5.Bardziej zaawansowane tematy.

Programowanie rekursywne i dynamiczne

Programy rekursywne to programy które "odwołują się do siebie samych". W *Mathematice* bardzo łatwo jest programować rekursywnie choć w przeciwieństwie do języków programistycznych takich jak LISP (do których język *Mathematiki* ma pewne podobieństwo) Mathematica nie jest zoptymalizowana do tego celu i nie uważnie napisane programy rekursywne mogą być bardzo powolne. Jednym ze sposobów przyspieszenia ich jest metoda zwana "programowaniem dynamicznym", lub, bardziej precyzyjnie, metoda "funkcji które zapamiętują swoje wartości". Jest to bardzo pożyteczna metoda którą najłatwiej zrozumieć rozpatrując przykłady jej użycia.

Szybkie obliczanie liczb Fibonacciego.

Liczby Fibonacciego są rozwiązaniami równania rekursywnego

$a(n) = a(n-2) + a(n-1)$
$a(0) = 0$
$a(1) = 1$

Funkcja RSolve rozwiązuje wiele równań tego typu:

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

```
Fibonacci[n]
```

Fibonacci[*n*] to po prostu zakodowana *n*-ta liczba Fibonacciego. Aby zobaczyć definicję używamy FunctionExpand:

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

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

Teraz spróbujemy sami zdefiniować liczby Fibonacciego. Zrobimy to na dwa sposoby. Pierwszy, to "zwyczajna rekursja".

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

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

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

```
{1.85534, 832040}
```

Niestety, to podejście jest bardzo powolne i już obliczenie 30tej liczby Fibonacciego zabiera zauważalną ilość czasu. Następnie użyjemy "programowania dynamicznego". Na pierwszy rzut oka

definicja wygląda dziwnie. Należy zwrócić uwagę na równoczesne użycie := i =.

```
Clear[Fib2]
```

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

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

```
Timing[Fib2[30]]
```

```
{0.000266, 832 040}
```

Tym razem działa to błyskawicznie.

Co się dzieje podczas wykonywanie tych dwóch definicji-programów można prześledzić za pomocą funkcji Trace. Rożnicę widać wyraźnie:

```
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}
```

Widzimy że Fib1 wielkorotnie powtarza te same obliczenia.

W przypadku Fib2 pierwsze oblicznie i następne są zupełnie inne. Najpierw usuńmy jesze raz Fib2 i definiujmy funkcję od początku:

```
Clear[Fib2]
Fib2[0] = 0; Fib2[1] = 1;
Fib2[n_] := Fib2[n] = Fib2[n - 1] + Fib2[n - 2]
```

```
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}
```

Teraz wywołajmy Fib2 jeszcze raz, tym razem dla $n = 6$:

```
Trace[Fib2[6]]
```

```
{Fib2[6], Fib2[6] = Fib2[6 - 1] + Fib2[6 - 2],
  {{{6 - 1, 5}, Fib2[5], 5}, {{6 - 2, 4}, Fib2[4], 3}, 5 + 3, 8}, Fib2[6] = 8, 8}
```

Widzimy wyraźnie że *Mathematica* zapamiętała wartości Fib2[*n*] dla $n \leq 5$ i więcej nie musiała ich obliczać.

Możemy także sprawdzić bezpośrednio co Matheamtica wie o Fib1 i Fib2

```
?Fib1
```

```
Global`Fib1
```

```
Fib1[0] = 0

Fib1[1] = 1

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]
```

Oczywiście to duży zysk w szybkości w obliczeniach dzięki użyciu programowania dynamicznego ma pewien koszt w użytej pamięci. Pamięć możemy odzyskać przez:

```
Clear[Fib2]
```

Imperatywne (proceduralne) i funkcyjne programowanie

Dotychczas rozważaliśmy dwa paradygmaty programowania używane w Mathematice: programowanie oparte na regułach oraz funkcyjne. Teraz przejdziemy do programowania proceduralnego zwanego także imperatywnym. Jest to oczywiście najbardziej powszechny paradygmat programowania, odający dosyć wiernie sposób działania większości współczesnych komputerów. Typowe dla tego paradygmatu jest przypisywanie wartości zmiennym i używanie pętli w celu zmiany tych wartości. Zaczniejmy od przykładu.

```
Clear[x]
```

```
x = 1; x = x + 1; x = x + 1; x = x + 1; x = x + 1; x = x + 1; x = x + 1;
```

Zauważmy że nie otrzymaliśmy żadengo wyniku. Tymniemniej *x* ma oczekiwaną wartość.

```
x
```

```
7
```

Przyjrzyjmy się formie wewnętrznej (FullForm) obliczanego wyrażenia:

```
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]]
```

Zauważmy że *Mathematica* automatycznie daje nam wartość ostatniego wyrażenia w ComposedExpression. Jeśli więc nie chcemy tego wyrażenia otrzymać, używamy ostatniego argumentu Null.
W wielu proceduralnych językach wartości zmiennej x nie zwracana o ile nie użyje się Return[x] lub Print[x]. Ponieważ Mathematica zawsze zwraca wartość obliczanego wyrażenia, Return[x] służy do zupełnie innego celu. Jeśli na końcu pętli w programie napisanym w Mathematice widzimy Return[x] (zamiast prostego x) możemy być prawie pewni że autor nauczył się programować w języku Fortran (lub C) i nadal pisze
programy w tym języku. Nie jest to najbardziej wydajne i eleganckie podejście do programowania w Mathematice ale działa! Teraz powtórzymy to samo używając pętli Do. Zauważmy że sama pętla Do nic nie zwraca - dla tego na końcu kodu umieściliśmy *x* :

```
Clear[x]
```

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

```
7
```

To samo można zapisać nieco krócej:

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

```
7
```

Mathematica ma także inne pętle, np. While i For. Ogólnie lepiej starać się ich unikać ponieważ zazwyczaj można uzyskać w *Mathematice* znacznie lepsze wyniki pod względem szybkości używając funkcyjnych konstrukcji Nest, Fold, FixedPoint i Accumulate. Jest to, oczywiście, zwiazane z naturą *Mathematiki*, nie z programowaniem w ogólności. Nawet w *Mathematice* różnica w szybkości programów napisanych proceduralnie i funkcyjnie znika w sytuacjach w których udaje się użyć tzw. kompilacji.

Zmienne Lokalne

Jeśli w programie przypisujemy zmiennym wartości i zmieniamy je, musimy zwracać szczegułną uwagę aby przypdakowo nie zmienić wartości zmiennych które chcialiśmy zachować. Najprostszym sposobem zabezpieczenia się przed taką ewentualnością jest używanie lokalnych zmiennych. *Mathematica* ma

trzy podstawowe konstrukcje lokalizujące zmienne: Block, Module and With. Każda z nich działa inaczej i ma swoje silne i słabe strony.

```
? Block
```

Block[{x, y, … }, expr] specifies that expr is to be evaluated with local values for the symbols x, y, … .
Block[{x = x0, … }, expr] defines initial local values for x, … . ≫

Najpierw prosty przykład lokalizacji przez Block. Przypisujemy zmiennej x wartość 3. Wewnątrz Block zmieniamy jej wartość na 1. Widzimy że "na zewnątrz" wartość x nie uległa zmianie.

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

```
1
```

```
x
```

```
3
```

Dokładnie tak samo zachowa się Module

```
? Module
```

Module[{x, y, … }, expr] specifies that occurrences of the symbols x, y, … in expr should be treated as local.
Module[{x = x0, … }, expr] defines initial values for x, … . ≫

```
x = 3; Module[{x, y, z = 1}, x = 5; y = x + z]
```

```
6
```

```
x
```

```
3
```

Block and Module działają zupełnie inaczej. Kiedy lokalizujemy zmienną używając Block jej wartość jest najpierw zapisana, potem tymczasowo odebrana wraz z wszystkimi atrybutami zmiennej, i na koniec, po wyjściu z Block oryginalna wartość zostaje przywrócona zmiennej. W przypadku Module nazwy zmiennych są zmienione wewnątrz Module, tak że nie konfliktują z zewnętrznymi. W przeciwieńswie do Module i Block, wszystkie lokalne zmienne w With muszą mieć przypisane wartości początkowe:

```
With[{x = 1, y = 2}, x + y]
```

```
3
```

Następnie, zauważmy jedną cechę która odróżnia Module i With od Block. W tych pierwszych inicializacja (przypisanie wartości początkowych) dla wszystkich zmiennych odbywa się niezależnie:

```
ClearAll[f]
```

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

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

```
1 + u
```

```
ClearAll[g]
```

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

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

```
1 + u
```

Block zachowuje się inaczej:

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

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

```
4
```

A oto inny ważny przykład ilustrujący różnicę między Block i Module:

```
foo := x
```

```
x = 1; foo
```

```
1
```

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

```
2
```

Chociaż foo było zdefiniowane na zewnątrz struktury Block, jego wartość zmieniała się wewnątrz Block. Module zachowuje się inaczej:

```
foo
```

```
1
```

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

```
1
```

A więc, ewaluacja wewnatrz Module zależna jest tylko od oryginalnej definicji zmiennej czy funkcji, co nie jest prawdą w przypadku Blocku.

Pętle i funkcyja iteracji

Programy napisane według imperatywnego paradygmatu zmieniają wartości przypisane jakiejś zmiennej, po czym, aby otrzymać wartość zmiennej należy ją jawnie ewaluować. Rozważmy prosty przykład używający pęteli Do:

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

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

Teraz zrobimy to samo w paradygmie funkcyjnej. Programując w tym stylu używamy funkcji które zwracają swoje wartości zamiast zmieniać wartość jakiejś zmiennej. Zamiast pęteli używamy funkcji których argumentami są funkcje. Przykładem takiej funkcji jest 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]]]]
```

Jest także blisko spokrewniona funkcja 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]]]]}
```

Zamiast używać pętli Do jak powyżej, uzyskujemy ten sam wynik programując funkcyjnie:

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

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

Zauważmy dużą różnicę w szybkości.

```
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]}
```

Pierwszy argument FoldList musi być funkcją dwóch argumentów. Poniższe przykłady ilustrują działanie 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}
```

Zwróćmy uwagę na jeszcze jedną funkcję tego typu:

```
? 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}
```

Oczywiście ten sam wynik można uzyskać przy pomocy bardziej ogólnej funkcji FoldList:

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

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

Jednakże, bardziej wyspecializowana funkcja Accumulate, jest szybsza w działaniu na numerycznych argumentach:

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

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

```
{0.000056, Null}
```

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

```
{0.000853, Null}
```

```
First[b] / First[a]
```

```
15.2321
```

```
Last[a] == Last[b]
```

```
True
```

Większa szybkość wyspecializowanych funkcji w porównaniu z bardziej ogólnymi jest ważnym elementem w programowaniu w *Mathematice*.

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

Block i zmienne globalne.

Block jest najczęściej używany w celu tymczasowej zmiany wartości zmiennych globalnych. Na przykład, globalna zmienna $RecursionLimit ma domyślną wartość:

```
$RecursionLimit
```

```
256
```

Oznacza to że pętla w której długość rekursji przekroczy 256 kroków zostanie autmatycznie zatrzymana. Oczywiście ma to ułatwić wyłapywanie błędów programistycznych które bez tego ograniczenia prowadziłyby do nieskończonych pętli. Czasem jednak to ogranicznie jest nie wygodne. Dla przykładu, wróćmy jeszcze raz do dobrze znanej nam definicji:

```
Clear[Fib]
```

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

Spróbyjmy

```
Fib[3000]
```

$RecursionLimit::reclim : Recursion depth of 256 exceeded. ≫

$RecursionLimit::reclim : Recursion depth of 256 exceeded. ≫

```
54 122 222 371 037 658 776 676 579 571 233 761 483 351 206 693 809 497
  Hold[Fib[2745 - 1] + Fib[2745 - 2]] +
 87 571 595 343 018 854 458 033 386 304 178 158 174 356 588 264 390 370
  Hold[Fib[2746 - 1] + Fib[2746 - 2]]
```

Ograniczenie zmiennej $RecursionLimit do 256 powoduje że nasz kod nie chce działać. Mogłibyśmy zmienić $RecursionLimit do wiekszej liczby albo całkowicie go usunąć robiąc go równym ∞, ale robiąc to narażamy się na nieprzyjemne konsekwencje w przyszłości. Dużo lepiej jest zmienić tymczasowo wartość $RecursionLimit za pomocą Block. Zanim jednak to zrobimy musimy usunąć wszystkie definicje Fib bo *Mathematica* zapamiętała "zatrzymane" wartości.

```
Clear[Fib]
```

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

```
Block[{$RecursionLimit = ∞}, 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
```

Zauważmy że globalna wartość $RecursionLimit pozostaje nie zmieniona:

```
$RecursionLimit
```

```
256
```

Przykład: Symulacja Ruchu Browna

Jako przykład użycia funkcji FoldList, NestList i Accumulate podajemy konstrukcję symulacji ruchu Browna (procesu Wienera). W Mathematice 9 konstrukcja staje się dużo łatwiejsza dzięki nowym wbudowanym funkcjom RandomFunction i WienerProcess. Kod który podajemy poniżej działa w wersjach wyższych niż 6.

Jedna ścieżka

Ścieżkę ruchu Browna (przebytą w czasie 1) przybliżamy przez losowe błądzenie z n krokami, gdzie każdy krok jest liczbą rzeczywistą o rozkładzie normalnym z średnią 0 i standardowym odchyleniem $\sqrt{\frac{1}{n}}$. Jest szereg sposobów zaprogramowanie tego w *Mathematice.* Podajemy trzy

1. Accumulate (najszybsza metoda)

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

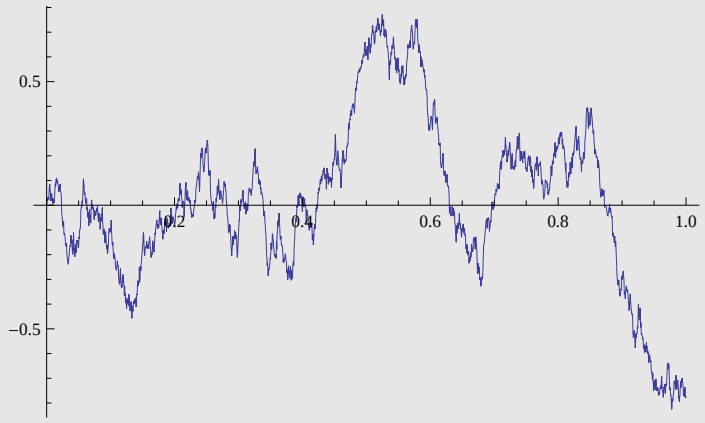
2. FoldList (powolniejsza)

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

3. NestList (jeszcze powolniejsza)

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

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

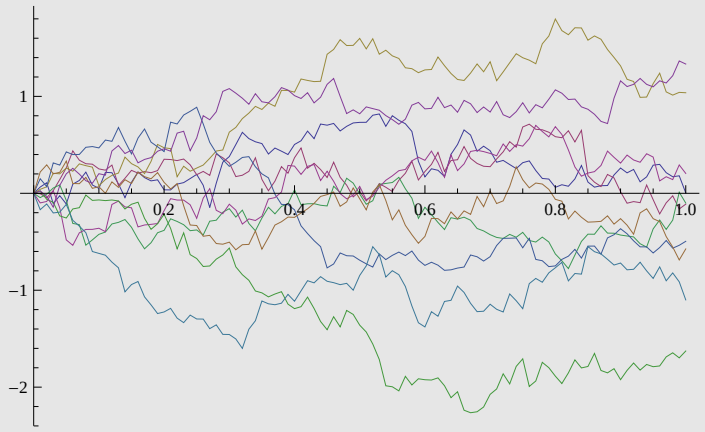


Wiele ścieżek

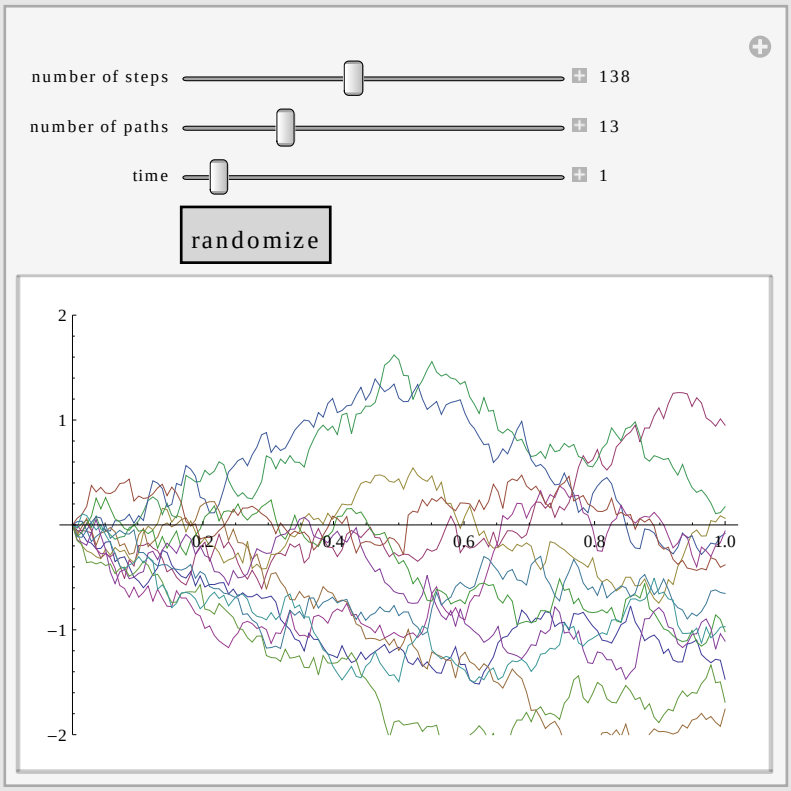
```
Clear[BrownianMotion]
```

```
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[BlockRandom[SeedRandom[r];
    ListLinePlot[BrownianMotion[time, steps, paths], DataRange → {0, time}, PlotRange → {-2, 2}]],
  {{steps, 100, "number of steps"}, 10, 300, 1, Appearance → "Labeled"},
  {{paths, 10, "number of paths"}, 1, 50, 1, Appearance → "Labeled"},
  {{time, 1, "time"}, 0.5, 10, Appearance → "Labeled"},
  {{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