# Numerical Integration: Monte Carlo Approach

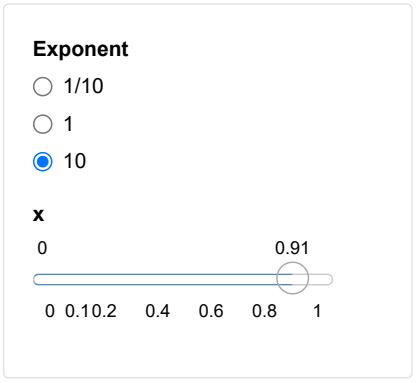
Ivan Yi-Fan Chen

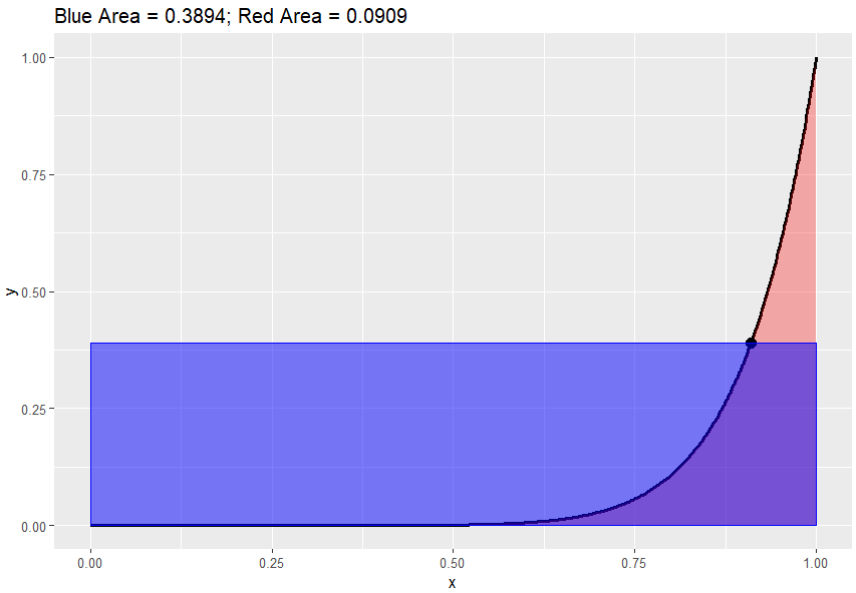
2023 Fall

## Monte Carlo Integration

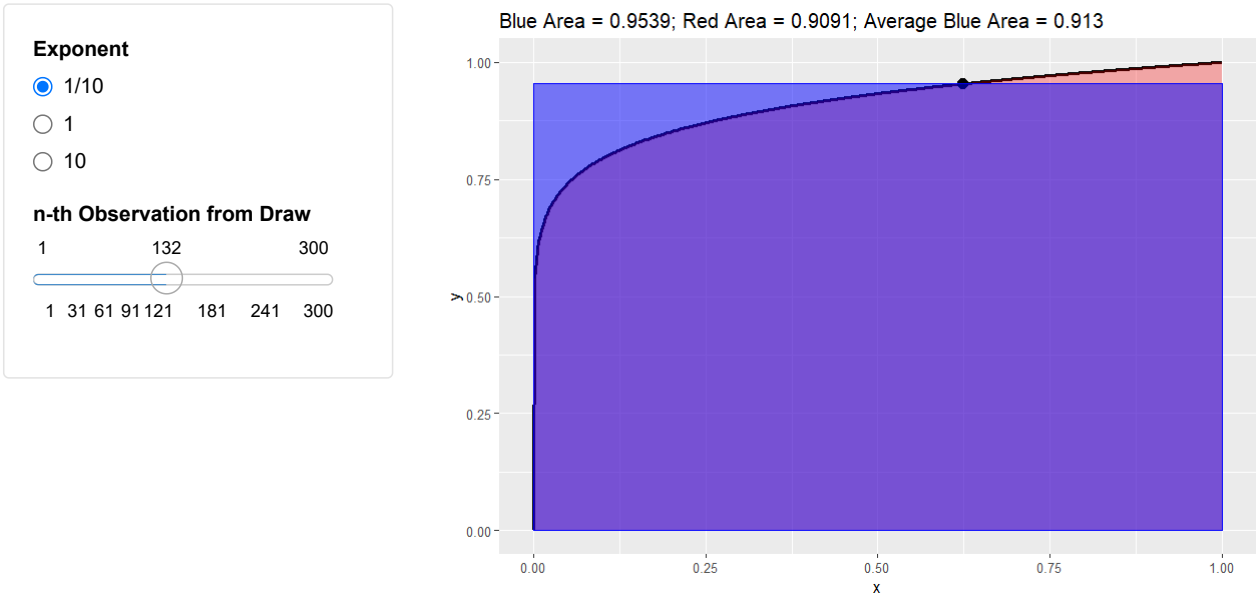
### Basics of Monte Carlo Integration

- Integration is an estimation in the sense that we use the areas of rectangles to approximate the area under a curve.
- What if we use just one rectangle to represent the area under the curve? Obviously a long shot, but this can be a possible way.
- Consider a power function $f(x) = x^k$ that maps $x \in [0, 1]$ to $y \in [0, 1]$, and we pick $x$ to construct a rectangle of area as $A(x) = f(x) \times 1 = x^k$.




- Observations:
  - For $y = x^{\frac{1}{10}}$ we find that $A(x)$ is oftentimes quite large even when $x$ is close to 0.
  - For $y = x^{10}$ we find that $A(x)$ is very close to 0 for $X < 0.5$.
  - For $y = x$, $A(x)$ is the same as $x$.
- Idea: How about we draw a sequence of $x$ from $x \in [0, 1]$ uniformly, compute $A(x)$ and then take average? It should be very similar to the area under $y = x^k$, i.e., $I = \int_0^1 x^k dx$.
  - For $y = x$, it is easy to see that the true value $I = 0.5$. Since $A(x) = x$, a uniform draw on $x \in [0, 1]$ leads to $E(A(x)) = 0.5$.
  - For $y = x^{\frac{1}{10}}$, we almost always get $A(x) \in (0.5, 1)$ even for small $x$. Therefore $E(A(x)) \in (0.5, 1)$ and is consistent with the fact that $I \in (0.5, 1)$.
  - For $y = x^10$ we see that $A(x)$ is very close to 0 for $x < 0.5$, meaning that $E(A(x)) < 0.5$ should hold. This is also consistent with that $I < 0.5$.
- Try out the idea interactively: We draw 300 observations from $x \in [0, 1]$. For each $x$ we compute $A(x)$ and reports it as the value of the "Blue Area". The average is computed as $\sum_i A(x_i)/300$. The true area under $y = x^k$ is obtained by $\int_0^1 x^k dx$ and is reported as the value of the "Red Area".




- The above exercise gives us a conclusion: For large enough $N$, we have

$$I = \int_0^1 x^k dx \approx \sum_{i=1}^{N} \frac{x_i^k}{N}$$

The idea behind is so-called **Monte Carlo Integration**.

- **Monte Carlo Integration:** For the integration problem of $f(x)$ on an interval $[a, b]$, we can reformulate the problem to be proportional to the expected value of $f(x)$ with respect to an uniform distribution $U[a, b]$

$$I = \int_a^b f(x)dx = (b-a)\int_a^b f(x)\frac{1}{b-a}dx = (b-a)E(f(x))$$

Therefore, we can approximate the integral $I$ by

$$I \approx (b-a)\sum_{i=1}^{N}\frac{f(x_i)}{N}$$

with a large enough sample $x_i$ drawn from $U[a, b]$.

  - The previous example is a special case that $b = 1$ and $a = 0$ so $b - a = 1$.

- Why can we do this? The underlying mechanism is **Law of Large Numbers**:

  - **Weak Law of Large Numbers:** The sequence $\{\sum_{i=1}^{n}\frac{f(x_i)}{n}\}$ converges to $E(f(x))$ **in probability**:

$$\lim_{n\to\infty}\Pr\left(\left|\sum_{i=1}^{n}\frac{f(x_i)}{n} - E(f(x))\right| < \epsilon\right) = 1$$

That is, for all $\epsilon > 0$ and all $\delta > 0$ there exists an $N$ such that for all $n > N$ we have

$$\left|\Pr\left(\left|\sum_{i=1}^{n}\frac{f(x_i)}{n} - E(f(x))\right| < \epsilon\right) - 1\right| < \delta$$

In plain words, as long as we have a large enough sample size the event that $\sum_{i=1}^{n}\frac{f(x_i)}{n}$ being very close to $E(f(x))$ is *very likely* to happen in the sense that the probability is arbitrarily close to 1. ***BUT*** it is NOT of probability 1, meaning that this event **might** not happen: $\sum_{i=1}^{n}\frac{f(x_i)}{n}$ can drift away from $E(f(x))$ even when $n$ is large… *possible but unlikely* (a quote from the film *The Exorcist*).

- **Strong Law of Large Numbers:** The sequence $\{\sum_{i=1}^{n} \frac{f(x_i)}{n}\}$ converges to $E(f(x))$ **almost surely** / **with probability 1**:

$$\Pr\left(\lim_{n\to\infty} \sum_{i=1}^{n} \frac{f(x_i)}{n} = E(f(x))\right) = 1$$

That is, for all $\epsilon > 0$ there exists an $N$ such that for all $n > N$ we have

$$\Pr\left(\left|\sum_{i=1}^{n} \frac{f(x_i)}{n} - E(f(x))\right| < \epsilon\right) = 1$$

In plain words, as long as we have a large enough sample size we **always** see that $\sum_{i=1}^{n} \frac{f(x_i)}{n}$ being very close to $E(f(x))$. There is no uncertainty about this event: if it gets close, then it gets close forever. This means that $\sum_{i=1}^{n} \frac{f(x_i)}{n}$ can be effectively treated as the constant $E(f(x))$.

- These Laws of Large Numbers do not come for free. There are restrictions on the mean of the sample, depending on whether the sample is generated independently and identically. But this is beyond the scope of this course already. You may refer to the classical book by Halbert White if you are such a masochist.

- Still informative to introduce **Kolmogorov's Law of Large Number**: as long as $x_i$ is drawn **independently** from an **identical distribution**, then $\sum_{i=1}^{n} \frac{f(x_i)}{n}$ converges to $E(f(x))$ **with probability 1**. This is almost always true in this course since we always do random draw from a given distribution.

- ***Monte Carlo Integration always converges as a result of Strong Law of Large Number (Kolmogorov, in our case).***

### In-Class Exercise

Perform Monte Carlo Integration for $\int_0^1 x^3 dx$ with a random vector of length 500 drawn from $U[0,1]$.

## Extension to Infinite Intervals

- What if the interval of integration is large, e.g., $x \in (-1000, 1000)$? Obviously drawing from $U[-1000, 1000]$ is going to take a huge load of draws to be representative, and is time consuming!

- Even worse, what if the interval is not finite, say $x \in (0, \infty)$?

- **Change of Variable** helps us to suppress the interval of integration into $[0,1]$.

- Recall that we need to find some differentiable $y = h(x)$ such that $x \in [a,b]$ becomes $y \in [0,1]$ and $dy = h'(x)dx$. Preferably we want $h(x)$ to be **linear**. Fortunately there are ready-to-use results:

  - For $x \in [a,b]$, we let $x = a + (b-a)y$ so $dx = (b-a)dy$ and $y \in [0,1]$.
  - For $x \in (-\infty, \infty)$, we let $x = \frac{2y-1}{y-y^2}$ so $dx = \frac{2y^2-2y+1}{(y^2-y)^2}dy$ and $y \in [0,1]$.
  - For $x \in [a, \infty)$, we let $x = a + \frac{y}{1-y}$ so $dx = \frac{1}{(y-1)^2}dy$ and $y \in [0,1]$.
  - For $x \in (-\infty, b]$, we let $x = b + \frac{y-1}{y}$ so $dx = \frac{1}{y^2}dy$ and $y \in [0,1]$.

- Example:

$$\int_{-\infty}^{\infty} \sqrt{1+x^2} e^{-\frac{x^2}{3}}\, dx$$

There are two approaches for this question: convert into $y \in U[0,1]$ and perform Monte Carlo integration, or simply let $y \equiv x/\sqrt{3}$ and then perform Gauss-Hermite Quadrature. We will try both so that we know if we are doing it right for the Monte Carlo approach:

1. Gauss-Hermite: we let $y \equiv x/\sqrt{3}$ thus $dy \equiv 1/\sqrt{3}dx$, then

$$\int_{-\infty}^{\infty} \sqrt{1+x^2} e^{-\frac{x^2}{3}} dx = \int_{-\infty}^{\infty} e^{-y^2} \sqrt{3} \sqrt{1+3y^2} dy$$

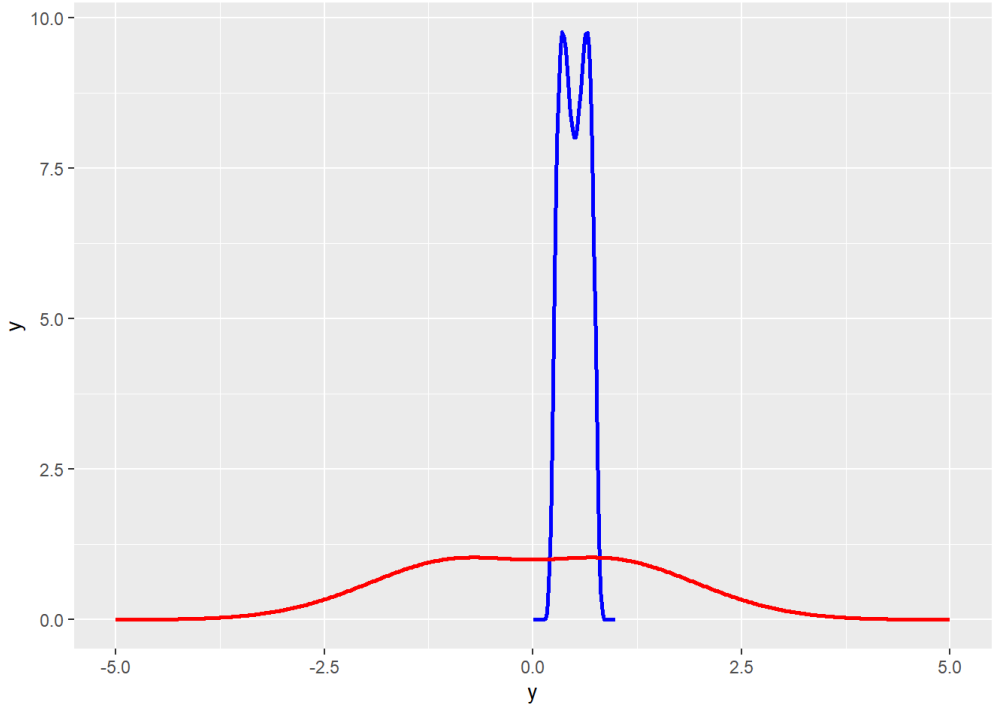
2. Monte Carlo: we perform a change of variable by letting $x = \frac{2y-1}{y-y^2}$, hence $dx = \frac{2y^2-2y+1}{(y^2-y)^2} dy$ and $y \in [0,1]$. Then the problem is restated as follows:

$$\int_{-\infty}^{\infty} \sqrt{1+x^2} e^{-\frac{x^2}{3}} dx = \int_0^1 \sqrt{1+(\frac{2y-1}{y-y^2})^2} e^{-\frac{(\frac{2y-1}{y-y^2})^2}{3}} \frac{2y^2-2y+1}{(y^2-y)^2} dy$$

Now we are ready to implement it.

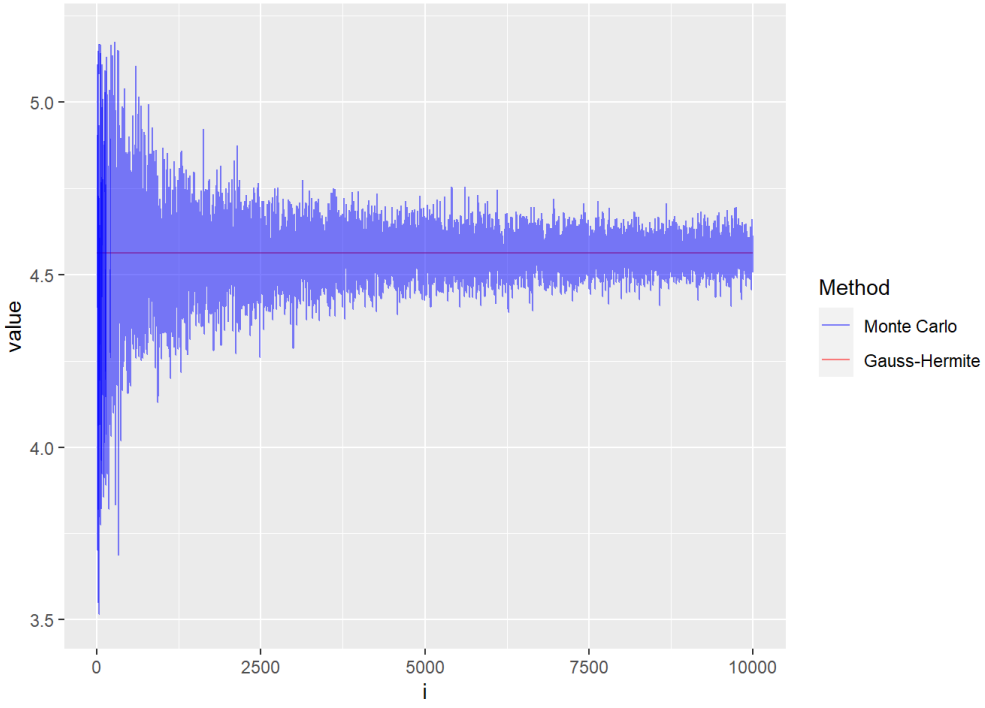
```
#Monte Carlo v.s. Gauss-Hermite
set.seed(65536)
intfun.og<-function(x) {sqrt(1+(x^2))*exp(-((x^2)/3))}
cv<-function(y) {((2*y-1)/(y-(y^2)))}
jc<-function(y) {(2*(y^2)-2*y+1)/(((y^2)-y)^2)}

intfun.mc<-function(y) {
  y[y<=0]<-NA #This is to avoid ggplot from plotting it beyond [0,1].
  y[y>=1]<-NA #This is to avoid ggplot from plotting it beyond [0,1].
  rs<-intfun.og(x=cv(y))*jc(y=y)
  return(rs)
}

ggplot(data=data.frame(y=c(-5,5)),aes(y))+
      stat_function(fun=intfun.mc,n=500,size=1,color="blue")+
      stat_function(data=data.frame(x=c(-5,5)),aes(x),fun=intfun.og,n=500,size=1,color="red")
```

```
## Warning: Removed 450 row(s) containing missing values (geom_path).
```



```
#Monte Carlo from 1 to 10000 times.
val.mc<-c()
for (i in 1:10000) {
  y<-runif(i,0,1)
  tmp<-sum(intfun.mc(y))/i
  val.mc<-c(val.mc,tmp)
}

#Gauss-Hermite with 30 nodes.
intfun.gh<-function(x) {sqrt(3)*sqrt(1+3*(x^2))}
gh<-gaussHermite(n=30)
val.gh<-sum(intfun.gh(gh$x)*gh$w)

#Plot out the results for each iteration and compare with Gauss-Hermite
df<-data.frame(i=rep(1:10000,2),
               value=c(val.mc,rep(val.gh,10000)),
               Method=c(rep("Monte Carlo",10000),rep("Gauss-Hermite",10000)))

#Looks difficult to converge!
ggplot(data=df,aes(x=i,y=value,color=Method))+
  geom_line(alpha=0.5)+
  scale_color_manual(values=c("blue","red"),
                     breaks=c("Monte Carlo","Gauss-Hermite"))+
  ylim(c(3.5,5.2))
```

```
## Warning: Removed 2 row(s) containing missing values (geom_path).
```



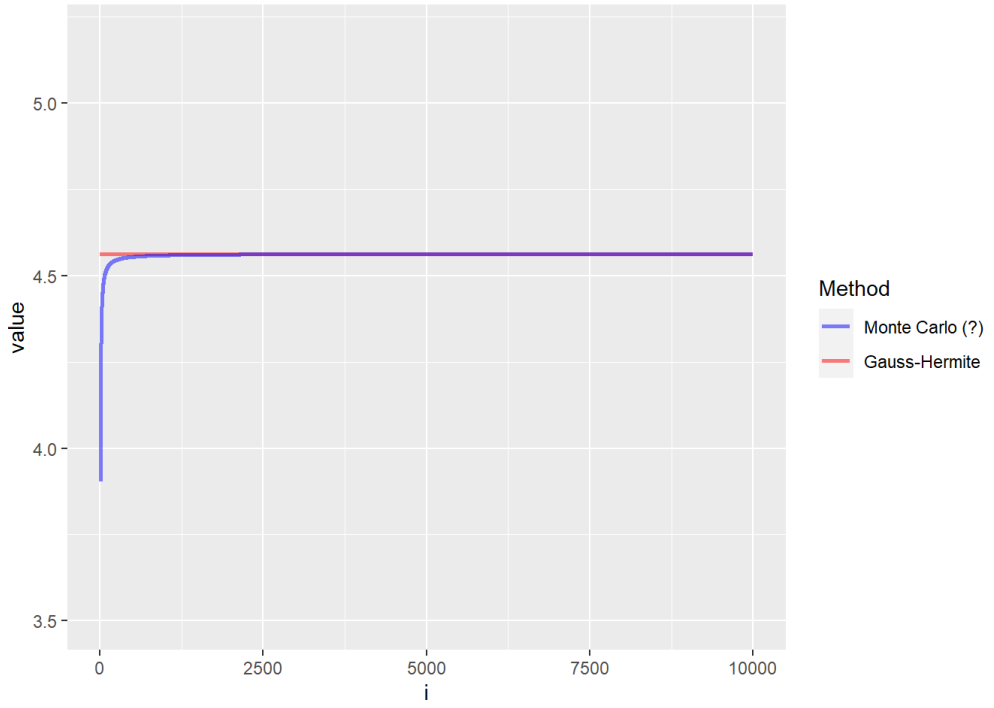

- We just have learned that Monte Carlo integration always converge. But the result above is so disappointing! The convergence looks so slow and the error is so large. This motivates the Quasi Monte Carlo Integration with Low Discrepancy Sequences.

# Quasi Monte Carlo Integration

- A sequence drawn randomly is a realization, hence it doesn't need to be "evenly" distributed. For example, drawing a sequence of length 30 from $0, 1$ is perfectly possible to get a realization with 20 or more realizations being 0. Similarly, drawing from $U[0, 1]$ doesn't imply that the realized sequence is "uniform" enough.
- In other words, we frequently see that the points draw randomly cluster together while large spaces on the interested domain are left out. We can refine this problem by drawing even more points, but it is hard to say about how many should be drawn.
- If we are able to convert a 1-D integration problem into a $[0, 1]$ interval, then how about we divide this interval into many evenly spaced nodes, and then use these nodes as if they are drawn from $U[0, 1]$ for a Monte-Carlo-like integration? Let's try the previous example.

```
#Monte Carlo (?) v.s. Gauss-Hermite
intfun.og<-function(x) {sqrt(1+(x^2))*exp(-((x^2)/3))}
cv<-function(y) {((2*y-1)/(y-(y^2)))}
jc<-function(y) {(2*(y^2)-2*y+1)/(((y^2)-y)^2)}

intfun.mc<-function(y) {
  rs<-intfun.og(x=cv(y))*jc(y=y)
  return(rs)
}

#Monte Carlo (?) with evenly spaced sequence, from length 1 to 10000.
val.mc<-c()
for (i in 1:10000) {
  y<-seq(0,1,length.out=i)
  tmp<-sum(intfun.mc(y),na.rm=T)/i #Need to use na.rm since this function is undefined at 0 and 1.
  val.mc<-c(val.mc,tmp)
}

#Gauss-Hermite with 30 nodes.
intfun.gh<-function(x) {sqrt(3)*sqrt(1+3*(x^2))}
gh<-gaussHermite(n=30)
val.gh<-sum(intfun.gh(gh$x)*gh$w)

#Plot out the results for each iteration and compare with Gauss-Hermite
df<-data.frame(i=rep(1:10000,2),
               value=c(val.mc,rep(val.gh,10000)),
               Method=c(rep("Monte Carlo (?)",10000),rep("Gauss-Hermite",10000)))

#Looks visually perfect to call it a convergence.
ggplot(data=df,aes(x=i,y=value,color=Method))+
  geom_line(alpha=0.5,size=1)+
  scale_color_manual(values=c("blue","red"),
                     breaks=c("Monte Carlo (?)","Gauss-Hermite"))+
  ylim(c(3.5,5.2))
```

```
## Warning: Removed 3 row(s) containing missing values (geom_path).
```




- The approach above is what we referred to as the **grid approach**. We simply set up a series of nodes on the interval (a **grid**), and then compute by force. This approach is also used for optimization.
- The integration performed here is similar to Monte Carlo in the sense that we grab nodes on the interval and compute the average of the corresponding areas. Yet *no randomnesses are involved*. This is what we refer to as the **Quasi Monte Carlo Integration**.

## Low Discrepancy Sequence

- The grid approach is a reasonable choice for that the nodes spread evenly on the interval. But there are two drawbacks:
    1. Inflexible if you need to add more nodes. Suppose that you divide the interval into 100 nodes with equal step size. Now you feel that you need more, say 150 nodes to improve the accuracy. Then you need to regenerate a sequence of 150 nodes. You CANNOT just "insert" additional 50 nodes into your existing nodes while still keeping the same distances between nodes.
    2. Pairing of nodes is also a problem. Suppose that you have $x$ and $y$ dimensions, on each dimension you want to have 10 nodes. Then you need to construct a grid of $10 \times 10 = 100$ points. You will need to do the pairing on your own, i.e., $(x_1, y_1)$,$(x_1, y_2)$,...,$(x_1, y_1 00)$,$(x_2, y_1)$,$(x_2, y_2)$,... This can be time consuming and taking up computational resources.
- **Low Discrepancy Sequence (LDS)** can be used to replace equidistant grids.
    - It is *deterministic*. No randomnesses are involved.
    - It fills an $[0, 1]^D$ space with points of similar distances by construction. The sequence thus fills the space as evenly as possible.
    - It is **gap-filling**: by construction an LDS is such that it inserts new nodes into existing gaps gradually.
    - "Additive": This is a result of the gap-filling property. For an LDS of length $n$, its subsequence of length $m$ is also an LDS. This means that, for example, if we divide the interval with an LDS of 100 nodes and then we want to add additional 50 nodes, it is perfectly fine to generate another 50 nodes from an LDS generator and lump with the existing node. We still have an LDS.
    - Easily paired. If we need 100 points in total for a 2-D problem, then for each dimension we generate 100 nodes from LDS' of different bases. Simply "binding" the two sequences into a matrix completes the pairing of points. No need to

undergo complicated matrix manipulations or multi-layer loops like when using grids.

- Frequently used LDS' include the **Halton Sequence** and the **Sobol Sequence**. We will cover only the former.
- **Halton Sequence:** A Halton sequence is constructed using a prime number as its base. The idea is to choose a base number $b$. Then first divide the unit interval by $1/b$, and then by $1/b^2$, and then by $1/b^3$ and so on. The sequences are ordered in the way that the elements of subsequence of exponent $k$ fill the gaps left by the elements of the subsequence of exponent $k-1$.

  See Wikipedia: https://en.wikipedia.org/wiki/Halton_sequence (https://en.wikipedia.org/wiki/Halton_sequence)

  For formal definition, see p.578 of Greene (2008).

- Let's look at a Halton sequence of base 2 up to exponent 3:
  - $\{1/2\}$
  - $\{1/2^2, 3/2^2\}$. Note that $2/2^2$ is not included since it repeats with $1/2$.
  - $\{1/2^3, 3/2^3, 5/2^3, 7/2^3\}$. Note that $2/2^3$, $4/2^3$ and $6/2^3$ are not included since they repeat elements in the previous two sets.
  - Ordered as follows:

$$\{\frac{1}{2}, \frac{1}{2^2}, \frac{3}{2^2}, \frac{1}{2^3}, \frac{5}{2^3}, \frac{3}{2^3}, \frac{7}{2^3}\}$$

- How about of base 3 up to exponent 2?
  - $\{1/3, 2/3\}$
  - $\{1/3^2, 2/3^2, 4/3^2, 5/3^2, 7/3^2, 8/3^2\}$
  - Ordered as follows:

$$\{\frac{1}{3}, \frac{2}{3}, \frac{1}{3^2}, \frac{4}{3^2}, \frac{7}{3^2}, \frac{2}{3^2}, \frac{5}{3^2}, \frac{8}{3^2}\}$$

- Easier illustrated than said… Let's implement first then try to visualize.

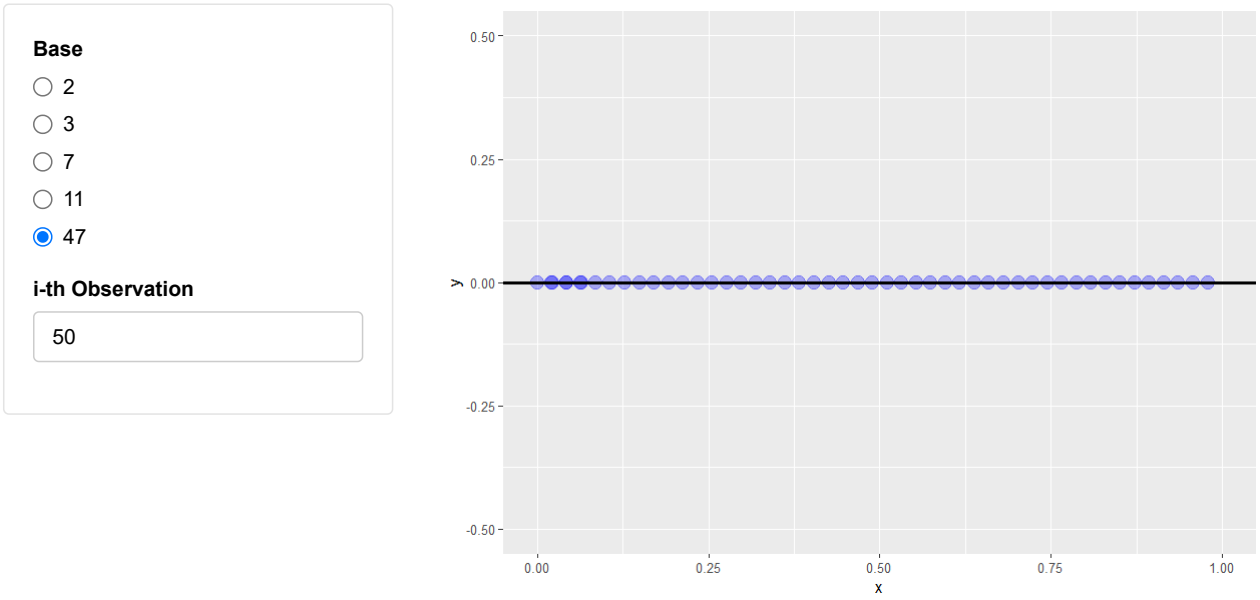
## Implementation on R

- `randtoolbox` implements Halton sequence as `halton(n, dim)`.
  - `n` is the number of points in the sequence.
  - `dim` is the number of dimension.
  - Returns a matrix of `n` rows and `dim` columns, where the i-th column corresponds to the sequence generated using the i-th prime number.
  - There are more input arguments, but we will not be covering them.
  - `randtoolbox` also provides `torus()` and `sobol()` other than Halton sequence. But we will also not be covering them.
- Generate a 3-d Halton sequence of 20 elements on each dimension.
  - The first column corresponds to an 20 element Halton sequence of base 2, the second one is of base 3, and the third one is of base 5.

```
#Demonstrate Halton sequence with randtoolbox
library(randtoolbox)
halton(n=20, dim=3)
```

```
##          [,1]       [,2] [,3]
##  [1,] 0.50000 0.33333333 0.20
##  [2,] 0.25000 0.66666667 0.40
##  [3,] 0.75000 0.11111111 0.60
##  [4,] 0.12500 0.44444444 0.80
##  [5,] 0.62500 0.77777778 0.04
##  [6,] 0.37500 0.22222222 0.24
##  [7,] 0.87500 0.55555556 0.44
##  [8,] 0.06250 0.88888889 0.64
##  [9,] 0.56250 0.03703704 0.84
## [10,] 0.31250 0.37037037 0.08
## [11,] 0.81250 0.70370370 0.28
## [12,] 0.18750 0.14814815 0.48
## [13,] 0.68750 0.48148148 0.68
## [14,] 0.43750 0.81481481 0.88
## [15,] 0.93750 0.25925926 0.12
## [16,] 0.03125 0.59259259 0.32
## [17,] 0.53125 0.92592593 0.52
## [18,] 0.28125 0.07407407 0.72
## [19,] 0.78125 0.40740741 0.92
## [20,] 0.15625 0.74074074 0.16
```

- Visualizing the gap-filling property.




- Observations:

  1. Earlier gaps are filled first. This explains that, for example, why the 5th element of the base 2 series is $5/2^3$ instead of $3/2^3$. If we fill-in by numeric order for each exponent $k$, then it will be taking too long to complete the $k$-th "cycle" of gap-filling. Think of the case that we generate a sequence of length 5. If it follows a numeric order then we see that the points are clustered in the segment $[0, 1/2]$. Then this sequence is no better than a randomly drawn one.
  2. The use of prime numbers alleviates repetition of numbers to some extent. Think about what happens if we hand-make a series of base 4: it certainly repeats by a lot with the sequence of base 2. Not what we want when attempting to fill a 2-d domain. This also implies that we want to do "prime jumping" when using Halton sequence for high-dimensional problems: 2 and 3 repeats every 6 times, but 2 and 7 repeats every 14 times.

3. Gaps are larger for smaller prime numbers like 2 and 3. But gap-filling is very slow for larger prime numbers like 47. This feature implies that we should not go for prime numbers too large. By using 100 nodes the one of base 2 might still have large gaps, but at least we fill the unit interval quite evenly. This is not the case if we use base 11 or 47 instead. The 47-based sequence is still try to do the second round filling!

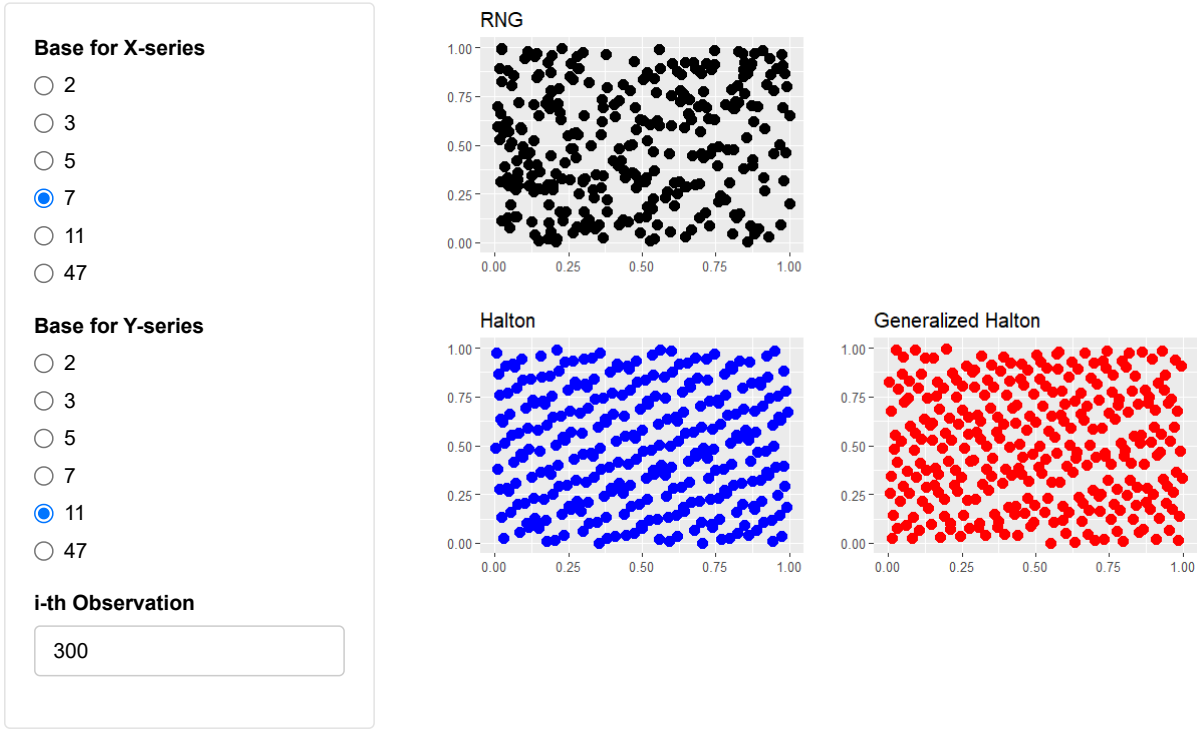
- The first and the third observations imply that Halton sequence is predictable, and can leave a quite obvious pattern.
- **Generalized Halton Sequence** tackles this problem. It is designed in the sense that we shuffle the Halton sequence. See Faure and Lemieux (2009) if you want some challenge.
- Implemented by `qrng` as `ghalton(n, d, method)`
  - `n` for number of elements.
  - `d` for dimension.
  - `method` is the way to generate the sequence, and needs to be a string. Can take either `"generalized"` or `"halton"`.

```
#Generalized Halton: a shuffled version of Halton
library(qrng)
ghalton(n=20, d=3, method="generalized")
```

```
##             [,1]       [,2]       [,3]
##  [1,] 0.26237171 0.96116118 0.35662601
##  [2,] 0.76237171 0.29449452 0.95662601
##  [3,] 0.01237171 0.62782785 0.55662601
##  [4,] 0.51237171 0.73893896 0.15662601
##  [5,] 0.38737171 0.07227229 0.75662601
##  [6,] 0.88737171 0.40560563 0.27662601
##  [7,] 0.13737171 0.85005007 0.87662601
##  [8,] 0.63737171 0.18338341 0.47662601
##  [9,] 0.32487171 0.51671674 0.07662601
## [10,] 0.82487171 0.99819822 0.67662601
## [11,] 0.07487171 0.33153155 0.39662601
## [12,] 0.57487171 0.66486489 0.99662601
## [13,] 0.44987171 0.77597600 0.59662601
## [14,] 0.94987171 0.10930933 0.19662601
## [15,] 0.19987171 0.44264266 0.79662601
## [16,] 0.69987171 0.88708711 0.31662601
## [17,] 0.29362171 0.22042044 0.91662601
## [18,] 0.79362171 0.55375378 0.51662601
## [19,] 0.04362171 0.92412415 0.11662601
## [20,] 0.54362171 0.25745748 0.71662601
```

- Compare the performance of Halton, Generalized Halton and uniform random draw in a $[0,1]^2$ space.
- We see the following:
  - Uniform draw leaves lots of gaps and clusters.
  - Do NOT use the Halton sequences of the same prime. It gives you a 45 degree line hence does not really fill the space evenly.
  - Halton can leave obvious patterns, and is particularly obvious for certain pairs e.g., 7 against 47. In all cases in the interactive widget we find self-similarity. This is bad since it implies correlation, just weaker than the 45 degree line (perfectly correlated).
  - Larger primes in Halto are problematic due to slow gap-filling. When used against other large primes like 11, we have an obvious pattern because both are filling slowly. When used against small prime numbers like 2, we have small clusters because one is filling slowly while the other is not.

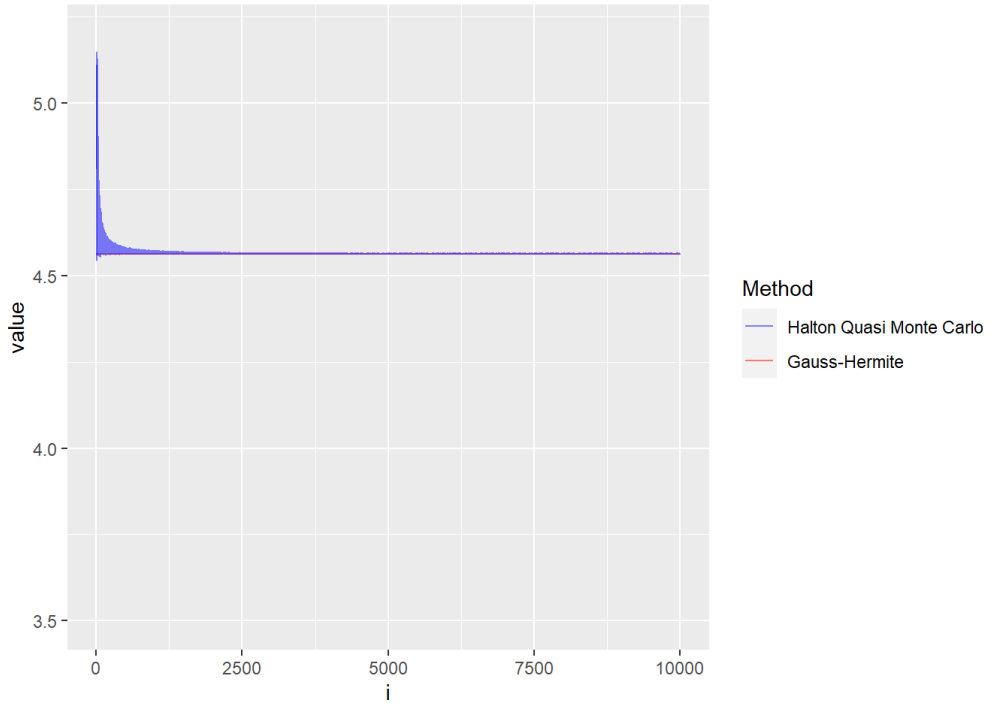
- Generalized Halton seems to be less problematic. It still has a pattern but not as obvious.




- Suggestions for using Halton sequence:
  1. Jump primes, but try to avoid certain pairs. 2 and 11 is OK (althought the result is still self-similar), but 7 and 11 is not very preferable. We should not pick primes too large.
  2. Do not use the very first points in the sequence as they are most correlated. This can be easily done in `halton()` by using `start` and `usetime` to start from the middle of a Halton sequence, or set `init=F` so that we don't start from the very first element every time we call `halton()`.
  3. Just switch to the Generalized Halton Sequence.
- Redo the example $\int_{-\infty}^{\infty} \sqrt{1+x^2} e^{-\frac{x^2}{3}} dx$ using Halton sequence. Also compare its performance with Gauss-Hermite. Turns out it works much better than "true" Monte Carlo.

```
#Quasi-Monte Carlo v.s. Gauss-Hermite
intfun.og<-function(x) {sqrt(1+(x^2))*exp(-((x^2)/3))}
cv<-function(y) {((2*y-1)/(y-(y^2)))}
jc<-function(y) {(2*(y^2)-2*y+1)/(((y^2)-y)^2)}

intfun.mc<-function(y) {
  y[y<=0]<-NA #This is to avoid ggplot from plotting it beyond [0,1].
  y[y>=1]<-NA #This is to avoid ggplot from plotting it beyond [0,1].
  rs<-intfun.og(x=cv(y))*jc(y=y)
  return(rs)
}

#Use Halton Sequence of Base 3, try length from 1 to 10000
hlt<-halton(10000,dim=2)[,2] #2nd column is for base 3

val.hqmc<-c()
for (i in 1:10000) {
  y<-hlt[1:i]
  tmp<-sum(intfun.mc(y))/i
  val.hqmc<-c(val.hqmc,tmp)
}

#Gauss-Hermite with 30 nodes.
intfun.gh<-function(x) {sqrt(3)*sqrt(1+3*(x^2))}
gh<-gaussHermite(n=30)
val.gh<-sum(intfun.gh(gh$x)*gh$w)

#Plot out the results for each iteration and compare with Gauss-Hermite
df<-data.frame(i=rep(1:10000,2),
               value=c(val.hqmc,rep(val.gh,10000)),
               Method=c(rep("Halton Quasi Monte Carlo",10000),rep("Gauss-Hermite",10000)))

ggplot(data=df,aes(x=i,y=value,color=Method))+
  geom_line(alpha=0.5)+
  scale_color_manual(values=c("blue","red"),
                     breaks=c("Halton Quasi Monte Carlo","Gauss-Hermite"))+
  ylim(c(3.5,5.2))
```

```
## Warning: Removed 7 row(s) containing missing values (geom_path).
```




### In-Class Exercise

Consider the integration problem

$$\int_2^{20} \frac{1}{1+x} dx$$

Integrate it using Gauss Quadrature, Monte Carlo, Quasi Monte Carlo with Halton and Generalized Halton sequences, and the grid approach. Recall that the true value of this problem is $\ln 7$, find the numbers of nodes or iterations needed for each of the numerical results to converge at $10^6$ level.

## Importance Sampling (introduction only)

- Some problems are particularly difficult to sample. For example, if the integrand has a vertical asymptote near 0, then draws near 0 lead to very large values. The results from Monte Carlo integration can vary quite much, and shows no sign of convergence while in fact the solution exist.
    - $x^{-0.99999}$ is only of the examples when integrated on $(0, 1]$. You are likely to encounter draws near 0 and each of them will be giving you extremely large numbers and lead to a seemingly divergence. But in fact this integral is well-defined, with the true value being 100000.
- Random sampling do not guarantee to cover the space evenly, hence can perform badly when some segments are particularly ill-behaved. The points near the asymptote is an example: they don't appear in the sample very often, but they impact the results by a lot when they appear.
- **Importance Sampling** is a technique to "reweight" the segments to draw while not affecting the true results. It helps to alleviated the problem raised above by sampling more points near the ill-behaved segments, say near the asymptote, to ensure the stability of outcome.
- Consider an integration problem

    $$\int_a^b g(x)q(x)dx$$

    where $q(x)$ is a pdf. In the case without uncertainty we have $q(x) = 1$ for all $x$ in the domain. We can find another

probability density function $p(x)$ and rewrite the original problem as

$$\int_a^b g(x)q(x)dx = \int_a^b \frac{g(x)q(x)}{p(x)}p(x)dx \equiv \int_a^b h(x)p(x)dx = E_{p(x)}(h(x))$$

where $h(x) \equiv \frac{g(x)q(x)}{p(x)}$ and $x$ is now drawn from $p(x)$ instead of from $q(x)$.

Then by Law of Large Number, we can do Monte Carlo integration by **drawing $x_i$ from the "new distribution"** $p(x)$ so that

$$\sum_{i=1}^N \frac{h(x)}{N} \approx \int_a^b h(x)p(x)dx = \int_a^b g(x)q(x)dx$$

- The new distribution $p(x)$ is oftentimes referred to as the **sampling distribution** or **proposal distribution**. When $q(x)$ is a probability density, then we refer to $q(x)/p(x)$ as the **likelihood ratio**.
- The key of this technique lies in choosing a *right* sampling distribution. There are some tips:
  1. Choose a $p(x)$ that belongs to the same **family** with $q(x)$ so that they have similar shapes. For example, if $q(x)$ is log-normal (exponential family), Weibull distribution with a fixed shape parameter is permissible for $p(x)$ as it also belongs to the exponential family.
  2. $p(x)$ needs to have a **thicker / fatter** tail than $q(x)$. That is, we want $p(x) > q(x)$ for large $x$. If not, then we will have a large likelihood ratio $q(x)/p(x)$ and potentially it leads to unbounded $g(x)q(x)/p(x)$ so that the integral becomes unstable (similar to the vertical asymptote problem in the sense of an exploding behavior).
  3. $p(x)$ needs to be easy to sample from. We don't want it to be ill-behaved on the interval we are interested in, e.g., asymptote.
- The GHK Simulator / Sampler is a famous example of importance sampling. It uses truncated normal probability as its sampling distribution to draw from a multivariate normal distribution. See pp.39-40 of Rossi, Allenby and McCulloch (2005) for details.

# High-Dimensional Integration and Curse of Dimensionality

- We have discussed how we can do multi-dimensional integration using Gaussian Quadrature Rules.
- Suppose that we work with the an N-dimensional problem:

$$\int_{a_1}^{b_1} \dots \int_{a_N}^{b_N} g(x_1, \dots, x_N)\, dx_1 \dots dx_N$$

the workflow is as follows:

Step 1: Choose the quadrature to use, and make whatever modifications needed, e.g., supplying $p(x)$ or doing change of variable for all dimensions to change the intervals. From this step we will get
$g(x_1, \dots, x_N) = p(x_1, \dots, x_N)f(x_1, \dots, x_N)$.

Step 2: Determine the number of nodes $n$, generate nodes $v^{(i)}$ and weights $\omega^{(i)}$.

Step 3: **Form a grid**. For example, assume that we look at a 3-dimensional problem ($x$, $y$, and $z$) with 2 nodes. Then Step 2 gives us $v^{(i)}$ and $v^{(i)}$. But we need to apply it to apply it to all 3 dimensions. So we have a grid of $2^3 = 8$ points: $(v^{(i)}, v^{(i)}, v^{(i)})$, $(v^{(i)}, v^{(i)}, v^{(i)})$, $(v^{(i)}, v^{(i)}, v^{(i)})$, and so on.

Step 4: Compute the sum where $x_k^{(i_k)}$ is replaced with $v^{(i)}$

$$\sum_{i_N=1}^{n} \dots \sum_{i_1=1}^{n} f(x_1^{(i_1)}, \dots, x_N^{(i_N)})) \, \omega_1^{(i_1)} \dots \omega_N^{(i_N)}$$

- Much easier using Monte Carlo or Quasi Monte Carlo:

  Step 1: Let $n$ denote the number of random draw. Do a random draw to get a sample, each of the sample point is defined as $(x_1^{(i)}, \dots, x_N^{(i)})$ where $i = 1, \dots, n$. Typically we first convert all dimensions to be in the $[0, 1]$ interval, and **independently** draw $n$ observations for each dimension. The the $i$-th draw from all dimension forms the $i$-th sample point $(x_1^{(i)}, \dots, x_N^{(i)})$. For Quasi Monte Carlo, just generate the points using Low Discrepancy Sequences like Halton sequence.

  Step 2: Compute the mean

  $$\frac{\sum_{i=1}^{n} g(x_1^{(i)}, \dots, x_N^{(i)})}{n}$$

- Try out this example:

  $$\int_0^1 \int_0^1 \int_0^1 (\frac{1}{1+x} + \sqrt{y} + 2z^3) \, dx \, dy \, dz$$

  We use `integral3()` in `pracma` package to compute the supposedly accurate solution as the benchmark. Then we compare the performance of using Gauss-Legendre, Monte Carlo, and Halton Monte Carlo.

```
set.seed(65536)
int.fun<-function(x,y,z) {
  (1/(1+x))+sqrt(y)+2*(z^3)
}

#Use integral3()
val.int3<-integral3(fun=int.fun, xmin=0, xmax=1, ymin=0, ymax=1, zmin=0, zmax=1)

#Gauss-Legendre with 30 points for each dimension
gl<-gaussLegendre(n=30, a=0, b=1)
#One might want to use triple for loop since we need to fill-in the 3-d space.
#But here we can use a Kronecker product to produce the grid needed.
unit<-matrix(1,30,1) #The unit matrix for Kronecker operation.

gl.grid.x<-gl$x
gl.grid.w<-gl$w
#Adding a new column takes a Kronecker product. We start from 1 column and need to add 2 columns to e
ventually have 3 columns. So we need to iterate 2 times.
for (i in 1:2) {
  gl.grid.x<-cbind(gl.grid.x%x%unit,rep(gl$x,30^i))
  gl.grid.w<-cbind(gl.grid.w%x%unit,rep(gl$w,30^i))
}
gl.grid.w<-as.data.frame(gl.grid.w)%>%'colnames<-'(c("w.x","w.y","w.z"))
gl.grid.x<-as.data.frame(gl.grid.x)%>%'colnames<-'(c("x","y","z"))
wt<-gl.grid.w$w.x*gl.grid.w$w.y*gl.grid.w$w.z
val.gl<-sum(int.fun(gl.grid.x$x,gl.grid.x$y,gl.grid.x$z)*wt)

#Halton Monte Carlo
hlt<-halton(n=27000, dim=3)
val.hlt<-mean(int.fun(hlt[,1],hlt[,2],hlt[,3]))

#Monte Carlo
mc.x<-cbind(runif(27000,0,1),runif(27000,0,1),runif(27000,0,1))
val.mc<-mean(int.fun(mc.x[,1],mc.x[,2],mc.x[,3]))

#Compare results.
c(val.int3,val.gl,val.hlt,val.mc)%>%'names<-'(c("int3","Gauss-Legendre","Halton","Monte Carlo"))
```

```
##           int3 Gauss-Legendre         Halton    Monte Carlo
##       1.859825       1.859817       1.859626       1.863223
```

- `int3` and Gauss-Legendre are accurate. Halton Quasi Monte Carlo gives similar results, but is a bit faster if you time it. Monte Carlo is also fast, but not as accurate if we plot out the value for each iteration.
- The advantage of Quasi Monte Carlo and Monte Carlo will stand out if we further increase the dimension:
  - The estimation error for Monte Carlo is $\sigma(g)/\sqrt{n}$, where $\sigma(g)$ is the standard deviation of the function to our interest $g$ and $n$ is the number of random draws. The error reduces as $n$ gets larger, but at a rate of $1/\sqrt{n}$ (rate of convergence).
  - The rate of convergence for Quasi Monte Carlo is $1/n$. For $n > 1$, we have $1/n < 1/\sqrt{n}$, meaning that Quasi Monte Carlo converges (much) faster than Monte Carlo.
  - Most importantly, the rate of convergence for both approaches are **independent** of the number of dimensions. Even in a 100-D environment we still have the same rate of convergence. This is NOT the case for Gaussian Quadrature, where its rate of convergence deteriorates as we add additional dimensions (**Curse of Dimensionality**) hence will eventually become too expensive to compute with really high dimensions.

- How high can the dimension be? Think about a general equilibrium quantitative trade model:
  - Typically we use WIOD so that there are about 60 countries plus a "rest of the world". Then the *bilateral trade costs* are country-pair-direction specific, e.g., Taiwan's export cost to USA is different from that of importing from USA. Then we have a 60-by-60 matrix of trade cost parameter for both types of trade costs.
  - We also have multiple sectors, typically 20 if we follow ISIC to some good extent. There are input-output linkages between these sectors (a sector's input intensities of materials from different sectors). So another 20-by-20 matrix. What's worse, this matrix is oftentimes **country-specific** so we have 60 of this kind of matrix in total.
  - We have other parameters, like TFP (country-time-specific), depreciation rate, preference parameters, and even migration elasticity that is location-sector-skill-specific.
  - If you are asked to compute the welfare gains from trade by a trade agreement that involves in multiple countries, then you will need to integrate over the bilateral trade costs of the relevant countries. It is high-dimensional, and it is tough to compute since changes in these costs entail general equilibrium effect that also alters endogenous allocations in this global economy!
  - Monte Carlo and Quasi Monte Carlo are your only way out.
  - Otherwise try to make specific modeling assumptions so that the welfare formula takes a CES form. This is the approach of the quantitative models based on Eaton Kortum (2002) and Caliendo and Parro (2015).

## Kronecker Product

- We just used a technique called the **Kronecker Product** when using Gauss-Legendre to speed up the computation.
- How it works: Consider two matrices

$$\mathbf{A}_{M\times N} \equiv \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1N} \\ a_{21} & a_{22} & \dots & a_{2N} \\ \dots & \dots & \dots & \dots \\ a_{M1} & a_{M2} & \dots & a_{MN} \end{bmatrix}$$

and

$$\mathbf{B}_{R\times S} \equiv \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1S} \\ a_{21} & a_{22} & \dots & a_{2S} \\ \dots & \dots & \dots & \dots \\ a_{R1} & a_{R2} & \dots & a_{RS} \end{bmatrix}$$

The Kronecker product $\mathbf{A}_{M\times N} \otimes \mathbf{B}_{R\times S}$ is defined as

$$\mathbf{A}_{M\times N} \otimes \mathbf{B}_{R\times S} = \begin{bmatrix} a_{11}\mathbf{B}_{R\times S} & a_{12}\mathbf{B}_{R\times S} & \dots & a_{1N}\mathbf{B}_{R\times S} \\ a_{21}\mathbf{B}_{R\times S} & a_{22}\mathbf{B}_{R\times S} & \dots & a_{2N}\mathbf{B}_{R\times S} \\ \dots & \dots & \dots & \dots \\ a_{M1}\mathbf{B}_{R\times S} & a_{M2}\mathbf{B}_{R\times S} & \dots & a_{MN}\mathbf{B}_{R\times S} \end{bmatrix}_{(M\times R)\times(N\times S)}$$

- In R, Kronecker product is called by `%x%`, and is used as ordinary arithmetic operation. For example, if we want to compute the Kronecker product between matrices / vectors `X` and `Y`, we simply do `X %x% Y`.
- The following code chunk demonstrates Kronecker product, and how we use this technique to build the grid needed for Gauss-Legendre in the previous example.

```
#Kronecker product of two 2-by-2 matrices
X<-matrix(c(1,3,5,7),2,2)
Y<-matrix(c(2,4,6,8),2,2)
X
```

```
##      [,1] [,2]
## [1,]    1    5
## [2,]    3    7
```

```
Y
```

```
##      [,1] [,2]
## [1,]    2    6
## [2,]    4    8
```

```
X%x%Y
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    2    6   10   30
## [2,]    4    8   20   40
## [3,]    6   18   14   42
## [4,]   12   24   28   56
```

```
Y%x%X #Order of taking Kronecker product matters!
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    2   10    6   30
## [2,]    6   14   18   42
## [3,]    4   20    8   40
## [4,]   12   28   24   56
```

```
#Use Kronecker product to expand dimension
#Example 1: Expand X into a 4-by-4 matrix
u<-matrix(1,2,2)
u
```

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

```
X%x%u
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    1    5    5
## [2,]    1    1    5    5
## [3,]    3    3    7    7
## [4,]    3    3    7    7
```

```
#Example 2: Expand a 3-by-1 vector into 9-by-1
X<-matrix(1:3,3,1)
u<-matrix(1,3,1)
X%x%u
```

```
##       [,1]
##  [1,]    1
##  [2,]    1
##  [3,]    1
##  [4,]    2
##  [5,]    2
##  [6,]    2
##  [7,]    3
##  [8,]    3
##  [9,]    3
```

```
#A simplified example demonstrating how we expand gl into a grid for 4 dimensions.
x<-matrix(1:2,2,1)
kp<-matrix(1,2,1)
out<-x
for (i in 1:3) {
  out<-cbind(out%x%kp,rep(x,2^i))
}
out
```

```
##       [,1] [,2] [,3] [,4]
##  [1,]    1    1    1    1
##  [2,]    1    1    1    2
##  [3,]    1    1    2    1
##  [4,]    1    1    2    2
##  [5,]    1    2    1    1
##  [6,]    1    2    1    2
##  [7,]    1    2    2    1
##  [8,]    1    2    2    2
##  [9,]    2    1    1    1
## [10,]    2    1    1    2
## [11,]    2    1    2    1
## [12,]    2    1    2    2
## [13,]    2    2    1    1
## [14,]    2    2    1    2
## [15,]    2    2    2    1
## [16,]    2    2    2    2
```

- Kronecker Product is a **matrix operation** and is faster than loops in general. We can construct a high dimensional grid from a sequence at high speed.
- This is particularly important for problems involving in really higher dimensions. Think about performing a 5-d integration using 30 Gauss-Legendre nodes: $30^5 = 24300000$ points to be computed and can even take away 30 minutes or more if the problem is highly non-linear!!
- Use the previous 3-d integration problem to demonstrate how the Kronecker product approach is more efficient than the brainless triple loop approach.

```
int.fun<-function(x,y,z) {
  (1/(1+x))+sqrt(x)+2*(z^3)
}
gl<-gaussLegendre(n=30, a=0, b=1)

#Kronecker product approach
ptm <- proc.time() #Timer start.

unit<-matrix(1,30,1)
gl.grid.x<-gl$x
gl.grid.w<-gl$w

#Add one dimension / column at a time. Each time involves in an expansion via Kronecker product and b
inding with the nodes of the new dimension. So we cannot avoid loops, but at least we reduce the numb
er of loops used.
for (i in 1:2) {
  gl.grid.x<-cbind(gl.grid.x%x%unit,rep(gl$x,30^i))
  gl.grid.w<-cbind(gl.grid.w%x%unit,rep(gl$w,30^i))
}

gl.grid.w<-as.data.frame(gl.grid.w)%>%'colnames<-'(c("w.x","w.y","w.z"))
gl.grid.x<-as.data.frame(gl.grid.x)%>%'colnames<-'(c("x","y","z"))
wt<-gl.grid.w$w.x*gl.grid.w$w.y*gl.grid.w$w.z

rs.kron<-sum(int.fun(gl.grid.x$x,gl.grid.x$y,gl.grid.x$z)*wt)

proc.time() - ptm #Stop timer and show proc.time.
```

```
## 使用者    系統    流逝
##   0.02    0.00    0.01
```

```
#Now for the brainless triple loop approach
ptm <- proc.time() #Timer start.

v<-c()
for (i in 1:30) {
  for (j in 1:30) {
    for (k in 1:30) {
      v<-c(v,int.fun(gl$x[i],gl$x[j],gl$x[k])*gl$w[i]*gl$w[j]*gl$w[k])
    }
  }
}
rs.loop<-sum(v)

proc.time() - ptm #Stop timer and show proc.time. Much longer than Kronecker!
```

```
## 使用者    系統    流逝
##   1.33    0.00    1.33
```

```
#Compare evaluated outcome.
rs.kron==rs.loop
```

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

### In-Class Exercise

Try out the following 5-dimensional integration

$$\int_0^1 \int_0^1 \int_0^1 \int_0^1 \int_0^1 (\frac{1}{1+x} + \sqrt{y} + 2z^3 + \sqrt{2p} + q^{\frac{1}{3}})\, dx\, dy\, dz\, dp\, dq$$

Specifically, do it using Gauss-Legendre, Monte Carlo, and Halton Monte Carlo. For Gauss-Legendre, use 30 nodes for each dimension so that we have 24300000 nodes in total in the 5-d grid. For other approaches, draw sequences of 24300000 points for each dimension.

# Assignment

This assignment is going to be challenging, and can worth a whole class for discussion. Consider the following function defined on $r \in (0, 1)$:

$$f(r) = r^{-\frac{k}{s-1}}(1-r)^{\frac{k-1}{s-1}-1}$$

where $k = 1.56$ and $s = 1.667$. Your missions are as follows:

1. Plot out $f(r)$ on $r \in (0, 1)$. If you do it right then you'll see that it has two vertical asymptotes, one is near 0 and the other is near 1. If the asymptote near 0 do not come out (Mathematica has this problem, and I'm not sure if it is also the case in R), please try to plot another figure on $(0, 0.001)$.
2. Integrate $f(r)$ on $(2 \times 10^{-6}, 1)$ and call it $f_1$. Also integrate $f(r)$ on $(10^{-6}, 1)$ and call it $f_2$. Compute $f_1 / f_2$. The true value should be somewhere around 0.197671. Since you have learned various techniques for numerical integration:

   2a. Do this using Monte Carlo, and report the number of draws needed for $f_1 / f_2$ to converge at $10^5$ (for this one, all you need to do is to stop when the error falls below $10^5$).

   2b. Do the same using Halton or Generalized Halton Sequence.

   2c. Use Gauss Quadrature, and report the nodes needed to meet the above convergence criteria.

   2d. If the integration technique does not work out, plot out the result for all of your iterations to visually tell if there are signs of convergence.

# Reference


- Caliendo, Lorenzo, and Fernando Parro (2015), "Estimates of the Trade and Welfare Effects of NAFTA," *Review of Economic Studies*, 82, 1-44.
- Eaton, Jonathan, and Samuel Kortum (2002), "Technology, Geography, and Trade," *Econometrica*, 70, 1741-1779.
- Faure, Henri, and Christiane Lemieux (2009), "Generalized Halton Sequences in 2008: A Comparative Study," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 19, 1-31.
- Greene, William, H. (2008), *Econometric Analysis* 6th ed., Pearson.
- Rossi, Peter E., Greg M. Allenby, and Robert McCulloch (2005), *Bayesian Statistics and Marketing*, John Wiley & Sons.
- White, Halbert (2000), *Asymptotic Theory for Econometricians* Revised ed., Academic Press.