

Computational Statistics Summary

Maximilian Pfundstein

2019-03-13

Contents

1	Handling Computational Errors	2
2	Difference Quotient	2
3	Variance Estimators	2
4	Optimization	3
4.1	optimize()	3
4.2	optim()	3
5	Sampling Based on Size	5
6	Inverse CDF Method	6
7	Acceptance / Rejection Method	8
8	Metropolis-Hastings Algorithm	10
9	Gelman-Rubin Factor	12
10	Monte Carlo Integration	14
11	Gibbs Sampling	15
12	Hypothesis Testing	16
12.1	Defining a Test-Statistics	16
12.2	Non-Parametric Bootstrap	16
12.3	Density of non-parametric T-Values	17
12.4	Defining the Hypothesis Test (Permutation Test)	17
12.5	Crude Estimate of the Power	18
13	Bootstrap with Bias-Correction	19
14	Variance Estimation using Jackknife	20
15	Confidence Intervals with Respect to Length and Location	20
16	Genetic Algorithms	21
17	Expectation-Maximization Algorithm (EM)	26
17.1	Expectation-Step (E-Step)	27
17.2	Maximization-Step (M-Step)	27
17.3	EM-Implementation	28
18	Uniform Sampling	29
19	Normal Sampling	29

20 Pareto Distribution	30
21 Geometric Sampling	30
22 Parabolic Interpolation	31
23 Miscellaneous Plots	32
23.1 Histogram	32
23.2 Histogram with Mean	32
23.3 Simple X/Y Plot	33
23.4 Variance Plot	33
23.5 Scatterplot With Geom Smoother	33
24 Useful Code Snippets	33
24.1 RMarkdown Setup	33
24.2 Knitr options	34
24.3 Including Source Code	34

1 Handling Computational Errors

```
x1 = 1/3
x2 = 1/4

if (all.equal(x1-x2, 1/12)) {
  print("Subtraction is correct.")
} else {
  print("Subtraction is wrong.")
}
```

2 Difference Quotient

$$f'(x) = \frac{f(x + \epsilon) - f(x)}{\epsilon}$$

```
f_prime = function(x, epsilon = 10^(-5)) {
  return( (f(x + epsilon) - f(x)) / epsilon)
}
```

3 Variance Estimators

Krzysztof:

$$\text{Var}(\vec{x}) = \frac{1}{n-1} \left(\sum_{i=1}^n x_i^2 - \frac{1}{n} \left(\sum_{i=1}^n x_i \right)^2 \right)$$

```
myvar = function(x) return(1/(length(x)-1) * (sum(x^2) - (sum(x)^2)/length(x)))
```

My:

$$s = \frac{\sum_{i=1}^n (x_i - \bar{x})^2}{n - 1}$$

```
custom_variance = function(x) {  
  diff_mean = x - mean(x)  
  return(sum(diff_mean^2 / (length(x) - 1)))  
}
```

4 Optimization

4.1 optimize()

```
myMSE = function(lambda, pars) {  
  model = loess(Y ~ X, data=pars, enp.target = lambda)  
  prediction = predict(model, newdata = pars$Xtest)  
  mse = sum((prediction - pars$Ytest)^2)/length(pars$Ytest)  
  return(mse)  
}  
  
# parameters for the myMSE function -----  
pars = list(X = train$Day, Y = train$LMR, Xtest = test$Day, Ytest = test$LMR)  
lambdas = seq(from = 0.1, to = 40, by = 0.1)  
# applying the myMSE function to all lambdas -----  
mses = sapply(X = lambdas, FUN = myMSE, pars = pars)  
  
o = optimize(myMSE, tol = 0.01, interval = c(0.1, 40), pars = pars)  
o$minimum  
o$objective
```

Plotting a function with a minimum:

```
lambdas[which.min(mses)]  
length(lambdas)  
  
df = data.frame(lambdas, mses)  
  
ggplot(df) +  
  geom_line(aes(x = lambdas, y = mses), color = "#C70039") +  
  geom_point(aes(x = seq(0.1, 40, by = 0.1)[which.min(mses)],  
    y = mses[which.min(mses)], color = "min MSE"),  
    colour = "blue") +  
  labs(title = "Lambdas VS MSEs", y = "MSE", x = "Lambda") +  
  theme_minimal()
```

4.2 optim()

General usage:

```
optim(35, myMSE, method = "BFGS", pars = pars, control = list(fnscale = 1))
```

For optimizing likelihood:

```
# c(mu, sigma)
neg_llik_norm = function(par) {
  n = nrow(as.matrix(data))
  p1 = (n/2)*log(2*pi)
  p2 = (n/2)*log(par[2]^2)
  sum = sum((data - par[1])^2)
  p3 = 1/(2*par[2]^2) * sum
  return(p1+p2+p3)
}

# c(mu, sigma)
neg_llik_norm_prime = function(par) {
  n = nrow(as.matrix(data))
  mu_prime = -1/(n*par[2]^2) * sum(data-par[1])
  sigma_prime = 1/(2*par[2]^2) * (n - (1/(par[2]^2)) * sum((data-par[1])^2))

  return(c(mu_prime, sigma_prime))
}

optim(c(0, 1), neg_llik_norm, method = "CG")
optim(c(0, 1), neg_llik_norm, method = "CG", gr = neg_llik_norm_prime)
optim(c(0, 1), neg_llik_norm, method = "BFGS")
optim(c(0, 1), neg_llik_norm, method = "BFGS", gr = neg_llik_norm_prime)
```

Answer: The negative log-likelihood function for the normal distribution is defined by:

$$\mathcal{L}(\mu, \sigma^2, x_1, \dots, x_{100}) = \frac{n}{2} \ln(2\pi) + \frac{n}{2} \ln(\sigma^2) + \frac{1}{2\sigma^2} \sum_{j=1}^n (x_j - \mu)^2$$

The estimators are:

$$\hat{\mu}_n = \frac{1}{n} \sum_{j=1}^n x_j$$

and

$$\hat{\sigma}_n^2 = \frac{1}{n} \sum_{j=1}^n (x_j - \hat{\mu})^2$$

Answer: The partial derivatives for the negative log-likelihood are given by:

$$\frac{\partial \mathcal{L}(\mu, \sigma^2, x_1, \dots, x_{100})}{\partial \mu} = -\frac{1}{n\sigma^2} \sum_{j=1}^n (x_j - \mu)$$

$$\frac{\partial \mathcal{L}(\mu, \sigma^2, x_1, \dots, x_{100})}{\partial \sigma^2} = \frac{1}{2\sigma^2} \left(n - \frac{1}{\sigma^2} \sum_{j=1}^n (x_j - \mu)^2 \right)$$

Be aware that the following code actually has a flaw as it is not finding the correct centroid.

```

d_abs = function(x, y) {
  return(rowSums(abs(x - y)))
}

d_sq = function(x, y) {
  return(rowSums((x - y)^2))
}

fratchet_mean = function(y, X, w = 1, dist) {
  return(sum(w * dist(X, y)))
}

rng_bivariate = function(n, p) {
  U = rng_rnorm(n)
  V = rng_rgeom(n, p)

  return(as.matrix(data.frame(U, V)))
}

sample_10 = rng_bivariate(10, p = 1/3)

y_10_opt_abs = optim(y_init_10, fratchet_mean, method = "BFGS", w = 1,
  dist = d_abs, X = sample_10, control = list(fnscale = 1))

samples_10_df_abs = data.frame(sample_10)
samples_10_df_abs$meanU = mean(samples_10_df_abs$U)
samples_10_df_abs$meanV = mean(samples_10_df_abs$V)
samples_10_df_abs$medianU = median(samples_10_df_abs$U)
samples_10_df_abs$medianV = median(samples_10_df_abs$V)
samples_10_df_abs$optU = y_10_opt_abs$par[1]
samples_10_df_abs$optV = y_10_opt_abs$par[2]

ggplot(samples_10_df_abs) +
  geom_point(aes(x = U, y = V, colour = "Samples"), size = 2, stroke = 1) +
  geom_point(aes(x = meanU, y = meanV, colour = "Mean"), size = 2, stroke = 1) +
  geom_point(aes(x = medianU, y = medianV, colour = "Median"), size = 2, stroke = 1) +
  geom_point(aes(x = optU, y = optV, colour = "Optimum"), size = 2, stroke = 1) +
  labs(title = "10 Samples with absolute distance", y = "V", x = "U", color = "Legend") +
  scale_color_manual(values = c("#7BA9FF", "#FFC300", "#ff4444", "black")) +
  scale_shape_manual(values = c(21, 21, 21)) +
  theme_minimal()

```

5 Sampling Based on Size

Task: Use a uniform random number generator to create a function that selects 1 city from the whole list by the probability scheme offered above (do not use standard sampling functions present in R).

```

get_city_by_urn_wo = function(city_pool) {

  # We take the cumulative sum and then runif from 1 to max(cumulative sum).
  # This way we respect the proportions. As we need every intermediate result,
  # we use a loop

```

```

cumulative_pop_sum = 0

for (i in 1:nrow(city_pool)) {
  cumulative_pop_sum = cumulative_pop_sum + city_pool$Population[i]
  city_pool$CumSum[i] = cumulative_pop_sum
}

# Now we get a random value between 1 to max(cumulative sum). As larger muni-
# cipalities have larger ranges, this works as expected
selection =
  floor(runif(n = 1, min = 1, max = city_pool$CumSum[nrow(city_pool)]))

# Return the first city which has a greater CumSum than the selection
return(city_pool[city_pool$CumSum > selection,][1, c(1, 2)])
}

```

Task: Use the function you have created in step 2 as follows:

- Apply it to the list of all cities and select one city
- Remove this city from the list
- Apply this function again to the updated list of the cities
- Remove this city from the list
- ... and so on until you get exactly 20 cities.

Answer: We will combine all of these steps in one function. We're lazy.

```

get_n_cities = function(data, n) {

  # Create a copy to not touch the original data.
  city_pool = data
  selected_cities = data.frame()

  # As long as we don't have n samples, get one and remove it from the pool,
# as we sample without replacement
  while(nrow(selected_cities) < n) {
    selected_city = get_city_by_urn_wo(city_pool)
    selected_cities = rbind(selected_cities, selected_city)
    city_pool = city_pool[!rownames(city_pool) %in% rownames(selected_cities),]
  }

  return(selected_cities)
}

sample = get_n_cities(data, 20)

```

6 Inverse CDF Method

The double exponential (Laplace) distribution is given by formula

$$DE(\mu, \alpha) = \frac{\alpha}{2} e^{-\alpha|x-\mu|}$$

Task: Write a code generating double exponential distribution $DE(0, 1)$ from $Unif(0, 1)$ by using the inverse CDF method. Explain how you obtained that code step by step. Generate 10000 random numbers from this

distribution, plot the histogram and comment whether the result looks reasonable.

1. Derive the CDF from the PDF (`dfunc()`) by taking the integral $\int_{-\infty}^x \text{CDF} dx$. This function is the `pfunc()` (CDF!, cumulative).
2. Swap `x` and `y` to receive the quantile function `qfunc()`.
3. Combine both functions to create the `rfunc()`.

This can look like this:

```
# double exponential (Laplace) distribution

# PDF
ddel = function(x = 1, mu = 0, b = 1) {
  return(1/(2*b) * exp(-abs(x-mu)/(b)))
}

# CDF
pdel = function(x = 1, mu = 0, b = 1) {
  return(1/2 + 1/2 * sgn(x-mu) * (1 - exp(-abs(x-mu)/b)))
}

# Quantile
qdel = function(p, mu = 0, b = 1) {
  if (p < 0 | p > 1) stop("p must be in range (0, 1)")
  if (p <= 0.5) return(mu + b * log(2 * p))
  return (mu - b * log(2 - 2 * p))
}

# Random
rdel = function(n = 1, mu = 0, b = 1) {
  quantiles = runif(n = n, min = 0, max = 1)
  rdels = sapply(X = quantiles, FUN = qdel, mu = mu, b = b)
  return(rdels)
}
```

To look how the distribution looks like with different parameters, use the following code and plots:

```
sample_rdel_0_1 = rdel(10000, mu = 0, b = 1)
sample_rdel_0_2 = rdel(10000, mu = 0, b = 2)
sample_rdel_0_4 = rdel(10000, mu = 0, b = 4)
sample_rdel_m5_4 = rdel(10000, mu = -5, b = 4)

df = data.frame(sample_rdel_0_1, sample_rdel_0_2,
  sample_rdel_0_4, sample_rdel_m5_4)

p1 = ggplot(df) +
  geom_histogram(aes(x = sample_rdel_0_1),
    color = "#FFC300", fill = "#FFC300", binwidth = 0.01) +
  xlim(-10, 10) +
  ylim(0, 60) +
  ggtitle("RD(0, 1)") +
  theme_minimal()

# p2, p3, p4

grid.arrange(p1, p2, p3, p4, nrow = 2)
```

7 Acceptance / Rejection Method

For plotting the PDF and CDF to compare for instance two of them:

```
sequence = seq(from = -10, to = 10, by = 0.01)

dnorm_samples = sapply(X = sequence, FUN = dnorm)
ddel_samples = sapply(X = sequence, FUN = ddel)
pnorm_samples = sapply(X = sequence, FUN = pnorm)
pdel_samples = sapply(X = sequence, FUN = pdel)

df = data.frame(dnorm_samples, ddel_samples, pnorm_samples, pdel_samples)

ggplot(df) +
  geom_line(aes(x = sequence, y = dnorm_samples,
                colour = "Normal Distribution (PDF)")) +
  geom_line(aes(x = sequence, y = ddel_samples,
                colour = "Double Exponential Distribution (PDF)")) +
  labs(title = "dnorm() and ddel()", y = "Density",
       x = "X", color = "Legend") +
  scale_color_manual(values = c("#17202A", "#C70039")) +
  theme_minimal()

ggplot(df) +
  geom_line(aes(x = sequence, y = pnorm_samples,
                colour = "Normal Distribution (CDF)")) +
  geom_line(aes(x = sequence, y = pdel_samples,
                colour = "Double Exponential Distribution (CDF)")) +
  labs(title = "pnorm() and pdel()", y = "Cumulative Density",
       x = "X", color = "Legend") +
  scale_color_manual(values = c("#17202A", "#C70039")) +
  theme_minimal()
```

For calculating the C value, use this code:

```
c = max(dnorm_samples / ddel_samples)
```

Sometimes we cannot do this this easily or have to derive the optimal C by hand. This can be done as follows:

$$C \geq \frac{\text{target}}{\text{proposal}} = \frac{f(x)}{g(x)}$$

We then calculate:

$$\max_x \frac{f(x)}{g(x)}$$

Which is taking the derivative and setting it to 0. Then we solve for x and plug in x into

$$\frac{f(x)}{g(x)} = C$$

Note: Sometimes $C = M$.

For creating nice plots, use this:


```

df$scaled_envelop = c * df$ddel_samples

ggplot(df) +
  geom_line(aes(x = sequence, y = dnorm_samples,
                colour = "Normal Distribution (PDF)")) +
  geom_line(aes(x = sequence, y = ddel_samples,
                colour = "Double Exponential Distribution (PDF)")) +
  geom_line(aes(x = sequence, y = scaled_envelop,
                colour = "Scaled Double Exponential Distribution (PDF)")) +
  labs(title = "Envelope", y = "Density",
       x = "X", color = "Legend") +
  scale_color_manual(values = c("#17202A", "#C70039", "#581845")) +
  theme_minimal()

ggplot(df) +
  geom_ribbon(aes(x = sequence, ymin = df$dnorm_samples, ymax = df$scaled_envelop),
            alpha = 0.8, fill = "#C70039", color = "#C70039") +
  geom_ribbon(aes(x = sequence, ymin = 0, ymax = df$dnorm_samples),
            alpha = 0.8, fill = "#DAF7A6", color = "#DAF7A6") +
  labs(title = "Acceptance and Rejection Regions", y = "Density",
       x = "X", color = "Legend") +
  scale_color_manual(values = c("#17202A", "#C70039", "#581845")) +
  theme_minimal()

```

And for actually using it, use:

```

rs = c()
rs_rejected = c()

while (length(rs) < 2000) {
  # Take a random sample from our proposal (x-axis)
  z = rdel(n = 1, mu = 0, b = 1)

  # Take a uniform, thus a random y value
  u = runif(n = 1, min = 0, max = c * ddel(z))

  # Check in which region this on lies
  if (u <= dnorm(z)) {
    rs = c(rs, z)
  }
  else {
    rs_rejected = c(rs_rejected, z)
  }
}

```

Plot the drawn samples:

```

df2 = as.data.frame(rs)

ggplot(df2) +
  geom_histogram(aes(x = rs),
                color = "#C70039", fill = "#C70039", binwidth = 0.01) +
  xlim(-5, 5) +
  ylim(0, 30) +
  ggtitle("N(0, 1) sampled from DE(0, 1)") +

```

```
theme_minimal()
```

Expected rejection rate:

```
1 - 1/c
```

Observed rejection rate:

```
length(rs_rejected) / (length(rs)+length(rs_rejected))
```

8 Metropolis-Hastings Algorithm

We have given a target function (probably only a proportional one).

```
# Target function with original scaling
f = function(x) {
  return(120 * x^5 * exp(-x))
}

# Target function
df = function(x) {
  x = ifelse(x <= 0, 0.000001, x)
  return(x^5 * exp(-x))
}

sequence = seq(from = 0.01, to = 20, by = 0.01)

real_f = f(sequence)
plotdf = data.frame(sequence, real_f)

ggplot(plotdf) +
  geom_line(aes(x = sequence, y = real_f), color = "#6091ec") +
  labs(title = "Target Density Function", y = "Density",
       x = "X", color = "Legend") +
  theme_minimal()
```

The Metropolis-Hastings algorithm is implemented by:

```
#' Metropolis Hasting Algorithm
#'
#' @param n Number of samples from the target distribution.
#' @param x_0 Initial state from Pi.
#' @param b Burn-in steps to remove from samples.
#' @param proposal Selects the proposal function.
#' @param keep_burnin Decides if to keep the samples during the burn-in period. #'
#' @return Returns a list containing samples.
#' @export
#'
#' @examples
metropolis_hastings = function(n = 1, x_0 = 1, b = 50, proposal = "lnorm",
                              keep_burnin = FALSE) {

  # Vectors to store the samples in
  samples = c()
```

```

# Samples from proposal from the random walk (MC)
xt = x_0
xt_1 = x_0

while (length(samples) < n) {

  # Generate proposal state
  if (proposal == "lnorm") {
    x_star = rlnorm(n = 1, meanlog = log(xt), sdlog = 1)

    # Calculate correction factor C
    c = dlnorm(xt_1, meanlog = log(xt), sdlog = 1) /
      dlnorm(x_star, meanlog = log(xt), sdlog = 1)
  }
  else if (proposal == "chisquared") {
    x_star = rchisq(n = 1, df = floor(xt + 1))

    # Calculate correction factor C
    c = dchisq(x = xt_1, df = floor(xt + 1)) /
      dchisq(x_star, df = floor(xt + 1))
  }
  else {
    stop("Invalid proposal.")
  }

  # Calculate acceptance probability alpha
  if (df(xt_1) <= 0) {
    # We need this to avoid troublesome areas where the density is so low that
    # it's 0 from a computational point of view.
    alpha = 0
  }
  else {
    alpha = min(1, df(x_star)/df(xt_1) * c)
  }

  # Generate u from uniform
  u = runif(n = 1, min = 0, max = 1)

  # Decide if to accept or reject the proposal
  if (u <= alpha) { # Accept
    xt_1 = xt
    xt = x_star
    samples = c(samples, x_star)
  }
  else {
    # Reject
    xt = xt_1
  }
}

# Return samples
if (keep_burnin) return(samples) {
  return(samples[b+1:length(samples)])
}

```

```
}
```

Create a fancy plot of the burn-in period using:

```
burnin = 10

results = metropolis_hastings(100, x_0 = 40, b = burnin, proposal = "lnorm",
                              keep_burnin = TRUE)

plotdf = data.frame(index = 1:length(results), values = results)

ggplot(plotdf) +
  geom_line(aes(x = index, y = values), color = "#6091ec") +
  labs(title = "Traceplot For The Burn-In Period (burnin = 10)", y = "X*",
       x = "Iteration", color = "Legend") +
  geom_vline(xintercept = burnin, color = "#C70039") +
  theme_minimal()
```

And to plot the distribution of the samples, use:

```
results_lnorm = metropolis_hastings(10000, x_0 = 5, b = 100, proposal = "lnorm")

plotdf = data.frame(results_lnorm)

ggplot(plotdf) +
  geom_histogram(aes(x = results_lnorm),
                 color = "#000000", fill = "#C70039",
                 bins = length(results_lnorm)/100) +
  labs(title = "Samples From Target Function Using Normal Proposal",
       y = "Density",
       x = "X", color = "Legend") +
  theme_minimal()
```

9 Gelman-Rubin Factor

The custom implementation looks like this:

```
gelman_rubin_factor = function(sequence_matrix) {

  k = nrow(sequence_matrix)
  n = ncol(sequence_matrix)

  B = n/(k-1) * sum((rowMeans(sequence_matrix) - mean(sequence_matrix))^2)
  S_squared = rowSums((sequence_matrix - rowMeans(sequence_matrix))^2 / (n - 1))
  W = sum(S_squared / k)
  V = ((n - 1) / n * W) + (1/n * B)
  R = sqrt(V / W)
  return(R)
}
```

And calling it with the previously defined Metropolis-Hastings algorithm, use:

```
k = 10 # row
n = 1000 # col
```

```

sequence_matrix = matrix(NaN, nrow = k, ncol = n)

for (i in 1:k) {
  sequence_matrix[i,] = metropolis_hastings(n, x_0 = i, b = 0,
                                           proposal = "chisquared", keep_burnin = TRUE)
}

print(gelman_rubin_factor(sequence_matrix))

k = 10 # row
n = 5 # col

sequence_matrix = matrix(NaN, nrow = k, ncol = n)

for (i in 1:k) {
  sequence_matrix[i,] = metropolis_hastings(n, x_0 = i, b = 0,
                                           proposal = "chisquared", keep_burnin = TRUE)
}

print(gelman_rubin_factor(sequence_matrix))

```

There is also a built-in implementation which can be invoked with the following code:

```

library(coda)

results5 = list()
results10 = list()
results50 = list()
k = 10

for (i in 1:k) {
  results5[[i]] = mcmc(metropolis_hastings(5, x_0 = i, b = 0,
                                           proposal = "chisquared", keep_burnin = TRUE))
}

for (i in 1:k) {
  results10[[i]] = mcmc(metropolis_hastings(10, x_0 = i, b = 0,
                                           proposal = "chisquared", keep_burnin = TRUE))
}

for (i in 1:k) {
  results50[[i]] = mcmc(metropolis_hastings(50, x_0 = i, b = 0,
                                           proposal = "chisquared", keep_burnin = TRUE))
}

mcmc_list5 = mcmc.list(results5)
gelman.diag(mcmc_list5)

mcmc_list10 = mcmc.list(results10)
gelman.diag(mcmc_list10)

mcmc_list50 = mcmc.list(results50)
gelman.diag(mcmc_list50)

```

10 Monte Carlo Integration

Let's say we want to integrate the following integral:

$$\int_0^{\infty} x^4 e^{-2x} dx$$

so

$$f(x) = x^4 e^{-2x}$$

For estimating this integral with MCMC we need to split $f(x)$ into $d(x)g(x)$ which can be chosen in any way, as long as the following conditions for $g(x)$ are met:

- $g(x) \geq 0$ for the drawn interval.
- The integral inside the interval is finite, so $\int_a^b g(x) = C < \infty$

$g(x)$ must be drawn from the whole integral range, so from a to b , and exactly the range. If one of the limits is ∞ , $g(x)$ must have the same limit (like a gamma distribution for instance for 0 to ∞).

Then draw samples from $g(x)$, x_i .

Then calculate:

$$\frac{1}{n} \sum_{i=1}^n d(x_i)$$

Which is the mean of the $d(x_i)$ values.

In this example this looks like this:

The sampling function is given by:

$$d(x, \alpha = 2, \beta = 2) = 4xe^{-2x}$$

Note that $\text{Gamma}(\alpha) = (\alpha - 1)!$

We then identify:

$$g(x) = \frac{1}{4}x^3$$

So we simply put our drawn samples into $g(x)$ and calculate the mean.

```
g = function(x) return(1/4 * x^3)

mean(g(samples))
```

Note: When the limits do not include infinity, we have to scale the results. So we take the result times C where $C = b - a$. This might be wrong, but that's what I got from different examples.

11 Gibbs Sampling

For Gibbs Sampling it is mandatory to find the Marginals. For doing that, do the following:

- First write down the prior and the likelihood. There is no general way for this, it depends on the problem.
- Then write down the posterior which is the product of both.
- For finding the marginals, drop every factor that is not dependent on the marginal variable.

If we have to combine two normal distributions, it can be done using the following rules:

The product of two normal distributions is given by (see <https://www.johndcook.com/blog/2012/10/29/product-of-normal-pdfs/> as a recap if not present):

$$\sigma_{new}^2 = \frac{\sigma_1^2 \sigma_2^2}{\sigma_1^2 + \sigma_2^2}$$
$$\mu_{new} = \frac{\sigma_1^{-2} \mu_1 + \sigma_2^{-2} \mu_2}{\sigma_1^{-2} + \sigma_2^{-2}}$$

For implementing the Gibbs Sampler we write one function for getting the marginals and one main function for doing the actual sampling.

```
get_mu = function(i, sigma = 0.2, mu, Y) {  
  
  if (i == 1) {  
    # First Marginal  
    return(rnorm(n = 1, mean = (mu[2] + Y[1])/2, sd = sqrt(1/2 * sigma)))  
  }  
  else if (i == 50) {  
    # Last Marginal  
    return(rnorm(n = 1, mean = (2 * mu[i-1] - Y[i-1] + 2 * Y[i])/3,  
                  sd = sqrt(2/3 * sigma)))  
  }  
  
  # General Marginals  
  return(rnorm(n = 1, mean = (2 * mu[i-1] + 2 * mu[i+1] + 2 * Y[i] - Y[i-1])/5,  
                sd = sqrt(2/5 * sigma)))  
}  
  
gibbs_sample = function(n, Y, include_matrix = FALSE) {  
  
  # Vectors to store the samples in  
  mu_matrix = matrix(0, ncol = length(Y), nrow = n)  
  mu = rep(0, length(Y))  
  
  for(j in 1:n) {  
    for (i in 1:length(Y)) {  
      mu[i] = get_mu(i, sigma = 0.2, mu = mu, Y = Y)  
    }  
    mu_matrix[j,] = mu  
  }  
  
  if (include_matrix) {  
    return(list(means = colMeans(mu_matrix), matrix = mu_matrix))  
  }  
}
```

```

    return(colMeans(mu_matrix))
}

```

Then this function can be used like this. This also includes a plot of the random walk.

```

gibbs_sample_result = gibbs_sample(1000, Y, TRUE)

data$gibbs_samples = gibbs_sample_result$means

ggplot(data) +
  geom_line(aes(x = X, y = Y), color = "#C70039") +
  geom_line(aes(x = X, y = gibbs_samples), color = "#581845") +
  labs(title = "Dependence from Concentration on Day of Measurement", y =
        "Concentration", x = "Day of Measurement", color = "Legend") +
  theme_minimal()

```

If a trace plot is needed, use this:

```

df = data.frame(
  iteration = seq(from = 1, to = length(gibbs_sample_result$matrix[,50]), by = 1),
  mu_n = gibbs_sample_result$matrix[,50])

ggplot(df) +
  geom_line(aes(x = iteration, y = mu_n), color = "#C70039") +
  labs(title = "Trace of mu_n", y = "Value of mu_n", x = "Iteration",
        color = "Legend") +
  theme_minimal()

```

12 Hypothesis Testing

12.1 Defining a Test-Statistics

Pay attention how **exactly** this on is defined, as here for instance Y and \hat{Y} mixed.

$$T = \frac{\hat{Y}(X_b) - \hat{Y}(X_a)}{X_b - X_a}, \text{ where } X_b = \operatorname{argmax}_X Y(X), X_a = \operatorname{argmin}_X Y(X)$$

```

data = data.frame(X = lottery$Day_of_year, Y = lottery$Draft_No)

test_statistics = function(X, Y, Y_hat) {

  b_index = which.max(Y)
  a_index = which.min(Y)

  return((Y_hat[b_index] - Y_hat[a_index]) / (X[b_index] - X[a_index]))
}

```

12.2 Non-Parametric Bootstrap

```

f = function(data, ind) {
  data1 = data[ind,]

```



```

model = loess(Draft_No ~ Day_of_year, data1)

T_value =
  test_statistics(data1$Day_of_year, data1$Draft_No, Y_hat = model$fitted)

return(T_value)
}

# T(D) for the original data
data$Y_hat = loess(Draft_No ~ Day_of_year, lottery)$fitted
T_value_original = test_statistics(data$X, data$Y, data$Y_hat)

# T for the bootstrapped samples
nonparam_bootstrap =
  boot(lottery, statistic = f, R = 2000, parallel = "multicore")
p_value_original = mean(nonparam_bootstrap$t > 0)

```

12.3 Density of non-parametric T-Values

```

df = data.frame(nonparam_bootstrap$t)

ggplot(df) +
  geom_density(aes(x = nonparam_bootstrap.t, color = "black",
    fill = "#dedede", alpha = 0.25)) +
  geom_vline(aes(xintercept = T_value_original), color = "orange") +
  labs(title = "Density of non-parametric T-Values",
    y = "Density", x = "T-Value", color = "Legend") +
  theme_minimal()

```

12.4 Defining the Hypothesis Test (Permutation Test)

```

test_hypothesis = function (data_input, statistics, B = 2000, T_org) {

  t_values = rep(NA, B)

  for (i in 1:B) {

    ind = sample(1:nrow(data_input))
    data_input$X = data_input$X[ind]

    model = loess(Y ~ X, data_input)

    t_values[i] = statistics(data_input$X,
      data_input$Y, Y_hat = model$fitted)
  }

  return(sum(abs(t_values) >= abs(T_org))/B)
}

p_value_permutated = test_hypothesis(data, test_statistics, 2000, T_value_original)

```

12.5 Crude Estimate of the Power

Question: Make a crude estimate of the power of the test constructed in Step 4:

- Generate (an obviously non-random) dataset with $n = 366$ observations by using same X as in the original data set and $Y(x) = \max(0, \min(\alpha x + \beta, 366))$, where $\alpha = 0.1$ and $\beta \sim N(183, sd = 10)$.
- Plug these data into the permutation test with $B = 200$ and note whether it was rejected.
- Repeat Steps 5a-5b for $\alpha = 0.2, 0.3, \dots, 10$.

What can you say about the quality of your test statistics considering the value of the power?

```
simulate_data = function(X, Y_hat, hypothesis = test_hypothesis,
                        statistics = test_statistics, alpha = 0.1,
                        beta_mean = 183, beta_sd = 10, b = 200, limit = 366) {

  artificial = function(X, alpha) {
    beta = rnorm(n = nrow(lottery), mean = beta_mean, sd = beta_sd)
    return(max(0, min(alpha * X + beta, limit)))
  }

  X_dataframe = X
  Y_dataframe = sapply(X, artificial, alpha)

  Y_hat = loess(Y_dataframe ~ X_dataframe)$fitted

  Y_hat_dataframe = Y_hat

  data_artificial = data.frame(X_dataframe, Y_dataframe, Y_hat_dataframe)
  colnames(data_artificial) = c("X", "Y", "Y_hat")

  T_val = test_statistics(data_artificial$X, data_artificial$Y, data_artificial$Y_hat)

  return(test_hypothesis(data_artificial, statistics, b, T_val))
}

alphas = seq(from = 0.1, to = 10.0, by = 0.1)

no_cores = detectCores()
cl = makeCluster(no_cores)

clusterExport(cl, list("simulate_data", "lottery", "test_hypothesis",
                      "test_statistics", "data"))

simulated_p_values =
  parSapply(cl, alphas, FUN = function(alpha) {
    simulate_data(alpha = alpha, X = lottery$Day_of_year, Y_hat = data$Y_hat)
  })

stopCluster(cl)
```

Plot for p-values with respect to α and p-value line:

```
df = data.frame(alphas, simulated_p_values)

ggplot(df)+
  geom_point(aes(x = alphas, y = simulated_p_values), color = "black",
```

```

    fill = "#dedede", shape = 21) +
  geom_hline(aes(yintercept = 0.05), color = "orange") +
  labs(title = "p-values depending on alpha",
       y = "p-value", x = "Alpha", color = "Legend") +
  theme_minimal()

```

Calculating the actual power is done by:

```
1 - mean(simulated_p_values > 0.05)
```

13 Bootstrap with Bias-Correction

Doing a normal bootstrap.

```

f_prices = function(data, ind) {
  data1 = data[ind,]

  return(mean(data1$Price))
}

house_bootstrap = boot(prices, statistic = f_prices, R = 2000,
                      parallel = "multicore")

plot(house_bootstrap)

```

The estimate of the mean price is:

```

bootstrap_mean_price = mean(house_bootstrap$t)
print(bootstrap_mean_price)

```

The bootstrap-bias-correction is given by the following values. The first one is the bias-correction and the second one is the bias corrected mean.

```

prices_mean = mean(prices$Price)
print(prices_mean - bootstrap_mean_price)
print(2 * prices_mean - bootstrap_mean_price)

```

The variance of the mean is given by:

```

bootstrap_variance_price = as.numeric(var(house_bootstrap$t))
# bootstrap_mean_price = 1 / (B-1) * sum((house_bootstrap$t -
#                                     mean(house_bootstrap$t))^2)

print(bootstrap_variance_price)

```

Now we will create the 95% confidence intervals.

```

confidence_interval = boot.ci(house_bootstrap)
print(confidence_interval)

```

As this output does not include the BCa, we will print the intervals manually. Note that the last two values in each row represent the confidence interval.

```

confidence_interval$percent
confidence_interval$bca
confidence_interval$normal

```

14 Variance Estimation using Jackknife

```
f_prices_jackknife = function(ind, data) {
  data1 = data[-ind,]
  return(mean(data1$Price))
}

# First create the statistics using jackknife
n = length(prices$Price)

indices = seq(from = 1, to = n, by = 1)
jackknife_statistics = sapply(indices, f_prices_jackknife, prices)

# For the variance estimate we will first calculate Ti_star
Ti_star = sapply(jackknife_statistics, FUN = function(tdi, n, prices_mean) {
  return(n * prices_mean - ((n - 1) * tdi))
}, n = n, prices_mean = prices_mean)

# Now we calculate J_T
J_T = mean(Ti_star)

# And now we can calculate the Variance
variance_jackknife = 1 / (n * (n - 1)) * sum((Ti_star - J_T)^2)

print(variance_jackknife)
```

15 Confidence Intervals with Respect to Length and Location

```
bootstrap_estimated_mean_corrected = (2 * prices_mean - bootstrap_mean_price)
```

The length of the CI is:

```
print(confidence_interval$percent[5] - confidence_interval$percent[4])
```

The mean is located the following “percent” of the CI range.

```
(bootstrap_estimated_mean_corrected - confidence_interval$percent[4]) /
  (confidence_interval$percent[5] - confidence_interval$percent[4]) * 100
```

```
ggplot(prices) +
  geom_histogram(aes(x = prices$Price),
                 color = "#000000", fill = "#dedede") +
  annotate("rect", xmin=confidence_interval$percent[4],
           xmax=confidence_interval$percent[5], ymin=0, ymax=Inf,
           alpha=0.5, fill="#86b4ff") +
  geom_vline(aes(xintercept = bootstrap_estimated_mean_corrected),
            color = "orange") +
  labs(title = "Histogram of Prices",
       y = "Frequency",
       x = "Prices", color = "Legend") +
  theme_minimal()
```

16 Genetic Algorithms

First define the function to minimize or maximize.

```
f = function(x) {  
  left = x^2/exp(x)  
  exponent = (-9*sin(x))/(x^2+x+1)  
  right = 2*exp(exponent)  
  return(left-right)  
}
```

Plotting this function:

```
sequence = seq(from = 0, to = 30, by = 0.1)  
f.sequence = f(sequence)  
df = data.frame(sequence, f.sequence)  
  
ggplot(df) +  
  geom_line(aes(x = sequence, y = f.sequence), color = "#C70039") +  
  labs(title = "f(x)", y = "f(x)", x = "x") +  
  theme_minimal()
```

Max value at x.

```
print(sequence[which.max(f.sequence)])
```

And y or respect f(x) for that:

```
print(max(f.sequence))
```

Then the crossover function, might be different though.

```
crossover = function(x, y) return((x+y)/2)
```

Then the mutation.

```
mutate = function(x) return((x^2)%%30)
```

Initial Population.

```
X = seq(from = 0, to = 30, by = 5)
```

Corresponding values.

```
Values = f(X)
```

And then the algorithm itself. Keep in mind that this one **does not save the overall best individual**.

```
genetic = function(X, Values, maxiter = 100, mutprob = 0.05) {  
  
  best_individual = NaN  
  
  for (i in 1:maxiter) {  
  
    # 1)  
    parents = sample(1:length(X), size = 2)  
  
    # 2) I don't know why we should order here!?  
    victim = which.min(Values)
```

```

# 3)
child = crossover(X[parents][1], X[parents][2])
if (mutprob > runif(n = 1, min = 0, max = 1)) {
  child = mutate(child)
}

# 4)
X[victim] = child
Values[victim] = f(child)

# 5)
best_index = which.max(Values)
best_individual = list(cx = X[best_index], cy = Values[best_index])
}
return(list(best = best_individual, population = X))
}

best_individual = genetic(X, Values, 100, 0.05)$best

```

Plot the best individual.

```

df = data.frame(sequence, f.sequence)
best_individual = as.data.frame(best_individual)

ggplot(df) +
  geom_line(aes(x = sequence, y = f.sequence), color = "#C70039") +
  geom_point(aes(x = cx, y = cy), data = best_individual, color = "black",
    fill = "#7BA9FF", shape = 21, size = 3, stroke = 2) +
  labs(title = "f(x)", y = "f(x)", x = "x") +
  theme_minimal()

```

Plot the initial distribution.

```

df = data.frame(sequence, f.sequence)
initial.population = data.frame(X, Values)

ggplot(df) +
  geom_line(aes(x = sequence, y = f.sequence), color = "#C70039") +
  geom_point(aes(x = X, y = Values), data = initial.population, color = "black",
    fill = "#DAF7A6", shape = 21, size = 3, stroke = 2) +
  labs(title = "f(x)", y = "f(x)", x = "x") +
  theme_minimal()

```

Six grid plot with different settings.

```

set.seed(12345)

genetic1 = genetic(X, Values, 10, 0.1)
genetic2 = genetic(X, Values, 10, 0.5)
genetic3 = genetic(X, Values, 10, 0.9)
genetic4 = genetic(X, Values, 100, 0.1)
genetic5 = genetic(X, Values, 100, 0.5)
genetic6 = genetic(X, Values, 100, 0.9)

population1 = data.frame(genetic1$population, f(genetic1$population))
best1 = as.data.frame(genetic1$best)

```

```

population2 = data.frame(genetic2$population, f(genetic2$population))
best2 = as.data.frame(genetic2$best)

population3 = data.frame(genetic3$population, f(genetic3$population))
best3 = as.data.frame(genetic3$best)

population4 = data.frame(genetic4$population, f(genetic4$population))
best4 = as.data.frame(genetic4$best)

population5 = data.frame(genetic5$population, f(genetic5$population))
best5 = as.data.frame(genetic5$best)

population6 = data.frame(genetic6$population, f(genetic6$population))
best6 = as.data.frame(genetic6$best)

p1 = ggplot(df) +
  geom_line(aes(x = sequence, y = f.sequence), color = "#C70039") +
  geom_point(aes(x = genetic1.population, y = f.genetic1.population.),
    data = population1, color = "black",
    fill = "#FFC300", shape = 21, size = 2, stroke = 1) +
  geom_point(aes(x = cx, y = cy), data = best1, color = "black",
    fill = "#7BA9FF", shape = 21, size = 2, stroke = 1) +
  labs(title = "maxiter = 10, mutprob = 0.1", y = "f(x)", x = "x") +
  theme_minimal()

grid.arrange(p1, p2, p3, p4, p5, p6, nrow = 3)

```

Another implementation could look like this:

```

to_base_10 = function(v) {
  return(Reduce(function(s,r) {s*2+r}, v))
}

transform_to_interval = function(u, a = -1, b = 2, m = 15) {
  return(a + u * ((b - a)/(2^m - 1)))
}

chrom_to_x = function(chromosome) {
  return(transform_to_interval(to_base_10(chromosome)))
}

chrom_to_y = function(chromosome) {
  return(f(transform_to_interval(to_base_10(chromosome))))
}

tournament_selection = function(population) {

  # Select fighting partners / groups
  fighters = sample(1:nrow(population), 4)

  # Get their values
  fighter_A = population[fighters[1],]
  fighter_B = population[fighters[2],]
  fighter_C = population[fighters[3],]

```

```

fighter_D = population[fighters[4],]

# Let them fight, group A
if (chrom_to_y(fighter_A[2:length(fighter_A)]) <
    chrom_to_y(fighter_A[2:length(fighter_A)])) {
  winner_group_A = fighter_A
  loser_group_A = fighter_B
}
else {
  winner_group_A = fighter_B
  loser_group_A = fighter_A
}

# Let them fight, group B
if (chrom_to_y(fighter_A[2:length(fighter_C)]) <
    chrom_to_y(fighter_A[2:length(fighter_D)])) {
  winner_group_B = fighter_C
  loser_group_B = fighter_D
}
else {
  winner_group_B = fighter_D
  loser_group_B = fighter_C
}

# Return them, first two are winners, last two are loser
results = data.frame()
results = rbind(results, winner_group_A)
results = rbind(results, winner_group_B)
results = rbind(results, loser_group_A)
results = rbind(results, loser_group_B)

return(results)
}

# Generate a population of size n with m digits as genes
generate_population = function(n, m = 15) {

  population = data.frame()

  for (i in 1:n) {
    individual = c(i, ifelse(round(rnorm(m, mean = 0, sd = 1)) > 0, 1, 0))
    population = rbind(population, individual)
  }

  colnames(population) = c("index", as.character(seq(from = 1, to = m, by = 1)))
  return(population)
}

# Does a crossover where the children take one half of each parent
crossover = function(chrom_A, chrom_B) {
  chrom_A = unlist(chrom_A)
  chrom_B = unlist(chrom_B)

```



```

cut = round(length(chrom_A)/2)

child_A = c(chrom_A[2:cut], chrom_B[(cut+1):length(chrom_B)])
child_B = c(chrom_B[2:cut], chrom_A[(cut+1):length(chrom_B)])

return(list(child_A, child_B))
}

# Mutates a single bit based on prob
mutate_with_prob = function(x, prob) {
  if (runif(n = 1) < prob) {
    return(ifelse(x == 0, 1, 0))
  }
  else {
    return(x)
  }
}

# Mutates on average prob genes, each gene has a prob of prob/length to mutate
mutate = function(chrom, prob = 0.05) {
  chrom = unlist(chrom)
  # To get an average of 0.05 bits mutated
  digit_prob = prob / length(chrom)
  chrom[2:length(chrom)] = sapply(chrom[2:length(chrom)], mutate_with_prob, prob)
  return(chrom)
}

# Helper function to get the best individual from a population
get_best_individual = function(population) {
  index_best = which.min(apply(population[,2:ncol(population)], 1, chrom_to_y))
  return(population[index_best,])
}

# Use all pre defined methods to actually implement the genetic algorithm
genetic_algorithm = function(population_size = 55, mutprob = 0.05, runs = 100) {

  # Generate population
  population = generate_population(population_size)

  best_individual = get_best_individual(population)

  best_individual_iteration = 0

  for (i in 1:runs) {

    # Selection
    fighters = tournament_selection(population)

    # Create children of winners
    children = crossover(fighters[1,], fighters[2,])

    # Mutate the new individuals
    children[[1]] = mutate(children[1], prob = mutprob)

```

```

children[[2]] = mutate(children[2], prob = mutprob)

# Replace the losers with the new winners
population[fighters[3,1], 2:ncol(population)] = children[[1]]
population[fighters[4,1], 2:ncol(population)] = children[[2]]

# Save the overall best individual
current_best = get_best_individual(population)
if (chrom_to_y(current_best[2:length(current_best)]) <
    chrom_to_y(best_individual[2:length(best_individual)])) {
  best_individual = current_best
  best_individual_iteration = i
}
}

return(list(best = best_individual, population = population,
            iteration = best_individual_iteration))
}

```

Call like this:

```

results0077 = genetic_algorithm(population_size = 55, mutprob = 0.0077, runs = 100)
print(results0077$best)
print(results0077$iteration)

X = apply(results0077$population[,2:ncol(results0077$population)], 1, chrom_to_x)
Y = f(X)

X_best = chrom_to_x(results0077$best[2:length(results0077$best)])
Y_best = f(X_best)

population = data.frame(X, Y)
best = data.frame(X_best, Y_best)

ggplot(df) +
  geom_line(aes(x = sequence, y = f.sequence), color = "#C70039") +
  geom_point(aes(x = population$X, y = population$Y),
             data = population, color = "black", fill = "#FFC300", shape = 21,
             size = 2, stroke = 1) +
  geom_point(aes(x = best$X, y = best$Y), data = best, color = "black",
             fill = "#7BA9FF", shape = 21, size = 2, stroke = 1) +
  labs(title = "mutprob = 0.0077", y = "f(x)", x = "x") +
  theme_minimal()

```

17 Expectation-Maximization Algorithm (EM)

Plot for two lines in one graph.

```

ggplot(data) +
  geom_line(aes(x = X, y = Z, color = "Z")) +
  geom_line(aes(x = X, y = Y, color = "Y")) +
  labs(title = "Dependence of Z and Y versus X", y = "Z/Y", x = "X") +
  scale_color_manual(values = c("#C70039", "#581845")) +

```

```
theme_minimal()
```

In general, perform:

- E-Step (Expectation)
- M-Step (Maximization)

17.1 Expectation-Step (E-Step)

For the E-Step, we write down:

$$Q(\theta, \theta^k) = E[\mathcal{L}(\theta|Y, Z)|\theta^k, Y]$$

where θ is the unknown parameter (for instance λ). Then we take the log likelihood and estimate the latent parameters, for instance by using the expected value. This depends on the distribution. The log likelihood is usually the log of the product of the given distributions.

It could look like this:

$$\mathcal{L}(\lambda|Y, Z) \sim \ln \left(\prod_{i=1}^n \frac{X_i}{\lambda} e^{-X_i/\lambda} * \frac{X_i}{2\lambda} e^{-X_i/2\lambda} \right)$$

And then taking the expected value for the missing variables.

$$E[\mathcal{L}(\lambda|Y, Z)] = -n * \ln(2\lambda^2) + 2 \sum_{i=1}^n \ln(X_i) - \sum_{i=1}^n \frac{X_i Y_i}{\lambda} - E \left[\sum_{i=1}^n \frac{X_i Z_i}{2\lambda} \right]$$

So the E-Step concludes with:

$$Q(\lambda, \lambda^k) = E[\mathcal{L}(\lambda|Y, Z)] = -n * \ln(2\lambda^2) + 2 \sum_{i=1}^n \ln(X_i) - \sum_{i=1}^n \frac{X_i Y_i}{\lambda} - \sum_{i=1}^{\beta} \frac{X_i Z_i}{2\lambda} - (n - \beta) \frac{\lambda_k}{\lambda}$$

17.2 Maximization-Step (M-Step)

Usually this can be done by taking the derivative with respect to the parameters to estimate. There might be different ways to do the maximization though.

$$\frac{\partial Q(\lambda, \lambda^k)}{\partial \lambda} = -\frac{2n}{\lambda} + \sum_{i=1}^n \frac{X_i Y_i}{\lambda^2} + \frac{1}{2} \sum_{i=1}^{\beta} \frac{X_i Z_i}{\lambda^2} + (n - \beta) \frac{\lambda_k}{\lambda^2}$$

And then simply solve for the parameter that maximizes.

$$\lambda = \frac{\sum_{i=1}^n X_i Y_i + \frac{1}{2} \sum_{i=1}^{\beta} X_i Z_i + (n - \beta) \lambda_k}{2n}$$

17.3 EM-Implementation

The implementation can look like this:

```
# To keep it simple we will not pass the data to the function.
lambda_estimate_em = function (input_iterations = 100, input_threshold = 0.0001) {

  iterations_max = input_iterations
  iterations = 0
  threshold = input_threshold

  n = nrow(data)
  lambdas = c()
  lambda = NaN
  lambda_k = 100

  Z = data$Z[!is.na(data$Z)]
  A = data$Z[is.na(data$Z)]

  Z_index = which(!is.na(data$Z))
  A_index = which(is.na(data$Z))

  X = data$X[Z_index]
  X_A = data$X[A_index]

  beta = length(Z)

  for (i in 1:iterations_max) {
    iterations = iterations + 1

    # E/M-Step
    lambda = lambda_k
    lambda_k = (sum(data$X * data$Y) + 0.5 * sum(X * Z) +
                ((n - beta) * lambda_k)) / (2 * n)
    lambdas = c(lambdas, lambda_k)

    if (abs(lambda_k - lambda) < threshold ) break
  }

  return(list(lambdas, iterations))
}

em_result = lambda_estimate_em()
lambda_result = em_result[[1]][em_result[[2]]]

print(em_result)
```

If we have to plot the expected values, take the formula from that from the distribution, for instance for the exponential one it would look like this.

$$E[Y] = \frac{\lambda}{X_i}, \quad E[Y] = \frac{2\lambda}{X_i}$$

```

data$Y_E = (lambda_result)/data$X
data$Z_E = (2*lambda_result)/data$X

ggplot(data) +
  geom_line(aes(x = X, y = Z, color = "Z")) +
  geom_line(aes(x = X, y = Y, color = "Y")) +
  geom_line(aes(x = X, y = Z_E, color = "E[Z]")) +
  geom_line(aes(x = X, y = Y_E, color = "E[Y]")) +
  labs(title = "Dependence of Z, Y, E[Z] and E[Y] versus X", y = "Z/Y", x = "X") +
  scale_color_manual(values = c("#581845", "#FFC300", "#C70039", "#6091ec")) +
  theme_minimal()

```

18 Uniform Sampling

```

rng_runif = function(a, m, x_zero, nmax) {

  if (!(a >= 0 && a < m)) stop("a in [0, m) is required")
  if (nmax%%1 != 0) stop("nmax has to be an integer")

  storage = vector(mode = "numeric", length = nmax+1)
  storage[1] = x_zero

  for (i in 1:(nmax-1)) {
    storage[i+1] = ((a * storage[i]) %% m)
  }

  return(storage[2:length(storage)]/m)
}

```

19 Normal Sampling

```

rng_rnorm = function(n, a, m) {
  if (!(a >= 0 && a < m)) stop("a in [0, m) is required")

  n_half = ceiling(n/2)

  storage = vector(mode = "numeric", length = n)

  storage_theta = rng_runif(a = a, m = m, x_zero = 123456789, nmax = n_half) * 2 * pi
  storage_d = rng_runif(a = a, m = m, x_zero = 12345, nmax = n_half)

  for (i in 1:n_half) {
    storage[2*i] = sqrt(-2 * log(storage_d[i])) * cos(storage_theta[i])
    storage[2*i+1] = sqrt(-2 * log(storage_d[i])) * sin(storage_theta[i])
  }

  return(storage)
}

```

```

rng_rnorm = function(n = 1) {

  theta = runif(ceiling(n/2), 0, 2*pi)
  d = runif(ceiling(n/2))

  res1 = sqrt(-2 * log(d)) * cos(theta)
  res2 = sqrt(-2 * log(d)) * sin(theta)

  res = c(res1, res2)

  return(res[1:n])
}

```

20 Pareto Distribution

```

# PDF
dpar = function(x, alpha = 3, beta = 1) {
  if (x > beta && alpha > 0 && beta > 0) {
    return(alpha * beta^alpha / (x^alpha + 1))
  }
  return(0)
}

# CDF
ppar = function(x, alpha = 3, beta = 1) {
  if (x <= 1) return(0)
  return(1 - (beta^alpha/x^alpha))
}

# Inverse
qpar = function(x, alpha = 3, beta = 1) {
  return(beta/((1-x)^(1/alpha)))
}

# Random
rpar = function(n = 1, alpha = 3, beta = 1) {
  quantiles = runif(n = n, min = 0, max = 1)
  rpars = sapply(X = quantiles, FUN = qpar, alpha, beta)
  return(rpars)
}

```

21 Geometric Sampling

```

rng_rgeom = function(n = 1, p = 0.5) {

  samples = numeric(n)

  for (i in 1:n) {
    cnt = 0

```

```

    while (sample(c(0,1), 1 ,replace=FALSE, prob=c(1-p,p)) == 0) {
      cnt = cnt +1
    }
    samples[i] = cnt
  }
  return(samples)
}

samples_geom = rng_rgeom(10000)

```

22 Parabolic Interpolation

Interpolate an interval.

```

interpolate = function(A, X, func, gradient = NULL) {

  # Interpolation function which is so be used
  f_tilde = function(x, a0, a1, a2) a0 + a1 * x + a2 * x^2

  # Error function, here MSE
  f_error = function(A., X. = X, func. = func) {
    return(sum((func.(X.) - f_tilde(X., A.[1], A.[2], A.[3]))^2))
  }

  # Optimize f_error for parameters in A
  res = optim(A, f_error, gradient, method = "BFGS")
  #res = optim(A, f_error, gradient, method = "CG")

  # Now define with optimized parameters
  f_tilde = function(x) res$par[1] + res$par[2] * x + res$par[3] * x^2

  # Return parameters and function
  return(list(A = res$par, f_tilde = f_tilde))
}

```

Interpolate a whole function.

```

f_approximate = function(func_target, bins, A_init, func_target_gradient = NULL) {

  # Initialize interval length and return matrix
  upper_boundary = 0
  interval_length = 1/bins
  res = matrix(0, nrow = bins, ncol = 5)
  colnames(res) = c("lower_boundary", "upper_boundary", "a0", "a1", "a2")

  # Approximate for each bin
  for (i in 1:bins) {
    lower_boundary = upper_boundary
    upper_boundary = lower_boundary + interval_length

    # We rely on three known points
    known_values = c(lower_boundary, (lower_boundary + upper_boundary)/2,
                     upper_boundary)
  }
}

```

```

# Now we optimize for this interval using the previous function
interpolated = interpolate(func = func_target, X = known_values, A = A_init,
                           gradient = func_target_gradient)$A

# And fill in the matrix with all necessary values
res[i, 1] = lower_boundary
res[i, 2] = upper_boundary
res[i, 3:5] = interpolated
}

return(res)
}

```

Call.

```

f1_res = f_approximate(f1, bins = 1000, rep(0.001, 3))

f_tilde = function(x, a0, a1, a2) a0 + a1 * x + a2 * x^2

df$f1_y_interpolated = sapply(sequence, FUN = function(x) {
  target_row = f1_res[x >= f1_res[,1] & x < f1_res[,2]]
  return(-f_tilde(x, target_row[3], target_row[4], target_row[5]))
})

```

23 Miscellaneous Plots

23.1 Histogram

General rule for n datapoints: \sqrt{n} bars.

```

ggplot(sample) +
  geom_histogram(aes(x = Population), bins = nrow(sample), color = "black",
                 fill = "#C70039") +
  ggtitle("Histogram of selected cities")

```

23.2 Histogram with Mean

```

prices_mean = mean(prices$Price)
print(prices_mean)

ggplot(prices) +
  geom_histogram(aes(x = prices$Price, y = ..density..), color = "black", fill = "#dedede") +
  geom_vline(aes(xintercept = prices_mean), color = "#FFC300") +
  labs(title = "Histogram of Prices",
       y = "Density",
       x = "Prices", color = "Legend") +
  theme_minimal()

```


23.3 Simple X/Y Plot

```
ggplot(data) +  
  geom_line(aes(x = X, y = Y), color = "#c70039") +  
  labs(title = "Dependence from Concentration on Day of Measurement",  
        y = "Concentration", x = "Day of Measurement", color = "Legend") +  
  theme_minimal()
```

23.4 Variance Plot

```
for (i in 1:length(v)) {  
  Xi = v[1:i]  
  X$vec_customvar[[i]] = custom_variance(as.vector(Xi))  
}  
  
ggplot(X[2:nrow(X),]) +  
  geom_point(aes(x = index, y = value, colour = "Difference")) +  
  geom_point(aes(x = index, y = vec_myvar, colour = "my_var()")) +  
  geom_point(aes(x = index, y = vec_var, colour = "var()")) +  
  geom_point(aes(x = index, y = vec_customvar, colour = "custom_variance()")) +  
  labs(title = "Difference in Variance", y = "Variance",  
        x = "Sequence", color = "Legend") +  
  scale_color_manual(values = c("#17202A", "#C70039", "#407AFF", "#FFC300")) +  
  scale_x_log10() +  
  theme_minimal()
```

23.5 Scatterplot With Geom Smoother

```
ggplot(lottery) +  
  geom_point(aes(x = Day_of_year, y = Draft_No), color = "black",  
             fill = "#dedede", shape = 21) +  
  geom_smooth(mapping = aes(x = Day_of_year, y = Draft_No),  
             method = "loess", size = 1.5, color = "#000000") +  
  labs(title = "Day of Year VS Draft Number",  
        y = "Draft Number", x = "Day of Year", color = "Legend") +  
  theme_minimal()
```

24 Useful Code Snippets

24.1 RMarkdown Setup

```
# ---  
#title: "Computational Statistics Summary"  
#author: "Maximilian Pfundstein"  
#date: "`r Sys.Date()`"  
#output:  
#  html_document:  
#    df_print: paged
```

```
#   toc: true
#   toc_float: true
#   number_sections: true
# pdf_document:
#   toc: true
#   toc_depth: 3
#   number_sections: true
# ---
```

24.2 Knitr options

```
{r setup, include=FALSE}
#knitr::opts_chunk$set(echo = TRUE, cache = TRUE, include = TRUE, eval = FALSE,
#                        fig.pos = 'H')
```

24.3 Including Source Code

```
{r, ref.label=knitr::all_labels(), echo = TRUE, eval = FALSE, results = 'show'}
```