A Finite Math Companion in R Markdown

Ryan J. Cooper

12/19/2020

Contents

A Finite Math Companion in R Markdown

Most disciplines in statistics and probability build upon elementary subjects in Finite mathematics.

The subjects of matrix operations, linear programming, and set theory provide a basis for statistical analysis projects. A thorough understanding of the concept of expected value underpins most formulas concerning statisics and probability theory.

Many important financial formulas are also considered germane to the application of Finite math, providing formulas to describe widely used instruments in everyday finance. Subjects of interest, annuities, sinking funds, amortization, and present value can help answer questions like - how much will I pay for a car or home loan?, or how much do I need to save to reach a specified retirement goal?

In this paper, we will explore some of the ways that Finate math principles can be applied in R, and highlight the natural fit of R to the visualization of sets. We will explore how counting formulas may be used to define sample spaces, and how R can be used to quickly generate randomized elements for set wise comparison, and to analyze those sets for intersections, unions, complements and subsets, as well as to test and confiemr that the sets conform to De Morgans properties.

It is from sample spaces that expected value can be derived, and the probability of any given outcome can be calculated using Bayesian statistics and rules about conditional probability under conditions of dependence, independence, or mutual exclusivity.

R and RMarkdown provides a powerful platform for general statistical programming. Visualization capabilities in R are enhanced through the GGPlot2 and GGForce packages, and data manipulation functions like select, mutate, and summarize are added by the dplyr package, enabling language that resemble most structured query languages.

```r
library(pracma)
library(dplyr)
library(ggplot2)
library(ggforce)
```

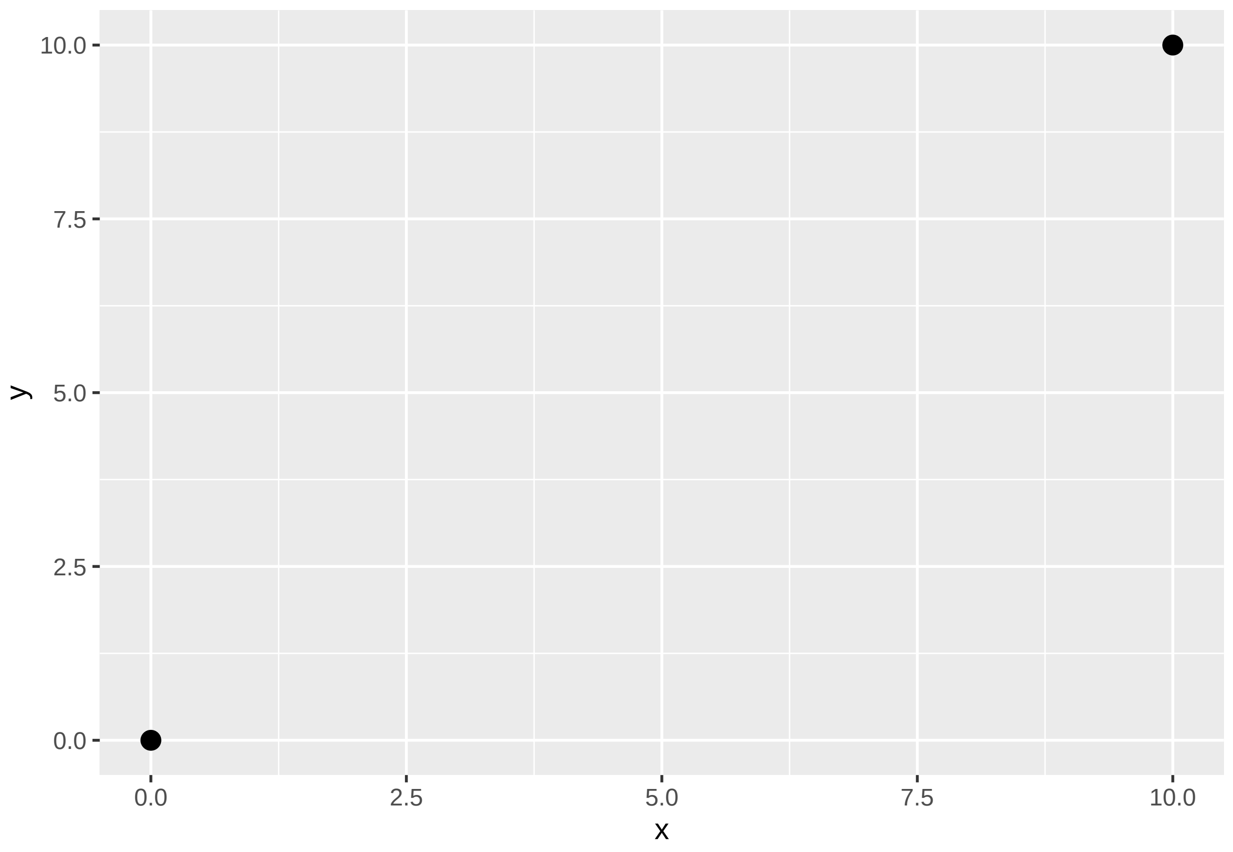
Linear Equations

This section demonstrates some ways to perform basic visualizations that we will build on later. Many applied problems in finite math involve drawing lines on a cartesian plane. GGplot makes this fast, easy and convenient.

Graphing 2 Points on a Plane

Create two points and build a simple plot with just those 2 points at {0,0} and {10,10}. We will make them size 3 to ensure they are easy to see.

```r
x_points <- c(0,10)
y_points <- c(0,10)
coords <- data.frame(x=x_points,y=y_points)

simple_plot <- ggplot(data=coords, aes(x=x, y=y)) +
  geom_point(size=3)
simple_plot
```



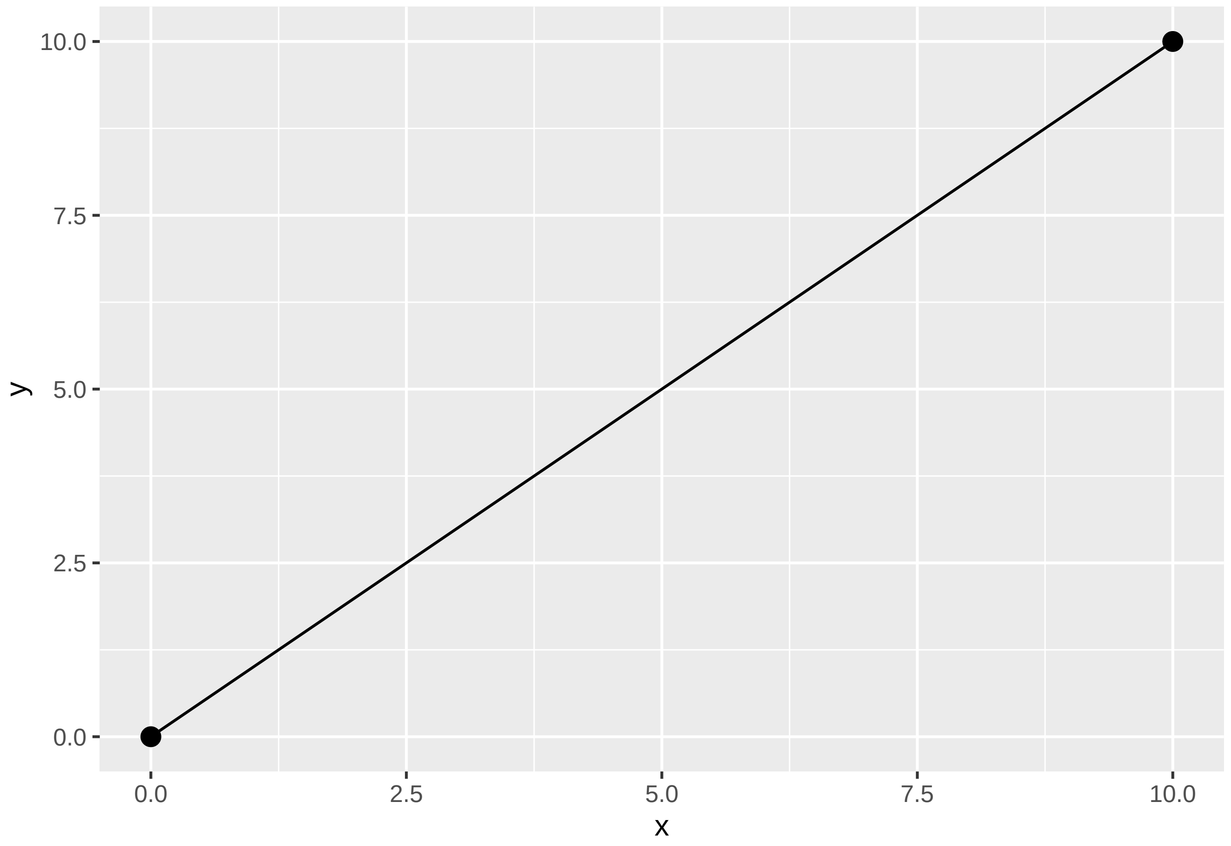
Graphing a Line between 2 Points

Create two points and build draw a line with just 2 points at {0,0} and {10,10}

To add a line between these two points, simply add a geom_line object. In the GGPlot syntax - the aes() argument defines where the data is comiong from (coords) and which fields in the dataset should correspond to x/y points drawn on the chart.

```r
x_points <- c(0,10)
y_points <- c(0,10)
coords <- data.frame(x=x_points,y=y_points)
```

```r
simple_plot <- ggplot(data=coords, aes(x=x, y=y)) +
  geom_point(size=3) +
  geom_line()
simple_plot
```

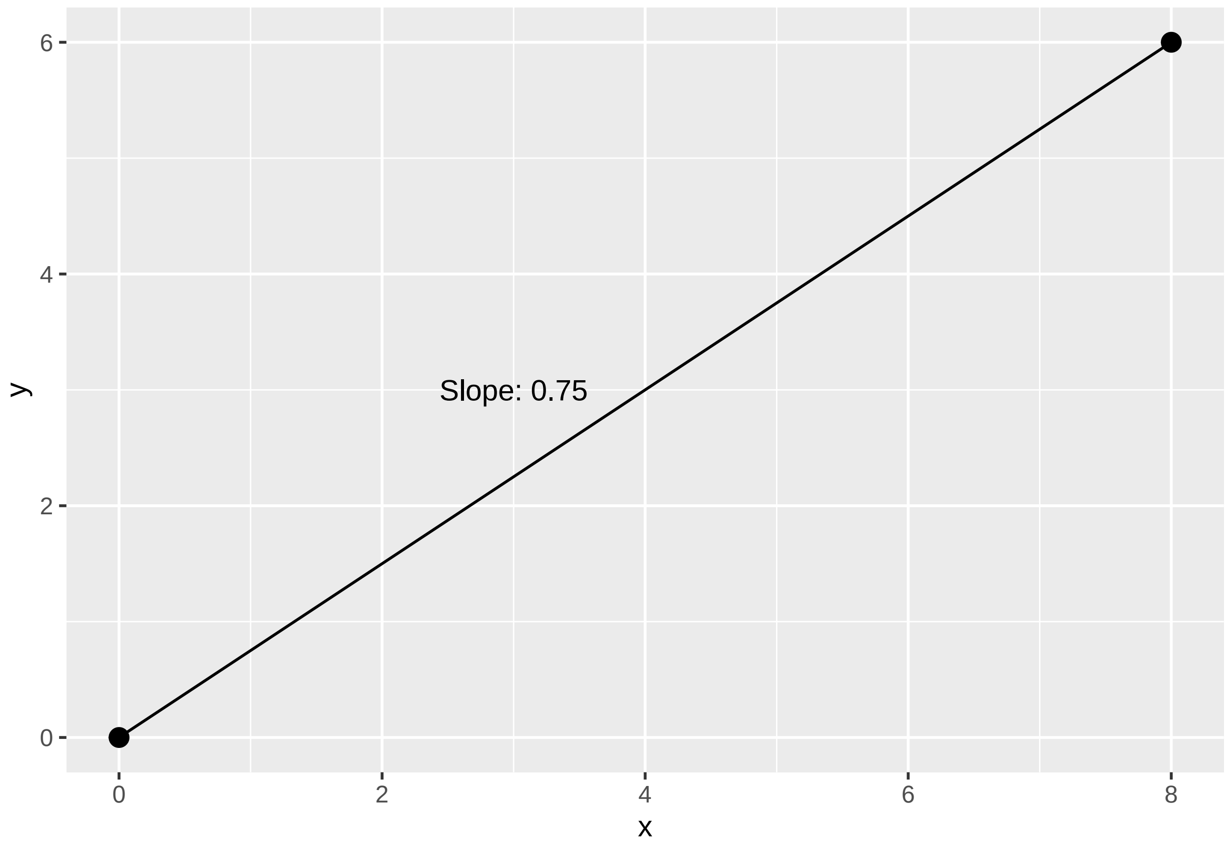


Slope of a Line

Our first equation will be to calculate the slope of a line. For each formal equation we create, we will create a function that encapsulates that function.

Calculate the slope of a line and display it on a chart as an annotation:

```r
#x/y points
x_points <- c(0,8)
y_points <- c(0,6)
coords1 <- data.frame(x=x_points,y=y_points)

#slope of a line equation
slope_line <- function(coords1){
  m <- (coords1$y[2] - coords1$y[1]) / (coords1$x[2] - coords1$x[1])
  m
}

#run the function and set the output to variable m
m <- slope_line(coords1)

#plot
simple_plot <- ggplot(data=coords1, aes(x=x, y=y)) +
  geom_point(size=3) +
  geom_line() +
  annotate("text", x=3,y=3,label = paste("Slope:",m))
simple_plot
```



Graphing a Line with geom_smooth

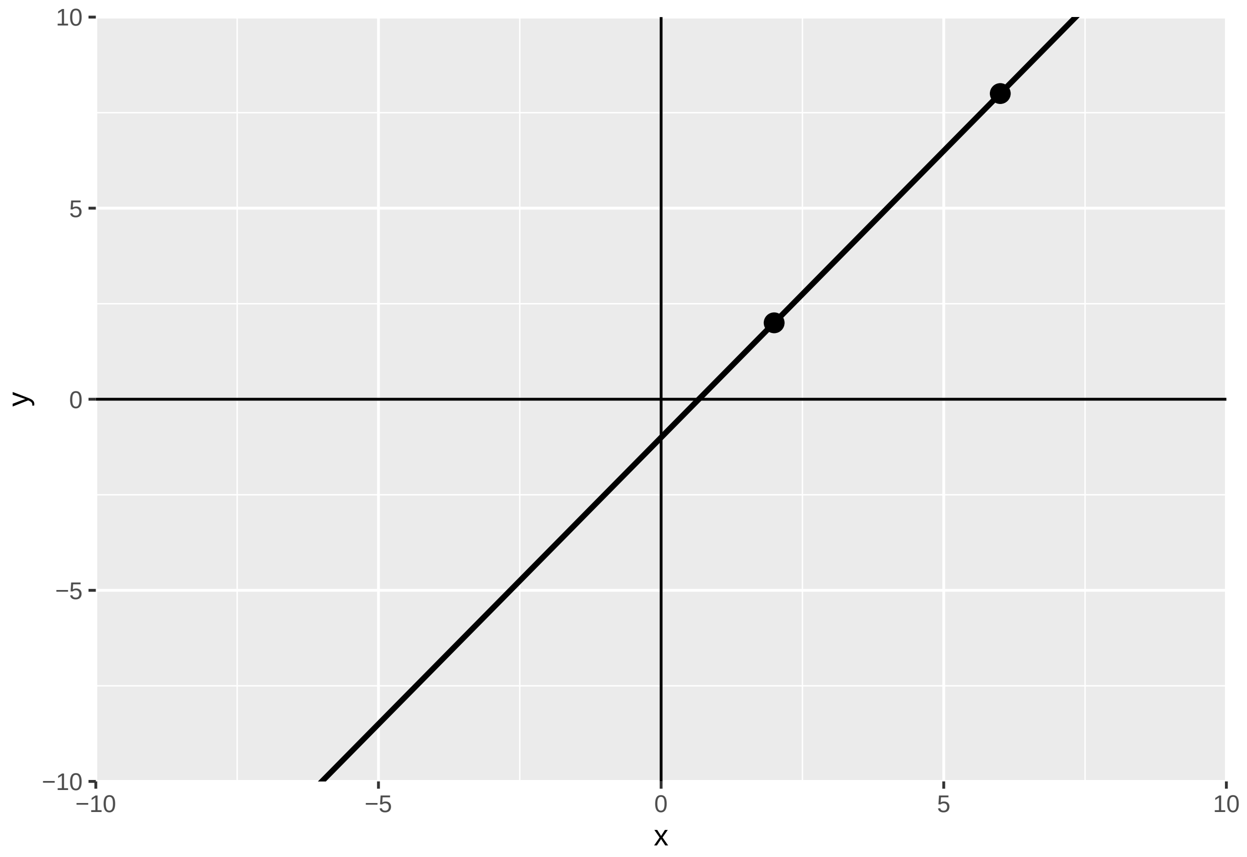
This is a variant of the above line graphs. the geom_smooth produces a *linear regression* line between n points. This will draw a line, based on your points, using a model of your choosing. We will start with a linear model WIth only 2 observations - this will produce a straight line betwee n these two points.

This object will be important in later chapters as we explore plots that contain many points.

In ths section we are introducing the coord_cartesian object. This object will "zoom in" on a portion of the area, without dropping any points. This is similar to the scale object which sets the scale of the area. The difference is that scale removes observations outside the visible scale limits.

Adding the fullrange=TRUE tells GGplot to extend the line into an infinite space, filling the chart area from edge to edge. This is appropriate for linear programming formulas since the line is not finite, and we are iunterested in knowing where 2 or more lines intersect, without defining a start or end point of the lines. The starting and ending points of teh line don't exist - all that exists is a set of observations. A line should follow the slope of the known observations without terminating at any specified location in an x/y coordinate space.

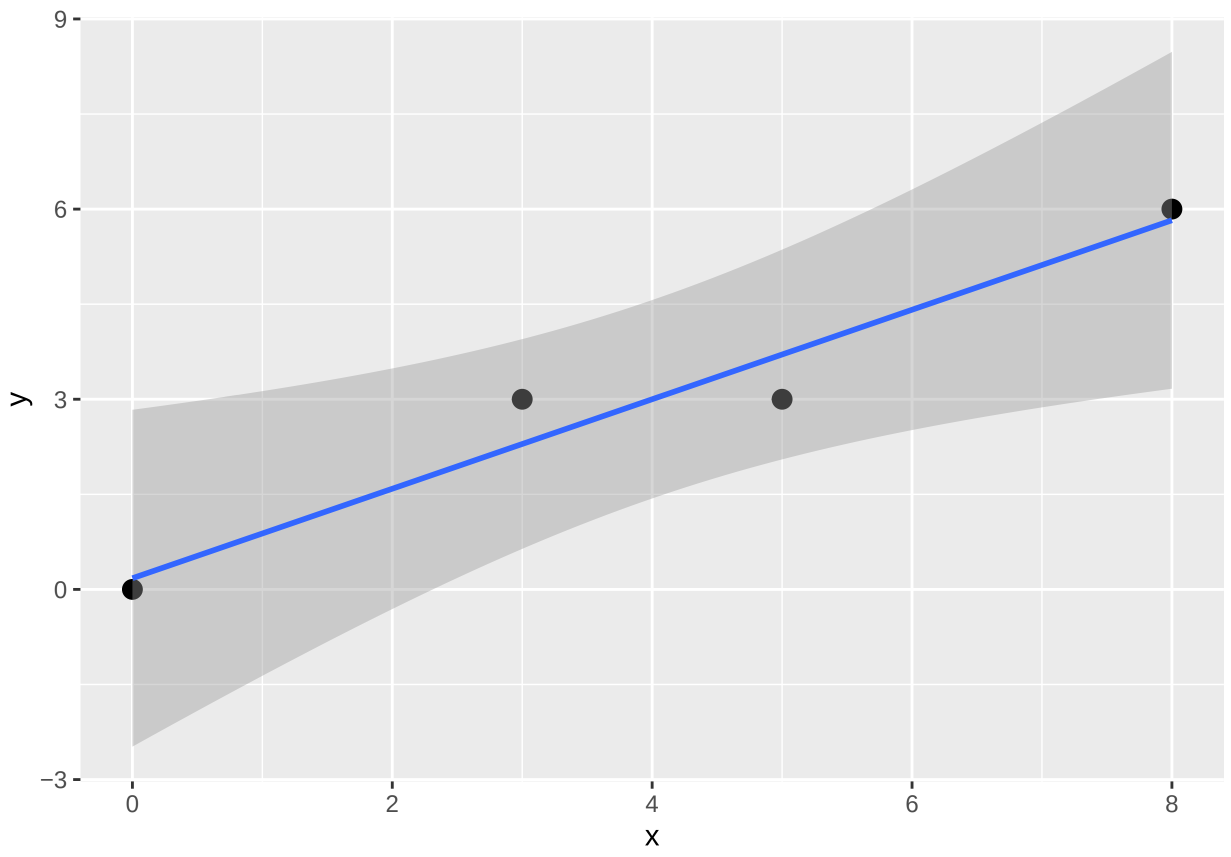
```r
#x/y points
x_points <- c(2,6)
y_points <- c(2,8)
coords1 <- data.frame(x=x_points,y=y_points)

#set bounds of focus area
upper_bound <- 10
lower_bound <- -10
left_bound <- -10
right_bound <- 10

#set limits to extend beyond the bounds
lower_bound_f <- ifelse(lower_bound>0,0,lower_bound)
left_bound_f <- ifelse(left_bound>0,0,left_bound)
upper_bound_f <- ifelse(upper_bound<0,0,upper_bound)
right_bound_f <- ifelse(right_bound<0,0,right_bound)

#plot
simple_plot <- ggplot() +
  geom_point(data=coords1, size=3, aes(x=x, y=y)) +
  xlim(left_bound_f*2, right_bound_f*2) +
  ylim(lower_bound_f*2, upper_bound_f*2) +
  geom_smooth(data=coords1, aes(x=x, y=y),method="lm",fullrange=TRUE,color="black") +
  geom_hline(yintercept=0) +
  geom_vline(xintercept=0) +
  coord_cartesian(xlim = c(left_bound, right_bound),
                  ylim = c(lower_bound, upper_bound),
                  expand = FALSE)
simple_plot
```



Graphing a Linear Equation with 3 or more Points with geom_smooth

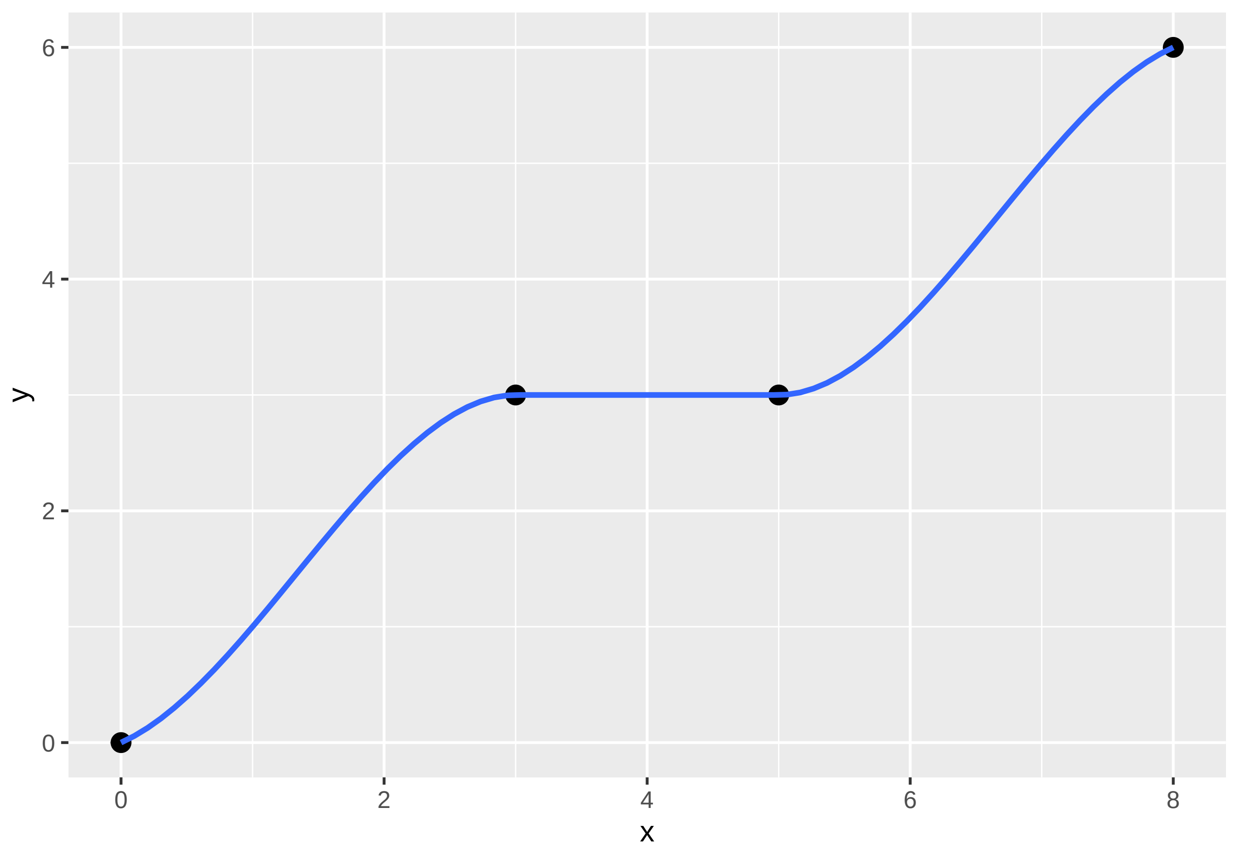
This geom_smooth produces a linear regression line between 3 points. The shaded area represents the standard error of this regression line using a 95% confidence interval by default.

```r
#x/y points
x_points <- c(0,3,5,8)
y_points <- c(0,3,3,6)
coords1 <- data.frame(x=x_points,y=y_points)

#plot
simple_plot <- ggplot(data=coords1, aes(x=x, y=y)) +
  geom_point(size=3) +
  geom_smooth(method="lm")
simple_plot
```



This geom_smooth may also be used to draw a line using a loess kernal method to produce a curved regression line between 3 or more points. Drawing a smoothed curve is not strictly considred part of the discipline of finite maths, but it will be important in later chapters on probability and statistics.

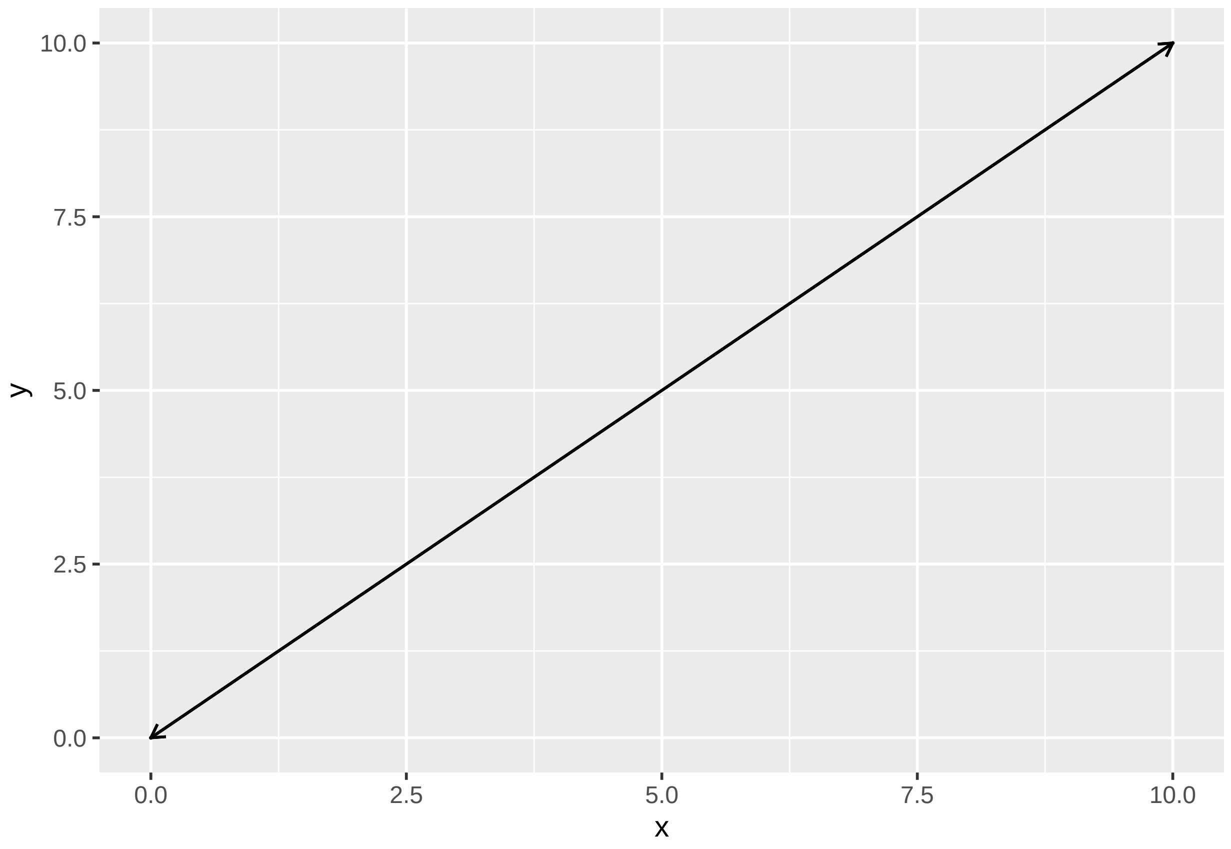
```r
#x/y points
x_points <- c(0,3,5,8)
y_points <- c(0,3,3,6)
coords1 <- data.frame(x=x_points,y=y_points)

#plot
simple_plot <- ggplot(data=coords1, aes(x=x, y=y)) +
  geom_point(size=3) +
  geom_smooth(method="loess")
simple_plot
```



Graphing a line with arrows with geom_segment

Another way to represent an infinite line that continues into an endless space, is to use the geom_segment object. This geom_segment object produces a line between 2 points with an arrowhead on the end. You can add two lines that are both covering the same x/y space, but going opposite directions, with points size set to 0, to simulate the appearance of an infinite line as it might appear in a traditional math text. This approach is a bit cumbersome!

```
x_points <- c(0,10)
y_points <- c(0,10)
coords <- data.frame(x=x_points,y=y_points)

#plot
simple_plot <- ggplot(data=coords, aes(x=x, y=y)) +
  geom_point(size=0) +
  geom_segment(x=coords$x[1],
               xend=coords$x[2],
               y=coords$y[1],
               yend=coords$y[2],
               arrow = arrow(length = unit(.2, 'cm'))) +
  geom_segment(x=coords$x[2],
               xend=coords$x[1],
               y=coords$y[2],
               yend=coords$y[1],
               arrow = arrow(length = unit(.2, 'cm')))
simple_plot
```



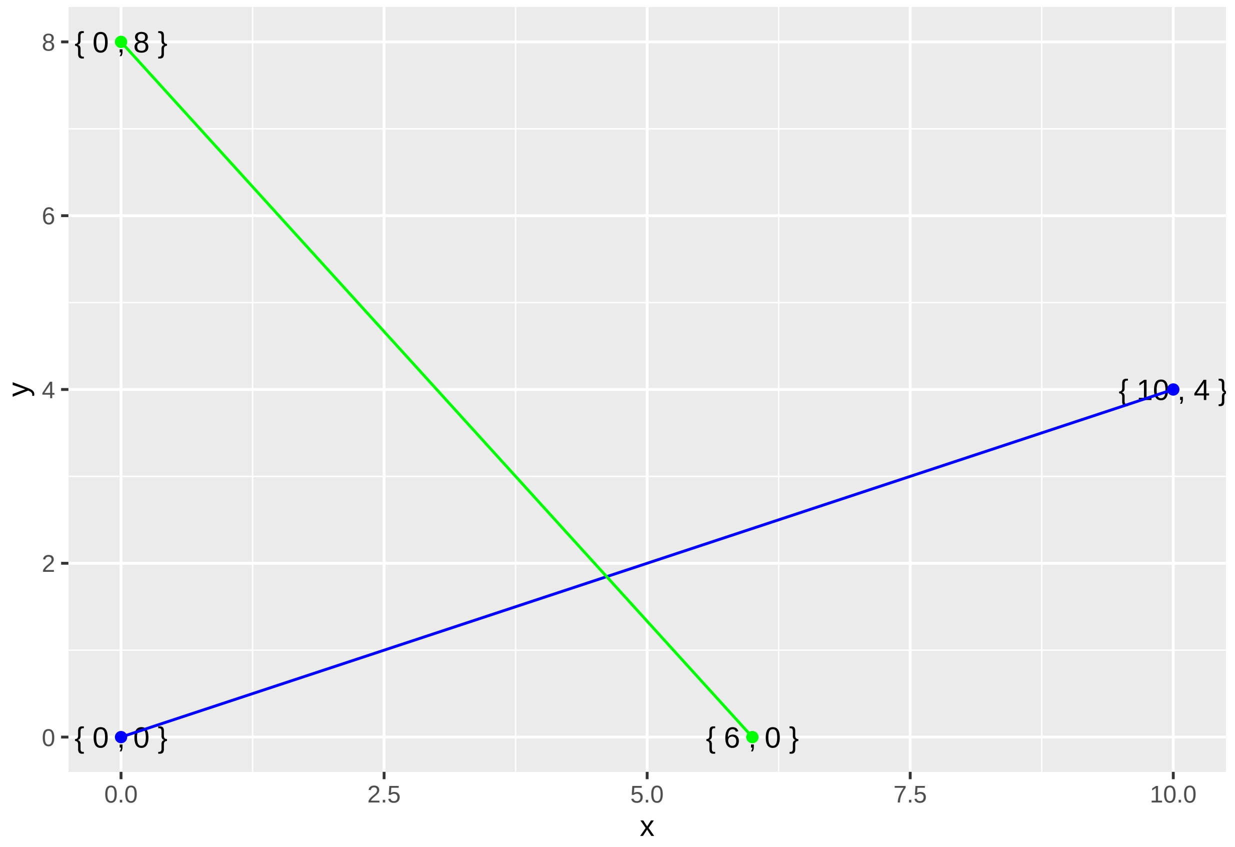
Graphing 2 Lines

Create 2 sets of 2 points and draw a line between them.

In this example, the two equations are being treated as 2 separate *data frames*. This enables us to pass these into the ggplot objects **data** parameter for each element we want to draw. We are also adding some text to show the {x,y} coordinates of each point.

Tech Note: The data parameter in this example has moved out of the ggplot() object, and into the individual geom_ elements. This is because the ggplot constructor does not have one "primary" dataset, but two. Any settings specified in the ggplot() constructor including data or aes mappings, will be inherited by child objects added to the plot via the + operator. Further, any parameters set in the child object will override any setting in the ggplot constructor. If you are familiar with CSS stylesheets - GGplot has similar inheritance structures. You set styles in the GGplot that cascade from a less specific element (GGplot) to a more specific element like geom_line, geom_text, etc.

```r
x_points <- c(0,10)
y_points <- c(0,4)
coords1 <- data.frame(x=x_points,y=y_points)

x_points <- c(0,6)
y_points <- c(8,0)
coords2 <- data.frame(x=x_points,y=y_points)

multi_plot <- ggplot() +
  geom_text(data=coords1, aes(x=x, y=y, label=paste("{",x,",",y,"}")),color="black") +
  geom_point(data=coords1, aes(x=x, y=y),color="blue") +
  geom_line(data=coords1, aes(x=x, y=y),color="blue") +
  geom_text(data=coords2, aes(x=x, y=y, label=paste("{",x,",",y,"}")),color="black") +
  geom_point(data=coords2, aes(x=x, y=y),color="green") +
  geom_line(data=coords2, aes(x=x, y=y),color="green")
multi_plot
```

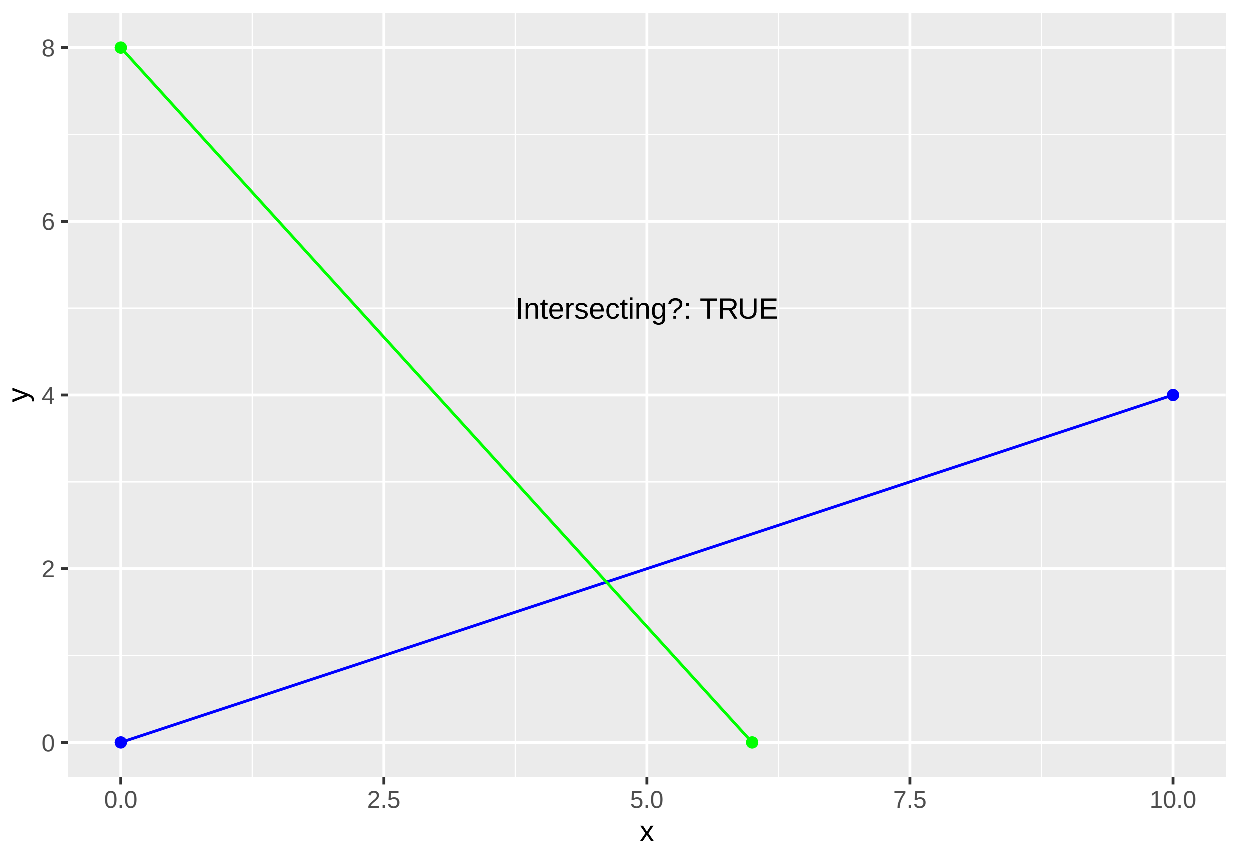


Check Intersection of 2 Lines

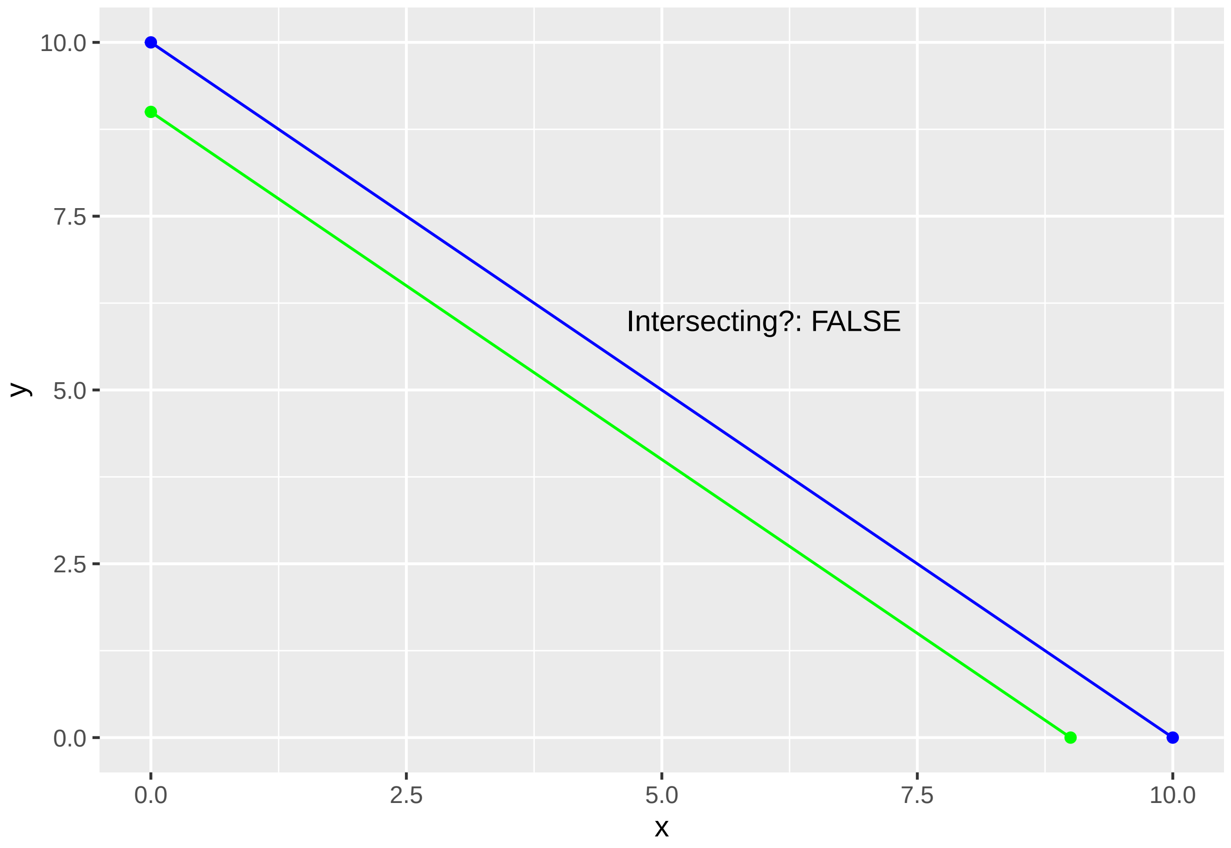
This function will to determine if 2 lines intersect by checking the slope of each line using our slope_line function on both sets of coordinates. If the slope is not eequal - the lines must intersect. The function will return TRUE if the lines intersect.

Tech Note: This function assumes that the lines continue forever in both directions. It is only checking that the slope of the two lines is not equal, not detecting if any two segments overlap.

```
#x/y points
x_points <- c(0,10)
y_points <- c(0,4)
coords1 <- data.frame(x=x_points,y=y_points)

x_points <- c(0,6)
y_points <- c(8,0)
coords2 <- data.frame(x=x_points,y=y_points)

#check if the slopes intersect
check_intersect <- function(coords1,coords2){
  m1 <- slope_line(coords1)
  m2 <- slope_line(coords2)
  is_intersecting <- ifelse(m1 == m2,FALSE,TRUE)
  is_intersecting
}

#run intersect check
intersects <- check_intersect(coords1,coords2)

#plot
multi_plot <- ggplot() +
  geom_point(data=coords1, aes(x=x, y=y),color="blue") +
  geom_line(data=coords1, aes(x=x, y=y),color="blue") +
  geom_point(data=coords2, aes(x=x, y=y),color="green") +
  geom_line(data=coords2, aes(x=x, y=y),color="green") +
  annotate("text", x=5,y=5,label = paste("Intersecting?:",intersects))
multi_plot
```



```r
#x/y points
x_points <- c(0,10)
y_points <- c(10,0)
coords1 <- data.frame(x=x_points,y=y_points)

x_points <- c(0,9)
y_points <- c(9,0)
coords2 <- data.frame(x=x_points,y=y_points)

#run intersect check
intersects <- check_intersect(coords1,coords2)

#plot
multi_plot <- ggplot() +
  geom_point(data=coords1, aes(x=x, y=y),color="blue") +
  geom_line(data=coords1, aes(x=x, y=y),color="blue") +
  geom_point(data=coords2, aes(x=x, y=y),color="green") +
  geom_line(data=coords2, aes(x=x, y=y),color="green") +
  annotate("text", x=6,y=6,label = paste("Intersecting?:",intersects))
multi_plot
```



Systems of Linear Equations

In this chapter we consider how one might solve a system of equations in R using matrix methods.

Single Linear Expression

This code demonstrates how you might set up a linear equation. In the example below - you are trying to determine the values of x1 and x2 for each set, such that eval_linear would evaluate to TRUE. First we will establish the basic logic, then, we will build some code to simulate the steps required to solve a system of linear equations.

```r
#8x + 6y = 3580

#define coefficients
a1 <- 8
a2 <- 6

#define right hand side
b <- 3580
```

```r
#set variables to 0
x1 <- 0
x2 <- 0

set1 <- data.frame(a1=a1,a2=a2,b=b,x1=x1,x2=x2)

#--------

#a linear equation with 2 variables has the form
eval_linear1 <- (set1$a1 * set1$x1) + (set1$a2 * set1$x2) == set1$b

eval_linear1
```

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

Set of Linear Expressions

Expanding on the section above - this code represents the setup of a set of 2 linear equations each having 2 variables.

```r
#8x + 6y = 3580

#define coefficients
a1 <- 8
a2 <- 6

#define right hand side
b <- 3580

#set variables to 0
x1 <- 0
x2 <- 0

set1 <- data.frame(a1=a1,a2=a2,b=b,x1=x1,x2=x2)


#--------

#define coefficients
a1 <- 1
a2 <- 1

#define right hand side
```

```
b <- 525

#set variables to 0
x1 <- 0
x2 <- 0

set2 <- data.frame(a1=a1,a2=a2,b=b,x1=x1,x2=x2)

#--------


#a linear equation with 2 variables has the form
eval_linear1 <- (set1$a1 * set1$x1) + (set1$a2 * set1$x2) == set1$b
eval_linear2 <- (set2$a1 * set2$x1) + (set2$a2 * set2$x2) == set2$b

eval_linear1
```

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

```
eval_linear2
```

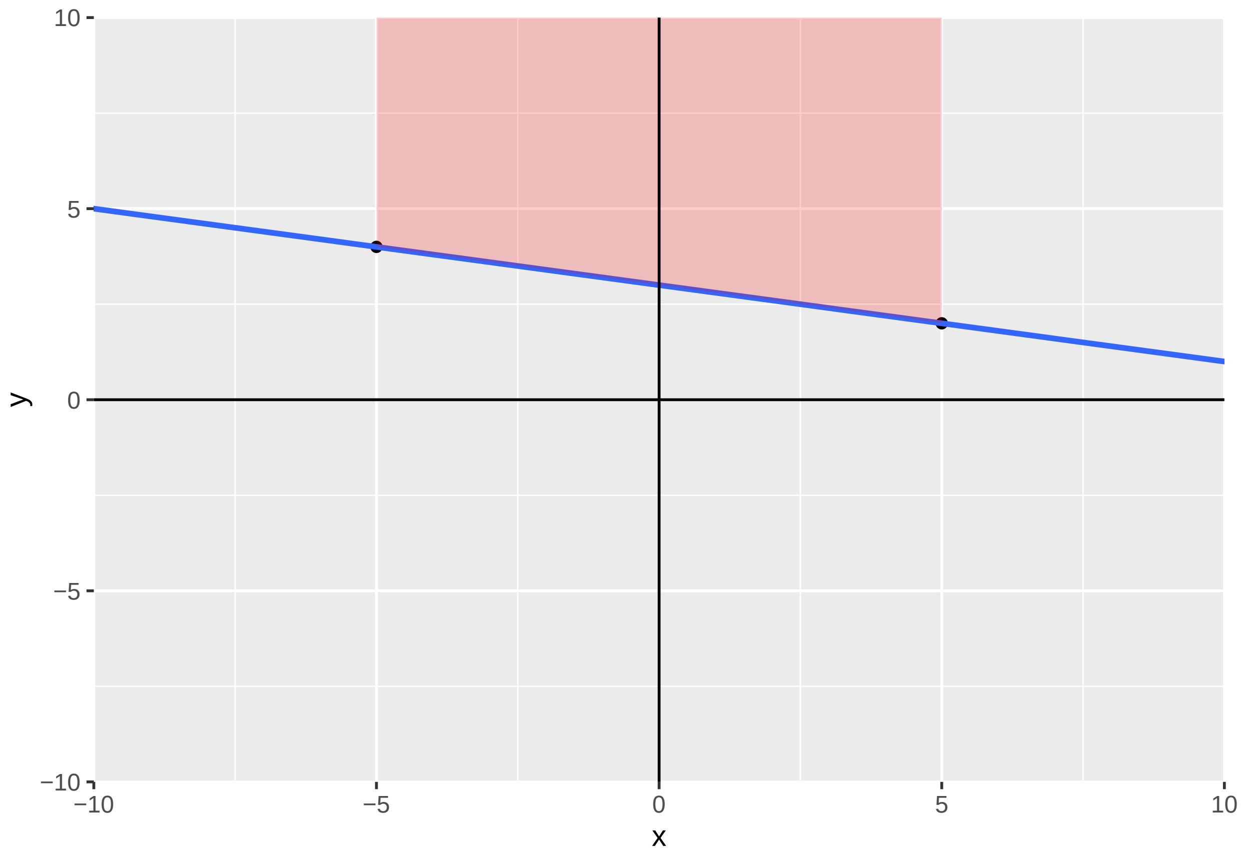
```
## [1] FALSE
```

Shading an Area above a Line

This chart includes a polygon shading of the area above the line. This could be useful for linear programming and optimization problems.

```
buildPoly <- function(xr, yr, slope = 1, intercept = 0, above = TRUE){
    #Assumes ggplot default of expand = c(0.05,0)
    xrTru <- xr + 0.05*diff(xr)*c(-1,1)
    yrTru <- yr + 0.05*diff(yr)*c(-1,1)

    #Find where the line crosses the plot edges
    yCross <- (yrTru - intercept) / slope
    xCross <- (slope * xrTru) + intercept

    #Build polygon by cases
    if (above & (slope >= 0)){
        rs <- data.frame(x=-Inf,y=Inf)
        if (xCross[1] < yrTru[1]){
            rs <- rbind(rs,c(-Inf,-Inf),c(yCross[1],-Inf))
        }
        else{
            rs <- rbind(rs,c(-Inf,xCross[1]))
        }
        if (xCross[2] < yrTru[2]){
            rs <- rbind(rs,c(Inf,xCross[2]),c(Inf,Inf))
        }
        else{
            rs <- rbind(rs,c(yCross[2],Inf))
        }
    }
    if (!above & (slope >= 0)){
        rs <- data.frame(x= Inf,y= -Inf)
        if (xCross[1] > yrTru[1]){
            rs <- rbind(rs,c(-Inf,-Inf),c(-Inf,xCross[1]))
        }
        else{
            rs <- rbind(rs,c(yCross[1],-Inf))
        }
        if (xCross[2] > yrTru[2]){
            rs <- rbind(rs,c(yCross[2],Inf),c(Inf,Inf))
        }
        else{
            rs <- rbind(rs,c(Inf,xCross[2]))
        }
    }
    if (above & (slope < 0)){
        rs <- data.frame(x=Inf,y=Inf)
        if (xCross[1] < yrTru[2]){
            rs <- rbind(rs,c(-Inf,Inf),c(-Inf,xCross[1]))
        }
        else{
            rs <- rbind(rs,c(yCross[2],Inf))
        }
        if (xCross[2] < yrTru[1]){
            rs <- rbind(rs,c(yCross[1],-Inf),c(Inf,-Inf))
        }
        else{
            rs <- rbind(rs,c(Inf,xCross[2]))
        }
    }
    if (!above & (slope < 0)){
        rs <- data.frame(x= -Inf,y= -Inf)
        if (xCross[1] > yrTru[2]){
            rs <- rbind(rs,c(-Inf,Inf),c(yCross[2],Inf))
        }
        else{
            rs <- rbind(rs,c(-Inf,xCross[1]))
        }
        if (xCross[2] > yrTru[1]){
            rs <- rbind(rs,c(Inf,xCross[2]),c(Inf,-Inf))
        }
        else{
            rs <- rbind(rs,c(yCross[1],-Inf))
        }
    }

    return(rs)
}
```

```
#x/y points
x_points <- c(-5,5)
y_points <- c(4,2)
coords1 <- data.frame(x=x_points,y=y_points)

#set bounds of focus area
upper_bound <- 10
lower_bound <- -10
left_bound <- -10
right_bound <- 10

#set limits to extend beyond the bounds
lower_bound_f <- ifelse(lower_bound>0,0,lower_bound)
left_bound_f <- ifelse(left_bound>0,0,left_bound)
upper_bound_f <- ifelse(upper_bound<0,0,upper_bound)
right_bound_f <- ifelse(right_bound<0,0,right_bound)

sl <- diff(coords1$y) / diff(coords1$x)
int <- coords1$y[1] - (sl*coords1$x[1])

#Build the polygon
datPoly <- buildPoly(range(coords1$y),range(coords1$x),
            slope=sl,intercept=int,above=FALSE)

min_x <- min(coords1$x)
max_x <- max(coords1$x)

df_poly_above <- coords1 %>%
  add_row(x = c(max_x, min_x),
                y = c(Inf, Inf))

#plot
simple_plot <- ggplot() +
  geom_point(data=coords1, aes(x=x, y=y)) +
  xlim(left_bound_f*2, right_bound_f*2) +
  ylim(lower_bound_f*2, upper_bound_f*2) +
  geom_smooth(data=coords1, aes(x=x, y=y),method="lm",fullrange=TRUE) +
  #geom_ribbon(data=coords1, aes(ymin = y, ymax = y + 10,x = x), fill = "grey70") +
  geom_polygon(data = df_poly_above,
             aes(x = x, y = y),
             fill = "red",
             alpha = 1/5) +
  geom_hline(yintercept=0) +
  geom_vline(xintercept=0) +
  coord_cartesian(xlim = c(left_bound, right_bound),
                ylim = c(lower_bound, upper_bound),
                expand = FALSE)
simple_plot
```



Matrix Operations

Matrix operations are important in solving systems of equations. The mnatrix construct in R enables a simple vecotor to be converted into an m by n matrix. Standard addition, multiplication, and other arithmetic operations may then be perfomed on the matrices. R's natural handiling of matrix algebra makes it a great tool for mathematical analysis projects.

```r
# generate 2 3x3 matrix objects A and B
A <- matrix(c(1, 2, 3, 1, 3, 2, 3, 2, 1), 3, 3, byrow = TRUE)
B <- matrix(c(1, 2, 3, 1, 3, 2, 3, 2, 1), 3, 3, byrow = TRUE)
A
```

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

```r
B
```

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

```r
# get the additive matrix sums
AplusB <- A + B
AplusB
```

```
##      [,1] [,2] [,3]
## [1,]    2    4    6
## [2,]    2    6    4
## [3,]    6    4    2
```

```r
# get the multiplicative matrix product
AtimesB <- A * B
AtimesB
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    9
## [2,]    1    9    4
## [3,]    9    4    1
```

```
# generate 2 3x3 matrix objects A and B
A <- matrix(c(1, 2, 3, 2, 3, 2, 3, 2, 1), 3, 3, byrow = TRUE)
A
```

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

```
#perform a row operation
r1 <- A[1,]
r2 <- A[2,]
r3 <- A[3,]

R2 <- (r1 * -2) + r2
A[2,] <- R2
A
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    0   -1   -4
## [3,]    3    2    1
```

```
#perform another row operation
r1 <- A[1,]
r2 <- A[2,]
r3 <- A[3,]

R3 <- (r1 * -3) + r3
A[3,] <- R3
A
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    0   -1   -4
## [3,]    0   -4   -8
```

```
#perform another row operation
r1 <- A[1,]
r2 <- A[2,]
r3 <- A[3,]

R3 <- (r2 * -4) + r3
A[3,] <- R3
A
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    0   -1   -4
## [3,]    0    0    8
```

We are not going to focus very heavily on Matrices in this report - there are R packages available that can provide most matrix operation features. For example, in the pracma package, the rref function can be used to to find the reduced row echelon form of a matrix using gaussian elimination. An example is shown below, which was taken from the documentation for the pracma package.

```
#FROM: https://www.rdocumentation.org/packages/pracma

# generate a 3x3 matrix and solve
A <- matrix(c(1, 2, 3, 1, 3, 2, 3, 2, 1), 3, 3, byrow = TRUE)
A
```

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

```
rref(A)
```

```
##      [,1] [,2] [,3]
## [1,]    1    0    0
## [2,]    0    1    0
## [3,]    0    0    1
```

```
# generate a 3x4 matrix and solve
A <- matrix(data=c(1, 2, 3, 2, 5, 9, 5, 7, 8,20, 100, 200),
            nrow=3, ncol=4, byrow=FALSE)
A
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    5   20
## [2,]    2    5    7  100
## [3,]    3    9    8  200
```

```
rref(A)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    0    0  120
## [2,]    0    1    0    0
## [3,]    0    0    1  -20
```

```
# Use rref on a rank-deficient magic square:
A = magic(4)
A
```

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

```
R = rref(A)
zapsmall(R)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    0    0    1
## [2,]    0    1    0    3
## [3,]    0    0    1   -3
## [4,]    0    0    0    0
```

Quick Tip:

The default matrix notation in LaTeX uses the bmatrix:

$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$

However the above function does not appear to support the vertical bar. To represent a matrix with a right hand side separated by a vertical bar in LaTeX use the array function surrounded by left and right brackets:

$$\left[\begin{array}{ccc|c} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 3 \\ 0 & 0 & 1 & -3 \end{array}\right]$$

Linear Programming with 2 Variables

Lets try an application of the plots above to a the "Vitamin Manufacturer" problem you will see in chapter 4.

Application: Donut Shop

We will take a donut shop as an example. A baker has 2 award winning donut recipes that both use chocolate and rainbow sprinkles.

```
#name the object you will be studying
subject <- "Donut"
madeby <- "Bakery"

#name your recipes/mixtures x and y
recipe_x <- "Chocolate Dream Donut"
recipe_y <- "Birthday Surprise Donut"

#define x and y for plots
x_def <- paste("Number of",recipe_x)
y_def <- paste("Number of",recipe_y)

#two ingredients available to mix in a recipe
ing1_def <- "Chocolate Sprinkles"
ing2_def <- "Rainbow Sprinkles"

#the maximum amount of each ingredient available using same units as below
max_ing1_g <- 3600
max_ing2_g <- 2400
```

```
#number of items produced per batch
item_count <- 100

#the amount of ingredients 1 and 2 needed by recipe x using same units as above
requires_ing1_x <- 120
requires_ing2_x <- 40

#the amount of ingredients 1 and 2 needed by recipe y using same units as above
requires_ing1_y <- 60
requires_ing2_y <- 120

#the profit per item on recipe x
recipe_x_profit <- .22

#the profit per item on recipe y
recipe_y_profit <- .18

wordproblem <- paste("A ",madeby," has ", max_ing1_g," grams of ", ing1_def," and ", max

wordproblem
```

```
## [1] "A Bakery has 3600 grams of Chocolate Sprinkles and 2400 grams of Rainbow Sprinkl
```

A Bakery has 3600 grams of Chocolate Sprinkles and 2400 grams of Rainbow Sprinkles available per day. Each batch of the Chocolate Dream Donut recipe takes 120 grams of Chocolate Sprinkles and 40 grams of Rainbow Sprinkles. Each batch of the Birthday Surprise Donut recipe takes 60 grams of Chocolate Sprinkles and 120 grams of Rainbow Sprinkles. The Chocolate Dream Donut generates a profit of 0.22 per Donut, while the Birthday Surprise Donut generates 0.18 profit per Donut. Each batch produces 100 Donuts. How can the company maximize profit assuming all items will sell out? What will the total profit be?

In the example above, we have set up the variables in the code block, and then placed them inline into a word problem format. This allows the RMarkdown file to create meaningful interpretations of data in the report body.

The challenge here is to frame the problem and solve it using R and RMarkdown.

First we will define the objective to be maximized: P (Profit)

In this linear programming problem with 2 variables, P = 0.22 * X + 0.18 * Y

Subject to the constraints:

120 * X + 60 * Y <= 3600

40 * X + 120 * Y <= 2400

X >= 0

Y >= 0

First, let's see how we can visualize the problem using the charting strategies demonstrated above.

The shaded area plots are useful for this type of problem. We need to see where the set of *feasible points* is.

Using the above code we can start by drawing the first of the constraints on to the chart.

Assuming X = 0 then Y <= 60

Assuming Y = 0 then X <= 30

```
coords0 <- data.frame(x=0,y=0)

#setup the first set of coordinates
first_x <- max_ing1_g / requires_ing1_y
first_y <- max_ing1_g / requires_ing1_x

#x/y points
x_points <- c(0,first_y)
y_points <- c(first_x,0)
coords1 <- data.frame(x=x_points,y=y_points)

paste("coords1:",coords1)
```

```
## [1] "coords1: c(0, 30)" "coords1: c(60, 0)"
```

```
#setup the second set of coordinates
second_x <- max_ing2_g / requires_ing2_y
second_y <- max_ing2_g / requires_ing2_x

#x/y points
x_points <- c(0,second_y)
y_points <- c(second_x,0)
coords2 <- data.frame(x=x_points,y=y_points)

paste("coords2:",coords2)
```

```
## [1] "coords2: c(0, 60)" "coords2: c(20, 0)"
```

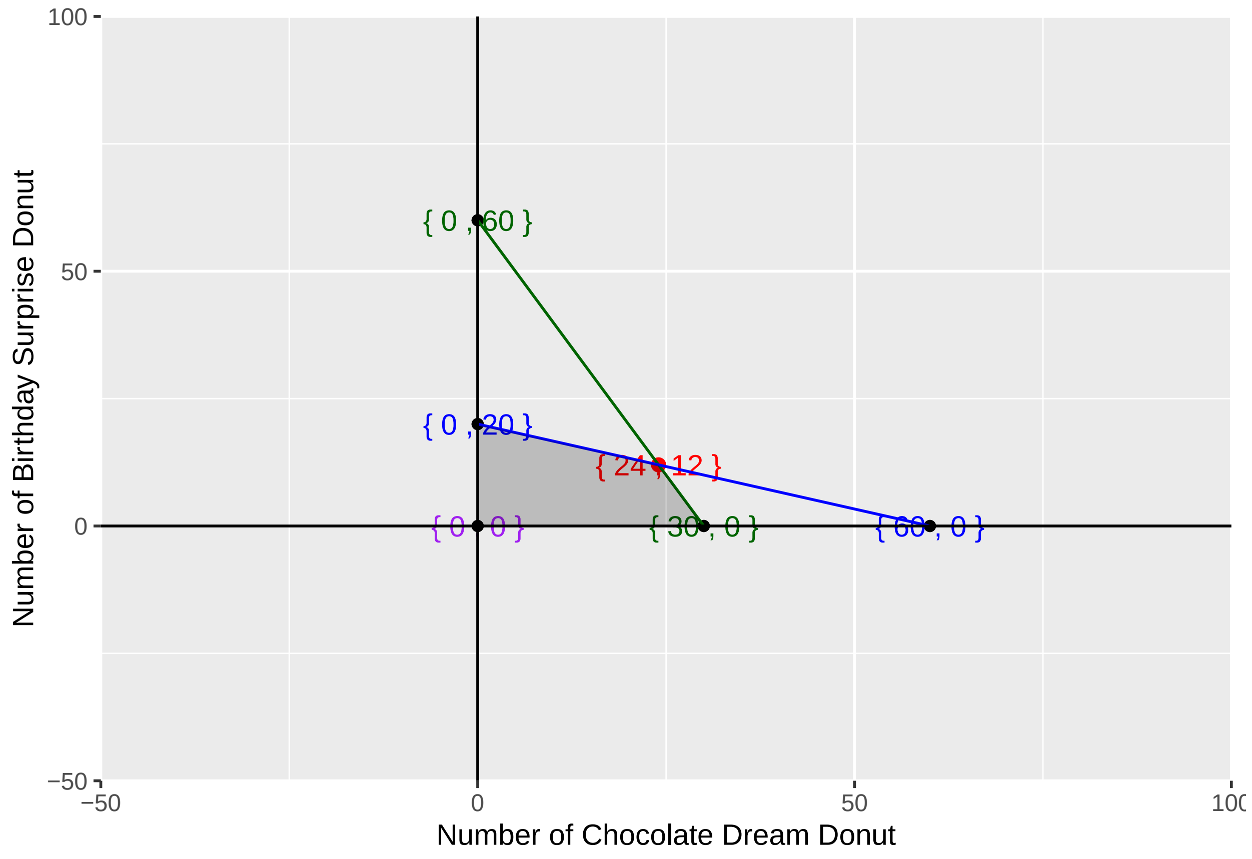
```
# generate a 2x3 matrix and solve for the intersection point
A <- matrix(c(requires_ing1_x, requires_ing1_y, max_ing1_g,
              requires_ing2_x, requires_ing2_y, max_ing2_g), 2, 3, byrow = TRUE)
A
```

```
##      [,1] [,2] [,3]
## [1,]  120   60 3600
## [2,]   40  120 2400
```

```
#use pracma for the rref
R <- rref(A)
R
```

```
##      [,1] [,2] [,3]
## [1,]    1    0   24
## [2,]    0    1   12
```

```
#assign to data frame intersectionpoint
intersectionpoint <- data.frame(x=R[1,3],y=R[2,3])

#set bounds of focus area
upper_bound <- 100
lower_bound <- -50
left_bound <- -50
right_bound <- 100

#list & arrange feasible points
points <- data.frame()
points <- points %>% bind_rows(coords0)
points <- points %>% bind_rows(coords1[2,])
points <- points %>% bind_rows(coords2[1,])
points <- points %>% bind_rows(intersectionpoint)
points <- points %>% arrange(x,y)

#plot
simple_plot <- ggplot() +
  geom_point(data=coords0, aes(x=x, y=y)) +
  geom_point(data=coords1, aes(x=x, y=y)) +
  geom_point(data=coords2, aes(x=x, y=y)) +
  geom_point(data=intersectionpoint, aes(x=x, y=y),size=2,color="red") +
  geom_text(data=intersectionpoint, aes(x=x, y=y, label=paste("{",x,",",y,"}")),color="r
  geom_text(data=coords0, aes(x=x, y=y, label=paste("{",x,",",y,"}")),color="purple") +
  geom_text(data=coords1, aes(x=x, y=y, label=paste("{",x,",",y,"}")),color="darkgreen")
  geom_text(data=coords2, aes(x=x, y=y, label=paste("{",x,",",y,"}")),color="blue") +
  geom_line(data=coords1, aes(x=x, y=y),color='darkgreen') +
  geom_line(data=coords2, aes(x=x, y=y),color='blue') +
  geom_polygon(data=points, aes(x=x, y=y),fill='black',alpha=.2) +
  geom_hline(yintercept=0) +
  geom_vline(xintercept=0) +
  coord_cartesian(xlim = c(left_bound, right_bound),
                  ylim = c(lower_bound, upper_bound),
                  expand = FALSE) +
  labs(x=x_def,y=y_def)
simple_plot
```



```
#select the feasible points. This process may require additional
#analysis of the plot to determine which points are feasible.
#0,0 has been omitted, and we are rounding to the nearest whole number.
#in practice it may be necessary to use floor rounding (always round down) to avoid ov

#get first feasible point
location0x <- round(0,0)
```

```r
location0y <- round(0,0)
outcome0 <- (location0x * recipe_x_profit + location0y * recipe_y_profit) * item_count
row0 <- data.frame(x=location0x,y=location0y,total=outcome0)

#get first feasible point
location1x <- round(coords1[2,]$x,0)
location1y <- round(coords1[2,]$y,0)
outcome1 <- (location1x * recipe_x_profit + location1y * recipe_y_profit) * item_count
row1 <- data.frame(x=location1x,y=location1y,total=outcome1)

#get second feasible point
location2x <- round(coords2[1,]$x,0)
location2y <- round(coords2[1,]$y,0)
outcome2 <- (location2x * recipe_x_profit + location2y * recipe_y_profit) * item_count
row2 <- data.frame(x=location2x,y=location2y,total=outcome2)

#get third feasible point
location3x <- round(intersectionpoint[1,]$x,0)
location3y <- round(intersectionpoint[1,]$y,0)
outcome3 <- (location3x * recipe_x_profit + location3y * recipe_y_profit) * item_count
row3 <- data.frame(x=location3x,y=location3y,total=outcome3)

#build a the table of outcomes
outcomes <- data.frame()
outcomes <- outcomes %>% bind_rows(row0)
outcomes <- outcomes %>% bind_rows(row1)
outcomes <- outcomes %>% bind_rows(row2)
outcomes <- outcomes %>% bind_rows(row3)

#return the table of outcomes ordered by the total
outcomes %>% arrange(-total)
```

```
##    x  y total
## 1 24 12   744
## 2 30  0   660
## 3  0 20   360
## 4  0  0     0
```

```r
#assign the best outcome to bestoutcome
bestoutcome <- outcomes %>% arrange(-total) %>% slice(1)

#set some variables
bestx <- bestoutcome$x
besty <- bestoutcome$y
```

```r
besttotal<- bestoutcome$total

#show the result.
result <- paste("To maximize profits, you should make ",bestx," batches of ",recipe_x,"
```

> To maximize profits, you should make 24 batches of Chocolate Dream Donut and 12 batches of Birthday Surprise Donut for a total profit of $744

We have completed the linear programing model and determined the total number of each batch that should be produced to maximize profit given the constraints provided, and plotted the key details on a GGPlot.

Finance

In this section we will set up a few equations in R that will perform some of the standard financial formula operations, similar to the linear programming problem we worked out above.

This section contains several key formulas related to interest, annuities, amortization, and sinking funds.

Simple Interest

The simple interest formula is as it sounds - simple!

$I = Prt$

Also, the amount owed is a fairly simple addition to this formula:

$A = P + I$

```r
#create simple interest function
simpleinterest <- function (P,r,t){
  r <- r/100
  I <- P * r * t
  I
}

#amount of principal
P <- 2000

#% rate of loan
r <- 5

#number of years
t <- 3

#interest (unknown)
I <- 0

#run the function
I <- simpleinterest(P,r,t)

# A = Principal plus Interest
A <- P + I

#write a sentence to summarize the result
result <- paste("The simple interest on a loan of $",P," at an annual interest rate of "
#result
```

The simple interest on a loan of $2000 at an annual interest rate of 5%, after 3 years, is: $300 and the total amount due is: $2300

Discounted Loans

Let r be the per annum rate of interest, t the time in years and L the amount of the loan. The proceeds R is is given by:

$R = L - Lrt = L(1 - rt)$

```r
#create discounted loan function
discountedloan <- function (L,r,t){
  r <- r/100
  R <- L * (1- (r * t))
  R
}

#amount of loan
L <- 10000

#% rate of loan
r <- 8

#number of years
t <- .25

#run the function
R <- discountedloan(L,r,t)

#write out the result
result <- paste("The proceeds of a discounted loan of $",L," at an annual interest rate

#result
```

The proceeds of a discounted loan of $10000 at an annual interest rate of 8%, after 0.25 years is: $9800

Amount of a Discounted Loan

The amount needed to repay a discounted loan can be found using:

$(\frac{L}{R})A$

```
#nested function
discountedloanamount <- function (L,r,t){
  R <-discountedloan(L,r,t)
  A <- L / R
  A <- L * A
  A
}

#run the function
A <- discountedloanamount(L,r,t)

result <- paste("A discounted loan of $",L," at an annual interest rate of ",r,"%, after

#result
```

A discounted loan of \$10000 at an annual interest rate of 8%, after 0.25 years, will require repayment in the amount of: \$10204.08

Compound Interest

The formula for compount intgerest can be defined as:

The amount A after t years due to a principal P invested ant an annual interest rate r compounded n times per year is:

$A = P * (1 + \frac{r}{n})^{nt}$

```r
#create discounted loan function
compoundinterest <- function (P,r,t,n){
 r <- r/100
 i <- (1+(r/n))^(n*t)
 A <- P * i
}

#amount of loan
P <- 300000

#% rate of loan
r <- 3.2

#number of years
t <- 30

#number of years
n <- 12

A <- compoundinterest(P,r,t,n)

result <- paste("A loan of $",P," at an annual interest rate of ",r,"%, after ",t," year

#result
```

A loan of \$3e+05 at an annual interest rate of 3.2%, after 30 years, compounding 12 times per year will cost a total of: \$782508.47

Continuous Compounding Interest

The amount A after t years due to a principal P invested ant an annual interest rate r compounded continuously is:

$A = Pe^{nt}$

Where e is Eulers constant (~ 2.718281827), which can be assigned precisely using the R function exp(1).

```r
#create simple interest function
continuouscompoundinterest <- function (P,r,t){
  r <- r/100
  e <- exp(1)
  I <- (P * e)^(r * t)
  I
}

#amount of principal
P <- 2000

#% rate of loan
r <- 8

#number of years
t <- 1

#interest (unknown)
I <- 0

#run the function
I <- continuouscompoundinterest(P,r,t)

# A = Principal plus Interest
A <- P + I

#write a sentence to summarize the result
result <- paste("A loan of $",P," at an annual interest rate of ",r,"%, after ",t," year

result
```

```
## [1] "A loan of $2000 at an annual interest rate of 8%, after 1 years, compounding con
```

A loan of \$2000 at an annual interest rate of 8%, after 1 years, compounding continuously will cost a total of: \$2001.99

Present Value

The present value P of A dollars to be received after t years assuming a per annum interest rate r compounded n times per year is given by:

$P = A \cdot (1 + \frac{r}{n})^{-nt}$

```r
#create simple interest function
presentvalue <- function (A,r,n,t){
  r <- r/100
  i <- r / n
  P <- (1+i)^(-n*t)
  P * A

}

#amount we want to get to
A <- 900000

#% rate of return on loan
r <- 6

# number of payments
n <- 12

# time in years
t <- 30

#interest (unknown)
P <- 0

#run the function
P <- presentvalue(A,r,n,t)

#write a sentence to summarize the result
result <- paste("The amount that should be invested now at ",r,"% per annum to reach a s

result
```

```
## [1] "The amount that should be invested now at 6% per annum to reach a savings goal o
```

The amount that should be invested now at 6% per annum to reach a savings goal of $9e+05 after 30 years with interest compounding 12 times per year is: $149437.74

Present Value with Continuous Compounding

The present value P of A dollars to be received after t years assuming a per annum interest rate r with continuous compounding is given by:

$P = A \cdot (1 + \frac{r}{n})^{-nt}$

```
#create simple interest function
presentvaluecontinuous <- function (A,r,t){
  r <- r/100
  e <- exp(1)
  P <- A * e ^ -(r * t)
  P

}

#amount we want to get to
A <- 10000

#% rate of return on loan
r <- 4

# time in years
t <- 2

#interest (unknown)
P <- 0

#run the function
P <- presentvaluecontinuous(A,r,t)

#write a sentence to summarize the result
result <- paste("The amount that should be invested now at ",r,"% per annum to reach a s

result
```

```
## [1] "The amount that should be invested now at 4% per annum to reach a savings goal o
```

The amount that should be invested now at 4% per annum to reach a savings goal of $10000 after 2 years with interest compounding continuously is: $9231.16

Amount of an Annuity

Assuming P is the deposit made at the end of each payment period for an annuity paying an interest rate of i per payment period. The amount A of an annuity after n deposits is given by:

$A = P \cdot [\frac{(1+i)^n - 1}{i}]$

Assuming that $i = \frac{r}{n}$ where r is the annual rate and n is the number of compounding periods per year.

```
#create simple interest function
amountofannuity <- function (P,r,y,n){
  r <- r/100
  i <- r / y
  z <- ((1+i)^n)-1
  A <- P * (z/i)
  A
}

#amount of principal
P <- 100

#% rate of loan
r <- 5

#number of compounding period per year
y <- 12

# number of payments
n <- 8

#interest (unknown)
A <- 0

#run the function
A <- amountofannuity(P,r,y,n)

# A = Principal plus Interest
A <- A

#write a sentence to summarize the result
result <- paste("After making ",n," deposits of $",P," each, an annuity paying ",r,"%, c

result
```

```
## [1] "After making 8 deposits of $100 each, an annuity paying 5%, compounded 12 times
```

After making 8 deposits of $100 each, an annuity paying 5%, compounded 12 times per year, is worth: $811.76

Present Value of an Annuity

Suppose an annuity earns at an interest rate of i per payment period. If n withdrawals of P are made at each payment period, the amount V required is:

$V = P \cdot [\frac{1-(1+i)^{-n}}{i}]$

Assuming that $i = \frac{r}{n}$ where r is the annual rate and n is the number of compounding periods per year.

```
#create simple interest function
presentvalueofannuity <- function (P,r,y,n){
  r <- r/100
  i <- r / y
  z <- (1-(1+i)^-n)
  V <- P * (z/i)
  V
}

#amount of principal
P <- 1000

#% rate of loan
r <- 5

#number of compounding period per year
y <- 1

# number of payments
n <- 4

#interest (unknown)
V <- 0

#run the function
V <- presentvalueofannuity(P,r,y,n)

# A = Principal plus Interest
V <- V

#write a sentence to summarize the result
result <- paste("To make ",n," payments of $",P," each, while receiving the proceeds of
options(scipen=999)
result
```

```
## [1] "To make 4 payments of $1000 each, while receiving the proceeds of an annuity pay
```

To make 4 payments of \$1000 each, while receiving the proceeds of an annuity paying 5%, compounding 1 times per year, the amount needed to invest in the annuity now is: \$3545.95

Amortization

The payment P required to pay off a loan of V dollars borrowed for n payment periods at a rate of interest i per payment period is given by:

$P = V \cdot [\frac{i}{1-(1+i)^{-n}}]$

Assuming that $i = \frac{r}{n}$ where r is the annual rate and n is the number of compounding periods per year.

```
#create simple interest function
amortization <- function (V,r,y,n){
  r <- r/100
  i <- r / y
  z <- (1-(1+i)^-n)
  P <- V * (i/z)
  P
}

#amount of principal
V <- 500000

#% rate of loan
r <- 4

#number of compounding period per year
y <- 12

# number of payments
n <- 12 * 30

#interest (unknown)
P <- 0

#run the function
P <- amortization(V,r,y,n)

#write a sentence to summarize the result
result <- paste("The payment required to pay off a loan of ",V," dollars borrowed for ",
options(scipen=999)
result
```

```
## [1] "The payment required to pay off a loan of 500000 dollars borrowed for 360 paymen
```

The payment required to pay off a loan of 500000 dollars borrowed for 360 payment periods at a rate of interest 4 is: 2387.07647732727

Double an Investment

The number of years t required to multiply an investment m times when r is the interest rate compounded n times per year:

$t = \frac{\ln(m)}{n \cdot \ln(1+\frac{r}{n})}$

```r
#create simple interest function
multiplyinvestment <- function (m,r,n){
  r <- r/100
  top <- log(m,base=exp(1))
  bot <- n * log(1+(r/n),base=exp(1))
  t <- (top/bot)
  t
}


# % interest rate
r <- 4

# number of times to multiple initial investment
m <- 2

# number of compound periods per year
n <- 1

# time
t <- 0

#run the function
t <- multiplyinvestment(m,r,n)


#write a sentence to summarize the result
result <- paste("To multiply an investment by ",m," times, when ",r,"% is the interest r

result
```

```
## [1] "To multiply an investment by 2 times, when 4% is the interest rate compounded 1
```

To multiply an investment by 2 times, when 4% is the interest rate compounded 1 times per year, it will take: 17.7 years.

Time to Reach a Savings Goal with Continuous Compounding

The number of years t required to multiply an investment m times when r is the interest rate compounded n times per year:
$t = \frac{\ln(A) \cdot \ln(P)}{r}$

```r
#create simple interest function
timetoreachgoal <- function (A,P,r){
  r <- r/100
  ln_a <- log(A,base=exp(1))
  ln_p <- log(P,base=exp(1))
  t <- (ln_a - ln_p) / r

}


# goal amount
A <- 100000

# initial investment
P <- 20000

# interest rate
r <- 5

# time
t <- 0

#run the function
t <- timetoreachgoal(A,P,r)


#write a sentence to summarize the result
result <- paste("It will take ",t," years to reach a savings goal of ",A,", earning ",r,

result
```

```
## [1] "It will take 32.188758248682 years to reach a savings goal of 100000, earning 5%
```

It will take 32.188758248682 years to reach a savings goal of 100000, earning 5% interest compounding continuously on a principal investment of $20000.

Compute Inflation

If the rate of inflation averages r over n years, the the CPI index after n years is:

$A = P \cdot (1 - r)^n$

```r
#create simple interest function
computeinflation <- function (P,r,n){
  r <- r/100
  A <- P * (1-r)^n
  A

}

#principal
P <- 1000

# % avg interest rate
r <- 2

# number of year
n <- 3

#run the function
A <- computeinflation(P,r,n)

#write a sentence to summarize the result
result <- paste("Assuming an average annual inflation rate of ",r,"%, $",P," after " ,n,

result
```

```
## [1] "Assuming an average annual inflation rate of 2%, $1000 after 3 years will be wor
```

> Assuming an average annual inflation rate of 2%, $1000 after 3 years will be worth 941.19.

Compute the Consumer Price Index (CPI)

If the rate of inflation averages r over n years, the the CPI index after n years is:

$CPI = CPI_0(1 + \frac{r}{100})$

```r
#create simple interest function
computecpi <- function (cpi0,r,n){
  cpi <- cpi0 *((1+(r/100))^n)
  cpi

}
#CPI start year
cpiyear0 <- 2010

#CPI in 2010
cpi0 <- 217.6

# % avg interest rate
r <- 3

# number of year
n <- 10

# second year
cpiyear1 <- cpiyear0 + n

# time
cpi1 <- 0

#run the function
cpi1 <- computecpi(cpi0,r,n)


#write a sentence to summarize the result
result <- paste("Assuming an average annual inflation rate of ",r,"%, the CPI in ",cpiye

result
```

```
## [1] "Assuming an average annual inflation rate of 3%, the CPI in 2020 (after 10 years
```

Assuming an average annual inflation rate of 3%, the CPI in 2020 (after 10 years) would be 292.4 given a starting CPI of 217.6 in 2010.

Sets

In this section we will set up functions to perform set operations.

```r
setoperations <- function (A,B){

#Set teh length variables
A_len <- length(A)
B_len <- length(B)

#get teh union
A_un_B <- unique(c(A,B))

#get the intersection
A_in_B <- A[which((A%in%B))]
B_in_A <- B[which((B%in%A))]

#determine set rules
A_sub_B <- FALSE
B_sub_A <- FALSE
B_super_A <- FALSE
A_super_B <- FALSE
A_psub_B <- FALSE
B_psub_A <- FALSE
B_psuper_A <- FALSE
A_psuper_B <- FALSE
A_disjoint_B <- FALSE

if(length(A_in_B)>0){
  #if n(A) is < n(B)
  if(A_len < B_len){
    #If the elements of A that are in B are all the elements of A
    if(length(A_in_B) == length(A)){
    #A is a proper subset of B
    A_psub_B <- TRUE
    B_psuper_A <- TRUE
    }
  }
  #if n(A) is < n(B)
  if(A_len <= B_len){
    #If the elements of A that are in B are all the elements of A
    if(length(A_in_B) == length(A)){
    #A is a proper subset of B
    A_sub_B <- TRUE
    B_super_A <- TRUE
    }
  }

  #if n(A) is < n(B)
  if(B_len < A_len){
    #If the elements of A that are in B are all the elements of A
    if(length(B_in_A) == length(B)){
    #A is a proper subset of B
    B_psub_A <- TRUE
    A_psuper_B <- TRUE
    }
  }
  #if n(A) is < n(B)
  if(B_len <= A_len){
    #If the elements of A that are in B are all the elements of A
    if(length(B_in_A) == length(B)){
    #A is a proper subset of B
    B_sub_A <- TRUE
    A_super_B <- TRUE
    }
  }

}
if(length(A_in_B) == 0){
  A_disjoint_B <- TRUE
}

A_intersect_B <- A_in_B
A_union_B <- A_un_B

return_obj <- list(
  A = A,
  B = B,
  A_in_B = A_sub_B,
  B_in_A = B_sub_A,
  A_sub_B = A_sub_B,
  B_sub_A = B_sub_A,
  A_super_B = A_super_B,
  B_super_A = B_super_A,
  A_psub_B = A_psub_B,
  B_psub_A = B_psub_A,
  A_psuper_B = A_psuper_B,
  B_psuper_A = B_psuper_A,
  A_union_B = A_union_B,
  A_intersect_B = A_intersect_B,
  A_disjoint_B = A_disjoint_B

)

return_obj

}

A <- unique(round(rnorm(n=50,mean=100,sd=50),0))
B <- unique(round(rnorm(n=50,mean=100,sd=50),0))
set_ops <- setoperations(A,B)
```

From the two vectors A and B defined above we can show the following set equalities in LaTeX:

Proper Subset / Superset Equalities:

$A \subset B$ FALSE

$B \subset A$ FALSE

$A \supset B$ FALSE

$B \supset A$ FALSE

Subset / Superset Equalities:

$A \subseteq B$ FALSE

$B \subseteq A$ FALSE

$A \supseteq B$ FALSE

$B \supseteq A$ FALSE

Intersection of A and B:

$A \cap B$ 32, 94, 17, 20, 106, 108, 72, 85

Union of A and B:

$A \cup B$ 147, 190, 37, 145, 32, 94, 93, 111, 88, 10, 134, 21, 47, 118, 95, 76, 17, 138, 124, 136, 196, 20, -2, 74, 84, 122, 33, 31, 149, 48, 87, 60, 80, 62, 155, 30, 78, 106, 108, 154, 133, 79, -14, 72, 85, 69, 109, 158, 82, 127, 68, 56, 139, 192, 141, 181, 27, 203, 200, 41, 107, 70, 90, -25, 110, 35, 83, 75, 120, 123, 39, 105, 59, 50, 191, 89, 7, 92, 172, 101, 65, 52, 13

Are these sets Disjoint?

FALSE

Counting Formula

The counting formula holds that:

$n(A \cup B) = n(A) + n(B) - n(A \cap B)$

We will prove that our set operations function is correct by using the counting formula. If the set opration is correct then the same_length should be TRUE between A_union_B and the product of the counting formula.

```
len_A_union_B_1 <- length(set_ops$A) + length(set_ops$B) - length(set_ops$A_intersect_B
len_A_union_B_2 <- length(set_ops$A_union_B)

same_length <- len_A_union_B_1 == len_A_union_B_2
same_length
```

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

The counting formula agrees with the length of A union B.

De Morgan's Properties

De morgans properties provide a mechanism to describe the relationship of sets and their complements, observing that:

$\overline{A \cup B} = \overline{A} \cap \overline{B}$

$\overline{A \cap B} = \overline{A} \cup \overline{B}$

Venn Diagrams

Now let's examine how to visualize these two sets as a venn diagram using GGPlot.

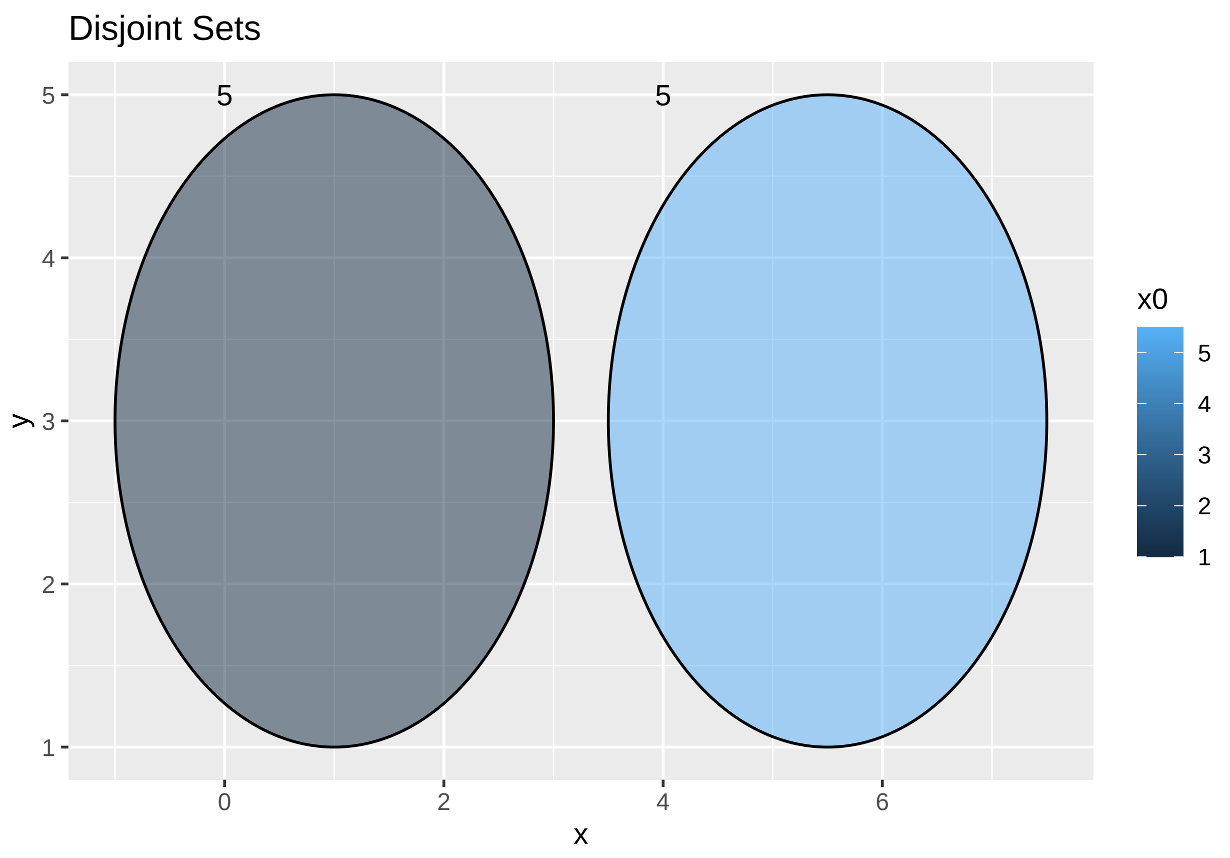
```
venndiagram <- function (A,B){

set <- setoperations(A,B)

x_points <- rep(1:10)
y_points <- rep(1:10)
coords <- data.frame(x=x_points,y=y_points)

x_points <- c(3,6)
y_points <- c(4,5)
circles <- data.frame(x=x_points,y=y_points)
```

```
if(!set$A_disjoint_B){
  # the groups are not disjoint, so they must overlap
  if(set$A_sub_B | set$B_sub_A){
      #if one set is contained in the other

    if(set$A_sub_B){
      circlecolor <- c("A","B")
      circlex <- c(2,2)
      circley <- c(3,3)
      circlesize <- c(1.5,2)
    }else if(set$B_sub_A){
      circlecolor <- c("A","B")
      circlex <- c(2,2)
      circley <- c(3,3)
      circlesize <- c(2,1.5)
    }else{
      circlecolor <- c("A","B")
      circlex <- c(1,3)
      circley <- c(3,3)
      circlesize <- c(2,2)

    }

      circles <- data.frame(
          x0 = circlex,
          y0 = circley,
          r = circlesize,
          fill = circlecolor
        )

      totals <- data.frame(
        x = c(0,4),
        y = c(5,5),
        num = c(length(A),length(B))
      )

      counts <- data.frame(
        x = c(0.25,3.75),
        y = c(3,3),
        num = c(length(A)-length(set$A_intersect_B),length(B)-length(set$A_intersect_B)
      )

      intersect <- data.frame(
        x = c(2),
        y = c(3),
        num = c(length(set$A_intersect_B))
      )

      # plot
      ggplot() +
        geom_circle(aes(x0 = x0, y0 = y0, r = r, fill=fill), alpha=.5, data = circles) +
        geom_text(aes(x = x, y = y, label=num), data = totals)+
        geom_text(aes(x = x, y = y, label=num), data = counts)+
        geom_text(aes(x = x, y = y, label=num), data = intersect)  +
       labs(title="Sets with Subsets")

  }else{
     #neither set is contained in the other


    circles <- data.frame(
          x0 = c(1,3),
          y0 = c(3,3),
          r = c(2,2)
        )

      totals <- data.frame(
        x = c(0,4),
        y = c(5,5),
        num = c(length(A),length(B))
      )

      counts <- data.frame(
        x = c(0,4),
        y = c(3,3),
        num = c(length(A)-length(set$A_intersect_B),length(B)-length(set$A_intersect_B)
      )

      intersect <- data.frame(
        x = c(2),
        y = c(3),
        num = c(length(set$A_intersect_B))
      )

      # plot
      ggplot() +
        geom_circle(aes(x0 = x0, y0 = y0, r = r, fill=x0), alpha=.5, data = circles) +
        geom_text(aes(x = x, y = y, label=num), data = totals) +
        geom_text(aes(x = x, y = y, label=num), data = counts) +
        geom_text(aes(x = x, y = y, label=num), data = intersect) +
        labs(title="Intersecting Sets")

  }


}else{
 # the groups are disjoint, so they do not overlap

  circles <- data.frame(
    x0 = c(1,5.5),
    y0 = c(3,3),
    r = c(2,2)
  )
  totals <- data.frame(
    x = c(0,4),
    y = c(5,5),
    num = c(length(A),length(B))
  )
  counts <- data.frame(
    x = c(0,4),
    y = c(3,3),
    num = c(length(A)-length(set$A_in_B),length(B)-length(set$A_in_B))
  )

ggplot() +
  geom_circle(aes(x0 = x0, y0 = y0, r = r, fill=x0), alpha=.5, data = circles) +
  geom_text(aes(x = x, y = y, label=num), data = totals) +
  #geom_text(aes(x = x, y = y, label=num), data = counts) +
  labs(title="Disjoint Sets")


}


}
```

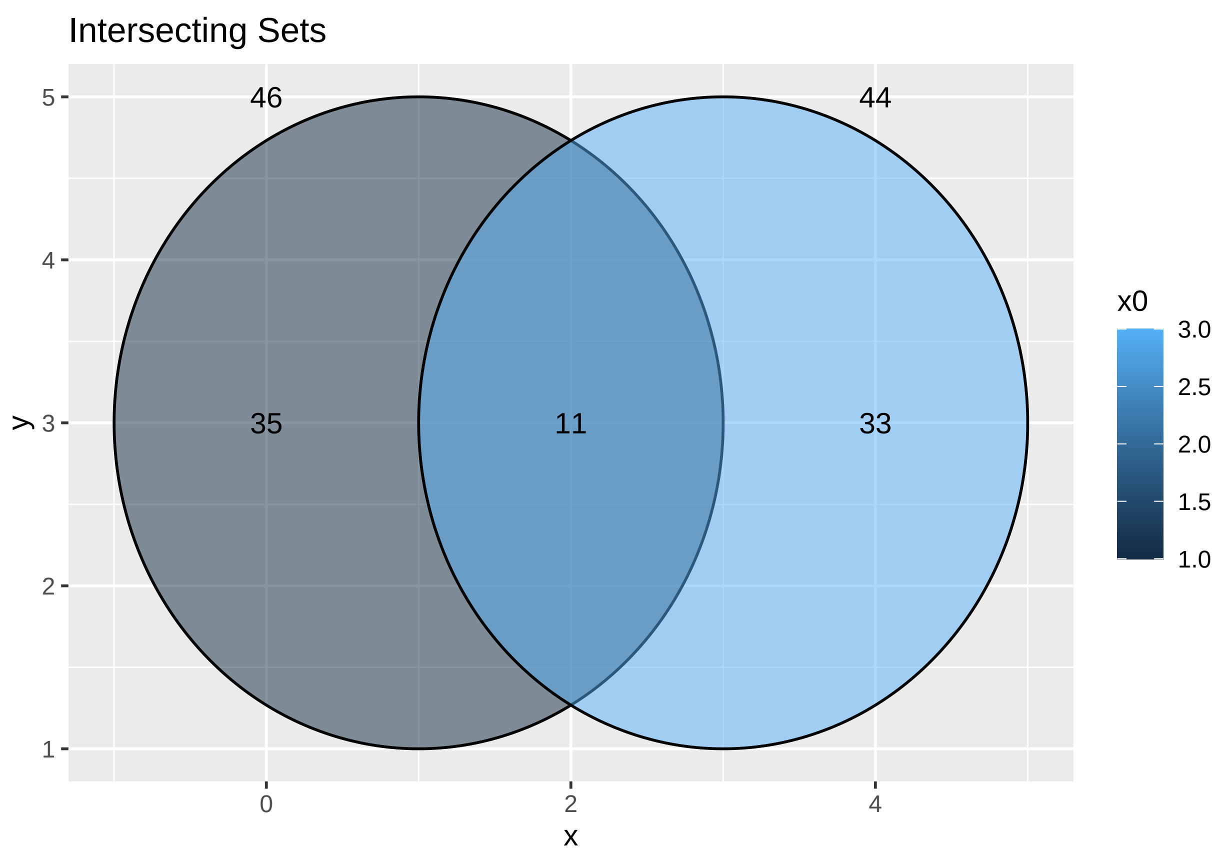
Now we will create some random sets and see if they are intersecting, disjoint, or if one set is a subset of another set.

```
#randomly generate a disjoint set
A <- unique(round(rnorm(n=5,mean=50,sd=10),0))
B <- unique(round(rnorm(n=5,mean=150,sd=10),0))
```

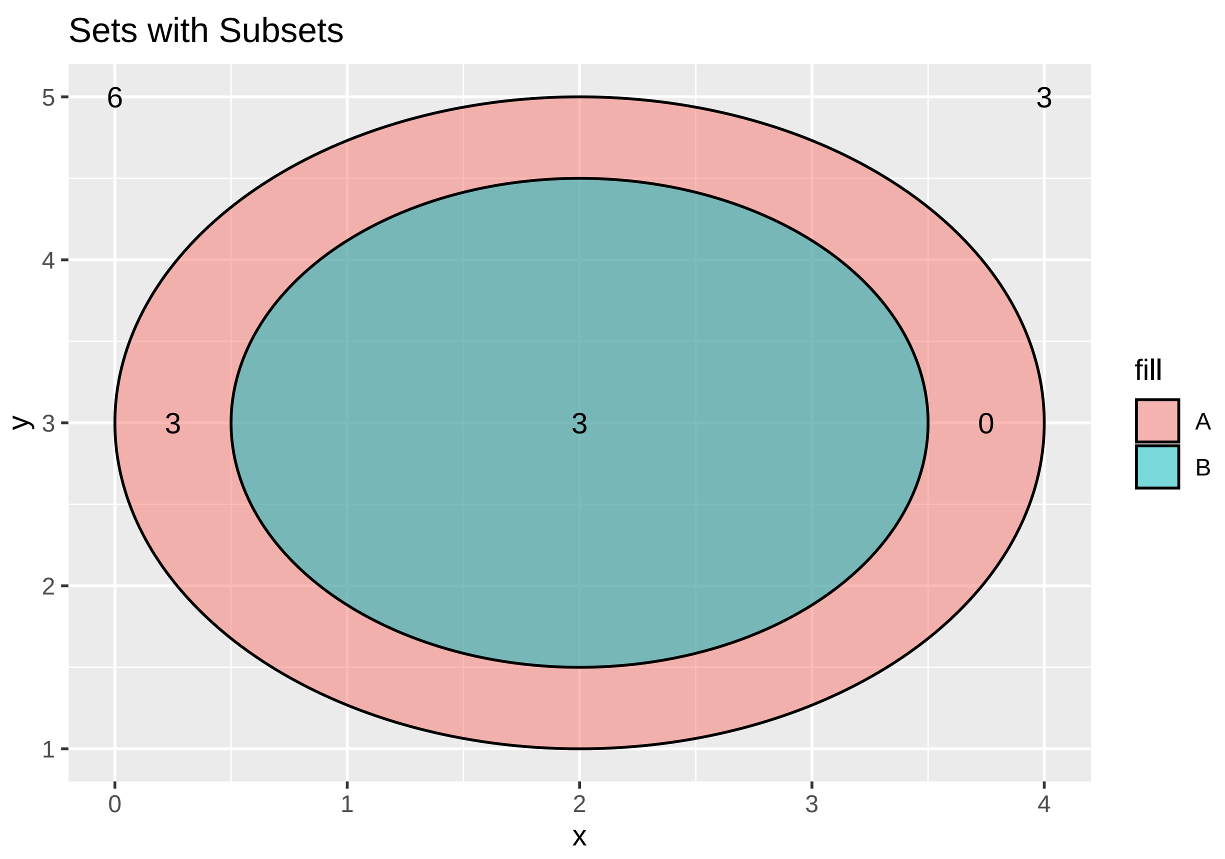
```
venndiagram(A,B)
```



```
#randomly generate a (usually) intersecting set
A <- unique(round(rnorm(n=50,mean=100,sd=50),0))
B <- unique(round(rnorm(n=50,mean=100,sd=50),0))
venndiagram(A,B)
```



```
#create a set with a subset
A <- c(1,2,3,4,5,6)
B <- c(3,4,5)
venndiagram(A,B)
```


Sets with Subsets
6
3
3
3
0
y
x
5
4
3
2
1
0
1
2
3
4
fill
A
B

Multiplication Principle of Counting

When completing a sequence of choices, there are p choices for the first, q for the second, r for the third, and so on, then the multiplication principle of counting may be used to determine how many different ways D that this sequence can be completed.

This formula holds that: $D = p \cdot q \cdot r...$

This formula is useful for determining things like the number of paths through a maze, the number of item combinations on a menu, or the number of possible members of a committee or focus group, when selecting from multiple pools of candidates.

```r
#set a series of choices that can be combined
choices <- c(4,6,3)

#the total will multiple each choices following the first, byu the sum of all prior pr
total <- choices[1]
for (x in choices[-1]){
  total <- total * x
}

# result
result <- paste("A menu with ", choices[1]," appetizer options,  ", choices[2]," entrees
#result
```

> A menu with 4 appetizer options, 6 entrees, and 3 drinks provides a total of 72 distinct meal choices.

Expected Value

To find the expected value, you must define a sample space, determine the probability of each outcome, and assign a payoff (or cost) to each outcome.

$E = (p_1 \cdot m_1) + (p_2 \cdot m_2) + ... + (p_n \cdot m_n)$

Considering the roll of a dice - if you have to pay to roll, and have 1 chance to win, the probability of each outcome is 1 divided by the number of sides on the dice. The list of payouts is 1 times the winning amount, and sides - 1 times cost to play.

If the game result in an expected value of 0 - it is considered a “fair” game. If you played the game enough times, your net earnings would be 0. If the expected value is greater than 0, the game favors the player, but if the value is less than 0, the game favors the house. A game that favors the player will eventually lead to more money won than lost. This is why most casino games are not “fair games” - otherwise the casino would lose money over time.

```r
#expected value function
expected_value <- function (p,m){
 E <- sum(p * m)
 E
}

sides <- 8
side_chance <- 1/sides
payoff <- 3
cost <- -.50

#likelihood of each outcome in teh sample space
p <- rep(side_chance,sides)

#payoff for each outcome
m <- c(payoff,rep(cost,sides-1))

E <- expected_value(p,m)

# result
result <- paste("You are playing a dice game. To win $",payoff," you must roll a 1 on a
```

> You are playing a dice game. To win $3 you must roll a 1 on a 8 sided dice. If it costs $-0.5 to play, the expected value is -0.06

Combinations & Permutations

To determine the number of possible ordered arrangements of groups of elements, We must use the factorial operator. A factorial is defined as the product of an integer and all the integers below it.

A factoral can be expressed mathematically as:

$n! = n \cdot (n-1) \cdot (n-2) \cdot (n-3) \cdot \ldots \cdot 3 \cdot 2 \cdot 1$

And the value of 0! is 1 per the convention for an Empty Product.

Number of Ordered arrangements of r objects chosen from n objects when the n objects are distinct, and repetition is allowed

n^r

```
#create combination function
ordered_arrangements <- function (n,r){
  p0 <-  n^r
  p0
}

#number of possible choices
n <- 9
#number of choices used
r <- 7

#call the function
p0 <-  ordered_arrangements(n,r)

# result
result <- paste("The number of possible ",r," digit codes using ",n," possible digits is
#result
```

The number of possible 7 digit codes using 9 possible digits is4782969

Number of Permutations of r Objects selected from n Distinct possibilities with no Repetition

$P(n,r) = \frac{n!}{(n-r)!}$

```r
#create permutation function
permutations_nonrepeated <- function (n,r){
  p0_top <-  factorial(n)
  p0_bottom <-  factorial(n-r)*factorial(r)

  p0 <- p0_top / p0_bottom
}

n <- 26
r <- 4

p0 <- permutations_nonrepeated(n,r)


# result
result <- paste("The number of possible ",r," letter words choosing from ",n," possible
#result
```

The number of possible 4 letter words choosing from 26 possible letters is 14950 if no letters can be repeated.

Number of Permutations of n distinct objects using all n of them:

$P(n, n) = n!$

```r
#create permutation function
permutations_using_all <- function (n){
  p0 <-  factorial(n)
  p0
}

#number of items
n <- 6

#call the function
p0 <-  permutations_using_all(n)

# result
result <- paste("The number of arrangements of a ",n," letter word where all the letters
#result
```

The number of arrangements of a 6 letter word where all the letters are different, using all 6 letters is 720.

Number of Combinations of n Distinct Objects taken r at a Time

$C(n,r) = \frac{n!}{(n-r)! \cdot r!}$

```r
#create permutation function
combinations_using_some <- function (n,r){
  p0_top <-  factorial(n)
  p0_bottom <-  factorial(n-r)*factorial(r)
  p0 <- p0_top / p0_bottom
  p0
}

#number of items
n <- 8

#number of items used
r <- 3

# result
result <- paste("You are betting on a horse race and you must choose ",r," horses to win
#result
```

> You are betting on a horse race and you must choose 3 horses to win. You must choose accurately which horse will finish in 1st (win), 2nd (place), and 3rd (show), in the correct order. The number of ways that 8 horses could finish in these 3 positions is:720.

This concludes this section of the R Markdown Math companion. In a future installment, "A Probability Companion in R Markdown", we will explore subjects of probability, more complex applications of expected value, and subjects of modeling and inference.