Mathematica Bootcamp[*]

A.J. Hildebrand

Contents

[*]Notes from a Computational Bootcamp held May 30 – June 1, 2018, at the University of Illinois, as part of the University's PI4 program. Funding provided by National Science Foundation grant DMS 1345032 MCTP: PI4: Program for Interdisciplinary and Industrial Internships at Illinois. I thank John Haug, TA for the Bootcamp, for assistance with preparing these notes.

About the Bootcamp

This Mathematica Bootcamp is based on a series of workshops I have developed for summer REU programs during the past three years. Most participants in these programs had no knowledge of Mathematica at the outset; a few had some basic familiarity with Mathematica, e.g., through courses, research projects, or NetMath. By the end of the program, all had attained a high level of proficiency in Mathematica, and several of the participants created interactive visualizations that have been published at the Wolfram Demonstrations website.[1]

These workshops are aimed both at novices to Mathematica, and at those with some basic knowledge of Mathematica. The goal of these workshops is for you to:

- **Develop a solid *general* foundation in Mathematica.** Get familiar with the structure and syntax of Mathematica, learn how to do common tasks in a wide range of areas of mathematics, and get a taste of the power and elegance of Mathematica.
- **Develop *in-depth* knowledge of applications in selected areas.** We will focus on topics relevant for the PI4 program, such as series expansions and asymptotics.
- **Develop and practice good coding skills.** Learn useful tips and tricks, and learn to avoid common mistakes and pitfalls.
- **Learn how to use Mathematica as a tool for research.** You'll learn how to use Mathematica efficiently to perform large scale numerical/visual experimentation, and to use such experimentation to discover/guess new results.

Format. Each workshop will be about two hours long and focus on a particular topic, e.g., Mathematica basics, lists, functions, graphics. The first hour or so of each workshop will consist of a presentation that you can follow along. In the second hour you will do exercises and a project, while I will be circulating through the room to provide individual help. In the projects you will learn how to use Mathematica as a tool for research. You'll use experimentation and visualization with Mathematica to come up with conjectures and discover new results.

Course materials. There will be a handout accompanying each workshop that includes the exercises and the project, highlights some key points, and provides tips and advice on developing good coding habits and avoiding common mistakes and pitfalls.

Course materials will also be made available through a U of I Box folder. If you have not used Box (through the U of I interface) before, you may need to set up an account first. Go to http://box.illinois.edu, and click on "Sign up".

[1]To see these, go to demonstrations.wolfram.com, and type any of the following names into the search box: Narken Aimambet, Chingis Matayev, Khoa Tran, Laila Zhexembay, Jose Sanchez, Madina Bolat, Daniyar Omarov.

How to get the most out of the workshops

- **Try to work with your neighbor(s) and help each other out in case one of you gets stuck.**
- **Those new to Mathematica.** Follow the presentation, using the accompanying Mathematica notebook to try things out yourself and get comfortable with the basic syntax of Mathematica, and with working with notebooks. Try making small changes to the commands, and observe the output; use this opportunity to see what works, and what doesn't, and try to learn from errors. Next, do the exercises on the handout (after opening up a new notebook), and finally try the projects.
- **Those with prior knowledge of Mathematica.** If you are already familiar with the topic covered, do only selected exercises (e.g., pick one or two from each type), then move on to the projects. Focus on developing good coding habits, and unlearning any bad habits you may have acquired (such as using "For" or "While" loops in Mathematica ...).
- **Carefully read all of the tips/tricks/advice sections on the handouts.** These sections focus on the most important points and the most common sources of errors, and they help you develop good coding habits and guard against mistakes. **Pay particular attention to the boxed/highlighted sections of the handout; these include some of the most useful tips.**
- **After the workshop, finish any left-over exercises/projects, review the handouts, and continue practicing Mathematica on your own.** Mathematica is available on the computers in 239 Altgeld, and in the graduate student offices and labs in Altgeld and Coble Hall, and on all machines in the Illini Union lab and other University computer labs. If you want to use Mathematica on your own laptop, you can download an inexpensive version (and free for faculty/staff) from WebStore, `webstore.illinois.edu`

.

Questions/comments/suggestions. If you have questions about the material of the workshop, just ask during the workshop.

I'd be happy to assist with any Mathematica-related problems you encounter after the workshop (e.g. in your PI4 research). I will be on campus through mid/late July; just email me at ajh@illinois.edu.

I welcome comments and suggestions about these notes; email me at ajh@illinois.edu.

1 Getting Started

1.1 First steps

1. **Find Mathematica on the computer and open it.** All computers in 239 Altgeld have Mathematica installed; look for the orange "spikey" icon. If you want to use Mathematica on your own laptop, you can either download a free 15 day trial version from Wolfram, `wolfram.com` or purchase a student version through Webstore, `webstore.illinois.edu` ($35 per semester for students, free for faculty/staff).

2. **Open a Mathematica Notebook.** A Mathematica Notebook is where you do all of your work. Open a new notebook, by clicking on "New Document", then "Notebook." In addition, you might want to download and open the notebook accompanying this workshop using the "Open" tab. (Access will be provided to registered participants.)

3. **Follow along the class presentation and do the exercises for the first workshop (see the section "Mathematica Basics" below).**

1.2 Working with Mathematica Notebooks: Tips and Tricks

1. **Increase the magnification in your notebook to 125% (or larger), to make the text more readable (see the button on the lower right corner of the window).** This makes it easier to distinguish between different types of parentheses, especially, e.g., curly braces `{}` and square brackets `[]`, a common source of errors.

2. **Press Shift-Enter after each command to evaluate this command and generate its output.** Pressing Enter doesn't do anything; also, a semi-colon (;) at the end of the command has the effect of **suppressing** the output instead of showing the output. (Occasionally, e.g., with commands that generate huge amounts of output, this may be desirable, but normally you want to see the output.) To evaluate (or re-evaluate) all commands in a notebook, click on the "Evaluation" tab, then select "Evaluate Notebook."

3. **Save your work frequently.** Click on "File", then "Save". As you are learning Mathematica, you will make frequent mistakes, and some of these may cause Mathematica to crash. **If you are using one of the lab computers, email the notebook file to yourself at the end of each session so you have it available the next time.**

4. **Dealing with freezes and crashes.** If Mathematica gets "stuck" (e.g., because a calculation takes too long), click on "Evaluation", then "Abort Evaluation." Sometimes, it takes several tries to "kill" an evaluation. Occasionally, it doesn't work at all, and Mathematica ends up crashing ... Hence the advice to frequently save your work.

5. **Selecting, copying/pasting, and deleting material.** A notebook is organized into cells, either an **input cell** (indicated by `In[...]=...`) or an **output cell** (indicated by `Out[...]=...`). Each cell has an accompanying bracket on the right side of the window. To select the cell, click on this bracket. You can use the Edit menu to delete/cut, or copy the cell. For example, if a calculation inadvertently generated large amount of output, you can delete the output cell using this technique.

6. **Text entry vs. visual entry of symbols.** Under the "Palettes" tab Mathematica provides an options to enter mathematical notation (e.g., squareroots, fractions, exponents, or mathematical constants π, e, i, etc.) by picking appropriate symbols from a palette, instead of typing in the corresponding text version. You are, of course, free to do this if you want, and some tutorials will teach you this, but my advice is to not bother with the palette:

TIP: Don't bother with palettes, and things liks Math Assistant, Class Assistant, etc. For mathematical symbols and special letters just type in the appropriate text equivalent: `Pi` for π, `Sqrt` for a squareroot, `a/b` (with a forward slash) for the fraction $\frac{a}{b}$, `a^b` for the power a^b, etc. Hunting for symbols in a palette will only slow you down, and you'll have know the text equivalents anyway in order to be able to read code others have written.

7. **Getting help.** Click the Help tab to open access the Mathematica Documentation. If you don't know how to do something in Mathematica use the search function in the Documentation to find appropriate help pages. Particularly useful are the sections "See Also" and "Tutorials" near the bottom of each help page.

TIP: Keep a documentation window open at all times, so you can access it if needed.

8. **Quick access to help pages.** To get the documentation for a particular command, start typing the command until a circled "i" symbol pops up. Alternatively, you can slowly move your mouse through the commandname from left to right until the circled "i" symbol pops up. Clicking on this symbol will pop up a window with the documentation page for this command.

TIP: Use built-in documentation rather than searching online. The built-in documentation (accessed through the "Help" tab in a Notebook window, or by clicking on a circled "i" as described above) gives more reliable information and is specific to the particular Mathematica version you have installed.

1.3 Additional resources

For those with prior programming experience, I recommend the tutorial **Fast Introduction for Programmers**, available at `www.wolfram.com`. (Just type the title, "Fast Introduction", into the search box.) This is a short interactive quickstart type tutorial that emphasizes the differences and similarities to other programming languages. There is a version for Python programmers and another for Java programmers.

2 Mathematica Basics

2.1 Top things to know about the Mathematica Syntax

1. **All built-in mathematical constants and functions begin with a capital letter.** For example, sin must be typed as `Sin`, not `sin`; the functions for sums, products, and integrations are all capitalized: `Sum`, `Product`, `Integrate` The same applies for mathematical constants: `E` is the Euler constant, `I` is the imaginary unit, and `Pi` denotes the number π.

2. **For your own variables and functions always use lower case names.** This is to avoid conflicts with built-in functions and constants. For example, `n`, `n1`, `nmax`, `nmin`, `average`, are all acceptable variables or function names, but `N`, `Average` should be avoided.

3. **Mathematica uses square brackets, not round parentheses, for function arguments.** For example, $\sin(\pi)$ must be coded as `Sin[Pi]`, not `Sin(Pi)`. *This is different from languages like Python or R.*

4. **Round parentheses are used for grouping:** For example, $(1+x)^2$ is coded as `(1+x)^2`, and e^{x^2} as `E^(x^2)`. *This is different from LaTeX, which uses curly braces,* `{....}`*, for grouping.*

5. **Mathematica evaluates expressions symbolically, rather than numerically, unless you explicitly tell it otherwise.** If Mathematica cannot evaluate the expression symbolically, it will reproduce the input. For example, while `Sin[Pi/2]` gives the expected answer, 1, `Log[Pi/2]` just reproduces the input. To get a numerical value, use the `N` function: `N[Log[Pi/2]]`.

2.2 Exercises

1. **Basic arithmetic.** Use Mathematica to evaluate the following expressions, exactly or numerically as indicated.
(Hint: You'll need the following built-in functions/constants: `Pi`, `E`, `I`, `Log`, `Sin`, `Sqrt`, `N`.)

   (a) $\frac{355}{113}$ (numerical)
   (b) $\sin(\pi/6)$ (exact and numerical)
   (c) $e^{i\pi/6}$ (exact and numerical)
   (d) e (numerical)
   (e) $(1+1/10^6)^{10^6}$ (exact and numerical)
   (f) 2^{1000} (exact)
   (g) $\log_2 1024$ (exact)
   (h) $\log 10$ (numerical—here log denotes the natural log)
   (i) $\frac{1}{2}(1+\sqrt{5})$ (exact and numerical)

2. **Sums and products. Use Mathematica to evaluate the following sums and products:** Use exact evaluation unless otherwise indicated.
(Hint: You'll need the functions `Sum`, `Product`, and the built-in "constant" `Infinity`. The basic syntax is `Sum[i^2,{i,1,5}]`.)

   (a) $\sum_{k=1}^{10}\frac{1}{k}$ (exact and numerical evaluation)
   (b) $\sum_{k=1}^{n}2^{-k}$

(c) $\sum_{k=0}^{\infty} x^k$

(d) $\sum_{k=0}^{\infty} \frac{x^k}{k!}$

(e) $\prod_{k=0}^{n} \left(1 + x^{2^k}\right)$

TIP: The functions Sum, Product, all have the same basic syntax, as do the functions Integrate, NIntegrate, Plot, Table below. In all of these cases, the first argument specifies the summand (or function to integrate or plot, or value in a table), and the second argument specifies a range using a curly brace expression such as {k,1,5} (meaning that k ranges from 1 to 5). Make sure to practice this syntax and get comfortable with it.

3. **Integrals.** Use Mathematica to evaluate the following integrals: (Hint: The relevant functions are Integrate (for symbolic/exact integration) and NIntegrate (for numerical integration). Both functions follow the syntax of Sum and Product.)

   (a) $\int_1^5 \log x \, dx$ (exact and numerical evaluation)

   (b) $\int_{-\infty}^{\infty} e^{-x^2} dx$ (exact and numerical evaluation)

   (c) $\int_0^5 e^{-x^2} dx$ (numerical evaluation)

   (d) $\int \log x \, dx$ (indefinite integral)

4. **Plots.** Plot the following functions, using the Plot function. The syntax is the same as that of Integrate.

   (a) $\log x$ from $x = 1$ to $x = 5$.

   (b) $\sin x$ from $x = 0$ to $x = 2\pi$.

   (c) $x^3 e^{-x^2}$ from $x = -5$ to $x = 5$.

5. **Tables (lists) of values.** The standard way to construct lists is via Table command, which has the same syntax as Sum, Product, Integrate, Plot.

TIP: Do not use loops (For, While, etc.) in Mathematica. In contrast to other programming languages, there are almost always built-in functions that achieve the same result as a loop, but are much more efficient, and easier and quicker to code.

   (a) The first 10 squares (i.e., $1, 4, 9, \ldots, 100$).

   (b) The first 10 powers of 2, starting with 2^0.

   (c) The numerical values of the first 10 partial sums of the series $\sum_{k=1}^{\infty} 1/k^2$.

   (d) A list of plots of the function $\sin(2kx)$, over the range $0 \le x \le \pi$, for $k = 1, 2, 3, 4$.

6. **Defining functions.** A function definition is of the form `f[n_]:=...`, with an underscored variable, `n_`, on the *left* side of the definition (but not on the right!) and a colon/equal sign (:=) between the left and the right side. Multivariable functions can be defined analogously.
   Examples: `f[x_]:=x^2+1`, `f1[x_]:=Integrate[Log[t],{t,1,x}]`, `squaresSum[x_,y_]:=x^2+y^2`.
   Recall that names for your own functions should be in lower case.
   In each of the following exercises, define the given function, and test your definition by evaluating the function at a few arguments.

   (a) A function, `partialsum[n_]`, that computes the n-th partial of the series $\sum_{k=1}^{\infty} 1/k^2$ *numerically.*

   (b) A function, `piapprox[n_]`, which gives π to n digits.

   (c) A function, `sinPlot[a_]`, which plots the function $\sin(ax)$ over the interval $0 \le x \le 1$.

   (d) A function, `norm[x_,y_]`, that outputs $\sqrt{x^2 + y^2}$, the norm of the vector (x, y).

Project: Approximating e by $\left(1+\frac{1}{n}\right)^n$

We know from calculus that $(1+1/n)^n$ converges to e. Let $\epsilon_n = e - (1+1/n)^n$ be the error in this approximation. Then ϵ_n goes to zero as $n \to \infty$.
The goal of this project is to confirm this behavior experimentally, and to (experimentally) determine the rate at which ϵ_n approaches 0.

1. **Define functions.** First let's define functions, `approx[n_]`, and `approxerror[n_]`, to denote our approximation $(1+1/n)^n$, and the approximation error ϵ_n.
   Hints: Since we are interested in *numerical* (instead of exact) computations, use the `N[...]` function in these definitions.

   Test your definitions by evaluating it at a few n-values.

2. **Generate tables of values.** Next, use the `Table` function to generate lists of values ϵ_n to see if we can spot a pattern.

   Do this first for a small range of n (e.g., from 1 to 50, with step size 1), then try a larger range with a correspondingly larger step size (e.g., $n = 1000$ to $n = 50000$ with steps of 1000).

3. **Plots the values.** For further insight plot the list of values obtained by wrapping the list inside `ListPlot` or `ListLinePlot`. To make the code more readable, give the lists appropriate names (e.g., `list50` for the first 50 values, or `list1000` for the first 1000 values), so that you can do `ListLinePlot[list50]`, etc.

4. **Experiment and make conjecture.** Examining the data leads one to suspect that the error, ϵ_n, decreases at a rate proportional to $1/n$. To study this further, it is natural to consider the product $n\epsilon_n$. Using the `Table` command, generate tables of values of this product and convince yourself that it indeed seems to converge to a certain constant, with a numerical value of roughly 1.3. This means that ϵ_n behaves like c/n, as $n \to \infty$, were c is this constant.

5. **Finding the constant.** As a final challenge, try to find/guess this constant. There are two great tools for this: The first is WolframAlpha, `www.wolframalpha.com`. A second, and much more powerful, tool is the "Inverse Symbolic Calculator" at `https://isc.carma.newcastle.edu.au/standard` (Google), one of the most amazing online resources in mathematics! To use these tools, you'll need to know the constant to sufficient accuracy (try to get at least 5 digits), so do the above calculation with large enough n-values. (Taking $n = 10^5$ should be no problem on any computer; $n = 10^6$ might still work, depending on the machine.)

6. **Make further conjectures.** To carry this experiment further, one can examine the difference between ϵ_n and c/n. A similar analysis as above shows very convincingly that this difference is very close to d/n^2, for a certain constant d, suggesting that one has the more precise approximation $\epsilon_n \approx \frac{c}{n} + \frac{d}{n^2} + O\left(\frac{1}{n^3}\right)$, where d and d is another constant.

Remark: The point of this exercise is the discovery of an asymptotic relation through *numerical* experimentation. Later we'll learn how to use Mathematica's *symbolic* capabilities to quickly obtain expansions of the above type.

Reminders: Definition of Functions

Top things to know about definitions of variables and functions

1. **Always use lower case names for your own variables and functions.** *A capital letter inside the name is okay; for example, `plotHarmonic` would be fine, but `PlotHarmonic` should be avoided.*

2. **Pay attention to the colors in the displayed command; if something looks out of the ordinary, there is probably a syntax error.**

3. **Press Shift-Enter after each definition to activate it.** *Notice the change in color of the function name; this indicates that the function is now available for use.*

4. **After making a definition, test it out by evaluating it at a few sample arguments.**

5. **Break up complex commands by defining functions for intermediate steps.** This makes the code easier to read and work with, and it makes it easier to locate errors. For example, to plot a list of values (e.g., partial sums of the harmonic series), define first a function, say `values[n_]`, whose output is the list of the first n partial sums, then define a function, say `plot[n_]`, that takes this list as input and outputs a plot. Test each of these functions immediately by evaluating them at a few n-values.

6. **Use `Clear[...]` to "clear out" old definitions.** If you have previously defined a function by the same name, defining it again may cause conflicts and hard-to-diagnose errors. To avoid this, clear out your old definitions as follows: `Clear[plotHarmonic]`, followed by the new definition, `plotHarmonic[n_]:=...`.
Conflicts among existing definitions can arise even if the old definition has been deleted from the notebook, and also if the definitions are in separate notebooks.

Resources

- **Elementary introduction to the Wolfram language.** Free interactive online tutorial at `www.wolfram.com`. (Just google the title.) Very basic, Comes with lots of exercises, all with solutions. **Highly recommended for beginners as a complement to these workshops. Focus on the first 6 sections (covering elementary arithmetic, functions, lists).**

- **Fast Introduction for Programmers**: Short interactive tutorial, available at `www.wolfram.com`. (Just type the title, "Fast Introduction", into the search box.) There is a version for Python programmers and another for Java programmer, but you don't have know either of these languages to get something out of the tutorial. *Focus in particular on the sections "Lists", "Iterators", "Assignments", and "Function Definitions".*

3 Lists, Random Functions, and Random Walks

A "list" is the most important, and most useful, data structure in Mathematica. Make sure to master this concept, and learn how to use lists effectively in your own work.

About this workshop

In this workshop, you will learn how to:

- Generate lists using the `Table` command.
- Perform arithmetic operations on lists ("list magic").
- Use list commands such as `Length` (number of elements), `Total` (sum of elements)), `Max`, `Min`, `Sort`, `Accumulate` (list of partial sums),
- Extract elements from lists using the double bracket notation `[[...]]`, and `First`, `Last`.
- Generate random numbers and lists of random numbers of various kind, using built-in random functions such as `RandomReal`, `RandomInteger`, and `RandomChoice`.
- Create and visualize random walks using these random functions along with `Accumulate`, `ListPlot`, `ListLinePlot`, and `Manipulate`.

Top things to know about lists in Mathematica

1. **A list is mathematically an ordered tuple; it is not a set.** A "list" in Mathematica is like an array in other languages, or an ordered tuple in mathematics. The order of the elements matters, and repeated elements are allowed.
2. **A list is denoted by curly braces, not round parentheses.** For example: `{1,3,1,2,5}`, not `(1,3,1,2,5)`.
3. **Vectors, points, and matrices are all represented by lists, and thus require curly braces, not round parentheses.** For example `{2,3}` represents the point (or vector) $(2, 3)$, while `{{1,2},{3,4}}` (a list of lists) represents the matrix $\left(\begin{smallmatrix}1 & 2\\ 3 & 4\end{smallmatrix}\right)$.
4. **The command to create a list of values is `Table`, not `List`.** While there exists a command called `List`, this is a low-level function that has a different purpose and that you won't need.
5. **The command to get the sum of the elements of a list is `Total`, not `Sum`.**

3.1 Generating Lists

1. **Explicit specification of list elements.** The simplest way to specify a list is by listing its elements, enclosed in curly braces. For example, `mylist1={1,3,1,2,5}` defines a list consisting of the five elements $1, 3, 1, 2, 5$, and assigns this list the name `mylist1`.
2. **Generating lists of consecutive inters: The `Range` command**

   `Range` allows one to quickly generate a list of integers, e.g., to use as a test list. Here are some examples:

   - `Range[5]` (generates the list $\{1, 2, 3, 4, 5\}$)
   - `Range[2,5]` (generates the list $\{2, 3, 4, 5\}$)
3. **The Table Command.** This is, by far, the most common and must useful way to generate a list. The `Table` command has the form `Table[generalterm,range]`, with two arguments; for example, `Table[i^2, {i,1,10}]`.

- The first argument, `generalterm`, denotes the **general term** in the list you want to create (`i^2` in the above example).
- The second argument, `range`, denotes the **range** you want to iterate over to create the list (`{i,1,10}` in the above example).

Specifying the range in `Table`, `Sum`, and similar commands

The `range` in a `Table` is itself a list, and can be specified in several forms:

- `{iterator,start,end}`; e.g., `{i,1,10}` means that i ranges over the numbers $1, 2, \dots, 10$.
  Shortcut: `{i,10}` is equivalent to `{i,1,10}`.
- `{iterator,start,end,increment}`; e.g., to get the odd numbers from 1 to 11 (inclusive), use the range specification `{i,1,11,2}`. to get the list $0, 0.1, 0.2, \dots, 0.9, 1$, specify 0.1 as increment: `{x,0,1,0.1}`.
- `{iterator,{value1,value2,...}}`, e.g., `{i,{2,4,8,16}}` means that i ranges over all values in the set $\{2, 4, 8, 16\}$. **Notes:**
  - The set of values in this specification is itself a list and must be enclosed in braces.
  - The values specified in the list here don't have to be numerical; they can be pretty much anything: functions (e.g., `{f,{Cos,Sin,Tan}}`), options in Mathematica (e.g., `{color,{Red,Blue,Black}}`, etc.

3.2 Generating random numbers and lists of random numbers

In contrast to other programming languages, Mathematica's built-in random number functions `RandomReal`, `RandomInteger`, `RandomChoice` can generate *lists* of random numbers without needing loops. In fact, this extends to *lists of any dimension*, e.g., lists of points (representing a point as a list of its coordinates). **This capability makes doing random simulations, and generating random walks of all kinds, extremely easy in Mathematica, often with a one-liner. The project below illustrates this.**

- **Creating single random numbers**
  - `RandomReal[{-1,1}]` (generates a random *real* number in the interval $[-1, 1]$, with uniform probability)
  - `RandomInteger[{1,6}]` (generates a random *integer* in $\{1, 2, \dots, 6\}$, i.e., simulates a roll of a die)
  - `RandomChoice[{-1,1}]` (generates a random number in the set $\{-1, 1\}$, i.e., picks -1 and 1 with equal probability) (The difference to `RandomReal` and `RandomInteger` is that with the `RandomChoice` command the elements 1 and -1 listed in `{-1,1}` are the *only* choices, whereas with the other two commands, these elements represent the endpoints of a *range* from which the random numbers are chosen.)
- **Creating lists of random numbers.** To get a list of random numbers, just provide a second argument to the above functions that gives the number of random numbers you want. You can even generate matrices or multi-dimensional lists (e.g., 10 pairs of random numbers, or two lists of 10 random numbers) by specifying the dimensions enclosed in curly braces (e.g., `{10,2}` for 10 pairs, and `{2,10}` for 2 lists of 10). Here are some examples:
  - `RandomInteger[{1,6},10]` (generates a list of 10 random integers from $\{1, 2, \dots, 6\}$, i.e., simulates 10 rolls of a die). *This is equivalent, but shorter, than using the* Table *command to construct the list:* `Table[RandomInteger[{1,6}],{i,1,10}]`.
  - `RandomReal[{-1,1},10]` (generates a list of 10 random real numbers, chosen from the interval $[-1, 1]$)
  - `rlist[n_]:=RandomChoice[{-1,1},n]` (defines a function, `rlist`, that generates a list of n random ± 1's)
  - `RandomChoice[{-1,1},{10,2}]` (generates a 10×2 matrix of random numbers in $\{-1, 1\}$ from the interval $[-1, 1]$, i.e., list of 10 *pairs* of such numbers. This is equivalent to generating 10 two-dimensional *vectors*, or points in the plane, with random coordinates ± 1.)

  – `RandomChoice[{-1,1},{2,10}]` (generates a 2×10 matrix of random numbers in $\{-1, 1\}$, i.e., 2 lists of 10 such numbers)

3.3 Exercises

The List Manipulation Documentation Page

The documentation for list operations is scattered among dozens of individual pages. The single best starting point is the page "List Manipulation." (Type the title into the search box on the Wolfram Documentation page.)
Keep this page open in a separate Window so you can refer to it as needed. Click on the section headers (e.g. "Math and Counting Operations") for more detailed information on a particular topic.

1. **Basic list operations:** Define a sample list, e.g., `list1={3,1,4,1,5,9}`, or `randlist=RandomReal[{0, 1},5]`, to practice with. Then use Mathematica to find:
   (a) its *number of elements* (`Length`)
   (b) its *sum* (`Total`)
   (c) its 3rd element (`list1[[3]]`)
   (d) its *first element* (`First`)
   (e) its *last element* (`Last`)
   (f) its *mean* (`Mean`)
   (g) its *second last element* (`list1[[-2]]`)
   (h) its *maximal* element (`Max`)
   (i) the list in *reverse* (`Reverse`)
   (j) the *sorted* version of the list (`Sort`)
   (k) the list of *partial sums* (`Accumulate`)

2. **List magic**. A neat thing about lists is that most ordinary arithmetic operations can be applied to lists and produce the expected results *even if it doesn't make sense from a strict mathematical point of view.* That is, the output is the list obtained by carrying out the arithmetic operations element-wise).

   For example, given a list `list1` (e.g., the one above), try out the following:
   (a) `list1+1` (adds the constant 1 to each element of the list)
   (b) `list1^2` (outputs the list of the squares the elements of the list)
   (c) `1/list1` (outputs the list of the reciprocals of the elements)
   (d) `2^list1` (outputs the list of 2^x, where x runs through the original list)
   (e) `list1-Mean[list1]` (normalizes the list by subtracting its mean)

3. **Creating lists with the `Range` function.** Recall that `Range[n]` gives the list $\{1, 2, \ldots, n\}$. Use this function along with a bit of list magic to create:
   (a) The list of the first 10 powers of 2, i.e., the list $\{2^1, 2^2, \ldots, 2^{10}\}$.
   (b) The list of the first 10 cubes, i.e., the list $\{1^3, 2^3, \ldots, 10^3\}$.
   (c) A function, `harmonic[n_]`, that outputs the list $\{1, 1/2, \ldots, 1/10\}$. (Test this function for a few n-values.)
   (d) A function, `harmonicSum[n_]`, that gives the sum of this list, i.e., the n-th partial sum of the harmonic series, $\sum_{i=k}^{n} 1/k$.
   (e) A list of the first 10 of these partial sums, i.e., the list $(1, 1 + 1/2, \ldots, 1 + 1/2 + \cdots + 1/10)$.

4. **More list magic: Creating a mean and variance function.** Define two functions, `mean[x_]`, and `var[x_]`, that take a *list*, denoted by the variable `x`, of numbers as input and whose output is the mean, resp. variance, of these numbers. For example `mean[{1,2,3}]` should give 2 as output. (Recall that the variance of a list of numbers $\{x_1, \dots, x_n\}$ is defined as $(1/n)\sum_{i=1}^n (x_i - \mu)^2$, where $\mu = (1/n)\sum_{i=1}^n x_i$ is the mean.)

   Note: While there exist built-in functions for mean and variance that accomplish the same, the point of this exercise is to avoid these functions and instead come up with a short and elegant definition for such functions using only basic list operations and list magic.

Advanced list magic: Coding the chi-squared goodness of fit test.
The chi-squared test is a statistical test to determine how well a a set of observations fits a given finite distribution. The inputs to the test are (1) a finite probability vector $(p_1, \dots, p_k)$ (so that $p_i > 0$ for all i and $\sum_{i=1}^k p_i = 1$) and (2) a vector $(n_1, \dots, n_k)$ representing the number of observations for each category. The output is a “chi-squared” value, defined as follows:

$$\chi^2 = \sum_{i=1}^k \frac{(O_i - E_i)^2}{E_i},$$

where O_i is the actual number of observations for category i, and E_i is the expected number of observations for this category. Using the notations (1) and (2), we have $O_i = n_i$, and $E_i = np_i$, where $n = \sum_{i=1}^k n_i$ is the total number of observations.
Create a function, say `chisquared[prob_,counts_]`, that has the two lists (i.e., vectors) (1) and (2) as inputs, and which outputs the χ^2 statistic defined above (as a *numerical* value). For example `chisquared[{0.5,0.2,0.3},{11,5,6}]`, should output the chi-squared statistic for the probability vector $(0.5, 0.2, 0.3)$ and the count vector $(11, 5, 6)$.
There are two basic approaches to coding this function: (1) A relatively straightforward approach using the `Sum` command. (2) A slick approach that avoids `Table` altogether and uses instead a combination of list magic tricks.

Project: Generating and visualizing random walks

This project illustrates the power of Mathematica's random functions, list functions (in particular `Accumulate`), and plot functions (in particular, `ListLinePlot`) to easily generate random walks of various kinds.
The following steps guide you through creating a simple, but powerful and impressive looking random walk visualization. To make the code easier to read, debug, and generalize, you'll create functions that represent the individual steps in this process. (In principle, the whole thing could be coded as a one-liner, but this makes it hard to read.)

1. **Generate a list of random numbers in $\{-1, 1\}$.** Define a function, `rlist[n_]`, that generates a list of n random numbers in $\{-1, 1\}$. Check it on a few n-values to make sure it works as expected.

2. **Generate the partial sums of this list.** Define a second function, `rsum[n_]`, that generates the *partial sums* of such a random list (i.e., the sums $S_k = \sum_{i=1}^k X_i$, $k = 1, 2, \dots, n$, where the X_i's are the random numbers). Again, check it on a few n-values to make sure it works as expected. (Hint: `Accumulate`)

3. **Plot these partial sums (i.e., plot S_k as a function of k).** Define a function, `rplot[n_]`, that plots a 1-dimensional random walk with n steps $+1$ and -1). (Hint: `ListLinePlot`)

4. **Two-dimensional random walks.** Now do the same to get plots of two-dimensional random. The only difference is in the first step: Define a function, `rlist2D[n_]`, that generates a list of n *pairs* of random numbers in $\{-1, 1\}$. As always, check your definition on a few n-values. (Hint: To get *pairs* of random numbers use a construct like `RandomChoice[{-1,1},{n,2}]`, which creates an $n \times 2$ matrix of random ± 1's.)

5. **Add options to ListLinePlot to make it prettier.** For the 2D version, add the option `ColorFunction->"Rainbow"` to the `ListLinePlot` command. This option "colorizes" the random walk for a nice visual effect. (Simply replace `ListLinePlot[list]` by `ListLinePlot[list,option1,option2,...]`, where `option1`, `option2`, etc. are your options. There are many other options available; see the documentation for `ListLinePlot`.

6. **Using Table for experimentation:** A very effective use of `Table` is to generate multiple plots of the above type in a single command: For example, suppose `plot1[n_]:=ListLinePlot[...]` is a function that generates one of the above plots. Then try the following:

   - `Table[plot1[10^3],{i,10}]` (generates 10 random walk plots with $n = 10^3$ steps each).
   - `Table[plot1[n],{n,2^Range[8,15]}]` (generates random walk plots for $n = 2^8, 2^9, \dots, 2^{15}$)

7. **Create a one-line app:** Replace `Table` by `Manipulate` in the last command to turn this into an interactive "app". (To run it as an animation, click on the "plus" (+) icon on the top right of the panel and select "Autorun.") Play with it, expand the range, and add some features. For example, add a `SeedRandom` function to generate a fresh set of random walks each time the seed is changed. (To do this, add (1) `SeedRandom[s];` at the beginning of the `Manipulate`, and (2) `{s,1,100,1}` (or similar) at the end.)

4 Functions, and Random Simulations

About this workshop

In this workshop, you will learn how to:

- Define functions of various types.
- Use `Nest` and `NestList` to iterate functions.
- Define **pure** functions, using the `#` and `&` notations.
- Use `Select`, along with the `#` and `&` notations, to construct sublists, and to compute frequencies of list elements satisfying certain conditions.
- Perform random simulations using `RandomReal` and similar built-in functions.
- Use statistical functions, such as `Median`, `Mean`, `StandardDeviation`, and `Histogram`.

4.1 Defining a function

The basic form of a function definition is `f[x_]:=...`, where `x_` (note the underscore!) is the variable. when defining a function. The function name, `f` in the above case, can be pretty much any alpha-numeric string (though you should avoid strings beginning with a capital letter), and the right-hand side of the definition can be pretty much any Mathematica expression. When calling up the function, use `f[x]` (or `f[1]`, `f[a]`, etc.) *without the underscore.*

Here are some examples of function definitions:

- `f[x_]:=x^2+2*x+3`
- `rnumbers[n_]:=RandomReal[{0,1},n]` (generates n random real numbers in $(0,1)$)
- `rpoints[n_]:=RandomReal[{0,1},{n,2}]` (generates n *pairs* of random real numbers in $(0,1)$, or equivalently, n random points in the unit square)
- `squareSum[x_,y_]:=x^2+y^2`.
- `squareSum2[vec_]:=Total[vec^2]` (takes a *list* `vec` as argument, squares the list componentwise, then adds up the components (`Total`))
- `gauss[x_]:=NIntegrate[(1/Sqrt[2*Pi])E^(-t^2/2),{t,x,Infinity}]` (computes the Gauss integral $\int_x^\infty (1/\sqrt{2\pi})e^{-t^2/}$
- `binomProb[n_,k_,p_]:=N[Binomial[n,k]p^k(1-p)^(n-k)]` (computes the probabilities in the binomial distribution). (Note the `N` function to force numerical computation.)

Colon-equal (:=) versus equal (=) in definitions

A colon-equal sign (:=) in a definition of a variable or function denotes a **delayed assignment**: The definition is evaluated at the time it is *applied*, instead of right away. For functions (of one or more variables), this is always what you want. When the function, say $f(n)$, is applied with a specific n-value, you want the value of the function at this particular value.

For assignments of *static* variables (i.e., variables that don't change values), there is no need for delayed evaluation, so using an equal sign in the definition is enough; for example, `piapprox=3.14`.

An interesting special case is the assignment of variables to random quantities: For example, consider `rand1=RandomReal[{0,1}]` and `rand2:=RandomReal[{0,1}]`. Both definitions give a random real number in $[0,1]$, but the first definition, `rand1`, produces the same random number every time it is used, whereas the second definition, `rand2`, produces a different random number each time since each time it is applied, it is evaluated again. To see the effect of this, try `Table[rand1,{i,10}]` and `Table[rand2,{i,10}]` and compare the two outputs. Sometimes you want the first behavior, while at other times, you want the second behavior.

Tips and pitfalls in function definitions

Errors in function definitions can be extremely hard to diagnose, so you have to be very careful with such definitions and make sure to use the correct syntax. Here are some tips on avoiding common errors.

1. **Always use lower case names for your own variables and function names.** This is to avoid conflicts with built-in variables and functions. The first letter should always be lower case, though other letters can be upper case. For example, `binomialSum` would be perfectly fine as a function name.

2. **Never use underscores in function names.** Underscores have a special meaning in Mathematica and should not be used in function names. For example, `binomial_Sum` might cause strange errors; use `binomialSum` instead. (Numbers are okay; for example, `binomialSum3` would work fine.)

3. **Avoid re-using function names or use "Clear" before re-using or re-defining the names.** A common situation is when you start out with a simple function definition, e.g., `f[x_]:=x^2+1`, and then want to generalize it by introducing a second variable, e.g., `f[x_,n_]:=x^n+1`. This will give an error. To avoid this, use either a different name for the more general function (e.g., `ff[x_,n_]:=x^n+1`), or "clear" the function from memory with the command `Clear[f]` before redefining the function.

4. **Pay attention to visual clues such as the colors displayed.** An expression that is highlighted/colored in red or blue typically indicates some problem, e.g., a mis-typed name of a function, or a mismatched parentheses, etc.

5. **If the output is identical to the input, something is likely wrong, even if there is no error message.** Usually, this indicates that Mathematica doesn't know how to evaluate the expression, e.g., because of a mistyped function name or constant (e.g., `e` instead of `E`), or because the expression can only evaluated numerically, but you forgot to explicitly ask for a numerical evaluation.

4.2 Exercises

1. **Functions of lists.** Define a function `norm`, that takes a *vector* (i.e., a one-dimensional list) as input and outputs its norm (length). For example, `norm[{1,2,3}]` should produce $\sqrt{1^2+2^2+3^2}$, `norm[{1,2,3,4}]` should produce $\sqrt{1^2+2^2+3^2+4^2}$, etc. Notes:

   - The above approach using lists as function argument is useful in many contexts: Its main advantage is that it works for vectors of arbitrary dimensions. Without lists, one would need to define a separate function for each dimension, e.g., `norm2[x_,y_]`, `norm3[x_,y_,z_]`, etc.
   - Make sure to provide the arguments to this function as a list, encloed in a pair of braces, `{}`, i.e., `norm[{1,2,3}]`, not `norm[1,2,3]`.

2. **Iterating functions with `Nest` and `NestList`:** Functions can be iterated using the `Nest` and `NestList` commands. For example, `Nest[Cos,0,3]` produces the 3rd iterate of cos with initial value 0, while `NestList[Cos, 0,3]` generates the list of the all k-th iterates up to $k=3$.

   Use `NestList` to determine experimentally the limit when iterating the cosine function, with 0 as initial value, i.e., the value of $\cos(\cos(\cos(\ldots 0)))$. (Hint: Use the `N` function to force Mathematica to carry out the computations numerically.)

3. **"Pure" function:** To iterate the function $x \to x^2$, we could introduce a definition, say `square[x_]:=x^2`, and then use `Nest[square,2,5]` as before.

   A shorter, and more elegant approach is to put the definition of this function inside the `Nest` or `NestList` commands: `NestList[#^2&,2,5]`. Here `#^2&` denotes a **pure** function, that takes a number, denoted by `#`, and squares it; the ampersand, &, is a required part of the pure function syntax and denotes the end of the function definition. **IMPORTANT: Don't forget the ampersand symbol, `&`, here!**

   Using this approach, investigate the iterates of the *logistic map* $x \to cx(1-x)$, where c is a constant ≤ 4. More precisely:

   (a) Define a function `logistic[n_,c_,x0_]` that generates n iterates of this map with initial value `x0`.

   (b) Test out your definition on a few examples: For instance, `logistic[50,3,.4]`, `logistic[50,2,.4]` `logistic[50,4,.4]`. Also try random initial values, e.g., `logistic[50,4,RandomReal[{0,1}]`.

   (c) Use the `ListLinePlot` and `Histogram` functions to plot the lists produced by the `logistic` function.

4.3 Random Simulations

Mathematica's random functions (`RandomReal`, etc.), along with list functions (`Table`, etc.) make it easy to code random simulations in order to come up with conjectures, visualize known results. The following projects illustrate this.

Project: Simulating sums of random numbers: Watching the bell curve taking shape.

One of the fundamental results in probability is the Central Limit Theorem which says (roughly) that, whenever a large number of independent random quantities are added up, the resulting quantity tends to have normal (i.e., "bell curve") distribution. A nice way to see the normal curve taking shape is by performing a large number of simulations of sums of n random numbers in $[0, 1]$, and observing the shape of the resulting distribution as n gets larger and larger. The following sequence of exercises illustrates this:

1. Define a function `rsum[n_]` that generates n random numbers in $[0, 1]$ (`RandomReal[1, n]`) and outputs their sum (`Total`). Test this out for small values of n.
2. The result of a single run of `rsum[n_]` is a sum of n random numbers. To see how these random sums are distributed, we need to repeat such a run a large number of times and keep track of the outputs. The `Table` command makes this a breeze: For example, `Table[rsum[10],{i,1,3}]` generates 3 sums of 10 random numbers. (The index i is a dummy variable.)

   Now generalize this by defining a function, `sim[k_,n_]`, that carries out k runs of `rsum[n]` and produces a *list* of the resulting k random sums. Test this out for small values of n and k to make sure it works as expected.
3. Apply `Mean` and `Median` to the list produced by `sim[k,n]`. Test this out with small values of n (say, $n \le 5$) and moderate or large values of k (say, $k = 10$, $k = 100$, $k = 10000$). Do the results make sense? (The mean and median of a sum of n random numbers in $[0, 1]$ should be very close to $0.5n$.)
4. Apply `Histogram` to the list produced by `sim[k,n]` to see the shape of the distribution. You'll need to take k fairly large, e.g., $k = 10^4$ or $k = 10^5$, to get good results. Instead of `Histogram`, you might also try `SmoothHistogram`, which generates a "smooth" version of the histogram that looks more like a real probability distribution.
5. Finally, using `Table`, generate a list of these histograms for $n = 1, 2, \ldots, 6$, say. Observe how the shape of the histograms approximates a normal (bell-shaped) curve as n gets larger.

Project: Computation of π via random simulations

As a fun application of random simulations here is a way to numerically compute π: Generate a random point (x, y) in the square $[0,1] \times [0,1]$. The probability that this point satisfies $x^2 + y^2 < 1$ is equal to the area of the part of the unit disk inside that square, i.e., $\pi/4$. Performing a large number of such simulations yields an approximation to this probability, and hence an approximation to $\pi/4$. (This method is known as the Monte Carlo method; it provides a way to numerically compute areas and volumes of very general regions.)
In Mathematica, this is particularly easy to implement, using arithmetic operations on lists:

1. First generate a list of n sums of two squares of random numbers in $[0,1]$, $x^2 + y^2$. (Hint: Remember that you can "square" a list and add two lists.).

2. Next, we want to extract the terms in this list of sums of random squares that are < 1: The key is to apply the `Select` function to extract the list values that satisfy our condition, as follows: `Select[list,#<1&]`, where `list` is our list (in this case, the result of `rlist`), and the second argument, `#<1&`, is the criterion for our selection, coded in much the same way as a pure function. **Don't forget the ampersand, &, at the end of the condition.** Test this out first with an explicit list, e.g., `Select[{.2,.5,1.2,3},#<1&]`, then apply it to the list produced in the previous step: Define a function `goodlist[n_]` that extracts the "good" terms (i.e., those < 1) from our original list `rlist[n]`.

   Test it out on a few n-values.

3. Next, define a function `approxPi[n_]` that outputs $4g/n$, where g is the number of terms in `goodlist[n]`, and n is the number of terms in the original list. (Hint: `Length[list]` gives the number of terms in `list`.)

   Again, test this out to make sure it works.

4. Finally, tabulate the approximations for an appropriate sequence of n-values, e.g., 10^k, for $k = 1, 2, \ldots, 6$. If your code is correct, these should approach the actual value of π.

Project: The probability that the sum of n random numbers in $[0, 1]$ is < 1.

Here we try to solve experimentally the following question:

> *Suppose n numbers are chosen at random from the interval $[0, 1]$. What is the probability, $P(n)$, that their sum is < 1?*

This question can be answered theoretically using multiple integrals, but it is more fun to determine the answer experimentally, using Mathematica's random number functions.

1. To start with, create a function, `sim[k_,n_]`, to generate a list of k sums of n random numbers in $[0, 1]$. Check that the function produces the expected results.
2. Next, use the `Select`, `Length`, and `N` functions as above to extract the list values that are < 1, and compute their numerical proportion. Call this proportion `prob[k_,n_]`.

   Test the `prob` function with explicit values of n and k to make sure it works. (Take a small n, e.g., $n = 4$, and a large k, e.g., $k = 10^5$).

   Try this out first on an explicitly specified list, then apply it to `sim` function, and test the results with small k and n values.
3. Finally, tabulate the results for the first few n-values, and try to guess the formula for `prob[n]`. Take k larger as needed to get sufficiently accurate results.

5 The Map Command

The `Map` command, and its shortcut version, `/@`, is one of the most powerful and versatile functions in Mathematica. Using `Map` can greatly simplify the code, make it more readable, and make it run more efficiently. Many of the code examples at the Wolfram Demonstrations site and elsewhere involve some version of Map (usually in the shortcut form, `/@`). **If you want to read and understand this code, and to do some programming with Mathematica on your own, you should become thoroughly familiar with this command.**

5.1 Map in a Nutshell

The `Map` command applies a function to each element of a list, and it returns a list of the outputs. The basic syntax is one of the following:

- `Map[function,list]` (explicit form)
- `function /@ list` (shortcut form, using prefix notation)

where `function` is some function and `list` is some list. The function can be pretty much anything: a **built-in** function like `Cos` or `Length`; a **user-defined** function (e.g., `average`); a **pure** function (e.g., `#^2&`).

5.2 Examples

Start by generating two test lists:

- A simple *numerical* list, e.g., `list1=RandomInteger[{1,6},10]`
- A simple *2-dimensional* list, e.g., `list2=RandomReal[1,{10,2}]`.

Then try the following commands and observe the results:

1. `Map[Cos,list1]` (generates the cosines of the list elements).
   This is equivalent to `Table[Cos[x],{x,list1}]`, but shorter and more elegant.
2. `Cos /@ list1` (same, but using the `/@` notation)
3. `Map[{Cos[#],Sin[#]}&,list1]` (generates a list of pairs $(\cos x_i, \sin x_i)$, where x_i runs through the elements of the list). (Note the ampersand `&` notation at the end of a pure function.)
4. `{Cos[#],Sin[#]}& /@ list1` (same using the `/@` notation).
5. `Map[#[[1]]&,list2]` (Outputs the list of x-coordinates of the elements in `list2`. Note that each `#` represents a *list* of two elements (namely x_i and y_i), and to extract the first of these elements, x_i, you need to use double brackets: `#[[1]]`)
6. `Map[#[[1]]^2+#[[2]]^2&,list2]` (Outputs $x_i^2 + y_i^2$ for each element (x_i, y_i) in `list2`.)
7. `Map[Total[#^2]&,list2]` (same, but shorter and more elegant, using a bit of list magic)
8. `Total[#^2]& /@ list2` (same, but even shorter, using the `/@` shortcut)
9. `Map[Sort,list2]` (sorts each of the *sublists*, i.e., each pair (x_i, y_i) in the original list is sorted so that the first element is the smaller one and the second element is the larger one).
10. `Map[Length,Subsets[{1,2,3,4}]]` (The `Subsets` command generates a list of all subsets of $\{1, 2, 3, 4\}$, `Length` gives the number of elements of these subsets.)
11. `Length /@ Subsets[{1,2,3,4}]]` (same using the `/@` shortcut)

5.3 Exercises

Start by generating some test lists:

- A simple *one-dimensional* list `list1` of positive real numbers or integers, such as the one above.
- A simple *two-dimensional* list `list2`, whose elements are pairs of positive real numbers, such as the one above.
- A 2-dimensional list (i.e., a list of lists), whose elements are lists of positive integers or real numbers, not all of the same length; e.g. `list3={{4,3,2,4},{3,4},{4,3,4},{2,3}}`
  For example, such a list might represent the list of grades of students at a university. Each sublist represents the grades of a particular student. Since the number of courses taken varies from student to student, we need the sublists to have variable lengths to model this situation.

1. **Computing averages:**
   (a) Write a function `avg[list_]` that takes a list, denoted here by `list`, and computes its *numerical* average (Hint: Use `Total`, `Length`, and `N`). Test this function on a *single* (one-dimensional) list.
   Remarks: (1) Using a suggestive name like "list" as the argument in the function definition serves as a reminder that you have to apply the function to a "list" rather than a scalar quantity. However, this is just a "dummy variable", and you can give it pretty much any other name as long as it doesn't conflict with predefined variables.)
   (2) The built-in function, `Mean`, accomplishes the same, but for practice try to code this without using `Mean`.)
   (b) Apply this function to the 2-dimensional list `list3` above, using the Map function. The result should be a list of the averages of the sublists.
   (c) Rewrite the latter as a standalone one-line command, using the *pure* function and `/@` notations, i.e., something of the form `.../@ list3`.

2. **Scaling lists:**
   (a) Write a function `scale[list_]` that takes a list and scales it so that the sum of the elements add up to 1. For example, `{1,3}` should turn into `{0.25,0.75}`, `{1,7,2}` should turn into `{0.1,0,7,0.2}`
   (b) Now use Map to apply this function to a two-dimensional list like `list3`.
   (c) Rewrite the latter as a standalone one-line command, using the *pure* function and `/@` notations.

3. **Converting lists of real numbers to points on the number line.** Suppose we have a list of real numbers $x_1, \ldots, x_n$, and we want to draw these points on the real axis. To this end, we first need to convert each x_i to the point $(x_i, 0)$ on the plane. Use `Map`, to write a function that accomplishes this task. That is, given a list of real numbers, such as `{0.3,0.7,0.59,3.14}`, it should output a list of points with the given numbers as x-coordinates and with y-coordinates 0. e.g., in the above example, we want as output `{{0.3,0},{0.7,0},{0.59,0},{3.14,0}}`.

4. **Converting complex numbers to points in the plane.**
   (a) Generate a test list of complex numbers, e.g., `complexlist=RandomComplex[1+I,10]` (generates 10 complex numbers with real and imaginary parts between 0 and 1)
   (b) Create a function, say `complexToPoint[z_]`, that converts a complex number, say $z = x + iy$, to the point (x, y). (Hints: Use the built-in functions `Re`, `Im`. Recall that in Mathematica a 2D point is a *list* of two elements, i.e., it must be written using curly braces.)
   (c) Use `Map` to apply this function to the list `complexlist` above.
   (d) Rewrite the Map function using a *pure* function instead of the explicit function `complexToPoint`, and the `/@` notation instead of an explicit `Map`. (This type of construct is very common. An example appears on the documentation page for `RandomComplex`. Click on "Applications.")

5. **Extracting first digits:**

   (a) Write a function, `firstDigit[n_]`, that takes a positive integer, and outputs its first digit. (Hint: use the built-in function `IntegerDigits`, which takes a positive integer as input and outputs its list of decimal digits. For example `IntegerDigits[347]` outputs `{3,4,7}`.)

   (b) Use this function along with Map to create a function, `firstDigitList[list_]`, that takes a *list* of positive integers as input and outputs the list of first digits. For example, given the input `{1,3,57,39, 213}`, the output should be `{1,3,5,3,2}`.

   Test this function on some sample lists, for example, `powerlist=2^Range[20]`.

   (c) Create a function, `firstDigitFrequencies[list_]` that takes a list of integers as input, and outputs a list of the *numerical* frequencies of first digits in a list of integers. The output should be a list such as `{0.1,0.13,...,0.21}` with 9 elements representing the frequencies of digits $1, 2, \dots, 9$.

   (Hint: Use a combination of `Count`, `Length`, `N` to get the frequencies of these digits.)

   (d) Apply the `firstDigitFrequencies` function to some large lists, e.g., the first 10000 powers of 2 or powers of 3, or the first 10000 Fibonacci numbers (`Fibonacci[n]`). Use `BarChart` to display the resulting frequencies.

Project: Infinitely nested squareroots and fractions

This project asks you to experimentally find the values of the infinitely nested expressions

$$A = \sqrt{1+\sqrt{1+\sqrt{1+\dots}}}, \tag{a}$$

$$B = \frac{1}{1+\frac{1}{1+\frac{1}{1+\frac{1}{1+\dots}}}}. \tag{b}$$

These expressions should be interpreted as limits of recursively defined sequences:

$$A = \lim_{n\to\infty} a_n, \quad a_{n+1} = \sqrt{1+a_n}, \quad a_1 = 1. \tag{a}$$

$$B = \lim_{n\to\infty} b_n, \quad b_{n+1} = \frac{1}{1+b_n}, \quad b_1 = 1. \tag{b}$$

One can evaluate these limits theoretically (a great exercise for Math 347, 444, and similar courses!), but for this exercise you should use Mathematica to determine the limits numerically to high accuracy, and then use the “Inverse Symbolic Calculator” mentioned in Workshop 1 to come up with a guess for the theoretical values.

Project: Generating Random Walks using `NestList`

In a previous workshop, we used `RandomChoice` or `RandomReal` together with `Accumulate` to generate random walks. This project illustrates a different approach, using iteration with the `NestList` function. The basic idea is that a random walk is simply the result of iteratively applying the function $x \to x + \ rstep$, where $rstep$ is a random step. In Mathematica, this iteration can be implemented using the `NestList` function. For example,

```
NestList[#+RandomChoice[{-1,1}]&,0,10]
```

generates a 10-step one-dimensional random walk starting at 0, with steps -1 and 1, each chosen with equal probability.

1. Test out the above function with a larger number of steps (e.g., 10^4 or 10^5), and use `ListLinePlot` to plot the random walk generated. Compare the result to plots generated using the earlier approach, with `Accumulate`. Is one approach significantly faster than the other?

2. Now generalize this function to create random walks of arbitrary dimensions, and with arbitrary sets of steps. Specifically, create a function, `rwalk[n_,start_,steps]`, where n is the number of steps, `start` the starting point, and `steps` the *list* of possible steps (which will be chosen with equal probability). In particular, this function should allow specifying the steps in an arbitrary manner. For example:

   - `rwalk[10,0,{-1,1}]` should produce the above one-dimensional random walk.
   - `rwalk[10,{0,0},{{-1,1},{1,1},{1,-1},{-1,-1}}]` should produce a two-dimensional random walk with 10 steps, randomly chosen among the four vectors $(1,1),(1,-1),(-1,1),(-1,-1)$.
   - `rwalk[10,{0,0},{{0,1},{1,0},{0,-1},{-1,0}}]` should produce a two-dimensional random walk with 10 steps, randomly chosen among the four vectors $(0,1),(1,0),(0,-1),(-1,0)$.
   - `rwalk[100,{0,0,0},{{0,0,1},{0,0,-1},{1,0,0},{-1,0,0},{0,1,0},{0, -1,0}}]` should produce a 3-dimensional random walk of 100 steps with steps of the form $\pm\mathbf{i},\mathbf{j},\mathbf{k}$, where $\mathbf{i},\mathbf{j},\mathbf{k}$ are the unit basis vectors in $\mathbb{R}^3$.

Project: Iterating the "sum-of-digit-squares" function

Suppose you take an integer, form the sum of the squares of its decimal digits to get another integer, then form the sum of squares of that integer, and keep going. The number 1 is a fixed point in this iteration, so if at some point in the process you reach 1, then you stay there. For example:

$$19 \to 1^2+9^2=82 \to 8^2+2^2=68 \to 6^2+8^2=100 \to 1^2+0^2+0^2=1 \to 1 \to \dots.$$

An interesting question is whether this process always eventually reaches the fixed point 1. The goal of this project is to explore this question experimentally. Proceed as follows:

1. Create a function, say `sumDigitSquares[n_]`, that takes a positive integer as input and outputs the sum of the squares of its digits. (Hint: `IntegerDigits`.) Test this function on some sample inputs to make sure it works as expected.

2. To examine what happens when you iterate this function, use `Nest` to create another function `sumDigitSquaresIterated[n_]`, that iterates this function a large number of times (100 should be enough) and outputs the results of these 100 iterations. Again, test this function on sample inputs to make sure it works.

3. To get a large sample of data, use Map to apply the `sumDigitSquaresIterated` function the first 10^4 or 10^5 positive integers (how do you get a *list* of these integers?). The output will be a very large list, so apply `DeleteDuplicates` or `Union` (both of these commands have the same effect of only listing unique elements) to get a (very short) list of "end values" reached after 100 iterations. Also, apply `Sort` to make it easier to spot patterns.

4. Experiment some more (e.g., by varying the number of iterations) to convince yourself that the list of possible end values does not change, and examine what happens if you apply the original `sumDigitSquares` function to these end values. With enough experimentation you should come up with a general conjecture on the behavior of the sum-of-digit-squares iteration. In particular, find all cycles that the sum-of-digit-squares iteration leads to.

Final challenge: Automate the cycle-finding process. Create a function, `findCycle[list_]`, that takes a finite list as input and tries to find its smallest "cycle", i.e., the shortest pattern that repeats itself. For example, in the list `{3,19,42,5,33,7,42,5, 33,7,42,5}`, the last four terms, $33, 7, 42, 5$, form a cycle (and it is the smallest such cycle), so the output should be the list consisting of those four terms, `{33,7,22,5}`. If a list has no non-trivial cycle (i.e., no repeating term), then the output should just be the original list. Use this function in conjunction with `NestList` to find all cycles in the sum-of-digit-squares iteration, and do the same for the sum-of-digit-cubes iteration (where the behavior is much more complicated).

6 Plotting and Interactive Visualization

About this workshop

One of the key benefits of Mathematica are its powerful graphics features and capabilities, which far exceed those of Matlab, Maple, and other software. In this workshop you will learn to:

- Use `Plot`, `Plot3D`, and similar functions to quickly create 2D and 3D plots of functions.
- Use `ListPlot`, `ListLinePlot` to visualize *lists* of data.
- Use `Manipulate` to make these plots interactive.

6.1 Plotting functions with `Plot` and `Plot3D`

Mathematica has many built-in plotting functions that make generating good-looking plots as easy as computing sums and integrals with `Sum` and `Integral`. The most important such functions are: `Plot` (for plots of one-variable functions) and `Plot3D` (for 3D plots of two-variable functions).

TIP: `Plot` and `Plot3D` have the same basic syntax as `Integrate`, `Sum`, `Table`, and similar functions:

- The **first argument** always denotes the function(s) to be plotted (or integrated, or summed, or listed in a table).
- The **second argument** always denotes the range, which is specified in the usual form `{variable,start,end}`, e.g., `{x,0,1}`. (For functions of two variables, just add two such range specifications, e.g., `{x,0,1},{y,0,1}`.)
- Any additional arguments represent options, specified in the form `option->value`, e.g., `ImageSize->Large`. The plotting functions have numerous options, which are listed on the documentation pages for these functions.

Here are some examples.

1. `Plot[Sin[x],{x,0,10}]` (basic plot of $\sin x$ for x from 0 to 10)
2. `Plot[{Sin[x],Cos[x]},{x,0,10}]` (plots both $\sin x$ and $\cos x$ in the same plot)
   Note that if there is more than one function to plot, the functions must be enclosed in braces to create a list.
3. `Plot3D[Sin[x*y],{x,0,10},{y,0,10}]` (3D plot of $\sin(xy)$).
   For functions of 2 variables, you must use `Plot3D`; `Plot` will give an error.
4. `Plot[Sin[x],{x,0,1},PlotRange->{{-2,2},{-2,2}}]` (a plot with an explicit plotrange)
   Specifying an explicit range can be useful in animations to prevent the axes from moving around.
5. `Table[Plot[Sin[a*x]*Sin[x^2],{x,0,10}],{a,1,5}]` (a list of plots of $\sin(ax)\sin(x^2)$, for different values of a).
   Using `Table` to generate many plots at the same time for different parameters is a great way to experiment. It is much quicker and more efficient than changing the parameters manually.

6.2 Plotting lists with `ListPlot` and `ListLinePlot`

TIP: The syntax for the list plot functions `ListPlot` and `ListLinePlot` is completely different from that of ordinary plotting functions like `Plot`, and **Plot and ListPlot are therefore not interchangeable**.
In its simplest form, `ListPlot` takes a single *list* as argument, and outputs an appropriate plot of this list. For example, if `list1` is a list of numerical values x_i, then `ListPlot[list1]` plots these values against the index, i.e., it plots the points (i, x_i). When given a set of 2-dimensional points, (x_i, y_i), `ListPlot` simply plots these points in the plane.
`ListLinePlot` works the same as `ListPlot`, except that it also connects the points.

Here is how to use list plot functions to plot one- and two-dimensional random walks.

1. First create functions that will generate the random walks to be plotted.

   `rlist1[n_]:=Accumulate[RandomChoice[{1,-1},n]]` (generates an n-step one-dimensional simple random walk)
   `rlist2[n_]:=Accumulate[RandomChoice[{1,-1},{n,2}]]` (generates an n-step two-dimensional simple random walk)

   Test the `rlist` commands for small values of n, to make sure they behave in the expected way.

2. Now apply `ListLinePlot` to the outputs of these commands.

   ```
   ListLinePlot[rlist1[100]]
   ListLinePlot[rlist2[1000]]
   ```

3. Plotting multiple random walks, with different colors for each walk is just as easy: Simply apply `ListLinePlot` to multiple lists, enclosed in braces (e.g., `{list1,list2,...}`). For example:

   ```
   ListLinePlot[{rlist1[1000],rlist1[1000]}]
   ListLinePlot[Table[rlist1[1000],{i,10}]]
   ```

4. Plotting functions have numerous options, which can be specified in the form `option->value` at the end of the plot command. Here are some examples:

   ```
   ListLinePlot[rlist1[1000],Filling->Axis]
   ListLinePlot[rlist1[1000],PlotRange->{-100,100}]
   ListLinePlot[rlist1[1000],ImageSize->Large]
   ```

6.3 Creating interactive animations with `Manipulate`

Build an App in 60 Seconds. The `Manipulate` command is a fantastic tool for creating great-looking interactive visualizations. Once you have created the basic code for a static visualization, it takes only minutes to turn this into a cool interactive "app" with fancy buttons, sliders, check boxes, etc. In fact, the Wolfram website has a great 20 minute video titled "Build an App in 60 Seconds", which shows how to do this. Go to wolfram.com and type the title into the search box.

The basic syntax of `Manipulate` is the same as that of `Table`: `Manipulate[command,range]`, where `command` is the command (e.g., `Plot[...]`) that generates what you want to show in the visualization and `range` is the range for the parameter that you want to "manipulate" interactively, specified in the usual way, e.g., `{a,0,1}`. You can chain together multiple commands (separated by semicolons (;)) and multiple ranges (separated by commas):

```
Manipulate[command1;command2;...,range1,range2,...]
```

(For readability, I recommend putting each command and each range on a separate line). Here are some examples:

1. Create an animation of the plot of $\sin(ax)$ for $0 \leq x \leq 10$ with a slider for the parameter a:
   `Manipulate[Plot[Sin[a*x]*Sin[x^2],{x,0,10}],{a,1,5}]`

   You can move the slider manually, or click on the plus symbol on the right of the slider to run the animation automatically.

2. Add a second parameter, b, to the animation:
   `Manipulate[Plot[Sin[a*x]*Sin[b*x^2],{x,0,10}],{a,1,5},{b,1,5}]`

3. Make the function itself a variable that can be selected.
   `Manipulate[Plot[f[a*x],{x,0,10}],{a,1,5},{f,{Cos,Sin}}]`

4. Change the range for the parameter a to a discrete set of values:
   `Manipulate[Plot[f[a*x],{x,0,10}],{a,{1,2,3,4,5}},{f,{Cos,Sin}}]`

6.4 Creating plots with Mathematica's `Graphics` command

Functions such as `Plot` and `ListPlot` are high level plotting functions that are easy to use, but have somewhat limited functionality. To take full advantage of the powerful graphics capabilities of Mathematica, one needs to use the low level functions `Graphics` and `Graphics3D`. A full treatment of `Graphics` is beyond the scope of this workshop, but here is an example illustrating how to plot random walks using `Graphics` instead of `ListLinePlot`:

1. Define functions that generate n-step 2D and 3D random walks as above, `rlist2[n_]` and `rlist3[n_]`. The outputs of these functions should be *lists* of 2D or 3D points.

2. Now plot these random walks using the `Graphics` and `Graphics3D` commands as follows:

   `Graphics[Line[rlist2[1000]]]` (2D walk of 1000 steps)
   `Graphics3D[Line[rlist3[1000]]]` (3D walk of 1000 steps)

Project: Determining the behavior of Fresnel integrals

Here is a fun and instructive experiment that illustrates a surprising behavior of the integrals

$$f(t) = \int_0^t \cos(x^2)dx, \quad g(t) = \int_0^t \sin(x^2)dx.$$

Integrals of this type are called Fresnel integrals, they cannot be evaluated in terms of elementary functions, and it is not obvious from the definitions how the functions $f(t)$ and $g(t)$ behave as $t \to \infty$. For example, are these functions unbounded, do they oscillate between finite values, or do they converge to a finite limit? In this exercise we'll try to determine the behavior experimentally.

1. **Define appropriate Mathematica functions.** First, define Mathematica functions to do these integrations: `f[t_]=Integrate[Cos[x^2],{x,0,t}]` and similarly `g[t_]`. **Note:** For efficiency reasons, it is better to use `f[t_]=...` (without colon) here instead of the usual notation `f[t_]:=`. This is one of the rare exceptions to the usual rule of using the colon-equal notation in function definitions.

2. **Plot each function.** Plot $f(t)$ and $g(t)$ for $1 \leq t \leq 10$, using the `Plot` function. To show both functions in the same plot, use `{f[t],g[t]}` (with curly braces) as function specification.

3. **Create a 2D plot.** Now plot $(f(t), g(t))$ for $0 \leq t \leq 10$, using the `ParametricPlot` function. Experiment with the `ColorFunction` option, and use `ImageSize->Full` to see the plot in full glory.

4. **Create an interactive animation.** Next, wrap this code inside a `Manipulate` to generate an animation, using the following steps:

   - Change the upper limit for t in the plot from the fixed value 10 to a variable value x (which will be the animation variable)
   - Add a range specification for x, e.g., `{x,1,10,0.1}`. Thus, the command should be of the form `Manipulate[.........,{x,1,10,0.1}]`, where the first argument contains the code for the plot function.
     Note: Starting the range at 0 causes an error (possibly because the integral is over an empty range), so start at $x = 1$.
   - To stabilize the image, specify an explicit range in the plot: `PlotRange->{{0, 1},{0,1}}`.

5. **Take the animation into 3 dimensions.** With a few easy modifications you can generate a cool 3D animation:

   - Change the parameter function $(f(t), g(t))$ to a 3-dimensional function, e.g., $(f(t), g(t), t/10)$. (The normalization $t/10$ ensures that for $0 \leq t \leq 10$ all coordinates stay within a unit cube.)
   - Replace `ParametricPlot` by `ParametricPlot3D`.
   - If you added a `PlotRange` option, make sure to change it to an appropriate 3-dimensional range, e.g., `PlotRange->{{0,1},{0,1},{0,1}}`.

7 Series and Asymptotic Expansions

7.1 Series Basics

1. **Creating power series.** The command to expand a function into a power series is `Series`. Here are some examples.

   - `Series[E^x,{x,0,3}]` (expands e^x into a power series in x about 0, up to order 3, with an error $O(x^4)$)
   - `Series[E^x,{x,1,3}]` (same, but expands about $x = 1$)
   - `Series[x*Sin[1/x],{x,Infinity,6}]` (expands about ∞, i.e., in terms of powers of $1/x$)
   - `Series[Sin[x*y],{x,0,3},{y,0,3}]` (expands into a double series in powers of x and y)
   - `Series[(1+x)^p,{x,0,5}]` (expands $(1 + x)^p$ for general p into a binomial series about 0 to order 5; the coefficients are the generalized binomial coefficients $\binom{p}{n}$)

2. **Extracting coefficients from series.** Assume `series` is the output of a `Series` command like those above, with x being the series variable. For example, `series=Series[ArcSin[x],{x,0,8}]`.

   - `SeriesCoefficient[series,3]` (gives the coefficient of x^3)
   - `Coefficient[series,x^3]` (alternative approach)
   - `CoefficientList[series,x]` (gives the list of all coefficients of powers of x, including 0 coefficients)

3. **The `Normal` command.** The output of the `Series` command is a special data structure that involves a (generalized) Taylor polynomial plus an O-term such as $O(x^4)$ at the end. To get rid of the O-term and obtain an ordinary polynomial, apply `Normal` to the output of `Series`. This may be needed for certain applications, e.g., for plotting.

4. **Series magic.** Much like with lists, one can apply standard arithmetic operations to the output of a `Series` command, producing the expected output. For example, `series1*series2` gives the series obtained by multiplying each term in `series1` with each term in `series2`, `1/series` is the series expansion of the reciprocal of `series`, `Exp[series]` produces the expansion obtained by substituting `series` into the exponential function.

7.2 Exercises

1. **Basic series operations.**

   (a) Expand $1/\sin(x)$ into a series in powers of x out to 5 terms. (Note that the series produced is in fact a Laurent series.)

   (b) Find the first 10 terms of the Taylor series for $e^{\sin x}$ in two ways: (i) by applying `Series` directly to this function; (ii) by applying `Series` to $\sin x$ and exponentiating the output.

   (c) Find the coefficient of x^{10} in the expansion of e^{e^x-1}. (This function is the exponential generating function of the *Bell numbers*, which are implemented in Mathematica as `BellB[n]`, so the result should be equal to `BellB[10]/10!`.)

2. **The partition function via series.** Find the first 10 coefficients in the expansion of $f(x) = \prod_{k=1}^{10}(1 - x^k)^{-1}$. (These coefficients are given by the *partition function*, $p(n)$, which is implemented in Mathematica as `PartitionsP[n]`.)

3. **Fibonacci numbers via series.** The function $1/(1 - x^2 - x)$ is the generating function of the Fibonacci numbers, i.e., the n-th Fibonacci number is equal to the coefficient of x^n. Find the 100-th Fibonacci number by expanding $1/(1 - x^2 - x)$ into a power series and extracting the coefficient of x^{100}. Compare with the buit-in function `Fibonacci[n]`. (Note that the indexing of the latter function is shifted by 1.)

4. **Project 1 Revisited.** In the first project, we have used *numerical* experimentation to show that $(1+1/n)^n = e - (e/2)/n + O(1/n^2)$. The `Series` command allows one to compute such expansions *symbolically*, out to a very large number of terms. Try this out: First, confirm the above formula, then extend the expansion to 10 (or more) terms.

5. **Exercises in Asymptotic Analysis, the easy way.** The following are exercises one might see in courses on Asymptotic Analysis. Using the `Series` command, such exercises become painless:

   (a) **An oscillating integral.** Let $f(x) = \int_0^x \frac{\sin t}{t} dt$. Obtain an asymptotic estimate for $f(x)$ as $x \to \infty$ with an error $O(1/x^3)$.

   (b) **Harmonic sums.** Let $S_n = \sum_{k=1}^n 1/k$. Obtain an asymptotic estimate for S_n as $n \to \infty$ with an error order $O(1/n^3)$. (Note that the `Series` function can be applied to discrete functions like S_n, with n playing the role of the x variable.)

   (c) **Stirling's formula.** Let $S_n = \sum_{k=1}^n \log k$. Obtain an asymptotic estimate for S_n as $n \to \infty$ with an error order $O(1/n^3)$. (Since $S_n = \log n!$, this is essentially equivalent to a version of Stirling's formula.)

Project: Cyclotomic polynomials with "large" coefficients

The cyclotomic polynomial of degree n, $C_n(x)$, is available in Mathematica as `Cyclotomic[n,x]`. For example, $C_3(x) = 1 + x + x^2$, $C_6(x) = 1 - x + x^2$, $C_{20}(x) = 1 - x^2 + x^4 - x^6 + x^8$. Remarkably, the coefficients of these polynomials are all ± 1 or 0. In fact, the same is true for the first 100 cyclotomic polynomials, and one might be tempted to conjecture that *all* cyclotomic polynomials have only coefficients ± 1 or 0. It turns out that this is not the case; from theoretical considerations it is known that there are cyclotomic polynomials with arbitrarily large coefficients. The goal of this project is to confirm this *experimentally*:

1. Begin by creating a function, say `maxCoefficient[n_]`, that calculates the largest absolute value of a coefficient of $C_n(x)$.
   Test the function on some small values of n.

2. Next, write a function, say `record[k_]`, that for a given k-value attempts to find the first index n for which the maximal absolute value of a coefficient of $C_n(x)$ is equal to k. Calculate `record[k]` for as many k-values as possible. (You'll probably not get very far since the coefficients grow very slowly.)

Project: A general partition function and the Frobenius Problem

The classical partition function, $p(n)$, denotes the number of ways n can be written in the form

$$n = \sum_{k=1}^{n} a_k k,$$

where the coefficients a_k are nonnegative integers. This function is available in Mathematica as `PartitionsP`.

We generalize this function as follows: Given a *finite* set S of distinct positive integers, let $p_S(n)$ denote the number of partitions of n into parts from S, i.e., the number of solutions to the equation $n = \sum_{k\in S} a_k k$, with nonnegative integers a_k. This number is equal to the coefficient of x^n in the product

$$\prod_{k\in S}(1-x^k)^{-1}.$$

1. **Implementing a generalized partition function.** Write a function, say `partition[parts_,n_]`, that implements this generalization: That is, given a *list* `parts` (the list S above), and a positive integer n, the function should output the number of partitions of n into parts from S. (If there is no such partition, the number should be 0.)

   To test your function, apply it with the set S being the set of the first n integers (in which case $p_S(n) = p(n)$), and compare the result with the build-in function `PartitionsP`.

2. **The Frobenius problem (coin problem).** Suppose the greatest common divisor of the elements in S is 1. It is known that in this case every *sufficiently large* positive integer n has a partition into parts from S. Hence either every positive integer has such a partition, or there exists a largest positive integer n, called the **Frobenius number** of S, that does *not* have such a partition. Denote this number by $F(S)$ (so that $F(S)$ is the maximal value of n such that $p_S(n) = 0$), and extend the definition to all finite sets S by setting $F(S) = 0$ if every positive integer has a partition into parts from S, and $F(S) = \infty$ if there exist infinitely many n that do not have such a partition (i.e., if the coprimality condition does not hold)

   The **Frobenius problem** is the problem of finding the Frobenius number for a general set S. It is also called the **coin problem** or because of its interpretation in terms of making change for a given amount using coins of given denominations.

3. **Implementing the Frobenius problem in Mathematica.** Using the partition function you have created above (or starting from scratch using `Series`), write a function, `frobenius[S_]`, that takes a list, `S`, of distinct positive integers as input and outputs the Frobenius number, $F(S)$.

   Compare the result with the built-in function, `FrobeniusNumber`.

8 Tips, Tricks, Shortcuts, and General Advice

8.1 Writing code efficiently and quickly

1. **Efficient copying and pasting:** You can do this with the Edit menu, but it is much quicker to just highlight the part you want to cut/copy with the mouse, then use the keyboard shortcuts **Control-X** (Cut), **Control-C** (Copy), and **Control-V** (Paste).

2. **Take advantage of the `@` and `/@` shortcuts.** For example, if you have a list constructed with lengthy `Table[...]` command and you want to sort (`Sort`) and reverse (`Reverse`) this list, using the `@` notation is quicker, less prone to errors, and makes the code easier to read:

   `Reverse@Sort@Table[....]` instead of `Reverse[Sort[Table[...]]]`

   The `/@` symbol is a similar shortcut for the `Map` function.

   Note: These shortcuts work only for operations that don't require additional arguments. For example, with `ListPlot`, one usually wants to add plot options, so one needs to use the standard bracket form, e.g., `ListPlot[Table[...],PlotRange->...]`.

3. **Take advantage of the percent (%) notation.** In Mathematica, the percent symbol, %, refers to the previous output. For example, to get a histogram of a list of values that you have just generated, use `Histogram[%]`. More generally, `%13` refers to output number 13 (i.e., the expression `Out[13]=...`), so you can use, for example, `Histogram[\%13]`..

   Notes: (i) Keep in mind that `%` always refers to the *last* output. If you want to refer to an earlier output, use the explicit form, e.g., `Histogram[%13]`.
   (ii) While the percent notation is very convenient at draft/scratch stages as you are experimenting with code, it should not be used in the final version of the code. (By contrast, the `@` and `/@` shortcuts are perfectly fine for final versions of code, and in fact are *very* common in the code examples found at the Wolfram Demonstrations site.)

4. **Add comments:** To make your code more readable, consider adding comments.

   - **Short comments:** For short comments use `(*...*)`, which works like the slash/star syntax (`/*..... */`) for comments in C/C++.
     You can also use the `(*...*)` notation to comment out part of your code. For example, if you want to experiment with a different option to a Graphics command, put your current version inside the comment symbols, and add the new version.
     Note: Make sure not to "lose" any commas, braces, etc., in the process; to prevent this, move these symbols to a separate line before adding the comment symbols.
   - **Longer comments, section headers, titles, etc.** In addition to short comments, you might want to add titles, section headers and text. To do this, click on Format→Style, and pick the appropriate option. Some handy shortcuts are:
     - **Alt-1** (title),
     - **Alt-4** (section)
     - **Alt-7** (ordinary text, e.g., for explanations of your code)
     - **Alt-9** (Mathematica input—the default).

     To change the style of an existing cell, select the appropriate cell bracket on the right, then click on Format-Style, and pick an option.
     Note: Use those tools sparingly! It is easy to get sidetracked by the myriads of formatting options and styles that are available, but your focus should be on the code!

8.2 Avoiding common mistakes and pitfalls

1. **Increase the magnification in your notebook to 125% (or larger), to make the text more readable.** One of the most common sources of errors is mismatched parentheses, especially, mixing up curly braces `{}` and square brackets `[]`, or having them in the wrong order. With the default magnification of 100%, the different parentheses may be difficult to distinguish. To increase the magnification, use the pop-up menu at the lower right corner of the notebook window.

2. **Know the meaning in Mathematica of the different types of parentheses and be sure not to mix them up:**
   - Single brackets `[...]` denote function arguments.
   - Curly braces `{...}` denote lists.
   - Double brackets `[[...]]` denote elements of lists.
   - Round parentheses `(....)` denote grouping.
3. **Remember that all built-in mathematical constants and functions begin with a capital letter.** A common mistake is to type the Euler constant, e, or the imaginary unit, i, in lower case. In Mathematica, you must use capital letters for these constants: `E` for the Euler constant, `I` for the imaginary unit, `Pi` for π, etc. The same applies to built-in functions: sin must be typed as `Sin`, not `sin`.
4. **Use lower case names and symbols for your own definitions of variables and functions.** This avoids conflicts with built-in functions. For example, in the sum $\sum_{n=1}^{N} \log n$, the upper limit, N, should be renamed to a lower case variable name, e.g., `nmax`: `Sum[Log[n],{n,1,nmax}]`.
5. **Do NOT use underscores (`_`) in variable or function names.** In ordinary programming, underscores is a way to avoid spaces in filenames; for example, `mathematica_workshop_6.pdf` is a perfectly good filename that works on both Unix/Linux and Windows platforms. However, in Mathematica, underscores in user-defined variables and functions such as `random_walk[...]`, would cause problems since the underscore symbol (`_`) has a special meaning in Mathematica (it indicates a pattern).
   (A good name for the "random walk" function would be `randomWalk[n_]:=....` This agrees the naming conventions in Mathematica, where names with multiple components (e.g., `ConvexHull`) are written as a single word, with each component capitalized. The only difference is that, as mentioned, the first letter in user-defined functions should be lower case.)
6. **Use an explicit multiplication symbol `*`, or a blank space, to indicate a product.** A common mistake is to write $a + bx$ as `a+bx`. This causes Mathematica to interpret `bx` as a single variable, not the product of b and x. The correct way to input $a + bx$ is as `a+b*x` or `a+b x`, with a space between the two variables.
7. **Pay attention to visual clues (e.g., red or blue text) of syntax errors.** An expression that is highlighted/colored, typically indicates some problem, e.g., a mis-typed name of a function, mismatched parentheses, etc. For example, if you type `sin[1.3]`, the function `sin` will be colored blue since it is undefined (why?).
8. **If the output is identical to the input, something is likely wrong.** For example, `Integrate[sin[x],{x,0,Pi}]` will simply reproduce the integral (why?).
   Another situation where the output would be simply a copy of the input is when Mathematica tries to evaluate the expression symbolically, but cannot do so. For example, typing `Log[2]` would give `Log[2]` as output, not the numerical value of $\log 2$ that you probably wanted. (To fix this, use the `N` function: `N[Log[2]]` or `N@Log[2]`.)
9. **Be careful when redefining variables or functions. To be safe, use `Clear[...]` to clear out old definitions before redefining a quantity.** As you are developing code, you will often find yourself trying out several different ways to define functions before you come up with a definition that works the way want. This can cause problems and unexpected behavior; these types of errors can be extremely hard to diagnose and track down.

TIP: Definitions that you have deleted from the notebook may still be stored in Mathematica's kernel and can cause problems. Even definitions from other notebooks that you have open can cause problems and unexpected behavior.

8.3 Learn and practice good coding habits

Writing code can at times be extremely frustrating, things may not work as expected, and you may end up spending most of your time trying to fix your code and figure out what's wrong. However, by following a few simple guidelines, you can dramatically cut down on such problems, and produce code that is more robust, easier to read, and easier to maintain and adapt.

1. **Break up your task into small steps, and work on each step independently and test it out thoroughly, before putting everything together.** This is perhaps the most important piece of advice. Don't try to do everything at once!

2. **Test your code at every step.** Whenever you define a new function, test it out immediately to make sure it works as expected. Prepare appropriate test data for that purpose. For example, if your code operates on lists, create some test lists, e.g., `testlist1={1,3,2,5}`, `testlist2=Range[5]`, `testlist3=RandomReal[{0,1},5]`. If it operates on a set of points in the plane, create a test list of points, e.g., `testpoints=RandomReal[{0,1},{5,2}]`.

3. **Focus on getting the core code working, and don't worry about cosmetic details such as colors and plot labels until the very end.** The most difficult part of any project is to get the core code written, tested, and working properly, so focus on this aspect first. Leave the finetuning, the tinkering with options, until the very end. For example, with graphics leave the options at their default settings (i.e., don't specify them explicitly), or pick a simple temporary choice (e.g., `Thickness[Large]`, or `ColorFunction->"Rainbow"`), and stick with it throughout the development phase. Once everything is working properly, you can experiment with different option settings to get the desired "look".

4. **During the testing phase, put each command or definition in a separate cell.** This is accomplished by pressing Shift-Enter after each input line to cause the expression to be evaluated. (To break up an existing cell that contains multiple commands, select the bracket corresponding to the cell, then click on Cell→Divide Cell.)

   This serves two purposes: First, it makes a function definition available for use, so you can test it. Second, it causes each definition to get its own cell. Without pressing Shift-Enter, multiple pieces of code would be lumped into a single cell, and evaluating the cell will cause all pieces to be evaluated. This makes locating errors in the code a much harder task.

   Note: The final program may require cells containing many commands, e.g., a single cell consisting of a `Manipulate` animation, along with all the subcommands that are executed as part of the animation. However, this should only be done during the final phase of the development, once all of the pieces of the code have been independently tested.

5. **During the testing phase, limit the number of variables and use temporary fixed values instead.** For example, a program might eventually require variables n, $nmax$, x, x_0, etc., and involve multiple functions each depending on some or all of these variables.

   Instead of trying to define the most general form of these functions, start out by fixing some of the variables, e.g., $x_0 = 0.5$ and $n = 100$, in your definitions, and, for example, define a function as a one variable-function $f1(x)$ instead of the more general $f(x, x_0, n)$. If the function works as expected with fixed n and x_0, then modify it to take a second argument, e.g., x_0, test the function again, etc.
   Make sure to use different names for the modified function, or use `Clear[...]` before modifying your function.

6. **Keep at least two notebooks open, one (or more) for scratch work and testing, and another for the final version of your code.** As you are developing code, you will need to do a lot of testing and trial and error. Do this work in a scratch notebook; once you have a good piece of code, copy it over to another notebook in which you put together the final version of the code. Make sure to save your notebooks frequently. Crashes can and do happen, and you don't want to lose your work.

TIP: For quick scratch work, e.g., trying out a new command, use any of the Mathematica help pages as your scratch notebook. The documentation pages are live notebooks, and you can use them just like any notebook.

7. **Write code that is easy to parse, and easy to understand, modify, and improve:** Here are some ways to make the code more readable:

   - **Spread out lengthy commands.** For example, for a `Graphics` command put each graphics item (e.g., lines, points) in a separate line along with the associated directives (e.g., `Thickness[0.02]`, and put each plot option (e.g., `ColorFunction->"Rainbow"`), on a separate line. While this will cause the command to extend over many lines, it makes it much more readable, easier to modify, and easier to diagnose in case there are errors.
   - **Introduce intermediate variables.** For example, instead of writing a `Graphics` command that includes all the code to produce the points and lines, put the definition of the points and lines into auxiliary variables, say `pts=Point[....]` and `lines=Line[.....]`, and then use `Graphics[{pts,lines,...}, ...]`. A `Manipulate` command would then have the following structure:

     ```
     Manipulate[pts=Points[...];
                   lines=Lines[...];
                   ...;
                   Graphics[{pts,lines,...},...]
                   {x,0,1},....
                ]
     ```

     (Of course, the Graphics part would normally be spread out over many lines.)
   - **Add comments as needed.** Ideally, your code should be "self-explanatory", so that anyone with basic knowledge of Mathematica will see how it works and will be able to tinker with the code. That said, adding comments or explanatory text may be helpful at times. In Mathematica comments can be enclosed in $(* \cdots *)$, or put in a text cell. To get a text cell, use Alt-7, or click on the FormatStyle→Text. For more complex code, you might want to break it up into sections, and add section headers (use Format→Style→Section, or Alt-4).

8. **Force numerical evaluation unless you want symbolic output.** Normally, Mathematica evaluates expressions exactly (i.e., symbolically); you can force *numerical* evaluation with an explicit `N` command.

The decimal point trick for forcing numerical evaluation: If any part of an expression is written in decimal form (e.g., `2.0` instead of `2`), then Mathematica assumes that you want numerical output, and uses numerical evaluation, so an explicit `N` is not necessary. For example, `Sin[2*Pi/7]` will not give a numerical output, but `Sin[2.0*Pi/7]`, `Sin[2*Pi/7.0]` and `0.0+Sin[2*Pi/7]` will all give numerical values.

9 Tutorials, Books, and Further Resources

9.1 Online Tutorials

The following tutorials are available at the Wolfram website, `wolfram.com.` Just go to the site and put the title in the search box.

- [Beginner] **Fast Introduction for Programmers.** A quick, self-guided tutorial on the basic structure of the language. An excellent guide, especially for those who have some programming background and want to get quickly up to speed with the syntax of Mathematica.
- [Beginner/Intermediate] **An Elementary Introduction to the Wolfram Language.** A new tutorial by Stephen Wolfram, available both in book form and as a free online tutorial. The coverage is extremely broad (there are 47 chapters altogether, each devoted to a different topic), so it's best to pick and choose those topics relevant to what you want to do rather than work through the tutorial from beginning to end. The pace is very gentle, and there are numerous exercises.

9.2 Free E-Books

Note: There exist many books on Mathematica, but most are too old to be of much use today. The books below are reasonably current and are available for free in electronic form for UIUC students. (Go to `http://www.library.illinois.edu`, and type the title in the search box.)

- **[Beginner] Jonathan Borwein and Matthew Skerritt, An Introduction to Modern Mathematical Computing with Mathematica.** A beginner-level introduction with exercises. The main focus is on number theory, calculus, and linear algebra. Available as an E-Book through the University Library website, `http://www.library.illinois.edu`.
- **[Beginner] Bruce Torrence, Eve Torrence, A Student's introduction to Mathematica.** A beginner's guide. Available as an E-Book through the University Library website, `http://www.library.illinois.edu`.
- **[Beginner/Intermediate] Stan Wagon, Mathematica in Action (3rd edition, 2010).** An intermediate-level introduction, with lots of mathematical examples. Available as an E-Book through the University Library website, `http://www.library.illinois.edu`.
- **[Intermediate/Advanced] Paul Wellin, Programming with Mathematica (2013).** An excellent book on Mathematica, at an intermediate/advanced level. The most recent edition (2013) is available as an E-Book through the University Library website, `http://www.library.illinois.edu`. Wellin's book is one of the few Mathematica books that comes with exercises. The entire set of exercises, with complete solutions, is available for free at the Cambridge University Press website.

TIP: Leonid Shifrin, "Mathematica programming: an advanced introduction". This is, far and away, the best advanced book on Mathematica that I know of. It is a completely free online book, available for download at `http://www.mathprogramming-intro.org.` The focus is on the structure of the language and good coding techniques. It offers lots of general advice on coding in Mathematica that cannot be found elsewhere.

9.3 Other resources.

- **Wolfram Demonstration Project,** `http://demonstrations.wolfram.com`**.** Over 10,000 Mathematica "demonstrations"—interactive visualizations and animations to illustrate concepts in mathematics and other fields. The source code can be viewed online or downloaded. All demonstrations published at this site have undergone a review process, so the code you see there is generally of very high quality.
- **Mathematica Stack Exchange,** `mathematica.stackexchange.com`**.** The best discussion forum for Mathematica questions. If you run into a problem that you cannot find an answer to in the Mathematica documentation, search on Mathematica Stack Exchange.
  In particular, I recommend the thread "What are the most common pitfalls awaiting new users?".