5.More Advanced Topics.

Some Other topics

Dynamic Programming

Recursive programs are programs that "refer to themselves". Such programs can be very slow because they use a lot of "stack memory". One way to speed them up is by means of "dynamic programming" or "functions that remember their values". The best way to understand this method is by means of an example.

Fast Fibonacci numbers.

```
a[n] /. RSolve[{a[n] == a[n - 1] + a[n - 2], a[0] == 0, a[1] == 1}, a[n], n][[1]]
```

F_n

```
FunctionExpand[Fibonacci[n]]
```

$$\frac{1}{\sqrt{5}}\left(\left(\frac{1}{2}\left(1+\sqrt{5}\right)\right)^n-\left(\frac{2}{1+\sqrt{5}}\right)^n\cos(\pi n)\right)$$

Let's compare the following two definitions of the Fibonacci numbers. First we start with a usual recursive definition.

```
Fib1[0] = 0; Fib1[1] = 1;
```

```
Fib1[n_] := Fib1[n - 1] + Fib1[n - 2]
```

```
Timing[Fib1[30]]
```

{2.26725, 832040}

This is very slow and already the 30th Fibonacci number takes a noticeable time to compute. Next, we try "dynamic programing". The definition looks a little strange; note the use of := and = .

```
Clear[Fib2]
```

```
Fib2[0] = 0; Fib2[1] = 1;
```

```
Fib2[n_] := Fib2[n] = Fib2[n - 1] + Fib2[n - 2]
```

Timing[Fib2[30]]

{0.000267, 832040}

Computing even the 50th Fibonacci number takes virtually no time.

We can see the difference between the two definitions by using the function Trace:

Trace[Fib1[5]]

```
{Fib1[5], Fib1[5 - 1] + Fib1[5 - 2], {{5 - 1, 4}, Fib1[4],
  Fib1[4 - 1] + Fib1[4 - 2], {{4 - 1, 3}, Fib1[3], Fib1[3 - 1] + Fib1[3 - 2],
   {{3 - 1, 2}, Fib1[2], Fib1[2 - 1] + Fib1[2 - 2], {{2 - 1, 1}, Fib1[1], 1},
    {{2 - 2, 0}, Fib1[0], 0}, 1 + 0, 1}, {{3 - 2, 1}, Fib1[1], 1}, 1 + 1, 2},
  {{4 - 2, 2}, Fib1[2], Fib1[2 - 1] + Fib1[2 - 2], {{2 - 1, 1}, Fib1[1], 1},
   {{2 - 2, 0}, Fib1[0], 0}, 1 + 0, 1}, 2 + 1, 3},
 {{5 - 2, 3}, Fib1[3], Fib1[3 - 1] + Fib1[3 - 2],
  {{3 - 1, 2}, Fib1[2], Fib1[2 - 1] + Fib1[2 - 2], {{2 - 1, 1}, Fib1[1], 1},
   {{2 - 2, 0}, Fib1[0], 0}, 1 + 0, 1}, {{3 - 2, 1}, Fib1[1], 1}, 1 + 1, 2}, 3 + 2, 5}
```

Trace[Fib2[5]]

```
{Fib2[5], Fib2[5] = Fib2[5 - 1] + Fib2[5 - 2],
 {{{5 - 1, 4}, Fib2[4], Fib2[4] = Fib2[4 - 1] + Fib2[4 - 2],
   {{{4 - 1, 3}, Fib2[3], Fib2[3] = Fib2[3 - 1] + Fib2[3 - 2],
     {{{3 - 1, 2}, Fib2[2], Fib2[2] = Fib2[2 - 1] + Fib2[2 - 2],
       {{{2 - 1, 1}, Fib2[1], 1}, {{2 - 2, 0}, Fib2[0], 0}, 1 + 0, 1}, Fib2[2] = 1, 1},
      {{3 - 2, 1}, Fib2[1], 1}, 1 + 1, 2}, Fib2[3] = 2, 2},
   {{4 - 2, 2}, Fib2[2], 1}, 2 + 1, 3}, Fib2[4] = 3, 3},
 {{5 - 2, 3}, Fib2[3], 2}, 3 + 2, 5}, Fib2[5] = 5, 5}
```

The first function repeatedly performs the same computation (see Fib1[3] and Fib2[3] in the output of Trace above).

We can also check what Mathematica knows about the functions Fib1 and Fib2

?Fib1

```
Global`Fib1

Fib1[0] = 0

Fib1[1] = 1

Fib1[n_] := Fib1[n - 1] + Fib1[n - 2]
```

DownValues[Fib1]

```
{HoldPattern[Fib1[0]] :↦ 0, HoldPattern[Fib1[1]] :↦ 1,
 HoldPattern[Fib1[n_]] :↦ Fib1[n - 1] + Fib1[n - 2]}
```

?Fib2

```
Global`Fib2
```

```
Fib2[0] = 0

Fib2[1] = 1

Fib2[2] = 1

Fib2[3] = 2

Fib2[4] = 3

Fib2[5] = 5

Fib2[n_] := Fib2[n] = Fib2[n - 1] + Fib2[n - 2]
```

The point is that the last definition makes Fib2 remember each value that it has once computed so it never has to compute it again. The result is much better performance at the cost of some memory consumption, of course. This can be recovered by using

```
Clear[Fib2]
```

```
Fibonacci[30] // Timing
```

{0.000012, 832040}

$$\text{e1} = \frac{1}{2}\left(\sqrt{5}+1\right);$$

$$\text{e2} = \frac{1}{2}\left(1-\sqrt{5}\right);$$

$$\text{b1} = \frac{1}{10}\left(\sqrt{5}+5\right);$$

$$\text{b2} = \frac{1}{10}\left(5-\sqrt{5}\right);$$

$$\text{Fib3(n\_)} := \text{Expand}\left[\text{b1}\,\text{e1}^{n-1} + \text{b2}\,\text{e2}^{n-1}\right]$$

```
Fib3[30]
```

832040

Imperative (Procedural) and Functional programming

So far we have considered two programming styles that one can use in *Mathematica* - rule based and functional. There is also another style, called procedural or imperative. This is the style used by most traditional programming languages, such as C. The characteristic of this style is the use of explicit assignments to variables, and of loops that change the state of a variable. Here is an example of a procedural program which changes the state of a variable x by using assignments:

```
Clear[x]
```

$$x = 1; x = x + 1; x = x + 1; x = x + 1; x = x + 1; x = x + 1; x = x + 1;$$

```
x
```

7

This can be also written as

```
FullForm[Hold[x = 1; x = x + 1; x = x + 1; x = x + 1; x = x + 1; x = x + 1; x = x + 1;]]
```

```
Hold[CompoundExpression[Set[x, 1], Set[x, Plus[x, 1]], Set[x, Plus[x, 1]],
    Set[x, Plus[x, 1]], Set[x, Plus[x, 1]], Set[x, Plus[x, 1]], Set[x, Plus[x, 1]], Null]]
```

Note that *Mathematica* automatically returns the value of the last argument of ComposedExpression. In most procedural languages no final value would be returned and you need an explicit Return or Print statement Return[x] or Print[x] at the end. In *Mathematica* you never need these statements for this purpose.
Below is another way to do the same thing, using a Do loop. Note that the Do loop does not return anything, so we need x at the end if we wish to return the value of x.

```
Clear[x]
```

```
x = 1; Do[x = x + 1, {6}]; x
```

```
7
```

```
x
```

```
7
```

```
x = 1; Do[x++, {6}]; x
```

```
7
```

Mathematica has other procedural loops: While and For. They should be used sparingly, particularly the last one which is very inefficient. In general you can get much better performance from Mathematica by using functional constructions: Nest, Fold and FixedPoint and in version 6, Accumulate.

Local Variables

Because in imperative programs assignments are used, one has to be careful not to accidentally use or redefine variables to which values may have been assigned earlier. The best way to protect oneself from this possibility is by means of local variables. *Mathematica* has three basic constructions for localizing variables: Block, Module and With.

```
?Block
```

Block[{*x*, *y*, ...}, *expr*] specifies that *expr* is to be evaluated with local values for the symbols *x*, *y*, ...
Block[{*x* = x_0, ...}, *expr*] defines initial local values for *x*, ... >>

```
x = 3; Block[{x}, x = 1]
```

```
1
```

```
x
```

```
3
```

Although inside Block we set the value of *x* to 1, outside it remained equal to 3. The same will happen if we use Module

```
?Module
```

Module[{*x*, *y*, ...}, *expr*] specifies that occurrences of the symbols *x*, *y*, ... in *expr* should be treated as local.
Module[{*x* = x_0, ...}, *expr*] defines initial values for *x*, ... >>

```
x = 3;
```

```
Module[{x, y, z = 1}, x = 5; y = x + z]
```

```
6
```

```
x
```

```
3
```

Block and Module work in a quite different way. When you localize a variable in Block its value is first stored, then erased, than after Block is exited the old (stored) value is restored. In the case of Module the variables are renamed so that their names do not conflict with any other names. Another construction that localizes variables is With. Note that, unlike in Module and Block, all local variables in With must be initialized so you can't use assignments to local variables in With.

```
ClearAll[f]
```

```
f[x_List] := Module[{u = Length[x], v}, v = u + 1]
```

```
f[{1, 2, 3}]
```

```
4
```

```
g[x_List] := Module[{u = Length[x], v = u + 1}, v]
```

```
g[{1, 2, 3}]
```

```
u + 1
```

```
ClearAll[g]
```

```
g[x_List] := With[{u = Length[x], v = u + 1}, v]
```

```
g[{1, 2, 3}]
```

```
1 + u
```

```
h[x_List] := Module[{u = Length[x], v = u + 1}, v]
```

```
h[{1, 2, 3}]
```

```
1 + u
```

```
g[x_List] := Block[{u = Length[x], v = u + 1}, v]
```

```
g[{1, 2, 3}]
```

```
4
```

Another important example:

```
foo := x
```

```
x = 1; foo
```

```
1
```

```
Block[{x = 2}, foo]
```

```
2
```

Although foo was definied outside Block, its value inside Block is changed. This does not happen when we use Module:

```
foo
```

1

```
Module[{x = 2}, foo]
```

1

Thus, Module depends only on the original definition, Block on the evaluation. (Lexical scoping vs dynamic scoping).

Loops and Functional Iteration

Programs written in this style change the values of some variable. In order to get the changed value one has to explicitly evaluate the variable. Here is a simple procedural program which uses the Do loop.

?Do

Do[*expr*, {i_{max}}] evaluates *expr* i_{max} times.
Do[*expr*, {*i*, i_{max}}] evaluates *expr* with the
    variable *i* successively taking on the values 1 through i_{max} (in steps of 1).
Do[*expr*, {*i*, i_{min}, i_{max}}] starts with $i = i_{min}$.
Do[*expr*, {*i*, i_{min}, i_{max}, *di*}] uses steps *di*.
Do[*expr*, {*i*, {i_1, i_2, ...}}] uses the successive values i_1, i_2, ...
Do[*expr*, {*i*, i_{min}, i_{max}}, {*j*, j_{min}, j_{max}}, ...] evaluates *expr* looping over different values of *j*, etc. for each *i*. >>

```
Timing[x = 1; Do[x++, {20 000}]; x]
```

```
{0.011248, 20 001}
```

We can do the same thing by using the functional style. When programming in this style we use functions which return values rather then change states of variables. Instead of loops we use "higher functions", that is functions whose arguments are functions. One of such functions is Nest.

?Nest

Nest[*f*, *expr*, *n*] gives an expression with *f* applied *n* times to *expr*. >>

```
Nest[f, a, 4]
```

```
f[f[f[f[a]]]]
```

There is also a related function NestList

?NestList

NestList[*f*, *expr*, *n*] gives a list of the results of applying *f* to *expr* 0 through *n* times. >>

```
NestList[f, a, 4]
```

```
{a, f[a], f[f[a]], f[f[f[a]]], f[f[f[f[a]]]]}
```

Instead of using the Do loop above we can obtain the same result using the functional approach as follows:

Nest[# + 1 &, 1, 20 000] // Timing

```
{0.000682, 20 001}
```

The program runs much faster.

FoldList[*f*, *a*, {*b*, *c*, *d*, *e*}]

```
{a, f[a, b], f[f[a, b], c], f[f[f[a, b], c], d], f[f[f[f[a, b], c], d], e]}
```

Here `f` has to be a function of two arguments. Here is an example which shows the working of FoldList:

FoldList[Plus, 0, {*a*, *b*, *c*, *d*}]

```
{0, a, a + b, a + b + c, a + b + c + d}
```

FoldList[Times, 1, {*a*, *b*, *c*, *d*}]

```
{1, a, a b, a b c, a b c d}
```

Finally there is one new and very useful function that appeared in *Mathematica* 6.

? Accumulate

Accumulate[*list*] gives a list of the successive accumulated totals of elements in *list*. ≫

Accumulate[{*a*, *b*, *c*, *d*, *e*}]

```
{a, a + b, a + b + c, a + b + c + d, a + b + c + d + e}
```

The same result can be achieved using FoldList, however, Accumulate, being a more specialised function, is considerably faster.

FoldList[Plus, 0, {*a*, *b*, *c*, *d*, *e*}]

```
{0, a, a + b, a + b + c, a + b + c + d, a + b + c + d + e}
```

ls = RandomInteger[{1, 100}, {1000}];

***a* = (Accumulate[ls]; // Timing)**

```
{0.00005, Null}
```

***b* = (Rest[FoldList[Plus, 0, ls]]; // Timing)**

```
{0.000465, Null}
```

First[*b*] / First[*a*]

```
9.3
```

Last[*a*] == Last[*b*]

```
True
```

This much greater speed of a more specialized function compared with a more general one is typical of *Mathematica* programming.

```
http://reference.wolfram.com/mathematica/tutorial/
  ApplyingFunctionsRepeatedly.html
```

Block and global variables.

One of the most common uses of Block is to change temporarily the value of a global variable. For example, the global variable $RecursionLimit has by defaul the value:

```
$RecursionLimit
```

```
256
```

The reason for this is to stop accidental infinte recurssion from occuring as a result of programming errors. However, this can sometimes be inconvenient. Here is a familiar example.

```
Clear[Fib]
```

```
Fib[1] = 1; Fib[2] = 1; Fib[n_] := Fib[n] = Fib[n - 1] + Fib[n - 2];
```

```
Fib[3000]
```

```
$RecursionLimit::reclim : Recursion depth of 256 exceeded. >>
```

The value of 256 for $RecursionLimit prevents the code from working. Using Block we can temporarily change this value:

```
Clear[Fib]
```

```
Fib[1] = 1; Fib[2] = 1; Fib[n_] := Fib[n] = Fib[n - 1] + Fib[n - 2];
```

```
Block[{$RecursionLimit = 10 000}, Fib[3000]]
```

```
410 615 886 307 971 260 333 568 378 719 267 105 220 125 108 637 369 252 408 885 430 926 905 ⁖
  584 274 113 403 731 330 491 660 850 044 560 830 036 835 706 942 274 588 569 362 145 476 502 ⁖
  674 373 045 446 852 160 486 606 292 497 360 503 469 773 453 733 196 887 405 847 255 290 082 ⁖
  049 086 907 512 622 059 054 542 195 889 758 031 109 222 670 849 274 793 859 539 133 318 371 ⁖
  244 795 543 147 611 073 276 240 066 737 934 085 191 731 810 993 201 706 776 838 934 766 764 ⁖
  778 739 502 174 470 268 627 820 918 553 842 225 858 306 408 301 661 862 900 358 266 857 238 ⁖
  210 235 802 504 351 951 472 997 919 676 524 004 784 236 376 453 347 268 364 152 648 346 245 ⁖
  840 573 214 241 419 937 917 242 918 602 639 810 097 866 942 392 015 404 620 153 818 671 425 ⁖
  739 835 074 851 396 421 139 982 713 640 679 581 178 458 198 658 692 285 968 043 243 656 709 ⁖
  796 000
```

However, note that the global value of $RecursionLimit remains unchanged:

```
$RecursionLimit
```

```
256
```

Example 1: Simulating Brownian Motion

One Path

```
BrownianMotion[n_] := Accumulate[Prepend[RandomReal[NormalDistribution[0, Sqrt[1/n]], {n}], 0]]
```

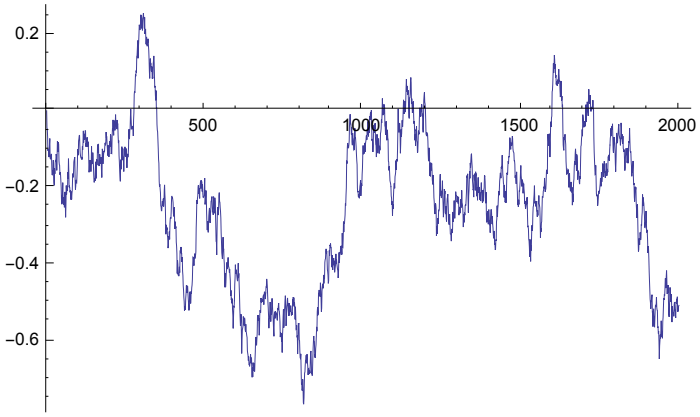
slower alternative

```
BrownianMotion[n_] :=
 FoldList[Plus, 0, RandomReal[NormalDistribution[0, Sqrt[1 / n]], {n}]]
```

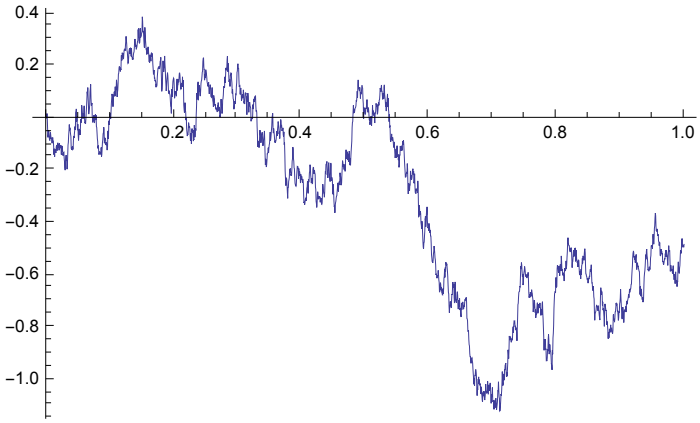
and even slower

```
BrownianMotion[n_] :=
 NestList[# + RandomReal[NormalDistribution[0, Sqrt[1 / n]]] &, 0, n]
```

```
ListLinePlot[BrownianMotion[2000]]
```




```
ListLinePlot[BrownianMotion[2000], DataRange → {0, 1}]
```

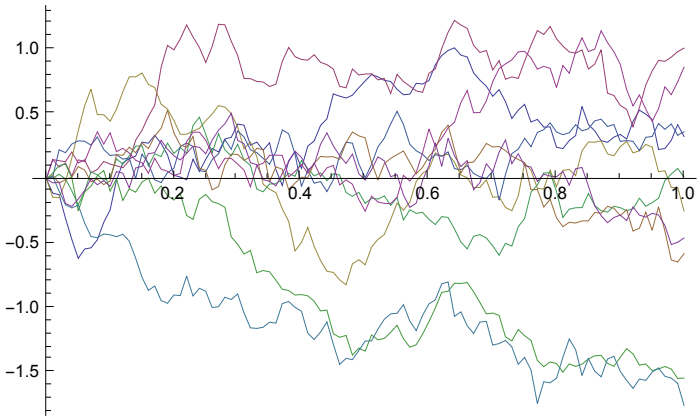


Many Paths

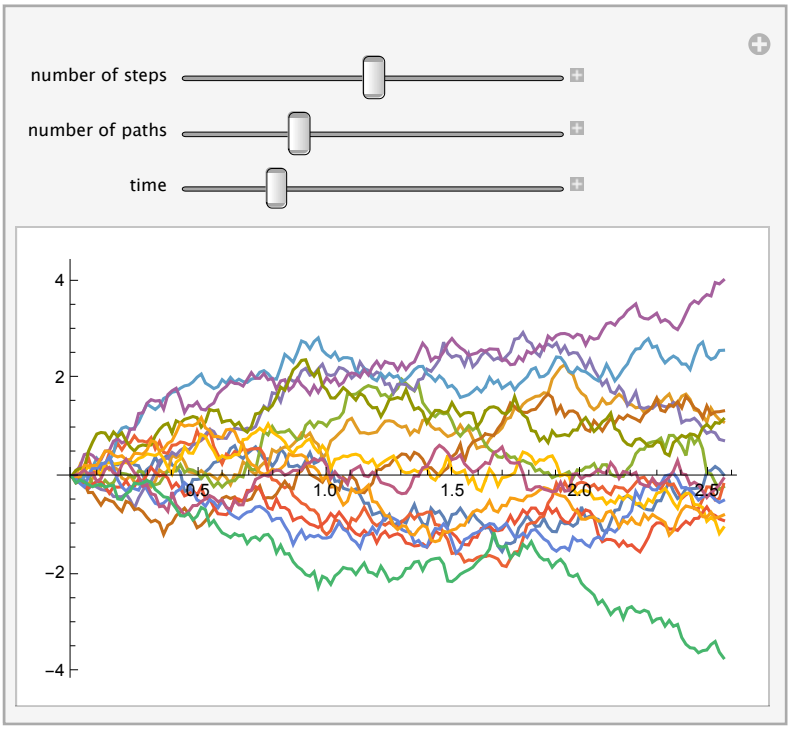
```
Clear[BrownianMotion]
```

```
BrownianMotion[time_, steps_, paths_] := Transpose[Accumulate[Join[{ConstantArray[0, paths]},
      Transpose[RandomReal[NormalDistribution[0, Sqrt[time/steps]], {paths, steps}]]]]]
```

```
ListLinePlot[BrownianMotion[1, 100, 10], DataRange → {0, 1}, PlotRange → All]
```

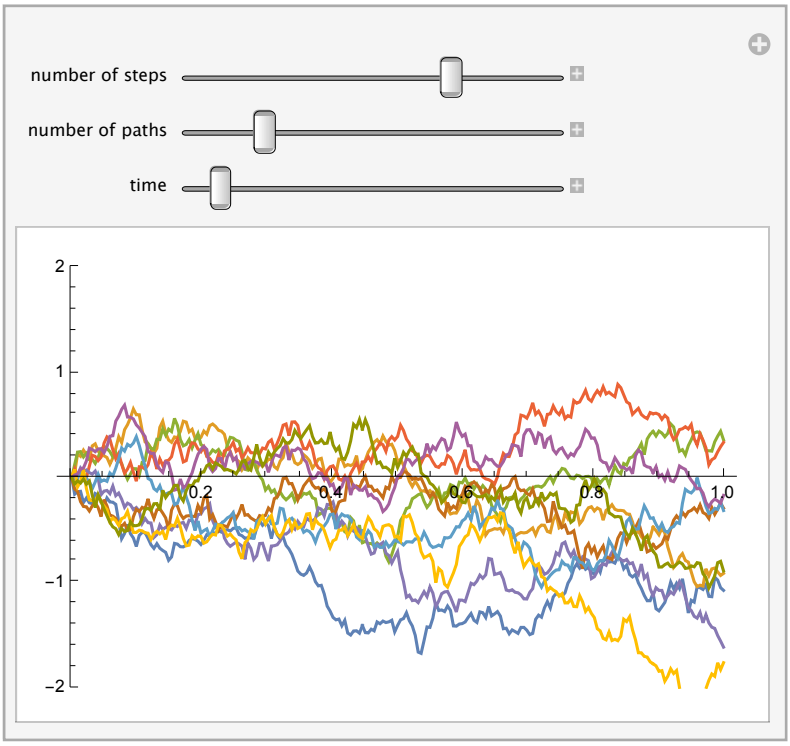



```
Manipulate[ListLinePlot[BrownianMotion[time, steps, paths], DataRange → {0, time}, PlotRange → All],
  {{steps, 100, "number of steps"}, 10, 300, 1}, {{paths, 10, "number of paths"}, 1, 50, 1},
  {{time, 1, "time"}, 0.5, 10}, SaveDefintions → True]
```

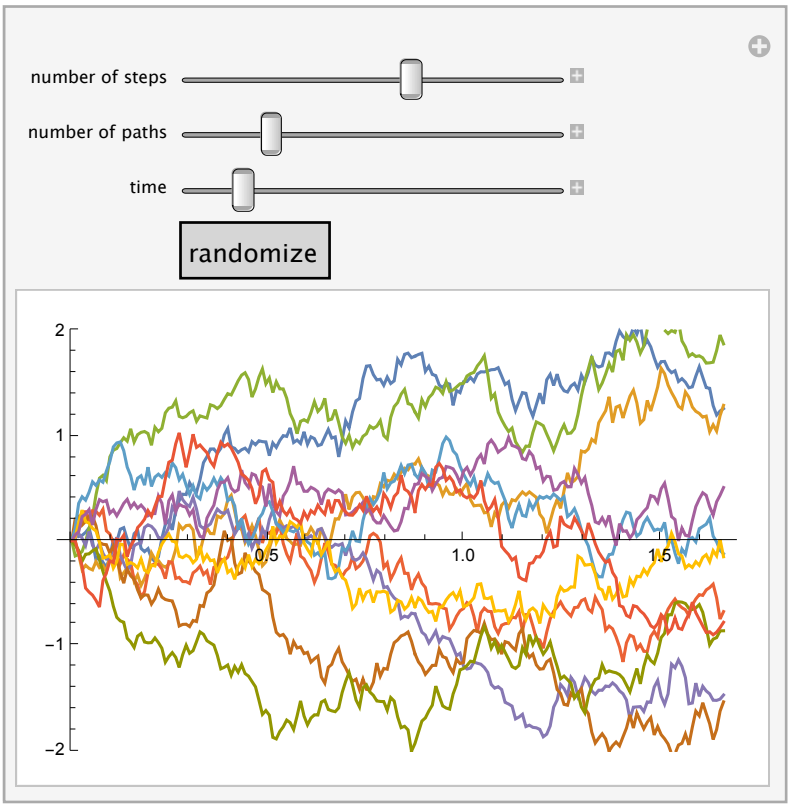


ListLinePlot::lpn : BrownianMotion[2.56, 156, 15] is not a list of numbers or pairs of numbers. ≫

```
Manipulate[BlockRandom[
    ListLinePlot[BrownianMotion[time, steps, paths], DataRange → {0, time}, PlotRange → {−2, 2}]],
  {{steps, 100, "number of steps"}, 10, 300, 1}, {{paths, 10, "number of paths"}, 1, 50, 1},
  {{time, 1, "time"}, 0.5, 10}, SaveDefitition → True, Initialization :→
    (BrownianMotion[time_, steps_, paths_] := Transpose[Accumulate[Join[{ConstantArray[0, paths]},
          RandomReal[NormalDistribution[0, Sqrt[time/steps]], {steps, paths}]]]])]
```



```
Manipulate[BlockRandom[SeedRandom[r];
    ListLinePlot[BrownianMotion[time, steps, paths], DataRange → {0, time}, PlotRange → {−2, 2}]],
  {{steps, 100, "number of steps"}, 10, 300, 1}, {{paths, 10, "number of paths"}, 1, 50, 1}, {{time, 1, "time"},
    0.5, 10}, {{r, 0, ""}, Button["randomize", r = RandomInteger[2^64 − 1]] &}, SaveDefitition → True,
  Initialization :→ (BrownianMotion[time_, steps_, paths_] := Transpose[Accumulate[Join[{ConstantArray[0,
        paths]}, RandomReal[NormalDistribution[0, Sqrt[time/steps]], {steps, paths}]]]])]
```



http://reference.wolfram.com/mathematica/tutorial/PseudorandomNumbers.html

---

A few words about Dynamic and Manipulate

In version 6 of Mathematica, new features appeared which made it possible to use the Front End in a new way. The main idea is that expressions with head Dynamic are updated when their "displayed form" changes. The simplest case is:

Dynamic

```
Dynamic[x]
```

0.

```
x = 5
```

5

```
DateString[]
```

Wed 16 Nov 2011 14:29:47

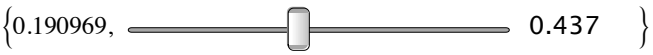
```
Dynamic[Refresh[DateString[], UpdateInterval → Infinity]]
```

Mon 21 Jul 2014 15:22:15

```
{Slider[Dynamic[x], Appearance → "Labeled"], Dynamic[x^2]}
```


```
Slider[Dynamic[x]]
```

```
DynamicModule[{x}, {Dynamic[x^2], Slider[Dynamic[x], Appearance → "Labeled"]}]
```




```
DynamicModule[{x},
 {Dynamic[x^2], Slider[Dynamic[x], {0, 10, 1}, Appearance → "Labeled"]}]
```


```
DynamicModule[{x}, {Dynamic[x^2], PopupMenu[Dynamic[x], Range[10]]}]
```


```
DynamicModule[{x}, {Dynamic[x^2], SetterBar[Dynamic[x], Range[10]]}]
```

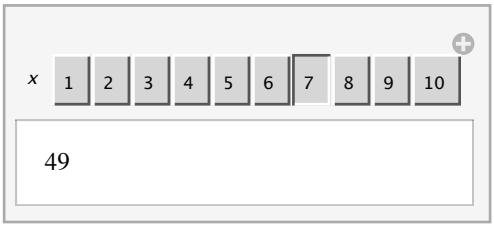

```
DynamicModule[{x = 1},
 {Dynamic[x], Dynamic[Slider[x, {0, 1}, Appearance → "Labeled"]]}]
```


```
DynamicModule[{x = 1},
 {Dynamic[x], Slider[Dynamic[x], {0, 1}, Appearance → "Labeled"]}]
```

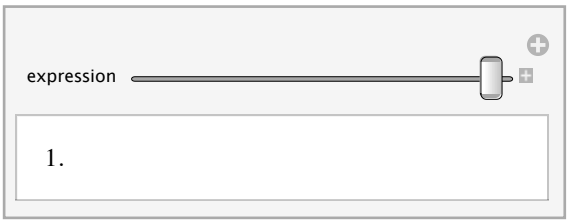

```
DynamicModule[{n = 1}, Row[{Dynamic[Plot[x^n, {x, -1, 1}, PlotRange → All]],
   Slider[Dynamic[n], {0, 5, 1}, Appearance → "Labeled"]}]]
```

Manipulate

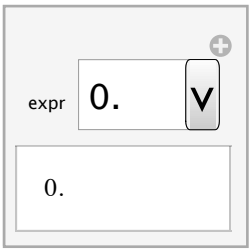
```
Manipulate[x^2, {x, 1, 10, 1, SetterBar}]
```



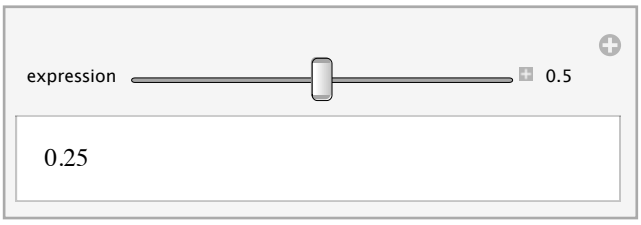

```
Manipulate[#^2 &@expr, {{expr, 0, "expression"}, 0, 1, 0.1}]
```



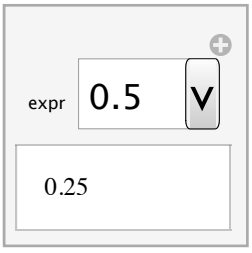

```
Manipulate[#^2 &[expr], {expr, Table[i, {i, 0, 1, 0.1}]}]
```



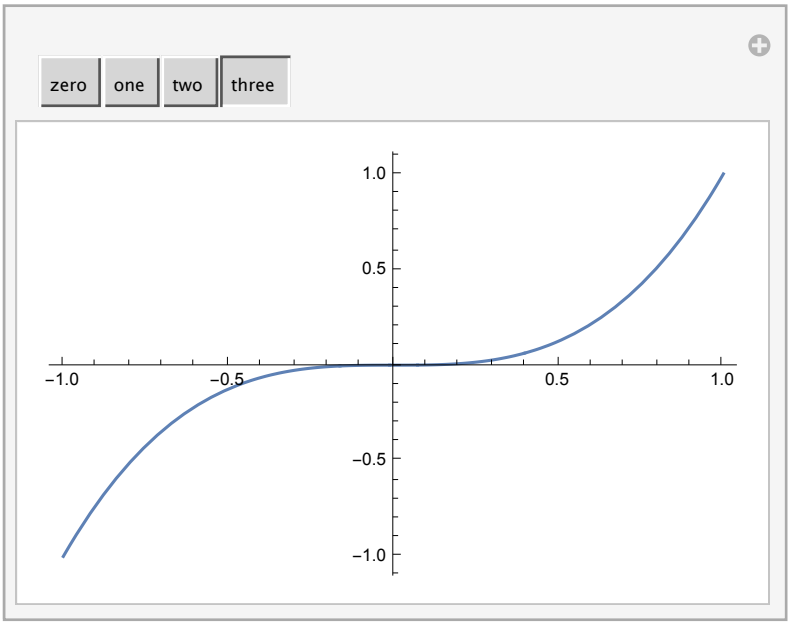

```
Manipulate[#^2 &@expr,
 {{expr, 0, "expression"}, 0, 1, 0.1, Appearance → "Labeled"}]
```



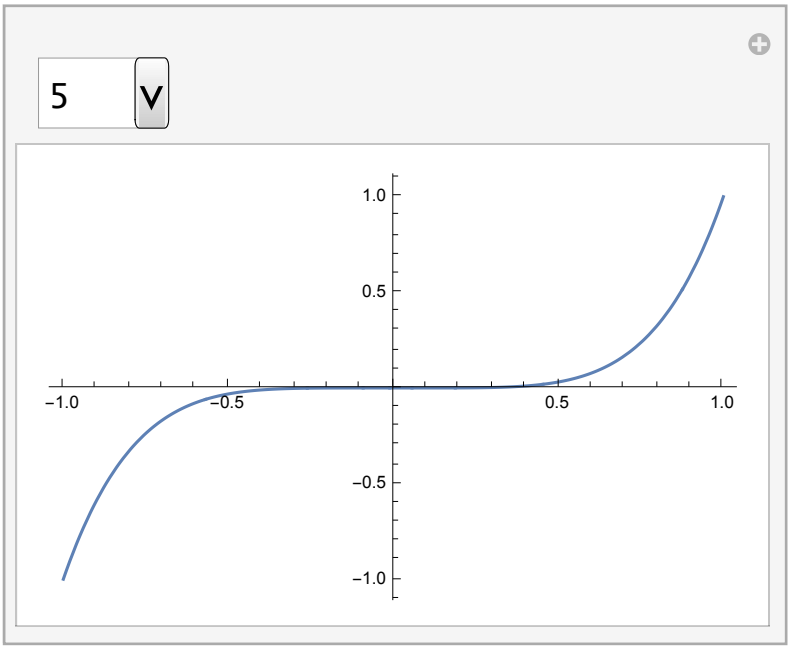

```
Manipulate[#^2 &[expr], {expr, Table[i, {i, 0, 1, 0.1}]}]
```



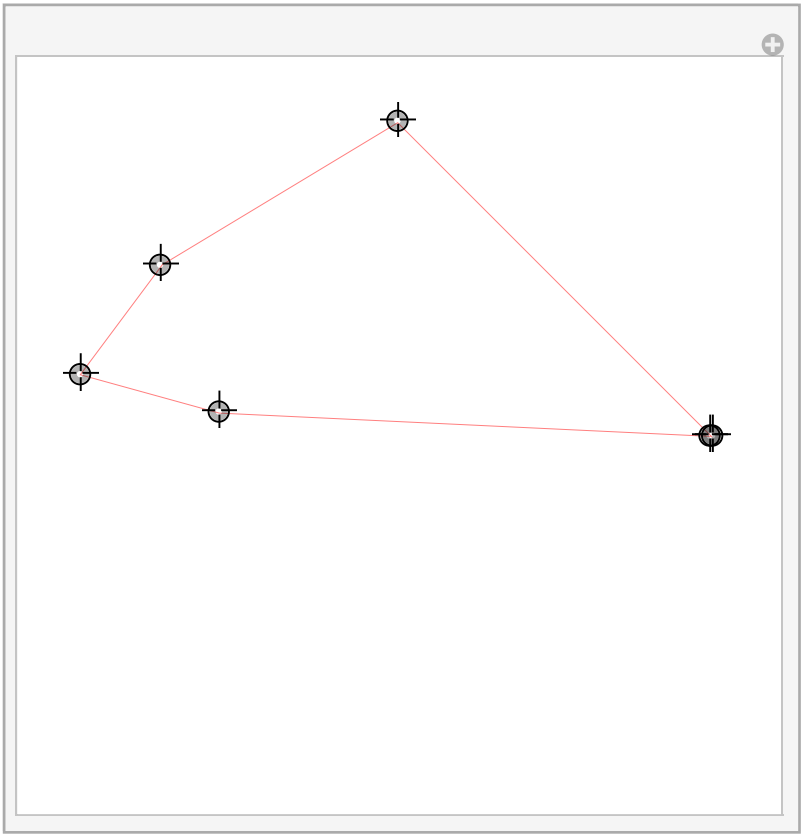
```
Manipulate[Plot[x^n, {x, -1, 1}, PlotRange → All],
 {{n, 0, ""}, {0 → "zero", 1 → "one", 2 → "two", 3 → "three"}}]
```



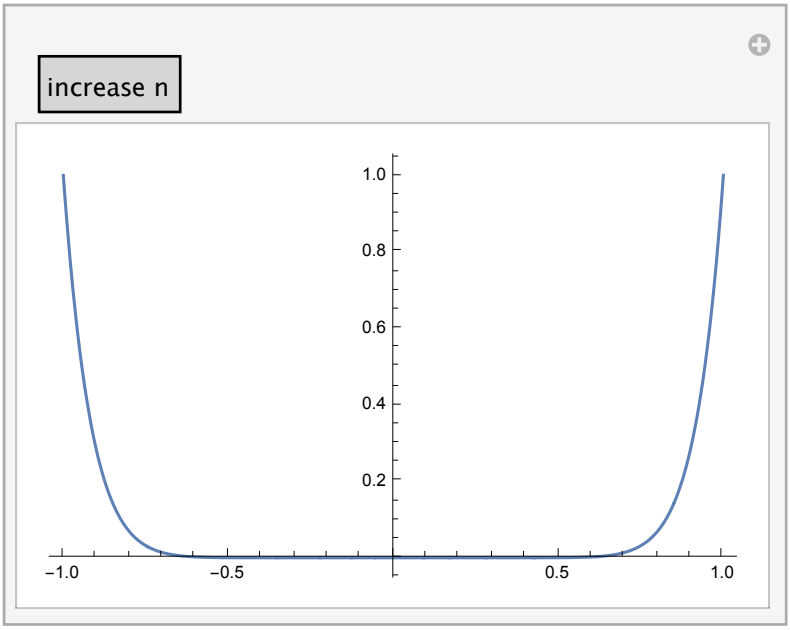

```
Manipulate[Plot[x^n, {x, -1, 1}, PlotRange → All],
 {{n, 0, ""}, Evaluate[Table[i → ToString[i], {i, 0, 10}]]}]
```



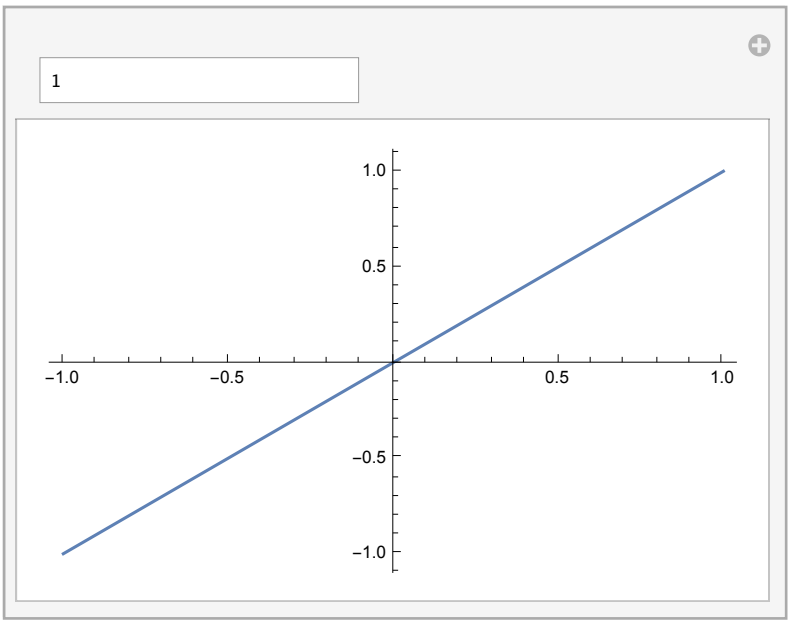
```
Manipulate[
 Graphics[{Pink, Line[pts]}, PlotRange → 1.1],
 {{pts, {{0, 0}, {1, 0}, {0, 1}}}, Locator, LocatorAutoCreate → True}]
```

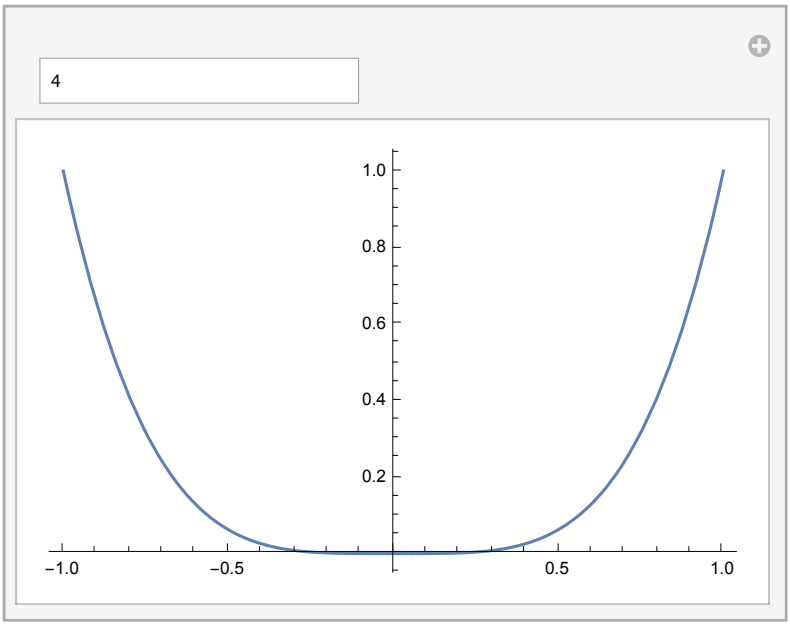


```
Manipulate[Plot[Tooltip[x^n, "x"^n], {x, -1, 1}, PlotRange → All],
 {{n, 1, ""}, Button["increase n", n = n + 1] &}]
```



```
Manipulate[Plot[Tooltip[x^n, "x"^n], {x, -1, 1}, PlotRange → All],
 {{n, 1, ""}, InputField}]
```





Links

http://
 reference.wolfram.com/mathematica/tutorial/IntroductionToDynamic.html

http://reference.wolfram.com/mathematica/tutorial/
  IntroductionToManipulate.html