1. Exercise 1

The aim of this exercise is to fit a linear regression model analysing the relationship between the dependent variable 'ascore' a measure of attitude towards a women's role in child care, and a variety of independent variables extracted from the British Household Panel Survey (BHPS) data. A full list of all variables used within this study and subsequent coding can be found below in **Table 1**. The extracted British Household Panel Survey (BHPS) data consists of a sample of 659 women aged between 16 and 44 who were childless when interviewed in 'Wave A', in 1991. Note, 'Wave C' corresponds to 1996.

Table 1. Description of the Variables Used within the Study

Label	Name	Coding
Dependent Variable		
Family role score at Wave A	ascore	From 6 to 30
Independent Variables		
Age at data of interview for Wave A	aage	In years
Economic activity at Wave A	aecact	1 = employed full-time 2 = employed part-time 3 = other 4 = family care
Became a parent between Waves A and C	bepac	0 = no 1 = yes
Whether had co-resident partner at Wave A	apart	0 = no 1 = yes
Highest educational qualification at Wave A	aeduc	1 = degree or higher 2 = GCE A-levels or nursing qualification 3 = GCE O-levels or equivalent 4 = CSE, apprenticeship or other 5 = none
Mother not working when respondent was 14	amumnw	0 = yes 1 = no

1.1 Exercise A

Numerical Variable Descriptive Statistics

Descriptive statistics for the variables can be found below in **Table's 2-7**. For numerical variables, the summary statistics calculated were the mean, variance, and standard deviation.

```
# reading the data into the r environment
BHPS <- read.table("BHPS.dat", header = T)
```

```
BHPS
# Using the summary() function to produce summary statistics
summary(BHPS$aage)
summary(BHPS$ascore)
# Using the var() and sd() functions to obtain variance and
standard deviation
var(BHPS$aage)
var(BHPS$ascore)
sd(BHPS$aage)
sd(BHPS$ascore)
```

Table 2. Mean, Standard Deviation and Variance of Numerical Variables

Variable	mean	Standard deviation	Variance
Aage	25.47	6.78	45.96
Ascore	21.17	3.67	13.46

Categorical Variable Descriptive Statistics

The categorical variables were first converted to factors with the as.factor function(). The provided summary statistics are the count and proportion for each category. The table() function is used to count the occurrence of each category for the desired categorical variable. The prop.table() function is used to convert these stored counts to a proportion. These proportions are rounded to three decimal places and multiplied by 100 to produce a percentage.

```
# Preparing factors
Aecact <- as.factor(BHPS$aecact)
Bepac <- as.factor(BHPS$bepac)
Apart <- as.factor(BHPS$apart)
Aeduc <- as.factor(BHPS$aeduc)
Amumnw <- as.factor(BHPS$amumnw)
```

Aecact

```
# Using the table() and prop.table() function to calculate the
proportions
# Aecact Summary Statistics
Aecact_table <- table(Aecact)
Aecact_table
round(prop.table(Aecact_table), 3) * 100
```

Table 3. 'Aecact' Categorical Frequencies

	Categories			
	Employed full-time (1)	Employed part-time(2)	Other (3)	Family Care(4)
Count	485	36	125	13
Proportion (%)	73.6	5.5	19.0	2.0

Bepac

```
# Bepac Summary Statistics
Bepac_table <- table(Bepac)
Bepac_table
round(prop.table(Bepac_table), 3) * 100
```

Table 4. 'Bepac' Categorical Frequencies

	Categories	
	No (0)	Yes (1)
Count	574	85
Proportion (%)	87.1	12.9

Apart

```
# Apart Summary Statistics
Apart_table <- table(Apart)
Apart_table
round(prop.table(Apart_table), 3) * 100
```

Table 5. 'Apart' Categorical Frequencies

	Categories	
	No (0)	Yes (1)
Count	364	295
Proportion (%)	55.2	44.8

Aeduc

```
# Aeduc Summary Statistics
Aeduc_table <- table(Aeduc)
Aeduc_table
round(prop.table(Aeduc_table), 3) *  100
```

Table 6. 'Aeduc' Categorical Frequencies

	Categories				
	Degree or higher (1)	GCE A-levels or nursing qualification (2)	GCE O-levels or equivalent (3)	CSE, apprenticeship or other (4)	None (5)
Count	193	147	217	72	30
Proportion (%)	29.3	22.3	32.9	10.9	4.6

Amumnw

```
# Amumnw Summary Statistics
Amumnw_table <- table(Amumnw)
Amumnw_table
round(prop.table(Amumnw_table), 3) * 100
```

Table 7. 'Amumnw' Categorical Frequencies

	Categories	
	Yes (0)	No (1)
Count	244	415
Proportion (%)	37	63

Plotting Dependent Variable (Y) 'ascore' Against the Independent Variable 'Agee' (X)

To explore the relationship between 'ascore' and 'aage' a scatterplot was constructed using the ggplot2 package. ggplot() function requires the data used and the aesthetic, 'aes', where the x and y variables to be plotted are defined. The output of this scatterplot can be found below in **Figure 1**. This figure indicates a negative relationship, as 'aage' increases 'ascore' decreases. However, this relationship appears to be weak.

```
# Installing the package ggplot2
install.packages("ggplot2")
library(ggplot2)


# using ggplot to plot 'agge' and 'ascore
ggplot(BHPS, aes(x=aage, y=ascore)) +
```

Geom_point() is used to create a scatter plot. Within this function the colour and size of the points are set. Additionally, the 'alpha' is set to 0.5 to make the points slightly transparent to identify overlapping points.

```
# Calling geom_point() within ggplot to create a scatter plot
  geom_point(color = "steelblue",
             size = 2.5,
             alpha = 0.5) +
```

geom_smooth() is used to fit a line of best fit in order to visualise the effect of 'agge' on 'ascore'. The 'method' is set to "lm" as this is a linear regression.

```
# Calling geom_smooth() to plot a line of best fit using "lm"
as this is a linear regression

  geom_smooth(method="lm") +
```

scale_x_continuous() changes the scale of the x-axis. The breaks are set to be a sequence between 15 and 45, in increments of 5. The x-axis lower limit is set to 15 and the upper limit to 45.

```
# Calling  scale_x_continuous() to change the scale of the x-
axis
  scale_x_continuous(breaks = seq(15,45,5),
                     limits = c(15,45)) +
```

Labs() is used to assign labels to the x and y axis and to create a title for the plot.

```
# calling labs() within ggplot
  labs(x = "Age at date of interview",
       y = "Family role score",
       title = "Age vs. Family role Score",
       subtitle = "Wave A")
```

Plots for the Dependent Variable (Y) 'ascore' Against Categorical Independent Variables (X)

To explore the relationships between 'ascore' and the categorical variables boxplots were constructed using geom_boxplot(). **Figure 2** below, displaying the boxplot between 'ascore' and 'aecact' identifies the category 'family care', coded as '4' as having the lowest mean family role score.

Aecact

```
# aecact
ggplot(BHPS, aes(x = Aecact, y = ascore)) +

#calling the geom_boxplot() to create a boxplot
  geom_boxplot() +
  ggtitle("boxplot of ascore by Aecact")")
```

Figure 1. Scatter Plot of Age vs Family Role Score

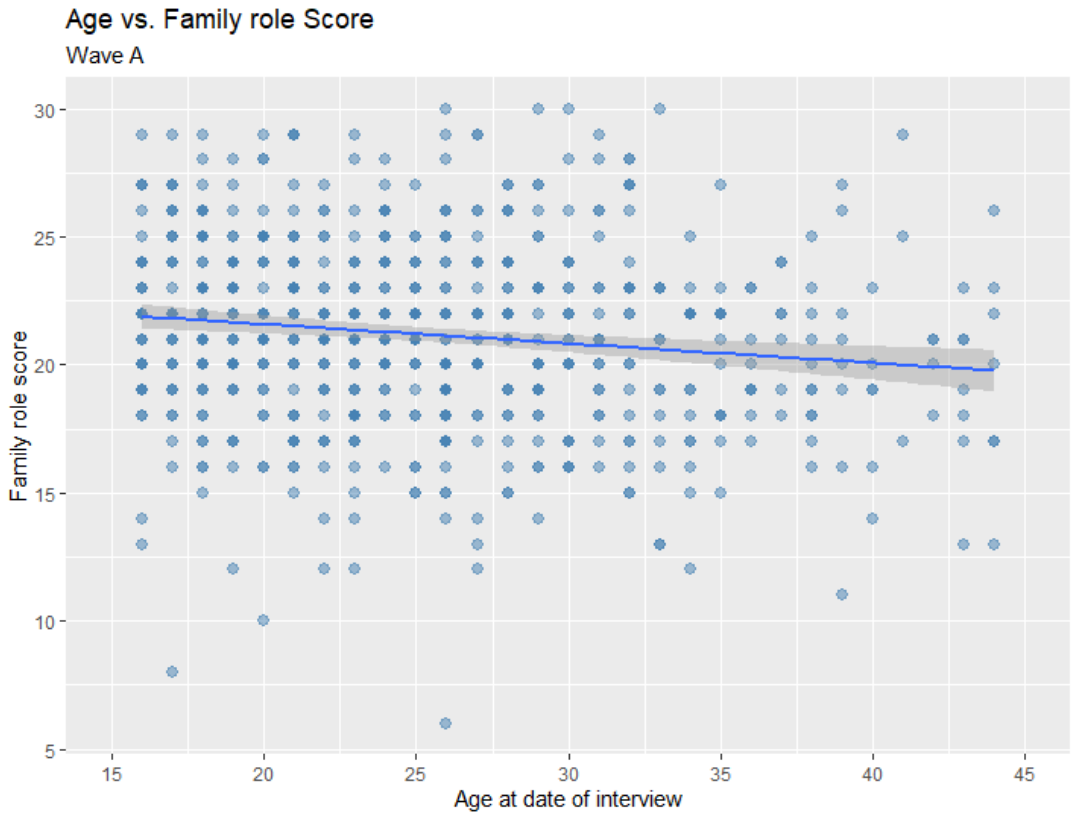
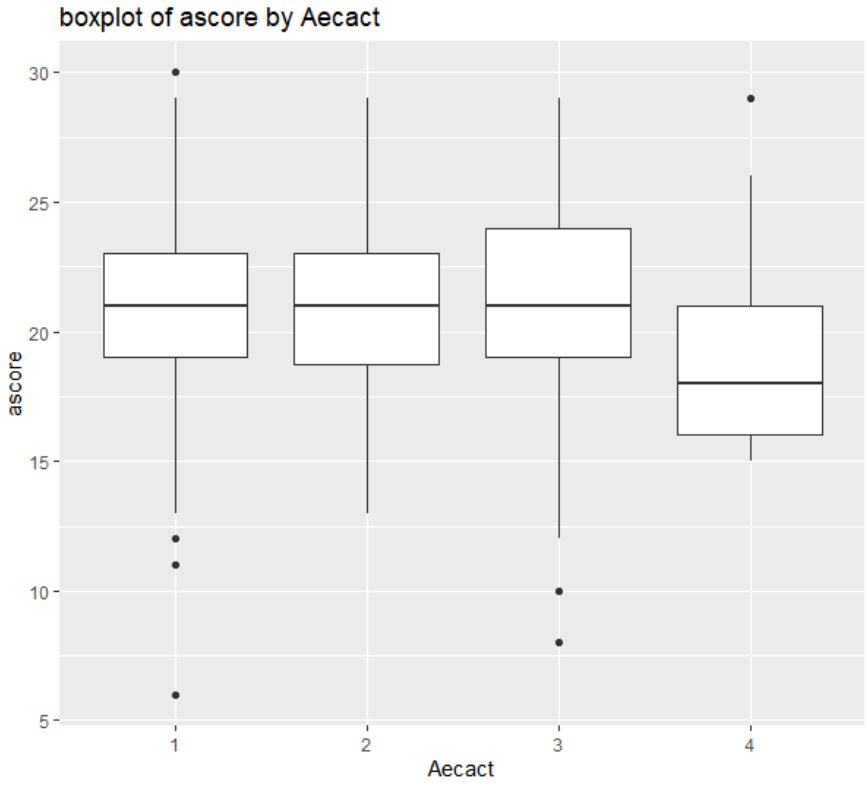


Figure 2. Boxplot of Aecact Vs. Ascore



Bepac

```
# bepac box plot
  ggplot(BHPS, aes(x = Bepac, y = ascore)) +
  geom_boxplot()
```

Apart

```
# apart box plot
  ggplot(BHPS, aes(x = Apart, y = ascore)) +
  geom_boxplot()
```

Aeduc

```
# Aeduc box plot
ggplot(BHPS, aes(x=Aeduc, y = ascore)) +
  geom_boxplot()
```

Amumnw

```
# Amumnw box plot
  ggplot(BHPS, aes(x = Amumnw, y = ascore)) +
  geom_boxplot()
```

1.2 Exercise B

Fitting Forward Selection Model

The Forward Selection method was used to fit a linear regression model with 'ascore' as the dependent variable. This method involves starting with the null model and adding each variable sequentially to identify the most significant variable. This identified variable is then added to the model and the process is repeated until all significant variables are added. Within R this process is automated using the step() function. The step function uses the Akaike Information Criterion (AIC) as criteria for model selection. Within the step function the null model is provided as the first argument, the second argument 'scope' defines the range of models to be examined. In this case the null model and the full model. Finally, the direction is set to "forward". Summary() is called on this final model to obtain the coefficient estimates. The outputs and variables selected by the Forward Selection can be found below in **Table 8**.

```
# Fitting the null model
Null_model <- lm(BHPS$ascore~1)
# Fitting the full Model
Full_model <- lm(BHPS$ascore ~ Aecact + Aeduc + Amumnw + Apart
+ Bepac + BHPS$aage, data = BHPS)
# Using step() to perform forward selection
forward_model <- step(Null_model, scope=list(lower=Null_model,
upper=Full_model), direction = "forward")
# using summary() to view coefficients, significance, and R-
square
summary(forward_model)
```

Table 8. Outputs of the Forward Selection Model

Coefficients	Estimate	Significance
Intercept	23.42	< 2e-16
Amumnw1	1.03	0.000465
BHPS$aage	-0.08	0.000170
Aeduc2	-0.58	0.155894
Aeduc3	-1.17	0.001441
Aeduc4	-1.33	0.007883
Aeduc5	-1.54	0.029432
Bepac1	-0.68	0.101708
Multiple R-squared	Adjusted R-squared	p-value
0.062	0.052	5.383e-07

Forward Model Interpretation

The variables selected by the Forward Selection as being significant explainers of 'ascore' were 'Mother not working when respondent was 14' (Amumnw1), 'Age at date of interview for Wave A' (BHPS$aage), 'Highest educational qualification at Wave A' (Aeduc2-5), and 'Became a parent between Waves A and C' (Bepac1).

The fitted regression model for the forward selection:

$$\hat{y} = 23.42 + 1.03\,Amumnw1 - 0.08\,aage - 0.58\,Aeduc2 - 1.17Aeduc3 - 1.33Aeduc4 - 1.54Aeduc5 - 0.68Bepac1$$

The model suggests, controlling for all other variables within the model respectively, the coefficient for 'Amumnw1' shows that if the respondents mother was working when the respondent was 14 they have an estimated 'ascore' 1.03 higher compared to respondents mothers who were not working. The coefficient 'BHPS$aage' suggests that a one-year increase in the respondents age is associated with a decrease in expected ascore of 0.08. For the variable 'Aeduc' the reference category is category 1, 'degree or higher'. The estimates suggest when the respondent's highest education qualifications at Wave A is equal to 'GCE A-levels or nursing qualification' ,'GCE O-levels or equivalent', 'CSE, apprenticeship or other' or 'none' the respondent has respectively 0.58, 1.17, 1.33 and 1.54 decrease in the expected 'ascore' compared to the reference category. The coefficient for 'bepac' suggests if the respondent became a parent between Waves A and C, they have an expected 'ascore' 0.68 higher compared to respondents who did not become parents.

Fitting Interaction

From the model identified using automatic forward selection it was hypothesised that there may be an interaction present between the respondents age at interview 'aage' and if the respondent became a parent between Waves A and C 'bepac'. Does the effect of age on 'ascore' differ whether the respondent become a parent or not? This interaction was visualised using a grouped bar chart using geom_bar(). Using 'fill' to generate two bars for each age, one for the respondents who became parents and one for respondents who did not. Using position_dodge() ensures these bars are side by side. scale_fill_discrete() is used to change the labels of the legend. As can be seen in **Figure 3**, the effect of 'aage' on 'ascore' does appear to change depending on whether the respondent became a parent.

```
# Visualizing potential Interaction
ggplot(data = BHPS, aes(x=aage, y = ascore, fill =
factor(bepac))) +
  geom_bar(stat = "identity", position=position_dodge()) +
  scale_fill_discrete(name="Became A Parent",
                      breaks = c("0","1"),
                      labels =c("No", "Yes") )
```

Figure 3. Bar Chart of 'Ascore' by 'Aage' Grouped by 'Bepac'

This Interaction was fitted to the forward selection model. The output of this interaction model can be found in **Table 9**. The significance of adding this interaction was tested using a partial t-test with the anova() function as the two models are nested. The outputs of this test can be found below in **Table 10**. The results of including the interaction term show an increase in the Multiple R-squared and Adjusted R-squared respectively by 0.05 and 0.03 compared to the forward model. However the results of the ANOVA suggest the inclusion of this interaction term does not significantly improve the model fit as $p = 0.075$, $p > 0.05$, thus the final model selected should be the simpler, parallel slopes, forward selection model.

```
# testing an interaction within the model between aage and
bepac using *
interaction_model <- lm(BHPS$ascore ~ Amumnw + BHPS$aage +
Aeduc + Bepac + BHPS$aage * Bepac)
# Calling summary of the model to view the coefficients and R-
squared
summary(interaction_model)
# testing the significance of this interaction using ANOVA
anova(forward_model,interaction_model)
```

Table 9. Outputs of the Interaction Model

Coefficients	Estimate	Significance
Intercept	23.70	< 2e-16
Amumnw1	0.99	0.000705
BHPS$aage	-0.09	4.1e-05
Aeduc2	-0.59	0.143330
Aeduc3	-1.16	0.001484
Aeduc4	-1.26	0.012077
Aeduc5	-1.40	0.048097
Bepac1	-4.47	0.038988
BHPS$aage:Bepac1	0.14	0.074624
Multiple R-squared	Adjusted R-squared	p-value
0.067	0.055	3.546e-07

Table 10. Outputs of ANOVA between the Forward Selection and Interaction Models

	Sum of Squares	F Value	Significance – Pr(>f)
Interaction Model compared with Forward Model	40.56	3.19	0.075

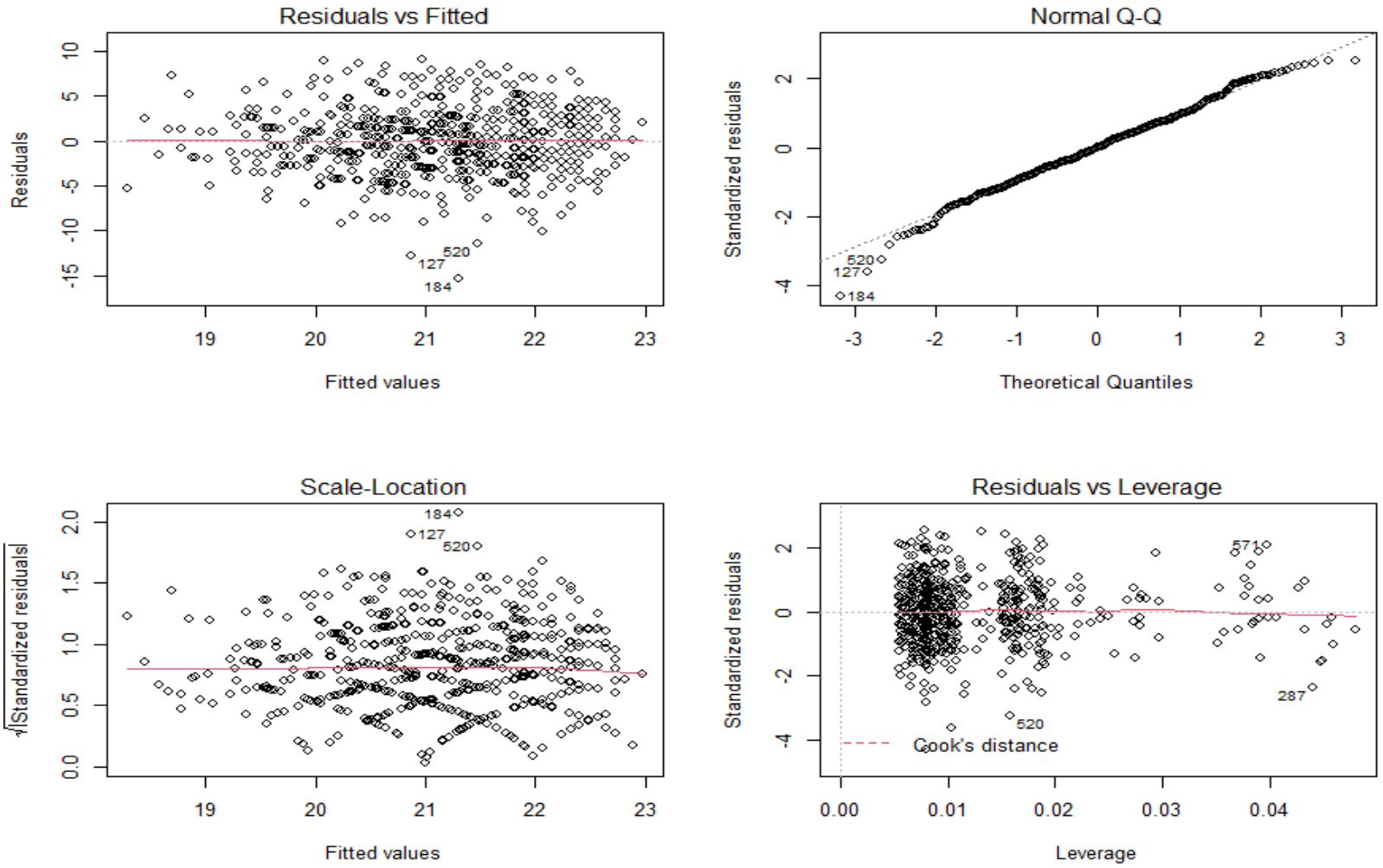
1.3 Exercise C

For the final selected model, the forward model, assumption violations of linear regression were assessed through the following diagnostics plots; Residual Vs Fitted, Normal Q-Q, Scale-Location, and Residuals Vs Leverage Plots. These plots can be found below in **Figure 4**. These plots were created using the base R plot() function on the forward model.

From the Residuals vs Fitted plot (top left) no clear pattern can be observed, suggesting the assumption of linearity has not been violated. However, the large density of residual associated with the higher values for x may indicate heteroscedasticity. The Normal Q-Q plot (top right) suggests the assumption of normality has not been violated as most of the points fall on the line. However, there appear to be outliers at the lower and higher values of x. Finally, the last two plots, Scale-Location, and Residuals Vs Leverage Plots (bottom left and bottom right), show that the cases 184 , 520, and 127 as outliers. However, the cooks distance suggests no outliers hold significant leverage within the model. These results indicate the assumptions of linear regression have not been violated and thus the results can be considered reliable.

```
# this function Defines a 2 by 2 multifigure display - filled
by rows (2x2, producing 4 figures)
par(mfrow = c(2,2))


# creating diagnostic plots with the R base function plot()
plot(forward_model)
```

Figure 4. Model Diagnostic Plots



2. Exercise 2

2.1 Creating a Linear Congruential Generator

First a Linear Congruential Generator (LCG) was created to produce random numbers where u ~ U(0,1), producing uniform random numbers between 0 and 1. These random numbers will be required in the binomial random generator. The parameters a,c, and m are the same as the ANSI C implementation (Saucier, 2000).[1] The first parameter n refers to the number of random numbers to generate.

```
Cong_Gen <- function(n, a=1103515245,c=12345,m=2^32) {

  # Initiate an empty variable called x, this will store the
sequence of random numbers
  x <- NULL

  # A seed is required to generate the first number
  # common practice is to use the current system time in
microseconds
  seed <- as.numeric(Sys.time()) * 1000

# use for loop to iterate from 1 to n, the given quantity of
numbers
  for (i in 1:n) {

    # generate the first random value from the seed
    # Store this new value within seed
    # this will mean the previous number will be used to
generate the next number
    seed <- (a*seed + c) %% m

    # convert the current number stored in 'seed'  to a value
between 0 and 1 using the modulus, store this value in x
    x[i] <- seed/m
  }

  # return the final vector containing n random numbers
between 0 and 1
  return(x)
}
```

[1] Saucier, R., 2000. *Computer generation of statistical distributions* (No. ARL-TR-2168). ARMY RESEARCH LAB ABERDEEN PROVING GROUND MD.

2.2 Exercise A

Binomial Generator

For the binomial generator two arguments are required, *n*, the sample size per observation, and p, the probability of success per observation. To generate from the binomial distribution the variable n, is used to generate n random numbers between 0 and 1 using the Linear Congruential Generator (LCG) previously defined. Each of these random numbers are treated as a separate Bernoulli trial, if the number is equal to, or less than, the probability of *p*, then this number is a 'success'. the total count of 'successes' in the sample is the excepted 'successes' based on the given n and p for the binomial distribution.

Within the function *n* is required to be an integer. Base R does not have any way to check for integers. The package "ttutils" was chosen for the isInteger() function.

```
# Installing "ttutils" package as base R does not have a way
to check for integers
# calling ttutils from the library
install.packages("ttutils")
library(ttutils)
```

The function is first defined with the "function" keyword, requiring two arguments n (sample size) and p (probability of success).

```
psuedo_binomial <- function(n, p){
```

Within this function a variable called 'x' is defined. This is the number of trials required, or how many times to repeat the binomial generator. For this exercise 'x' is set to 1000 to obtain a large sample to examine.

```
  # set x, the number of observations
  x <- 1000
```

An empty vector called 'binomials' is initiated. This vector will store the binomial random numbers for each sample (storing the amount of 'successes' per trial).

```
  # initiate an empty vector called binomials
  binomials <- c()
```

Before the loop begins an if() statement is used to check if the value of n is an integer using isInteger() from "ttutils". If n is not an integer the repeat loop will not begin, and an else statement is used to print an error. If n is an integer, then the repeat loop will begin. This repeat loop is used to iterate over the block of code until a break condition is met.

```
   # if statement used to ensure n is an integer
  if(isInteger(n)){

    # use a repeat loop to iterate over block of code
    repeat{
```

The first step of the repeat loop is to generate n random numbers from the LCG. These numbers are stored in the variable 'random_numbers'.

```
      # generate n (sample size) random numbers
      random_numbers <- Cong_Gen(n)
```

Using these random numbers, the amount of 'successes' are counted. Successes are defined as any number less than or equal to *p*, the probability. As this is a logical statement as.integer() is used to convert 'yes' to 1 and 'no' to 0. The sum() function is then used to count all the 1's, or successes. This total value of successes in the sample is stored in the variable 'successes'.

```
      # counting the 'successes' in the sample
      successes <- sum(as.integer(random_numbers <= p))
```

The previously defined vector 'binomials' is appended with itself to ensure all 'successes' counts from previous trials are carried over. Then it is appended with the 'successes' from the current trial.

```
      # appending the binomials vector with the successes
      binomials <- c(binomials,successes)
```

Once the random numbers and successes have been calculated for the current trial, Sys.sleep() is used to pause the function for 0.001 seconds. This is due to the use of 'system time' as the seed for the LCG. Using system sleep ensures each trial has a unique system time to generate the random numbers. Otherwise, the function will run faster than the system time changes, and so sequential observations will share the same system time and thus produce identical numbers. Sys.sleep() prevents this issue and ensures random numbers are produced for each observation.

```
      # Using Sys.sleep() to pause the function for 0.001
seconds
      Sys.sleep(0.001)
```

The if() statement below is the break statement for the repeat loop. First the length() function is used to count how many numbers are currently stored in the vector 'binomials'. i.e. count how many trials have been completed. If this length is equal to x, the number of desired trials, then the 'break' statement is used to end the repeat loop. Once the repeat loop has been broken the vector 'binomials' is returned to the user. The final 'else' statement of the code relates to when n is not an integer. An error is printed to the user informing them of this. The function ends here.

```
      # This is the break statement for the repeat loop
      if(length(binomials) == x){
        break
      }
    }
    # return the final vector 'binomials' to the user
    return(binomials)
  }
# if n is not an integer print the following statement
  else {
    print("Error, provided n is not an integer")
  }
  }
```

2.3 Binomial Generator Documentation

The Binomial Distribution

Description

The binomial distribution refers to when there are exactly two mutually exclusive outcomes of a trial, success, or failure**.** This function generates random samples for the binomial distributions with parameters n, sample size, and p, probability of success. The function returns a vector of random numbers from the binomial distribution for 1000 trials.

Usage

```
psuedo_binomial(n, p)
```

Arguments

n	Number of trials (must be an integer).
p	Probability of success on each trial

Example

```
# This will return the predicted 'successes' in a sample of 100 with a probability of success being 0.5. This will be repeated 1000 times, producing 1000 different observations.

psuedo_binomial(100, 0.5)
```

2.4 Evidence the Function Works Correctly Using Mean, Variance, and Histograms

To identify if the function works correctly three tests were conducted with three different values for *n* and *p*. The mean and variance for each test was calculated and compared with the expected mean and variance. The means and variance for each test can be found below in **Table 11**. Additionally, **Figure 5** displays histograms for the distribution of numbers produced by each test. As can be seen in **Table 11** the achieved mean and variance is very close to the expected values, considering random variation. The histograms in **Figure 5** indicates a symmetrical pattern of distribution further suggesting the function works correctly.

Test 1

For the first test the n and p respectively were 100 and 0.5. Within ggplot() the data was set to NULL to avoid errors. geom_histogram() was used to plot a histogram and the 'bins' were set to 50. geom_vlin() was used to place a vertical line on the histogram at the mean of the test.

```
# Test 1
Test1 <- psuedo_binomial(100, 0.5)
mean(Test1)
var(Test1)
# histogram
# setting the data to NULL to avoid errors within ggplot
hist_1 <- ggplot(data = NULL, aes(x=Test1)) +

  # Using geom_histogram to plot a histrogram
  geom_histogram(bins = 50, binwidth = 1) +

  # Using geom_vline to place a vertical line at the mean of
Test 1
  # setting the colour and line type to make this more visible
  geom_vline(xintercept = mean(Test1), color = "blue",
             linetype = "dashed", size = 1)
```

Test 2

For the second test the n and p respectively were 500 and 0.3.

```
# Test 2
Test2 <- psuedo_binomial(500, 0.3)
mean(Test2)
var(Test2)
# Test 2 Histrogram
hist_2 <- ggplot(data = NULL, aes(x=Test2)) +
  geom_histogram(bins = 50, binwidth = 1) +
  geom_vline(xintercept = mean(Test1), color = "blue",
             linetype = "dashed", size = 1)
```

Test 3

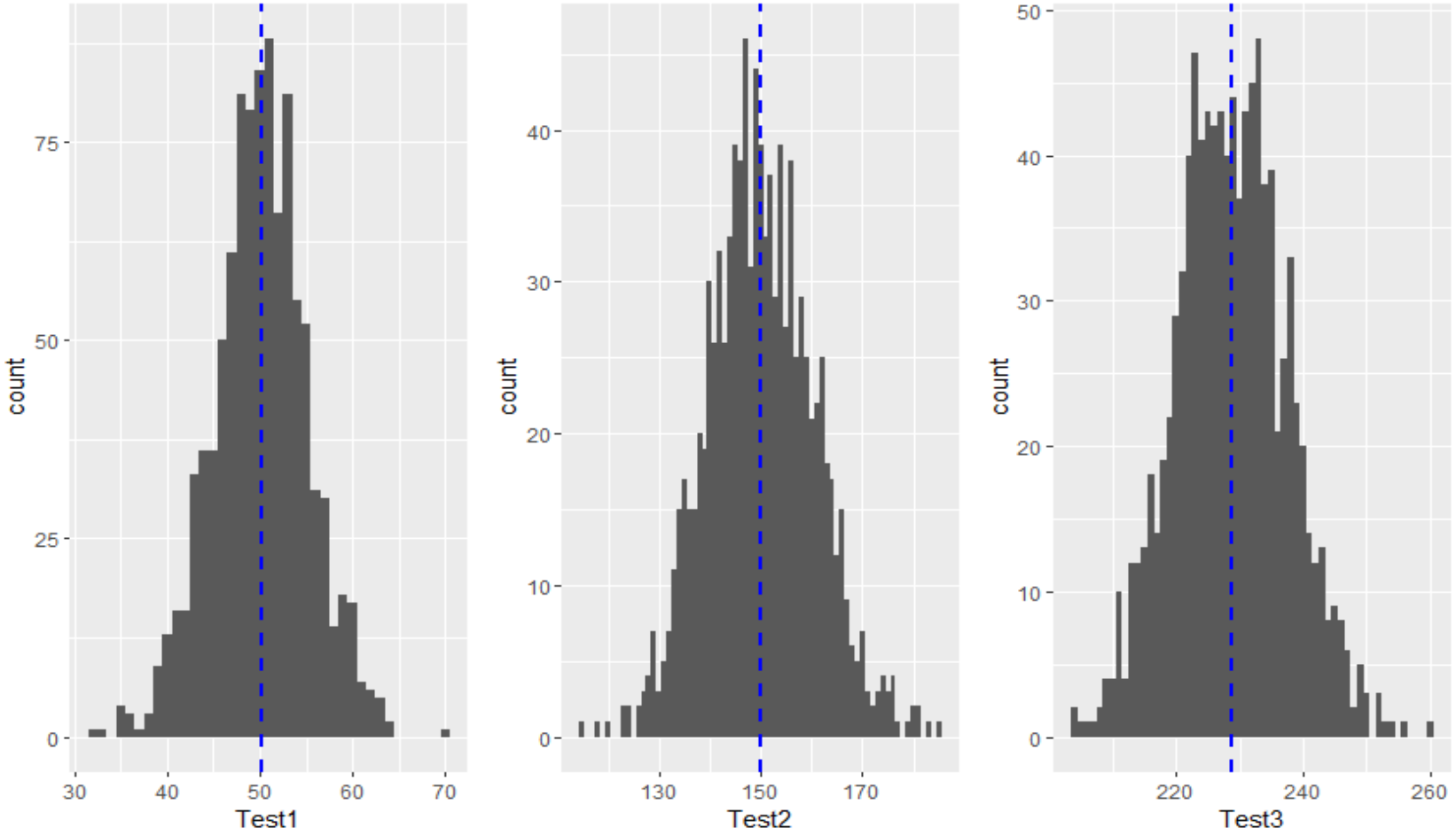
For the third and final test the n and p respectively were 352 and 0.65. The library “patchwork” was used to combine the three histograms as base R does not have this functionality.

```
# Test 3
Test3 <- psuedo_binomial(352, 0.65)
mean(Test3)
var(Test3)
# Test 3 Histrogram
hist_3 <- ggplot(data = NULL, aes(x=Test3)) +
  geom_histogram(bins = 50, binwidth = 1) +
  geom_vline(xintercept = mean(Test3), color = "blue",
             linetype = "dashed", size = 1)
# Using 'patchwork' to combine the three histograms
library(patchwork)
hist_1 + hist_2 + hist_3
```

Table 11. Mean, Expected Mean, Variance, and Expected Variance for All Tests

Test	Calculated Mean	Expected Mean	Calculated Variance	Expected Variance
1 (100, 0.5)	49.9	50	25.12	25
2 (500, 0.3)	150.01	150	108.63	105
3 (352, 0.65)	228.95	228.8	79.95	80.08

Figure 5. Distribution of Numbers Generated for Each Test



3. Exercise 3

In this exercise the relationship between family weekly expenditure on food and explanatory variables 'family income', 'children in the family', and 'location of family's residence' will be explored using ordinary least square regression. All variables within the study and coding can be found below in **Table 12**.

Table 12. Variables and Coding Used Within Study

Label	Name	Coding
Dependent Variable		
Weekly expenditure on food	Expenditure	In Dollars
Independent Variables		
Family Income	Income	In $1000
Children In the Family	Children	0 = No 1 = Yes
Family's Residence	Location	0 = City 1= Rural 2 = Suburb

3.1 Exercise A

To investigate this relationship, the data was created in R. Individual vectors are created for each variable and the associated data using 'c()' to concatenate the values together. Next these variables are combined together into a single data frame using data.frame(), with 'string as factors' equal to 'FALSE' to ensure the categories retains their names.

```
# creating vectors and storing the data

# expenditure data
expenditure <- c(88,86,90,60,37,39,71,80,45,103,53,37,89,
                 79,79,48,97,39,63,35)

# income data
income <- c(25,35,22,37,14,20,15,26,16,38,32,17,22,16,27,
            30,35,13,35,21)

# children data
children <- c('yes','yes','yes','no','no',
              'no','yes','yes','no','yes','no','no',
              'yes','yes','yes','no','yes','no','no','no')

# location data
```

```
location <- c('suburb','rural','city','city',
              'city','suburb','rural','rural','city','city',

suburb','city','city','city','suburb','suburb','suburb',
              'city','city','rural')


# creating a new data frame from these vectors using
data.frame()
expenditure_data <- data.frame(expenditure,income,
                               children,location,
stringsAsFactors = FALSE)
```

3.2 Exercise B

To explore the relationship between the independent variable, 'expenditure' and the dependent variables a linear regression model is fitted. 'Dummy' variables for the categorical variables 'children' and 'location' are calculated. A logical statement is used to separate the variable into yes/no for being equal to desired category. as.integer() is used to convert these yes/no's into 1/0's. For the variable 'location' the reference category is 'city'. The reference category for 'children' is 'no'.

```
# calculating dummy variables
# location = suburb
suburb <- as.integer(expenditure_data$location == 'suburb')
# location = rural
rural <- as.integer(expenditure_data$location == 'rural')
# children = yes
children <- as.integer(expenditure_data$children == 'yes')
```

Next a linear model is fitted using these dummy variables, the output for this regression can be found below in **Table 13**.

```
# Creating a linear model with all variables
expenditure_model <- lm(expenditure ~ income + children +
suburb + rural), data = expenditure_data)
```

However, due to the very small sample size of 20 observations, a non-parametric bootstrap was performed for the regression parameters. Non-parametric bootstrap was chosen as no assumptions are made on the distributional form of the data. The essential idea of the non-parametric bootstrap is to draw a sample of size *n* from the data, sampling with replacement. The data is treated as an estimate of the population. For this exercise, the regression model residuals will be bootstrapped. For both exercise b and c the package "boot" is used. boot offers two useful functions "boot" and "boot.ci". This package and functions perform the same non-parametric bootstrap as the package "bootstrap". This is due to the default bootstrap in "boot" being "ordinary", referring to ordinary non-parametric bootstrap.

Table 13. Output for the Linear Regression Between 'expenditure' and the Dependent Variables

Coefficients	Estimate	Significance
Intercept	25.18	3.24e-08
Income	1.00	3.03e-08
Children	40.32	7.81e-14
Suburb	-6.04	0.00419
Rural	-11.56	4.61e-05
Multiple R-squared	Adjusted R-square	p-value
0.98	0.98	3.993e-13

Residual Bootstrap

In order to bootstrap the residuals using boot() a function must be defined to return the statistic to be bootstrapped. This function takes two arguments 'data', the data to be bootstrapped, and 'indices', an index vector to generate a bootstrap sample of the residuals.

```
# Loading the boot library
library(boot)


# creating function to obtain residual bootstraps
residual_boot <- function(data, indices){
```

The first line of the function calculates the bootstrap predicted values (newY), equal to the 'expenditure_model' fitted values plus the models residuals. '[indices]' is used in order to obtain the models 'residuals' to be bootstrapped by boot().

```
newY <- expenditure_model$fitted.values +
expenditure_model$residuals[indices]
```

In the final lines of the function the bootstrap coefficients are obtained by fitting a new model for every bootstrap sample using the calculated newY variable.

```
  fit <- lm(newY ~ income + children + suburb + rural, data =
data)
  return(coef(fit))
}
```

The boot() function is then used to perform a non-parametric bootstrap. The first argument supplied is the data to be used in the 'residual_boot' function. The second argument 'statistic' refers to the function that will be used to calculate the model coefficients, in this case this function is 'residual_boot'. 'R' determines the number of replications. The output of this bootstrap is called to obtain the standard errors (standard deviation), found below in **Table 14**.

```
# using boot() to perform non-parametric bootstrap
residual_bootstrap.output <- boot(expenditure_data, statistic
= residual_boot, R = 2000)
residual_bootstrap.output
```

Finally the confidence intervals for each coefficient are calculated using this bootstrap output in the boot.ci() function. The first argument is the bootstrapped output. The second argument is the confidence level required. The third argument is the type of confidence interval ("perc" = percentile). The final argument 'index' refers to the coefficient of interest determined by the order the variables are fitted to the model. i.e. index 1 refers to the intercept. The outputs can be found below in **Table 14**.

```
# calculating confidence intervals for each coefficient
# intercept, index = 1
boot.ci(residual_bootstrap.output, conf = 0.95, type = "perc",
index = 1)
# income, index = 2
boot.ci(residual_bootstrap.output, conf = 0.95, type = "perc",
index = 2)
# children, index = 3
boot.ci(residual_bootstrap.output, conf = 0.95, type = "perc",
index = 3)
# suburb, index = 4
boot.ci(residual_bootstrap.output, conf = 0.95, type = "perc",
index = 4)
# rural, index = 5
boot.ci(residual_bootstrap.output, conf = 0.95, type = "perc",
index = 5)
```

3.3 Exercise C Original Data Bootstrap

For exercise c, the original data was bootstrapped using a similar method to exercise b, using boot() and boot.ci(), the outputs for both can be found below in **Table 14**. First a function is created containing two arguments, data, and indices. The numbers stored in the original data are obtained using data[indices,] and stored in the variable 'data_boot'.

```
# Function to obtain original data bootstrap
original_boot <- function(data, indices) {
  data_boot <- data[indices, ]
```

The final lines of the function use the identified original values to fit a linear model, with the data argument set to 'data_boot'. This will generate bootstrap multivariate samples; the bootstrap coefficients are obtained by fitting this model for every bootstrap sample.

```
  fit <- lm(expenditure ~  income + children + suburb + rural,
data = data_boot)
  return(coef(fit))
```

Next boot() is used to perform a non-parametric bootstrap with 2000 replications. The output is called to obtain the standard errors (standard deviation) for each coefficient.

```
# using boot() to perform original data non-parametric
bootstrap
original_bootstrap_output <- boot(expenditure_data, statistic
```

```
= original_boot, R = 2000)
original_bootstrap.output
```

The confidence intervals for each coefficient are calculated using this bootstrap output in the boot.ci() function in the same way as exercise b.

```
# calculating confidence intervals for each coefficient
# intercept, index = 1
boot.ci(original_bootstrap.output, conf = 0.95, type = "perc",
index = 1)
# income, index = 2
boot.ci(original_bootstrap.output, conf = 0.95, type = "perc",
index = 2)
# children, index = 3
boot.ci(original_bootstrap.output, conf = 0.95, type = "perc",
index = 3)
# suburb, index = 4
boot.ci(original_bootstrap.output, conf = 0.95, type = "perc",
index = 4)
# rural, index = 5
boot.ci(original_bootstrap.output, conf = 0.95, type = "perc",
index = 5)
```

Table 14. Standard Deviations and Percentile-based 95% Confidence Intervals for Bootstrapping the Residuals and the Original Data

Coefficient	Method	Standard Error (Standard Deviation)	95% Confidence Intervals
Intercept	Residual	2.12	[20.85, 29.09]
	Original Data	4.07	[15.62, 31.91]
Income	Residual	0.08	[0.83, 1.15]
	Original Data	0.16	[0.60, 1.21]
Children	Residual	1.41	[37.48, 43.09]
	Original Data	2.67	[33.09, 43.72]
Suburb	Residual	1.58	[-9.223, -3.027]
	Original Data	2.91	[-5.622, 5.618]
Rural	Residual	1.79	[-15.29, -8.09]
	Original Data	3.35	[-6.46, 6.48]

Bootstrap Interpretation

Table 14, displaying the results of bootstrapping with the regression residuals and using the original data show a clear pattern. Bootstrapping with the original data yields both higher standard errors and much wider confidence intervals, this is due to the uncertainty about the form of the regression model. This indicates bootstrapping with the residuals provides a more precise estimate of the population coefficients compared to bootstrapping the original data.

4. Exercise 4

In this exercise the book “Gulliver’s travels” will be analysed using Python.

```
# Download the required book and converting into string format
import urllib.request
text_url = "https://www.gutenberg.org/files/829/829-0.txt"
res = urllib.request.urlopen(text_url)
data = str(res.read())
# Importing numpy
import numpy as np
# Importing matplotlib library
import matplotlib.pyplot as plt
```

4.1 Exercise A

The first property of the book to be analysed is the mean and standard deviation of the words per line. The data is split using the split() function, split by “\\r\\n” which indicates a new line.

```
# Splitting the data into lines using split()
lines = data.split("\\r\\n")
print(len(lines))
```

An empty vector, ‘count’, is initialised. This vector will store the number of words per line. A for loop is used to iterate through the data stored in ‘lines’, iterating through every line of the book. The append function is used to add a new value to ‘count’, this value is the number of words for the current line. This is calculated by first using split() to separate the line into individual words. The len() function is then used to return the number of items within the split line, thus counting how many words for that line and appending this value to ‘count’. This is repeated for every line in the book.

```
# Initializing an empty vector called 'count' to store the
number of words per line
count = []

# using a for loop to iterate through every line stored in the
vector "lines"
for line in lines:
    count.append(len(line.split()))
```

Numpy's mean() and std() functions are used on the 'count' vector to obtain the mean and standard deviation of words per line. The round() function rounds this value to three decimal places. A print() statement is used to print the results. It was found that the mean number of words per line was 10.88 and the standard deviation 4.17.

```
# Using numpy to calculate the mean and standard deviation for
'count'
mean = round(np.mean(count), 3)
standard_deviation = round(np.std(count), 3)
print("the mean number of words per line is " + str(mean) + "
and the standard deviation is " + str(standard_deviation))
```

A histogram of the distribution of words per line was also produced. This can be found below in **Figure 6**. The hist() function from the 'matplotlib' library was used.

```
# using the hist() function of matplotlib to generate a
histogram of "count"
plt.hist(count, bins = 100)
plt.show()
```

However, this plot also includes lines with zero words. To overcome this a new vector was constructed containing only lines with words called 'remove_0'. A for loop is used to iterate through every value stored in 'count'. An if statement is used to check if the current value is not equal to 0, i.e. a line with words. If this is true, the current value is appended to 'remove_0', this is repeated for all values in 'count'. This new vector is then plotted and can be found below in **Figure 7**. The final histogram shows a left skew, with much of the words per line falling between 11 and 13.

```
# Removing the empty line counts and appending to the new
vector
remove_0 = []

for i in count:
    if i != 0:
        remove_0.append(i)
# This allows for a much clearer plot
plt.hist(remove_0, bins = 100)
plt.show()
```

Figure 6. Distribution of the Number of Words Per Line with Empty Lines

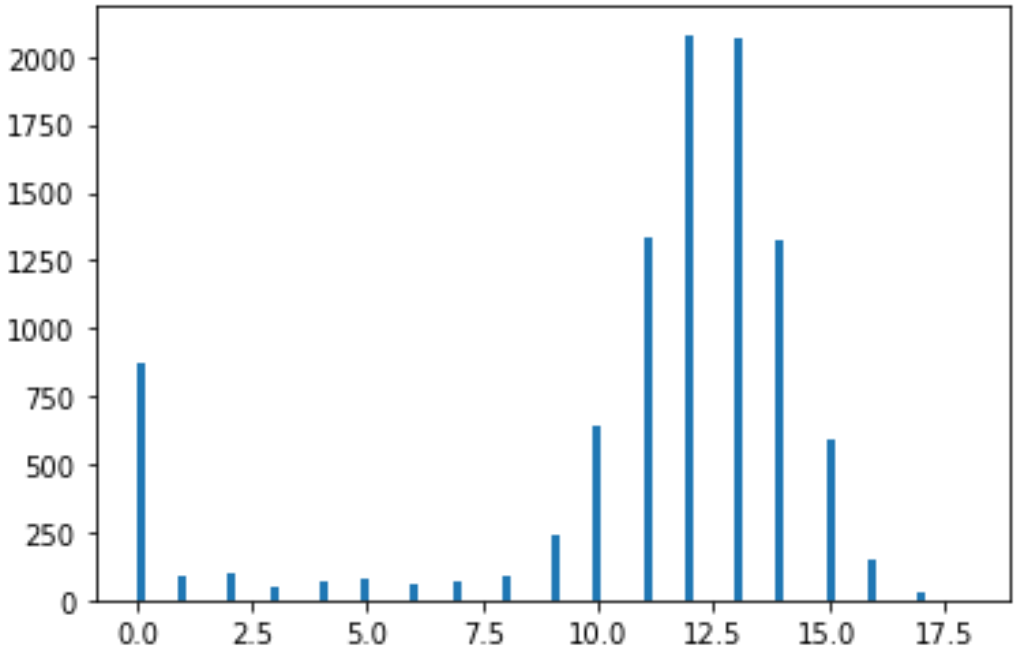
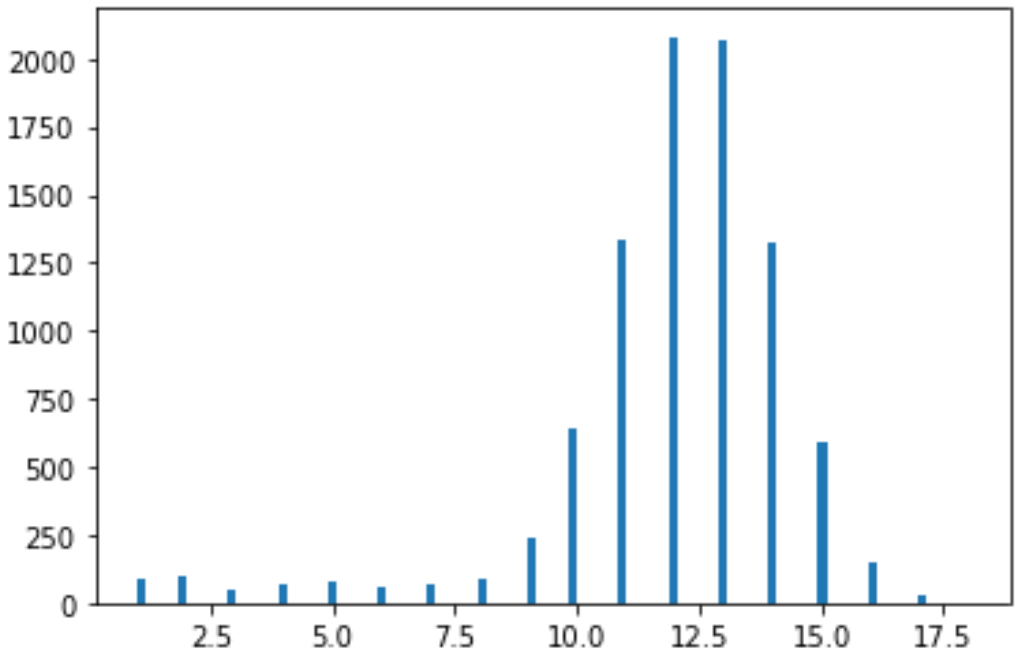


Figure 7. Distribution of the Number of Words Per Line without Empty Lines



4.2 Exercise B

In this section the proportion of words by chapter will be analysed. At the end of the final chapter there is considerable text that follows the footnote, these are not part of the book and could influence the word count. The data was first split by the word "FOOTNOTES". This splits the data into the words before and after footnote using split(). The latter section, 'after footnote', is removed using pop(1) referring to the second value stored in 'removing foot_note'. The remaining data is the main body of the book we want.

```
# first separate the data into two sections "before footnote"
and "after footnote" words
removing_footnote = data.split("FOOTNOTES")
removing_footnote.pop(1)
```

This data stored in 'removing_footnote[0]' is then further split by the word "Chapter" splitting into separate chapters. However, the value chapters[0] contains all the words that appear before chapter 1, these are also removed.

```
# Splitting the book without the footnote by the word
"CHAPTER"
chapters = removing_footnote[0].split("CHAPTER")
chapters.pop(0)
```

Words per chapter are calculated in the same way as the words per line in exercise a. A for loop is used to iterate through every value stored in 'chapters', iterating over every chapter of the book. The current chapter is split into individual words using split(). The length of this new data is obtained using len(), this length is the total words for this chapter, which is then appended to the vector 'chapter_counts'. This is repeated for all chapters.

```
# Iniatilizing an empty vector to store the total words for
each chapter
chapter_counts = []


# using a for loop to obtain the words for each chapter
for chapter in chapters:
    chapter_counts.append(len(chapter.split()))
print(chapter_counts)
```

The numpy function sum() is used on 'chapter_counts' in order to obtain the total word count for all chapters.

```
# Calculating the sum of the words in each chapter
sum_word_count = np.sum(chapter_counts)
```

A new vector is created to store the word count for each chapter as a proportion. A for loop is used to divide each chapter's word count by the total word count, which is multiplied by 100. This value is rounded to 1 decimal place and appended to the 'proportions' vector.

```
# creating a new vector to store the proportion for each
chapter
proportions = []
for proportion in chapter_counts:
    proportions.append(round((proportion / sum_word_count) *
100, 1))
```

A for loop is used to print this proportion for each chapter. The size of the vector is iterated, iterating from 0 to 38 by increments of 1. This enables correct labelling of each chapter. However, as python uses 0 indexing 'p+1' is used to add 1 to the current iteration number so chapter 1 is not chapter 0. The proportion of the book by word count for each chapter can be found below in **Table 15**. It can be seen chapter 6 contains the highest proportion of words and chapter 27 contains the lowest proportion of words.

```
# Using a for loop to print the proportions for each chapter
for p in range(len(proportions)):
    print("Chapter " + str(p+1) + " contains " +
str(proportions[p]) + "% of words" )
```

Table 15. Proportion of Words By Chapter

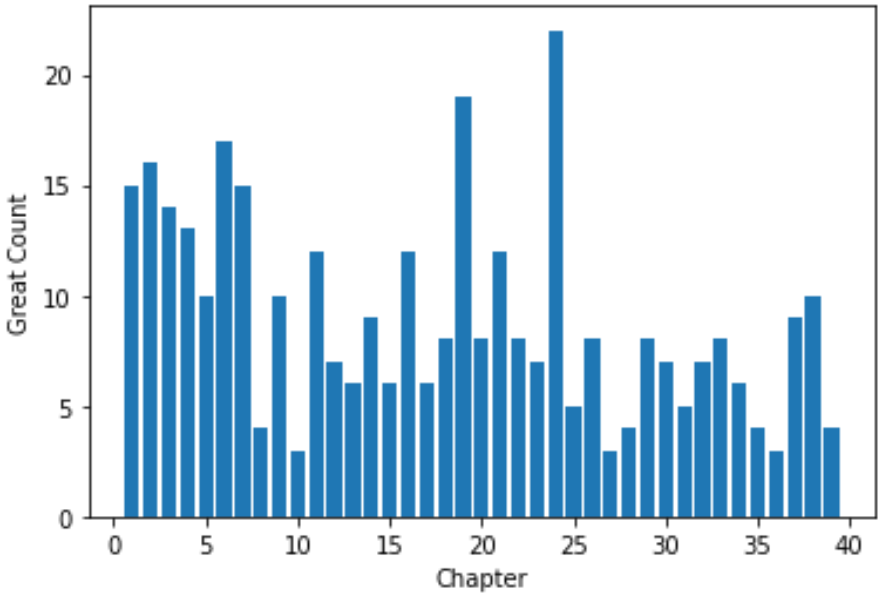
Chapter	Proportion (%)	Chapter	Proportion (%)
1	3.9	**21**	2.5
2	3.6	**22**	1.9
3	2.7	**23**	1.4
4	1.8	**24**	2.0
5	2.4	**25**	1.3
6	3.6	**26**	2.9
7	2.7	**27**	1.2
8	2.3	**28**	2.6
9	4.8	**29**	2.2
10	2.2	**30**	2.1
11	3.8	**31**	1.9
12	1.7	**32**	2.3
13	3.7	**33**	2.3
14	3.1	**34**	2.6
15	2.3	**35**	2.1
16	4.6	**36**	1.9
17	2.0	**37**	2.7
18	2.9	**38**	3.0
19	2.5	**39**	2.3
20	2.1	-	-

4.3 Exercise C

The final property to be analysed is the occurrence of the word "great" in both the text and per chapter. The previously created vector containing the words per chapter ('chapters') is used to identify how many times the word great appears per chapter. A for loop is used to iterate over each chapter. For the current chapter the count() function is used to count the occurrence of "great". This value is then appended to 'great_count' and repeated for all chapters. The distribution of "great" per chapter was then plotted. This can be found below in **Figure 8**. It was found that the word great appears a total of 350 times across all chapters. Plotting the distribution of great by chapter, found in **Figure 8**, suggests that the word great appears more frequently in earlier chapters of the book. However, it can be seen that in chapters 19 and 24 great appears at its highest occurences.

```
# intialising an empty vector to store the occurence of
"great" per chapter
great_count = []

# using a for loop to iterate over each chapter in 'chapters'

# Append the count to 'great_count' and repeat for all
chapters
for chapter in chapters:
    great_count.append(chapter.count("great"))

# Print the sum of this vector to see how many times this word
appears in the chapters
print(sum(great_count))

# plot the distribution of great in each chapter
plt.hist(great_count, bins = 10)
plt.show()
```

Figure 8. Bar chart of Occurrence of the Word 'great' for Each Chapter



5. Exercise 5

The aim of exercise 5 is to create a python function implementing the Newton-Raphson method, an algorithm that finds a root of of a function f using information about the function's gradient.

5.1 Exercise A

The function to be investigated is $f(x)=1.6x^3 + 0.1x^4 + 2x + 3$ and the gradient of this function is $f'(x)=4.8x^2 + 0.4x^3 + 2$. The function will be explored for the interval [-3,3] only.

A python function is created to calculate the value of the function $f(x)$ for any value of x.

```
# Function to calculate the value of the function f (x)=1.6 x
3 + 0.1 x 4 +2 x + 3 for any x

def fx(x):
    return((1.6 * x**3) + (0.1 * x**4) + (2 * x) + 3)
```

Additionally, a Python function is created to calculate the gradient of the function, $f'(x)$, for any value of x.

```
# gradient of the function: 1.6x3 + 0.1x4 + 2x+3  for any
value of x
def gradient_fx(x):
    return((4.8 * x**2) + (0.4 * x**3) + 2)
```

5.2 Exercise B

To visualise the above function a plot for values of x between -3 and 3 was created. numpy linspace() is used to generate 100 evenly spaced numbers between -3 and 3 for the x-axis (x). For the y axis (y), The function $f(x)$ is calculated for every value generated by linspace. The same is repeated for the gradient, using $f'(x)$ instead to calculate x2 and y2 for a gradient plot.

```
# Importing the numpy and matplotlib libraries
import numpy as np
import matplotlib.pyplot as plt

# using numpy linspace to generate numbers between -3 to 3
# for the y-axis the f(x) is calcuted  for all values of x
generated by linspace
x = np.linspace(-3, 3, 100)
y = fx(x)

# calculating the gradient of f(x) for all values of x
x2 = np.linspace(-3, 3, 100)
y2 = gradient_fx(x2)
```

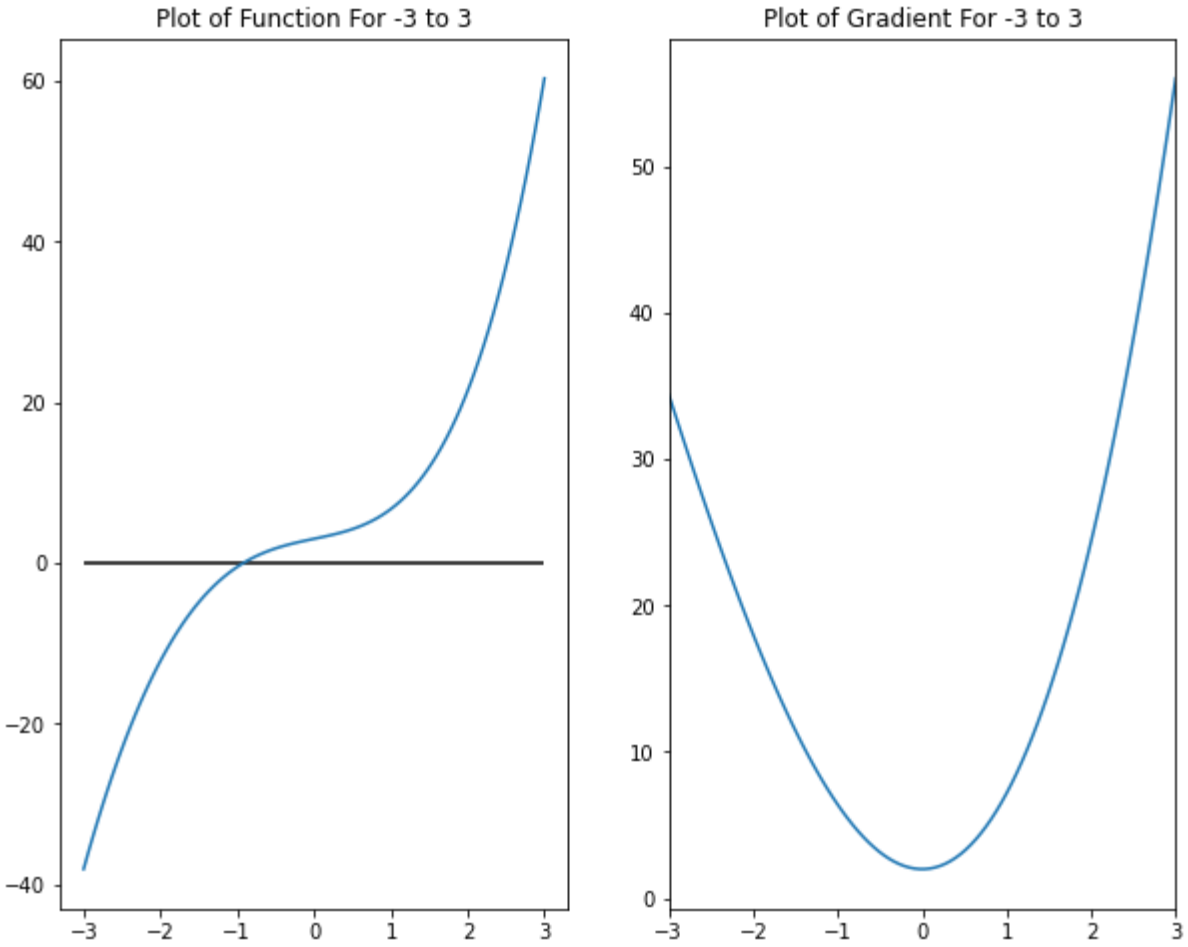
Matplotlib is used to plot the function $f(x)$ using the x and y variables. figure() is used to set the figure size. Subplot(1,2,1) indicates that this plot has 1 row, 2 columns and this plot is the first column. i.e. 1x2 side-by-side figures. Plot() is used to plot the x and y variables. A title and horizonal line at 0 on the y axis are created.

```
# Plotting the function -3, 3

# using matplotlib to plot the function from the created x and
y variables
# using hlines to plot a horizonal line at 0
plt.figure(figsize=(10,8))
plt.subplot(1,2,1)
plt.plot(x,y)
plt.title('Plot of Function For -3 to 3')
plt.hlines(0,-3,3)
```

A similar method is used to plot the gradient $f'(x)$ of the function. Subplot(1,2,2) indicates that this plot is the second column of the figure. x2 and y2 are then plotted and a title added. plt.show() is used to display the side-by-side plots. This output can be found below in **Figure 9**, the output suggests this function has one root in the interval [-3,3].

```
# Plotting the gradient -3, 3
# using matplotlib to plot the function from the created x2
and y2 variables
plt.subplot(1,2,2)
plt.plot(x2,y2)
plt.title('Plot of Gradient For -3 to 3')

plt.show()
```

Figure 9. Plot of the Function (left) and Gradient (right) for X Values Between -3 and 3



5.3 Exercise C

The function created to implement the Newton-Raphson algorithm, titled 'root_finder', requires two arguments *x*, the starting value, and *y* the max iterations. Within the function the provided *y* is assigned to the variable max_iteration and the provided *x* assigned to x0.

```
# two arguments for this function, x = starting value for x, y
= max_iterations
def root_finder(x, y):

    # setting the max iterations to the value of y
    max_iteration = y
    # setting x0 to the given starting value of x
    x0 = x
```

An epsilon is defined within the function to determine how close to zero the approximate root from the algorithm must be. In this case the epsilon is set to 0.0001.

```
    # defining the episolon, 0.0001
    episolon = 0.0001
```

The current iteration of the algorithm, 'i', is initialised and set to 1.

```
    # current iteration set to 1
    i = 1
```

A while loop is used to iterate over the block of code while the current iteration is less than or equal to the max iteration. If the current iteration is greater than the max iteration the while loop will break, and the code will stop.

```
    # using a while loop to iterate over this code
    while i <= max_iteration:
```

Within the while loop the previously defined function 'fx' is used to calculate $f(x)$ for the initial value of *x*0. Next the value of its derivative (gradient), $f'(x)$, for *x0* is calculated using the previously defined function 'gradient_fx'.

```
        # calculating the f(x0) using previously defined
function
        x0_f = fx(x0)


        # calculating the gradient of f(x0) using previously
defined function
        x0_gf = gradient_fx(x0)
```

From these two calculations a new value, *x*1 is calculated from the following formula:

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

The function for x1 is then calculated f(x1). The absolute value of f(x1) is then obtained using abs().

```
        # calculating x1
        x1 = x0 - (x0_f / x0_gf)

        # defining a new value containing the absolute value
of the f(x1)
        absx1 = abs(fx(x1))
```

This absolute value 'absx1' is tested against the epsilon. If the value of 'absx1' is less than or equal to epsilon, then the current value of *x1* has been identified as the approximate root of f(x). The value of *x1* is returned to the user and the function is stopped using 'break'. If the absolute value of f(x1) is greater than epsilon, then the approximate root has not been found. The initial value, x0, is set to the calculated value of x1 and the above code is repeated with a new x1 being calculated and assigned to x0 for each iteration. Current iteration, 'i', is incremented by 1. The code will be repeated until the approximate root is found or max iterations exceeded.

```
        # checking if this absolute f(x1) is less than or
equal to epsilon
        if absx1 <= epsilon:
            # if yes we have found the approximate root
            print(str(x1) + " is the root. Found in " + str(i)
+ " iterations")
            # return the value of x1 as this value is the
approximation to a root of f(x)
            return(x1)
            # break the loop ending the code
            break
        # if the absolute of f(x1) is greater than epsilon we
have not found the approximate root
        else:
            # increment the iteration by 1
            i += 1
            # set x0 to the calculated x1 and re-run the code
with the new x0
            x0 = x1
```

5.4 Exercise D

To demonstrate the function works correctly three random numbers were chosen between -3 and 3. These three numbers were put through the root_finder() function with ten max iterations. The three random numbers selected, and the calculated roots can be found below in **Table 16**. Additionally, a plot of these root locations can be found below in **Figure 10**.

To generate the random numbers the 'random' library was used with a seed set to '10' for reproducibility.

```
# importing the random library
# setting the seed in order for recreating of results
import random as rand
rand.seed(10)
```

Next linspace() is used to generate 100 evenly spaced numbers between -3 and 3. These numbers are stored in a variable.

```
# using numpy to generate 100 evenly spaced numbers between -3
and 3
random_numbers = np.linspace(-3, 3, 100)
```

rand.coice() function from the random library is used on the linspace() numbers to choose a random number between -3 and 3. This is repeated three times for three random numbers. These numbers are rounded to three decimal places. These selected random numbers will be the *x* argument provided to the root_finder().

```
# using the rand.choice() method from the random library to
choose a random number
# rounding these numbers to 3 decimal places
# print these random numbers
rand1 = round(rand.choice(random_numbers), 3)
rand2 = round(rand.choice(random_numbers), 3)
rand3 = round(rand.choice(random_numbers), 3)
print("random number 1 =  "  + str(rand1) + "\n random number
2 = " + str(rand2) + "\n random number 3 = " + str(rand3))
```

The root_finder() function is used to identify the approximate root using the three different *x* values.

```
# for each random x0 find the root with 10 max iterations
# using the previously defined function 'root_finder'
root_rand1 = root_finder(rand1, 10)
root_rand2 = root_finder(rand2, 10)
root_rand3 = root_finder(rand3, 10)
```

As can be seen in **Table 16** below, for all three random numbers the root identified fall very close to one another, rounded to 3 decimal places the root for all random numbers is -0.918. This suggests that the root_finder() function is working correctly.

Table 16. Random Numbers and the Root Identified Through Root_Finder()

Random Number	Root Identified	Iterations Taken to Find
1.424	-0.9175216657124341	5
-2.758	-0.9175170194698702	5
0.273	-0.9175189771335379	4

To display the location of the calculated root for all three random numbers the function $f(x)$ was calculated for values between -3 and 3 to generate the x and y axis as seen in exercise b.

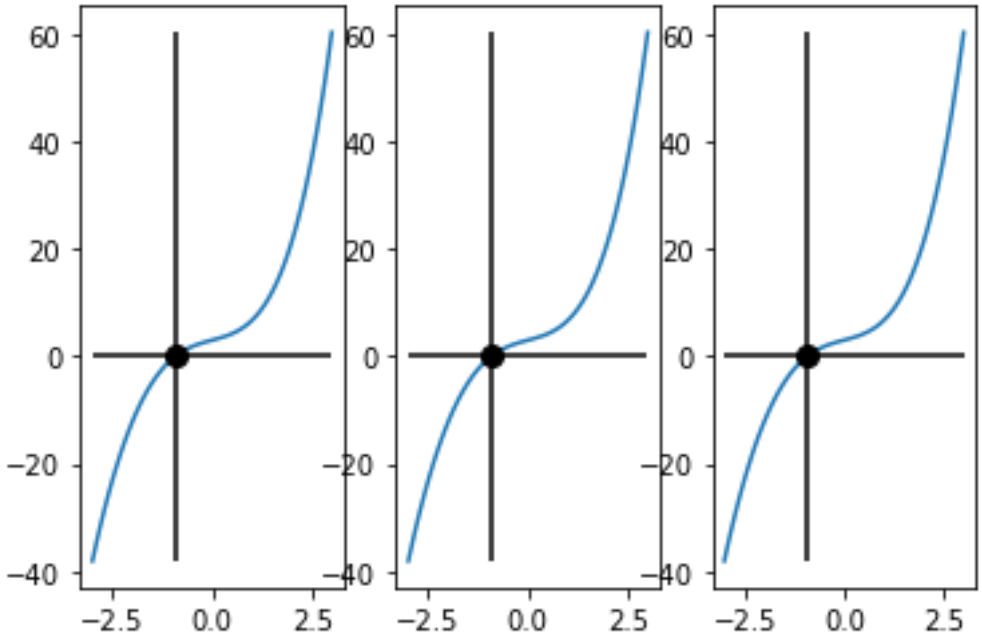
```
# creating the variables to plot the function f(x) for values
between -3 and 3
x = np.linspace(-3, 3, 100)
y = fx(x)
```

The subplot is defined as (1,3,1) indicate this will be a 1x3 figure and this plot will be the first figure. The calculated *x* and *y* values are plotted. The plot() function is used to plot the location of the root, the x argument is set to the calculated root for random number 1. The y argument is set to the function of this root. The 'o' indicates the point will marked with a circle and the colour will be black. To further aid visualisation a horizontal line is placed at 0 on the y axis and a vertical line placed at the calculated root on the x axis. This is repeated for all three random numbers and plt.show() to display the side-by-side plots. This can be found below in **Figure 10**. This figure provides further evidence that the function works correctly as the root location identified is where the function intersects 0 on the y axis.

```
# plotting the location of the roots of each random number on
the plot

# Random number 1 (1.424)
# using plot(), to plot the root
# using hlines and vlines to indicate the position of the root
plt.subplot(1,3,1)
plt.plot(x,y)
plt.plot(root_rand1, fx(root_rand1), 'o', color = 'black',
markersize = 8)
plt.hlines(0,-3,3)
plt.vlines(root_rand1, min(y), max(y))

# Random number 2 (-2.758)
plt.subplot(1,3,2)
plt.plot(x,y)
plt.plot(root_rand2, fx(root_rand2), 'o', color = 'black',
markersize = 8)
plt.hlines(0,-3,3)
plt.vlines(root_rand2, min(y), max(y))
```

```
# Random number 3 (0.273)
plt.subplot(1,3,3)
plt.plot(x,y)
plt.plot(root_rand3, fx(root_rand3), 'o', color = 'black',
markersize = 8)
plt.hlines(0,-3,3)
plt.vlines(root_rand3, min(y), max(y))

plt.show()
```

Figure 10. Plot of the Location of the Root for Random Number 1 (left) , 2 (centre) and 3 (right)



5.5 Exercise E

Within the general case of finding a root between [-a, a] start algorithm at a, converge towards a root, when you observe multiple negative and positive sign changes, sufficiently close to the root, restart the algorithm. Restart with x0 = to new x1 identified root.

$$(x \in \mathrm{R}: |\, x - \text{root}\,| < \varepsilon) \text{ for } \varepsilon \text{ small enough.}$$

6. Exercise 6

Within exercise 6 the 'Titanic' dataset from the seaborn library will be examined. This dataset includes information about the passengers onboard, with each row representing one passenger.

```
# reading the required data in python
# importing the padna, numpy and matplotlib libraries
import seaborn as sns
import pandas as pd
import numpy as np
import matplotlib.pyplot as plot

titanic_df = sns.load_dataset("titanic")
```

6.1 Exercise A

For exercise a, the proportion of passengers who embarked at Southampton will be calculated. First the data is viewed using head() to view the different columns.

```
# Viewing the different columns of the dataset
titanic_df.head()
```

A variable was created containing the total number of people who embarked, used later to calculate the proportion.'embarked_southampton_bool' is then created, this variable is a Boolean equal to true for the passenger if the 'embark_town' is equal to 'Southampton' and false otherwise. This variable is used to create 'embarked_southampton' containing only passengers who were equal to 'true' for embarking from Southampton.

```
# obtaining the total number of people who embarked
embarked = titanic_df['embark_town']

# creating a Boolean that is true if the embark town is equal
to Southampton
embarked_southampton_bool = titanic_df['embark_town'] ==
'Southampton'


# creating a new variable that only has the cases for people
embarking from Southampton
embarked_southampton = titanic_df[embarked_southampton_bool]
```

The proportion was then calculated using the shape() attribute of both variables. The 'shape' of both variables has two values, shape[0], equal to the rows (i.e. number of people) and shape[1], equal to the columns (variables). By using shape[0] we are calculating passenger proportions by dividing those who embarked at Southampton by the total number of passengers, multiplied by 100. This number is rounded to 3 decimal places using round(). These results are printed with str() used to convert the numerical to a string. It was found that 72.278% of the passengers embarked from Southampton.

```
# obtaining the total number of people who embarked
embarked = titanic_df['embark_town']

# calculating the proportion who embarked from Southampton
embarked_southampton_proportion =
round((embarked_southampton.shape[0] / embarked.shape[0]) *
100, 3)
# Printing the answer
print(str(embarked_southampton_proportion) + "% of the
passengers embarked from Southampton")
```

6.2 Exercise B

For exercise b, the distribution of passengers by age will be explored. Additionally, the question of whether this distribution varied by ticket class will also be investigated. 'titanic_df["age"]' accesses only the data stored in the "age" column, a histogram of this data is produced using pandas built in hist() function. A plot title, axis labels, and a line at the mean are added. The output of this histogram can be found below in **Figure 11**. This figure shows the average age of the passengers to be close to 30 years, with many of the passengers falling in between the range of 20 – 40 years old. As age increases the count for passengers decreased.

```
# Distribution of passengers by age
# plotting this in a histogram
# adding a dashed line at the mean
titanic_df["age"].hist(bins = 30, grid = False)
plt.title("Distribution Of Passengers By Age")
plt.xlabel("Age")
plt.ylabel("Count")
plt.axvline(np.mean(titanic_df["age"]), color='k',
linestyle='dashed', linewidth=1)
```

To visualise the distribution by ticket class a scatter plot and boxplot was created using the 'seaborn' library. This output is displayed in **Figure 12**. The mean age for each ticket class was calculated, the 'titanic' data is grouped by 'class' and the agg() function used to apply several aggregation methods. In this case the mean of the 'age' variable for each ticket class is calculated. The output can be found below in **Table 17**. Both outputs show a clear relationship between age and ticket class, as age increases so too does ticket class. First class passengers had the highest mean age at 38.23- and third-class passengers had the lowest mean age at 25.14.

```
# Did the distribution of age vary by class of ticket?
# using seaborn to plot a scatter plot and box plot
sns.catplot(x="class", y="age", data=titanic_df)
sns.catplot(x="class", y="age", kind="box", data=titanic_df)
# Calculating the mean age for each class
titanic_df.groupby('class').agg({'age':np.mean})
```

Figure 11. Histogram of Distribution of Passengers By Age

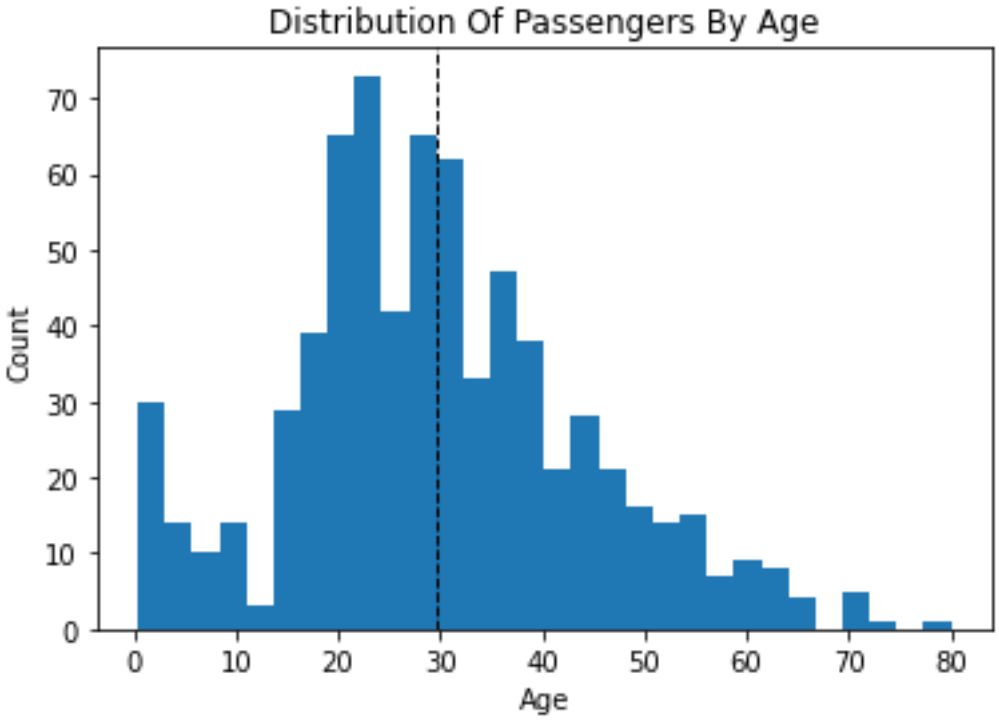


Figure 12. Scatter Plot (left) and Box Plot (right) of Ticket Class By Age

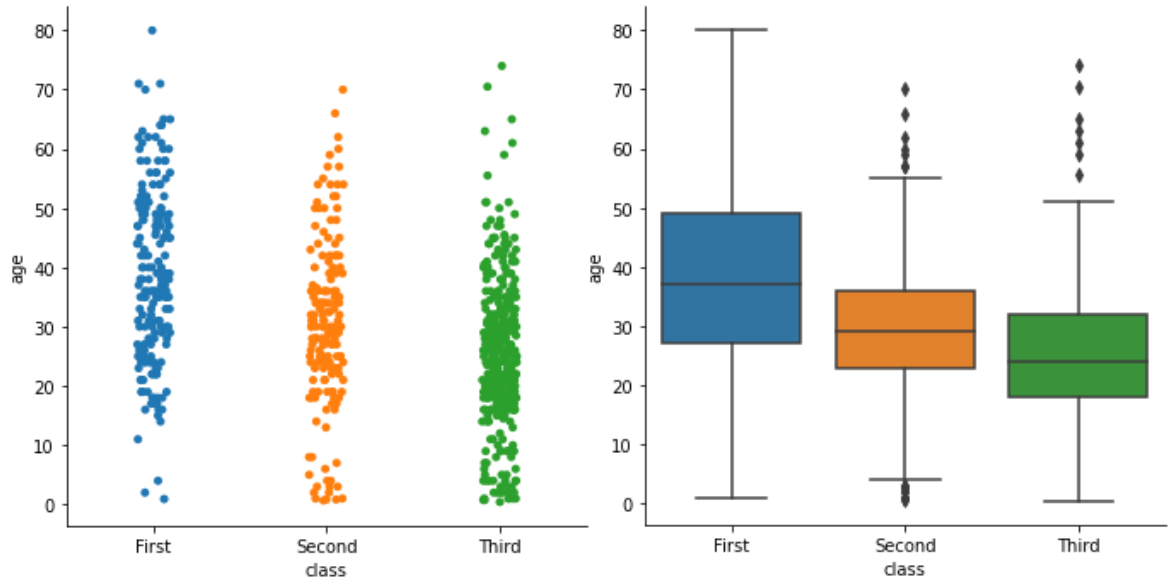


Table 17. Mean age for each Ticket Class

Ticket Class	Mean Age
First	38.23
Second	29.88
Third	25.14

6.3 Exercise C

In this exercise the passenger survival by ticket class will be explored. To identify if first class passengers were more likely the count for passenger survival (survival = 1) and death (survival = 0) for each class was plotted using 'catplot' of kind 'count'. This can be found in **Figure 13**. Additionally, the mean passenger survival for each ticket class is calculated and can be found below in **Table 18**. First the 'titanic' data is grouped by 'class'. Next the agg() function used to calculate the mean passenger survival variable for each ticket class.

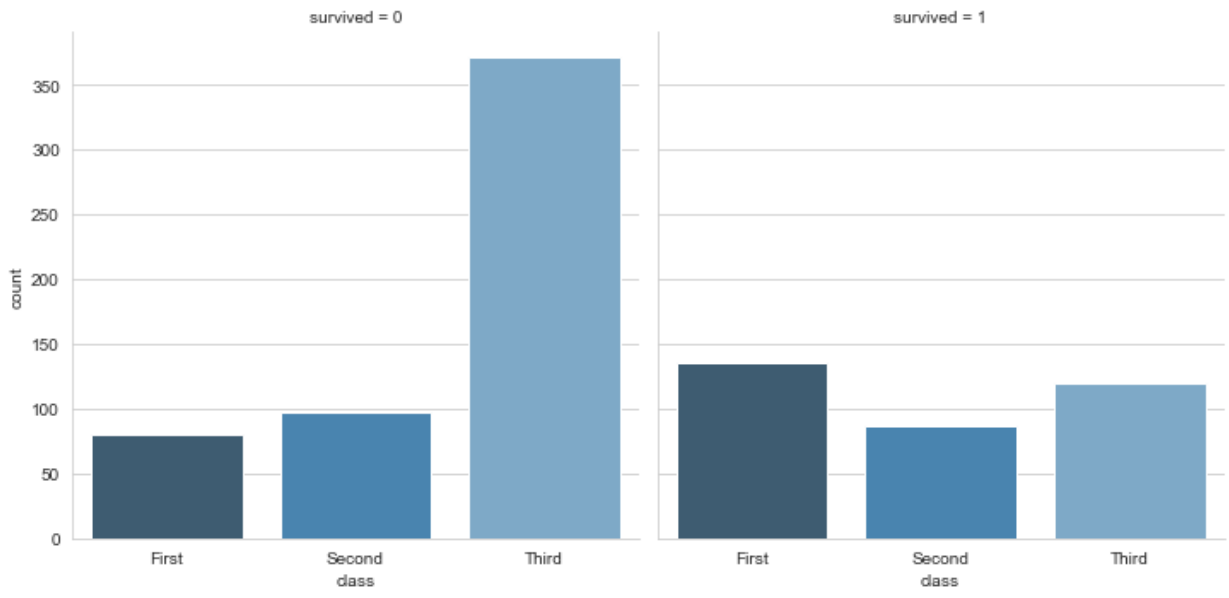
```
# Using seaborn catplot of type 'count' to plot the count for survival and death for each class
sns.catplot(x='class', col='survived', data = titanic_df, kind = 'count', palette='Blues_d')

# Calculating the mean survival for each class
titanic_df.groupby('class').agg({"survived":np.mean})
```

Table 18 shows First Class ticket holders having the highest mean survival whereas Third Class passengers having the lowest mean survival. This is further supported in **Figure 13**, Displaying the counts for death (left) and survival (right) for each ticket class. First class holders had the lowest count for deaths and highest count for survival. Whereas third class ticket holders have the highest count for death.

Table 18. Mean Survival for each Ticket Class

Ticket Class	Mean Survival
First	0.63
Second	0.47
Third	0.24

Figure 13. Bar Chart Comparing the Counts for Death (left) and Survival (right) for Each Ticket Class



6.4 Exercise D

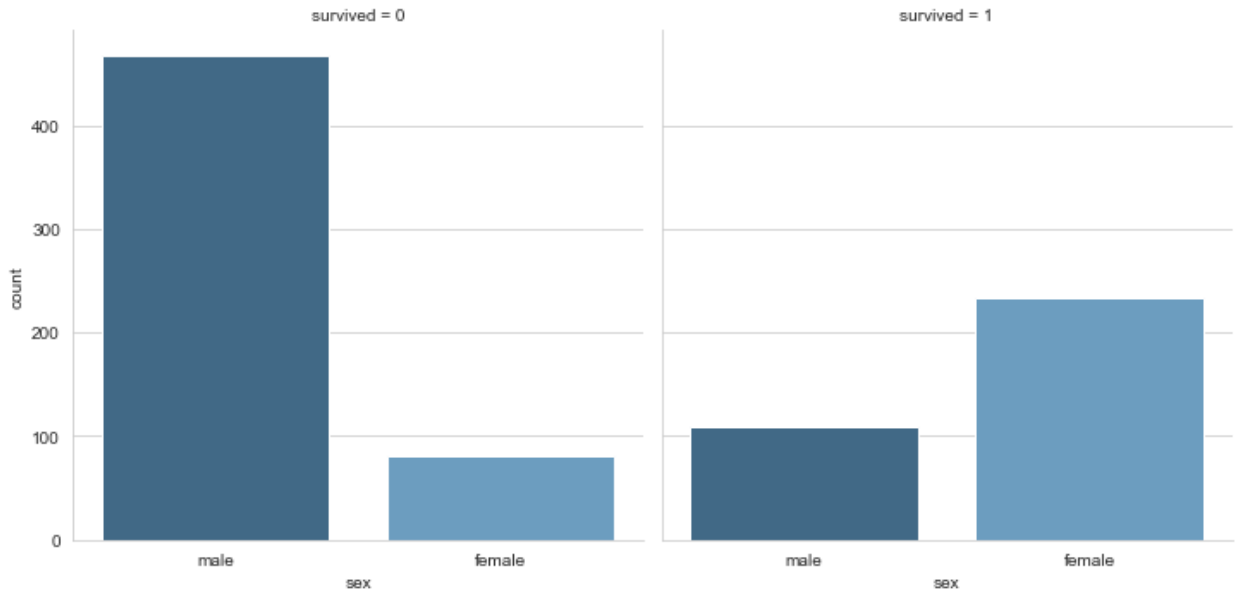
In exercise d, the survival of each sex will be explored. To identify whether males or females were more likely to survive the same methods as outline above in exercise c were used. A bar plot of the count for passenger survival and death for each sex was created, shown in **Figure 14**. Additionally, the mean passenger survival for each sex was calculated, **Table 19**, these results show females having a higher mean for survival and **Figure 14** shows females having significantly less deaths compared to males.

```
# Were males or females more likely to survive?
# plotting the count of death and survival by sex
sns.catplot(x='sex', col='survived', data = titanic_df, kind =
'count', palette='Blues_d')
# Calculating the mean survival for each class
titanic_df.groupby('sex').agg({"survived":np.mean})
```

Table 19. Mean Survival for each Sex

Sex	Mean Survival
Female	0.74
Male	0.19

Figure 14. Bar Chart Comparing the Counts for Death (left) and Survival (right) by Sex



6.5 Exercise E

In this section the missing data within 'titanic' will be identified. The variables with missing data were found using the isnull() function. To calculate the total missing values for each variable the sum() function is used. The variables 'age' 'embarked', 'deck', and 'embark_town' all contained missing values, these results are displayed below in **Table 20**. The total missing values were calculated by taking the sum() of the total missing data for each variable. In total there were 869 missing values. A heatmap of missing values was produced using the heatmap() function. This can be seen below in **Figure 15**.

```
# Identifying the columns with missing data
titanic_df.isnull()
print(titanic_df.isnull().sum())
# calculating the total number of missing data
titanic_df.isnull().sum().sum()
# Visualising these missing values using a heatmap
sns.heatmap(titanic_df.isnull(), cbar = False)
```

In terms of how this may impact the results, the variable 'age' had the second largest total missing values at 177. When plotting by 'age', and making interpretations, the missing data may lead to incorrect conclusions as it may not be an actual representation of the data. However as can be seen below in **Figure 16**, displaying a boxplot of age by ticket class, with the missing data removed, the pattern remains the same as the previous conclusion derived with the missing data included. The missing data was first removed from 'titanic' using dropna(), this data with removed missing values is then store in the variable 'remove_na' and used within the catplot() function of seaborn.

```
# Removing the cases with missing data to see if the results
change
remove_na = titanic_df.dropna(axis = 0, how = 'any')
sns.catplot(x="class", y="age", kind="box", data=remove_na)
```

Table 20. Variables With Missing Data

Variable	Missing Data
Age	177
Embarked	2
Deck	688
Embark_town	2

Figure 15. Heatmap of Missing Values

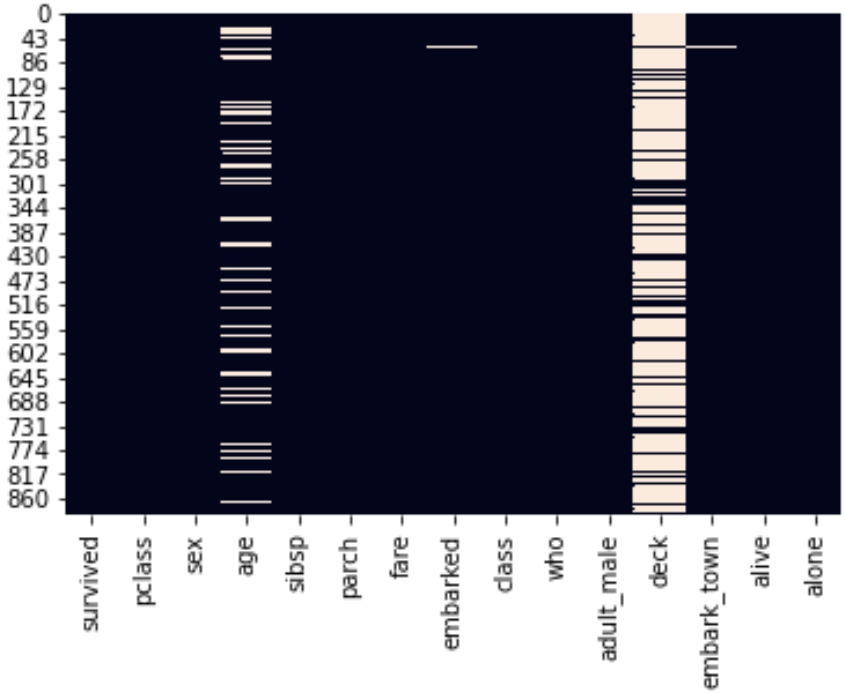


Figure 16. Box Plot of Age by Ticket Class with Missing Values Removed