such that $f(x) = e^{x^2} g(x)$.
  - Choose `n` and call `gaussHermite()`.
  - Retrieve `x` and `w`.
  - Plug `x` into $f(x)$, multiply by `w` and then sum across them.
- Use Normal Expected Value as an example. Let $y = \frac{x-\mu}{\sqrt{2}\sigma}$ so we have $y \in (-\infty, \infty)$ and $dx = \sqrt{2}\sigma dy$. Then we can rewrite the expected value of normal:

$$\int_{-\infty}^{\infty} \frac{1}{\sqrt{2\pi}\sigma} e^{-\left(\frac{x-\mu}{\sqrt{2}\sigma}\right)^2} x dx = \int_{-\infty}^{\infty} \frac{\sqrt{2}\sigma}{\sqrt{2\pi}\sigma} e^{-y^2} \left(\sqrt{2}\sigma y + \mu\right) dy = \int_{-\infty}^{\infty} \frac{\sqrt{2}\sigma y + \mu}{\sqrt{\pi}} e^{-y^2} dy$$

  Obviously we have $p(y) = e^{-y^2}$ and $f(y) = \frac{\sqrt{2}\sigma y + \mu}{\sqrt{\pi}}$. Then we compute it with Gauss-Hermite

```
gh<-gaussHermite(n=10) #Try 10 nodes
gh #Inspect the weights and nodes.
```

```
## $x
##  [1] -3.4361591 -2.5327317 -1.7566836 -1.0366108 -0.3429013  0.3429013
##  [7]  1.0366108  1.7566836  2.5327317  3.4361591
##
## $w
##  [1] 7.640433e-06 1.343646e-03 3.387439e-02 2.401386e-01 6.108626e-01
##  [6] 6.108626e-01 2.401386e-01 3.387439e-02 1.343646e-03 7.640433e-06
```

```
#Define f(y) following our derivation as a function of y, m (mu), and s (sigma)
normal.mean<-function(y,m,s) {(sqrt(2)*s*y+m)/sqrt(pi)}

#Focus on the case that mu=2 and sigma=5
sum(normal.mean(gh$x,m=2,s=5)*gh$w) #Get the exact answer!
```

```
## [1] 2
```

- Let's try integrating the following function by force:

$$\int_{-\infty}^{\infty} \frac{1}{z^2+1} dz$$

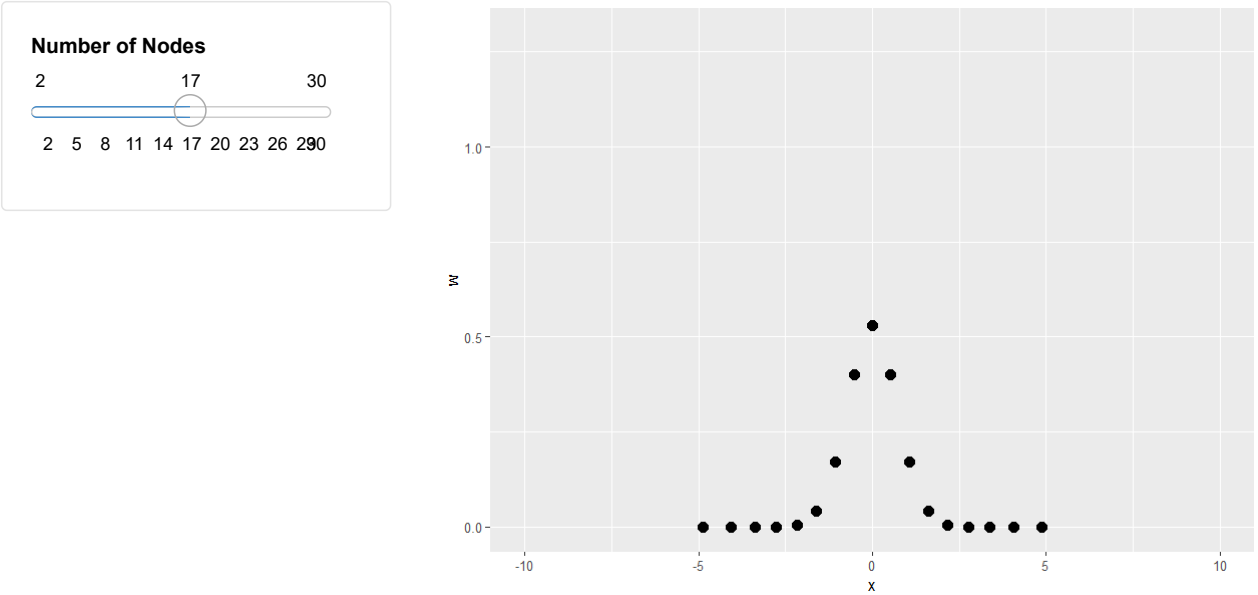
We can do it with Gauss-Hermite by artificially supplying $p(z) = e^{-z^2}$, hence

$$\int_{-\infty}^{\infty} e^{-z^2} \frac{1}{z^2+1} e^{z^2} dz = \int_{-\infty}^{\infty} e^{-z^2} f(z) dz$$

```
gh<-gaussHermite(n=350) #Try 350 nodes

#Define f(z) following our derivation
gh.intfun<-function(z) {exp(z^2)/(1+z^2)}

sum(gh.intfun(gh$x)*gh$w) #Not very good. The correct answer should be pi = 3.14159...
```

```
## [1] 2.923854
```

- How do the weights and nodes look like under Gauss-Hermite? It looks something like Normal distribution, with the center portion being most important. For the very far away tails, they literally receive 0 weights!



Gauss-Laguerre

- Gauss-Laguerre Rule integrates a function $g(x) = p(x)f(x)$ with $p(x) = e^{-x}$ on the interval $[0, \infty)$.
- Once the nubmer of nodes is picked, the nodes and weights are then generated with the Laguerre polynomial. See https://en.wikipedia.org/wiki/Gauss%E2%80%93Laguerre_quadrature (https://en.wikipedia.org/wiki/Gauss%E2%80%93Laguerre_quadrature)
- Implemented by `gaussLaguerre(n, a)`
  - It is a more general version such that $p(x) = x^a e^{-x}$. What we have been discussing is the case where $a = 0$.
  - `n` is the number of nodes, which must be an integer greater than 2.
  - `a` is the exponent for x^a in $p(x)$, and must be greater or equal to 0. We will stick with the case that `a=0` for consistency.
- Workflow.

- Manually decompose the function as $g(x) = e^{-x}f(x)$ first. If $g(x)$ by itself does not contain any parts similar to e^{-x}, then supply it artificially so that we have $g(x) = e^{-x}f(x)$ such that $f(x) = e^{x}g(x)$.
- Choose `n` and call `gaussLaguerre(n, a=0)`.
- Retrieve `x` and `w`.
- Plug `x` into $f(x)$, multiply by `w` and then sum across them.

- Try out finding the mean of standard Pareto:

$$\int_1^\infty kx^{-k}dx$$

where $k > 1$. Since Gauss-Laguerre integrates from 0 to infinity, we need to do a transformation by letting $y = x - 1$ and supplement $p(x)$ if possible. Therefore,

$$\int_1^\infty kx^{-k}dx = \int_0^\infty k(y+1)^{-k}dy = \int_0^\infty e^{-y}k(y+1)^{-k}e^{y}dy = \int_0^\infty e^{-y}f(y)dy$$

```
gla<-gaussLaguerre(n=10,a=0) #Try 10 nodes

#Define f(y) following our derivation as a function of y and k
gla.intfun<-function(y,k) {
  if(k<=1) {stop("k needs to be strictly greater than 1")} else {
    k*((y+1)^(-k))*exp(y)
  }
}

sum(gla.intfun(gla$x,k=2)*gla$w) #The correct answer is k/(k-1)
```

```
## [1] 1.945212
```

- Compute the mean of an exponential distribution:

$$\int_0^\infty txe^{-tx}dx$$

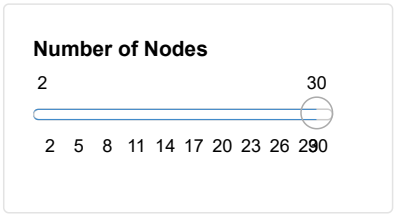
where $t > 0$. This is obvious: we simply let $y = tx$ hence $dy = tdx$ to restate the problem

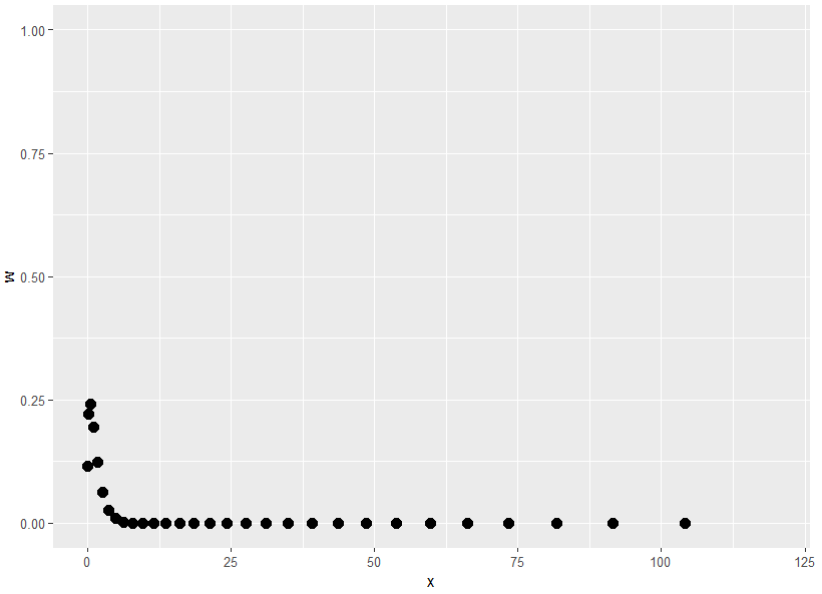
$$\int_0^\infty txe^{-tx}dx = \int_0^\infty \frac{y}{t}e^{-y}dy = \int_0^\infty e^{-y}f(y)dy$$

```
gla<-gaussLaguerre(n=10,a=0) #Try 10 nodes

#Define f(y) following our derivation as a function of y and t
gla.intfun<-function(y,t) {
  if(t<=0) {stop("t needs to be strictly positive")} else {
    y/t
  }
}

sum(gla.intfun(gla$x,t=2)*gla$w) #The correct answer is 1/t, which is exact!
```

```
## [1] 0.5
```

- The nodes and weights of Gauss-Laguerre looks similar to Exponential distribution, with the left tail being very important. The right tail too far literally receive 0 weights. Let's visualize it!





Assignment: Which Rule to Use?

- We have seen that there are various ways to perform transformation so that an integration problem can fit into multiple Gaussian Quadrature Rules. But obviously some rules are not very accurate / efficient in handling the problem of interest. Put it differently, some types of functions are particularly difficult to integrate with a given quadrature rule: being slow, inaccurate or not even converging.
- My hunch is that we should respect the designs of the rules. The rules are optimized against the form of $p(x)$, hence we should just go for those rules if our problem is already aligned with $p(x)$. That is, we should avoid supplying $p(x)$ artificially. Then we ask if we need to do change of variable so that the intervals are aligned. In short, we go for the rule that can be applied with minimal twists to our integration problem.
- For your assignment, consider again the problem

$$\int_{-\infty}^{\infty} \frac{1}{z^2+1} dz$$

The correct answer is π. We have tried to use Gauss-Hermite for it, but the answer is always somewhere around 2.99 despite that we have used 350 nodes! Using more nodes doesn't seem to improve the results much, and can even lead to missing solutions if we add even more nodes. Your missions are as follows:

1. Convert this problem into a finite interval and do it with Gauss-Legendre.
2. Use the additivity of integral to reformulate the problem as

$$\int_{0}^{\infty} \frac{1}{z^2+1} dz + \int_{-\infty}^{0} \frac{1}{z^2+1} dz$$

and then do it with Gauss-Laguerre.

3. How many nodes are needed for the solutions to be close to π at $\epsilon = 10^{-6}$? In the cases that R fails to evaluate for nodes too many, find how close to π the results are.

Further Discussions

- `pracma` offers a lot more rules (Gaussian or not) to perform numerical integration such as Gauss-Chebyshev, Clenshaw-Curtis, and Gauss-Konrod. You can explore around with its documentation.

Adaptive Quadratures

- A refinement to improve the convergence of numerical integration, especially for functions that are difficult to integrate, e.g., functions defined on (0,1) but with vertical asymptotes / singularities at both endpoints.
- General work flow:

  Step 1. Do numerical integration on the interval of interest $[a, b]$.

  Step 2. Check the approximation error.

  Step 3. If the error is higher than the tolerance level ϵ, then divide the interval $[a, b]$ into two subintervals $[a, c]$ and $[c, b]$. Otherwise stop.

  Step 4. For each subinterval, repeat Steps 1 to 3 until all the numerical integration are converged (error less than ϵ) in all subintervals.

  Step 5. Sum up the results and report.
- For Gaussian Quadratures, the errors are in practice measured by the difference between results from n Gaussian nodes and n Gaussian plus $2n + 1$ Konrod nodes.
- Or one can also use the Cauchy Sequence notion by defining the error as the difference of results from n and $n + 1$ nodes.
- In `pracma`, adaptive quadrature is (automatically) implemented by `integral()`. Three rules are provided: Gauss-Konrod, Simpson, and Clenshaw-Curtis. But note Clenshaw-Curtis in this function is not yet made adaptive.

High-dimension Integration and Curse of Dimension

- We have been discussing numerical integration on one-dimension. How about we have two or more dimensions? Say,

$$\int_{l_y}^{u_y} \int_{l_x}^{u_x} f(x, y) dx dy$$

- The quadrature rules still hold here:

$$\int_{l_y}^{u_y} \int_{l_x}^{u_x} f(x, y) dx dy \approx \sum_{i=1}^{n_y} \sum_{j=1}^{n_x} \omega_i \omega_j f(x_j, y_i)$$

  where nodes and weights are obtained from the corresponding quadrature rules.
  - The `pracma` package implements a two-dimensional integration by `integral2()`, and a 3-d integration by `integral3()`.
  - The `mvQuad` package also implements multi-dimensional numerical integration for the quadrature we have discussed. But personally I don't like it as the functions are full of black-boxes, plus its definitions to quadratures seem to be very different from convention.
- But how many nodes are needed? Using 2 nodes for each dimension will lead to 4 nodes in total. Using 10 nodes for each dimension then lead to 100 nodes in total! This is so called **curse of dimensionality**: we need more nodes for accuracy, but the nodes grow in **exponential** manner as we have more dimensions! It's going to be slow and eventually become inefficient / infeasible.