

6. Functional Programming



6.1 Introduction

What is functional programming?
Objectives

6.2 Map

Mapping functions over lists
Keeping your code general
Mapping functions over expressions
Shorthand for **Map**
Exercises

6.3 Apply, Thread and Outer

Apply

Exercise

Thread

Exercises

Outer

Exercise

6.4 Anonymous Functions

Defining a function without giving it a name
Exercise

6.5 Nest, NestList and FixedPoint

Nest

NestList

Exercise

FixedPoint

Exercises

6.6 Summary: When to Use Functional Programming

Don't dismember lists
Map, Apply, Thread and **Outer**
Anonymous functions
Nest, NestList and **FixedPoint**

6.7 Problems

Problem 1: Calculating the centre of gravity
Problem 2: Smoothing data
Problem 3: Exploring vehicle suspension dynamics
Problem 4: Discovering repeated data
Problem 5: Distances from a fixed point
Problem 6: Newton's method for any number of variables (More difficult!)

6.1 Introduction

What is functional programming?

- Functional programming is a style of programming which consists of functions applied to arguments, for example the elements of a list, *and to other functions*.
- Surprisingly, you can write very sophisticated programs quite easily like this.
- Functional programming allows us to deal with lists and data structures efficiently, by manipulating them as wholes, rather than by taking each element and manipulating it separately as procedural programming would.

Although functional programming can be applied to general expressions, not just lists, our introductory examples will usually be to lists because operations on data structures are very common, and since they are the easiest type of expression to visualize.

Objectives

- To understand the concept of functional programming.
- To be able to manipulate expressions using functions.
- To be able to write programs using functional programming techniques.

6.2 Map

Mapping functions over lists

We begin by discussing **Map**, one of the more important functions used in functional programming.

Suppose we have a list of elements **{a, b, c}** and we wish to make a list of all the *logs* of these numbers. We can **Map** the **Log** function over the list.

```
Map[Log, {a, b, c}]  
  
{Log[a], Log[b], Log[c]}
```

Of course, the *inbuilt listability* of the **Log** function means we could also have obtained the same result by entering the simpler expression **Log[{a, b, c}]**. So when we are using simple inbuilt functions, **Map** is often no advantage over using their listability properties.

```
Log[{a, b, c}]  
  
{Log[a], Log[b], Log[c]}
```

However, it may be that you have defined one of your own functions, and would like to **Map** it over a list in the same way.

For example suppose we were working with lists of data and wanted to put any negative elements in the lists to zero. We can easily define a function that does this *for just one element*.

```
Clear[putToZero]

putToZero[x_] := If[x < 0, 0, x]
```

(Remember, if you come across a function that you have not met before, (like **If** above), use the **?** to find out more about it or check it out in the Help Browser.)

Now we can **Map** the function **putToZero** over our data list. **Map** takes two arguments. The first is the function and the second is the list. For example

```
Map[putToZero, {1.2, 0.8, -0.1, 1.1, -0.2}]
{1.2, 0.8, 0, 1.1, 0}
```

◆ Example

Suppose you have a list **P** of coordinates of points in three dimensional space, and you want to create a list **R** of their distances from the origin. (Of course, in a real example the length of the list may be much longer than the one below).

```
P = {{1, 2, 0}, {0, 3, 5}, {1, 0, 2}, {a, 7, b}};
```

We can define a function to compute the distance from the origin for *one* element of the list

```
Clear[distance]

distance[{x_, y_, z_}] := Sqrt[x^2 + y^2 + z^2]
```

We test **distance** on one element to see if it works as we expect.

```
distance[{a, b, c}]
Sqrt[a^2 + b^2 + c^2]
```

Now we can **Map** it over our list **P**.

```
R = Map[distance, P]
{Sqrt[5], Sqrt[34], Sqrt[5], Sqrt[49 + a^2 + b^2]}
```

Keeping your code general

One of the principles of good programming is that *you should write your code to apply in the most general cases possible*.

In particular this means that if you write code to manipulate lists, you should write it so *lists of any length can be used*. Once you get the hang of it, this is actually easier than thinking all the time how to manage lists of fixed lengths.

We see that the inbuilt **Map** function is a good example as it will work on lists of any length, even empty ones. These special cases (like empty lists) are important, since your code might be used as part of some other application, and you do not want the other application to crash if it meets such a special case.

```
Map[distance, {}]
```

```
{}
```

However, the **distance** function that we designed above will only work for lists of *three* elements. With a little thought we can redesign it to apply to lists of *any* length (and hence be valid in spaces of any dimension, for example, 2 dimensions).

```
Clear[distance2]
```

```
distance2[x_] :=  $\sqrt{x.x}$ 
```

(In this definition we have used the inbuilt **Dot** function.)

```
FullForm[x.x]
```

```
Dot[x, x]
```

Mapping **distance2** over our original list of points **P** gives us the same result

```
R = Map[distance2, P]
```

```
{ $\sqrt{5}$ ,  $\sqrt{34}$ ,  $\sqrt{5}$ ,  $\sqrt{49 + a^2 + b^2}$ }
```

But we can also map this new definition over a more general list of lists **Q** of any lengths.

```
Q = {{}, {1}, {a, b}, {x, y, z}, {1, 2, 3, 4}};
```

```
Map[distance2, Q]
```

```
{0, 1,  $\sqrt{a^2 + b^2}$ ,  $\sqrt{x^2 + y^2 + z^2}$ ,  $\sqrt{30}$ }
```

The principles of good programming that we have discussed in this section are then

Write your program to work for special cases like empty lists.

Write your program to work for lists of any length.

Mapping functions over expressions

In this section we will begin to see the usefulness of the fact that all expressions in Mathematica have the same internal form. Mapping a function over a list is just a particular case of mapping a function over any expression.

For example, we had

```
Map[Log, {a, b, c}]
{Log[a], Log[b], Log[c]}
```

But instead of the *list* $\{a, b, c\}$, we could have had the *sum* $a+b+c$

```
Map[Log, a + b + c]
Log[a] + Log[b] + Log[c]
```

Or indeed, *any expression*

```
Map[Log, f[a, b, c]]
f[Log[a], Log[b], Log[c]]
```

Thus, **Map** works on the elements (arguments) of any expression and ignores the **Head** (in this case **f**), of the expression.

◆ Example

Suppose we had an expression involving the speed **N**, power **H**, torque **T** and efficiency η of an engine, and we wanted to modify the symbols by placing a "bar" over each of them.

OverBar[x] gives \bar{x} , so we could **Map** the function **OverBar** over the **Engine** function:

```
Map[OverBar, Engine[N, H, T, η]]
Engine[ $\bar{N}$ ,  $\bar{H}$ ,  $\bar{T}$ ,  $\bar{\eta}$ ]
```

Most of your use of **Map** will be with lists, but later, in more sophisticated programming its applicability to expressions with any Head will be very useful.

Mapping a function over a list is just a particular case of mapping a function over any expression.

Shorthand for Map

Map is so commonly used there is a shorthand for it. To **Map** a function **g** over an expression **f[x, y, ...]** you can write the operation in the form already discussed

```
Map[g, f[x, y, z]]
f[g[x], g[y], g[z]]
```

Or, you can use the shorthand /@

```
g /@ f[x, y, z]
f[g[x], g[y], g[z]]
```

You can read this as **g mapped at f[x,y,z]**, or as **g mapped over f[x,y,z]**.

Exercises

◆ Exercise: Mapping a function over a list of data

You have a list of data, and you want to create a new list of *logs* of the data elements. Make up a list of data and use **Map** to accomplish this.

◆ Exercise: Mapping a function over a list of data

You have a *list of lists* of data, and you want to find the number of data elements in each of the data lists.

Make up a list of lists of data using the **Table** and **Random** functions. Use the **Random** function to generate both the data elements *and* the lengths of the lists of data elements.

(*Hint: Random[Integer, {0,5}]* generates a random integer between 0 and 5).

Use **Map** to find the list of numbers of data elements in the sub-lists.

Repeat this several times by reentering the expression with which you generated your data to get a new data example.

◆ Exercise: Mapping a function over a list of data

You have a list of lists of data. Use **Map** to reverse the order of the elements in each of the sub-lists. Use the random list of lists generator that you developed above.

◆ Exercise: Mapping a function over a list of data

You have a list **P** of sound pressure levels p_i in Pa, and you want to convert them to dB using the conversion $\text{dB} = 10 \text{Log} \left[\left(\frac{p_i}{2 \times 10^{-5}} \right)^2 \right]$.

Generate a test list of sound pressure levels **P** using the **Table** and **Random** functions. Write a function to convert *one* pressure level to dB, and use **Map** to map this over **P**.

◆

6.3 Apply, Thread and Outer

Apply

Whereas **Map** acts on the arguments of an expression, **Apply** simply changes the **Head**. For example

```
Apply[List, a + b + c]
{a, b, c}
```

Apply[**List**,**a+b+c**], has changed the head **Plus** of the expression **Plus**[**a,b,c**] to **List**.

The shorthand for **Apply** is **@@**. For example

```
List @@ (a + b + c)
{a, b, c}
```

You can read this as **List applied to (a+b+c)**.

Sometimes you can get the same results with a rule. (We saw an example of this earlier with **gPlot**.)

```
(a + b + c) /. Plus → List
{a, b, c}
```

You will often use **Apply** when you want to construct a function by assembling its arguments, but not have it evaluated until the last moment when you change the Head of the function with **Apply**.

◆ Example

Calculate the average of the elements of a list **X** of data. For example if

```
X = {1.7, 7.9, 5.9, 0, 5, 3.4};
```

The average of the data is obtained by writing

```
(Plus @@ X) / Length [X]
3.98333
```

Exercise

◆ Exercise: Using Apply

Write a function called **average**, which takes a list as an argument and which calculates the average of the elements of the list. Your function should give the result **none** if the list is empty. *Hint:* Use **If** to deal with this special case, using **===** (whose **FullForm** is **SameQ**) in the first argument of **If** to check if the list is empty.

◆

Thread

We can use **Thread** to force functions which are not listable to be effectively listable.

Thread is particularly useful if we want to make multiple substitutions at the same time. Suppose we have an expression for the ratio of tensions **P** in a Vee-belt of sheave angle β , angle of wrap ϕ , and coefficient of friction μ .

$$P = e^{\frac{\mu \phi}{\sin[\beta]}} ;$$

We know we can evaluate the formula for given values of these design parameters by entering, a list of rules, for example

```
P /. {μ → 0.4, β → 18°, φ → π}
58.356
```

But often we will find it convenient to have the symbols and the data in separate lists, and so we would find it convenient to enter the evaluation in the form

```
P /. {μ, β, φ} → {0.4, 18°, π}
```

This will not work because Rule is not listable. However, by applying **Thread** to $\{\mu, \beta, \phi\} \rightarrow \{0.4, 18^\circ, \pi\}$, we get the list of rules we want.

```
Thread[{μ, β, φ} → {0.4, 18°, π}]
{μ → 0.4, β → 18°, φ → π}
```

And we can now use this to substitute the values in the form we want.

```
P /. Thread[{μ, β, φ} → {0.4, 18°, π}]
58.356
```

Exercises

◆ Exercise: Using **Thread** to create a list of rules

You have a list of symbols and a list of data and you want to generate a list of rules which can be used for substitution. Use **Thread** on the two lists to create the list of rules.

◆ Exercise: Exploring **Thread**

Explore the general use of **Thread** by entering

```
Thread[f[{x, y, z}, {u, v, w}]]
```

Outer

Outer is quite a powerful function for manipulating multiple lists and arrays of variables. Below we only introduce its capability. You may often find it useful when you need to generate all the combinations of elements of lists, where you can specify the *way* in which the elements are combined, for example adding (**Plus**), multiplying (**Times**), or pairing (**List**), ...

Outer basically does for functions of several variables what **Map** does for functions of one variable. For example

```
Outer[Plus, {a, b, c}, {x, y}]  

{{a + x, a + y}, {b + x, b + y}, {c + x, c + y}}
```

Of course the order of the lists is important.

```
Outer[Plus, {x, y}, {a, b, c}]  

{{a + x, b + x, c + x}, {a + y, b + y, c + y}}
```

◆ Example

You can construct the matrix resulting from the product of a *column vector* **{a,b,c}** into a *row vector* **{x,y,z}** by using **Outer**

```
MatrixForm[Outer[Times, {a, b, c}, {x, y, z}]]  


$$\begin{pmatrix} a x & a y & a z \\ b x & b y & b z \\ c x & c y & c z \end{pmatrix}$$

```

Another way to construct this type of matrix is to use the inbuilt **Dot** function

```
MatrixForm[Transpose[{{a, b, c}}].{{x, y, z}}]
```

$$\begin{pmatrix} ax & ay & az \\ bx & by & bz \\ cx & cy & cz \end{pmatrix}$$

Exercise

◆ Exercise: Using Outer with general functions

Suppose you have two lists of data and you want to discover how often a data element in the first list is repeated anywhere in the second list.

Using **Outer** and **SameQ**, create a matrix testing all the combinations.

◆

6.4 Anonymous Functions

Defining a function without giving it a name

We have already learned to define functions by means of the form $\mathbf{f}[\mathbf{x}_] := \dots$. For example a function which squares its argument could be written

$$\mathbf{f}[\mathbf{x}_] := \mathbf{x}^2$$

The entity that is defined by this definition is \mathbf{f} , which should be thought of as the actual function. The \mathbf{x} in the definition is a dummy variable which is not really there at all.

\mathbf{f} is called a *named* pure function.

We can either apply \mathbf{f} to one argument

```
 $\mathbf{f}[2]$ 
```

```
4
```

Or, if we want to **Map** the function \mathbf{f} over a list, we will get the corresponding list of squares.

```
Map[ $\mathbf{f}$ , {1, 2, 3, 4}]
```

```
{1, 4, 9, 16}
```

We can also write an *anonymous* pure function which squares its argument in the form $\#^2 \ \& .$ We apply it, just as we do \mathbf{f} , to an argument using the square brackets:

```
(#2 &) [2]
4
```

Or, we can map it over a list

```
Map[#2 &, {1, 2, 3, 4}]
{1, 4, 9, 16}
```

Here, we have not bothered to enter a definition separately such as $f[x_]:=x^2$, but simply written down the anonymous function $\#^2 \ \&$ *directly* in the form required to accomplish our goal of squaring the argument.

The syntax is that you use the $\#$ instead of the x on the right hand side of $f[x_]:=x^2$ and then finish it off with an ampersand $\&$.

Named functions are more powerful and general than anonymous functions, but sometimes anonymous functions can be convenient.

You can also have anonymous functions of any number of arguments. The i^{th} argument is represented by $\#i$. For example

```
((#1)3 + 2 (#2)2 + #3) &[a, b, c]
a3 + 2 b2 + c
```

◆ Example

Consider the example introduced at the beginning of this topic. The problem was to process a list of data, putting any negative values to zero. We defined a function **putToZero**

```
putToZero[x_] := If[x < 0, 0, x]
```

And then mapped it over our data list. For example

```
Map[putToZero, {1.2, 0.8, -0.1, 1.1, -0.2}]
{1.2, 0.8, 0, 1.1, 0}
```

An alternative would have been to use an anonymous function directly

```
Map[If[# < 0, 0, #] &, {1.2, 0.8, -0.1, 1.1, -0.2}]
{1.2, 0.8, 0, 1.1, 0}
```

Exercise

◆ Exercise: Writing and testing anonymous functions

Write and test anonymous functions to:

- 1) Square an argument.
- 2) Square an argument and add 1.
- 3) Test whether an argument is greater than 2.
- 4) Test whether an argument is the symbol \mathbb{F} (`(ESC)dsF(ESC)`).
- 5) Adds the **Floor** and the **Ceiling** of a number.

◆

6.5 Nest, NestList and FixedPoint

Nest

In engineering we often need to do calculations that repeatedly apply a function to the result of the previous calculation until the result does not change any more. For example we might want to keep on applying Newton's method to find the root of an equation.

The function that accomplishes this is **Nest**. **Nest** applies a function a given number of times to an initial expression. For example, we can apply a function **f** to the initial value **a** to get **f[a]**, then apply **f** again to the result **f[a]** to get **f[f[a]]**, then apply **f** again to the result **f[f[a]]** to get **f[f[f[a]]]**, ...

```
Nest[f, a, 6]
f[f[f[f[f[f[a]]]]]]
```

We could use **Nest** to generate repeated fractions

```
Nest[1 + 1/# &, a, 4]
```

$$1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{a}}}}$$

NestList

NestList does the same thing, but lists the results from each iteration.

```
NestList [  $1 + \frac{1}{\#}$  &, a, 4 ]
```

$$\left\{ a, 1 + \frac{1}{a}, 1 + \frac{1}{1 + \frac{1}{a}}, 1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{a}}}, 1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{a}}}} \right\}$$

Exercise

◆ Exercise: Exploring **NestList**

Discover using **NestList** what happens as you continue to apply **Cos** to a real number.

◆

FixedPoint

The function **FixedPoint** is equivalent to continuing to apply **Nest** *until there is no further change*. That is, you do not have to specify how many times to apply the function to your original starting value.

Of course you have to be fairly confident that the result will actually converge to something fixed, and not just head off to infinity or oscillate forever. It is a good idea to check if convergence is likely by first using **NestList**.

For example applying **FixedPoint** to the **Cos** function, starting with a real number gives

```
FixedPoint [ Cos, 1.0 ]
```

0.739085

But if we are tempted to start with 1 instead of 1.0, we would find from **NestList** that the results are not going to converge because **Cos**[1] is not evaluated to a real number like **Cos**[1.0] is.

```
NestList [ Cos, 1, 5 ]
```

```
{ 1, Cos [ 1 ], Cos [ Cos [ 1 ] ], Cos [ Cos [ Cos [ 1 ] ] ],
```

```
  Cos [ Cos [ Cos [ Cos [ 1 ] ] ] ], Cos [ Cos [ Cos [ Cos [ Cos [ 1 ] ] ] ] ] }
```

◆ Example

Suppose we have a curve defined by a function $y = x^3 - 1$ and we want to find its intersection with the x-axis near the point x_0 . (Of course the real root is $x=1$, but this is just an example). Newton's method is a classical algorithm for doing this root-finding. It consists of drawing a tangent to the curve at x_0 until it hits the x-axis at some point, call it x_1 . Draw another tangent at x_1 until it hits the x-axis at x_2 , and continue like this until you find the point of intersection of the curve with the axis.

To program this first define a function (**newpoint**, say) which performs a typical step in the calculation. **newpoint** would take as arguments the function **y**, the variable **x** and the current value of the point of intersection with the axis **x_i**, and return the new point of intersection.

```
Clear[newpoint]

newpoint[y_, x_, xi_] :=  $\left(x - \frac{y}{D[y, x]}\right) /. x \rightarrow N[xi]$ 
```

We can get started, let us say at the point **x = 2**.

```
newpoint[x3 - 1, x, 2]
1.41667
```

And then feed this result back for the next iteration

```
newpoint[x3 - 1, x, %]
1.11053
```

And again

```
newpoint[x3 - 1, x, %]
1.01064
```

This process is equivalent to applying **newpoint** repeatedly to the initial **x₀** until nothing changes. Hence it is a perfect application for **FixedPoint**.

```
FixedPoint[newpoint[x3 - 1, x, #] &, 2]
1.
```

We capture this into a function **newton**, say, so our final program can be written as the pair of functions

```
Clear[newpoint, newton]

newpoint[y_, x_, xi_] :=  $\left(x - \frac{y}{D[y, x]}\right) /. x \rightarrow N[xi]$ 

newton[y_, x_, x0_] := FixedPoint[newpoint[y, x, #] &, x0]
```

Testing this out gives what we expect

```
newton[x3 - 1, x, 2]
1.
```

```
newton[Sin[z] - z +  $\frac{1}{2}$ , z, 1]
```

```
1.4973
```

Of course, the inbuilt function **FindRoot** is the function for doing this sort of thing.

```
FindRoot[x3 - 1 == 0, {x, 2}]
```

```
{x → 1.}
```

```
FindRoot[Sin[z] - z +  $\frac{1}{2}$  == 0, {z, 1}]
```

```
{z → 1.4973}
```

Exercises

◆ **Exercise: Continued function applications that converge**

Discover what happens as you continue to apply **Cos** to a random number.

◆ **Exercise: Continued function applications that do not converge**

Discover what happens as you continue to apply **Sin** to a random number. (Be careful!).

◆

6.6 Summary: When to Use Functional Programming

Don't dismember lists

As we have already seen, it is most efficient (both computationally, and for algorithmic clarity) to work on data structures (lists) *as a whole* (rather than break them up and work on them one element at a time).

Functional programming is a way of harnessing this efficiency.

Map, Apply, Thread and Outer

Map, **Apply**, **Thread** and **Outer** are functions to use in functional programming.

Remember that they are not the only ones, but are four of the major ones to get started with.

When you are manipulating the data in lists, think how you might do what you want in this order:

1. Can I use the Listability property of the functions I am using?
2. Can I use **Map**, **Apply**, **Thread**, **Outer** or other list-manipulating function?
3. Can I use a rule, or several rules to do what I want?

The functions that you want to use with **Map**, **Apply**, **Thread** and **Outer**, may be

1. Inbuilt functions.
2. Functions you have defined for yourself.
3. Anonymous functions.

Anonymous functions

Remember, you do not need anonymous functions. People just use them in simple cases if they do not want to define a function specifically (and hence tie up a symbol). (But you should know how to use them, so that at least you can read other people's programs).

Nest, NestList and FixedPoint

Sometimes we want to repeat an operation over and over until we converge on the final result we want. Usually, each stage of the repetition will use information from the previous stage or stages. This is called *iteration*.

Nest, **NestList** and **FixedPoint** (among others, for example **FindRoot**) are designed to handle iteration efficiently. If you have an iteration problem

1. Try the inbuilt iteration functions like **Nest**, **NestList**, **FixedPoint** and **FindRoot** first.
2. Only then should you put the effort into building your own.

At first these functions are somewhat hard to understand. But with time they will become second nature.

6.7 Problems

Problem 1: Calculating the centre of gravity

Write a function called **centreOfGravity**, which takes a list of masses and a list of points as arguments and which calculates a list of two values, the total mass and the centre of gravity. Your function should work for any number of points masses in a space of any dimensions. Do not worry about special cases like zero mass or empty lists at this stage.

The list of masses and *corresponding* points are to be of the form

$$\{\{m_1, m_2, \dots\}, \{\{x_1, y_1, \dots\}, \{x_2, y_2, \dots\}, \dots\}\}$$

◆

Problem 2: Smoothing data

Write a function called **smooth**, which takes a list as an argument and which replaces any negative data by the average of the data.

Hint: Use the function **average** that you have already constructed in one of the earlier exercises.



Problem 3: Exploring vehicle suspension dynamics

The following program gives the general solution for the pair $\mathbf{X}=\{\mathbf{x}[\mathbf{t}], \mathbf{v}[\mathbf{t}]\}$ where $\mathbf{x}[\mathbf{t}]$ is the vertical displacement and $\mathbf{v}[\mathbf{t}]$ is the vertical velocity of an automobile wheel as a function of time \mathbf{t} . The matrix \mathbf{A} is a matrix of coefficients that determine the suspension inertia, stiffness and damping properties.

```
Clear[suspensionDynamics]

suspensionDynamics[X_, A_, t_] :=
  DSolve[Thread[D[X, t] == A.X], X, t]
```

The application to a general matrix of coefficients would then be of the form

```
suspensionDynamics[{x[t], v[t]}, {{a, b}, {c, d}}, t]
```

By researching the **DSolve** function, understand the construction of the program, and explain the use of the **Thread** function in this case.



Problem 4: Discovering repeated data

Suppose you have two lists of data and you want to discover the total number of instances for which each data element in the first list is repeated anywhere in the second list.

Use the result you got in one of the exercises above to write a function which takes the two lists as arguments and which returns the number.

Hint: You might check out the functions **Outer** and **SameQ**. You might also find the functions **Flatten**, and **Count** to be useful.



Problem 5: Distances from a fixed point

Suppose you have a list \mathbf{P} of coordinates of points in a space of any number of dimensions, and you want to create a list of their distances from a fixed point \mathbf{Q} in the same space. Write a function to do this.



Problem 6: Newton's method for any number of variables (More difficult!)

Show that the function `newton` in the example above can be extended to finding the points of intersection of any number of equations, by treating \mathbf{x} as a list of variables (vector) and \mathbf{y} as list of equations amongst those variables (vector function of \mathbf{x}).

This example underscores a general principle in Mathematica programming: in writing a program involving many variables, the best way to start is to get it working for the fewest number. Quite often the structure of the general program will be quite close to that of the simple one.

Get a program working for the simplest case, then modify it.

We find that our function `newpoint` will need to be changed (we change it to `newpoints`) to deal with lists of variables, but `newton` still remains valid for this extended application.

```
Clear[newpoints, newton]

newpoints[y_, x_, xi_] :=
  (x - Inverse[N[Outer[D, y, x]]].y) /. Thread[x → N[xi]]

newton[y_, x_, x0_] := FixedPoint[newpoints[y, x, #] &, x0]
```

Example application:

```
newton[{a2 + b2 - 13, a3 - b3 - 19}, {a, b}, {2, 1}]
{3., 2.}
```

Compare this to the result given by the built-in `FindRoot`.

```
FindRoot[{a2 + b2 - 13 == 0, a3 - b3 - 19 == 0}, {a, 2}, {b, 1}]
{a → 3., b → 2.}
```

Although these results are the same for this simple case, `FindRoot` is of course very much more computationally efficient and powerful.