Exploring Science, Engineering, Technology, and Mathematics (STEM) With MathRider

by Ted Kosan

Table of Contents

1 Preface

1.1 Dedication

This book is dedicated to Steve Yegge and his blog entry "Math Every Day" (http://steve.yegge.googlepages.com/math-every-day).

1.2 Acknowledgments

1.3 Support Email List

The support email list for this book is called **mathrider-users@googlegroups.com** and you can subscribe to it at http://groups.google.com/group/mathrider-users.

1.4 Recommended Weekly Sequence When Teaching A Class With This Book

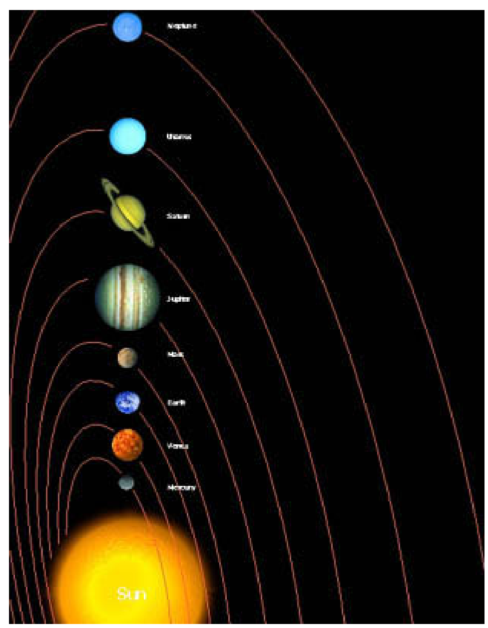
- Week 6: Sections 1-3.
- Week 7: Section 4
- Week 8: Sections 5-6
- Week 9: Section 7
- Week 10: Sections

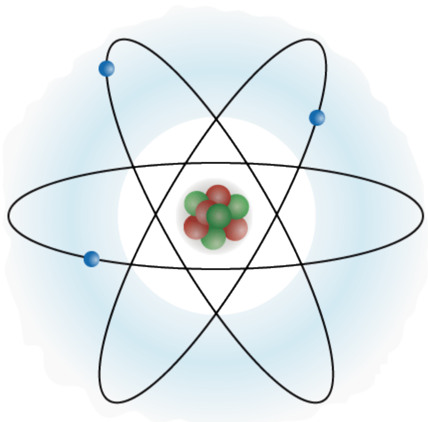
2 Introduction

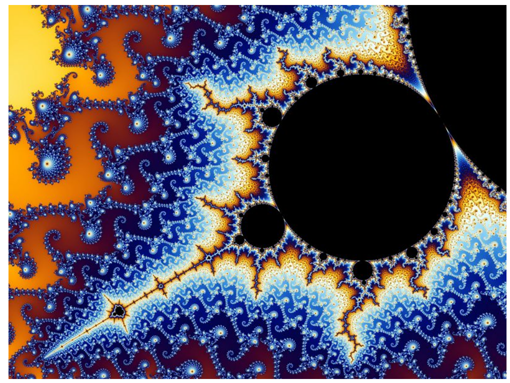
In the book *Introduction To Programming With MathRider And MathPiper*, the basic parts of MathRider were covered along with the fundamentals of MathPiper programming. This book shows how to use these skills to explore the exciting areas of Science, Engineering, Technology, and Mathematics which are collectively referred to as the STEM disciplines. It uses an intuitive approach to describing these areas which is based on the concept of patterns. While the ideas that are discussed are not presented with the mathematical rigor that they typically are, it is hoped that this intuitive approach will provide an additional path to understanding which complements the rigorous approach.

2.1 Patterns And STEM

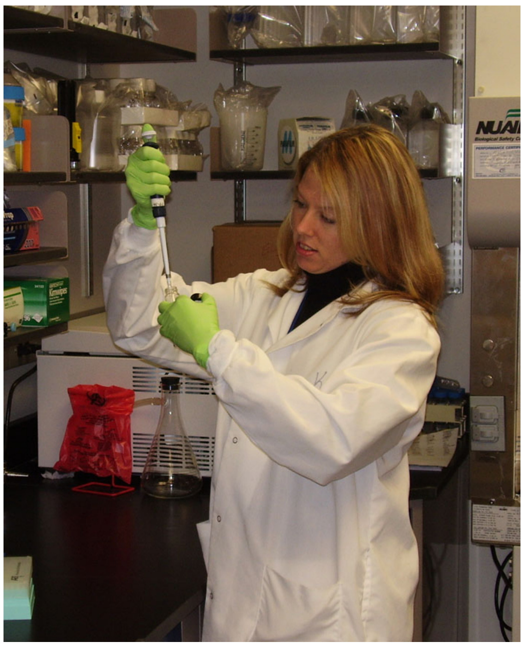
2.1.1 Patterns: The Fundamental "Stuff" Out Of Which Everything In The Universe Is Made



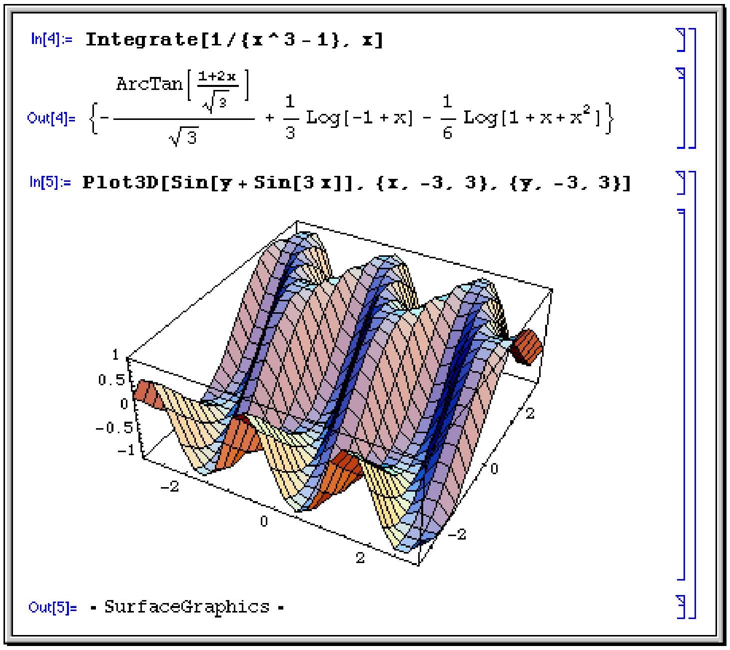




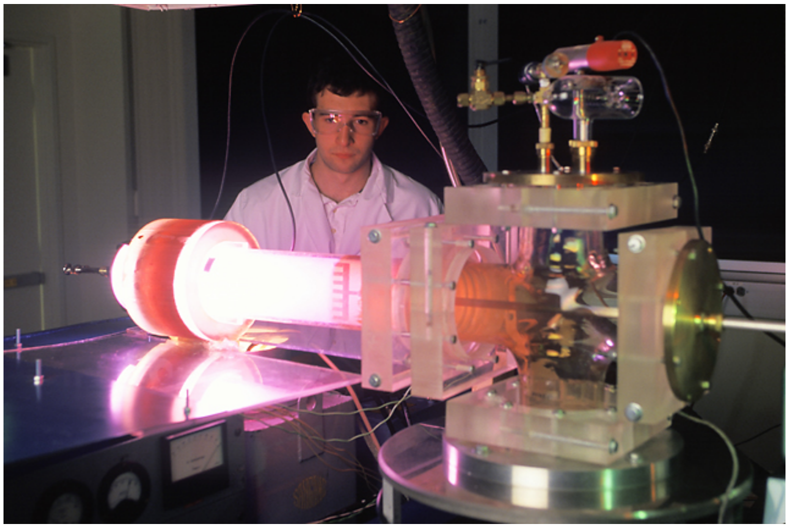
30 2.1.2 Science: A System For Discovering And Organizing Patterns



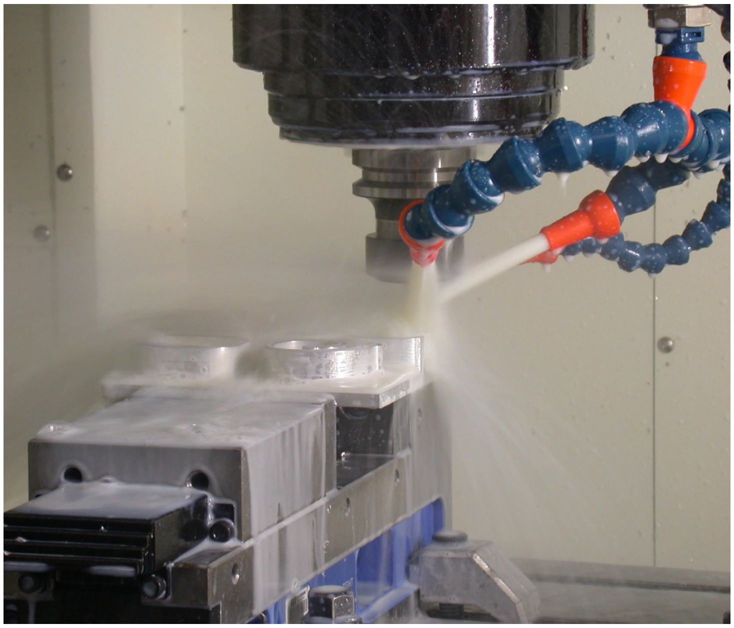
31 2.1.3 Mathematics: The Language Of Patterns



32 ### 2.1.4 Engineering: Designing Things With Patterns



33 ### 2.1.5 Technology: Making Things With Patterns



2.2 Tools For Analyzing And Manipulating Patterns

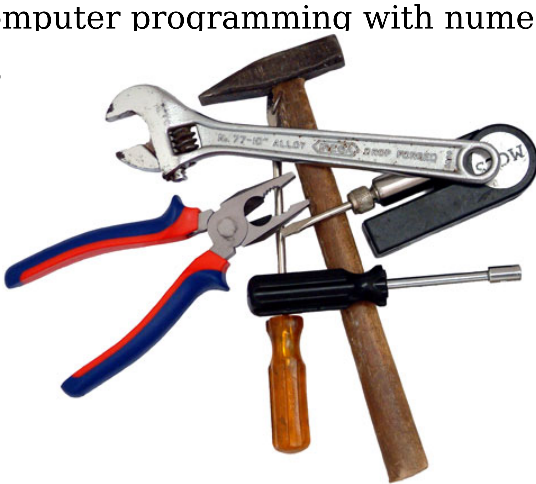
Tool levels:

Pencil marks on paper.

Hand held calculator.

Computer programming with numerical mathematics.

Co lic mathematics.



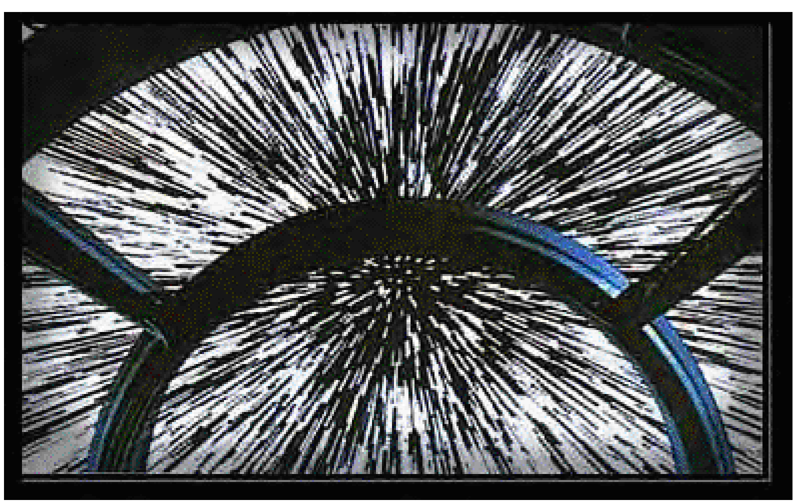
2.2.1 Pencil Marks On Paper = Walking

2.2.2 Hand Held Calculator = Automobile

2.2.3 Computer Programming With Loops = Spacecraft

```
%mathpiper

//Print the odd integers from 1 to 99.

x := 1;

While(x <= 100)
[
    Write(x);
    x := x + 2;  //Increment x by 2.
];

%/mathpiper

    %output,preserve="false"
      Result: True

      Side effects:
      1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43
      45 47 49 51 53 55 57 59 61 63 65 67 69 71 73 75 77 79 81 83
      85 87 89 91 93 95 97 99
  %/output
```

2.2.4 Computer Programming With Formulas/Theorems = Spacecraft With Hyperdrive



2.3 All STEM Areas Are Quickly Becoming Software Based

Physics	Computational Physics.
Biology	Computational Biology.
Mathematics	Computational Mathematics.
Chemistry	Computational Chemistry.
Geophysics	Computational Geophysics.
Forensics	Computational Forensics.

3 Patterns And Pattern Spaces

3.1 Exploring A Permutation Pattern Space

A pattern is anything which has aspects which repeat in a predictable manner and here are some examples of them:

- All plants and animals are made of patterns of cells.
- Cells and minerals are built with patterns of molecules and atoms.
- Atoms are patterns of protons, neutrons, and electrons.
- Music consists of sound patterns.
- Weather moves in weather patterns.
- Object that are dropped fall according to a specific pattern.
- Objects that are thrown follow a specific pattern.
- Words are composed of letter patterns.

3.1.1 Generating Permutation Patterns With PermutationsList()

A good place to begin our study of patterns is with **permutation patterns**. Imagine that a certain family has three children named Bill, Mary, and Tom and that they are going to the store in the family car. Bill, Mary, and Tom will be sitting in the back seat. How many different ways can they be seated? This can be shown using the **PermutationsList()** function, which takes a list as an argument and returns another list that has all of the patterns that can be generated from it:

```
In> PermutationsList({Bill,Mary,Tom})
Result: {{Bill,Mary,Tom},{Bill,Tom,Mary},{Tom,Bill,Mary},{Mary,Bill,Tom},
{Mary,Tom,Bill},{Tom,Mary,Bill}}
```

All of the different ways in which Bill, Mary, and Tom can be seated is an example of a **permutation pattern** and with this type of pattern, the **order** of its elements is taken into account. In mathematics, this kind of permutation is categorized under **combinatorics** and it is defined to be:

> "Possible arrangements of a set of objects in which the order of the arrangement makes a difference. Example: Determine all the different ways five books can be arranged in order on a shelf."
> schools.look4.net.nz/maths/maths_glossary/p

Instead of using permutations of names, lets use permutations that can be formed by **letters** of the English alphabet because they take up less space when printed. We will start with **2** letters and work up from there. How many different patterns can be formed by the letters **a** and **b**?:

```
In> PermutationsList({a,b})
Result: {{a,b},{b,a}}
```

It looks like two patterns. What about 3 letters?:

```
In> PermutationsList({a,b,c})
Result: {{a,b,c},{a,c,b},{c,a,b},{b,a,c},{b,c,a},{c,b,a}}
```

The total number of patterns is 6. The following two examples show the number of permutation patterns that can be generated with 4 and 5 letters:

```
In> PermutationsList({a,b,c,d})
Result: {{a,b,c,d},{a,b,d,c},{a,d,b,c},{d,a,b,c},{a,c,b,d},{a,c,d,b},
{a,d,c,b},{d,a,c,b},{c,a,b,d},{c,a,d,b},{c,d,a,b},{d,c,a,b},{b,a,c,d},
{b,a,d,c},{b,d,a,c},{d,b,a,c},{b,c,a,d},{b,c,d,a},{b,d,c,a},{d,b,c,a},
{c,b,a,d},{c,b,d,a},{c,d,b,a},{d,c,b,a}}
```

```
In> PermutationsList({a,b,c,d,e})
Result: {{a,b,c,d,e},{a,b,c,e,d},{a,b,e,c,d},{a,e,b,c,d},{e,a,b,c,d},
{a,b,d,c,e},{a,b,d,e,c},{a,b,e,d,c},{a,e,b,d,c},{e,a,b,d,c},{a,d,b,c,e},
{a,d,b,e,c},{a,d,e,b,c},{a,e,d,b,c},{e,a,d,b,c},{d,a,b,c,e},{d,a,b,e,c},
{d,a,e,b,c},{d,e,a,b,c},{e,d,a,b,c},{a,c,b,d,e},{a,c,b,e,d},{a,c,e,b,d},
{a,e,c,b,d},{e,a,c,b,d},{a,c,d,b,e},{a,c,d,e,b},{a,c,e,d,b},{a,e,c,d,b},
{e,a,c,d,b},{a,d,c,b,e},{a,d,c,e,b},{a,d,e,c,b},{a,e,d,c,b},{e,a,d,c,b},
{d,a,c,b,e},{d,a,c,e,b},{d,a,e,c,b},{d,e,a,c,b},{e,d,a,c,b},{c,a,b,d,e},
{c,a,b,e,d},{c,a,e,b,d},{c,e,a,b,d},{e,c,a,b,d},{c,a,d,b,e},{c,a,d,e,b},
{c,a,e,d,b},{c,e,a,d,b},{e,c,a,d,b},{c,d,a,b,e},{c,d,a,e,b},{c,d,e,a,b},
{c,e,d,a,b},{e,c,d,a,b},{d,c,a,b,e},{d,c,a,e,b},{d,c,e,a,b},{d,e,c,a,b},
{e,d,c,a,b},{b,a,c,d,e},{b,a,c,e,d},{b,a,e,c,d},{b,e,a,c,d},{e,b,a,c,d},
{b,a,d,c,e},{b,a,d,e,c},{b,a,e,d,c},{b,e,a,d,c},{e,b,a,d,c},{b,d,a,c,e},
{b,d,a,e,c},{b,d,e,a,c},{b,e,d,a,c},{e,b,d,a,c},{d,b,a,c,e},{d,b,a,e,c},
{d,b,e,a,c},{d,e,b,a,c},{e,d,b,a,c},{b,c,a,d,e},{b,c,a,e,d},{b,c,e,a,d},
{b,e,c,a,d},{e,b,c,a,d},{b,c,d,a,e},{b,c,d,e,a},{b,c,e,d,a},{b,e,c,d,a},
{e,b,c,d,a},{b,d,c,a,e},{b,d,c,e,a},{b,d,e,c,a},{b,e,d,c,a},{e,b,d,c,a},
{d,b,c,a,e},{d,b,c,e,a},{d,b,e,c,a},{d,e,b,c,a},{e,d,b,c,a},{c,b,a,d,e},
{c,b,a,e,d},{c,b,e,a,d},{c,e,b,a,d},{e,c,b,a,d},{c,b,d,a,e},{c,b,d,e,a},
{c,b,e,d,a},{c,e,b,d,a},{e,c,b,d,a},{c,d,b,a,e},{c,d,b,e,a},{c,d,e,b,a},
{c,e,d,b,a},{e,c,d,b,a},{d,c,b,a,e},{d,c,b,e,a},{d,c,e,b,a},{d,e,c,b,a},
{e,d,c,b,a}}
```

As you can see, for each letter that is added to the input list, the number of patterns that are generated increases significantly. Use MathRider right now to experiment with generating permutation patterns.

(Also, the **TableForm()** function can be used to view the contents of a list vertically if desired):

```
In> TableForm(PermutationsList({a,b,c}))
Result: True
Side Effects:
{a,b,c}
{a,c,b}
{c,a,b}
{b,a,c}
{b,c,a}
{c,b,a}
```

As you experimented with generating permutation patterns of letters in MathPiper, did you see how a small number of symbols (letters) which are arranged using a given set of rules can generate numerous thousands of patterns? In this book, a set of **symbols** along with **rules** for manipulating them is called a **pattern system** and the patterns that are generated by the system are called its **pattern space**.

If we wanted to determine the number of permutation patterns (or **pattern space size**) that could be generated by 2 letters, 5 letters, or even 8 letters, we could do this by having PermutationsList() generate the patterns for us and then we could count them using a loop. Here is a program which does this:

```
%mathpiper,title=""

patterns := PermutationsList({a,b,c,d,e,f,g,h});

numberOfPatterns := 0;

ForEach(pattern, patterns)
[
    numberOfPatterns++;
];

NewLine(2);

Echo("Number of patterns: ", numberOfPatterns);

%/mathpiper

    %output,preserve="false"
      Result: True

      Side Effects:
      Number of patterns: 40320
.   %/output
```

Note, we could also count the number of patterns with less code by letting the

Length() function do the looping for us:

```
%mathpiper,title=""

patterns := PermutationsList({a,b,c,d,e,f,g,h});

Length(patterns);

%/mathpiper

    %output,preserve="false"
      Result: 40320
.   %/output
```

However, I am purposely using While loops in these examples to emphasize the fact that **a loop is being used to count the patterns** .

How long did it take the While loop based program to arrive at its answer? Here is a version of the program with **timing code** added to it which answers this question:

```
%mathpiper,title=""

patternGenerationTime := Time() patterns := PermutationsList({a,b,c,d,e,f,g,h});

numberOfPatterns := 0;

countingTime := Time() ForEach(pattern, patterns)
[
    numberOfPatterns++;
];

NewLine(2);

Echo("Number of patterns: ", numberOfPatterns);

Echo("Pattern generation time: ", patternGenerationTime, "seconds.");

Echo("Counting time: ", countingTime, "seconds.");

Echo("Total Time: ", patternGenerationTime + countingTime, "seconds.");

%/mathpiper

    %output,preserve="false"
      Result: True

      Side Effects:
      Number of patterns: 40320
      Pattern generation time: 2.004516503 seconds.
      Counting time: 0.531656747 seconds.
      Total Time: 2.536173250 seconds.
.   %/output
```

Unfortunately, you will find that if you increase the number of letters to permute to **9 or higher**, the **pattern space** that is generated **becomes too large for the computer to handle**. But what if we wanted to know how many patterns could be generated by 15, 20, or more letters? Is there a way to determine this? The answer is yes, by using the **Permutations()** function.

3.1.2 The Permutations() Function And The Enormous Pattern Spaces It Can Work With

The **Permutations()** function is able to determine the number of permutation patterns that can be generated with a given number of symbols. There are two versions of the Permutations() function but we are only use the following one for now:

```
Pemutations(number_of_symbols)
```

The **number_of_symbols** parameter is an integer and it is used to indicate the total number of symbols that are to be worked with. Here are some examples which calculate the number of permutation patterns that can be generated with 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20, and 40 symbols (along with the times needed to make the calculations):

```
In> EchoTime() Permutations(1)
Result: 1
Side Effects:
0.000727397 seconds taken.

In> EchoTime() Permutations(2)
Result: 2
Side Effects:
0.000958431 seconds taken.

In> EchoTime() Permutations(3)
Result: 6
Side Effects:
0.000793467 seconds taken.

In> EchoTime() Permutations(4)
Result: 24
Side Effects:
0.000828178 seconds taken.

In> EchoTime() Permutations(5)
Result: 120
Side Effects:
0.000806806 seconds taken.
```

```
In> EchoTime() Permutations(6)
Result: 720
Side Effects:
0.000925119 seconds taken.

In> EchoTime() Permutations(7)
Result: 5040
Side Effects:
0.000907727 seconds taken.

In> EchoTime() Permutations(8)
Result: 40320
Side Effects:
0.000967442 seconds taken.

In> EchoTime() Permutations(9)
Result: 362880
Side Effects:
0.000965765 seconds taken.

In> EchoTime() Permutations(10)
Result: 3628800
Side Effects:
0.000993493 seconds taken.

In> EchoTime() Permutations(15)
Result: 1307674368000
Side Effects:
0.001352337 seconds taken.

In> EchoTime() Permutations(20)
Result: 2432902008176640000
Side Effects:
0.001340743 seconds taken.

In> EchoTime() Permutations(40)
Result: 815915283247897734345611269596115894272000000000
Side Effects:
0.001955836 seconds taken.
```

These examples contain two shocking surprises! The first surprise is how large the pattern spaces becomes for a relatively small number of symbols. Who would have thought that 40 symbols can generate 815915283247897734345611269596115894272000000000 different patterns!?

The second surprise is how quickly the size of even the largest pattern space in the example can be calculated. Even the enormous permutation pattern space for 40 symbols only took about $\frac{2}{1000}$ of a second (or 2 milliseconds) to calculate!

If it took our looping program around 2.5 seconds to determine the pattern space that 8 symbols generate, how can the **Permutations()** function calculate the size of the 40 symbol pattern space in around 2 milliseconds?!

There is definitely something almost magical going on here because somehow the Permutations() function is able to **move through a pattern space** way *way* faster than our looping program can. The question is, how is it able to do this? The next section explains this by discussing the different tools that can be used to move through a pattern space.

3.2 Tools For Moving Through A Pattern Space

Pattern spaces are present everywhere in the universe, from the microscopic level of atoms, molecules, and cells up through the macroscopic level of solar systems, galaxies, and super galaxies. The astonishing thing is that **most of these diverse patterns are similar to each other** in ways which enable humans to work with them using a **single set of tools**. If you know how to work with patterns using these tools, the universe (and everything in it) is yours to explore...

A common thing that needs to be done with a pattern space is to **move through it**. For example, here is a familiar patten space which consists of the numbers from 1 to 100:

```
In> integersList := 1 .. 100
Result:
{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28
,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,53
,54,55,56,57,58,59,60,61,62,63,64,65,66,67,68,69,70,71,72,73,74,75,76,77,78
,79,80,81,82,83,84,85,86,87,88,89,90,91,92,93,94,95,96,97,98,99,100}
```

In this section, we will use this pattern as an example because in some ways it is simpler than a permutation pattern.

3.2.1 Walking Through A Pattern Space With Pencil And Paper

An operation we may want to perform on these numbers is to find their **sum**. One way to do this is to write the numbers on a piece of paper and then add them together using a pencil. Here is the list shown in a form which is similar to what it might look like on a piece of paper:

```
In> PrintList(integersList," + ");
Result: "1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 + 11 + 12 + 13 + 14 + 15 +
16 + 17 + 18 + 19 + 20 + 21 + 22 + 23 + 24 + 25 + 26 + 27 + 28 + 29 + 30 +
31 + 32 + 33 + 34 + 35 + 36 + 37 + 38 + 39 + 40 + 41 + 42 + 43 + 44 + 45 +
46 + 47 + 48 + 49 + 50 + 51 + 52 + 53 + 54 + 55 + 56 + 57 + 58 + 59 + 60 +
61 + 62 + 63 + 64 + 65 + 66 + 67 + 68 + 69 + 70 + 71 + 72 + 73 + 74 + 75 +
76 + 77 + 78 + 79 + 80 + 81 + 82 + 83 + 84 + 85 + 86 + 87 + 88 + 89 + 90 +
91 + 92 + 93 + 94 + 95 + 96 + 97 + 98 + 99 + 100"
```

A typical person doing the adding would need to **slowly** move through this pattern space, visiting each number in turn in order to **add** it to an accumulating **sum**. The complete process will take perhaps **10 minutes** and when a human **manually** moves through a pattern space like this, it can be thought of as **walking** through it.

3.2.2 Flying A Spacecraft Through A Pattern Space With A Loop Based Program

A computer program is able to find the sum of these numbers significantly faster than a human can because the program is able to move through a pattern space *much* quicker than a human. The following program moves through this pattern space very quickly using a While loop, visiting each number in turn and adding it to an accumulating sum:

```
%mathpiper,title=""

sum := 0;

index := 1;

runTime := Time() While(index <= 100)
[
    sum := sum + index;

    index++;
];

Echo("Sum: ", sum);

Echo("Run time: ", runTime);

%/mathpiper

    %output,preserve="false"
      Result: True

      Side Effects:
      Sum: 5050
      Run time: 0.006013
.   %/output
```

The program took about $\frac{6}{1000}$ of a second (or 6 milliseconds) to find the sum of the numbers between 1 and 100 inclusive. How much faster is this than a typical human can do it by hand? Lets calculate it:

```
In> N((10 /*minutes*/ * 60 /*seconds per minute*/ )/runTime /*seconds*/);
```

```
Result: 99833.61065
```

About 100,000 times faster!

This program shows that one of the reasons that **computers** are so **powerful** is that they can **move through pattern spaces at very high rates of speed**. Moving through a pattern space with a computer can be thought of as **flying a spacecraft** through it and a significant part of this book is devoted to showing how a computer's speed can be used to do amazing things with patterns. However, before you become too taken with this enormous amount of power, you should know how easily a computer can be brought to it knees.

3.2.2.1 A Pattern Space Can Quickly Become Too Vast For Even A Computer "Spacecraft" To Traverse

Calculating the sum of the integers between 1 and 100 with a While loop did not take that long, but how long would it take if the highest number in the sum was 1000, 10000, 100000, etc.? Lets find out.

The following code consists of two parts which have been placed into separate folds. The **first** part is the **declaration of a function** called **sumNums()** which calculates the sum of the integers between 1 and a passed in integer. The **second** part is a program which passes increasingly larger integers to **sumNums()** and then prints the sums and how long it took to calculate them.

In order to run this code in MathRider, you must **first execute the sumNums() definition fold (but only once)** to create the sumNums() function. Then, you can execute the second fold as many times as you need to in order to experiment with it.

```
%mathpiper,title=""

sumOfIntegers(highestInteger) :=
[
    Local(sum, index);

    sum := 0;

    index := 1;

    While(index <= highestInteger)
    [
        sum := sum + index;

        index++;
    ];

    sum;
];

%/mathpiper
```

```
%mathpiper,title=""

ForEach(highNumber, {100,1000,10000,100000,1000000})
[
    runTime := Time() sum := sumOfIntegers(highNumber);

    Echo("High number: ", highNumber);

    Echo("Sum: ", sum);

    Echo("Run time: ", runTime, "seconds.");

    NewLine();
];

%/mathpiper

    %output,preserve="false"
      Result: True

      Side Effects:
      High number: 100
      Sum: 5050
      Run time: 0.007535663 seconds.

      High number: 1000
      Sum: 500500
      Run time: 0.07409049 seconds.

      High number: 10000
      Sum: 50005000
      Run time: 0.412270142 seconds.

      High number: 100000
      Sum: 5000050000
      Run time: 3.609522366 seconds.

      High number: 1000000
      Sum: 500000500000
      Run time: 35.263227167 seconds.
.   %/output
```

As you can see, the number that is being summed to is made **10 times larger**, the time it takes to calculate the sum becomes about **10 times longer**. Just like with the loop-based permutation counting program, it won't take too many additional digits before the generated pattern space is too large for this loop-based program to move through. Even though both of these loop-based programs move through their respective pattern spaces with a speed that is analogous to a spacecraft's, they also both become **overwhelmed when the pattern spaces become too large**.

At this point you may be wondering if there is a "magical" method for quickly calculating the sums of enormous sequences of integers that is analogous to the one that the Permutations() function used to quickly calculate the size of enormous permutation pattern spaces.

The answer is yes! And as Darth Vader might say:

> **Don't be too proud of this computational terror you've constructed. The ability to speed through a pattern space is insignificant next to the power of... mathematics!**

3.2.3 Hyperspace Jumping Through A Pattern Space With Mathematical Formulas And Theorems

At this point you should have a fairly good grasp of how fundamental computer programming works and how a loop-based program can quickly move through a pattern space in order to perform a calculation. The process mostly consists of doing some simple operations over and over again at a high rate of speed.

The techniques we are going to discuss next work very differently than this. Instead of moving through a pattern space in a **linear** fashion, they **jump** from one part of a pattern space to another **almost instantly**. The hard part is locating a jump point which will take you where you want to go. After you have located a jump point, you just make the jump and you are there!

3.2.3.1 Beginning The Search For An Integer Summing Jump Point

George Pólya was a mathematician who developed a set of principles for solving problems. One of the ways he devised for solving a difficult problem is to first solve a **simpler** problem which is similar to it and then use what you learned to solve the more **difficult** problem.

In our case we want to locate a jump point which will allow us to navigate to the location in the integer pattern space which contains the integer which is the sum of a given sequence of integers. This is a difficult problem to solve generally for all possible sequences of integers so we will begin by solving the **simpler** problem of finding the sum of the integers **1-10**.

It is said that the mathematician Karl Friedrich Gauss discovered this jump point while he was in elementary school. Evidently his math teacher use to occupy the class with calculating the sums of sequences of integers so he could do other things. Gauss was able to use this jump point to calculate the sums within seconds to the great astonishment of his teacher (http://en.wikipedia.org/wiki/Carl_Friedrich_Gauss). How did Gauss discover this jump point? By carefully studying patterns that are present in sequences of integers.

Lets start looking for the pattern he found by making a list of the numbers 1-10 in **forward** order, a second list of the numbers 1-10 in **reverse** order, and then placing them on top of each other like this:

1	2	3	4	5	6	7	8	9	10
10	9	8	7	6	5	4	3	2	1

Study these two rows of numbers for a while. What you are looking for are (surprise!) patterns that are contained in the numbers. Do you see any?

One way you can look at these numbers is as a set of columns and this can be more easily seen by drawing a rectangle around each column:

1	2	3	4	5	6	7	8	9	10
10	9	8	7	6	5	4	3	2	1

There are a number of operations that can be done to the numbers in these columns but perhaps the easiest one is to simply add them together:

	1	2	3	4	5	6	7	8	9	10
+	10	9	8	7	6	5	4	3	2	1
	11	11	11	11	11	11	11	11	11	11

After you have added all of the numbers in the columns together you can see that something astonishing has happened. All of the columns add to 11! Do you think this pattern will work for sequences of integers other than 1-10? Lets use MathPiper to find out. First, remember that 1 .. 10 produces a list of integers from 1 to 10 and 10 .. 1 produces a list of integers from 10 to 1 (also remember, spaces need to be placed on either side of the **..** operator:

```
In> 1 .. 10
Result: {1,2,3,4,5,6,7,8,9,10}
```

```
In> 10 .. 1
Result: {10,9,8,7,6,5,4,3,2,1}
```

Also, if two lists are added together, their individual elements are added and the sums are returned as a list:

```
In> {1,2,3} + {1,1,1}
Result: {2,3,4}
```

Now, we will start with 1-10 to make sure we did the above calculation correctly:

```
In> (1 .. 10) + (10 .. 1)
Result: {11,11,11,11,11,11,11,11,11,11}
```

Yes, we did the calculation correctly. Now, lets try some other sequences of numbers:

```
In> (1 .. 2) + (2 .. 1)
Result: {3,3}

In> (1 .. 3) + (3 .. 1)
Result: {4,4,4}

In> (1 .. 4) + (4 .. 1)
Result: {5,5,5,5}

In> (1 .. 5) + (5 .. 1)
Result: {6,6,6,6,6}

In> (1 .. 6) + (6 .. 1)
Result: {7,7,7,7,7,7}

In> (1 .. 7) + (7 .. 1)
Result: {8,8,8,8,8,8,8}

In> (1 .. 8) + (8 .. 1)
Result: {9,9,9,9,9,9,9,9}

In> (1 .. 9) + (9 .. 1)
Result: {10,10,10,10,10,10,10,10,10}

In> (1 .. 15) + (15 .. 1)
Result: {16,16,16,16,16,16,16,16,16,16,16,16,16,16,16}

In> (1 .. 20) + (20 .. 1)
Result: {21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21}
```

It works with other sequences of integers too! But where do we go from here? If you study these examples you should become aware of yet another pattern. Whatever the **highest** number is in a given sequence, the number that each column adds to is **one higher than it**. For example, if the highest number is 2, the sum of each column is 2+1 = 3. If the highest number is 7, the sum of each column is 7+1=8, and so on.

3.2.3.2 Switching To Mathematical Notation

However, instead of using the phrase "and so on" to indicate that this pattern will work for any sequence of integers from 1 to a given number, a better way is to express this idea is with **mathematical notation**.

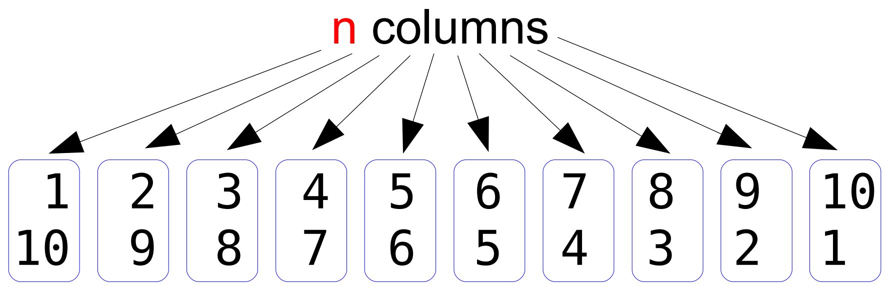
What is **mathematical notation**? Elementary algebra is written using a subset of mathematical notation so if you have done any elementary algebra, you have already used some mathematical notation. Elementary algebra uses numerals like 1,2,3 to represent numbers, letter symbols like a,b,c to represent variables, and various other symbols like +,-,*,/ to represent different operations. More advanced mathematical notation uses interesting looking symbols like Σ and Π (in addition to the elementary algebra operation symbols) to represent operations. In this book, we will mostly be using mathematical notation at the elementary algebra level and then use more advanced mathematical notation only periodically (to delight you!).

Something you may not know about elementary algebra is that, in a way, it is also a programming language. However, it is a special type of "programming" language because it was specifically designed to work with patterns. If you are finding that you are starting to like the patterns stuff we have been doing, you are going to discover that you will absolutely *love* algebra once you understand what it is capable of doing.

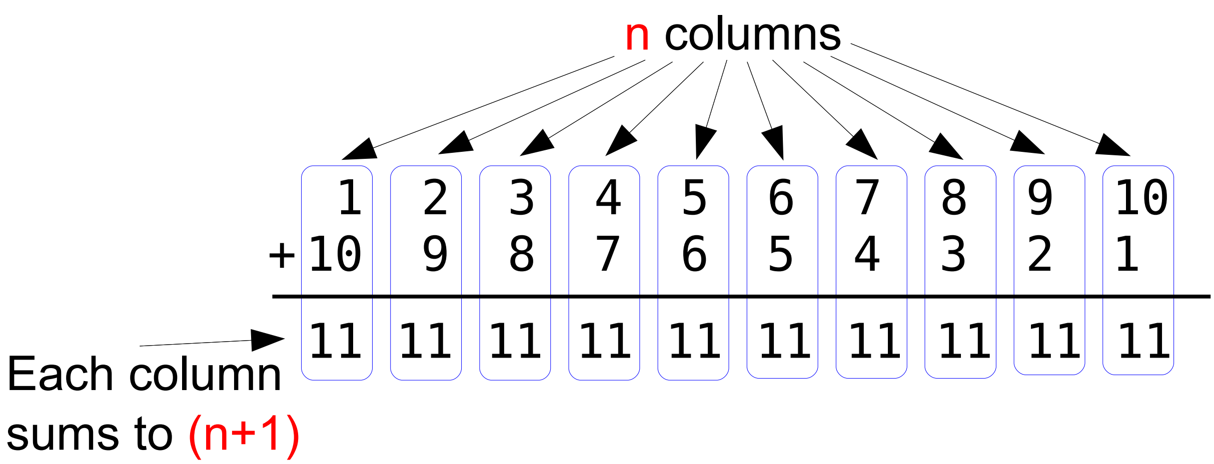
If you were never very good at algebra, don't worry because all of the programming knowledge you have gained up to this point should help you understand what algebra is and how it works. For example, variables in algebra are very similar to variables in computer programming so if you understand how programming variables work, this should give you a good feel for how algebra variable work too.

Now, lets see how the patterns we noticed in the above examples can be expressed in mathematical notation. Each sequence of integers we will be interested in working with will have a different number of integers in it and **the largest integer will be equal to the total number of integers in the sequence**. For example, the list {1,2,3} has 3 integers in it and the largest integer is 3. The list {1,2,3,4,5} has 5 integers in it and the largest integer is 5, etc.

Since this value varies for different sequences, it makes sense to represent it with a **variable**. Let use the variable n with n standing for the **n**umber of values in the sequence. In the following diagram you can see that n is equivalent to the number of columns we have identified:



And the next diagram shows that each column sums to (n+1):



Now that the number of columns and the value that each column sums to have been converted into mathematical notation, what can be done with this information? Lets experiment with it. What value would we get if we multiplied **n** and **(n+1)** together like this?:

$$n*(n+1) \tag{1}$$

For a sequence that had 10 integers in it, n would equal 10 and therefore $n*(n+1)$ would become $10*(10+1)$ which equals 110:

```
In> 10*(10 + 1)
Result: 110
```

What relationship does 110 have to the sum of the numbers 1-10? Lets calculate the actual sum using our sumOfIntegers() function:

```
In> sumOfIntegers(10)
Result: 55
```

The number 110 is twice the size of 55 and, if you think about it, this makes sense because $n*(n+1)$ represents the sum of **two** sequences of integers from 1-10 instead of just one. Since $n*(n+1)$ is always going to give us a result that is **twice** as large as the one we desire, all we need to do is to **cut it in half**. We can do this by dividing it by 2 and here is the adjusted mathematical expression:

$$\frac{n*(n+1)}{2} \tag{2}$$

The following example shows the new expression being tested with $n=10$ and checked with our sumOfIntegers() function:

```
In> (10*(10 + 1))/2
Result: 55

In> sumOfIntegers(10)
Result: 55
```

The new expression works with 10, but does it work with other values? Lets see:

```
In> (100*(100 + 1))/2
Result: 5050

In> sumOfIntegers(100)
Result: 5050


In> (1000*(1000 + 1))/2
Result: 500500

In> sumOfIntegers(1000)
Result: 500500
```

It seems to work! But now we need to determine if using this algebraic expression is really **faster** than the looping approach we have been using. In other words, is it a jump point? In order to test it we will put the expression into a function called **fastSumOfIntegers()** and then use the same timing code we used earlier to test sumOfIntegers():

```
%mathpiper,title=""
```

```
fastSumOfIntegers(n) := (n*(n + 1))/2;

%/mathpiper
```

```
%mathpiper,title=""

ForEach(highNumber, {100,1000,10000,100000,1000000,10000000,100000000,1000000000})
[
    runTime := Time() sum := fastSumOfIntegers(highNumber);

    Echo("High number: ", highNumber);

    Echo("Sum: ", sum);

    Echo("Run time: ", runTime, "seconds.");

    NewLine();
];

%/mathpiper

    %output,preserve="false"
      Result: True

      Side Effects:
      High number: 100
      Sum: 5050
      Run time: 0.004019436 seconds.

      High number: 1000
      Sum: 500500
      Run time: 0.004275194 seconds.

      High number: 10000
      Sum: 50005000
      Run time: 0.003729804 seconds.

      High number: 100000
      Sum: 5000050000
      Run time: 0.003852794 seconds.

      High number: 1000000
      Sum: 500000500000
      Run time: 0.005721676 seconds.

      High number: 10000000
      Sum: 50000005000000
      Run time: 0.005777341 seconds.

      High number: 100000000
      Sum: 5000000050000000
      Run time: 0.005822806 seconds.
```

```
    High number: 1000000000
    Sum: 500000000500000000
    Run time: 0.005872953 seconds.
.   %/output
```

As the timed results of this program indicate, the mathematical expression $(n*(n+1))/2$ definitely appears to behave like a hyperspace jump point for jumping through the integers pattern spaces! This is an exciting discovery and it should lead you to wonder if there are more pattern space jump points sprinkled throughout the universe of pattern spaces.

3.2.4 Formulas And Theorems Are Like Hyperspace Jump Points

As it turns out, pattern space jump points are present in all kinds of pattern spaces. You have seen for yourself how powerful a pattern space jump point can be. In fact, they are so important that there is a group of people who spend most of their time searching for new pattern space jump points and using existing ones. These people are called **mathematicians** and what we have been referring to as **jump points** they call **formulas** and **theorems**.

Many people have the impression that the life of a mathematician is extremely boring. Hopefully our discussion has shown you that discovering a jump point in a pattern space and then using it is actually exciting and that mathematicians have had us all fooled! When a mathematician does math, it looks like they are just sitting there not doing much of anything at all. But what they are actually doing is locating pattern space jump points and using them to navigate to useful and exotic places deep within pattern space, just like what Han and Chewie do in physical space.

From this point on we are going to being using the words **formula** and **theorem** when referring to pattern space jump points. However, instead of providing formal definitions for these words, we going to continue to use the intuitive descriptions that we have developed for them.

3.2.5 The Greek Letter Sigma (Σ) Is Used In Mathematical Notation To Represent The Sum Operation

3.2.5.1 Summing Without Applying An Operation

Earlier it was mentioned that mathematical notation is like a programming language and a good example of this is how it uses the Greek letter Sigma (Σ) to represent the operation of **summing**. As you recall, the MathPiper **Sum()** function is used to determine the sum of the values in a list:

```
In> Sum(1 .. 3)
Result: 6
```

The equivalent to this code in mathematical notation is as follows:

$$\sum_{x=1}^{3} x \tag{3}$$

In this notation, **x is a variable** which is initialized to 1 and it is set to all the integer values in the sequence between 1 and the integer at the top of the sigma symbol, which is 3. As **x** is moved through the sequence of integers, the **sum** of all these integers is **accumulated** and when the end of the sequence is reached, the accumulated sum is returned as a result. Here is a version of expression (3) which shows the accumulating steps and the result:

$$\sum_{x=1}^{3} x = 1+2+3=6 \tag{4}$$

Finally, the following program is also equivalent to this notation but, unlike the first code example, it shows the details of the summing logic:

```
%mathpiper,title=""

sum := 0;

x := 1;

While(x <= 3)
[
    sum := sum + x;

    x++;
];

sum;

%/mathpiper

    %output,preserve="false"
      Result: 6
.   %/output
```

3.2.5.2 Applying An Operation While Summing

An operation can also be applied to the variable during the summing operation. For example, the following version of the notation shows **x being multiplied by 2**, the intermediate values that are calculated, and the sum of the values which is the result:

$$\sum_{x=1}^{3} 2*x=2+4+6=12 \tag{5}$$

Finally, the following code shows what summing while applying an operation looks like as a program. It is the same as the previous program, except that **x is now being multiplied by 2** each time through the loop:

```
%mathpiper,title=""

sum := 0;

x := 1;

While(x <= 3)
[
    sum := sum + 2*x;

    x++;
];

sum;

%/mathpiper

    %output,preserve="false"
      Result: 12
.   %/output
```

3.2.6 Mathematical Notation Was Designed To Relay Complex Ideas With A Minimum Amount Of Writing (Or Typing!)

In the previous section you may have noticed that the mathematical notation of the summing operation was significantly shorter than its expanded program equivalents. Mathematical notation evolved slowly over hundreds of years and it was specifically designed to relay complex ideas with a minimum amount of writing (or typing!). The notation is often described as being "dense" because a little bit of it is capable of representing enormous amounts of information.

For a person who already knows mathematical notation, its high density is wonderful because it allows complex ideas to be communicated and manipulated with relatively little effort. However, for people who are just starting to learn mathematical notation, its density makes it challenging to learn.

3.2.6.1 Knowing How To Program Makes Learning Mathematics Easier

The nice thing about knowing how to program before learning mathematics (beyond the level of arithmetic) is that programs can be used to explain what a given section of mathematical notation is doing in an expanded form. This technique was used in the sigma notation section and it will continue to be used throughout the rest of the book.

*3.2.6.2 Using Juxtaposition Notation For Multiplication Instead Of The * Operator In Mathematical Notation*

One way that mathematical notation increases its density is by using **juxtaposition** instead of the * operator for multiplication. **Juxtaposition** simply means to place two things next to each other and in certain parts of mathematical notation, placing two symbols next to each other indicates that they should be multiplied. For example,

$$2 * x \tag{6}$$

can be written without the * symbol like this:

$$2\mathrm{x} \tag{7}$$

Using juxtaposition to indicate multiplication allows us to write formula (2) which we developed earlier for quickly summing sequences of integers from 1 to n as follows:

$$\frac{n(n+1)}{2} \tag{8}$$

And here is how expression (5) which used sigma notation that included a multiplication operation can be written:

$$\sum_{x=1}^{3} 2x = 2+4+6 = 12 \qquad (9)$$

In mathematical notation, juxtaposition is used to indicate multiplication much more often than * is and so the juxtaposition form will be the one used in the rest of this book.

3.3 Exercises

For the following exercises, create a new MathRider worksheet file called **book_2_section_3_exercises_<your first name>_<your last name>.mrw**. (**Note: there are no spaces in this file name**). For example, John Smith's worksheet would be called:

book_2_section_3_exercises_john_smith.mrw.

After this worksheet has been created, place your answer for each exercise that requires a fold into its own fold in this worksheet. Place a title attribute in the start tag of each fold which indicates the exercise the fold contains the solution to. The folds you create should look similar to this one:

```
%mathpiper,title="Exercise 1"

//Sample fold.

%/mathpiper
```

If an exercise uses the MathPiper console instead of a fold, copy the work you did in the console into the worksheet so it can be saved.

3.3.1 Exercise 1

The following code creates a permutation pattern space which contains two legitimate sentences and these sentences have been marked with an 'X':

```
In> TableForm(PermutationsList({Bill,ate,a,carrot}))
Result: True
Side Effects:
{Bill,ate,a,carrot} X
{Bill,ate,carrot,a}
{Bill,carrot,ate,a}
{carrot,Bill,ate,a}
{Bill,a,ate,carrot}
{Bill,a,carrot,ate}
{Bill,carrot,a,ate}
{carrot,Bill,a,ate}
```

```
{a,Bill,ate,carrot}
{a,Bill,carrot,ate}
{a,carrot,Bill,ate}
{carrot,a,Bill,ate}
{ate,Bill,a,carrot}
{ate,Bill,carrot,a}
{ate,carrot,Bill,a}
{carrot,ate,Bill,a}
{ate,a,Bill,carrot}
{ate,a,carrot,Bill}
{ate,carrot,a,Bill}
{carrot,ate,a,Bill}
{a,ate,Bill,carrot}
{a,ate,carrot,Bill}
{a,carrot,ate,Bill} X
{carrot,a,ate,Bill}
```

Use the PermutationList() function to create a permutation pattern space which contains more than two legitimate sentences.

3.3.2 Exercise 2

Give the MathPiper code that will calculate how many ways 30 books can be arranged on a bookshelf.

3.3.3 Exercise 3

a) Create a loop-based function called **productOfIntegers(highestInteger)** which takes a positive integer as its input and returns the product of 1 through this integer. For example, calling **productOfIntegers(3)** would calculate 1*2*3. Calling **productOfIntegers(4)** would calculate 1*2*3*4, etc.

b) The larger the integer that is sent to this function, the longer it will take to run. What is the smallest integer that still causes this function to take over 5 seconds to run?

3.3.4 Exercise 4

Create a program which will solve this riddle using only loops and the addition operator:

As I was going to St. Ives
I met a man with seven wives
Each wife had seven sacks
Each sack had seven cats
Each cat had seven kits
Kits, cats, sacks, wives
How many were going to St Ives?

Assume that the man and his wives are also going to St. Ives.

In order to help you visualize the riddle, here is a program which places the kits, cats, sacks, and wifes into a list:

```
%mathpiper,title=""

cat := FillList(kit,7);

sack := FillList(cat,7);

wife := FillList(sack,7);

man := FillList(wife,7);


%/mathpiper
```

And this program prints the list in an easy to read format:

```
%mathpiper

ForEach(wife, man)
[
    Echo("Wife");

    ForEach(sack, wife)
    [
        Echo("   Sack");

        ForEach(cat, sack)
        [
            Echo("       Cat");

            Space(12);

            ForEach(kit, cat)
            [
                Write(kit,,);

            ];
            NewLine();

        ];

    ];

];

%/mathpiper
```

3.3.5 Exercise 5

Give a mathematical expression which will solve the riddle from exercise 4.

3.3.6 Exercise 6

Create a program that is the equivalent of the following expression:

$$\sum_{x=1}^{10} 3x+4$$

4 Number System Patterns (Some Have Big Gaps, Some Have Small Gaps or No Gaps)

The **natural number** pattern space (or natural number **system**) was among the first patterns that humans developed and its study was the beginning of mathematics. Over time things in the physical world were discovered which could not be expressed with the natural number system and then a more comprehensive number system was developed by **extending** it. This more comprehensive number system eventually also needed to be extended and this lead to a series of number systems being developed by this process of extension.

In this section, the most commonly used number systems are described. These number systems are classified into **1)** those which have **big gaps** and **2)** those which have **small gaps or no gaps**. As you might have guessed, the **earlier** number systems are the ones that have **gaps** in them and these are the ones that will be discussed first.

4.1 Number Systems Which Have Big Gaps

4.1.1 $\mathbb{N}$ - The Natural (Or Counting) Numbers

The **natural number system** was the first number system that was developed and the types of problems it was designed to solve were those which involved **counting** things. For this reason, the natural numbers are also called the **counting numbers**. An example of an early need to count things is that of a shepherd who needed to determine how many sheep left a pen in the morning to graze so he could determine if any sheep were missing when they were put back into the pen at night. Other early examples which involved the need to count things are easy to imagine so there is no need to list further ones here.

The original version of the natural number system started with the number 1. Later, when the number 0 was invented, it was added to the natural number system and this version of the system is often referred to as the **whole numbers**. In this book we will us the version of the natural number system which starts with 0.

The first number in the natural number system is 0, the next number in the system is 1, the next one after that is 2, and so on to **infinity**. The way that one **moves from any given natural number to the next large one is by simply adding 1 to it**. In mathematical notation, $\mathbb{N}$ is the symbol which is used to represent the natural numbers and it stands for the word "Natural". The $\cdots$ symbol is used to indicate "**to infinity**". Expression (10) shows how the natural numbers are defined using mathematical notation:

$$\mathbb{N} = \{0, 1, 2, \cdots\} \qquad (10)$$

We can't actually show all the numbers from 1 to infinity because there is not enough matter in the universe to write them all on. However, a quote that comes to mind here is:

> “Money can't buy you happiness, but it can buy you a yacht big enough to pull up right alongside it.” David Lee Roth

While we can't show all of the numbers from 1 to infinity, with MathPiper we can show more of them that you could count in a lifetime! Here is a list of the natural numbers from 0 to 200:

```
In> Echo(0 .. 200)
Result: True
Side Effects:
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27
28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52
53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77
78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101
102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120
121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139
140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158
159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177
178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196
197 198 199 200
```

Feel free to use MathPiper to view more natural numbers than this, but don't be surprised if your computer runs out of memory and crashes if you try to display too many of them!

4.1.2 $\mathbb{Z}$ The Integers (Natural Numbers Plus Their Negative Equivalents)

The natural numbers are very useful, but people eventually encountered the need to work with counting-related ideas that the natural number system could not represent. For example, if a person has a total of 0 dollars and they borrow 5 dollars from someone, how much money do they have now? They have **less than** 0 dollars and negative numbers were invented to represent concepts like this. The notation for a negative number consists of placing a **minus sign** (**-**) in front of the **positive number which has the same magnitude as it** and therefore the person can be said to have **-5** dollars.

When the natural number system was extended to include negative numbers, a new number system was created which was called the **integer number system**. In mathematical notation, $\mathbb{Z}$ is the symbol which is used to represent the integers and it stands for the German word "Zahlen" which means "numbers".

Expression (11) shows how the natural numbers are defined using mathematical notation:

$$\mathbb{Z} = \{\cdots, -2, -1, 0, 1, 2, \cdots\} \tag{11}$$

Notice that $\cdots$ is used to represent negative infinity as well as positive infinity.

Finally, here is a list of the integers from -100 to +100:

```
In> Echo(-100 .. 100)
Result: True
Side Effects:
-100 -99 -98 -97 -96 -95 -94 -93 -92 -91 -90 -89 -88 -87 -86 -85 -84 -83
-82 -81 -80 -79 -78 -77 -76 -75 -74 -73 -72 -71 -70 -69 -68 -67 -66 -65 -64
-63 -62 -61 -60 -59 -58 -57 -56 -55 -54 -53 -52 -51 -50 -49 -48 -47 -46 -45
-44 -43 -42 -41 -40 -39 -38 -37 -36 -35 -34 -33 -32 -31 -30 -29 -28 -27 -26
-25 -24 -23 -22 -21 -20 -19 -18 -17 -16 -15 -14 -13 -12 -11 -10 -9 -8 -7 -6
-5 -4 -3 -2 -1 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72
73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97
98 99 100
```

4.1.3 The Big Gaps Which Exist Between Natural Numbers And Integers

4.1.3.1 Zooming Into The Area Between The Integers 0 And 1

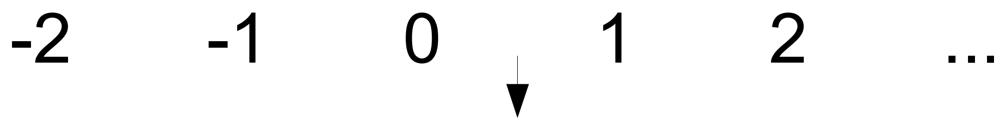
Now that we have discussed the natural numbers and integers, lets take a look at what lies between all of the numbers in these systems. We will do this by listing the integers near 0 and then zooming in towards the space that exists between 0 and 1:

... -2 -1 0 1 2 ...

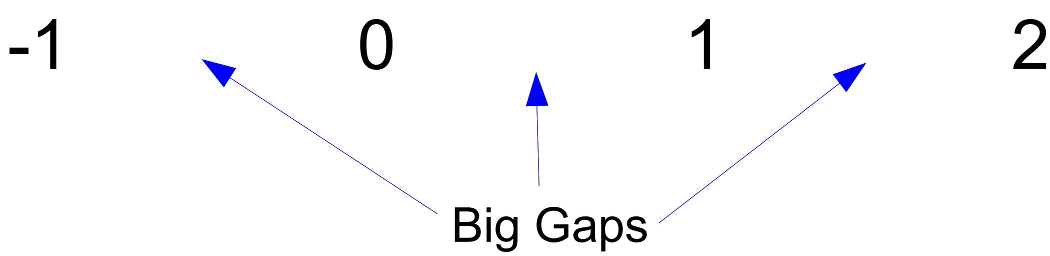
Zoom

... -2 -1 0 1 2 ...

Zoom



Zoom



As you can see, there is absolutely **nothing** between these numbers except **big gaps**. This means neither natural numbers nor integers can be used to represent **parts of things**. For example, if someone took an apple, cut it in half and then gave you one of the halves, there would be no way to represent this half with either of these number systems. Furthermore, if you are at a given number in these systems and you want to move to the next number, you do this by "jumping" over the gap that exists between them.

4.1.4 Natural Numbers And Integers Are Used When Counting Things (Discrete Mathematics)

You might think that natural numbers and integers are not very powerful because they can only represent **whole** objects. However, these numbers are extremely powerful because they are the numbers which are used when **counting things** and there are an enormous number of things in the universe that can be counted. For example, the permutation patterns we studied earlier are categorized as being part of discrete mathematics.

The area of mathematics which deals with these two number systems is called **discrete mathematics** (sometimes it is also referred to as **finite mathematics**, especially in the area of business). Here is a definition for the word discrete:

> Separate; distinct; individual; Something that can be perceived individually and not as connected to, or part of something else;
> en.wiktionary.org/wiki/discrete

You can see that the word **discrete** fits the concept of numbers separated by gaps fairly well.

Discrete mathematics and ways that it can be used will be discussed later in the book, but for now lets turn our attention to kinds of numbers which have small gaps or no gaps in them.

4.2 ℚ The Rational Numbers, A Number System Which Has Small Gaps

If an apple were cut it into 2 equals pieces with a knife, how can each piece be represented with numbers? One way this can be done is to use **two integers** and come up with a way to **relate them** which can be used to represent things which are **not whole**. Half an apple can be thought of as **1 apple** which was **cut into 2 pieces** and the given apple half is one of the pieces. Expressions (12), (13), and (14) are three equivalent notations which represent this concept:

$$\frac{1\,apple}{cut\,into\,2\,pieces} \text{ or } \frac{1}{2} \tag{12}$$

$$1\,apple \div cut\,into\,2\,pieces \text{ or } 1 \div 2 \tag{13}$$

$$1\,apple / cut\,into\,2\,pieces \text{ or } 1/2 \tag{14}$$

This system can also be used to represent 1 apple cut into 3 pieces $\left(\frac{1}{3}\right)$, 1 apple cut into 4 pieces $\left(\frac{1}{4}\right)$, 2 apples cut into 3 pieces $\left(\frac{2}{3}\right)$, 5 apples cut into 7 pieces $\left(\frac{5}{7}\right)$, etc. This system is called the **rational numbers system** because it consists of integers which are configured as a **ratio**. In mathematical notation, the rational numbers are represented by the symbol ℚ and it stands for the word "Quotients".

4.2.1 The Rational Number System Contains Equivalents To All The Integers

Rational numbers are good at representing parts of things, but they contain **equivalents to the integers** too. This means that rational numbers can represent anything integers can represent. For example, the rational number equivalent to the integer 0 is $\left(\frac{0}{1}\right)$, the equivalent of 1 is $\left(\frac{1}{1}\right)$, the equivalent of

2 is $\left(\frac{2}{1}\right)$, and so on.

4.2.2 Zooming Into The Area Between The Rational Numbers 0/1 And 1/1

When we zoomed into the area between the integers 0 and 1, all we found was an empty gap. Lets use the **NumberLinePrintZoom()** function to zoom into the area between the rational numbers 0/1 and 1/1 to see what is there. The calling format for **NumberLinePrintZoom()** is as follows:

```
NumberLinePrintZoom(low_value, high_value, divisions, depth)
```

The argument "low_value" indicates the lowest number in the zoom range, "high_value" indicates the highest number in the range, and "divisions" indicates how many pieces to divide the range into ("depth" will be covered in a moment). In the following code, the area between 0/1 and 1/1 is zoomed into and this area is divided into 8 divisions:

```
In> NumberLinePrintZoom(0/1,1/1,8,1)
Result: True
Side Effects:
0   1/8   1/4   3/8   1/2   5/8   3/4   7/8   1
```

First, notice that MathPiper converted 0/1 and 1/1 to their integer equivalents. MathPiper will **automatically convert any rational number into its integer equivalent if one exists**. Now, look at the area between 0/1 and 1/1. Instead of there being a gap between these rational numbers, we find that there are more rational numbers! Also notice that all of these rational numbers are in lowest terms because MathPiper **automatically converts all rational numbers to lowest terms**.

The following examples show the area between 0/1 and 1/1 being divided into 10 divisions, 20 divisions, and 50 divisions:

```
In> NumberLinePrintZoom(0/1,1/1,10,1)
Result: True
Side Effects:
0   1/10   1/5   3/10   2/5   1/2   3/5   7/10   4/5   9/10   1

In> NumberLinePrintZoom(0/1,1/1,20,1)
Result: True
Side Effects:
0   1/20   1/10   3/20   1/5   1/4   3/10   7/20   2/5   9/20   1/2   11/20
3/5   13/20   7/10   3/4   4/5   17/20   9/10   19/20   1

In> NumberLinePrintZoom(0/1,1/1,50,1)
Result: True
```

```
Side Effects:
0   1/50   1/25   3/50   2/25   1/10   3/25   7/50   4/25   9/50   1/5
11/50   6/25   13/50   7/25   3/10   8/25   17/50   9/25   19/50   2/5
21/50   11/25   23/50   12/25   1/2   13/25   27/50   14/25   29/50   3/5
31/50   16/25   33/50   17/25   7/10   18/25   37/50   19/25   39/50   4/5
41/50   21/25   43/50   22/25   9/10   23/25   47/50   24/25   49/50
```

You will find that as you increase the number of divisions that the area between 0/1 and 1/1 is divided into, more and more rational numbers will appear to mark the boundaries of the divisions. What this indicates is that there are an **infinite number of rational numbers between 0/1 and 1/1**. You can continue zooming into this area forever and never run out of rational numbers.

4.2.3 Going Beyond One Level In The Zoom (All Rational Numbers Have An Infinite Number Of Rational Numbers Between Them)

Not only do 0/1 and 1/1 have an infinite number of rational numbers between them, **all rational number have an infinite number of rational numbers between them**. One way the NumberLinePrintZoom() function can be used to show this is by sending it different values for "low_value" and "high_value":

```
In> NumberLinePrintZoom(1/10,2/10,10,1)
Result: True
Side Effects:
1/10   11/100   3/25   13/100   7/50   3/20   4/25   17/100   9/50   19/100
1/5
```

However, if the NumberLinePrintZoom() function's "depth" argument is set to a value which is greater than 1, the function will continue the zooming process to the number of levels specified by "depth". For example, the following code zooms into the area between 0/1 and 1/1 and then continues on to zoom into the area between 3/8 and 1/2 because "depth" is set to 2:

```
In> NumberLinePrintZoom(0/1,1/1,8,2)
Result: True
Side Effects:
0   1/8   1/4   3/8   1/2   5/8   3/4   7/8   1
                    |
-----------------------------------------------------------------
3/8   25/64   13/32   27/64   7/16   29/64   15/32   31/64   1/2
```

The **|** character is used to indicate which pair of rational numbers have been chosen for further division How did the function choose the space between 3/8 and 1/2 as the one it was going to zoom into? It was chosen randomly and if the function is executed again, a different pair of rational numbers will probably be chosen:

```
In> NumberLinePrintZoom(0/1,1/1,8,2)
Result: True
Side Effects:
0   1/8   1/4   3/8   1/2   5/8   3/4   7/8   1
                                              |
-----------------------------------------------------------------
7/8   57/64   29/32   59/64   15/16   61/64   31/32   63/64   1
```

Finally, here is the area between 0/1 and 1/1 being divided into 8 divisions and zoomed to a depth of 4:

```
In> NumberLinePrintZoom(0/1,1/1,8,4)
Result: True
Side Effects:
0   1/8   1/4   3/8   1/2   5/8   3/4   7/8   1
                           |
------------------------------------------------------------------
1/2   33/64   17/32   35/64   9/16   37/64   19/32   39/64   5/8
    |
-----------------------------------------------------------------------------------
1/2   257/512   129/256   259/512   65/128   261/512   131/256   263/512   33/64
                                                                            |
-----------------------------------------------------------------------------------------------------
263/512   2105/4096   1053/2048   2107/4096   527/1024   2109/4096   1055/2048   2111/4096   33/64
```

4.2.4 Irrational Numbers

Since there are an infinite number of rational numbers between any two given rational numbers, one might think that rational numbers are able to represent everything that can be represented by numbers. The ancient Greeks believed this until a Greek mathematician named Hippasus discovered that $\sqrt{2}$ couldn't be represented by a whole number nor a rational number. According to one legend, Greek mathematicians became so upset when they learned that the rational number system was inadequate for representing everything in the universe that they threw Hippasus into the sea!

Eventually mathematicians discovered that there were numerous numbers like $\sqrt{2}$ which could not be represented with rational numbers and they named these new numbers **irrational** (not rational) numbers. The discovery of this new kind of number meant that the **rational number system had small gaps in it** and therefore yet **another number system was needed which could fill the gaps**.

4.3 ℝ The Real Numbers And Decimal Representations

The real number system consists of all the **rational** numbers and all the **irrational** numbers and this system is able to represent most things in the universe that can be represented by numbers. Because of this, the areas of science and engineering rely heavily on the real numbers and most physical constants and variables are represented with them. In mathematical notation, the real numbers are represented by the symbol $\mathbb{R}$ which stands for the word "Real".

Computers are unable to work with real numbers directly because many of them have extremely long representations which would not fit into a computer's finite memory space. Instead, most computers work with real numbers using **approximate decimal representations** which have a **limited number of decimals**. However, computer algebra systems like MathPiper can also represent real numbers symbolically. For example MathPiper represents the real number which is the square root of 2 ($\sqrt{2}$) with **Sqrt(2)**.

Computer algebra systems represent real numbers symbolically whenever possible because these systems are designed to perform calculations as accurately as possible. This emphasis on accuracy is also why computer algebra systems work with rational numbers by default. However, these systems usually provide a way to obtain either 1) **exact decimal representations** of rational numbers (if they exist) or 2) **approximate decimal representations** of rational numbers and symbolically represented real numbers to as many decimals of precision as the user specifies (within the limits of the computer's memory).

4.3.1 Obtaining Decimal Representations Of Real Numbers With N()

In MathPiper, the **N()** function is used to obtain decimal representations of rational numbers and symbolically represented real numbers to a specified number of significant decimals (or precision). Here are the two calling formats for the N() function:

```
N(expression)
N(expression, precision)
```

The argument "expression" is any MathPiper expression and "precision" specifies how many significant digits the decimal approximation of the expression that N() returns contains. If no precision is specified, the system default precision of 10 significant digits is used. Lets begin our exploration of the the N() function by using it to obtain various decimal approximations of $\sqrt{2}$. First, here is what Sqrt(2) returns without using the N() function:

```
In> Sqrt(2)
Result: Sqrt(2)
```

Since Sqrt(2) is how MathPiper symbolically represents $\sqrt{2}$, it simply returned it as the result because this is the most accurate representation it has for $\sqrt{2}$. Now, lets use **N()** to obtain a decimal approximate of $\sqrt{2}$ to 1, 2, 3, 5, 10, 20, 50, and 100 significant digits of precision:

```
In> N(Sqrt(2), 1)
Result: 1

In> N(Sqrt(2), 2)
Result: 1.4

In> N(Sqrt(2), 3)
Result: 1.41

In> N(Sqrt(2), 5)
Result: 1.4142

In> N(Sqrt(2), 10)
Result: 1.414213562

In> N(Sqrt(2), 20)
Result: 1.4142135623730950488

In> N(Sqrt(2), 50)
Result: 1.4142135623730950488016887242096980785696718753769

In> N(Sqrt(2), 100)
Result:
1.414213562373095048801688724209698078569671875376948073176679737990732478
4621070388503875343276415727
```

These examples are performing what is called a **decimal expansion** on the number $\sqrt{2}$. If a number has an **infinite** number of decimals like $\sqrt{2}$ does, it is said to have an **infinite decimal expansion** and it is also said to have **infinite precision**. However, a computer is only capable of approximating infinite precision numbers to a given precision.

Also notice that the decimal approximate of this irrational number does **not have any pattern of repeated digits**. All irrational numbers have this property.

If no precision is specified, N() will return a decimal approximate to 10 significant digits of precision because this is MathPiper's default precision:

```
In> N(Sqrt(2))
Result: 1.414213562
```

4.3.2 Obtaining Decimal Representations Of Rational Numbers With N()

The N() function can also be used to obtain decimal representations of rational numbers. However, unlike symbolically represented real numbers like Sqrt(2) (which are always represented approximately), the decimal representations of rational numbers can either be **exact** or **approximate**. Here are some examples of rational numbers that can be represented by decimals exactly:

```
In> N(1/2)
Result: 0.5
```

```
In> N(1/4)
Result: 0.25
```

```
In> N(3/8)
Result: 0.375
```

```
In> N(5/16)
Result: 0.3125
```

Notice that even though the system is configured to use 10 digits of precision by default, less digits than this are needed to represent these rational numbers. The above exact decimal representations are called **finite decimal fractions** and they have **finite decimal expansions**.

The following rational numbers **cannot be represented exactly by decimals** so they are represented approximately to a specified precision:

```
In> N(1/3, 50)
Result: 0.33333333333333333333333333333333333333333333333333
```

```
In> N(1/9, 50)
Result: 0.11111111111111111111111111111111111111111111111111
```

```
In> N(1/27, 50)
Result: 0.037037037037037037037037037037037037037037037037037
```

```
In> N(1/11, 50)
Result: 0.090909090909090909090909090909090909090909090909091
```

```
In> N(1/81, 53)
Result: 0.012345679012345679012345679012345679012345679012346789
```

Each of these rational numbers have an infinite decimal expansions just like $\sqrt{2}$ does, but notice that these decimal expansions contain **repeating digits** where the decimal expansion for $\sqrt{2}$ did not contain repeating digits. Any rational number that cannot be represented with a finite decimal fraction has an infinite decimal expansion which contains repeating digits. Irrational numbers like $\sqrt{2}$, on the other hand, have infinite decimal expansions which do not contain repeating digits.

4.3.3 Obtaining Rational Representations Of Decimal Numbers

If you have a decimal number and you would like to have a rational number representation of it, the **Rationalize()** function can be used to obtain one. For example, Rationalize() can obtain the equivalent rational numbers for finite decimal fractions:

```
In> Rationalize(0.5)
Result: 1/2

In> Rationalize(0.25)
Result: 1/4

In> Rationalize(0.375)
Result: 3/8

In> Rationalize(0.3125)
Result: 5/16
```

However, for numbers which are not finite decimal fractions, only rational numbers which approximate them can be obtained:

```
In> Rationalize(0.33333333333333333333333333333333333333333333333333)
Result: 1666666667/5000000000

In> Rationalize(0.11111111111111111111111111111111111111111111111111)
Result: 138888889/1250000000

In> Rationalize(0.037037037037037037037037037037037037037037037037037)
Result: 740740741/20000000000

In> Rationalize(0.090909090909090909090909090909090909090909090909091)
Result: 2272727273/25000000000

In> Rationalize(0.012345679012345679012345679012345679012345679012346789)
Result: 617283951/50000000000
```

4.3.4 Zooming Into The Area Between The Real Numbers 0.0 And 1.0

When we zoomed into the area between the rational numbers 0/0 and 0/1, we found that further we zoomed, the more rational numbers we found. Lets use the **NumberLinePrintZoom()** function to zoom into the area between the real numbers 0.0 and 1.0 to see what is there.

```
In> NumberLinePrintZoom(0.0,1.0,8,1)
Result: True
Side Effects:
0.0   0.125   0.250   0.375   0.500   0.625   0.750   0.875   1.000
```

The following examples show the area between 0.0 and 1.0 being divided into 10 divisions, 20 divisions, and 50 divisions:

```
In> NumberLinePrintZoom(0.0,1.0,10,1)
Result: True
Side Effects:
0.0   0.1   0.2   0.3   0.4   0.5   0.6   0.7   0.8   0.9   1.0

In> NumberLinePrintZoom(0.0,1.0,20,1)
Result: True
Side Effects:
0.0   0.05   0.10   0.15   0.20   0.25   0.30   0.35   0.40   0.45   0.50
0.55   0.60   0.65   0.70   0.75   0.80   0.85   0.90   0.95   1.00

In> NumberLinePrintZoom(0.0,1.0,50,1)
Result: True
Side Effects:
0.0   0.02   0.04   0.06   0.08   0.10   0.12   0.14   0.16   0.18   0.20
0.22   0.24   0.26   0.28   0.30   0.32   0.34   0.36   0.38   0.40   0.42
0.44   0.46   0.48   0.50   0.52   0.54   0.56   0.58   0.60   0.62   0.64
0.66   0.68   0.70   0.72   0.74   0.76   0.78   0.80   0.82   0.84   0.86
0.88   0.90   0.92   0.94   0.96   0.98   1.00
```

You will find that as you increase the number of divisions that the area between 0.0 and 1.0 is divided into, more and more real numbers will appear to mark the boundaries of the divisions. What this indicates is that there are an **infinite number of real numbers between 0.0 and 1.0**. You can continue zooming into this area forever and never run out of real numbers.

4.3.5 Going Beyond One Level In The Zoom (All Real Numbers Have An Infinite Number Of Real Numbers Between Them)

Not only do 0.0 and 1.0 have an infinite number of rational numbers between them, **all real number have an infinite number of real numbers between them**. Again, the way the NumberLinePrintZoom() function can be used to show this is by sending it different values for "low_value" and "high_value":

```
In> NumberLinePrintZoom(.1,.2,10,1)
Result: True
Side Effects:
.1   0.11   0.12   0.13   0.14   0.15   0.16   0.17   0.18   0.19   0.20
```

Here is the area between 0.0 and 1.0 being divided into 8 divisions and zoomed to a depth of 4:

```
In> N(NumberLinePrintZoom(0.0,1.0,8,4),6)
Result: True
Side Effects:
0.0   0.125   0.250   0.375   0.500   0.625   0.750   0.875   1.000
```

```
                                     |
----------------------------------------------------------------------------------------------
0.500   0.515625   0.531250   0.546875   0.562500   0.578125   0.593750   0.609375   0.625000
                                        |
-------------------------------------------------------------------------------------------------
0.546875   0.548828   0.550781   0.552734   0.554687   0.556640   0.558593   0.560546   0.562499
                                                                            |
-------------------------------------------------------------------------------------------------
0.558593   0.558837   0.559081   0.559325   0.559569   0.559813   0.560057   0.560301   0.560545
```

The precision has been set to 6 in this example so that the output would fit on the page without wrapping. When you experiment with this example in MathRider, you can either remove the N() function or set it to a higher precision.

4.3.6 Rational Numbers And Real Numbers Are Used When Measuring Things (Continuous Mathematics).

While integers are used when counting things, rational numbers, or the decimal representation of a real numbers, are used to **measure things**. Types of measurements include length, width, thickness, weight, time, temperature, color, etc. Instead of having empty gaps, quantities like these are **continuous** which means that **between one measurement and another there are an infinite number of measurements**. Rational numbers and real numbers are well suited for representing measured quantities because the infinite number of rational numbers or real numbers between any two measurements can be used to represent the infinite amount of "stuff" which exists between two parts of something.

The area of mathematics which deals with continuous quantities is called **continuous mathematics** and we will be discussing continuous mathematics later in the book.

4.3.7 Imaginary Numbers And Complex Numbers

Real numbers are capable of representing most things that can be represented by numbers, but not all things. **Imaginary** numbers and **complex** numbers were invented to represent things that real numbers could not represent. However, instead of discussing them here, they will be discussed in a later section.

4.4 Exercises

For the following exercises, create a new MathRider worksheet file called **book_2_section_4_exercises_<your first name>_<your last name>.mrw**. (**Note: there are no spaces in this file name**). For example, John Smith's worksheet would be called:

book_2_section_4_exercises_john_smith.mrw.

After this worksheet has been created, place your answer for each exercise that requires a fold into its own fold in this worksheet. Place a title attribute in the start tag of each fold which indicates the exercise the fold contains the solution to. The folds you create should look similar to this one:

```
%mathpiper,title="Exercise 1"

//Sample fold.

%/mathpiper
```

If an exercise uses the MathPiper console instead of a fold, copy the work you did in the console into the worksheet so it can be saved but do not put it in a fold.

4.4.1 Exercise 1

The following code uses integers for counting. How many times does the Echo() function in this code get called when the code is executed?

```
%mathpiper,title="Counting."

outerMostLoopCounter := 0;

While(outerMostLoopCounter <= 5)
[
   middleLoopCounter := 0;

   While(middleLoopCounter <= 7)
   [
      innerMostLoopCounter := 0;

      While(innerMostLoopCounter <= 3)
      [
         //How many times does this Echo() function get called?
         Echo(outerMostLoopCounter,,,middleLoopCounter,,,innerMostLoopCounter);

         innerMostLoopCounter++;
      ];

      middleLoopCounter++;
   ];

   outerMostLoopCounter++;
];

%/mathpiper
```

4.4.2 Exercise 2

Give the code that will zoom into the area between two rational numbers which interest you.

4.4.3 Exercise 3

Give the code that will zoom into the area between two real numbers which interest you.

THE CONTENT BELOW THIS LINE IS IN DEVELOPMENT

4.4.4 Exercise 4

Obtain a ruler (you can use an Internet search engine to locate a printable ruler if you do not have one).

5 The Modeling And Simulation Of Systems

In this book, we are going to use the following definitions for **system**, **model**, and **simulation**:

> **System** A system exists and operates in time and space.
>
> **Model** A model is a simplified representation of a system at some particular point in time or space intended to promote understanding of the real system.
>
> **Simulation** A simulation is the manipulation of a model in such a way that it operates on time or space to compress it, thus enabling one to perceive the interactions that would not otherwise be apparent because of their separation in time or space.

> Modeling and Simulation is a discipline for developing a level of understanding of the interaction of the parts of a system, and of the system as a whole. The level of understanding which may be developed via this discipline is seldom achievable via any other discipline.
>
> A system is understood to be an entity which maintains its existence through the interaction of its parts. A model is a simplified representation of the actual system intended to promote understanding. Whether a model is a good model or not depends on the extent to which it promotes understanding. Since all models are simplifications of reality there is always a trade-off as to what level of detail is included in the model. If too little detail is included in the model one runs the risk of missing relevant interactions and the resultant model does not promote understanding. If too much detail is included in the model the model may become overly complicated and actually preclude the development of understanding. One simply cannot develop all models in the context of the entire universe, of course unless you name is Carl Sagan.
>
> A simulation generally refers to a computerized version of the model which is run over time to study the implications of the defined interactions. Simulations are generally iterative in there development. One develops a model, simulates it, learns from the simulation, revises the model, and continues the iterations until an adequate level of understanding is developed.
>
> Gene Bellinger http://www.systems-thinking.org/modsim/modsim.htm

In the next few sections we are going to use **integers** to model **counting-oriented (or discrete)** systems and **rational and real numbers** to model **measurement-oriented (or continuous)** systems. Then, we will use computer programs to simulate these systems to learn more about them and to make predictions with them.

6 Counting Based Simulations Which Use Probability

In a previous section **integers** and the **RandomInteger()** function were used to **simulate** the **flipping of a coin** and the **rolling of dice**. Both of these simulations were **counting** based (or **discrete**) simulations because flipping coins and rolling dice can only be **counted**, not measured. This section begins where those simulations left off and then builds upon the techniques they used to create more sophisticated discrete simulations.

Before we can experiment with discrete simulations, we first need to discuss the concept of **probability**. **Probability** is the likelihood of a particular event occurring and probabilities are associated with almost all parts of a person's life. For example, on a given day the weather forecast might predict a 50% chance of rain , you might have a 5% chance of dropping food on your shirt, and you may have a 10% chance of hearing a joke that makes you laugh so hard that it bring tears to your eyes.

Before we can model and simulate probabilities, we need to cover the following definitions which are related to it:

- **Fact** - a piece of information about circumstances that exist or events that have occurred. (http://wordnetweb.princeton.edu/perl/webwn). An example is information on which side of a die is currently facing up.
- **Data** - a collection of facts from which conclusions can be drawn. (http://wordnetweb.princeton.edu/perl/webwn). An example is which sides are currently facing up on two dice.
- **Experiment** - measuring or observing a system to collect data on it. An example is rolling two dice.
- **Outcome** - the result of a given experiment. An example is rolling a 3 and a 4 on two dice.
- **Sample space** - all of the possible outcomes of a given experiment. For the experiment of rolling two dice, the sample space is {2,3,4,5,6,7,8,9,10,11,12}.
- **Event** - a collection of one or more possible outcomes of an experiment that are of interest in a given situation. An example is rolling a die and obtaining a number that is ≤ 3 .

Now that these definitions have been presented, the next step is to present some rules related to probability.

6.1 Rules Related To Probability

- A **probability** consists of a **quantitative representation** of the likelihood of a particular event.

- Probability is represented with a real number between 0 and 1 (or 0% and 100%) inclusive.
- A probability of 0 means that the event will not occur.
- Rare events have probabilities which are close to 0.
- Common events have probabilities which are close to 1.
- A probability of 1 means that the event will definitely occur.
- The sum of all the probabilities for the events in a given sample space must equal 1.

6.2 The Three Kinds Of Probabilities

6.2.1 Subjective Probability

Subjective probability consists of a person using only their judgment to determine the likelihood of a given event occurring. This kind of probability is used when data related to an event does not exist and it would be impossible or too expensive to collect. Here are some examples where subjective probability may be used:

- The probability that the next time you visit the supermarket the floors would have just been waxed (perhaps .05 or 5%).
- The probability that the next time you visit a restaurant that has a buffet, the container which contains your favorite food is empty when you go to fill your plate (perhaps .3 or 30%).
- The probability that the next time you order a pop or soda at a restaurant it does not have enough carbonation in it (perhaps .2 or 20%).
- The probability that the next time you start to drift off to sleep, you suddenly feel like you are falling which causes you to wake up abruptly (perhaps .02 or 2%).

6.2.2 Theoretical Probability

Theoretical (or classical) probability is used when **1)** the **total number of possible outcomes** of a given experiment is **known** **2)** each of these outcomes are **equally likely to occur** and **3)** the **number of possible outcomes** in which a given event (lets call it event A) can occur is **known**.

We will use the following notation for representing the probability P of event A occurring:

$$P[A] = \textit{The probability of event A occurring.} \qquad (15)$$

The probability of event A occurring can be calculated using the following formula:

$$P[A] = \frac{\text{The number of possible outcomes in which event A can occur in an experiment.}}{\text{The total number of possible outcomes in the experiment's sample space.}} \quad (16)$$

6.2.2.1 Calculating The Probability That A Single Die Will Come Up 5 When Rolled Using Theory

An example where classical probability can be used is the rolling of a die. The experiment consists of rolling the die and event A is the outcome of the die coming up 5. The possible outcomes in the experiment is {1,2,3,4,5,6} and the total number of these outcomes is 6. The possible number of ways that the number 5 can come up on a single die is 1. The probability that the die will come up 5 can be calculated as follows:

$$P[A] = \frac{\text{1 possible outcome where the die can come up 5.}}{\text{6 possible outcomes in the die's sample space.}} = \frac{1}{6} = .167 \text{ or } 16.7\% \quad (17)$$

The number 5 has a probability of $\frac{1}{6}$ or .167 of coming up on any given roll. Actually, since all six sides of a die have an equal probability of coming up on a given roll, they all have a probability of $\frac{1}{6}$. Remember the probability rule which states that all the probabilities in a sample space must equal 1? Lets see if the rule holds for a die's sample space:

```
In> 1/6 + 1/6 + 1/6 + 1/6 + 1/6 + 1/6
Result: 1
```

Yes, it does! Now, lets say that event B is the outcome that a single die will come up less than 4. What is the probability for this event? The possible outcomes for event B are {1,2,3} and their total number is 3. therefore the calculation is

$$P[B] = \frac{\text{2 possible outcomes where the die can come up < 4.}}{\text{6 possible outcomes in the die's sample space.}} = \frac{2}{6} = .333 \text{ or } 33.3\% \quad (18)$$

These single die examples of using theoretical probability are very simple, but theoretical probability can be used with more complex problems like determining the probability of obtaining a certain combination of cards from a well-shuffled deck or the probability of winning a lottery.

Why is this kind of probability called theoretical probability? In order to determine this one must first understand what theory is and here is a definition for it:

> **Theory** - A theory, in the general sense of the word, is an analytic structure designed to explain a set of observations. (http://en.wikipedia.org/wiki/Theory)

This definition indicates that theory is based on an **analytic structure** and that this structure is designed to explain a set of observations. But what does "analytic structure" mean? Here is a definition for the word analytic:

> **Analytic** - using or subjected to a methodology using **algebra and calculus**. (wordnetweb.princeton.edu/perl/webwn)

What these definitions are indicating is that **theory** is based upon **algebraic and calculus-related structures** which contain information about something of interest that has been **observed**. For those people who don't know what calculus is yet (which means most people who are likely to be reading this book), when we refer to an **analytic structure**, you can think **algebraic structure**. An **algebraic structure** has a **specific meaning in mathematics** but instead of using the precise mathematical meaning, we can use an **intuitive pattern space based meaning** for now.

Theory can be thought of as **1)** a pattern space which is arranged in such a way that it **models** something of interest that has been observed and **2)** one or more **formulas** which can be used to **navigate** this pattern space. Analytic structures typically have **enormous pattern spaces** and **formulas** are used to **navigate** to areas of interest in this pattern space.

Since theoretical probability is based on theory, it also has an enormous pattern space and this pattern space is navigated by formula (16). The theoretical probability pattern space and formula can be used to model any thing in the universe which matches its structure. If you have the theory related to something, you already have an enormous amount of information about it and this information can be used to make predictions about how the something will behave under various conditions. The amazing thing is that all of these predictions can be done without having to work directly with the something of interest at all. The predictions can all be done on paper, or on a computer!

This section showed how theoretical probability can be used to predict what will happen when a single die is rolled and it also mentioned how it can be used to predict what will happen in situations like obtaining cards from a deck or playing a lottery. How, you may ask, was the theory for theoretical probability developed in the first place? This theory was developed by observing games of chance and then developing a pattern space which modeled how they worked and then developing formulas which could be used to navigate the pattern space.

However, as stated at the beginning of this section, theoretical probability can only be used if **1)** the **total number of possible outcomes** of a given experiment is **known** **2)** each of these outcomes are **equally likely to occur** and **3)** the **number of possible outcomes** in which a given event (lets call it event A) can occur is **known**. If all these criteria cannot be met, then theoretical probability cannot be used. This does not mean that it is impossible to determine probabilities for something if these criteria cannot be met and the next section discusses how the probabilities can be obtained using another technique.

6.2.3 Empirical Probability

Empirical Probability is used when **1)** the **total number of possible outcomes** in an experiment's sample space is **unknown** and **2)** the **number of possible outcomes in which an event can occur** in the experiment is also **unknown**. This sounds complicated but it is actually fairly simple. Lets start with a definition for the word "empirical":

> **Empirical** - derived from experiment and observation rather than theory. (http://wordnetweb.princeton.edu/perl/webwn)

Empirical just means to **perform some experiments** and **observe** what happens instead of using theory to calculate what should happen.

The probability of event A occurring can be calculated using the following formula:

$$P[A]=\frac{\text{The frequency in which event A did occur in an experiment.}}{\text{The total number of observations of experiment observations.}} \tag{19}$$

In the physical world, performing experiments, observing the outcomes, and recording these outcomes can be tedious. However, this process can also be simulated with a computer program and usually much easier, cheaper, and faster than it can be done physically.

6.2.3.1 Determining The Probability That A Single Die Will Come Up 5 When Rolled Using Simulation (And The Law Of Large Numbers)

In the section on theoretical probability, we used theory to calculate that the probability that a 5 will come up when a single die is rolled is $\frac{1}{6}$ or .167. Lets now use empirical probability and a simulation of rolling a die to see if the simulation and theory agree. The event we are looking for is a 5 being rolled. The following code simulates the rolling of a single die 10 times, counts the number of 5's that came up (which is the frequency with which our "5" event occurred), and then calculates the probability of this event using (19), the empirical probability formula:

```
In> dieRollsList := RandomIntegerVector(10,1,6)
Result: {2,6,5,1,1,4,4,1,2,4}

In> numberOfFives := Count(dieRollsList,5)
Result: 1

In> N(numberOfFives/10)
Result: 0.1
```

In this case, the frequency with which the 5's were observed was 2 and the total number of rolls was 10. The empirical probability was therefore 0.1. This is close to the probability of .167 which we calculated using theory, but it does not match it exactly. Lets see what happens if the number of rolls is increased to 100:

```
In> dieRollsList := RandomIntegerVector(100,1,6)
Result:
{4,2,4,1,2,1,3,6,4,3,1,1,5,6,2,5,1,4,1,1,1,2,6,3,3,2,5,5,4,6,2,6,5,6,1,2,5,
3,4,2,6,3,4,2,4,1,2,6,3,1,1,4,2,6,4,4,3,1,2,5,3,1,6,6,1,6,6,4,2,1,5,4,2,3,2
,5,6,3,6,4,1,1,6,5,3,4,5,1,2,1,5,6,4,6,2,5,2,6,4,6}

In> numberOfFives := Count(dieRollsList,5)
Result: 13

In> N(numberOfFives/100)
Result: 0.13
```

This time the empirical probability of 0.13 is closer to the theoretical probability of .167. It seems that increasing the number of times the experiment is run moves the empirical probability closer to the theoretical probability. Lets see if this is true by going one step further and increasing the number of rolls to 1000:

```
In> dieRollsList := RandomIntegerVector(1000,1,6)
Result:
{2,1,3,5,5,1,5,3,5,4,4,6,6,6,6,2,6,2,6,5,4,2,1,1,4,5,5,1,2,2,2,4,1,5,1,6,5,
4,4,6,3,5,1,6,6,3,3,5,2,2,2,2,6,5,3,4,6,5,3,5,2,2,3,6,5,5,5,2,6,6,3,4,6,6,1
,3,4,5,6,4,5,1,3,1,2,3,1,5,2,2,5,4,2,6,2,2,6,3,3,3,3,6,2,6,3,1,5,6,2,1,6,3,
4,3,1,6,2,4,3,6,4,6,5,5,6,1,5,5,4,3,4,4,3,6,2,4,1,6,3,4,6,4,6,1,6,6,3,2,6,5
,6,6,4,3,1,2,1,6,4,4,1,5,4,4,3,6,1,2,6,3,1,4,3,5,6,1,6,2,2,3,2,2,4,4,6,6,2,
5,1,3,3,2,1,3,2,6,1,4,2,6,3,2,5,3,4,1,1,2,4,3,5,3,6,6,3,5,5,5,1,3,3,6,5,6,3
,5,1,2,6,5,2,2,2,4,5,3,6,5,2,4,1,1,5,4,5,1,2,1,4,1,1,5,2,4,5,1,3,3,2,2,2,2,
6,3,6,2,6,2,4,4,6,1,4,2,1,6,5,4,4,5,1,1,1,3,4,5,3,1,5,6,6,5,1,1,4,6,2,5,6,2
,5,1,5,4,1,6,6,2,3,1,6,4,2,3,4,6,6,6,4,4,1,4,6,4,1,3,1,2,3,4,5,1,3,6,5,1,4,
1,6,2,4,4,3,5,4,2,1,5,2,3,1,1,5,4,6,5,1,1,5,2,6,4,4,4,3,4,1,6,2,6,4,1,3,5,1
,4,5,2,6,5,5,6,2,3,2,2,3,1,4,4,2,3,5,5,4,2,1,3,4,1,6,4,5,4,2,4,5,6,5,6,3,5,
```

```
5,5,2,6,3,2,6,4,6,5,4,4,1,4,5,1,6,5,1,6,4,2,2,6,4,1,1,3,5,1,2,4,2,4,3,5,3,6
,3,4,1,1,1,3,2,2,1,6,3,5,1,5,3,2,3,1,5,2,4,6,4,1,3,6,1,3,5,4,4,1,6,1,1,6,4,
5,1,3,2,4,6,2,5,6,1,6,6,3,6,2,3,2,1,6,2,1,5,4,4,2,6,1,2,2,4,4,4,5,4,3,4,1,6
,3,3,5,3,6,3,2,4,3,5,5,3,2,2,5,1,2,5,6,1,6,5,4,2,3,4,1,2,1,3,4,3,3,5,1,6,2,
3,5,1,4,4,5,2,1,4,1,6,1,3,4,6,6,1,1,3,2,1,1,5,6,6,1,3,2,5,1,5,6,3,2,2,4,3,1
,3,2,2,3,6,2,6,1,5,3,2,4,1,3,6,5,2,6,5,5,5,2,6,4,1,2,2,2,1,2,5,1,1,6,1,5,3,
3,2,5,6,1,3,2,3,5,3,1,6,3,6,1,6,4,4,2,6,2,6,3,1,3,2,5,5,5,4,3,4,6,1,6,4,5,2
,5,2,5,6,4,3,4,5,6,2,5,1,5,5,1,2,1,6,3,5,5,2,4,2,6,4,6,6,5,2,4,4,5,1,3,2,2,
5,1,5,1,6,1,2,1,4,2,5,6,4,5,2,5,4,6,3,3,1,3,3,4,1,5,5,5,3,2,4,5,2,2,1,5,2,1
,6,5,1,6,5,5,2,1,4,6,3,2,1,3,1,5,5,5,2,1,1,3,3,5,2,3,3,5,4,3,3,2,6,1,1,4,4,
3,1,3,3,6,3,6,2,4,4,3,5,5,2,1,3,5,1,2,1,4,2,6,6,4,5,2,3,6,1,6,6,3,3,6,6,6,3
,4,3,2,2,4,4,2,2,6,1,4,4,1,1,6,6,4,1,1,2,5,6,3,2,2,4,5,6,5,4,1,1,1,1,2,6,5,
4,5,1,6,3,4,1,2,5,6,6,3,1,4,2,3,3,3,6,2,4,3,6,5,2,3,1,3,4,2,5,3,4,6,3,3,1,1
,4,1,3,3,1,3,3,4,6,2,4,3,4,3,6,3,4,5,5,1,1,5,2,1,2,2,3,2,1,6,2,6,1,4,4,1,4,
1,6,6,1,4,3,3,3,5,1,6,5,5,6,3,5,1,5,4,1,5,4,6,3,6,3,1,6,1,1,4,6,5,4,5,2,6,3
,6,5,6,6,5,3,1,6,5,4,6,6,3,4,5,6,4,3,1,1,6,4,6,6,6}

In> numberOfFives := Count(dieRollsList,5)
Result: 164

In> N(numberOfFives/1000)
Result: 0.164
```

This time the empirical probability of 0.164 is very close to the theoretical probability of .167. There is a law called the **law of large numbers** which states that the **greater the number of times an experiment is run, the closer the empirical probability of the process (or simulation) will become to the analogous theoretical probability**. If we wanted to increase the number of rolls in our simulation beyond 1000 to see if this is true, it would be best to use a program in a fold instead of the MathPiper console so that the the rolls are not displayed. Here is a program that rolls a simulated die 10,000 times:

```
%mathpiper,title=""

totalNumberOfRolls := 10000;

dieRollsList := RandomIntegerVector(totalNumberOfRolls,1,6);

numberOfFives := Count(dieRollsList,5);

N(numberOfFives/totalNumberOfRolls);

%/mathpiper

    %output,preserve="false"
      Result: 0.1696
.   %/output
```

And this program rolls the simulated die 100,000 times:

```
%mathpiper,title=""

totalNumberOfRolls := 100000;

dieRollsList := RandomIntegerVector(totalNumberOfRolls,1,6);

numberOfFives := Count(dieRollsList,5);

N(numberOfFives/totalNumberOfRolls);

%/mathpiper

    %output,preserve="false"
      Result: 0.16691
.   %/output
```

The law of large numbers does indeed appear to be true!

6.2.4 Histograms

The numeric results that simulations usually produce are very useful, but these raw numbers are often difficult for humans to interpret. Thankfully, graphic tools exist which enable numeric information to be viewed graphically and one of these tools is the **histogram**. Here is a description for what a histogram is is:

> In statistics, a **histogram** is a graphical display of tabulated frequencies, shown as bars. It shows what proportion of cases fall into each of several categories: it is a form of data binning. The categories are usually specified as non-overlapping intervals of some variable. The categories (bars) must be adjacent. The intervals (or bands, or bins) are generally of the same size. (http://en.wikipedia.org/wiki/Histogram).

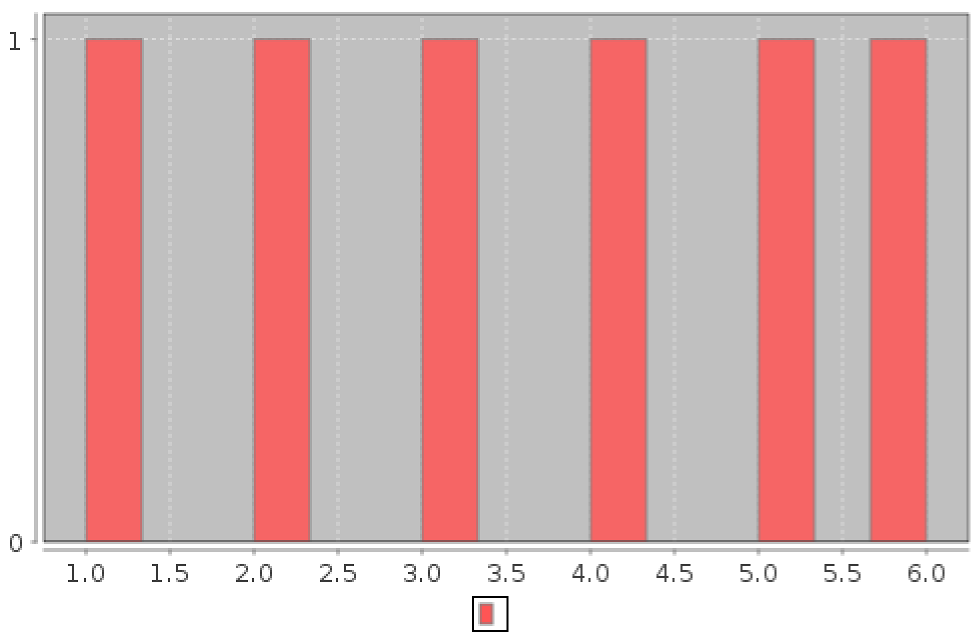
MathPiper has a function called **Histogram()** and it is capable of converting numerical data into a graphic histogram and then displaying it in the JFreeChart plugin. In this book graphics which display summaries of numeric data will be referred to as **charts**. The following sections show how to use the Histogram() function.

6.2.4.1 Plain Histogram With No Title

The Histogram() function accepts one or more arguments and the first argument is always a list which contains numbers. These numbers are counted by the Histogram() function and then placed into **bins**. The bins are represented by rectangular **bars**. The following code shows the numbers 1,2,3,4,5,6 being passed to Histogram():

```
%mathpiper,title="Plain Histogram No Title"

Histogram({1,2,3,4,5,6});

%/mathpiper

    %output,preserve="false"
      Result: org.jfree.chart.ChartPanel
.   %/output
```

And here is the chart that is produced:



This is a plain histogram without a title and it shows that each of the numbers 1-6 occurred in the list 1 time. Unfortunately, it is difficult to determine what information the histogram is trying to relay because it does not contain any explanatory text. Also, the bars are not uniformly placed but we will take care of that in a moment. The first improvement that will be made to this chart is to add a title.

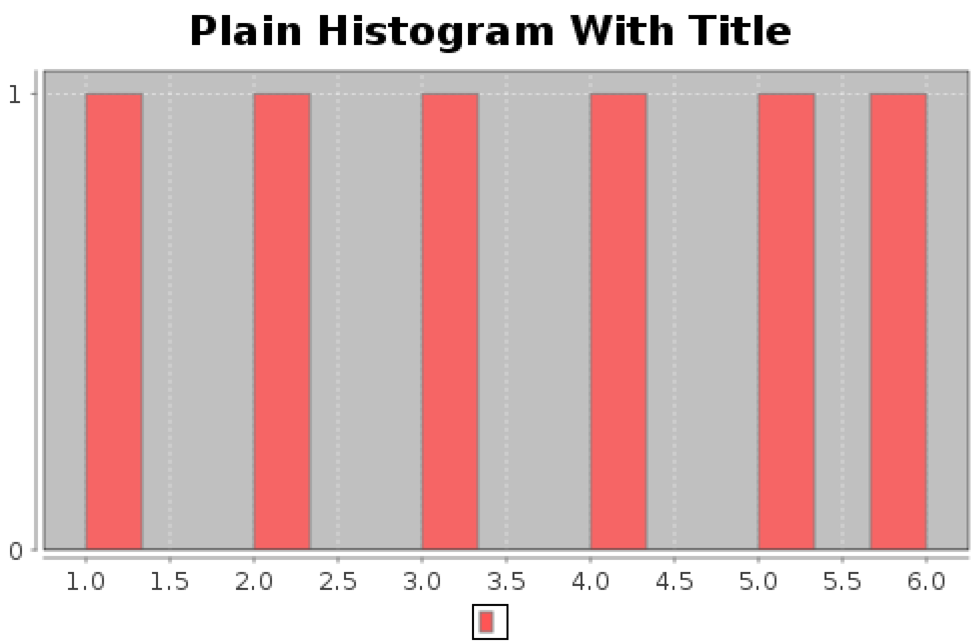
6.2.4.2 Adding A Title To A Histogram And The Options Operator (->)

A title can be added to a histogram by passing a **title option** to the **Histogram()** function as the following code shows:

```
%mathpiper,title="Plain Histogram With Title"

Histogram({1,2,3,4,5,6}, title -> "Plain Histogram With Title");
```

```
%/mathpiper

    %output,preserve="false"
      Result: org.jfree.chart.ChartPanel
.   %/output
```

Options are passed to functions by using the **->** operator. The **->** operator is an infix operator which means that its **first argument** is to its immediate **left** and the **second argument** is to its immediate **right**. The operator's **first argument is the name of the option** that will be set and the **second argument is the value that the option will be set to**. In this example, the **title** option is being set to the string value "**Plain Histogram With Title**" and here is the result:



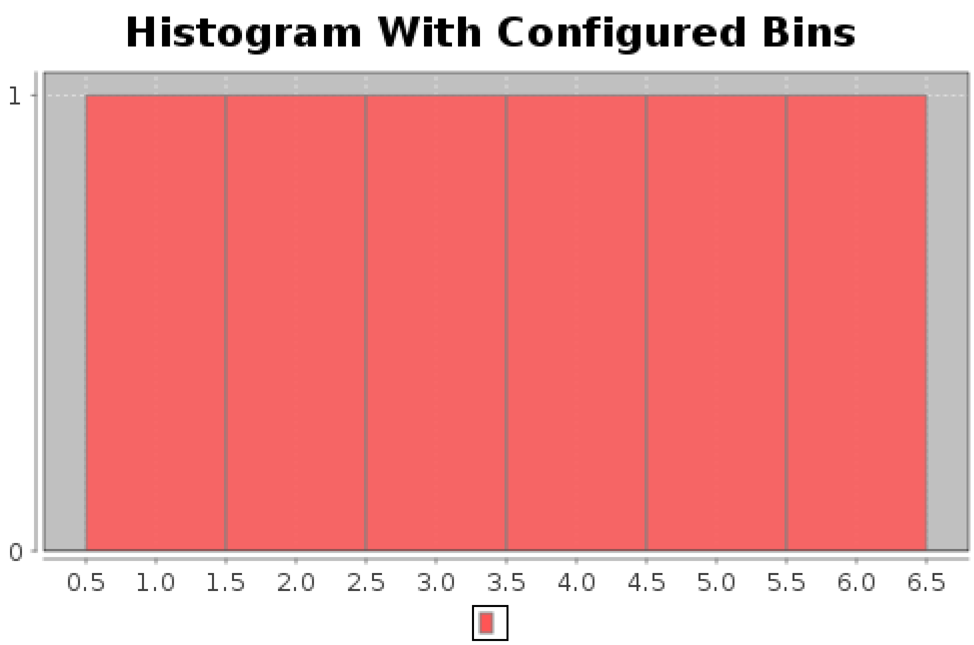
The addition of a title for the histogram is an improvement, but the bars are still not uniform so we will fix them next.

6.2.4.3 Histogram With Configured Bins

The **binMinimum**, **binMaximum**, and **numberOfBins** options are used to configure the Histogram() function's bins. In the following code, **binMinimum** is set to .5 which indicates that the left side of the leftmost bar will be at .5 on the X axis. Next, binMaximum is set to 6.5 which indicates that the right side of the rightmost bar will be at 6.5 on the X axis. Finally, numberOfBins is set to 6 since we want 6 bins to match the 6 different number values that can be placed into the list.

```
%mathpiper,title="Histogram With Configured Bins"

Histogram({1,2,3,4,5,6}, title -> "Histogram With Configured Bins", binMinimum -
```

```
> .5, binMaximum -> 6.5, numberOfBins -> 6,);

%/mathpiper

    %output,preserve="false"
      Result: org.jfree.chart.ChartPanel
.   %/output
```

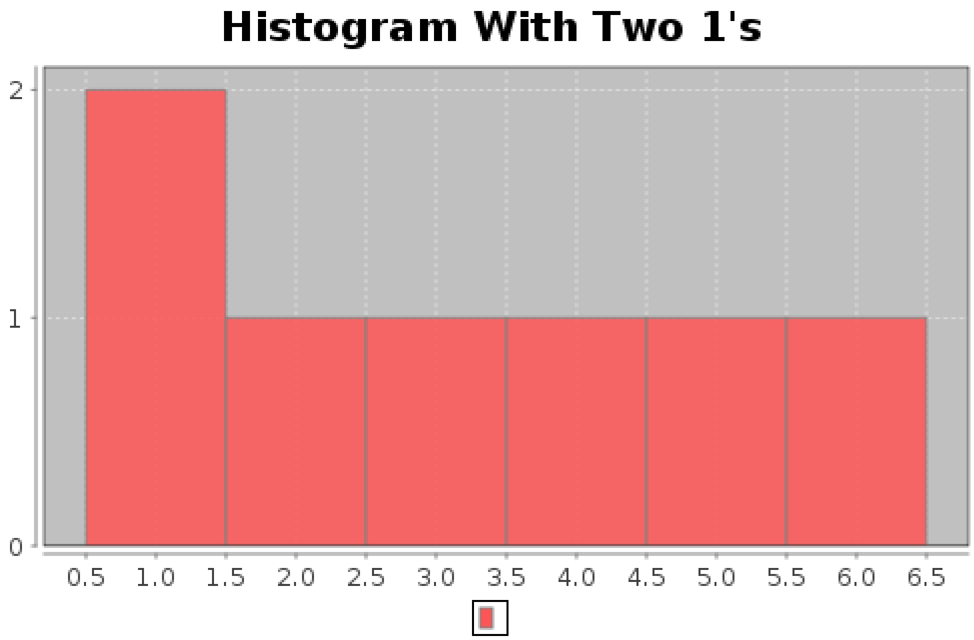


The bins in the chart which this code displays are now uniformly placed and centered on the number they represent. For example, the first bin is centered on 1.0, the second bin is centered on 2.0, and so on. However, it is still not completely clear what information this histogram is relaying and therefore in the next section we will see what happens when a seventh number is added to the list.

6.2.4.4 Histogram With Two 1's

In this example, an additional 1 has been placed into the list:

```
%mathpiper,title="Histogram With Two 1's"

Histogram({1,1,2,3,4,5,6}, title -> "Histogram With Two 1's", binMinimum -> .5,
binMaximum -> 6.5, numberOfBins -> 6,);

%/mathpiper

    %output,preserve="false"
      Result: org.jfree.chart.ChartPanel
.   %/output
```

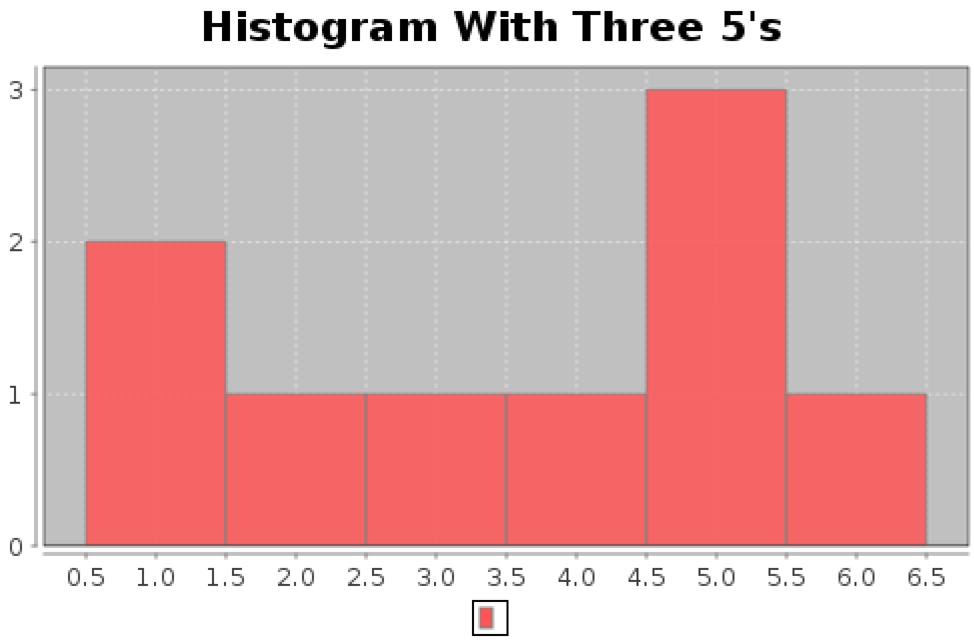


Notice that the height of the 1.0 bin is now set to 2 which indicates the list has two 1's in it. All the other bins have a height of 1 which indicate that there are only one of each of these numbers in the list. In the next section, two additional 5's added to the list

6.2.4.5 Histogram With Three 5's

The list in the code in this section has had two addition 5's added to it for a total of three 5's:

```
%mathpiper,title="Histogram With Three 5's"

Histogram({1,1,2,3,4,5,5,5,6}, title -> "Histogram With Three 5's", binMinimum -
> .5, binMaximum -> 6.5, numberOfBins -> 6,);

%/mathpiper

    %output,preserve="false"
      Result: org.jfree.chart.ChartPanel
.   %/output
```

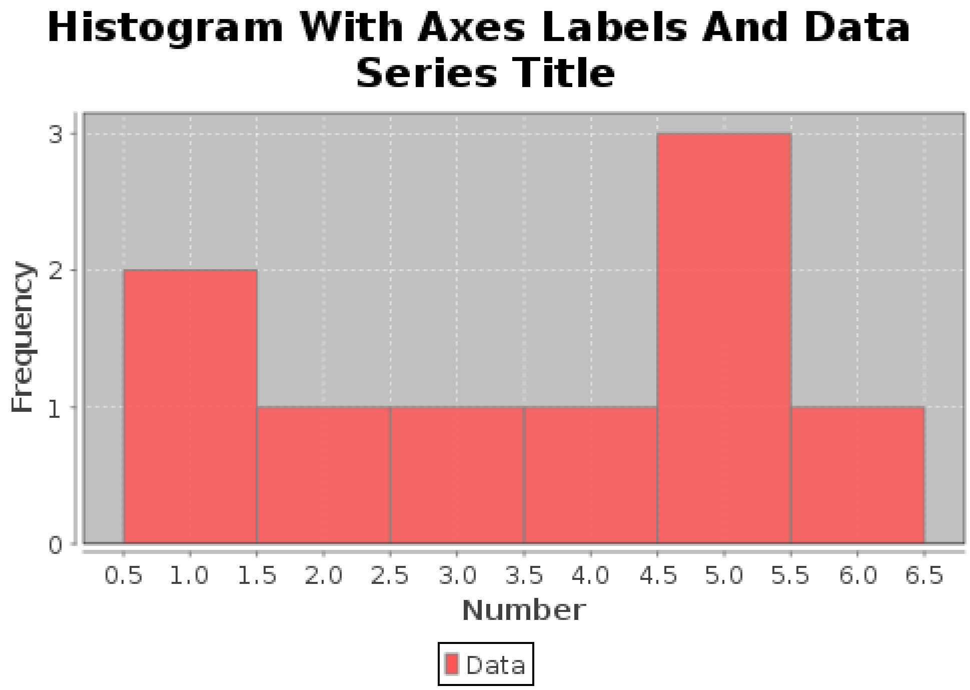


By now you should be able to see that the hight of a histogram's bars indicates how **frequently** each number occurs in the list that is passed to it. If you now go back an reread the description for a histogram, it should make more sense to you. This histogram is reasonably informative, but it can be improved by adding labels to the X and Y axes and also a title for the data. This is done in the next section.

6.2.4.6 Histogram With Axes Labels And Data Series Titles

The code in this section uses the **xAxisLabel**, **yAxisLabel**, and **seriesTitle** options to add additional explanatory text to the histogram:

```
%mathpiper,title="Histogram With Axes Labels And Data Series Title"

Histogram({1,1,2,3,4,5,5,5,6}, title -> "Histogram With Axes Labels And Data Series
Title", binMinimum -> .5, binMaximum -> 6.5, numberOfBins -> 6, xAxisLabel ->
"Number", yAxisLabel -> "Frequency", seriesTitle ->"Data");

%/mathpiper

    %output,preserve="false"
      Result: org.jfree.chart.ChartPanel
.   %/output
```

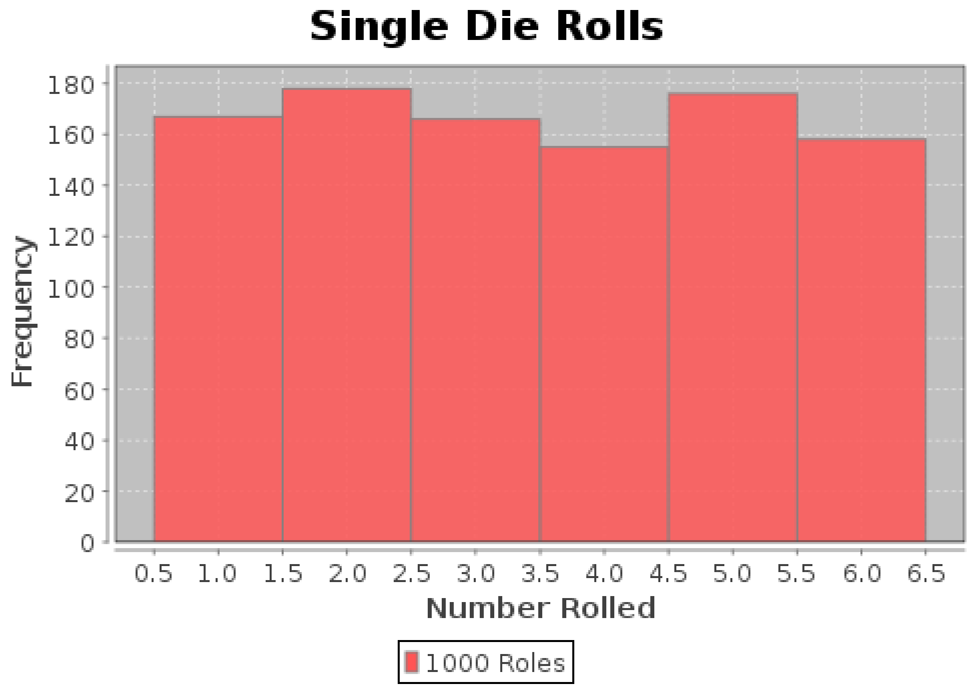


The X axis has been labeled "**Number**" because it represents the various numbers in the given list and the Y axis has been labeled "**Frequency**" because it indicates how frequently each number in the list occurs. The series of numbers in the list has been given the simple title "**Data**" to show how the data can be labeled. The Histogram() function has more capabilities than what has been show here, but the capabilities that have been covered are sufficient for displaying the results of simple simulations.

6.2.5 A Histogram Which Shows The Result Of Rolling A Single Simulated Die 1000 Times

Now that you know how to use the Histogram() function, lets use it to display the data from a simulation which rolls a single die 1000 times:

```
%mathpiper,title="Rolls Of A Single Die"

numberOfRolls := 1000;

dieRollsList := RandomIntegerVector(numberOfRolls,1,6);

Histogram(dieRollsList, binMinimum -> .5, binMaximum -> 6.5, numberOfBins -> 6,
title -> "Single Die Rolls", xAxisLabel -> "Number Rolled", yAxisLabel ->
"Frequency", seriesTitle -> String(numberOfRolls) : " Rolls");

%/mathpiper

    %output,preserve="false"
      Result: org.jfree.chart.ChartPanel
.   %/output
```



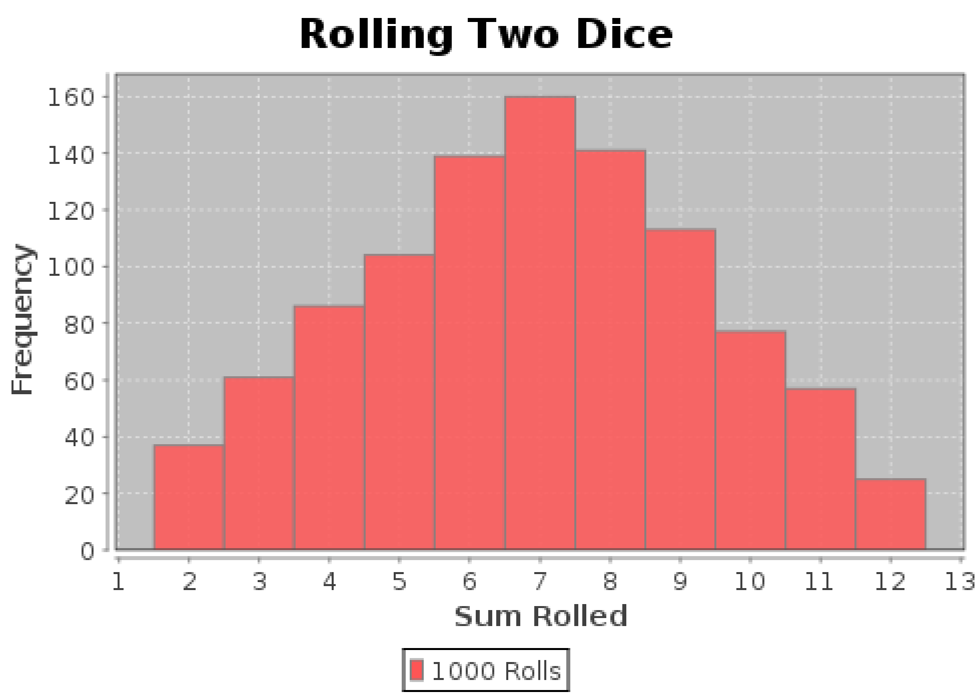
This histogram shows that each number on the simulated die has about the same probability of landing face up as the other numbers on the die. The kind of probability that a simulation like this produces is empirical probability. As discussed earlier, if the number of rolls in the simulation were increased, the simulation's observed empirical probability would become closer to the theoretical probability for this kind of experiment.

6.2.6 A Histogram Which Shows The Result Of Rolling Two Simulated Dice 1000 Times

The program in this section simulates the rolling of two dice 1000 times. Each time the two dice are rolled their sum is calculated and then appended to the list which is bound to the variable dieRollsList. After the simulation is complete, a histogram of the sum data which was accumulated in dieRollsList is displayed:

```
%mathpiper,title="Rolls Two Dice"

numberOfRolls := 1000;

dieRollsList := {};

Repeat(numberOfRoles)
[
   die1 := RandomInteger(6);

```

```
    die2 := RandomInteger(6);

    dieRollsList := Append(dieRollsList, die1 + die2);
];

Histogram(dieRollsList,
          binMinimum -> 1.5,
          binMaximum -> 12.5,
          numberOfBins -> 11,
          title -> "Rolling Two Dice",
          xAxisLabel -> "Sum Rolled",
          yAxisLabel -> "Frequency",
          seriesTitle -> String(numberOfRolls) : " Rolls"
          );

%/mathpiper

    %output,preserve="false"
      Result: org.jfree.chart.ChartPanel
.   %/output
```



The results of the simulation show that sums in the middle of the range between 2 and 12 inclusive occur more often than the sums near either end of the range.

6.3 Exercises

For the following exercises, create a new MathRider worksheet file called **book_2_section_6_exercises_<your first name>_<your last name>.mrw**. (**Note: there are no spaces in this file name**). For example, John Smith's worksheet would be called:

book_2_section_6_exercises_john_smith.mrw.

After this worksheet has been created, place your answer for each exercise that requires a fold into its own fold in this worksheet. Place a title attribute in the start tag of each fold which indicates the exercise the fold contains the solution to. The folds you create should look similar to this one:

```
%mathpiper,title="Exercise 1"

//Sample fold.

%/mathpiper
```

If an exercise uses the MathPiper console instead of a fold, copy the work you did in the console into the worksheet so it can be saved.

6.3.1 Exercise 1

7 Monte Carlo Resampling Simulations

7.1 The Three Doors Game

Suppose you are on a TV game show where the host shows you three closed doors and then tells you that behind one of the doors is a prize and the area behind the other two doors is empty. The host then tells to you pick which door you think the prize is behind and if your pick is correct, you can have the prize. You pick a door, but the host does not open it yet. At least one of the doors you didn't pick is empty and the host (who knows which door leads to the prize) **opens one of the remaining doors which leads to an empty area**.

Now comes the fun part! There are two unopened doors remaining (the one you picked and one which you did not pick) and the host then asks you if you would like to switch your pick to the second door. What should you do? Do you have a higher chance of winning if you stay with the original door you picked or if you switch to the other door? Or do you have the same change of winning regardless of which door you pick?

This is a very difficult problem to solve correctly by reasoning through it, but it is easy to solve with a simulation. Think about this problem for a little bit, record somewhere what you think the solution is, and then run the following simulation to see if you are correct:

```
%mathpiper,title="Three doors simulation."

numberOfTrials := 1000;

firstPickList := {};

secondPickList := {};

Repeat(numberOfTrials)
[
    doorsList := Shuffle({EMPTY, PRIZE, EMPTY});

    //The contestant always picks door 1 as their first pick.
    firstPick := doorsList[1];

    /*
      If door 2 is empty, the second pick is the 3rd door else
      the second pick is the 2nd door.  In a real game the contestant
      may or may not make a second pick, but in this simulation
      we always make the second pick in order to see what it is.
    */
    If(doorsList[2] = EMPTY,
        secondPick := doorsList[3],
        secondPick := doorsList[2]);
```

```
    //Save all of the first pick results in a list.
    firstPickList := Append(firstPickList, firstPick);

    //Save all of the second pick results in a list.
    secondPickList := Append(secondPickList, secondPick);
];

firstPickWins := Count(firstPickList,PRIZE);

secondPickWins := Count(secondPickList,PRIZE);


Echo("The number of times the first door picked contained a prize: ",
    firstPickWins, "/ ", numberOfTrials);
Echo("The probability of winning for always staying with the first pick: ",
    N(firstPickWins/numberOfTrials) );
NewLine();

Echo("The number of times the second door picked contained a prize: ",
    secondPickWins, "/ ", numberOfTrials);
Echo("The probability of winning for always changing the pick: ",
    N(secondPickWins/numberOfTrials) );


%/mathpiper
```

7.2 In A Room Full Of People, What Is The Probability That At Least Two Will Have The Same Birthday?

```
%mathpiper,title="Same birthday simulation."

birthdayMatchCounter := 0;

numberOfPeople := 20;

numberOfTrials := 40;

Repeat(numberOfTrials)
[
    //Create a random birthday for each simulated person in the room.
    birthdaysList := RandomIntegerVector(numberOfPeople, 1, 365);

    //Print the birthdays for this room of simulated people (comment
    //this line out for a large number of trials).
      Write(birthdaysList);

    /*
     Index through all of the days in a year and for each day scan
     the birthdays list to see if two people in the list have this
     day as their birthday.  If there is a match, increment
     birthdayMathCounter.
    */
```

```
    ForEach(day, 1 .. 365)
    [
        If(Count(birthdaysList,day) >= 2,
            [
                birthdayMatchCounter++;
                WriteString(" - Match on ");
                Write(day); Break();
            ]);
    ];

    NewLine();

];

    NewLine();
    Echo("The number of trials is: ", numberOfTrials);
    Echo("The number of people in the room is: ", numberOfPeople);
    Echo("The number of matches is: ",
        birthdayMatchCounter, "/ ", numberOfTrials);
    Echo("The probability of having a birthday match: ",
        N(birthdayMatchCounter/numberOfTrials) );

%/mathpiper

    %output,preserve="false"
      Result: True

      Side Effects:

      {323,362,99,208,102,237,116,199,25,174,23,160,312,351,72,205,40,114,110,212}
      {166,313,124,334,75,187,102,348,64,363,106,78,238,187,360,245,177,194,34,163} - Match on 187
      {79,148,15,5,81,173,315,50,272,140,319,262,305,95,209,269,14,170,55,179}
      {320,219,45,102,350,163,166,286,271,201,234,99,295,39,77,302,243,314,10,308}
      {343,251,320,322,351,204,281,57,228,213,134,286,187,285,2,13,62,116,266,258}
      {189,315,293,322,105,274,28,69,175,244,269,297,260,192,281,146,166,70,250,24}
      {22,295,307,248,194,135,322,234,350,80,249,108,214,69,85,75,148,22,352,172} - Match on 22
      {98,4,310,87,142,282,301,242,90,58,319,156,275,238,109,30,253,276,192,118}
      {115,290,360,171,14,297,147,86,84,120,365,39,347,204,15,137,227,313,47,42}
      {213,10,262,96,156,67,101,348,27,328,186,135,195,95,50,75,101,212,216,8} - Match on 101
      {51,149,48,218,275,200,289,197,132,283,211,101,9,336,146,238,43,15,46,120}
      {22,113,82,46,86,143,160,203,5,328,325,247,300,261,43,183,80,284,145,42}
      {321,28,323,262,161,194,336,272,62,152,5,107,127,27,320,2,124,190,2,228} - Match on 2
      {101,269,77,139,103,83,191,161,75,222,354,198,276,323,208,29,4,39,34,272}
      {89,27,333,1,347,302,173,176,32,202,281,211,148,230,97,137,306,177,269,37}
      {321,352,156,120,323,322,355,22,87,169,197,339,114,121,224,246,70,285,310,34}
      {10,347,172,238,19,245,214,349,43,216,347,83,263,120,185,14,174,113,86,156} - Match on 347
      {337,357,255,229,241,28,261,214,258,254,235,159,8,268,242,52,67,189,190,180}
      {22,77,62,232,190,275,274,111,16,355,118,365,69,214,298,86,323,275,231,29} - Match on 275
      {162,331,112,63,91,181,261,146,26,142,80,151,335,209,79,70,141,309,191,299}
      {155,89,3,246,172,290,351,78,327,136,216,204,243,110,38,11,70,140,293,310}
      {112,27,238,75,95,170,264,305,268,286,362,124,277,18,319,144,187,242,1,30}
      {334,185,230,180,345,123,42,289,239,39,336,354,290,332,219,154,352,90,49,206}
      {107,109,51,248,184,324,117,64,136,299,230,155,174,105,220,148,61,11,83,112}
      {274,345,318,112,272,295,239,247,80,302,258,158,38,252,280,264,156,296,203,35}
      {154,165,125,324,156,150,61,324,195,351,53,178,329,268,100,171,251,88,82,236} - Match on 324
      {194,251,186,208,192,299,175,151,224,161,221,106,125,115,165,64,339,52,294,219}
      {202,99,140,114,309,319,202,44,168,86,193,27,261,316,156,58,83,40,256,275} - Match on 202
      {167,349,111,351,48,263,29,24,164,108,292,92,19,242,98,296,233,119,349,365} - Match on 349
      {79,143,217,277,328,354,56,163,67,200,242,296,242,37,109,135,35,198,61,141} - Match on 242
      {284,96,94,228,60,149,344,31,232,225,195,328,153,317,97,207,191,58,71,270}
      {251,248,151,262,90,67,18,309,312,82,169,14,4,360,32,205,124,122,131,108}
      {189,304,56,245,308,139,73,155,124,61,318,153,127,226,296,243,199,25,304,282} - Match on 304
      {126,279,193,356,204,355,7,351,123,303,102,245,98,227,181,88,213,78,316,118}
      {153,102,115,333,29,14,192,333,74,70,266,221,295,266,270,81,51,110,215,291} - Match on 266
```

```
    {86,312,104,180,70,73,18,214,14,78,198,114,50,129,173,262,41,63,15,282}
    {62,199,234,311,294,88,102,314,288,219,55,34,151,337,279,41,328,124,43,276}
    {336,288,118,186,67,292,37,146,361,299,240,229,264,249,100,34,76,363,55,13}
    {314,24,315,343,211,42,335,343,115,139,301,312,95,331,41,33,206,220,281,211} - Match on 211
    {207,125,175,336,160,265,57,352,71,86,61,349,2,169,258,181,268,27,279,165}

    The number of trials is: 40
    The number of people in the room is: 20
    The number of matches is: 13 / 40
    The probability of having a birthday match: 0.325
.   %/output
```

7.3 What Is The Probability Of A Family With Four Children Having Three Boys?

```
%mathpiper,title="Three boys simulation"

numberOfTrials := 40;

numberOfChildren := 4;

numberOfBoys := 3;

threeBoysCounter := 0;

girlProbability := 49/100;

boyProbability := 51/100;

Repeat(numberOfTrials)
[
    //Create a random list of simulated children.
    childrenList := RandomSymbolVector(
        {{GIRL,girlProbability}, {BOY,boyProbability}},
        numberOfChildren);

    //Print the list of simulated children (comment
    //this line out for a large number of trials).
    Write(childrenList);

    /*
      If the list contains 3 boys, increment the three
      boys counter.
    */
    If(Count(childrenList, BOY) = numberOfBoys,
        [
            threeBoysCounter++;
            WriteString(" - ");
            Write(numberOfBoys);
```

```
            WriteString(" boys.");
        ]);

    NewLine();
];

NewLine();
Echo("The number of trials is: ", numberOfTrials);
Echo("The number of children is: ", numberOfChildren);
Echo("The number of trials which have 3 boys: ",
    threeBoysCounter, "/ ", numberOfTrials);
Echo("The probability of having 3 boys: ",
    N(threeBoysCounter/numberOfTrials) );

%/mathpiper

    %output,preserve="false"
      Result: True

      Side Effects:
      {GIRL,GIRL,BOY,GIRL}
      {GIRL,BOY,GIRL,GIRL}
      {GIRL,GIRL,BOY,GIRL}
      {GIRL,GIRL,BOY,BOY}
      {GIRL,BOY,BOY,BOY} - 3 boys.
      {GIRL,GIRL,BOY,GIRL}
      {BOY,BOY,BOY,BOY}
      {GIRL,BOY,GIRL,GIRL}
      {GIRL,BOY,GIRL,BOY}
      {GIRL,BOY,BOY,GIRL}
      {BOY,BOY,BOY,GIRL} - 3 boys.
      {GIRL,GIRL,GIRL,BOY}
      {BOY,BOY,GIRL,BOY} - 3 boys.
      {GIRL,GIRL,BOY,BOY}
      {GIRL,BOY,BOY,BOY} - 3 boys.
      {GIRL,GIRL,BOY,BOY}
      {GIRL,GIRL,GIRL,BOY}
      {BOY,GIRL,BOY,BOY} - 3 boys.
      {GIRL,GIRL,BOY,GIRL}
      {GIRL,BOY,BOY,BOY} - 3 boys.
      {BOY,GIRL,BOY,GIRL}
      {GIRL,GIRL,GIRL,BOY}
      {BOY,GIRL,GIRL,GIRL}
      {GIRL,BOY,BOY,GIRL}
      {GIRL,BOY,GIRL,GIRL}
      {GIRL,GIRL,BOY,BOY}
      {BOY,GIRL,GIRL,BOY}
      {BOY,BOY,GIRL,GIRL}
      {BOY,BOY,BOY,GIRL} - 3 boys.
      {GIRL,GIRL,BOY,GIRL}
      {GIRL,BOY,GIRL,GIRL}
      {GIRL,GIRL,BOY,BOY}
      {BOY,BOY,BOY,GIRL} - 3 boys.
      {GIRL,BOY,GIRL,BOY}
      {BOY,BOY,GIRL,GIRL}
      {BOY,BOY,BOY,GIRL} - 3 boys.
      {GIRL,GIRL,BOY,GIRL}
```

```
        {GIRL,BOY,GIRL,GIRL}
        {BOY,GIRL,BOY,GIRL}
        {GIRL,GIRL,BOY,BOY}

        The number of trials is: 40
        The number of children is: 4
        The number of trials which have 3 boys: 9 / 40
        The probability of having 3 boys: 0.225
.   %/output
```

7.4 What Is The Probability Of Having Three Or More Hits In 5 Basketball Free throws?

```
%mathpiper,title="Five basketball free throws."

numberOfTrials := 40;

//A success is defined as 3 or more hits.
successesCounter := 0;

numberOfThrows := 5;

hitProbability := .30;

missProbability := .70;

Repeat(numberOfTrials)
[
    sampleList := RandomSymbolVector(
        {{HIT,hitProbability}, {MISS,missProbability}},
        numberOfThrows);

    //Print the list of simulated throws (comment
    //this line out for a large number of trials).
    Write(sampleList);

    /*
      If the list contains 3 or more hits, increment the
      success counter.
    */
    If(Count(sampleList,HIT) >= 3,
        [
            successesCounter++;
            WriteString(" - ");
            WriteString(" success.");
        ]);

    NewLine();
];

NewLine();
Echo("The number of trials is: ", numberOfTrials);
```

```
Echo("The number of throws is: ", numberOfThrows);
Echo("The number of trials which have 3 or more hits: ",
    successesCounter, "/ ", numberOfTrials);
Echo("The probability of having 3 or more hits: ",
    N(successesCounter/numberOfTrials) );

%/mathpiper

    %output,preserve="false"
      Result: True

      Side Effects:
      {HIT,MISS,HIT,HIT,HIT} -  success.
      {HIT,MISS,MISS,HIT,MISS}
      {MISS,MISS,MISS,MISS,HIT}
      {MISS,MISS,MISS,HIT,HIT}
      {MISS,HIT,MISS,MISS,HIT}
      {MISS,MISS,MISS,HIT,MISS}
      {MISS,HIT,MISS,HIT,MISS}
      {MISS,HIT,HIT,MISS,MISS}
      {MISS,MISS,MISS,MISS,MISS}
      {MISS,HIT,MISS,MISS,MISS}
      {HIT,MISS,MISS,MISS,MISS}
      {HIT,HIT,HIT,HIT,MISS} -  success.
      {MISS,MISS,MISS,MISS,HIT}
      {MISS,MISS,MISS,MISS,MISS}
      {MISS,MISS,MISS,MISS,HIT}
      {MISS,MISS,MISS,MISS,HIT}
      {MISS,MISS,MISS,MISS,MISS}
      {HIT,MISS,MISS,HIT,MISS}
      {MISS,MISS,MISS,HIT,MISS}
      {MISS,MISS,MISS,MISS,HIT}
      {MISS,HIT,MISS,HIT,MISS}
      {MISS,HIT,MISS,MISS,MISS}
      {HIT,MISS,MISS,HIT,HIT} -  success.
      {MISS,HIT,MISS,MISS,MISS}
      {MISS,HIT,MISS,MISS,MISS}
      {MISS,MISS,MISS,MISS,MISS}
      {MISS,MISS,MISS,MISS,HIT}
      {MISS,HIT,MISS,HIT,HIT} -  success.
      {HIT,MISS,MISS,MISS,HIT}
      {MISS,HIT,MISS,MISS,MISS}
      {HIT,MISS,MISS,HIT,HIT} -  success.
      {MISS,HIT,MISS,MISS,HIT}
      {HIT,MISS,HIT,MISS,MISS}
      {HIT,MISS,MISS,MISS,MISS}
      {MISS,MISS,MISS,HIT,HIT}
      {HIT,HIT,MISS,HIT,MISS} -  success.
      {MISS,HIT,MISS,MISS,HIT}
      {MISS,MISS,HIT,MISS,MISS}
      {MISS,MISS,MISS,MISS,MISS}
      {MISS,MISS,MISS,MISS,HIT}

      The number of trials is: 40
      The number of throws is: 5
      The number of trials which have 3 or more hits: 6 / 40
      The probability of having 3 or more hits: 0.15
```

```
.   %/output
```

7.5 Shooting At A Target

```
%mathpiper,title="Shooting at a target."

numberOfTrials := 40;

numberOfShots := 5;

successCounter := 0;

Repeat(numberOfTrials)
[
    sampleList := RandomSymbolVector(
        {{BLACK,15/100}, {WHITE,55/100}, {MISS,30/100}},
        numberOfShots);

    //Print the list of simulated throws (comment
    //this line out for a large number of trials).
    Write(sampleList);


    /*
      If the list contains 1 hit in the black and 3 hits
      in the white, increment the successCounter.
    */
    If(Count(sampleList,BLACK)= 1 And Count(sampleList,WHITE) = 3,
        [
            successCounter++;
            WriteString(" - ");
            WriteString(" success.");
        ]);

    NewLine();

];

NewLine();
Echo("The number of trials is: ", numberOfTrials);
Echo("The number of shots per trial is: ", numberOfShots);
Echo("The number of trials which have 1 black hit and 3 white hits: ",
    successCounter, "/ ", numberOfTrials);
Echo("The probability of having 1 black hit and 3 white hits: ",
    N(successCounter/numberOfTrials) );

%/mathpiper

    %output,preserve="false"
      Result: True

      Side Effects:
```

```
    {WHITE,BLACK,WHITE,MISS,BLACK}
    {WHITE,BLACK,WHITE,WHITE,MISS} -  success.
    {MISS,WHITE,MISS,WHITE,WHITE}
    {MISS,BLACK,MISS,MISS,WHITE}
    {WHITE,MISS,MISS,MISS,WHITE}
    {WHITE,WHITE,WHITE,WHITE,WHITE}
    {WHITE,WHITE,WHITE,MISS,BLACK} -  success.
    {WHITE,WHITE,BLACK,BLACK,MISS}
    {MISS,WHITE,BLACK,BLACK,WHITE}
    {MISS,MISS,WHITE,WHITE,WHITE}
    {WHITE,MISS,MISS,MISS,WHITE}
    {BLACK,BLACK,BLACK,BLACK,WHITE}
    {MISS,MISS,WHITE,MISS,BLACK}
    {MISS,WHITE,WHITE,BLACK,WHITE} -  success.
    {MISS,MISS,MISS,WHITE,BLACK}
    {WHITE,WHITE,WHITE,WHITE,WHITE}
    {MISS,WHITE,WHITE,WHITE,MISS}
    {WHITE,WHITE,WHITE,BLACK,WHITE}
    {WHITE,WHITE,MISS,WHITE,WHITE}
    {MISS,WHITE,WHITE,BLACK,BLACK}
    {WHITE,BLACK,WHITE,WHITE,BLACK}
    {WHITE,WHITE,WHITE,MISS,WHITE}
    {WHITE,WHITE,WHITE,WHITE,MISS}
    {WHITE,WHITE,MISS,MISS,BLACK}
    {WHITE,MISS,WHITE,BLACK,WHITE} -  success.
    {WHITE,WHITE,WHITE,WHITE,MISS}
    {WHITE,MISS,BLACK,WHITE,WHITE} -  success.
    {BLACK,WHITE,WHITE,MISS,MISS}
    {BLACK,WHITE,WHITE,WHITE,WHITE}
    {MISS,BLACK,MISS,MISS,MISS}
    {MISS,WHITE,MISS,BLACK,MISS}
    {WHITE,WHITE,MISS,WHITE,MISS}
    {WHITE,MISS,WHITE,BLACK,MISS}
    {BLACK,BLACK,WHITE,MISS,WHITE}
    {WHITE,WHITE,WHITE,MISS,WHITE}
    {MISS,MISS,MISS,MISS,MISS}
    {WHITE,BLACK,MISS,MISS,WHITE}
    {MISS,MISS,WHITE,WHITE,MISS}
    {MISS,WHITE,WHITE,WHITE,WHITE}
    {MISS,WHITE,MISS,MISS,MISS}

    The number of trials is: 40
    The number of shots per trial is: 5
    The number of trials which have 1 black hit and 3 white hits: 5 / 40
    The probability of having 1 black hit and 3 white hits: 0.125
.   %/output
```

7.6 Two Stacks Of 10 Pennies

```
%mathpiper,title="Two stacks of pennies."

numberOfTrials := 20;
```

```
emptyStackCounter := 0;

Repeat(numberOfTrials)
[
    //Create two stacks of 10 pennies.
    stack1 := 10; stack2 := 10;

    maxFlips := 50;

    emptyMessage := "";

    /*
     Flip a coin (lets say it a quarter) up to maxFlips times.  If
     it comes up heads remove a penny from stack 2 and places it on
     stack 1.  If it comes up tails, remove a penny from stack 1 and
     place it on stack 2.  If either stack reaches 0 before maxFlips
     have been flipped, break out of the loop.
    */
    numberOfFlips := Repeat(maxFlips)
    [
        flip := RandomSymbol({{HEAD,1/2},{TAIL,1/2}});

        If(flip = HEAD, [stack1++; stack2--;],
            [stack1--; stack2++;] );

        If(stack1 = 0 Or stack2 = 0,
            [emptyStackCounter++; emptyMessage := " - Empty"; Break();] );
    ];

    //Print the state of the stacks after each flip (comment
    //this line out for a large number of trials).
    Echo("Stack 1: ", PadLeft(stack1,2),
        "    Stack 2: ", PadLeft(stack2,2),
        "    Number of flips: ", PadLeft(numberOfFlips,3), emptyMessage);
];

NewLine();
Echo("The number of trials is: ", numberOfTrials);
Echo("The maximum flips per trial: ", maxFlips);
Echo("The number of trials which resulted in an empty stack: ",
    emptyStackCounter, "/ ", numberOfTrials);
Echo("The probability of having an empty stack: ",
    N(emptyStackCounter/numberOfTrials) );

%/mathpiper

    %output,preserve="false"
      Result: True

      Side Effects:
      Stack 1: 14    Stack 2: 06    Number of flips: 050
      Stack 1: 08    Stack 2: 12    Number of flips: 050
      Stack 1: 20    Stack 2: 00    Number of flips: 037 - Empty
      Stack 1: 20    Stack 2: 00    Number of flips: 023 - Empty
      Stack 1: 10    Stack 2: 10    Number of flips: 050
      Stack 1: 04    Stack 2: 16    Number of flips: 050
      Stack 1: 02    Stack 2: 18    Number of flips: 050
```

```
    Stack 1: 02    Stack 2: 18    Number of flips: 050
    Stack 1: 04    Stack 2: 16    Number of flips: 050
    Stack 1: 06    Stack 2: 14    Number of flips: 050
    Stack 1: 20    Stack 2: 00    Number of flips: 035 - Empty
    Stack 1: 14    Stack 2: 06    Number of flips: 050
    Stack 1: 00    Stack 2: 20    Number of flips: 031 - Empty
    Stack 1: 12    Stack 2: 08    Number of flips: 050
    Stack 1: 08    Stack 2: 12    Number of flips: 050
    Stack 1: 00    Stack 2: 20    Number of flips: 023 - Empty
    Stack 1: 04    Stack 2: 16    Number of flips: 050
    Stack 1: 04    Stack 2: 16    Number of flips: 050
    Stack 1: 08    Stack 2: 12    Number of flips: 050
    Stack 1: 08    Stack 2: 12    Number of flips: 050

    The number of trials is: 20
    The maximum flips per trial: 50
    The number of trials which resulted in an empty stack: 5 / 20
    The probability of having an empty stack: 0.25
.   %/output
```

7.7 Vending Machine Simulation

```
%mathpiper,title="Pop vending machine."

daysToEmptyList := {};

numberOfTrials := 10;

daysBetweenRestocking := 3;

machineCansCapacity := 50;

Repeat(numberOfTrials)
[
    //Fill the machine to capicity before starting the trial.
    cansInMachine := machineCansCapacity;

    daysToEmptyCounter := 0;

    Echo("New Trial");

    //Run the simulation in an infinite loop until the machine is empty.
    Repeat()
    [
        //Simulate a 24 hour period.
        Repeat(24)
        [
            //Determine the number of cans which were perchased during this hour.
            numberOfCansPurchasedThisHour := RandomSymbol({
                {0,60/100},
                {1,20/100},
```

```
                {2,15/100},
                {3,05/100}});

            cansInMachine := cansInMachine - numberOfCansPurchasedThisHour;

            //If the machine has become empty, end the 24 hour simulation.
            If(cansInMachine <= 0, [cansInMachine := 0; Break();]);
        ];


        Echo("Day: " ,
            PadLeft(daysToEmptyCounter,2),
            "   Cans in machine: ",
            PadLeft(cansInMachine,2));


        //If the machine has become empty, end this trial.
        If(cansInMachine = 0, Break());

        /*
         Uncomment the following line to enable restocking.
        */
        /*
         If(Mod(daysToEmptyCounter,daysBetweenRestocking) = 0,
            cansInMachine := machineCansCapacity);
        */

        daysToEmptyCounter++;
    ];

    NewLine();

    //Save the days to empty data in a list for later analysis.
    daysToEmptyList := Append(daysToEmptyList, daysToEmptyCounter);

];

Echo("Days to empty data: ", daysToEmptyList);

Echo("Mean days to empty: ", N(Mean(daysToEmptyList)));
Echo("Standard deviation: ", N(StandardDeviation(daysToEmptyList)));


%/mathpiper

    %output,preserve="false"
      Result: True

      Side Effects:
      New Trial
      Day: 00   Cans in machine: 40
      Day: 01   Cans in machine: 25
      Day: 02   Cans in machine: 12
      Day: 03   Cans in machine: 00

      New Trial
      Day: 00   Cans in machine: 43
```

```
    Day: 01   Cans in machine: 32
    Day: 02   Cans in machine: 21
    Day: 03   Cans in machine: 04
    Day: 04   Cans in machine: 00

    New Trial
    Day: 00   Cans in machine: 38
    Day: 01   Cans in machine: 28
    Day: 02   Cans in machine: 12
    Day: 03   Cans in machine: 00

    New Trial
    Day: 00   Cans in machine: 31
    Day: 01   Cans in machine: 15
    Day: 02   Cans in machine: 01
    Day: 03   Cans in machine: 00

    New Trial
    Day: 00   Cans in machine: 36
    Day: 01   Cans in machine: 09
    Day: 02   Cans in machine: 00

    New Trial
    Day: 00   Cans in machine: 33
    Day: 01   Cans in machine: 08
    Day: 02   Cans in machine: 00

    New Trial
    Day: 00   Cans in machine: 35
    Day: 01   Cans in machine: 15
    Day: 02   Cans in machine: 00

    New Trial
    Day: 00   Cans in machine: 35
    Day: 01   Cans in machine: 20
    Day: 02   Cans in machine: 00

    New Trial
    Day: 00   Cans in machine: 30
    Day: 01   Cans in machine: 14
    Day: 02   Cans in machine: 05
    Day: 03   Cans in machine: 00

    New Trial
    Day: 00   Cans in machine: 27
    Day: 01   Cans in machine: 15
    Day: 02   Cans in machine: 00

    Days to empty data: {3,4,3,3,2,2,2,2,3,2}
    Mean days to empty: 2.6
    Standard deviation: 0.69920589886
.   %/output
```

7.8 The Probability Of Obtaining A Hand With Two Of A Kind In Cards

```
%mathpiper,title="Cards: two of a kind."

pairsCount := 0;

numberOfTrials := 40;

Repeat(numberOfTrials)
[
    deck := ShuffledDeckNoSuits();

    hand := Take(deck,5);

    Echo(hand);

    handPairCount := 0;

    ForEach(card, 1 .. 13)
    [
        If(Count(hand,card) = 2, handPairCount++);
    ];

    If(handPairCount = 1, handPairCount++);
];

NewLine();
Echo("Number of trials: ", numberOfTrials);
Echo("Probability of receiving a single pair: ",
    N(pairsCount/numberOfTrials) );

%/mathpiper

    %output,preserve="false"
      Result: True

      Side Effects:
      1 11 1 9 3
      11 8 13 12 7
      9 3 7 10 1
      9 11 4 11 4
      6 7 9 11 7
      4 1 8 3 12
      13 6 8 9 4
      6 3 13 11 7
      3 9 5 3 13
      9 11 9 3 6
      9 11 3 3 9
      1 12 4 2 1
      10 4 2 12 5
      8 12 5 12 10
      2 1 7 3 6
      10 11 13 6 9
      10 13 10 13 4
      8 1 7 6 1
```

```
    3 6 7 9 1
    9 1 3 3 8
    13 8 7 1 8
    8 9 10 6 10
    10 13 3 6 4
    6 1 2 12 6
    6 5 1 8 4
    1 13 8 3 7
    1 2 10 8 1
    10 8 3 5 13
    12 5 2 11 9
    7 8 3 7 11
    1 1 7 8 2
    3 1 2 13 8
    1 8 1 2 2
    1 10 8 10 8
    11 8 11 10 9
    10 1 9 1 13
    13 10 8 4 8
    12 4 8 3 11
    1 7 9 8 10
    11 10 6 4 11

    Number of trials: 40
    Probability of receiving a single pair: 0
.   %/output
```

7.9 The Probability Of Obtaining Two Pairs Vs. Three Of A Kind In Cards

```
%mathpiper,title="Cards: two pairs vs. three of a kind."

pairsCount := 0;

threeOfAKindCount := 0;

numberOfTrials := 1000;

Repeat(numberOfTrials)
[
    deck := ShuffledDeckNoSuits();

    hand := Take(deck, 5);

    //Echo(hand);

    handPairCount := 0;

    handThreeOfAKindCount := 0;
```

```

    ForEach(card, 1 .. 13)
    [
        If(Count(hand, card) = 2, handPairCount++);

        If(Count(hand, card) = 3, handThreeOfAKindCount++);
    ];

    If(handPairCount = 2, pairsCount++);

    If(handThreeOfAKindCount = 1, threeOfAKindCount++);
];

Echo("Probability of two pairs: ", N(pairsCount/numberOfTrials) );
Echo("Probability of three of a kind: ",
    N(threeOfAKindCount/numberOfTrials) );

%/mathpiper

    %output,preserve="false"
      Result: True

      Side Effects:
      Probability of two pairs: 0.071
      Probability of three of a kind: 0.025
.   %/output
```

7.10 A Random Walk Simulation

```
%mathpiper,title="Random walk."

targetPosition := {2,1};

successCount := 0;

numberOfTrials := 40;

numberOfSteps := 12;

Repeat(numberOfTrials)
[
    currentPosition := {0,0};

    walkPath := {};
```

```

    Repeat(numberOfSteps)
    [
        step:= RandomSymbol({
            { {1,0}, 1/4},
            { {-1,0}, 1/4},
            { {0,1}, 1/4},
            { {0,-1}, 1/4}});

        currentPosition := currentPosition + step;

        walkPath := Append(walkPath, currentPosition);
    ];

    Write(walkPath);

    If(Contains(walkPath, targetPosition),
        [successCount++; WriteString(" - Success");]);

    NewLine();
];

NewLine();
Echo("Number of trials: ", numberOfTrials);
Echo("The last walk path: ", walkPath);
Echo("Number of successes: ", successCount);
Echo("Probability of a success: ", N(successCount/numberOfTrials));


%/mathpiper

    %output,preserve="false"
      Result: True

      Side Effects:
{{0,1},{0,0},{0,-1},{-1,-1},{-1,0},{-1,-1},{-2,-1},{-1,-1},{0,-1},{1,-1},{2,-1},{3,-1}}
{{0,-1},{0,-2},{-1,-2},{-1,-1},{-1,-2},{-1,-1},{-1,0},{-1,1},{-1,0},{-1,1},{-2,1},{-2,0}}
{{1,0},{0,0},{-1,0},{-2,0},{-2,-1},{-3,-1},{-3,0},{-2,0},{-2,-1},{-3,-1},{-2,-1},{-3,-1}}
{{1,0},{1,1},{1,0},{0,0},{0,1},{0,2},{-1,2},{-1,1},{0,1},{-1,1},{0,1},{0,0}}
{{0,-1},{0,-2},{0,-1},{1,-1},{0,-1},{1,-1},{2,-1},{3,-1},{3,0},{3,1},{2,1},{2,2}} - Success
{{1,0},{1,1},{1,0},{2,0},{2,1},{3,1},{4,1},{4,0},{4,-1},{4,-2},{4,-3},{4,-4}} - Success
{{0,1},{0,2},{-1,2},{-1,3},{-1,4},{-1,3},{0,3},{1,3},{1,4},{0,4},{0,3},{-1,3}}
{{1,0},{2,0},{1,0},{2,0},{2,1},{1,1},{0,1},{0,2},{1,2},{1,1},{0,1},{-1,1}} - Success
{{1,0},{2,0},{2,1},{2,2},{1,2},{1,1},{2,1},{3,1},{2,1},{2,2},{2,3},{1,3}} - Success
{{-1,0},{-2,0},{-2,-1},{-1,-1},{-2,-1},{-1,-1},{-2,-1},{-1,-1},{-1,0},{-1,1},{-1,0},{-2,0}}
{{0,-1},{0,-2},{-1,-2},{-1,-1},{-1,-2},{-1,-3},{-1,-4},{-1,-5},{-1,-4},{-1,-5},{-1,-4},{0,-4}}
{{1,0},{1,1},{0,1},{-1,1},{-1,0},{-1,1},{-1,2},{-1,3},{-1,2},{-1,1},{-2,1},{-2,0}}
{{0,-1},{0,-2},{-1,-2},{-1,-3},{-1,-2},{0,-2},{-1,-2},{-1,-1},{-2,-1},{-2,0},{-2,1},{-3,1}}
{{1,0},{1,1},{1,0},{1,-1},{2,-1},{3,-1},{3,0},{3,1},{3,0},{3,1},{2,1},{3,1}} - Success
{{1,0},{1,1},{0,1},{0,0},{0,1},{0,0},{0,1},{0,0},{1,0},{0,0},{-1,0},{-2,0}}
{{0,1},{-1,1},{0,1},{0,2},{1,2},{1,3},{2,3},{2,4},{2,3},{2,2},{1,2},{2,2}}
{{-1,0},{-1,-1},{-1,-2},{-2,-2},{-2,-1},{-2,-2},{-2,-3},{-3,-3},{-3,-2},{-2,-2},{-3,-2},{-4,-2}}
{{0,1},{0,2},{-1,2},{-1,3},{-2,3},{-1,3},{0,3},{1,3},{1,2},{1,3},{2,3},{2,4}}
{{1,0},{2,0},{2,1},{1,1},{2,1},{2,0},{2,-1},{2,0},{1,0},{2,0},{2,1},{2,0}} - Success
{{0,1},{0,2},{-1,2},{-1,1},{0,1},{-1,1},{-2,1},{-2,0},{-2,1},{-1,1},{0,1},{-1,1}}
{{1,0},{1,1},{2,1},{3,1},{2,1},{2,2},{2,1},{3,1},{4,1},{5,1},{4,1},{5,1}} - Success
{{0,1},{0,0},{0,-1},{-1,-1},{-2,-1},{-2,0},{-2,-1},{-2,-2},{-2,-3},{-1,-3},{-2,-3},{-2,-4}}
{{0,1},{0,2},{0,1},{0,0},{0,-1},{0,-2},{0,-3},{0,-2},{1,-2},{2,-2},{2,-1},{2,-2}}
{{0,-1},{-1,-1},{-1,0},{-1,-1},{0,-1},{-1,-1},{-1,-2},{-2,-2},{-3,-2},{-3,-1},{-3,-2},{-4,-2}}
{{0,1},{0,2},{-1,2},{0,2},{-1,2},{-1,3},{-1,2},{-2,2},{-2,3},{-3,3},{-3,4},{-3,3}}
{{0,1},{0,2},{0,3},{1,3},{1,2},{1,1},{2,1},{2,2},{3,2},{3,1},{2,1},{3,1}} - Success
{{0,-1},{0,-2},{1,-2},{1,-3},{2,-3},{2,-2},{2,-1},{2,0},{2,1},{3,1},{3,2},{3,1}} - Success
```

```
{{-1,0},{-2,0},{-3,0},{-3,1},{-3,2},{-4,2},{-4,3},{-4,2},{-3,2},{-3,3},{-4,3},{-5,3}}
{{0,1},{-1,1},{0,1},{-1,1},{-1,2},{-1,3},{0,3},{0,2},{1,2},{0,2},{0,1},{1,1}}
{{-1,0},{0,0},{-1,0},{-2,0},{-2,1},{-2,2},{-3,2},{-2,2},{-1,2},{0,2},{1,2},{1,3}}
{{0,1},{0,0},{-1,0},{-2,0},{-3,0},{-4,0},{-4,-1},{-5,-1},{-6,-1},{-6,-2},{-5,-2},{-4,-2}}
{{0,-1},{-1,-1},{-1,0},{-1,1},{0,1},{1,1},{1,2},{0,2},{0,3},{0,4},{-1,4},{0,4}}
{{0,-1},{0,0},{-1,0},{-1,-1},{0,-1},{1,-1},{0,-1},{0,-2},{1,-2},{1,-1},{0,-1},{1,-1}}
{{-1,0},{-1,1},{-1,0},{-1,1},{0,1},{1,1},{2,1},{2,2},{2,3},{2,2},{2,3},{1,3}} - Success
{{-1,0},{-1,-1},{-1,-2},{0,-2},{1,-2},{2,-2},{1,-2},{2,-2},{2,-3},{3,-3},{4,-3},{5,-3}}
{{0,-1},{1,-1},{2,-1},{1,-1},{1,0},{2,0},{2,1},{1,1},{1,0},{2,0},{3,0},{3,-1}} - Success
{{0,1},{0,2},{-1,2},{-2,2},{-2,1},{-1,1},{0,1},{0,0},{0,1},{1,1},{1,2},{0,2}}
{{0,1},{0,2},{0,3},{-1,3},{-1,4},{0,4},{1,4},{2,4},{3,4},{4,4},{4,3},{4,2}}
{{1,0},{0,0},{0,-1},{0,-2},{0,-3},{0,-2},{0,-3},{-1,-3},{0,-3},{0,-4},{1,-4},{1,-3}}
{{0,1},{0,2},{0,3},{0,2},{0,3},{1,3},{1,4},{2,4},{1,4},{0,4},{-1,4},{-1,5}}


The last walk path: {{0,1},{0,2},{0,3},{0,2},{0,3},{1,3},{1,4},{2,4},{1,4},{0,4},{-1,4},{-1,5}}
 Number of trials: 40
 Number of successes: 11
 Probability of a success: 0.275
```

.   %/output

8 Mathematical Formulas, Mathematical Functions, Plotting, And GeoGebra

8.1 Applied Mathematics And Formulas

In this book we are exploring the areas of science, technology, engineering, and mathematics and therefore the branch of mathematics we are focusing on is **applied mathematics**. Here is a definition for applied mathematics:

> Applied Mathematics - Mathematics used to solve problems in other sciences such as physics, engineering or electronics, as opposed to pure mathematics. http://en.wiktionary.org/wiki/applied_mathematics

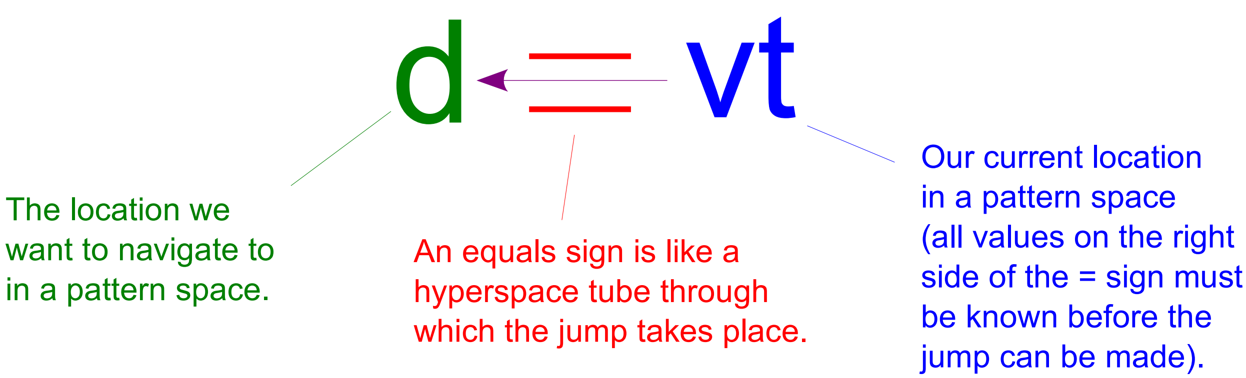
The applied mathematics which is used in science, technology, and engineering is full of **formulas** and a significant part of the work that scientists, technologists, and engineers do consists of working with these formulas. For example, one of the most often used formulas in physics is

$$distance = velocity \cdot time$$

or

$$d = vt$$

In this formula, ***d***, ***v***, and ***t*** are all **variables** which means that the **values** they represent can **vary**. In an earlier section we discussed how a mathematical formula can be thought of as a **jump point** which can be used to navigate through a pattern space. Most formula "jump points" are expressed as **equations** and the equals sign (=) can be thought of as a hyperspace **tube** through which the jump takes place. This idea is shown in the following diagram:



Lets put this formula into MathPiper and use it to make a jump. The following program deals with a car which has been traveling at a speed of 55 miles-per-hour for 1 hour. The program uses the above formula to "navigate" (calculate) to the location in pattern space which holds the number that represents the distance the car traveled during this hour:

```
%mathpiper,title=""

v := 55.0; //Miles per hour.

t := 1.0; //Hour.

d := v*t;

Echo("The speed of the car is ", v, "miles per hour.");
Echo("The amount of time the car has been traveling at this speed is ", t,
"hour.");
Echo("Therefore, the distance traveled is ", d, "miles.");

%/mathpiper

    %output,preserve="false"
      Result: True

      Side Effects:
      The speed of the car is 55.0 miles per hour.
      The amount of time the car has been traveling at this speed is 1.0 hour.
      Therefore, the distance traveled is 55.00 miles.
.   %/output
```

As indicated in the $d=vt$ diagram, all values on the **right** side of the = sign **must be known** before the jump can take place and in this program, the value **55.0** is assigned to variable **v** and the value **1.0** is assigned to variable **t**. Executing the program causes the jump to take place and the location we jumped to in pattern space indicates that the car traveled **55** miles during 1 hour of traveling at 55 miles-per-hour.

Notice that the values that are used in this program are **real numbers**. Most calculations which are made with science and engineering formulas are done with real numbers. The reason for this is that most phenomenon in the physical universe are continuous and therefore science and engineering formulas need to use **continuous variables** to represent these phenomenon.

8.2 Reconfiguring Jump Points With The == Operator And The Solve() Function

In the previous section, the formula "jump point" $d=vt$ was used with the known values of **velocity** and **time** to navigate to a location in pattern space which represented a **distance**. However, what if we are **currently at the location of the distance pattern** and we **want to navigate to the location of the**

velocity or the time pattern? An amazing property of formula jump points is that they can be **reconfigured** to jump **from any set of variables** in the formula **to any given variable** in the formula.

The process of reconfiguring a formula so that it jumps to a desired location in pattern space is called **solving the formula for a given variable**. All computer algebra systems are able to solve formulas for a given variable and in MathPiper, the **Solve()** function is used for this. Here is the calling format for the Solve() function:

```
Solve(equation, variable)
```

The first argument to Solve() is a **symbolic equation** and the second argument is the **unbound variable in the equation that is to be solved for** (which represents the destination in pattern space one wants to jump to). In MathPiper, the **==** "equals" operator is used to symbolically define an equation and in the following example, Solve() is used to solve the formula $d=vt$ for v:

```
In> Solve(d == v*t, v)
Result: {v==d/t}
```

Notice how the formula $d=vt$ is expressed as **d == v*t** in text form using the == operator. The result that is returned by Solve() is a symbolic equation which is also in text form and in this case it is v==d/t . The result is returned in a **list** because sometimes equations have more than one solution and all of these solutions are returned in the result list. Of course, a human could have easily solved this equation using the laws of algebra, but the Solve() function can also solve equations much larger than this one considerably quicker than a human can.

The following example shows the formula $d=vt$ being solved for **t**. The result is assigned to the variable **resultList** and then the solution is obtained from this list and assigned to the variable **solution**. Finally, Solve() is used to solve $t=\frac{d}{v}$ for the variable **d** and the result is the original $d=vt$ form of the equation which we started with:

```
In> resultList := Solve(d == v*t, t)
Result: {t==d/v}

In> solution := resultList[1];
Result: t==d/v

In> Solve(solution, d)
Result: {d==t*v}
```

8.3 Turning A Formula Into An Explicit Function, Independent And Dependent Variables

Earlier we wrote a program which used the formula $d=vt$ to calculate how far a car which was moving at a speed of 55 miles-per-hour had traveled after traveling for 1 hour. However, what if we wanted to calculate how far the car will travel during any time greater than or equal to 0 hours? The program we wrote could be made to do this by manually changing the value which is being assigned to ***t*** and then rerunning it, but a better approach is to turn $d=vt$ into an executable mathematical function. Here is a definition of a mathematical function:

> In traditional calculus, a function is defined as a relation between two terms called variables because their values vary. Call the terms, for example, x and y. If every value of x is associated with exactly one value of y, then y is said to be a function of x. It is customary to use x for what is called the "**independent variable,**" and y for what is called the "**dependent variable**" because its value depends on the value of x.[1] Therefore, $y = x^2$ means that y, the dependent variable, is the square of x, the independent variable.[1][2]
>
> The most common way to denote a "function" is to replace y, the dependent variable, by $f(x)$, where f is the first letter of the word "function." Thus, $y = f(x) = x^2$ means that y, a dependent variable, a function of x, is the square of x. Also, in this form, the expression is called an "explicit" function of x, contrasted with $x^2 - y = 0$, which is called an "implicit" function.
> http://en.wikipedia.org/wiki/Dependent_and_independent_variables

Turning our formula into an **executable explicit function** is easy and here is how it is done:

```
In> f(t) := 55*t
Result: True
```

In English this function would read "f of t equals 55 times t." Since we are currently only interested in using this function when the speed of the car is 55 miles-per-hour, we simply use the constant **55** in place of the variable ***v***. In this function, ***t*** is the **independent variable** and ***d*** (which is now represented by **f(t)**) is the **dependent variable**. Now that this function has been defined, it can be used to calculate how far the car will travel for various times:

```
In> f(1)
Result: 55
```

```
In> f(2)
Result: 110
```

```
In> f(5)
Result: 275

In> f(2.3)
Result: 126.5

In> f(7.75)
Result: 426.25
```

The **Table()** function can be used with **f(t)** to obtain a list which has multiple distances in it:

```
In> Table(f(t),t,1,10,1)
Result: {55,110,165,220,275,330,385,440,495,550}
```

If an **unbound symbolic variable** is passed to this function instead of a **numeric value**, the expression that the function uses to perform the calculation is returned:

```
In> f(t)
Result: 55*t

In> f(x)
Result: 55*x
```

8.4 The Domain And Range Of A Function

A function has a set of values that can be **sent to it** (called its **domain**) and a respective set of values that it can **return** (called its **range**). Here are definitions for the domain and range of a function:

> The **domain** of a function is the complete set of possible values of the independent variable in the function. The **range** of a function is the complete set of all possible **resulting values** of the dependent variable of a function, after we have substituted the values in the domain.
> http://www.intmath.com/Functions-and-graphs/2a_Domain-and-range.php

In the following program, a list of **domain values** which consists of the integers 0-10 inclusive is created and then **these values are each in turn sent to the function** $f(t)=55t$ **for evaluation**. The results of each evaluation are placed into a **range values list** and then the function, the domain list, and the range list are printed.

```
%mathpiper,title=""

f(t) := 55*t;

domainList := 0 .. 10;
```

```
rangeList := {};

ForEach(domainValue, domainList)
[
    rangeList := Append(rangeList, f(domainValue));
];

Echo("The function f(t) = ", f(t));
Echo("The function's domain is: ", domainList);
Echo("The function's range is: ", rangeList);

%/mathpiper

    %output,preserve="false"
      Result: True

      Side Effects:
      The function f(t) = 55*t
      The function's domain is: {0,1,2,3,4,5,6,7,8,9,10}
      The function's range is: {0,55,110,165,220,275,330,385,440,495,550}
.   %/output
```

It is usually **impossible** to work with **all** the values in a function's domain and range because the **amount of values is to large**. Therefore, most programs that use executable mathematical functions only use a **small subset** of the function's domain and range values like this program does.

In the program, the list of numerical distances which are calculated and then placed into **rangeList** can be thought of as representing jump point destination locations in the $d=vt$ formula's pattern space. Having these values in a list is more convenient than working with them separately, but it is still somewhat **difficult to visualize** what the **relationship** is between the range and domain values when they are viewed in this **textual format**. What would be **nice** to have is something like a **graphical map of a function's pattern space**, similar to a road map or a hyperspace jump point map. Luckily, tools for obtaining graphical "maps" like this are available and some of them are covered in the next section.

8.5 Plotting A Function (Obtaining A Graphic "Map" Of A Function's Pattern Space)

Plotting a function means to take a **sequence** of a function's **domain values**, **pair them** with their respective **range values**, and then **show each pair as a graphic point** on a two dimensional graphical background. Each point's **domain value, range value pair** is called its **coordinates** and these coordinates determine where a given point is located on the plot. In mathematical notation, the coordinates of a point are often placed inside of **parentheses** like this:

(domain_value, range_value)

For example, here are the coordinates of the points which were generated by the program in the previous section:

```
(0,0)
(1,55)
(2,110)
(3,165)
(4,220)
(5,275)
(6,330)
(7,385)
(8,440)
(9,495)
(10,550)
```

These points, together with the background they are drawn on, can be thought of as being a map of the function's pattern space. The subsections in this section show various ways to plot the function f(t) = 55*t.

8.5.1 Generating 11 Points With A ForEach Loop And Plotting Them With ScatterPlot()

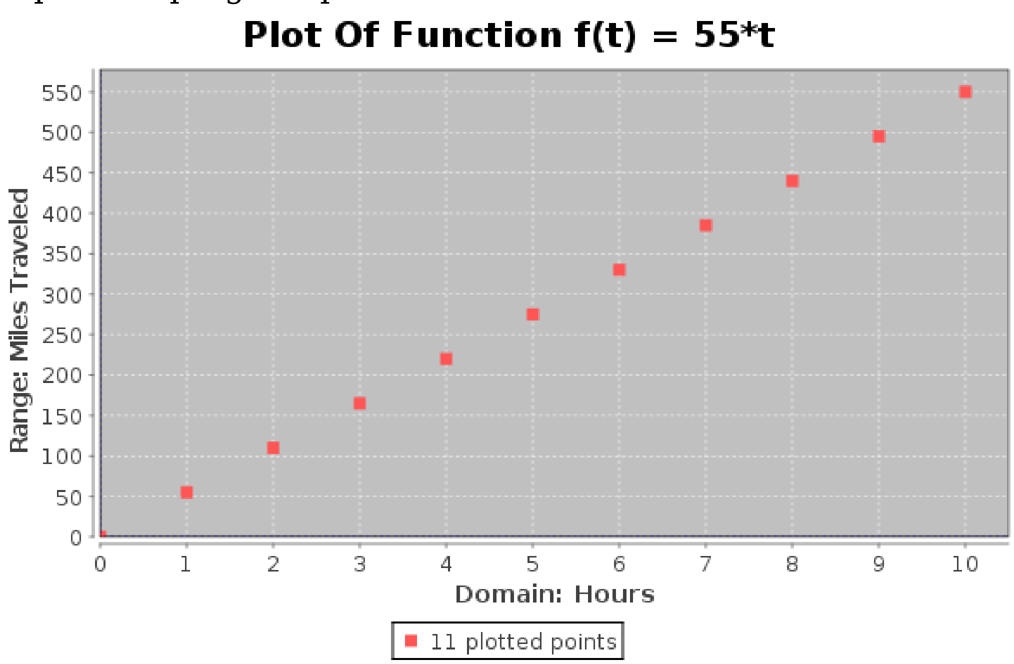
In the following program, a list of 11 domain values (0-10) called **domainList** is created and then a ForEach loop is used to send each of these domain values in turn to the function $f(t)=55t$ for evaluation. The resulting **range** values are placed into a list called **rangeList**. The contents of domainList and rangeList are printed and then sent to a function called **ScatterPlot()** for plotting in the JFreeChart plugin (the plot is shown below the program):

```
%mathpiper,title="ForEach() based program."

f(t) := 55*t;

domainList := 0 .. 10;

rangeList := {};

ForEach(domainValue, domainList)
[
    rangeList := Append(rangeList, f(domainValue));
];

Echo("The function f(t) = ", f(t));
Echo("The function's domain is: ", domainList);
Echo("The function's range is: ", rangeList);

ScatterPlot({domainList, rangeList},
    title -> "Plot Of Function f(t) = 55*t",
    xAxisLabel -> "Domain: Hours",
    yAxisLabel -> "Range: Miles Traveled",
    series1Title -> String(Length(domainList)):" plotted points");

%/mathpiper
```

```
    %output,preserve="false"
      Result: org.jfree.chart.ChartPanel

      Side Effects:
      The function f(t) = 55*t
      The function's domain is: {0,1,2,3,4,5,6,7,8,9,10}
      The function's range is: {0,55,110,165,220,275,330,385,440,495,550}
.   %/output
```

In this program, **ScatterPlot()** takes a **list** as its first argument. The list contains a **domain list** and its matching **range list** and these two lists **must have the same number of elements** in them or an error will occur. Notice that the ScatterPlot() function accepts the same **options** that the Histogram() function does.

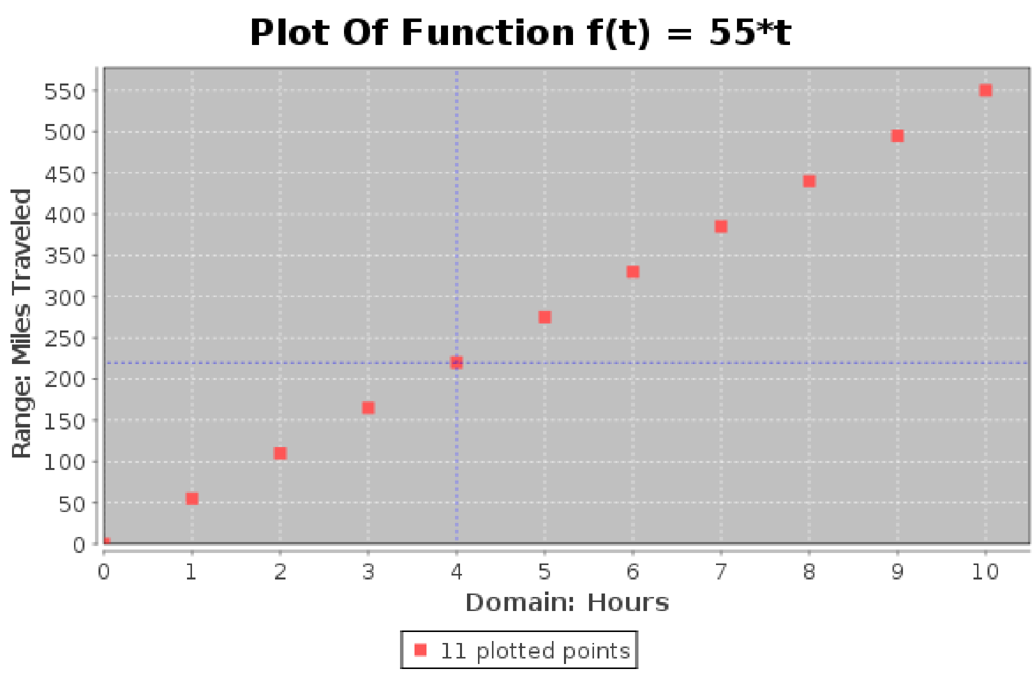
Here is the plot the program produces:



In this plot, the **Domain axis** runs **horizontally** along the bottom of the plot and the **Range axis** runs **vertically** along the left side of the plot. Each domain/range pair of values is represented as a red point. The **domain value** for each of these **points** can be found by locating the number which is **directly below it** on Domain axis and the **range value** can be found by finding the number which is **directly to its left** on the Range axis.

8.5.2 Analyzing The Plotted Points With Cross Hairs And Mouse Pointer Hovering

It is sometimes difficult to precisely determine the domain and range values for a point on a plot and therefore tools are available to help with this. If you select any one of the points in the plot from the previous section by clicking on it with your mouse pointer, **blue cross hairs** will appear on it as shown in the next plot:



The blue cross hairs make it easier to determine the domain and range values for any given point. If you would like to know the **exact** domain and range values for a given point, simply place your mouse pointer **over the point** and let it **sit there for a few seconds** without clicking it or moving it. This is called "**hovering**" and when the mouse pointer is made to hover over a point, a small message will be displayed which contains the exact domain and range values for the point.

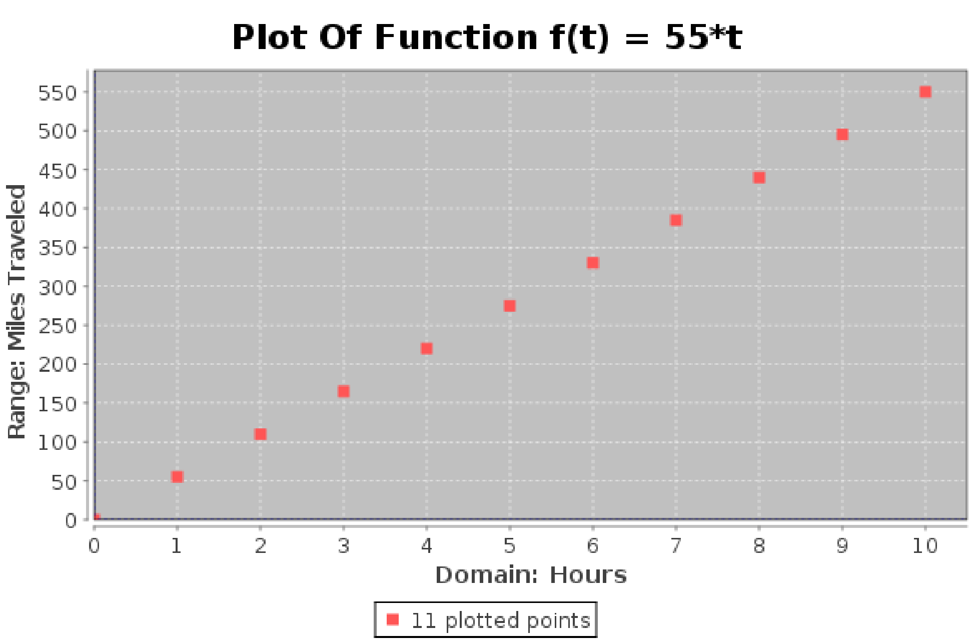
8.5.3 Generating 11 Points With Table() And Plotting Them With ScatterPlot()

The previous program shows how to use a ForEach loop to generate points with a function, but a more **convenient** way to do this is with the **Table()** function. The following program generates the **same data and plot** that ForEach() based program did, except it uses a Table() function call to generate the domain list and another Table() function call to generate the range list:

```
%mathpiper,title="Table() based program."

f(t) := 55*t;

beginValue := 0;
```

```
endValue := 10;

stepAmount := 1;

domainList := Table(x,x,beginValue,endValue,stepAmount);

rangeList := Table(f(t),t,beginValue,endValue,stepAmount);

Echo("The function f(t) = ", f(t));
Echo("The function's domain is: ", domainList);
Echo("The function's range is: ", rangeList);

ScatterPlot({domainList,rangeList},
    title -> "Plot Of Function f(t) = 55*t",
    xAxisLabel -> "Domain: Hours",
    yAxisLabel -> "Range: Miles Traveled",
    series1Title -> String(Length(domainList)):" plotted points");

%/mathpiper

    %output,preserve="false"
      Result: org.jfree.chart.ChartPanel

      Side Effects:
      The function f(t) = 55*t
      The function's domain is: {0,1,2,3,4,5,6,7,8,9,10}
      The function's range is: {0,55,110,165,220,275,330,385,440,495,550}
.   %/output
```

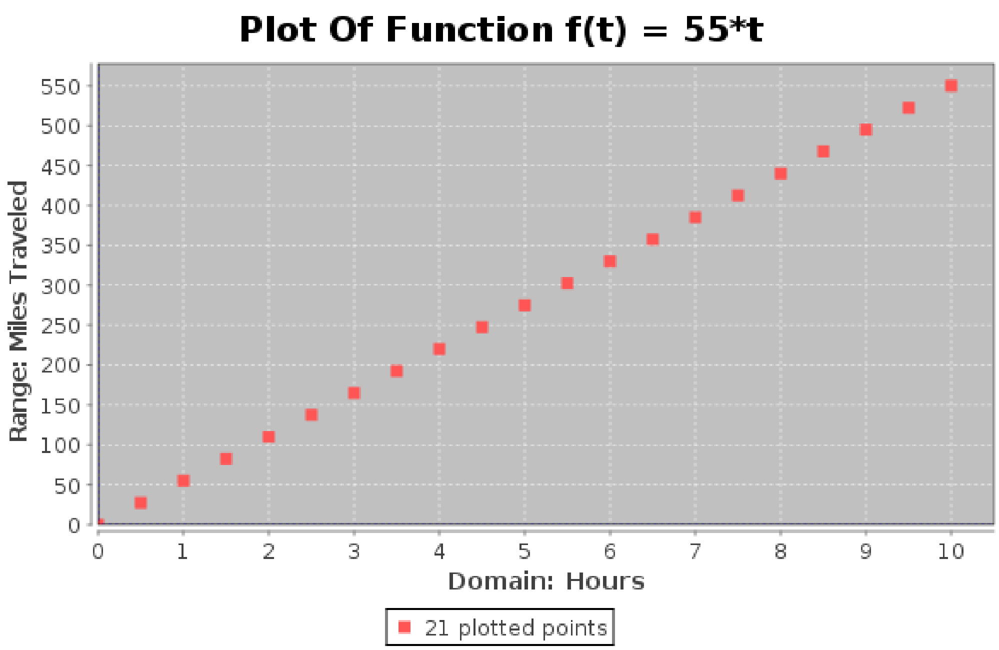
Here is the plot that this program generates. It is identical to the plot that was generated by the ForEach based code:



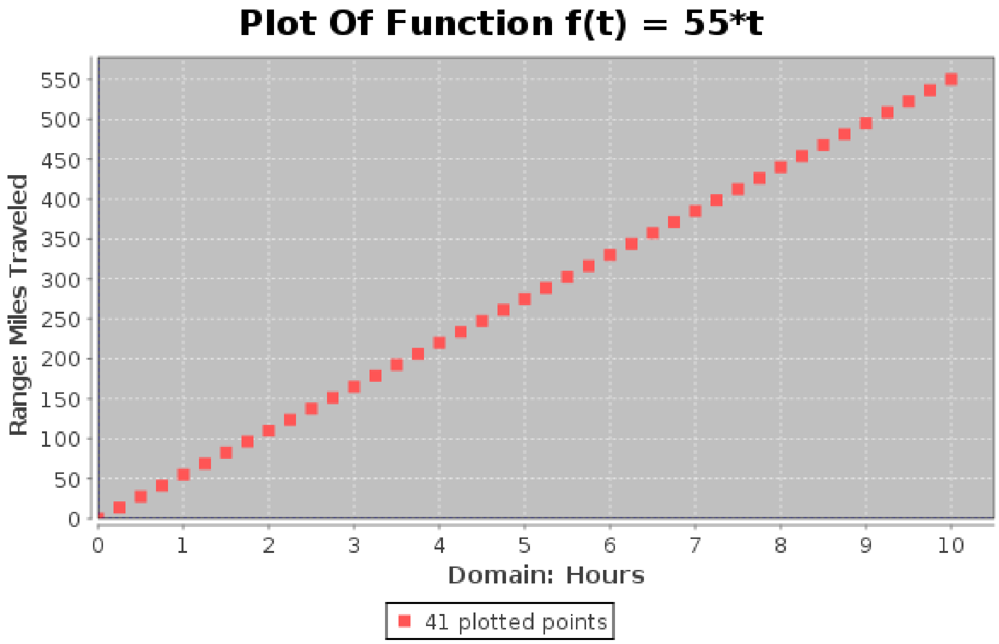
Notice that this program uses a variable called **stepAmount** to specify the **numerical distance** between the **domain axis values of the points** that are generated. As can be seen in this plot, a **stepAmount** value of 1 produces a fairly large space between the points. The **stepAmount** variable is used in the following sections to specify increasingly smaller distances between the domain axis values of the points until the visual space which is between them is eliminated.

8.5.4 Generating More Accurate Plots With Table() And ScatterPlot()

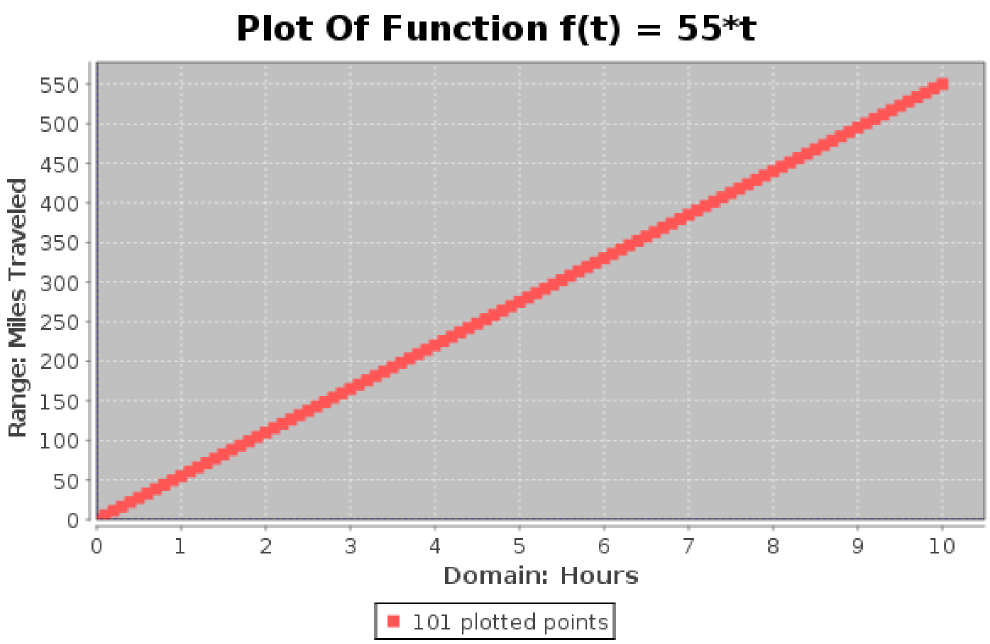
The plots of the function $f(t)=55t$ in the previous sections were a good start, but the **space** between the points was somewhat **large**. If our goal is to create a map of a function's pattern space, then a plot which has its points **closer together** would be a more usable map. The following plot contains **21** points instead of **11** and it was generated by the Table() based program by setting the variable **stepAmount** equal to **.5**:



This plot is better, but there are still fairly large spaces between the points. Lets **decrease the spaces between the points further** by setting **stepAmount** to **.25**:



This plot has **41** points and the **spaces between the points have almost disappeared**. Setting **stepAmount** to **.1** produces **101** points and the plot that is produced does **not have any visual space between the points**:



Even though there are an **infinite number of points** that could be plotted for a function, this plot shows that **only enough of them** need to be plotted so that there are **no visible spaces** between the points. All plotting functions use this principle when plotting.

8.5.5 Generating Points With Plot2D But Not Plotting Them

The ScatterPlot() based programs in the previous sections were fairly good for creating plots, but there are other plotting-oriented MathPiper functions which can plot mathematical functions with **less code**. One of these MathPiper functions is **Plot2D()** and here is an example of Plot2D() being used to **print** domain and range points for the function $f(t)=55t$:

```
%mathpiper,title=""

f(t) := 55 * t;

Plot2D(f(t), 0:10, output -> data);

%/mathpiper

    %output,preserve="false"
      Result: {{{0,0},{0.4166666668,22.91666667},{0.8333333335,45.83333334},
{1.250000001,68.75000006},{1.666666667,91.66666669},{2.083333334,114.5833334},
{2.500000001,137.5000001},{2.916666668,160.4166667},{3.333333334,183.3333334},
{3.750000001,206.2500001},{4.166666668,229.1666667},{4.583333335,252.0833334},
{5.000000001,275.0000001},{5.41666667,297.9166669},{5.833333335,320.8333334},
{6.25000000,343.7500000},{6.666666668,366.6666667},{7.083333335,389.5833334},
{7.500000002,412.5000001},{7.91666667,435.4166669},{8.333333335,458.3333334},
{8.75000000,481.2500000},{9.166666669,504.1666668},{9.583333335,527.0833334},
{10.00000000,550.0000000}}}
.   %/output
```

The **first** argument to Plot2D() is the **function to be plotted** and the **second** argument specifies the span of **domain values to plot over**. The value to the **left** of the colon indicates the **smallest** domain value in the plot and the value to the **right** of the colon specifies the **largest** domain value. Therefore, 0:10 means the plot starts at **0** in the domain and plots up through **10**.

Notice that unlike the ScatterPlot() based programs (which used **separate lists** for the **domain** and **range** values) **Plot2D() places each domain value with its respective range value in a sublist**. For example the domain and range value for the first point after {0,0} is {0.4166666668,22.91666667}. Also notice that the domain values are **not integers**, but rather they are **real numbers** which have been selected because they will produce a plot with no gaps visible between the points.

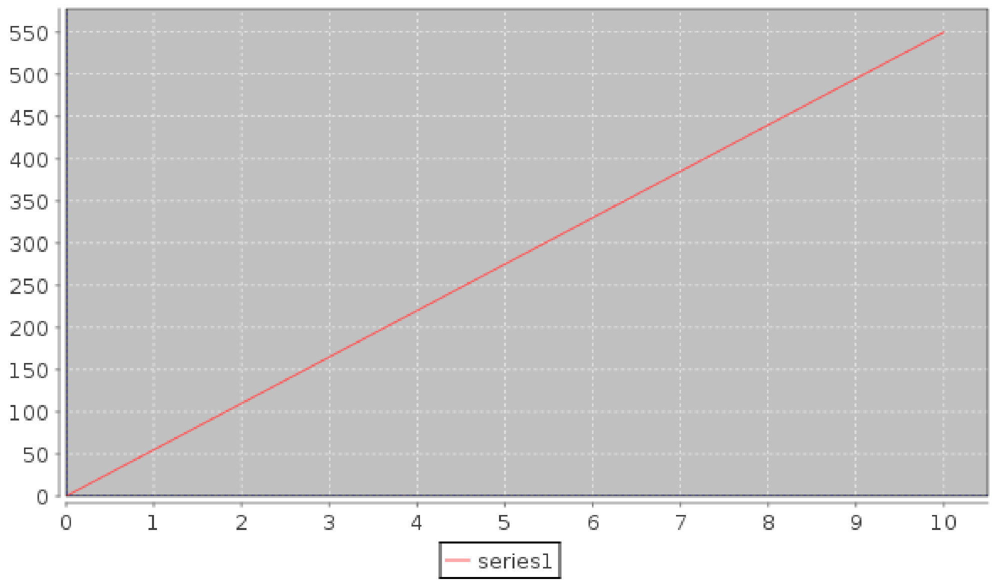
Finally, the option **output** has been set to **data** to have Plot2D() **print** its points instead of graphically plotting them.

8.5.6 Plotting A Function With Plot2D()

The following program uses Plot2D() to display a graphical plot of the function $f(t)=55t$:

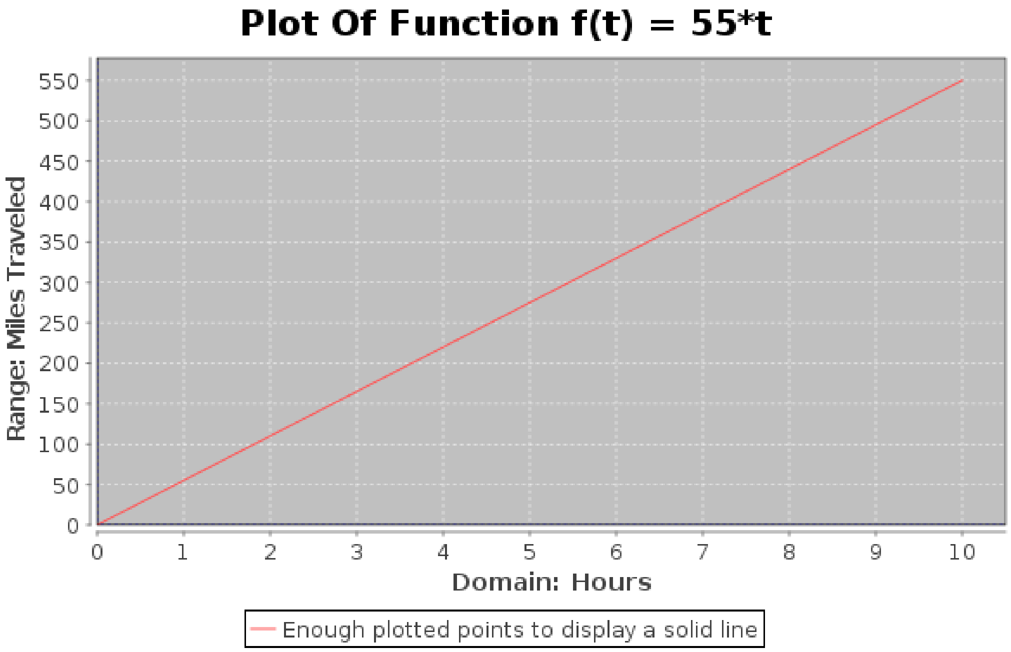
```
%mathpiper,title=""
```

```
f(t) := 55*t;

Plot2D(f(t),0:10);

%/mathpiper

    %output,preserve="false"
      Result: org.jfree.chart.ChartPanel
.   %/output
```



This is a good looking plot, but it would be more informative if it had a **title** and **labels** for the domain axis, the range axis and the data. Here is a program that adds this information:

```
%mathpiper,title=""

f(t) := 55 * t;

Plot2D(f(t),0:10,
    title -> "Plot Of Function f(t) = 55*t",
    xAxisLabel -> "Domain: Hours",
    yAxisLabel -> "Range: Miles Traveled",
    series1Title -> "Enough plotted points to display a solid line");

%/mathpiper

    %output,preserve="false"
      Result: org.jfree.chart.ChartPanel
.   %/output
```



8.5.7 Calculating The Slope Of The Function f(t) = 55t (Rise Over Run)

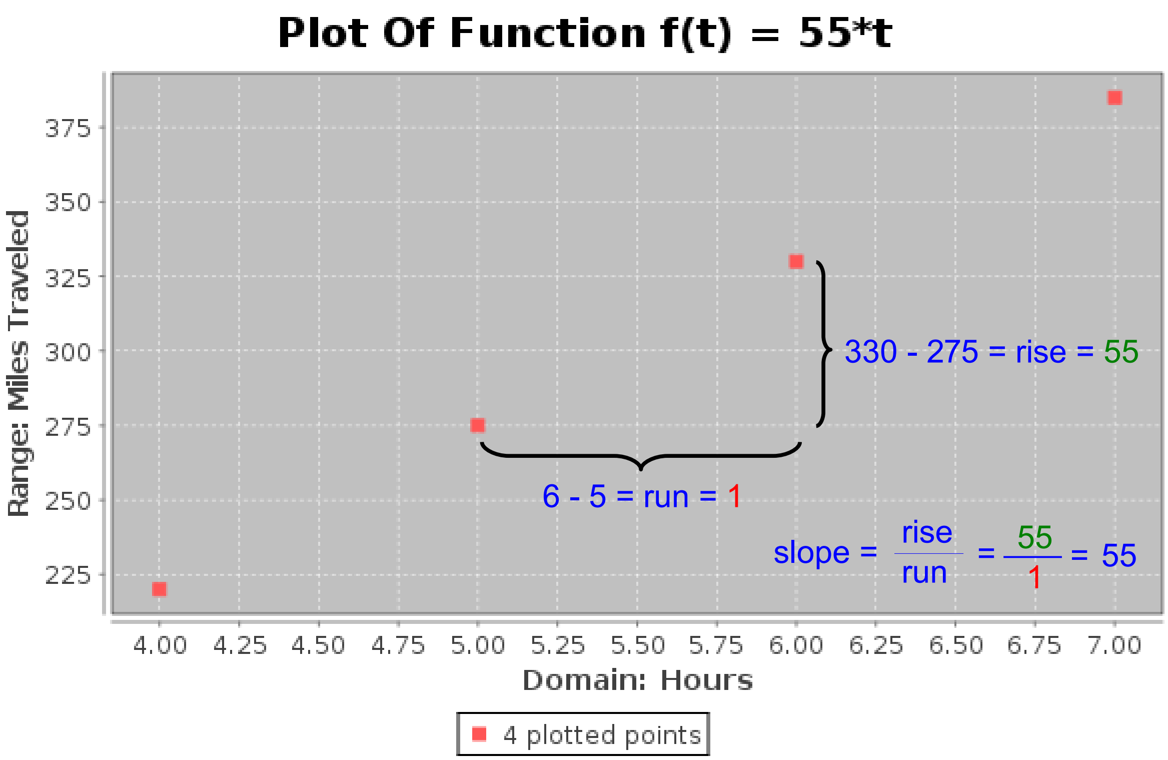
The plots of the function $f(t)=55t$ we have been working with enables a person to determine how far a car that is moving at a speed of 55 miles-per-hour has traveled during a given amount of time. This can be done by selecting an hour on the domain axis, locating the point on the line which is directly above this hour value, and then finding the miles value which is to the left of this point on the miles traveled axis.

If you will recall, the function $f(t)=55t$ was developed from the formula $d=vt$ and the **domain axis** of the plot indicates the values that the **time** variable ***t*** can be set to and the **range axis** indicates the values that the **distance** variable ***d*** can have. But what about the **velocity** variable ***v***, which we set to 55? Where is it represented in the plots?

It turns out that the velocity value of 55 is represented by the **steepness** of the plotted line and this **steepness** is referred to as the **slope** of the line. The following plot shows how the slope **of a line** can be determined by performing a simple calculation on two of the points in the line.

8.5.7.1 Calculating The Slope Using Two Adjacent Points

The following plot shows the slope of the line being calculated using two adjacent points:

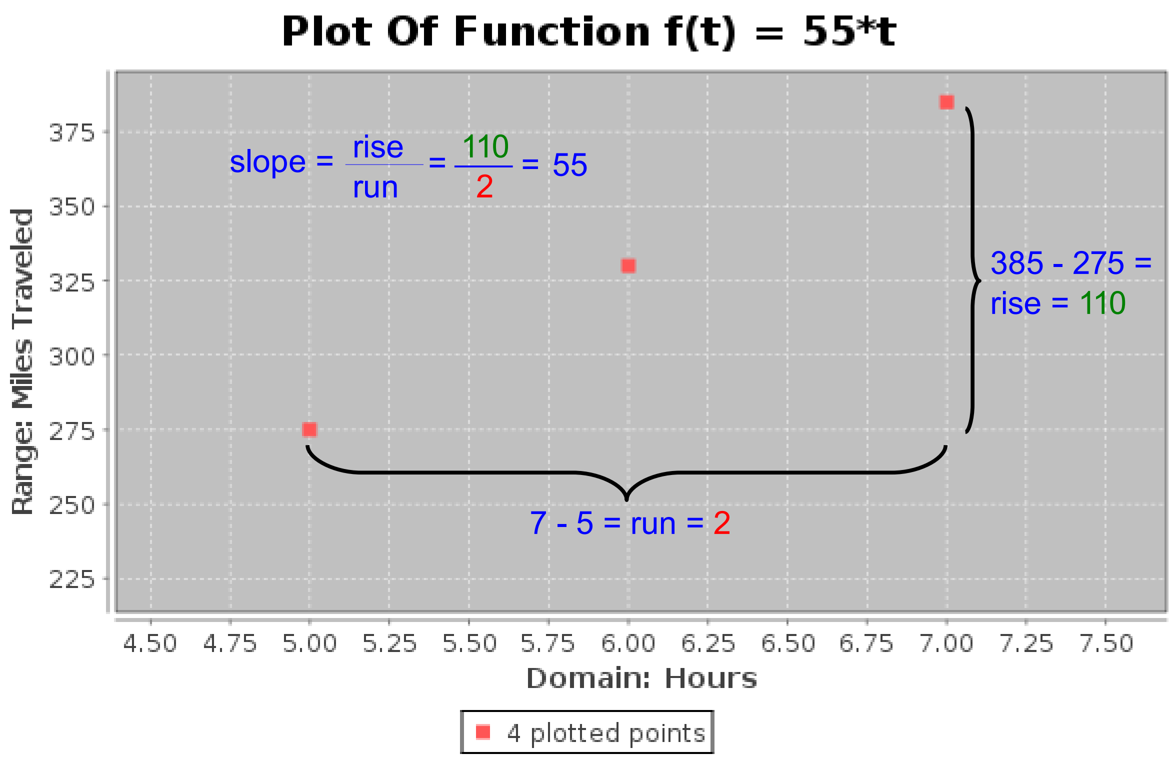


In this plot, the points (5,75) and (6,330) are used to calculate the **slope** of the line. First, the amount the domain value changed between the two points is determined by calculating the difference between the two point's domain values. This **domain value difference** is called the **run** and in this case the calculation is 6 - 5. Then, the amount the range value changed between the two points is determined by calculating the difference between the two point's range values. This **range value difference** is called the **rise** and here the calculation is 330 - 275. Finally, the **slope** is determined by **dividing** the **rise value** by the **run value** and this division is often referred to as calculating "**rise over run**." Here is the calculation:

$$slope = \frac{rise}{run} = \frac{55}{1} = 55$$

This calculation on two adjacent points indicates that the slope of the line is 55, but would the same slope be arrived at if the rise over run cal

8.5.7.2 Calculating The Slope Using Two Points Which Are Not Adjacent



$$slope = \frac{rise}{run} = \frac{110}{2} = 55$$