Mathematica for the Mostly Busy

Nathan Chapman

May 30, 2017

I dedicate this work to the people who strive for truth about our reality, otherwise known as scientists.

"Math: a lifetime of free entertainment" - Dr. Branko Ćurgus

Preface

This guide is meant to be the sequel to *Mathematica for the Constantly Busy.* As the title might allude, this guide is for people who want to learn more about *Mathematica* than just the basics. That being said, here we will see some slightly more advanced topics both in *Mathematica* and in math. It will follow a similar format to the previous guide, so if you liked *Constantly Busy* , you hopefully will like this one. If you did not like *Constantly Busy* , then you're out of luck because by the time you're reading this, I will have already wrote this and probably can't make changes for just you.

Since this is the second part of a series, I will assume the reader has the equivalent knowledge of the information from *Constantly Busy* . Therefore, I will not go over basic syntax like capitalizing functions, basic function format, and brackets.

Contents

Part I

Syntax

Chapter 1

Definitions & Assignments

Labeling a variable to be used later is one of the most key concepts in math and computer science, and that certainly isn't lost here. The most basic variation of labeling an object, or *setting* it as some other thing, can be used not only for numerical assignments, but also for graphical objects. In other words, you can set a plot or a graphic to have a label of whatever you choose. This is unbelievably helpful when dealing with results that contain A LOT of elements, especially when they are large sized plots, because when you want that set of plots, you can simply refer back to the label instead of either copying the actual output or needing to evaluate the expression again. *Mathematica* has a few different ways of labeling an object.

Here we will see more in-depth explanations of the different kinds of assignments in *Mathematica* .

1.1 Set

To perform the most basic assignment operation we use the **Set** command, or simply =.

To set the variable x, or rather assign the label x to the result that comes from the $expr$, we use

$$x = expr$$

You can also set multiple labels $x_1, x_2, \ldots$ to multiple objects $expr_1, expr_2, \ldots$ at the same time with

$$\{x, y, \ldots\} = \{expr_1, expr_2, \ldots\}$$

A very handy feature *Mathematica* has, as well as other programs, is the ability to assign values and objects to certain elements in lists and/or arrays, etc.

To assign the label $a[[i]]$, that's the i-th element in the list/array with label a, to the object v we use

$$a[[i]] = v$$

Note: This is **NOT** the same as mathematical equality, which we will get to in the sections. This is one of the biggest difficulties people encounter when they first start using *Mathematica* .

1.2 Mathematical Equality

Mathematica , much like other coding languages, has different syntax for mathematical equality. In the case of denoting mathematical equality, we use the notation of a double-equals. Rather, if the quantities x, y are mathematically equal we denoted this relationship with

$$x == y$$

This notation can be used when providing conditions for the solve function, or simplifying/reducing an equation, etc.

1.3 Delayed Definition

An extra nifty feature *Mathematica* has is the ability to delay an assignment until after a process has been evaluated. This is the notion we use to define functions and objects that need to be reevaluated every time they are ran/called.

For a variable x to be defined by the result of process *newly ran expr*, we use delayed definition by

$$x := expr$$

Note: There is a key difference between the delayed definition and the immediate "set" definition. The key here is with the *immediate* definition, the process *expr* is ran, then the result is assigned to the object x once, while with the *delayed* definition the process *expr* is called and evaluated when the object x is called.

For example, consider the code

$$x = \text{RandomReal}[]$$

This will evaluate the RandomReal function and assign a psuedo-random real number between in the interval $[0, 1]$ to the object x. Then whenever x is called, it will be replaced with this number.

Now consider the code

$$x := \text{RandomReal}[]$$

This will assign the *process* RandomReal to the object x, and therefore, every time the object x is called, the the command RandomReal will run and output a different number each time (probably, since there is a nonzero probability it will output the same number twice).

To further illustrate the point, the code

```
x = RandomReal[];
y := RandomReal[]
```

followed by the the next cell with input

```
Table[{x,y},{4}]//MatrixForm
```

will give the output

$$\begin{bmatrix} 0.76451 & 0.296771 \\ 0.76451 & 0.992581 \\ 0.76451 & 0.958455 \\ 0.76451 & 0.658801 \end{bmatrix}$$

This code calls the both x, y four times and outputs the result in the form of a matrix. It can be seen that each time x is called, it is the same number, while each time y is called, it is a different number. Hopefully this is enough to get the point across. If not, I highly suggest practicing and exploring the nature of this difference on your own before moving on to other content because this concept is used time and time again throughout application.

Chapter 2

Functional Operations

2.1 Functions

Mathematica has a few different ways not only to use functions, but also to define them.

If we want to make a function $f(x) = x^2$, then we can do

$$f[x\_] := \#^2\&[x]$$

or

$$f[x\_] := \text{Function}[x, x^2]$$

or

$$f[x\_] := x \mapsto x^2$$

or

$$f[x\_] := x^2$$

Note: Here we use delay definition, :=, to make sure the function is properly defined, so when we call the function, it evaluates in the right order.

The first option is using a delayed definition to set a **pure function** as a function. Usually we don't do this, but it is possible. Normally, pure functions are used once in code instead of having to define a whole new named function first.

The second option is using the function function. This is equivalent to the first option, except you can name your variables and the variable within the function function are treated as local.

The third option uses $\mapsto$ notation, which should be read "mapsto". This is more of a ruling rather than a concrete functional programming way to do things.

The fourth option is how we normally define functions.

While all of these are equivalent, we usually only use pure functions and the last option for defining functions.

2.2 Map

Lets say you have a function f, and a set $\{x_1, x_2, \ldots, x_n\}$ and you want $\{f(x_1), f(x_2), \ldots, f(x_n)$. *Mathematica* has a built in function for just the task.

To thread a function, not necessarily mathematical, over a list simply use

$$\text{Map}[f, list]$$

or

$$f/@list$$

This /@ command, as well as other shorthand syntactical features, is a nice, clean way of saying "Map f over $list$".

This is an incredibly useful function and makes it possible to create lists/sets of a function applied to each elements of a set, when the function isn't threadable from the start.

A great use of Map is to use it with pure functions. For example

$$\#^2\&/@\{x_1, x_2, \ldots, x_n\} = \{x_1^2, x_2^2, \ldots, x_n^2\}$$

2.3 Apply

Apply kind of does the opposite of Map, in the sense Apply makes the entire list the argument of the function.

To apply the function f, again not necessarily mathematical, to the expression $expr$ we use

$$\text{Apply}[f, expr]$$

or

$$f \ @@ \ expr$$

What this really does is replace a thing that's called the "head" of $expr$, with f.

Chapter 3

Patterns

Mathematica uses patterns to recognize things that have similar attributes.

3.1 Blank

We mostly use this pattern type to refer to variables in functions that can take any kind of argument.

For example if you want to create a function $f(x)$ where x can be anything, e.g. number, symbol, list, we use

f[x_]:= *expr*

This is usually how we do things, since you usually know what you're putting into the function.

3.2 Blank Sequence

If you want your function to only take multiple things of any kind, we use

f[x___]:= *expr*

We would use this in the case where our function is defined with other functions that only take lists as arguments. e.g. Length, Plus, etc.

3.3 Blank Null Sequence

If you want to be able to put any data type, of anything and treat it as a single variable we use

f[x____]:= *expr*

This will take what ever you stipulate as the function argument, whether it be lists of lists and number and symbols, or whatever, *Mathematica* will input that into *expr* and evaluate it.

3.4 Alternatives

A handy way to do multiple things in one command is to use alternatives.

If you want to say to *Mathematica* "Hey, *Mathematica* could you do me a favor and please replace every instance of x with y in $expr$?", we would use

$$expr/.x|y- > z$$

We will see this /. command used and defined again later.

3.5 Pattern

All of the previous sections have been about specific pre-defined patterns. Now we will see how to make our own.

To define a general pattern x, with the pattern obj we use

$$x{:}obj$$

3.6 Except

If we want our pattern to be anything, except a specific thing, *Mathematica* has got you covered.

To define a pattern x to be anything except obj, we use

$$x{:}\text{Except}[obj]$$

Similarly, if you want to define your pattern x to be obj, but not some element in obj, $s \in obj$, we use

$$x{:}\text{Except}[s,obj]$$

3.7 Restrictions

To restrict your pattern to certain patterns with specific head, *Mathematica* can do that as well.

To restrict a pattern x to anything with the head $head$, we use

$$x\_head$$

3.8 Condition

Mathematica not only has the "If" function, but also the feature to streamline the "If" process.

With the "Condition" function, you can apply your favorite "If" statements to any pattern *patt*, or definition *lhs := rhs* with

patt /; test

or

lhs:= rhs /; test

respectively.

3.9 PatternTest

If you want to create a pattern, that also tests for some condition being true, there's already a function for that.

Say you want to create a pattern x that matches to objects with the pattern p, and also tests for the condition *cond* being true, we use

x:p?cond

This pattern object, combined with the "Cases" function, is very similar to the "Select" function. The difference between these two functions is the criteria/-on to witch the object is being tested. I will leave an in-depth exploration as an exercise to the reader.

3.10 Optional

When using patterns in a function definition, we can stipulate if there are optional variables.

To define a function f with a necessary variable x of any pattern, and an optional variable y, which if omitted defaults to z, we use

f[*x_*,*y_*:*z*]:=*expr*

where *expr* is the machinery behind the function.

3.11 Default

If you want the same thing to happen for a bunch of variables when they are omitted, instead of needing to dictate what each one should do, we can set a default.

To set the default action x for omitted variables in a function f, we use

Default[f]=x

To tell *Mathematica* the variables *x,y* are optional and should use the default action when omitted, we use

f[*x*_.,*y*_.]:=*expr*

3.12 Cases

If you have a list *list* but you only want the elements of *list* that meet a certain pattern *pat*, you use the Cases command with

Cases[*list*,*pat*]

to give a list of the elements of *list* that meet the pattern *pat* in the same order they appear in *list*.

Note: Here the distinction of the criterion being a pattern is important. There are other functions that do the same operation for the criterion not being a pattern.

3.13 DeleteCases

While the Cases function give a list of elements that meet a pattern, the DeleteCases function does the complement of the operation.

If you have a list *list* but you only want the elements of *list* that ***do not*** meet a certain pattern *pat*, you use the DeleteCases function with

DeleteCases[*list*,*pat*]

to give a list of the elements of *list* that ***do not*** meet the pattern *pat* in the same order they appear in *list*.

3.14 Position

There will come a time when you have list and you want to know where in the list, the elements meeting a certain pattern are, or rather their position. *Mathematica* has you covered for that.

If you have a list *list* and want to know the position of the elements meeting a certain pattern *pat*, you use the Position function with

Position[*list*, *pat*]

3.15 Count

If you just want to know how many elements in your list *list* meet a certain pattern *pat*, you use Count by

Count[*list*,*pat*]

This does the same process and therefore gives the same result as Length[Cases[*list*,*pat*]].

Chapter 4

Rules

Part II

Math

Chapter 5

Algebra

In this section we will encounter orthogonal functions, infinite sums and how to apply them in certain situations, and more.

5.1 Zeroes of a Function

In some instances you might find yourself with either with an extremely complicated equation, or even a transcendental (read "analytically unsolvable") equation that you need to end up solving numerically. When this happens, *Mathematica* has your back.

To find the root of a function f, or equation $f(x) = g(x)$, we use

$$\text{FindRoot}[f, \{x, x_0\}]$$

or

$$\text{FindRoot}[lhs == rhs, \{x, x_0\}]$$

Note: Since we're doing this numerically, we need to tell *Mathematica* where to look. This is where the $\{x, x_0\}$ comes from. Basically we're asking *Mathematica* "Hey, *Mathematica* , if you're not too busy could you please find the root of this function or equation for x around x_0?".

5.2 Solving Equations

There are two main methods in which to solve equations in *Mathematica* , analytically and numerically. In order to implement these methods, *Mathematica* uses the functions *Solve* and *NSolve* respectively.

5.2.1 Solve

Given an *algebraic* equation $lhs = rhs$ where one, or both, of the sides is/are function(s) of a variable x

Chapter 6

Calculus

In *Constantly Busy* the calculus chapter was very short because all it was, was how to do the derivative and integral functions. Here, as with the rest of the book, we will look at calculus a little more in depth.

6.1 Derivatives

There are actually several different kinds of derivatives you can take in *Mathematica* . For instance you can take the total derivative of a function this is itself composed of several functions not of the same variable.

6.1.1 Partial Derivatives

Taking a partial derivative of a function in *Mathematica* is just about the easiest thing to do. In *Constantly Busy* , I just referenced this function as a general derivative, but now we can elaborate some more. There are several kinds of partial derivatives you can take, e.g. single derivatives, multiple derivatives, successive derivatives, vector derivatives (Gradient), etc.

Single Derivatives

To take a single, partial derivative of the function f with respect to x in *Mathematica* we use

$$\mathrm{D}[f, x]$$

We can tweak this with the very useful, built-in commands to suit our needs.

Multiple Derivatives

To take a multiple, partial derivative of the function f with respect to x n times in *Mathematica* we use

$$\mathrm{D}[f, \{x, n\}]$$

This is great if you only want the rate of change, curvature/rate of rate of change in one direction. But, normally, there is more than one direction.

Successive Derivatives

To take a successive, partial derivatives of the function f with respect to x, then y in *Mathematica* we use

$$\mathrm{D}[f, x, y]$$

These three kinds are great if we're only looking at one value, but if we want to look at the behavior of the whole vector we can do that as well.

Vector Derivatives

To take a Euclidean-vector, derivative of the function f with respect to $x_1, x_2, \ldots, x_n$ in *Mathematica* we use

$$\mathrm{D}[f, \{\{x_1, x_2, \ldots, x_n\}\}]$$

Note: Here there are two sets of curly brackets enclosing the variables to which we are differentiating with respect to. These are especially important, otherwise *Mathematica* would think we are trying to differentiate f with respect to x_1, x_2 times, and also there is a bunch of other stuff after that, that doesn't make sense so it would break anyways.

Later, in the vector analysis and partial differential equations sections and chapters, we will see the sum of the second partial derivatives of a multivariate function, otherwise known as the Laplacian

$$\nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$$

6.1.2 Total Derivatives

Single

The total derivative of a function $y = f(t, x_1(t), x_2(t), \ldots, x_n(t))$ with respect to t is defined as

$$\frac{\partial y}{\partial t} = \frac{\partial f}{\partial t} + \sum_{k=1}^{n} \left[f'(x_{k,1..T}) \prod_{i=1}^{T} x'_{k,i..T}(x) \right]$$

Where, with $(f \circ g)(x) = f(g(x))$,

$$f_{a..b} = f_a \circ f_{a+1} \circ \cdots \circ f_{b-1} \circ f_b$$

Note: This is when each argument is itself composed T times with functions of t. When each argument is only dependent on t once, $T = 1$, the above equation reduces to

$$\frac{\partial y}{\partial t} = \frac{\partial f}{\partial t} + \sum_{k=1}^{n} f'(x_k)x'_k(t)$$

To take a total derivative of the function f with respect to t in *Mathematica* we use

$$\text{Dt}[f, t]$$

Multiple

To take a multiple, total derivative of the function f with respect to t, n times in *Mathematica* we use

$$\text{Dt}[f, \{t, n\}]$$

6.1.3 Options

There are a few options for D that can come in handy.

Differentiate with respect to a function

While not mathematically making too much sense, in *Mathematica* you are able to differentiate a function with respect to another function, or rather a general variable.

We can differentiate the function $f(x) + f'(x)$ with respect to the general variable

$$\text{In: } \text{D}[f[x] + f'[x], f[x]]$$

$$\text{Out: } 1$$

Non Constant Variables

Sometimes you want to assume something about the variable in the function and what not.

If we want to differentiate the function $f(x, y)$, but assume y might be a function of x, we do

$$\text{D}[f(x, y), x, \text{ NonConstants} \to \text{y}]$$

6.2 Integrals

In this version we will skip single variable indefinite and definite integrals, but we will see both integrals over a rectangle and over a region. As with single variable integrals, one could use NIntegrate to numerically calculate integrals for which *Mathematica* doesn't have an explicit representation.

6.2.1 Rectangular Region

Just like with other similar things, a definite integral over a rectangular region in $\mathbb{R}^n$ can be done very easily.

To integrate a function $f(x, y)$,

$$\text{Integrate}[f, \{x, x_{min}, x_{max}\}, \{y, y_{min}, y_{max}\}]$$

6.2.2 Any Region

To integrate the function $f(x, y)$ over any region, we use

$$\text{Integrate}[f, \{x, y, \ldots\} \in reg]$$

For example

$$\text{Integrate[Abs[Sqrt[x]],}\{x, y\} \in \text{Disk[]]}$$

integrates the function $|\sqrt{x}|$ over the unit disk $\{(x, y) : x^2 + y^2 < 1\}$.

6.2.3 Options

For integrals, there are quite a few options.

Assumptions

Mathematica does its best to take into account of all the different possibilities for variables, like not restrict variable to be in the reals, but sometimes you know some of these restrictions are indeed true.

To integrate the function $f(x)$ with respect to x but you know $x \in \mathbb{Z}$ (the integers), we can use

$$\text{Integrate}[f[x], x, \text{ Assumptions} \to x \in \text{Integers}]$$

6.3 The Fourier Series

In the algebra chapter we saw the Fourier cosine and sine series

$$f(x) \sim a_0 + \sum_{n=1}^{\infty} a_n \cos\left(\frac{n\pi}{L}x\right) + \sum_{n=1}^{\infty} b_n \sin\left(\frac{n\pi}{L}x\right)$$

Now we that we have derivatives and integrals, we can use this representation to do calculus on an otherwise stubborn function.

Fun Fact: In the usual case of a Fourier series being the solution to a partial differential equation, the cosine and sine functions are called "eigenfunctions" of the corresponding boundary eigenvalue problem.

6.3.1 The Coefficients

The coefficients of the Fourier series above are defined as

$$a_0 = \frac{1}{2L}\int_{-L}^{L} f(x)dx$$

$$a_n = \frac{1}{L}\int_{-L}^{L} f(x)\cos\left(\frac{n\pi}{L}x\right)dx$$

$$b_n = \frac{1}{L}\int_{-L}^{L} f(x)\sin\left(\frac{n\pi}{L}x\right)dx$$

where $n \in \mathbb{N}$.

Mathematica can usually calculate and use these integrals quite easily, even when you don't assume n is in the integers. To make it even quicker and easier for *Mathematica* , you can tack on to the end "Assumptions $\to n \in$ Integers".

6.3.2 Using the Fourier Series

Now that we have the Fourier series, we can do some calculus, as well as other things, with it. It should be noted there are special conditions that must be met to differentiate a Fourier series, but if those are met then *Mathematica* can come in handy. For integrals, there are no such conditions, so we can integrate a Fourier series without worry.

Fun Fact: We can differentiate and integrate Fourier series because the derivative and integral are each what's called a linear operator. Because of this each operator is able to thread over each term in the series to turn the derivative of a sum into a sum of derivatives.

So, for example, say we have a piecewise smooth function $f(x)$ where $-\pi < x \leq \pi$. Then the Fourier coefficients are

$$a_0 = \frac{1}{2\pi}\int_{-L}^{L} f(x)dx$$

$$a_n = \frac{1}{\pi}\int_{-L}^{L} f(x)\cos(nx)\,dx$$

$$b_n = \frac{1}{L}\int_{-L}^{L} f(x)\sin(nx)\,dx$$

where $n \in \mathbb{N}$. Which gives the Fourier series

$$f(x) \sim a_0 + \sum_{n=1}^{\infty} a_n \cos(nx) + \sum_{n=1}^{\infty} b_n \sin(nx)$$

Then, under the appropriate conditions, we can differentiate the series to yield

$$f'(x) \sim -\sum_{n=1}^{\infty} a_n n \sin(nx) + \sum_{n=1}^{\infty} b_n n \cos(nx)$$

and integrate to produce

6.4 The Fourier Transform

The Fourier series is defined for $n \in \mathbb{Z}$, which tells us our collection of eigenfunctions $\{f_n\}$ is indexed in the naturals; but what if we had an uncountably infinite collection of eigenfunctions? Then we would need some sort of summation whose index is in an uncountably infinite set. Sound familiar? In the usual/physical case, this uncountably infinte indexing set is $\mathbb{R}$, the real numbers. When the function is "nice", we get to use the good ol' fashion Riemann integral. This turns our Fourier series of the function $f(x)$ into the Fourier Transform of $f(x)$

$$(\mathcal{F}f)(\xi) = \int_{-\infty}^{\infty} f(x) e^{-2\pi \imath x \xi} dx$$

6.5 The Laplace Transform

The Laplace transform is very closely related to the Fourier transform. The Laplace transform is used heavily in solving differential equations, and leads to a great many interesting topics, such as the Gamma function $\Gamma(n)$.

The Laplace transform of a function $f(t), t \geq 0$ is defined as

$$\mathcal{L}[f(t)](s) = \int_{0}^{\infty} f(t) e^{-st} dt$$

6.6 The Dirac Delta Function

The Dirac delta function is ever so convenient and also ubiquitous in quantum mechanics and differential equations in general. The Dirac delta "function",$\delta(x)$ is defined such that

$$\int_{-\infty}^{\infty} \delta(x) dx = 1$$

It can be thought of as

$$\delta(x) = \begin{cases} \infty & x = 0 \\ 0 & x \neq 0 \end{cases}$$

though it is not rigorously defined this way.

In *Mathematica* we implement this with

$$\text{DiracDelta}[x]$$

Un-augmented, this only evaluates to a nonzero value at $x = 0$, but we can simply translate it with

$$\text{DiracDelta}[x - a]$$

then the "definition" becomes

$$\delta(x) = \begin{cases} \infty & x = a \\ 0 & x \neq a \end{cases}$$

The Dirac delta function is also defined for more than one dimension, which can be implemented with

$$\text{DiracDelta}[x_1, x_2, \ldots]$$

6.7 Vector Analysis

When doing vector analysis operations in *Mathematica* , you first need to import the vector analysis package with

$$\text{Needs}[\text{“VectorAnalysis‘”}]$$

Note: The apostrophe at the end does need to be included or else the package will not be imported, and therefore the vector analysis operations will not be available.

Note: Unless you're working in *Mathematica* 10, or later, the package does need to be imported.

6.7.1 Divergence

The divergence of a vector field $\mathbf{F}(x_1, x_2, \ldots, x_n) = \langle f_1(x_1, x_2, \ldots, x_n), f_2(x_1, x_2, \ldots, x_n), \ldots, f_n(x_1, x_2, \ldots, x_n) \rangle$ can be defined as the inner product between the gradient operator

$$\nabla = \left\langle \frac{\partial}{\partial x_1}, \frac{\partial}{\partial x_2}, \ldots, \frac{\partial}{\partial x_n} \right\rangle$$

and the vector field $\mathbf{F}(x_1, x_2, \ldots, x_n)$ to yield

$$\nabla \cdot \mathbf{F} = \frac{\partial}{\partial x_1} f_1 + \frac{\partial}{\partial x_2} f_2 + \cdots + \frac{\partial}{\partial x_n} f_n = \sum_{k=1}^{n} \frac{\partial}{\partial x_k} f_k$$

Then in *Mathematica* , to take the divergence of the vector field $\mathbf{F} = f(x_1, x_2, \ldots, x_n)$ in Cartesian coordinates, you use

$$\text{Div}[\{f_1, f_2, \ldots, f_n\}, \{x_1, x_2, \ldots, x_n\}]$$

To take the divergence of the vector field $\mathbf{F}$ in the non-Cartesian coordinate system *chart*, you use

$$\text{Div}[\mathbf{F}, \{x_1, x_2, \ldots, x_n\}, \textit{chart}]$$

Examples of non-Cartesian coordinate systems are cylindrical coordinates, spherical coordinates, etc.

6.7.2 Curl

The curl of the three dimensional vector field $\mathbf{F}(x_1, x_2, x_3) = \langle f_1(x_1, x_2, x_3), f_2(x_1, x_2, x_3), f_3(x_1, x_2, x_3) \rangle$ can be defined as the cross product between the gradient operator

$$\nabla = \left\langle \frac{\partial}{\partial x_1}, \frac{\partial}{\partial x_2}, \ldots, \frac{\partial}{\partial x_n} \right\rangle$$

and the vector field $\mathbf{F}(x_1, x_2, x_3)$ to yield

$$\nabla \times \mathbf{F} = \left\langle \frac{\partial}{\partial x_2} f_3 - \frac{\partial}{\partial x_3} f_2, \frac{\partial}{\partial x_3} f_1 - \frac{\partial}{\partial x_1} f_3, \frac{\partial}{\partial x_1} f_2 - \frac{\partial}{\partial x_2} f_1 \right\rangle$$

Then in *Mathematica* , to take the curl of the vector field $\mathbf{F}(x_1, x_2, x_3)$ in Cartesian coordinates, you use

$$\text{Curl}[\{f_1(x_1, x_2, x_3), f_2(x_1, x_2, x_3), f_3(x_1, x_2, x_3)\}, \{x_1, x_2, \ldots, x_n\}]$$

To take the curl of the vector field $\mathbf{F}$ in the non-Cartesian coordinate system *chart*, you use

$$\text{Curl}[\mathbf{F}(x_1, x_2, x_3), \{x_1, x_2, \ldots, x_n\}, \textit{chart}]$$

Examples of non-Cartesian coordinate systems are cylindrical coordinates, spherical coordinates, etc.

6.7.3 Laplacian

The Laplacian of the scalar function $f(x_1, x_2, \ldots, x_n)$ can be defined as the inner product between the gradient operator

$$\nabla = \left\langle \frac{\partial}{\partial x_1}, \frac{\partial}{\partial x_2}, \ldots, \frac{\partial}{\partial x_n} \right\rangle$$

and the gradient of the scalar function $f(x_1, x_2, \ldots, x_n)$,

$$\nabla f = \left\langle \frac{\partial}{\partial x_1} f, \frac{\partial}{\partial x_2} f, \ldots, \frac{\partial}{\partial x_n} f \right\rangle$$

to yield

$$\nabla \cdot \nabla f = \frac{\partial^2}{\partial x_1^2} f + \frac{\partial^2}{\partial x_2^2} f + \cdots + \frac{\partial^2}{\partial x_n^2} f = \sum_{k=1}^{n} \frac{\partial^2}{\partial x_k^2} f$$

Then in *Mathematica* to take the Laplacian of a scalar function $f(x_1, x_2, \ldots, x_n)$ in Cartesian coordinates, you use

$$\text{Laplacian}[f, \{x_1, x_2, \ldots, x_n\}]$$

To take the Laplacian of the scalar function $f(x_1, x_2, \ldots, x_n)$ in the non-Cartesian coordinate system *chart*, you use

$$\text{Laplacian}[f(x_1, x_2, \ldots, x_n\}, \mathit{chart}]$$

Examples of non-Cartesian coordinate systems are cylindrical coordinates, spherical coordinates, etc.

Chapter 7

Linear Algebra

7.1 Arithmetic

7.2 Inner Product

7.3 Cross Product

7.4 Outer Product

7.5 Normalize

7.6 Orthogonalize

Chapter 8

Differential Equations

8.1 Ordinary Differential Equations

8.2 Partial Differential Equations

Part III

Data

Part IV

Important Functions

Chapter 9

Plotting

Here we will look at more of the customization options for Plot.

9.1 Plot

9.1.1 Options

9.2 Plot3D

9.3 VectorPlot

9.4 VectorPlot3D

9.5 StreamPlot

Chapter 10

Manipulate

Part V

Graphics

Conclusion