

# Uncertain Virtual Worlds

B. Ycart

LMC/IMAG, BP 53, 38041 Grenoble Cedex 09  
bernard.ycart@imag.fr

*Mathematica* packages for versions 2.0 and higher

Université Joseph Fourier

August 1997

# Contents

<b>0</b>	<b>Introduction</b>	<b>1</b>
<b>1</b>	<b>Billiard.m (Billiards with a round obstacle)</b>	<b>3</b>
1.1	Package . . . . .	3
1.2	Examples . . . . .	10
<b>2</b>	<b>ContSamp.m (Simulations of continuous distributions)</b>	<b>11</b>
2.1	Package . . . . .	11
2.2	Examples . . . . .	13
<b>3</b>	<b>DataRep.m (Data representations)</b>	<b>15</b>
3.1	Package . . . . .	15
3.2	Examples . . . . .	20
<b>4</b>	<b>DiscSamp.m (Simulations of discrete distributions)</b>	<b>21</b>
4.1	Package . . . . .	21
4.2	Examples . . . . .	23
<b>5</b>	<b>DynSyst.m (Dynamical Systems)</b>	<b>24</b>
5.1	Package . . . . .	24
5.2	Examples . . . . .	27
<b>6</b>	<b>Fractals.m (Deterministic and random Von Koch curves)</b>	<b>28</b>
6.1	Package . . . . .	28
6.2	Examples . . . . .	33
<b>7</b>	<b>Interact.m (Simulation of spin systems in the plane)</b>	<b>35</b>
7.1	Package . . . . .	35
7.2	Examples . . . . .	39
<b>8</b>	<b>Lorentz.m (Lorentz's attractor)</b>	<b>40</b>
8.1	Package . . . . .	40
8.2	Examples . . . . .	42
<b>9</b>	<b>PseuGene.m (Congruential and midsquare generators)</b>	<b>42</b>
9.1	Package . . . . .	42
9.2	Examples . . . . .	45
<b>10</b>	<b>RandWalk.m (Random walks and random vector fields)</b>	<b>45</b>
10.1	Package . . . . .	45
10.2	Examples . . . . .	48
<b>11</b>	<b>Stogho.m (Stochastic Ghost)</b>	<b>48</b>
11.1	Package . . . . .	48
11.2	Examples . . . . .	50

<b>12 TimeRep.m (Queues and time processes)</b>	<b>50</b>
12.1 Package . . . . .	50
12.2 Examples . . . . .	52
<b>13 ZeroOne.m (Lists of zeros and ones)</b>	<b>53</b>
13.1 Package . . . . .	53
13.2 Examples . . . . .	56

## 0 Introduction

The word “packages” is not totally appropriate here. The functions are left without input protections. They are not declared as packages in the language, and their denominations are not protected. This may cause some errors. However it should permit easier ‘on line’ testing and modifications. The 13 files should be copied in a subdirectory UVW of the directory MATH\PACKAGES. Then they will be accessible by a command of the type <<uvw\package.m.

The programming style is a compromise between two objectives. The first one is to respect the spirit and style of *Mathematica*, in order to use the language as efficiently as possible. The second one is to make the functions transparent and easy to modify for the user. As an illustration of that compromise, one can compare for instance the function `Distribution` in package UVW‘DiscSamp’, with the similar `Frequencies` of the standard package Statistics‘DataManipulation’.

The packages contain the following functions. All functions of the type

`RS... [...,n]`

return in a list a Random Sample of size `n` , i.e. a realization of an `n`-tuple of independent identically distributed random variables.

1. `Billiard.m` (Billiards with a round obstacle)
  - `NextBounce[currentposition, r]`
  - `Trajectory[shootingangle, tmax, r]`
  - `Bundle[listofangles, tmax, r]`
  - `Differences[alpha, deltaalpha, tmax, r]`
2. `ContSamp.m` (Simulations of continuous distributions)
  - `RSContinuousDistribution[density, a, b, n]`
  - `RSIndependent2D[density1, a1, b1, density2, a2 ,b2, n]`
  - `RSNormal2D[sigma1, sigma2, rho, n]`
  - `RSUnitBall[dim, n]`
3. `DataRep.m` (Data representations)
  - `Histogram[listofdata, listofbounds]`
  - `RegularHisto[listofdata, xmin, xmax, nx]`
  - `SamplePlot2D[listof2Ddata]`
  - `Histogram2D[listof2Ddata, xmin, xmax, nx, ymin, ymax, ny]`
  - `LargeNumbers[listofdata]`
  - `CentralLimit[listofdata, mu, sigma, n]`
4. `DiscSamp.m` (Simulations of discrete distributions)
  - `Distribution[listofdata]`
  - `RSPermutation[list, n]`
  - `RSExtract[list, k, n]`
  - `RSDiscreteDistribution[dist, n]`

5. DynSyst.m (Dynamical Systems)
- Mean[listofdata]
  - Variance[listofdata]
  - CovarianceFunction[listofdata, n]
  - AsymptoticVariance[listofdata, n]
  - CorrelationDimension[listofdata, step, nstep]
  - TentFunction[a, x]
  - LogisticFunction[a, x]
  - IterateAPhi[matrixA, functionPhi, vectorX0, n]
6. Fractals.m (Deterministic and random Von Koch curves)
- Triangle
  - Star
  - Island
  - Battlement
  - Hat[sharpness]
  - Wy[angle1, length1, angle2, length2]
  - RShWy[n]
  - RSTruncs[n]
  - TransformSegment[segment, pattern]
  - IteratePattern[listofsegments, pattern, n]
  - IterateRandomPattern[listofsegments, patterns, probas, n]
  - DrawSegments[listofsegments]
7. Interact.m (Simulation of spin systems in the plane)
- Checkerboard
  - Diagonals
  - RConfig[p, width, height]
  - Uniform[lambda, mu]
  - Ising[Alpha, Beta]
  - Contact[lambda]
  - Voter
  - Cyclic[n, bound]
  - RepartConfig[config]
  - Evolution[initialconfig, rates, niter]
  - DrawConfig[config, opts]
8. Lorentz.m (Lorentz's attractor)
- Lorentz[s, b, r]
  - LorentzArray[matrix]
9. PseuGene.m (Congruential and midsquare generators)
- CongruGenerator[seed, a, c, m, n]
  - MidsquareGenerator[seed, n]
  - CongruentialLoop[seed, a, c, m]
  - MidsquareLoop[seed]

10. RandWalk.m (Random walks and random vector fields)  
 RandomWalk[listof2Dvelocities, deltat]  
 VectorField[arrayof2Dvelocities]  
 VectorFieldTrajectory[arrayof2Dvelocities, deltat, tmax]
11. Stogho.m (Stochastic Ghost)  
 Stogho[c]  
 GalleryOfPortraits[matrix]
12. TimeRep.m (Queues and time processes)  
 Queue[interarrivals, services]  
 CumulatedTimes[listoftimes]  
 Geiger[listoftimes]
13. ZeroOne.m (Lists of zeros and ones)  
 RSZeroOne[p, n]  
 Binary[functionf, listofzeroones]  
 PlotZeroOne[listofzeroones]  
 AnimateShift[listofzeroones]  
 ActualLength[listofzeroones]  
 Weight[listofzeroones]  
 WeightedAlphabeticalOrder[listofzeroones]  
 Entropy[t]

## 1 Billiard.m (Billiards with a round obstacle)

### 1.1 Package

```
BeginPackage["UVW`Billiard`"]
```

```
NextBounce::usage = "
```

```
NextBounce[currentposition,r] computes the next bouncing point of a ball
inside a square table with a circular obstacle of radius r. The initial
conditions are given in the list currentposition that has four real
coordinates, respectively the abscissa, ordinate, incoming angle of the ball,
and the current time. The result is returned as another list of four
elements, the abscissa and ordinate of the new bouncing point, the new
direction of the ball and the current time incremented by the running time
between the two bounces."
```

```
Trajectory::usage ="
```

```
Trajectory[shootingangle,tmax,r] draws a square table with a centered
circular obstacle of radius r. Draws inside this support the
trajectory of a ball starting at the bottom left corner with initial
direction shootingangle, up to time tmax. "
```

```

Bundle::usage = "
Bundle[listofangles,tmax,r] draws a square table with a centered circular
obstacle of radius r. Draws inside the trajectories of balls starting at the
bottom left corner with initial directions read in listofangles, up to
time tmax. "

```

```

Differences::usage = "
Differences[angle,dangle,tmax,r] simulates two trajectories of balls in a
square table with a centered circular obstacle with radius r. Both
trajectories start from the bottom left corner. They are followed up to time
tmax. The shooting angle of the first trajectory is angle, its difference
with the second shooting angle is dangle. The function represents three
consecutive graphics. The first one is the billiard table with the two
trajectories . The second one is the evolution of the absolute difference
of angles as a function of time. The third one is the norm of the difference
of positions as a function of time. "

```

```

Begin ["Private"]

```

```

NextBounce[currentposition_List,r_]:=
Block[
  {xold,yold,aold,cosaold,sinaold,epsi=10^(-6),
  tnorth,teast,tsouth,twest,tcircle1,tcircle2,delta,beta,
  trun,xnew,ynew,anew},

  (* Old coordinates -----*)
  xold = currentposition[[1]];
  yold = currentposition[[2]];
  aold = currentposition[[3]];
  cosaold = Cos[aold];
  sinaold = Sin[aold];

  (* Hitting time of the four edges -----*)
  tnorth = (+1-yold)/sinaold; If[tnorth<epsi,tnorth=Infinity];
  teast = (+1-xold)/cosaold; If[teast <epsi,teast =Infinity];
  tsouth = (-1-yold)/sinaold; If[tsouth<epsi,tsouth=Infinity];
  twest = (-1-xold)/cosaold; If[twest <epsi,twest =Infinity];

  (* Hitting time of the circular obstacle -----*)
  delta = (xold*cosaold+yold*sinaold)^2-(xold^2+yold^2-r^2);
  If[delta>0,
  (
  delta = Sqrt[delta];

```

```

tcircle1 = -(xold*cosaold+yold*sinaold) + delta;
If[tcircle1<epsi,tcircle1=Infinity];
tcircle2 = -(xold*cosaold+yold*sinaold) - delta;
If[tcircle2<epsi,tcircle2=Infinity];
),
(
tcircle1 = Infinity;
tcircle2 = Infinity;
)
];

(* Next bounce is at minimal hitting time -----*)
trun = Min[{tnorth,teast,tsouth,twest,tcircle1,tcircle2}];

Switch[Position[{tnorth,teast,tsouth,twest,tcircle1,tcircle2},
trun][[1,1]],

1,          (* Next bounce on north edge -----*)
(
xnew = xold + trun*cosaold;
ynew = 1.;
anew = N[2*Pi]-aold;
Break[]
),
2,          (* Next bounce on east edge -----*)
(
ynew = yold + trun*sinaold;
xnew = 1.;
anew = N[Pi]-aold; If[anew<0,anew = anew+N[2*Pi]];
Break[]
),
3,          (* Next bounce on south edge -----*)
(
xnew = xold + trun*cosaold;
ynew = -1.;
anew = N[2*Pi]-aold;
Break[]
),
4,          (* Next bounce on west edge -----*)
(
ynew = yold + trun*sinaold;
xnew = -1.;
anew = N[Pi]-aold; If[anew<0,anew = anew+N[2*Pi]];
Break[]
),

```

```

5,          (* Next bounce on circle -----*)
(
  xnew = xold + trun*cosaold;
  ynew = yold + trun*sinaold;
  anew = Mod[2*ArcTan[xnew,ynew]-aold,N[2*Pi]];
  Break[]
),
6,          (* Next bounce on circle -----*)
(
  xnew = xold + trun*cosaold;
  ynew = yold + trun*sinaold;
  anew = Mod[2*ArcTan[xnew,ynew]-aold-N[Pi],N[2*Pi]];
  Break[]
)];

(* Protection against leaky corners -----*)
If[(Abs[xnew*ynew]>1-epsi),
If[(aold<N[Pi/2]),anew=N[3*Pi/2]-aold,
  If[(aold<N[Pi]),anew=N[5*Pi/2]-aold,
If[(aold<N[3*Pi/2]),anew=N[3*Pi/2]-aold,
  anew=N[3*Pi/2]-aold]]]];

  Return[{xnew,ynew,anew,currentposition[[4]]+trun}];

]

```

```

Trajectory[shootingangle_,tmax_,r_]:=
Block[
  {running=0.,traj={{-1.,-1.}},next},

  next = {-1.,-1.,N[shootingangle],0.};

  While [running<N[tmax],
  (
  next = NextBounce [next,r];
  traj = Append[traj,Take[next,2]];
  running = Last[next];
  )
];

  Show[
Graphics[{
Thickness[0.01],

```

```

Line[{{-1,-1},{-1,1},{1,1},{1,-1},{-1,-1}}],
Circle[{0,0},r],
Thickness[0.005],
Line[traj]
}],
AspectRatio->1
]
]

Bundle[listofangles_List,tmax_,r_]:=
Block[
  {nbtraj,running=0.,traj,next,i,g1,g2},

  nbtraj = Length[listofangles];
  traj=Table[{{-1.,-1.}},{nbtraj}];
  Do[(
next = {-1.,-1.,N[listofangles[[i]]],0.};
running = 0.;
While[running<N[tmax],
(
next = NextBounce[next,r];
traj[[i]] = Append[traj[[i]],Take[next,2]];
running = Last[next];
)
];
),{i,1,nbtraj}];

  g1 = Graphics[{
Thickness[0.01],
Line[{{-1,-1},{-1,1},{1,1},{1,-1},{-1,-1}}],
Circle[{0,0},r]
}];
  g2 = Table[Graphics
[{{
Thickness[0.005],
Line[traj[[i]]]
}},{i,1,nbtraj}];

  Show[Prepend[g2,g1], AspectRatio->1];

];

```

```

Differences[angle_,dangle_,tmax_,r_]:=
Block[
  {listofangles,running=0.,traj,next,i,g1,g2,
  angles,diffangles,aold,diffpos,i1,i2},

  listofangles = N[{angle,angle+dangle}];
  traj=Table[{{-1.,-1.,listofangles[[i]],0.}},{i,1,2}];

  Do[(
  next = {-1.,-1.,listofangles[[i]],0.};
  running = 0.;
  While[running<N[tmax],
  (
  next = NextBounce[next,r];
  traj[[i]] = Append[traj[[i]],next];
  running = Last[next];
  )
  ],
  ),{i,1,2}];

  (* Computation of differences -----*)
  angles = Table[Transpose[Drop[Transpose[traj[[i]],2]],{i,1,2}];
  aold = Abs[dangle];
  diffangles = {{0.,aold}};
  diffpos = {{0.,0.}};
  i1 = 2; i2 = 2;

  While[(i1<=Length[angles[[1]])&&(i2<=Length[angles[[2]]]),
  (
  If[angles[[1,i1,2]]<angles[[2,i2,2]],
  (
  diffangles = Append[diffangles,
  {angles[[1,i1,2]],aold}];
  aold = Abs[angles[[1,i1,1]]-angles[[2,i2-1,1]]];
  diffangles = Append[diffangles,
  {angles[[1,i1,2]],aold}];
  diffpos = Append[diffpos,
  {angles[[1,i1,2]],
  Sqrt[(traj[[1,i1,1]]-traj[[2,i2-1,1]]-
  (traj[[2,i2,1]]-traj[[2,i2-1,1]])*
  (angles[[1,i1,2]]-angles[[2,i2-1,2]])/
  (angles[[2,i2,2]]-angles[[2,i2-1,2]])
  )^2 +
  (traj[[1,i1,2]]-traj[[2,i2-1,2]]-
  (traj[[2,i2,2]]-traj[[2,i2-1,2]])*

```

```

        (angles[[1,i1,2]]-angles[[2,i2-1,2]])/
        (angles[[2,i2,2]]-angles[[2,i2-1,2]])
        )^2}}];
    i1 = i1+1;
),
(
    diffangles = Append[diffangles,
{angles[[2,i2,2]],aold}];
    aold = Abs[angles[[1,i1-1,1]]-angles[[2,i2,1]]];
    diffangles = Append[diffangles,
{angles[[2,i2,2]],aold}];
    diffpos = Append[diffpos,
{angles[[2,i2,2]],
Sqrt[(traj[[2,i2,1]]-traj[[1,i1-1,1]]-
(traj[[1,i1,1]]-traj[[1,i1-1,1]])*
(angles[[2,i2,2]]-angles[[1,i1-1,2]])/
(angles[[1,i1,2]]-angles[[1,i1-1,2]])
)^2 +
(traj[[2,i2,2]]-traj[[1,i1-1,2]]-
(traj[[1,i1,2]]-traj[[1,i1-1,2]])*
(angles[[2,i2,2]]-angles[[1,i1-1,2]])/
(angles[[1,i1,2]]-angles[[1,i1-1,2]])
)^2}]];
    i2 = i2+1;
)
];
)];

```

(\* Representation of trajectories -----\*)

```

g1 = Graphics[{
Thickness[0.01],
Line[{{-1,-1},{-1,1},{1,1},{1,-1},{-1,-1}}],
Circle[{0,0},r]
}];
g2 = Table[Graphics
[ {
Thickness[0.005],
Line[Transpose[Take[Transpose[traj[[i]]],2]]]
}],{i,1,2}];

```

```
Show[Prepend[g2,g1], AspectRatio->1];
```

(\* Representation of angle differences -----\*)

```

        Show[Graphics[{
Thickness[0.005],
Line[diffangles]}],
Axes->True,
Frame->True,
AxesLabel->{"Time","Difference of angles"}
];

(* Representation of position differences -----*)
        Show[Graphics[{
Thickness[0.005],
Line[diffpos]}],
Axes->True,
Frame->True,
AxesLabel->{"Time","Difference of positions"}
]
];

End[]

EndPackage[]

```

## 1.2 Examples

Here are some trajectories with a growing obstacle.

```

In[1]:= <<uvw\billiard.m
In[2]:= Trajectory[0.4,100,0.]
In[3]:= Trajectory[0.4,100,0.1]
In[4]:= Trajectory[0.4,100,0.5]
In[5]:= Trajectory[0.4,100,0.8]
In[6]:= Bundle[Range[Pi/12,5*Pi/12,Pi/12],10,0.]
In[7]:= Bundle[Range[Pi/12,5*Pi/12,Pi/12],50,0.]
In[8]:= Bundle[Range[Pi/12,5*Pi/12,Pi/12],50,0.1]
In[9]:= Bundle[Range[Pi/12,5*Pi/12,Pi/12],50,0.5]
In[10]:= Bundle[Range[Pi/12,5*Pi/12,Pi/12],50,0.8]

```

Here is how two trajectories, with close shooting angles, start differing chaotically after some bounces on the circular obstacle.

```

In[1]:= <<uvw\billiard.m
In[2]:= Differences[0.4,0.001,20,0.]
In[3]:= Differences[0.4,0.001,100,0.]
In[4]:= Differences[0.4,0.001,20,0.1]

```

```

In[5]:= Differences[0.4,0.001,20,0.5]
In[6]:= Differences[0.4,0.001,100,0.5]
In[7]:= Differences[0.4,0.001,200,0.5]

```

## 2 ContSamp.m (Simulations of continuous distributions)

### 2.1 Package

```
BeginPackage["UVW`ContSamp`"]
```

```

RSContinuousDistribution::usage = "
RSContinuousDistribution[functionf,a,b,n] returns a sample of size n
for the distribution with density functionf on the interval [a,b]."

```

```

RSIndependent2D::usage = "
RSIndependent2D[functionf1,a1,b1,functionf2,a2,b2,n] returns a sample of
size n {{x1,y1},...,{xn,yn}} of a two-dimensional random vector (X,Y) ,
where X and Y are independent, X has density functionf1 on the interval
[a1,b1], Y has density functionf2 on the interval [a2,b2]."

```

```

RSNormal2D::usage = "
RSNormal2D[Sigma1,Sigma2,rho,n] returns a sample of size n for the
two-dimensional Gaussian vector (X,Y). The means of X and Y are null,
their standard deviations are sigma1 and sigma2. Their correlation
coefficient is rho."

```

```

RSUnitBall::usage = "
RSUnitBall[dim,nb] returns a sample of size n of vectors uniformly
distributed in the unit ball of the space of dimension dim."

```

```
Begin["`Private`"]
```

```
RSContinuousDistribution[density_ , a_ , b_ , n_Integer]:=
```

```
Block[ {nbint,delta,abscissas,ordinates,cdf,inv},
```

```

nbint=100;
delta=N[(b-a)/nbint];

```

```

nbint=nbint+1;
abscissas=N[Range[a,b,delta]];
ordinates=Table[ density[abscissas[[i]]] ,{i,nbint}];

cdf = Drop[ordinates,-1]+Drop[ordinates,1];
delta = delta/2;
cdf = cdf*delta;
cdf = FoldList[Plus,0.,cdf];
cdf = cdf/Last[cdf];

cdf = Transpose[{cdf,abscissas}];
inv = Interpolation[cdf, InterpolationOrder->1];

Return[Table[ inv[Random[] ] , {n} ] ];

];

RSIndependent2D[density1_,inf1_,sup1_,density2_,inf2_,sup2_,n_Integer]:=
If[N[inf1]<=N[sup1] && N[inf2]<=N[sup2],
  Return[
  Transpose [
    {RSContinuousDistribution[density1,N[inf1],N[sup1],n],
    RSContinuousDistribution[density2,N[inf2],N[sup2],n]}
  ]
];

RSNormal2D[sigma1_,sigma2_,rho_,n_Integer]:=
Block[{matra={{sigma1,sigma2*rho},{0,sigma2*Sqrt[1-rho*rho]}},res={},
  u={},s},
  Do[(
  u = {Random[Real,{-1,1}],Random[Real,{-1,1}]};
  s = u . u;
  While[s>1,
    u = {Random[Real,{-1,1}],Random[Real,{-1,1}]};
    s = u . u;
  ];
  res = Append[res,u*Sqrt[-2Log[s]/s]]
  ),{n}];
  res=res.matra;

```

```

        Return[res];
    ];

RSUnitBall[dim_Integer,n_Integer]:=

Block[{res={},coord={}},
  Do[(
    coord=Table[Random[Real,{-1,1}],{dim}];
    While[coord.coord>1,
    coord=Table[Random[Real,{-1,1}],{dim}]
    ];
    res=Append[res,coord]
  ),{n}];
  Return[res];
];

End[]

EndPackage[]

```

## 2.2 Examples

Random samples of standard continuous distributions can be simulated using the standard function `Random`, together with the distributions defined in the package `Statistics`ContinuousDistributions``.

```

In[1]:= Table[Random[],{100}]
In[2]:= Table[Random[Real,{2,4}],{100}]
In[3]:= <<Statistics`ContinuousDistributions`
In[4]:= Table[Random[ExponentialDistribution[1.]],{100}]
In[5]:= Table[Random[NormalDistribution[0.,1.]],{100}]
In[6]:= Table[Random[WeibullDistribution[2.,1.]],{100}]

```

A random sample of a distribution with an arbitrary density can be simulated using `RSContinuousDistribution`. Notice that the function `f` only needs to be non negative over the prescribed interval. `RSContinuousDistribution` divides it automatically by its integral.

```

In[1]:= <<uvw\contsamp.m
In[2]:= <<uvw\dataarep.m
In[3]:= f[x_]:=1+Sin[x]
In[4]:= samp=RSContinuousDistribution[f,0,10,2000];

```

```

In[5] := g1=RegularHisto[samp,0,10,20,DisplayFunction->Identity]
In[6] := integral=NIntegrate[f[x],{x,0,10}]
In[7] := fnorm[x_]:=f[x]/integral
In[8] := g2=Plot[fnorm[x],{x,0,10},DisplayFunction->Identity]
In[9] := Show[g1,g2,DisplayFunction->${DisplayFunction}]

```

The function `RSIndependent2D` returns a two-dimensional sample with independent coordinates. Each coordinate is simulated using `RSContinuousDistribution`.

```

In[1] := <<uvw\contsamp.m
In[2] := <<uvw\dataarep.m
In[3] := f[x_]:=x^2
In[4] := para2=RSIndependent2D[f,-1,1,f,-1,1,2000];
In[5] := SamplePlot2D[para2,Frame->True,AspectRatio->1]
In[6] := Histogram2D[para2,-1,1,8,-1,1,8]
In[7] := marge1=Transpose[para2][[1]];
In[8] := RegularHisto[marge1,-1,1,10]

```

The package `Statistics`MultinormalDistribution``, delivered with version 3.0 of *Mathematica*, permits all sorts of manipulations with Gaussian vectors, including their simulation by the standard `Random` function. `RSNormal2D` returns samples of Gaussian vectors in the plane.

```

In[1] := <<uvw\contsamp.m
In[2] := <<uvw\dataarep.m
In[3] := gauss2=RSNormal2D[2,1,-0.8,2000];
In[4] := SamplePlot2D[gauss2]
In[5] := Histogram2D[gauss2,-6,6,10,-3,3,10]
In[6] := combi=gauss2.Transpose[{0.5,1.6}];
In[7] := RegularHisto[combi,-3,3,20]
In[8] := CentralLimit[combi,0,1,1]

```

Here is an illustration of the uniform distribution on the unit ball in dimensions 2, 3 and 4. The norm of a random point in the unit ball in dimension  $n$  tends to 1 as  $n$  tends to infinity.

```

In[1] := <<uvw\contsamp.m
In[2] := <<uvw\dataarep.m
In[3] := ball2=RSUnitBall[2,1000];
In[4] := SamplePlot2D[ball2,AspectRatio->1]
In[5] := norms=Sqrt[Table[ball2[[i]].ball2[[i]},{i,1000}]];
In[6] := RegularHisto[norms,0,1,10]
In[7] := ball3=RSUnitBall[3,1000];
In[8] := Show[Graphics3D[Table[Point[ball3[[i]]],{i,1000}]];
In[9] := norms=Sqrt[Table[ball3[[i]].ball3[[i]},{i,1000}]];
In[10] := RegularHisto[norms,0,1,10]

```

```

In[11]:= ball4=RSUnitBall[4,1000];
In[12]:= norms=Sqrt[Table[ball4[[i]].ball4[[i]],{i,1000}]];
In[13]:= RegularHisto[norms,0,1,10]

```

### 3 DataRep.m (Data representations)

#### 3.1 Package

```
BeginPackage["UVW`DataRep`"]
```

```
Histogram::usage = "
```

```
Histogram[listofdata,listofbounds] represents the histogram of the data
contained in listofdata. The bounds of the classes are read in
listofbounds."
```

```
RegularHisto::usage = "
```

```
RegularHisto[listofdata,xmin,xmax,nx] represents the histogram of the data
contained in list of data. There are nx regular classes between xmin
and xmax."
```

```
SamplePlot2D::usage = "
```

```
SamplePlot2D[listof2Ddata] plots in the plane the points whose coordinates
are read in listof2Ddata."
```

```
Histogram2D::usage = "
```

```
Histogram2D[listof2Ddata,xmin,xmax,nx,ymin,ymax,ny] represents a histogram
in 3 dimensions for the two dimensional data contained in listof2Ddata.
The classes are regular. There are nx classes in abscissa between xmin
and xmax, and ny classes in ordinate between ymin and ymax."
```

```
LargeNumbers::usage = "
```

```
LargeNumbers[listofdata] plots the partial means of the data contained in
listofdata."
```

```
CentralLimit::usage = "
```

```
CentralLimit[listofdata, mu, sigma, n] takes consecutive groups of n data in
listofdata. The sum of each group is centered by n*mu then divided
by Sqrt[n]*sigma. The results are represented on a regular histogram
with 20 classes."
```

```

Begin["Private"]

Histogram[listofdata_List,listofbounds_List,opts___]:=

  Block[{list,nbclasses,cardinal,i,ordinates,abscissas,g},

    list = Cases[listofdata,
x_/;First[listofbounds]<x<Last[listofbounds]];
    nbclasses = Length[listofbounds]-1;

    cardinal = Table[Length[Select[list,
listofbounds[[i]]<#<=listofbounds[[i+1]] &]],
{i,1,nbclasses}];

    cardinal = cardinal / (Drop[listofbounds,1]
- Drop[listofbounds,-1]);
    cardinal = cardinal / Length[list];

    ordinates = Flatten[Transpose[{cardinal,cardinal,
Table[0,{nbclasses}]}]];

    abscissas = Flatten[Transpose[{Drop[listofbounds,-1],
Drop[listofbounds,1],
Drop[listofbounds,1]}]];

    g = Transpose[{abscissas,ordinates}];
    g = Prepend[g,{First[listofbounds],0}];

    ListPlot[g,
opts,
PlotRange -> All,
PlotJoined -> True]
  ]

RegularHisto[listofdata_List,xmin_,xmax_,nx_,opts___]:=

  Block[{cardinal,list,step,i,abscissas,ordinates,g},

```

```

step = N[(xmax-xmin)/nx];

list = Cases[listofdata,x_/;xmin<x<xmax];
list = list - xmin;
list = list/step;
list = Ceiling[list];

cardinal=Table[Count[list,i],{i,1,nx}];
cardinal=cardinal / (step*Length[list]);

ordinates = Flatten[Transpose[{cardinal,cardinal,Table[0,{nx}]}]];
abscissas = Flatten[Transpose[{Range[xmin,xmax-step,step],
    Range[xmin+step,xmax,step],
    Range[xmin+step,xmax,step]}]];

g = Transpose[{abscissas,ordinates}];
g = Prepend[g,{xmin,0}];

ListPlot[g,
  opts,
  PlotJoined -> True,
  PlotRange -> All ]

]

Histogram2D[list2Dofdata_List,xmin_,xmax_,nx_,ymin_,ymax_,ny_,opts___]:=

Block[{cardinal,xmesh,ymesh,list,stepx,stepy,g,i,j},

  stepx = N[(xmax-xmin)/nx];
  stepy = N[(ymax-ymin)/ny];

  list = Cases[list2Dofdata,x_List/;
  xmin<First[x]<xmax && ymin<Last[x]<ymax];
  list = Transpose[ {(Transpose[list][[1]] - xmin)/stepx ,
(Transpose[list][[2]] - ymin)/stepy } ];
  list = Ceiling[list];

  cardinal = Table[Count[list,{i,j}],{i,1,nx},{j,1,ny}];
  cardinal = cardinal (Max[N[xmax-xmin],N[ymax-ymin]]/Max[cardinal]);
  xmesh = Range[xmin,xmax,stepx];

```

```

ymesh      = Range[ymin,ymax,stepy];

g=Table[{Polygon[{
{xmesh[[i]]      , ymesh[[j]]      ,0},
{xmesh[[i]]      , ymesh[[j]]      ,cardinal[[i,j]]},
{xmesh[[i+1]]    , ymesh[[j]]      ,cardinal[[i,j]]},
{xmesh[[i+1]]    , ymesh[[j]]      ,0}},
Polygon[{
{xmesh[[i]]      , ymesh[[j]]      ,cardinal[[i,j]]},
{xmesh[[i+1]]    , ymesh[[j]]      ,cardinal[[i,j]]},
{xmesh[[i+1]]    , ymesh[[j+1]]    ,cardinal[[i,j]]},
{xmesh[[i]]      , ymesh[[j+1]]    ,cardinal[[i,j]]}},
Polygon[{
{xmesh[[i+1]]    , ymesh[[j]]      ,0},
{xmesh[[i+1]]    , ymesh[[j]]      ,cardinal[[i,j]]},
{xmesh[[i+1]]    , ymesh[[j+1]]    ,cardinal[[i,j]]},
{xmesh[[i+1]]    , ymesh[[j+1]]    ,0}}]
},{i,1,nx},{j,1,ny}];

Show[Graphics3D[g],
     opts,
     PlotRange -> All,
     AspectRatio -> Automatic,
     Boxed -> True,
     Axes -> Automatic,
     Ticks ->Automatic,
     AxesLabel ->{"x","y","f[x,y]"}
     ]
]

```

```

SamplePlot2D[listof2Ddata_,opts___]:=

```

```

ListPlot[listof2Ddata,
  opts,
  PlotRange -> All,
  AxesLabel -> {"x","y"}];

```

```

LargeNumbers[listofdata_List,opts___] :=

```

```

Block[
  {listofmeans,len},

```

```

listofmeans = Rest[FoldList[Plus,0.,listofdata]];
len=Length[listofdata];
listofmeans = listofmeans / Range[len];

ListPlot[listofmeans ,

PlotRange -> All,
Prolog    -> PointSize[0.001],
opts
]
]

CentralLimit[ listofdata_List,mu_,sigma_,n_,opts___] :=
Block[
{listpart, max,min,g1,g2},

listpart = Transpose[Partition[listofdata,n]];
listpart = Apply[Plus,listpart];
listpart = listpart - (N[n] mu);
listpart = listpart/(Sqrt[N[n]] sigma);

min = Min[listpart];
max = Max[listpart];

g1 := RegularHisto[listpart,min,max,20,
DisplayFunction -> Identity];

g2 := Plot[Exp[-(x^2)/2] / Sqrt[2 Pi], {x,min,max},
DisplayFunction -> Identity];

Show[g1,g2,DisplayFunction->${DisplayFunction}]

]

End[]

EndPackage []

```

## 3.2 Examples

Here is an illustration of the Law of Large Numbers and the Central Limit Theorem applied to the uniform distribution on  $[0, 1]$ , and to the exponential distribution. The Central Limit Theorem states that the centered and reduced variables associated to the sum of  $n$  independent random variables is approximately normally distributed, for  $n$  large enough. For the uniform distribution, it is true with a reasonable precision, for  $n$  as low as 6. The exponential distribution, being more skewed, requires a much higher value of  $n$ .

```
In[1]:= <<uvw\datarep.m
In[2]:= uni=Table[Random[],{2000}];
In[3]:= Histogram[uni,{0.,0.1,0.3,0.6,0.8,1.}]
In[4]:= RegularHisto[uni,0,1,10]
In[5]:= LargeNumbers[uni]
In[6]:= CentralLimit[uni,0.5,Sqrt[1./12],6]
In[7]:= exp=Table[-Log[Random[]],{2000}];
In[8]:= RegularHisto[exp,0,5,10]
In[9]:= LargeNumbers[exp]
In[10]:= CentralLimit[exp,1.,1.,10]
```

The Law of Large Numbers is false if the distribution of the random variables in the independent sequence does not have an expectation. Here is what happens with the Cauchy distribution.

```
In[1]:= <<uvw\datarep.m
In[2]:= <<Statistics'ContinuousDistributions'
In[3]:= samp=Table[Random[CauchyDistribution[0.,1.]],{2000}];
In[4]:= LargeNumbers[samp]
In[5]:= CentralLimit[samp,0,1,10]
```

Here are several random samples in the plane, visualized through `SamplePlot2D` and `Histogram2D`.

```
In[1]:= <<uvw\datarep.m
In[2]:= uni=Table[{Random[],Random[]},{10000}];
In[3]:= SamplePlot2D[uni,Frame->True,AspectRatio->1]
In[4]:= Histogram2D[uni,0,1,8,0,1,8]
In[5]:= tri=Table[{Min[Random[],Random[]],Min[Random[],Random[]]},{10000}];
In[6]:= SamplePlot2D[tri,Frame->True,AspectRatio->1]
In[7]:= Histogram2D[tri,0,1,8,0,1,8]
In[8]:= x=Transpose[uni][[1]];
In[9]:= y=Transpose[uni][[2]];
In[10]:= uni2=Transpose[{x+y,x-y}];
In[11]:= SamplePlot2D[tri,AspectRatio->1]
In[12]:= tri2=Transpose[{x/(x+y),2*x*y/(x+y)}];
```

```
In[13]:= SamplePlot2D[tri2,AspectRatio->1]
In[14]:= Histogram2D[tri2,0,1,8,0,1,8]
```

## 4 DiscSamp.m (Simulations of discrete distributions)

### 4.1 Package

```
BeginPackage["UVW'DiscSamp'"]
```

```
Distribution::usage="Distribution[listofdata] returns a list of pairs.
The first element of each pair is a value appearing in listofdata,
the second one is the frequency of that value in listofdata."
```

```
RSPermutation::usage="RSPermutation[list,n] returns a list of n random
permutations of list."
```

```
RSExtract::usage="RSExtract[list,k,n] returns a list of n lists of length k,
extracted at random from list."
```

```
RSDiscreteDistribution::usage="RSDiscreteDistribution[dist,n] returns a
random sample of size n of the distribution dist, given under the form of
a list of nonnegative reals. The list dist is first divided by
its sum then interpreted as a probability distribution on the set
{1,...,Length[dist]}."
```

```
Begin["'Private'"]
```

```
Distribution[list_List]:=
```

```
Block[{red,len,shortlen,i},
```

```

    red = Union[list];
    len = Length[list];
    shortlen = Length[red];
    Return[Table[{red[[i]],
N[Count[ list,red[[i]] ]/len}],
{i,shortlen}]
];
];
```

```
RSPermutation[list_List,n_Integer]:=
```

```

    Block[{len,i,r,swap,perm,result={}},
len=Length[list];
Do[
    perm=list;
    For[i=1,i<len,i++,
r=Random[Integer,{i,len}];
swap=perm[[i]];
perm[[i]]=perm[[r]];
perm[[r]]=swap;
    ];
    result=Append[result,perm],
{n}];
Return[result];

];

```

```

RSExtract[list_List,k_Integer,n_Integer]:=

```

```

    Block[{r,listbool={},samp={},extr={},len},

len=Length[list];
Do[
    (
    listbool=Transpose[{list,Table[False,
{len}]}];
    extr={};

    Do[
        (
        r=Random[Integer,{1,len}];
        While[listbool[[r,2]],
r=Random[Integer,{1,len}]]
    ];
    extr=Append[extr,listbool[[r,1]]];
    listbool[[r,2]]=True
    )
    ,{k}];
extr = Sort[extr];
samp = Append[samp,extr]
)
,{n}];
Return[samp];

```

```

];

RSDiscreteDistribution[dist_List,n_Integer]:=

Block[{cumul={},j,choice,res={}},

If[VectorQ[dist],
(
cumul = N[dist];
cumul = cumul/Apply[Plus,cumul];
cumul = Rest[FoldList[Plus,0,cumul]];
Do[
(
choice=Random[];
j=1;
While[choice>cumul[[j]],
j=j+1];
res=Append[res,j]
)
,{n}];
Return[res];
)
];
];

End[]

EndPackage[]

```

## 4.2 Examples

Random samples of classical discrete distributions can be simulated using the standard function `Random`, together with the distributions defined in the package `Statistics`DiscreteDistributions``.

```

In[1]:= Table[Random[Integer],{100}]
In[2]:= Table[Random[Integer,{2,4}],{100}]
In[3]:= <<Statistics`DiscreteDistributions`
In[4]:= Table[Random[BinomialDistribution[3,0.5]],{100}]
In[5]:= Table[Random[GeometricDistribution[0.5]],{100}]
In[6]:= Table[Random[PoissonDistribution[1.]],{100}]

```

To play Loto, one has to extract a sample of 6 numbers from the set  $\{1, \dots, 49\}$ . It can be done with `RSPermutation` or `RSExtract`. One can use also the functions `RandomPermutation` or `RandomSubset` of the standard package `DiscreteMath`Combinatorica``. That package contains several other functions that return random discrete objects, such as graphs, tableaux, trees, heaps...

```
In[1]:= <<uvw\discsamp.m
In[2]:= fournine=Range[49];
In[3]:= perm=RSPermutation[fournine,10];
In[4]:= loto1=Transpose[Take[Transpose[perm],6]];
In[5]:= MatrixForm[%]
In[6]:= loto2=RSExtract[fournine,6,100];
In[7]:= Distribution[Flatten[loto2]]
In[8]:= dist=Transpose[%][[2]]
In[9]:= <<Graphics`Graphics`
In[10]:= BarChart[dist]
In[11]:= PieChart[dist]
```

The following example uses `RSDiscreteDistribution` to illustrate the Law of Large Numbers.

```
In[1]:= <<uvw\discsamp.m
In[2]:= dist={.2,.3,.5}
In[3]:= sample1=RSDiscreteDistribution[dist,100];
In[4]:= sample2=RSDiscreteDistribution[dist,1000];
In[5]:= sample3=RSDiscreteDistribution[dist,10000];
In[6]:= Distribution[sample1]
In[7]:= Distribution[sample2]
In[8]:= Distribution[sample3]
```

## 5 DynSyst.m (Dynamical Systems)

### 5.1 Package

```
BeginPackage["UVW`Dynsyst`"]
```

```
Mean::usage= "Mean[list] returns the arithmetic mean of list."
```

```
Variance::usage= "Variance[list] returns the maximum likelihood estimate of the variance of list."
```

```
TentFunction::usage= "TentFunction[a,x] returns 1-a*Abs[x-1+1/a]."
```

```
LogisticFunction::usage= "LogisticFunction[a,x] returns ax(1-x)."
```

IterateAPhi::usage="
IterateAPhi[matrixA,functionPhi,vectorX0,n] computes and returns in a list
the images by the function Phi of the vector X0 and its products by the
successive powers of the matrix A. The coordinates of these vectors are
reduced to their decimal parts."

CovarianceFunction::usage= "
CovarianceFunction[list,N] returns in a list the values cov(i) for i
ranging from 0 to N . The value cov(i) is the covariance of list with
its i-th shift."

AsymptoticVariance::usage= "AsymptoticVariance[list,N] returns the sum of
the values cov(i) for i ranging from 0 to N . The value cov(i) is the
covariance of list with its i\_th shift."

CorrelationDimension::usage= "
CorrelationDimension[list,step,nstep] computes the values of C(r) , r
being an integer multiple of step, up to nstep values. Then the values
Log[C(r)] as a function of r are plotted, and the linear regression
coefficients are computed. The slope of the regression line and the
correlation coefficient are printed."

Begin["'Private'"]

Mean[list\_]:=N[Apply[Plus,list]/Length[list]];

Variance[list\_]:=((list.list)/Length[list])-(Mean[list]^2);

LogisticFunction[a\_,x\_]:=N[a\*x\*(1-x)];

TentFunction[a\_,x\_]:= 1 - a\*Abs[x-1+1./a];

IterateAPhi[a\_,phi\_,x0\_,n\_]:=

Block[{f,x,res,i},

f[x\_]:= Mod[a.x,1.];

res = NestList[f,Mod[x0,1.],n];

res = Table[Mod[phi[ res[[i,1]],res[[i,2]] ],1.], {i,n}];

Return[res];

```

];

CovarianceFunction[listofdata_List,n_]:=
  Block[{lcent,k,len,covar},

    lcent = listofdata-Mean[listofdata];
    len   = Length[listofdata];
    covar = Table[(Drop[lcent,-k].
Drop[lcent, k])/(len-k),{k,0,n}];
    Return[covar];
  ];

AsymptoticVariance[listofdata_List,n_]:=
  Apply[Plus, CovarianceFunction[listofdata,n]];

CorrelationDimension[listofdata_List,step_,nstep_]:=
  Block[{n,twoover,list,sum,i,elem, work,
    abscissas, ordinates,mx,my,vx,vy,cova,corr,a,b},

    n      = Length[listofdata]-1;
    twoover = 2./(n*(n-1));
    list = listofdata;
    sum = Table[0,{nstep}];

    Do [
      (
        elem=First[list];
        list=Drop[list,1];
        work=Ceiling[Abs[(list-elem)/step]];
        sum=sum+Table[Count[work,i],{i,nstep}];
      ),{n}
    ];

    abscissas = Range[step,nstep*step,step];
    ordinates = Log[sum*twoover];

    mx = Mean[abscissas];
    my = Mean[ordinates];
    vx = Variance[abscissas];
    vy = Variance[ordinates];
    cova = (abscissas.ordinates)/nstep - (mx*my);
    corr = cova/Sqrt[vx*vy];

```

```

a = cov/vx;
b = -a*mx + my;

Print["          "];
Print["Slope = ",a];
Print["          "];
Print["Linear Correlation = ",corr];

g1 := ListPlot[Transpose[{abscissas,ordinates}],
DisplayFunction->Identity];
g2 := Plot[a*x+b,{x,0,Last[abscissas]},
DisplayFunction->Identity];
Show[g1,g2,DisplayFunction->$DisplayFunction]

];

End[]

EndPackage []

```

## 5.2 Examples

Can successive iterates of the logistic function be taken as random reals in the interval  $[0, 1]$ ?

```

In[1]:= <<uvw\dynsyst.m
In[2]:= <<uvw\datarep.m
In[3]:= f[x_]:=LogisticFunction[4.,x]
In[4]:= samp=NestList[f,0.23,1000];
In[5]:= RegularHisto[samp,0,1,10]
In[6]:= samp2=Partition[samp,2];
In[7]:= SamplePlot2D[samp2]
In[8]:= mu=Mean[samp]
In[9]:= sigma=Sqrt[AsymptoticVariance[samp,10]]
In[10]:= CentralLimit[samp,mu,sigma,10]

```

The function `IterateAPhi` generates better samples.

```

In[1]:= <<uvw\dynsyst.m
In[2]:= <<uvw\datarep.m
In[3]:= mat={{1,2},{1,1}}
In[4]:= phi[x_,y_]:=x+y
In[5]:= vec={0.23,0.12}
In[6]:= samp=IterateAPhi[mat,phi,vec,1000];
In[7]:= RegularHisto[samp,0,1,10]
In[8]:= samp2=Partition[samp,2];

```

```

In[9]:= SamplePlot2D[samp2]
In[10]:= mu=Mean[samp]
In[11]:= sigma=Sqrt[AsymptoticVariance[samp]]
In[12]:= CentralLimit[samp,mu,sigma,10]
In[13]:= CorrelationDimension[samp,0.05,5]

```

## 6 Fractals.m (Deterministic and random Von Koch curves)

### 6.1 Package

All fractal curve approximations are treated by this package as lists of segments, a segment being a list of two points, each of them being a list of two coordinates in the plane. The package contains a few examples of simple lists of segments (patterns). By replacing each of the segments of a list by a pattern, a new list is obtained, which can be used as another pattern, or a new starting point. This package contains the replacement and iteration functions that permit the production of many different curves, and also their representation.

```
BeginPackage["UVW`Fractals`"]
```

```
(*----- Examples of basic patterns -----*)
```

```
Triangle::usage="
```

```
Triangle returns an equilateral triangle as a list of three segments in
the plane."
```

```
Star::usage="
```

```
Returns a star as a list of six segments in the plane."
```

```
Island::usage="
```

```
Returns a notch and a triangle as a list of eight segments in the plane."
```

```
Battlement::usage="
```

```
Returns a battlement as a list of eight segments in the plane."
```

```
Hat::usage="
```

```
Hat[sharpness] returns a list of four segments in the plane. "
```

```
Wy::usage="
```

```
Wy[angle1,length1,angle2,length2] returns a Y-shaped list of four segments
in the plane. angle1,length1,angle2 and length2 are the parameters of the
two branches."
```

```
RSWy::usage="
```

```
RSWy[n] returns a random sample of size n of outputs of the function Wy."
```

```

RSTruncs::usage="
RSTruncs[n] returns a random sample of vertical segments in the plane."

(*----- Functions for the replacement of segments by patterns -----*)

TransformSegment::usage="
TransformSegment[segment,pattern] returns a list of segments obtained from
pattern by moving it in the plane so as to make its ends fit with
those of the initial segment."

IteratePattern::usage="
IteratePattern[listofsegments,pattern,n] applies TransformSegment to each
of the segments in listofsegments. Iterates n times."

IterateRandomPatterns::usage="
IterateRandomPattern[listofsegments,patterns,probas,n] iterates
n times the following operation : for each of the segments in listofsegments
one of the patterns in patterns is chosen randomly according to a
probability read in probas, and the segment is transformed accordingly
by TransformSegment."

(*----- Graphical representation -----*)

DrawSegments::usage="
DrawSegments[listofsegments] plots the elements of listofsegments in the
plane."

(*-----*)

Begin["Private"]

(*----- Examples of basic patterns -----*)

Triangle = { { {1.,0.},{0.,0.} },
             { {0.,0.},{0.5,N[Sqrt[3]/2]} },
             { {0.5,N[Sqrt[3]/2]},{1.,0.} } };

Island   = { { {0.,0.},{1.,0.} },
             { {1.,0.},{2.,-N[Sqrt[3]/2]} },
             { {2.,-N[Sqrt[3]/2]},{3.,-N[Sqrt[3]/2]} },
             { {2.,0.},{2.5,N[Sqrt[3]/2]} },
             { {2.5,N[Sqrt[3]/2]},{3.,0.} },
             { {3.,0.},{2.,0.} },

```

```

    { {3.,-N[Sqrt[3]/2]},{4.,0.} },
    { {4.,0.},{5.,0.} } };

Star      = { { {0.,0.},{1.,0.} },
    { {0.,0.},{0.5,N[Sqrt[3]/2]} },
    { {0.,0.},{-0.5,N[Sqrt[3]/2]} },
    { {0.,0.},{-1.,0.} },
    { {0.,0.},{-0.5,-N[Sqrt[3]/2]} },
    { {0.,0.},{0.5,-N[Sqrt[3]/2]} } };

Battlement = { { {0.,0.},{1.,0.} },
    { {1.,0.},{1.,1.} },
    { {1.,1.},{2.,1.} },
    { {2.,1.},{2.,0.} },
    { {2.,0.},{2.,-1.} },
    { {2.,-1.},{3.,-1.} },
    { {3.,-1.},{3.,0.} },
    { {3.,0.},{4.,0.} } };

Hat[sharpness_] :=
    { { {0.,0.},{1.,0.} },
    { {1.,0.},{1.5,N[sharpness]} },
    { {1.5,N[sharpness]},{2.,0.} },
    { {2.,0.},{3.,0.} } };

Wy[angle1_,length1_,angle2_,length2_] :=
    { { {0.,0.},{0.,1.} },
    { {0.,1.},{N[length1*Sin[angle1]],
    1.+N[length1*Cos[angle1]]} },
    { {0.,1.},{-N[length2*Sin[angle2]],
    1.+N[length2*Cos[angle2]]} },
    { {0.,1.3},{0.,2.} } };

RSWy[n_Integer] := Table[Wy[N[Pi/2]*Random[],Random[],
    N[Pi/2]*Random[],Random[]],{n}];

RSTruncs[n_Integer] :=
    Block[{abs,ord,height,truncs},

    truncs={};
    Do[(
    abs      = Random[Real,10.];
    ord      = Random[Real,3.];
    height   = Random[Real,{4.,7.}];
    truncs   = Append[truncs,{{abs,ord},{abs,ord+height}}];

```

```

    ),{n}];

Return[truncs];
];

(*----- Functions for the replacement of segments by patterns -----*)

TransformSegment[segment_List,pattern_List]:=

Block[{i,firstofpat,lastofpat,normofpat,
firstofseg,lastofseg,normofseg,
sinangle,cosangle,
point, transpat},

(* First and last point met in pattern and their distance --*)

firstofpat = First[Flatten[pattern,1]];
lastofpat  = Last[Flatten[pattern,1]];
normofpat  = N[Sqrt[ (lastofpat-firstofpat).
    (lastofpat-firstofpat) ]];

(* End points and length of segment -----*)

firstofseg = segment[[1]];
lastofseg  = segment[[2]];
normofseg  = N[Sqrt[ (lastofseg-firstofseg).
    (lastofseg-firstofseg)]];

(* Sine and Cosine of the angle between segment and pattern *)

cosangle   = ((lastofseg-firstofseg).(lastofpat-firstofpat))
    /(normofseg*normofpat);
sinangle   = (({0.,-1.},{1.,0.}).(lastofseg-firstofseg))
    .(lastofpat-firstofpat))
    /(normofseg*normofpat);

(* Transformations that map the end points of pattern -----*)
(* onto those of segment -----*)

translation[point_List]:=point-(firstofpat-firstofseg);

rotation[point_List]:={({cosangle,sinangle},
{-sinangle,cosangle})
.(point-firstofseg))
+ firstofseg;

```

```

homothety[point_List]:= (point-firstofseg)
  *normofseg/normofpat + firstofseg;

  (* Transformation of pattern point by point -----*)
transpat=Flatten[pattern,1];
transpat=Table[homothety[rotation[translation[transpat[[j]] ]]],
{j,1,Length[transpat]}];
transpat=Partition[transpat,2];

Return[transpat];
]; (* End of function TransformSegment -----*)

```

```

IteratePattern[listofsegments_List,pattern_List,n_Integer]:=

  Block[{i,jagged},

jagged = listofsegments;

Do[(
  (* Replace all segments in jagged by a transformed pattern -*)

  jagged = Table[TransformSegment[jagged[[i]],pattern],
{i,1,Length[jagged]}];
  jagged = Flatten[jagged,1];
),{n}];

Return[jagged];
]; (* End of function IteratePattern-----*)

```

```

IterateRandomPatterns[listofsegments_List,patterns_List,
  probas_List,n_Integer]:=

  Block[{i,j,cumul,jagged},

  (* Distribution function of probas -----*)
cumul = N[probas];
cumul = cumul/Apply[Plus,cumul];
cumul = Rest[FoldList[Plus,0,cumul]];

  (* Definition of random choice according to probas -----*)
choice :=
  Block[{j,ran},

```

```

    ran=Random[];
    j=1;
    While[ ran>cumul[[j]] , j=j+1];
    Return[j];
];

jagged = listofsegments;

Do[(
    (* Replace all segments in jagged by a transformed pattern -*)

    jagged = Table[TransformSegment[jagged[[i]],
    patterns[[choice]]
    ],
    {i,1,Length[jagged]}];
    jagged = Flatten[jagged,1];
    ),{n}];

Return[jagged];
]; (* End of function IteratePattern -----*)

(*----- Graphical representation -----*)

DrawSegments[listofsegments_List,opts___]:=
    Block[{i},

    Show[Graphics[Table[Line[listofsegments[[i]]],
    {i,1,Length[listofsegments]}]], opts,
    Frame->True, FrameTicks->None]
    ] (* End of function DrawSegments-----*)

End[]

EndPackage[]

```

## 6.2 Examples

The celebrated Von Koch curves are constructed by iteratively replacing each segment of a jagged line by a given pattern.

```

In[1]:= <<uvw\fractals.m
In[2]:= h=Hat[Sqrt[3]/2];
In[3]:= DrawSegments[h]

```

```

In[4] := snowflake=IteratePattern[Triangle,h,4];
In[5] := DrawSegments[snowflake,AspectRatio->1];
In[6] := snow1=IteratePattern[Triangle,h,1];
In[7] := snow2=IteratePattern[snow1,h,1];
In[8] := snow3=IteratePattern[snow2,h,1];
In[9] := snow4=IteratePattern[snow3,h,1];
In[10] := g1=DrawSegments[snow1,AspectRatio->1,DisplayFunction->Identity]
In[11] := g2=DrawSegments[snow2,AspectRatio->1,DisplayFunction->Identity]
In[12] := g3=DrawSegments[snow3,AspectRatio->1,DisplayFunction->Identity]
In[13] := g4=DrawSegments[snow4,AspectRatio->1,DisplayFunction->Identity]
In[14] := picture={{g1,g2},{g3,g4}};
In[15] := Show[GraphicsArray[picture],DisplayFunction->$DisplayFunction]
In[16] := DrawSegments[Battlement];
In[17] := IteratePattern[Battlement,Battlement,3];
In[18] := DrawSegments [%]
In[19] := y=Wy[Pi/6,1,Pi/6,1];
In[20] := DrawSegments[y,AspectRatio->1]
In[21] := IteratePattern[Star,y,4];
In[22] := DrawSegments[%,AspectRatio->1]
In[23] := DrawSegments[Island];
In[24] := IteratePattern[Island,Island,3];
In[25] := DrawSegments [%]

```

Random fractals are much better models for real life.

```

In[1] := <<uvw\fractals.m
In[2] := h=Hat[Sqrt[3]/2];
In[3] := base={{{0.,0.},{1.,0.}}};
In[4] := IterateRandomPatterns[base,{h,Island},{0.5,0.5},3];
In[5] := DrawSegments [%]
In[6] := IterateRandomPatterns[base,{h,Island},{0.5,0.5},3];
In[7] := DrawSegments [%]
In[8] := weights=Table[0.25,{4}];
In[9] := madhatter=Table[Hat[2*Random[]],{4}];
In[10] := IterateRandomPatterns[base,madhatter,weights,4];
In[11] := DrawSegments [%]
In[12] := madhatter=Table[Hat[2*Random[]],{4}];
In[13] := IterateRandomPatterns[base,madhatter,weights,4];
In[14] := DrawSegments [%]
In[15] := ry=Wy[(Pi/2)*Random[],Random[],(Pi/2)*Random[],Random[]];
In[16] := tree=IteratePattern[ry,ry,4];
In[17] := DrawSegments [tree]
In[18] := branches=RSWy[4];
In[19] := forest=IterateRandomPattern[RSTruncs[10],branches,weights,3];
In[20] := DrawSegments [forest]

```

## 7 Interact.m (Simulation of spin systems in the plane)

### 7.1 Package

This package contains functions, examples of configurations and transition rates for the simulation of spin system on two-dimensional square grids. The grids are supposed to be toric (periodic boundary conditions), and the flip rates at one site depend on the state (0 or 1) of the site as well as the number of its neighbors in state 1.

```
BeginPackage["UVW'Interact'"]
```

```
(*----- Examples of basic configurations -----*)
(* All configurations are two-dimensional arrays of 0's and 1's. The      *)
(* element config[[x,y]] is interpreted as the state of site (x,y).    *)
(*-----*)
```

```
Checkerboard::usage="
```

```
Checkerboard returns a 40-by-40 configuration of 0's and 1's arranged in
10 by 10 squares."
```

```
Diagonals::usage="
```

```
Diagonals returns a 40-by-40 configuration of 0's and 1's arranged in
diagonal stripes."
```

```
RConfig::usage="
```

```
RConfig[p,width,height] returns a width-by-height array of independent
random 0's and 1's, 1 being chosen with probability p."
```

```
(*----- Examples of rates -----*)
(* All lists of rates are returned as a 2-by-5 list of reals. in such a *)
(* list, rate[[i,j]] represents the rate at which the configuration will *)
(* change at a site in state i (0 or 1) having j (from 0 to 4) neighbors *)
(* in state 1.                                                              *)
(*-----*)
```

```
Uniform::usage="
```

```
Uniform[lambda,mu] are the rates corresponding to the case where the
configuration changes from 0 to 1 at rate lambda and from 1 to 0 at rate
mu, independently from the number of neighbors in state 1."
```

```
Ising::usage="
```

```
Ising[Alpha,Beta] returns the rates corresponding to the symmetric
Stochastic Ising Model, admitting a Gibbs measure as a reversible state.
Alpha is the potential of a site alone, Beta is the potential of a pair
of neighboring sites."
```

```

Contact::usage="
Contact[lambda] returns the rates of the contact process. The transition
rates from 1 to 0 (curing) is constant. The rate of transition from
0 to 1 is proportional to the number of neighbors at 1 (infecting)."

Voter::usage="
Voter returns the rates of the Voter Model, where the transition rate from
0 to 1 is proportional to the number of neighbors in state 1 and
vice versa."

(*----- Treatment of a configuration -----*)

Cyclic::usage="
Cyclic[n,bound] returns bound if n is 0, 1 if n is bound+1 , n in any other
case. (Periodic boundary conditions are assumed for all configurations.)"

RepartConfig::usage="
RepartConfig[config] returns a 2-by-5 list of integers. Its element
[[i,j]] is the number of sites in state i (0 or 1) having j (from 0 to 4)
neighbors in state 1."

Evolution::usage="
Evolution[initialconfig,rates,niter] simulates the evolution of a
configuration according to the spin system corresponding to rates. niter
iteration are performed. One iteration consists of picking up a site at
random and decide to flip its state or not."

(*----- Graphical representation -----*)

DrawConfig::usage="
DrawConfig[config,opts] plots config as a rectangular array of black and
white squares."

(*-----*)
(*-----*)

Begin["'Private'"]

(*----- Examples of basic configurations -----*)

Checkerboard :=
  Block[{i,j},
    Return[Table[If[ Xor[ Mod[Quotient[i,5],2]==0,
Mod[Quotient[j,5],2]==0 ]

```

```

,1,0],
    {i,0,39},{j,0,39}]
];
];

Diagonals :=
    Block[{i,j},
    Return[Table[If[ Mod[i+j,10]<5
,1,0],
    {i,0,39},{j,0,39}]
];
];

RConfig[p_Real,height_Integer,width_Integer] :=
    Table [If [Random[]<p,1,0],{height},{width}];

(*----- Examples of rates -----*)

Uniform[lambda_,mu_] :=
    Transpose[Table[{N[lambda],N[mu]},{5}]];

Ising[alpha_,beta_] :=
    Block[{i},
    Return[Transpose[
    Table[{N[Exp[alpha+(2*i-4)*beta]],
    N[Exp[-alpha+(4-2*i)*beta]]}
, {i,0,4}]
]
];
];

Contact[lambda_] :=
    Block[{i},
    Return[Transpose[Table[{N[lambda*i] , 1.} , {i,0,4}]]];
];

Voter :=
    Block[{i},
    Return[Transpose[Table[{N[i] , N[4-i]} , {i,0,4}]]];
];

(*----- Treatment of a configuration -----*)

Cyclic[n_Integer,bound_Integer] :=

```

```

Switch[n,0,bound,bound+1,1,_,n];

RepartConfig[config_List]:=
  Block[{w,h,x,y,state,nei,dist},
    (* Dimensions of the grid -----*)
    w = Length[Transpose[config]];
    h = Length[config];

    dist = Table[0,{2},{5}];

    Do[(
      (* For all sites -----*)
      (* State of site (x,y) -----*)
      state = config[[x,y]];
      (* Count its neighbors at 1 -----*)
      nei = 1;
      nei = nei + config[[Cyclic[(x-1),h],y]];
      nei = nei + config[[Cyclic[(x+1),h],y]];
      nei = nei + config[[x,Cyclic[(y-1),w] ]];
      nei = nei + config[[x,Cyclic[(y+1),w] ]];
      state = state+1;
      (* Increment dist[[state]][[nei]] -*)
      dist[[state,nei]] = dist[[state,nei]]+1;

      ),{x,1,h},{y,1,w}];

    Return[dist];
  ];

```

```

Evolution[initialconfig_List,rates_List,niter_Integer] :=

```

```

  Block[{w,h,config,proba,x,y,nei},
    (* INITIALIZATIONS *)
    (* Dimensions of the grid -----*)
    w = Length[Transpose[initialconfig]];
    h = Length[initialconfig];
    (* The curent configuration -----*)
    config = initialconfig;
    (* Probabilities to flip a site in *)
    (* a given state with n neighbors -*)
    proba = rates/Max[rates];

    (* MAIN LOOP *)

```

```

Do[
  (
    (* Choose a site at random -----*)
    x = Random[Integer,{1,h}];
    y = Random[Integer,{1,w}];
    (* Count its neighbors at 1 -----*)
    nei = 1;
    nei = nei + config[[Cyclic[x-1,h],y]];
    nei = nei + config[[Cyclic[x+1,h],y]];
    nei = nei + config[[x,Cyclic[y-1,w]]];
    nei = nei + config[[x,Cyclic[y+1,w]]];
    (* Decide to flip it or not -----*)
    If[Random[]<proba[[1+config[[x,y]],nei]],
      (* Flip it -----*)
      config[[x,y]] = 1-config[[x,y]]
    ];
  ),{niter}];

Return[config];
];

(*----- Graphical representation -----*)
DrawConfig[config_List,opts__]:=
  Show[Graphics[Raster[config]], opts
    Frame->True,
    FrameTicks->None,
    AspectRatio->
N[Length[config]/Length[Transpose[config]]]
]

End[]

EndPackage[]

```

## 7.2 Examples

Here is the evolution of the voter model on a square grid of  $40 \times 40$  after 5000 and 10000 iterations.

```

In[1]:= <<uvw\interact.m
In[2]:= whims=RConfig[0.5,40,40];
In[3]:= g1=DrawConfig[whims,DisplayFunction->Identity]
In[4]:= opinions=Evolution[whims,Voter,5000];
In[5]:= g2=DrawConfig[opinions,DisplayFunction->Identity]
In[6]:= opinions=Evolution[opinions,Voter,5000];
In[7]:= g3=DrawConfig[opinions,DisplayFunction->Identity]

```

```
In[8]:= Show[GraphicsArray[{g1,g2,g3}],DisplayFunction->$DisplayFunction]
In[9]:= RepartConfig[whims]
In[10]:= RepartConfig[opinions]
```

The Contact Process is a model of epidemy. If the parameter  $\lambda$  is smaller than its critical value, the population gets cured. If it is larger, the epidemy lasts forever.

```
In[1]:= <<uvw\interact.m
In[2]:= outburst=RConfig[0.05,20,20];
In[3]:= DrawConfig[outburst]
In[4]:= epidemy=Evolution[outburst,Contact[2],2000];
In[5]:= DrawConfig[epidemy]
In[6]:= cured=Evolution[epidemy,Contact[0.5],5000];
In[7]:= DrawConfig[cured]
In[8]:= RepartConfig[outburst]
In[9]:= RepartConfig[epidemy]
In[10]:= RepartConfig[cured]
```

Spin systems are also used in image analysis. Here is a simple example of an image, first blurred by a random noise, then randomly cleaned by two spin systems.

```
In[1]:= <<uvw\interact.m
In[2]:= check=Checkerboard;
In[3]:= DrawConfig[check]
In[4]:= noisy=Evolution[check,Uniform[1,1],100]
In[5]:= DrawConfig[noisy]
In[6]:= soaprates={{0,0,0,1,1},{1,1,0,0,0}};
In[7]:= clean1=Evolution[noisy,soaprates,1000];
In[8]:= DrawConfig[clean1]
In[9]:= clean2=Evolution[noisy,Ising[0,1],1000];
In[10]:= DrawConfig[clean2]
In[11]:= RepartConfig[check]
In[12]:= RepartConfig[noisy]
In[13]:= RepartConfig[clean1]
In[14]:= RepartConfig[clean2]
```

## 8 Lorentz.m (Lorentz's attractor)

### 8.1 Package

Reference: J.C. Culioli Introduction à *Mathematica*, Ellipses, Paris (1991).

```
BeginPackage["UVW'Lorentz'"]
```

```

Lorentz::usage="
Lorentz[s,r,b] computes and plots an approximate solution of the
Lorentz equations with parameters (s,r,b), by a Runge-Kutta method."

LorentzArray::usage="
LorentzArray[matrix] plots an array of solutions of the Lorentz equations
for the values of the parameters contained in matrix."

Begin["Private"]

Lorentz[s_,r_,b_,opts___]:=
  Block[{listofpoints},
    RKuttable[f_List, y_List, y0_List, {t1_,dt_}]:=
      Block[{k1,k2,k3,k4,yp0 = N[y0]},
        Table[ k1 = dt N[f /. Thread[y -> yp0]];
          k2 = dt N[f /. Thread[y -> yp0 + k1/2]];
          k3 = dt N[f /. Thread[y -> yp0 + k2/2]];
          k4 = dt N[f /. Thread[y -> yp0 + k3]];
          yp0 = yp0 + (k1 + 2k2 + 2k3 + k4)/6,
          {Round[N[t1/dt]]}]]];
    Length[f]==Length[y]==Length[y0];

    listofpoints= RKuttable[{- s (x-y), - x z + r x - y, x y - b z},
      {x,y,z}, {0,1,0},{20.,0.025}];

    Show[ Graphics3D[ {Line[listofpoints]} ], PlotRange->All, opts ]
  ]

LorentzArray[matrix_List]:=

  Show[ GraphicsArray[
    Table[ Lorentz[
      matrix[[i,j,1]],matrix[[i,j,2]],matrix[[i,j,3]],
      DisplayFunction -> Identity
    ],
      {i,Length[matrix]},
      {j,Length[First[matrix]]}
    ]]]

End[]

```

```
EndPackage []
```

## 8.2 Examples

The Lorentz function is rather slow. On a system powerful enough, it can be coupled with `Animate` or `ShowAnimation`.

```
In[1]:= <<uvw\lorentz.m  
In[2]:= Lorentz[3,26.5,1]  
In[3]:= para={{3,26.5,1},{3,25,1}},{{4,26.5,1},{4,25,1}}};  
In[4]:= LorentzArray[para]
```

## 9 PseuGene.m (Congruential and midsquare generators)

### 9.1 Package

```
BeginPackage["UVW'Pseugene'"]
```

```
CongruGenerator::usage = "
```

```
CongruGenerator[seed,a,c,m,n] returns a list of the n first iterates of the  
congruential generator
```

```
   $x(n+1) = a x(n) + c \text{ Modulo } m .$ 
```

```
seed is the first element x(0). If seed is an integer, the result will  
be a list of integers between 0 and m . If seed is real, all the results  
will be divided by m to return a list of reals between 0 and 1 ."
```

```
MidsquareGenerator::usage = "
```

```
MidsquareGenerator[seed,n] returns a list of the first iterates of the  
midsquare generator, starting with seed (four digits integer)."
```

```
CongruentialLoop::usage = "
```

```
CongruentialLoop[seed,a,c,m] returns in a list the loop of the congruential  
generator
```

```
   $x(n+1) = a x(n) + c \text{ Modulo } m ,$   
starting with  $x(0)=seed .$ "
```

```
MidsquareLoop::usage ="
```

```
MidsquareLoop[seed] returns in a list the loop of the midsquare generator,  
starting with seed."
```

```

Begin["Private"]

CongruGenerator[seed_,a_,c_,m_,n_]:=

  Block[ {f, y, res},

f[y_]:= Mod[ a y + c, m ];

If[ IntegerQ[seed],
  res = NestList[ f, seed, n-1 ] ,
  res = NestList[ f, m*(seed-Floor[seed]) , n-1 ] /m
];

Return[res];
];

MidsquareGenerator[seed_,n_]:=

  Block[ {mids,x},

mids[x_]:=Block[{res},
  res = Floor[x*x/100];
  res = res - (10000*Floor[res/10000]);
  Return[res];
];
Return[ NestList[ mids , seed , n-1 ] ];

];

CongruentialLoop[seed_,a_,c_,m_]:=

  Block[ {f, y , pos , element , loop={ } },

f[y_]:= Mod[ a y + c, m ];

loop = Append[loop,seed];
element= f[ seed ];

```

```

While[ !MemberQ[loop,element],
(
loop = Append[loop,element];
element = f[element]
)
];

pos = First[Flatten[Position[loop,element]]];
loop = Drop[ loop, pos-1 ];
loop = Append[loop,element];
Return[loop];

];

MidsquareLoop[seed_]:=

Block[ {f, x , pos , element , loop={} },

f[x_]:=Block[{res},
res = Floor[x*x/100];
res = res - (10000*Floor[res/10000]);
Return[res];
];
loop = Append[loop,seed];
element = f[ seed ];

While[ !MemberQ[loop,element],
(
loop = Append[loop,element];
element = f[element]
)
];

pos = First[Flatten[Position[loop,element]]];
loop = Drop[ loop, pos-1 ];
loop = Append[loop,element];
Return[loop];

];

End[]

EndPackage[]

```

## 9.2 Examples

The midsquare generator is not a very good one...

```
In[1] := <<uvw\pseugene.m
In[2] := MidsquareGenerator[1245,100]
In[3] := MidsquareGenerator[1246,100]
In[4] := MidsquareGenerator[1247,100]
In[5] := MidsquareLoop[4578]
In[6] := MidsquareLoop[9854]
```

Some congruential generators can be reasonably good, others very disappointing.

```
In[1] := <<uvw\pseugene.m
In[2] := samp=CongruGenerator[0.23,181,0,16384,2000];
In[3] := <<uvw\dataarep.m
In[4] := RegularHisto[samp,0,1,10]
In[5] := LargeNumbers[samp]
In[6] := CentralLimit[samp,0.5,Sqrt[1./12],6]
In[7] := samp2=Partition[samp,2];
In[8] := SamplePlot2D[samp2]
In[9] := Length[CongruentialLoop[10,181,0,16384]]
In[10] := Length[CongruentialLoop[1,181,0,16384]]
In[11] := CongruGenerator[10,181,0,16381,20]
In[12] := CongruGenerator[1,181,0,16381,20]
```

## 10 RandWalk.m (Random walks and random vector fields)

### 10.1 Package

```
BeginPackage["UVW`RandWalk`"]
```

```
RandomWalk::usage= "
```

```
RandomWalk[ListofVelocities,Deltat] represents the trajectory of a point
in a square. ListofVelocities is a list of two-dimensional vectors,
interpreted as consecutive speeds for the point. The point starts from
the center with the first speed vector of the list. It changes its speed
vector for the next one in the list at each integer multiple of deltat."
```

```
VectorField::usage= "
```

```
VectorField[arrayof2Dvelocities] represents graphically by segments on a
grid the values of a discrete vector field. Arrayof2Dvelocities is a list
with three levels. The first two correspond to the coordinates (i,j) of a
```

point on the grid. The last level corresponds to the two coordinates of the vector attached to point  $(i,j)$  :  $V_x(i,j)$  ,  $V_y(i,j)$  . The function represents the grid and the vector attached to each point."

```
VectorFieldTrajectory::usage= "
VectorFieldTrajectory[arrayof2Dvelocities, deltat, tmax] represents first a
vector field on a grid by calling VectorField[arrayof2Dvelocities]. Then
it draws the trajectory of a point starting at the center of the grid. At
each integer multiple of deltat, the velocity vector of the point is changed
for that of the vector field at the closest point on the grid.
The trajectory is followed up to time tmax. The boundary conditions are
periodic."
```

```
Begin["Private"]
```

```
RandomWalk[listofvelocities_List,deltat_,opts___]:=
```

```
    Block[{points},
```

```
        points = listofvelocities deltat;
        points = FoldList[Plus,{1.,1.},points];
        points = Mod[points,2.] - Table[{1.,1.},{Length[points]}];
```

```
        ListPlot[points,
        opts,
        Prolog      -> PointSize[0.001],
        PlotRange   -> {{-1,1},{-1,1}},
        Axes        -> False,
        Frame       -> True,
        FrameTicks  -> None]
```

```
    ]
```

```
VectorField[arrayofvelocities_List,opts___]:=
```

```
    Block[{nx,ny,vect,graph,i,j},
```

```
        nx  = Length[arrayofvelocities];
        ny  = Length[First[arrayofvelocities]];
        vect = N[arrayofvelocities];
```

```

vect = vect/ (2 Max[Abs[vect]]);

graph = Table[Line[{{i,j},{i,j}+vect[[i,j]]}],{i,nx},{j,ny}];

Show[Graphics[graph],
     opts,
     Prolog -> Thickness[0.001],
     PlotRange -> {{0,nx+1},{0,ny+1}},
     Axes -> False,
     Frame -> True,
     FrameTicks -> None]
]

```

```

VectorFieldTrajectory[arrayofvelocities_List,deltat_,tmax_]:=

```

```

Block[{nx,ny,maxx,maxy,nstep,vect,x,y,point,traj,i,j,gvect,gtraj},

nx = Length[arrayofvelocities];
maxx = nx - 1.;
ny = Length[First[arrayofvelocities]];
maxy = ny - 1.;
nstep = Floor[tmax/deltat];
x = nx/2.;
y = ny/2.;
point={x,y};
traj = {point};
vect = N[arrayofvelocities];

Do [
  (
    point = point + deltat*vect[[ Round[x]+1,Round[y]+1 ]];
    x = Mod[First[point],maxx];
    y = Mod[Last[point],maxy];
    point = {x,y};
    traj = Append[traj,point]
  ),
  {nstep}
];

traj=traj+Table[{0.5,0.5},{Length[traj]}];

vect = vect / (2 Max[Abs[vect]]);

```

```

gvect = Table[Line[{{i,j},{i,j}+vect[[i,j]]}],{i,nx},{j,ny}];

gtraj = Table[Point[traj[[i]]],{i,Length[traj]}];

Show[Graphics[gvect],Graphics[gtraj],

      Prolog -> Thickness[0.001],
      PlotRange -> {{0,nx+1},{0,ny+1}},
      Axes -> False,
      Frame -> True,
      FrameTicks -> None]

];

End[]

EndPackage[]

```

## 10.2 Examples

Here is a representation of a discretized Brownian motion.

```

In[1]:= <<uvw\randwalk.m
In[2]:= <<uvw\contsamp.m
In[3]:= vel=RSNormal2D[1,1,0,1000];
In[4]:= RandomWalk[vel,0.02];
In[5]:= vel=RSNormal2D[1,1,0,1000];
In[6]:= RandomWalk[vel,0.02];

```

Here are two trajectories, one in a deterministic vector field, the other in a random one.

```

In[1]:= <<uvw\randwalk.m
In[2]:= field1=N[Table[{Cos[(i+j)/5Pi],Sin[(i+j)/5Pi]},{i,19},{j,19}];
In[3]:= VectorField[field1]
In[4]:= VectorFieldTrajectory[field1,0.1,10]
In[5]:= field2=Table[{Random[Real,{-1,1}],Random[Real,{-1,1}]},{5},{5}]
In[6]:= VectorFieldTrajectory[field2,0.5,100]

```

## 11 Stogho.m (Stochastic Ghost)

### 11.1 Package

```

BeginPackage["UVW`Stogho`"]

```

```

Stogho::usage="
Stogho[c] portraits the gha(us)stly ghost known by the name of Stogho
in some old castles of Europe. His mood depends heavily on the sine of
the real c ."

GalleryOfPortraits::usage="
GalleryOfPortraits[matrix] draws an array of portraits of your favorite
star."

Begin["'Private'"]

Stogho[c_,opts___]:=

    Block[{h, body, eyes, mouth, pupils},

    h=Sin[N[c]]+3;

        body = ParametricPlot[{t,Exp[(-t^2)/h]},{t,-10,10},
DisplayFunction->Identity,PlotStyle->Thickness[0.009]];

        eyes = ParametricPlot[{{1/3Sin[t]-0.6,1/(h 20)Cos[t]+0.6},
{1/3Sin[t]+0.6,1/(h 20)Cos[t]+0.6}},{t,-Pi,Pi},
DisplayFunction->Identity,PlotStyle->Thickness[0.002]];

        mouth = ParametricPlot[{t,(1/10 h)(t^2)Sin[c]+0.2},{t,-0.75,0.75},
DisplayFunction->Identity,PlotStyle->Thickness[0.007]];

        pupils = Graphics[{PointSize[0.02],
Point[{0.6+(h/22)(-1)^Round[h],0.6}],
Point[{-0.6+(h/22)(-1)^Round[h],0.6}]
}];

        Show[body,eyes,mouth,pupils,
            Axes->False,opts,DisplayFunction->${DisplayFunction}

    ]

GalleryOfPortraits[matrixA_List]:=

```

```
Show[GraphicsArray[Table[Stogho[matrixA[[i,j]],
  DisplayFunction->Identity],
  {i,Length[matrixA]},
  {j,Length[First[matrixA]]}]]]
```

```
End[]
```

```
EndPackage[]
```

## 11.2 Examples

The body, mouth, eyes and pupils can easily be reparametrized in order to change the aspect of the ghost. The function `Stogho` can also be coupled with `Animate` or `ShowAnimation` if he is to be made a movie star.

```
In[1]:= <<uvw\stogho.m
In[2]:= Stogho[Random[Real,{-3,3}]]
In[3]:= moods=Partition[Range[-Pi,Pi,2 Pi/15],4];
In[4]:= GalleryOfPortraits[moods]
```

## 12 TimeRep.m (Queues and time processes)

### 12.1 Package

```
BeginPackage["UVW`Timerep`"]
```

```
Queue::usage= "Queue[interarrivals,services] represents on a graphics as a
function of time, the evolution of the number of customers in a queue
with one server.The times between consecutive arrivals are read
in the first list, the service times in the second one."
```

```
CumulatedTimes::usage= "CumulatedTimes[listoftimes] represents on a graphics
the function of time defined as follows. Starting from 0, it is
incremented by one at dates separated by the times read in listoftimes."
```

```
Geiger::usage= "Geiger[listoftimes] plots on a line the dates separated by
the durations read in listoftimes."
```

```
Begin["`Private`"]
```

```
Queue[interarrivals_List,services_List] :=
```

```

Block[{m, arr={}, dep={}, abs={}, ord={}, events={}},

m=Min[Length[interarrivals],Length[services]];

arr=Rest[FoldList[Plus,0,N[interarrivals]]];
dep=Append[dep,First[arr]+First[services]];

For[k=1, k<m, k++,
  If [(dep[[k]]<arr[[k+1]]),
dep = Append[ dep, arr[[k+1]]+services[[k+1]] ],
dep = Append[ dep, dep[[k]]+services[[k+1]] ]
  ]
];

arr = Transpose[{arr,Table[1.,{Length[arr]}]}];
dep = Transpose[{dep,Table[-1.,{m}]}];
events = Transpose[Sort[Join[arr,dep]]];
abs = events[[1]];
ord = FoldList[Plus,0,events[[2]]];
abs = Flatten[Transpose[{abs,abs}]];
ord = Flatten[Transpose[{Drop[ord,-1],Drop[ord,1]}]];
events = Transpose[{abs,ord}];
events = Prepend[events,{0.,0.}];

ListPlot [events, PlotJoined -> True,
AxesLabel -> {"t","N(t)"}
]

```

```

CumulatedTimes[listoftimes_List] :=

Block[{cumul={},ordinates={},g},

cumul = FoldList[Plus,0,N[listoftimes]];
ordinates = Range[0.,N[Length[listoftimes]-1]];
g = Transpose[ { Transpose[{Drop[cumul,-1],ordinates}] ,
  Transpose[{Drop[cumul,1],ordinates}] } ];

g = Table[Line[g[[i]]],{i,1,Length[g]}];

Show[ Graphics [g] ,

```

```

    Prolog    -> Thickness[0.003],
    Axes      -> True,
    AxesLabel -> {"t","N(t)"}
    ]

Geiger[listoftimes_List]:=

Block[{cumul,max,g},

    cumul = Rest[FoldList[Plus,0,N[listoftimes]]];
    max    = Max[cumul];

    g = Table[Line[{{cumul[[i]],0.},{cumul[[i]],0.01}},
        {i,Length[cumul]}];

    Show[ Graphics [g],
    Prolog    -> Thickness[0.003],
    PlotRange -> {{0.,max},{0.,1.}},
    Axes ->{True,False},
    AxesLabel ->{"t","t"}
    ]

End[]

EndPackage []

```

## 12.2 Examples

Here is an illustration of the Poisson process with different intensities.

```

In[1]:= <<uvw\timerep.m
In[2]:= <<Statistics'ContinuousDistributions'
In[3]:= times=Table[Random[ExponentialDistribution[1.]],{100}];
In[4]:= Geiger[times]
In[5]:= CumulatedTimes[times]
In[6]:= times=Table[Random[ExponentialDistribution[1.]],{500}];
In[7]:= CumulatedTimes[times]
In[8]:= times=Table[Random[ExponentialDistribution[2.]],{100}];
In[9]:= CumulatedTimes[times]
In[10]:= times=Table[Random[ExponentialDistribution[0.5]],{100}];
In[11]:= CumulatedTimes[times]

```

The M/M/1 single server queue has exponentially distributed interarrival and service times.

It may be in equilibrium or saturated according to the values of the mean interarrival and service times.

```
In[1]:= <<uvw\timerep.m
In[2]:= <<Statistics'ContinuousDistributions'
In[3]:= arr=Table[Random[ExponentialDistribution[1.]],{200}];
In[4]:= ser=Table[Random[ExponentialDistribution[1.1]],{200}];
In[5]:= Queue[arr,ser]
In[6]:= ser=Table[Random[ExponentialDistribution[0.9]],{200}];
```

Here is the same illustration with the D/M/1 queue (constant interarrival times).

```
In[1]:= <<uvw\timerep.m
In[2]:= <<Statistics'ContinuousDistributions'
In[3]:= arr=Table[1.,{200}];
In[4]:= ser=Table[Random[ExponentialDistribution[1.1]],{200}];
In[5]:= Queue[arr,ser]
In[6]:= ser=Table[Random[ExponentialDistribution[0.9]],{200}];
```

## 13 ZeroOne.m (Lists of zeros and ones)

### 13.1 Package

```
BeginPackage["UVW'Zeroone'",{"Graphics'Animation'"}]
```

```
RSZeroOne::usage= "
```

```
RSZeroOne[p,n] returns a random sample of n zeros and ones. One is chosen with probability p."
```

```
PlotZeroOne::usage= "
```

```
PlotZeroOne[listofzeroones] represents graphically a list of zeros and ones as black and white squares on a grey background."
```

```
AnimateShift::usage= "
```

```
AnimateShift[listofzeroones] forms a list of zeros and ones three times as long as the initial list, by adding first a list of same length of random digits, then copying the initial list at the end. Then the successive shifts are animated as arrays of black and white squares."
```

```
Binary::usage= "
```

```
Binary[functionf,listofzeroones] computes the real in [0,1] the binary decomposition of which is that of the list. Then the image by functionf (from [0,1] into R) is computed. The decimal part of it is returned under binary form as a new list of zeroones."
```

```

ActualLength::usage= "
ActualLength[listofzeroones] returns the length of the list obtained when
all zeros before the first one are dropped in listofzeroones."

Weight::usage= "
Weight[listofzeroones] returns the number of ones in the list."

WeightedAlphabeticalOrder::usage= "
WeightedAlphabeticalOrder[listofzeroones] computes the rank of the given
list of zeros and ones among lists of same actual length, when they are
ranked according to increasing weights and alphabetical order for lists
of same weight. That rank is returned in base 2 as another list of zeros
and ones."

BinaryNumbers::usage= "
BinaryNumbers[n] returns a list of all the 2^n lists of binary digits with
length n."

Entropy::usage= "
Entropy[t] returns -tLog[2,t]-(1-t)Log[2,1-t]"

Begin["Private"]

RSZeroOne[p_,n_]:= Table[If[Random[]<p,1,0],{n}];

PlotZeroOne[listofzeroones_List,opts___]:=

  Block[{complete,dim},

    dim      = Ceiling[Sqrt[N[Length[listofzeroones]]]];
    complete = Partition[listofzeroones,dim];

    Show[Graphics[Raster[complete]],
    opts,
    AspectRatio->0.9,
    Background->GrayLevel[0.5]]

  ]

AnimateShift[listofzeroones_List]:=

  Block[{dim,dim2,complete,nbshift,walt,disney,i},

```

```

dim      = Length[listofzeroones];
dim2     = Ceiling[Sqrt[N[Length[listofzeroones]]]];

complete = Join[listofzeroones,RSZeroOne[0.5,dim],listofzeroones];
nbshift  = 2*dim+1;
walt     = Table[Partition[
Take[complete,{i,i+dim-1}],dim2],{i,1,nbshift}];

disney   = Table[Graphics[Raster[walt[[i]]]],{i,1,nbshift}];
ShowAnimation[disney]

]

```

```
Binary[f_,listofzeroones_List]:=
```

```

Block[{x,y},

x = listofzeroones.Table[0.5^i, {i,1,Length[listofzeroones]}];
y = f[x];
y = Mod[y,1.];
Return[ Drop[ Flatten[ RealDigits[y,2] ] , -1 ] ];

]

```

```
ActualLength[listofzeroones_List]:=
```

```

Block[{j=1},
While[listofzeroones[[j]]==0,
j=j+1];
Return[Length[listofzeroones]-j+1];
];

```

```
Weight[listofzeroones_List]:=Apply[Plus,listofzeroones];
```

```
WeightedAlphabeticalOrder[listofzeroones_List]:=
```

```

Block[{r,p,u,l,i},

p = Weight[listofzeroones];
l = Length[listofzeroones];
r = 0;

```

```

u=0;
i=0;
While[u<p,
  (
    i=i+1;
    If[listofzeroones[[i]]==1,
      (r=r+Binomial[l,u]+Binomial[l-i,p-u] ; u=u+1)
    ]
  )
];

Return[IntegerDigits[r,2]];
];

```

```
BinaryNumbers[n_Integer]:=
```

```

Block[{m,l,i,number},

  m = 2^n-1;
  l = {Table[0,{n}]};
  i = 0;
  Do[(
    i=i+1;
    number=IntegerDigits[i,2];
    If[Length[number]<n,
      number = Join[Table[0,{n-Length[number]}],number];
      l = Append[l,number]
    ],{m}];

  Return[l];
];

```

```
Entropy[t_]:= -t Log[2,t] -(1-t) Log[2,t];
```

```
End[]
```

```
EndPackage[]
```

## 13.2 Examples

Here is an illustration of the Kolmogorov complexity of a random sequence of zeros and ones.

```
In[1]:= <<uvw\zeroone.m
```

```

In[2]:= samp=RSZeroOne[0.5,100];
In[3]:= len=ActualLength[samp]
In[4]:= wei=Weight[samp]
In[5]:= samp1=WeightedAlphabeticalOrder[samp];
In[6]:= len1=ActualLength[samp1]
In[7]:= wei1=Weight[samp1]
In[8]:= samp2=Binary[Sin,samp];
In[9]:= len2=ActualLength[samp2]
In[10]:= wei2=Weight[samp2]
In[11]:= g=PlotZeroOne[samp,DisplayFunction->Identity]
In[12]:= g1=PlotZeroOne[samp1,DisplayFunction->Identity]
In[13]:= g2=PlotZeroOne[samp2,DisplayFunction->Identity]
In[14]:= Show[GraphicsArray[{g,g1,g2}], DisplayFunction->${DisplayFunction}]

```

The following session illustrates the shifts of a sequence of zeros and ones. The animation may not work on all platforms.

```

In[1]:= <<uvw\zeroone.m
In[2]:= letter={0,0,0,0,0, 0,1,1,1,1, 0,0,0,0,1, 0,1,1,1,1, 0,0,0,0,0};
In[3]:= PlotZeroOne[letter]
In[4]:= AnimateShift[letter]

```