Introduction aux modèles de simulation

L'élaboration et l'étude de modèles de simulation est devenue une activité essentielle dans toute recherche scientifique. CE NOTEBOOK EST UNE BRÈVE INTRODUCTION QUI ILLUSTRE LES GRANDS TYPES DE MODÈLES. Nous consacrerons un notebook plus complet à chacun de ces types.

Le géographe peut distinguer trois grandes catégories de modèles de simulation : les macro-modèles, les micro-modèles, et des meso-modèles. Les premiers sont des ensembles d'équations différentielles ordinaires (EDO), partielles (EDP) ou stochastiques (EDS), ou des systèmes d'équations aux différences, appelées aussi équations de récurrence (ER). Les micro-modèles se divisent en systèmes d'automates cellulaires (AC) et en systèmes multi-agents (SMA). Enfin, les meso-modèles se servent de processus stochastiques. Ils sont parfois assimilés à des micro-modèles.

D'une façon très (trop) schématique, il est possible d'affirmer que les macro-modèles sont très utiles dans les approches théoriques, alors que les micro-modèles prennent mieux en compte les contraintes de la diversité du "réel".

Les macro-modèles de simulation avec Mathematica

La simulation de macro-modèles est rarement employée en géographie, malgré les travaux initiaux de Lena Sanders, du groupe Dupont-Grenoble (modèles AMORAL) et de rares thèses, dont celle de Damienne Provitolo. Ces premiers modèles déterministes étaient a-spatiaux. Ils étaient construits à l'aide d'EDO ou d'ER. En revanche, cette approche était l'objet de recherches intenses dans les disciplines voisines, notamment en économie, écologie ou épidémiologie. Citons les modèles proies-prédateurs, les modèles SIR et leurs dérivés, ou le modèle "Halte à la croissance". Bien évidemment ces modèles très efficaces pour comprendre un phénomène, ont un intérêt limité en termes de prévision.

Le géographe intéressé par la modélisation des systèmes dynamiques se pose généralement quatre questions. La première vise un objectif relativement simple : est-il possible de simuler l'évolution d'un système dynamique quand on connaît les mécanismes qui guident cette évolution? Par exemple peut-on reconstruire la courbe d'évolution de la population française, sachant qu'elle est façonnée par le taux de natalité, de mortalité, les entrées et sorties migratoires? Une fois cette courbe d'évolution obtenue, d'autres questions surgissent. D'abord, cette courbe présente-t-elle des points singuliers, notamment des points stationnaires ou d'équilibre quand l'évolution cesse de croître ou de décroître? Il s'agit de repérer ces états stationnaires du système dynamique étudié. Une nouvelle question surgit alors : ces états stationnaires repérés sont-ils stables ou instables quand ils sont soumis à une perturbation interne ou externe? Dernière question essentielle, les systèmes dynamiques subissent parfois des changements brutaux. Ce sont des changements d'état. Quand la température des eaux d'un lac descend au-dessous de 0°, le lac gèle. Ce passage brutal, pourtant lié à une décroissance régulière de la température, est une bifurcation. Repérer les bifurcations et les qualifier est une autre question essentielle que doit se poser le géographe. D'autres questions plus spécifiques interpellent le géographe. La plus explorée par les sciences contemporaines concerne la sensibilité du système dynamique aux conditions initiales. Si la réponse est positive, il est probable que le système adopte un comportement chaotique quand certains paramètres sont modifiés.

Un système EDO de stocks et de flux : le modèle épidémiologique SIR

Le modèle SIR simule l'évolution d'une épidémie en prenant en compte trois stocks : les sujets sains (S),

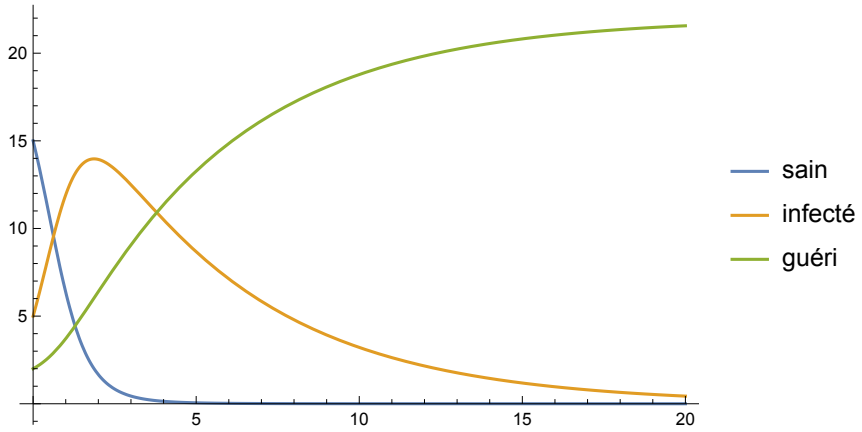
les malades infectés (I) et les sujets guéris susceptibles d'être infectés (R). Dans sa forme la plus simple, il s'écrit :
dS/dt = -aIS
dI/dt = aIS - bI
dR/dt = bI
Les paramètres a, b et c sont compris entre 0 et 1,
aIS est un produit qui quantifie le nombre d'individus susceptibles qui sont infectés par unité de temps, dt, bI est la quantité de personnes infectées qui guérissent.

Pour résoudre ce système de trois EDO, nous utilisons l'instruction **NDSolve**[], avec une valeur initiale différente pour chaque stock (15 pour S, 5 pour I, et 2 pour R). Le système est simulé pour 20 intervalles de temps. Dans l'instruction **NDSolve**[] sont inclus successivement les 3 équations, puis les valeurs initiales , puis le nom des trois variables (s, i, r), et enfin la durée de la simulation (20). Remarquez le double signe == dans chaque équation. En sortie, le programme donne trois fonctions d'interpolation, qui donnent l'évolution traduites en graphiques avec l'instruction **Plot**[]**.** Cette fonction trace la solution pour les trois séries temporelles correspondantes à l'évolution des trois stocks. L'instruction suivante **ParametricPlot**[] donne le champ de pente, donc toutes les évolutions possibles pour les deux stocks I et S. Enfin, l'instruction **StreamPlot**[] permet de figurer une solution qui correspond à des données initiales fournies. Les deux graphiques n'apparaissent pas, car nous avons ajouté un ; à la fin des instructions. Mais, ces deux graphiques sont joints en un seul schéma avec l'instruction **Show**[], et la solution précise se distingue bien des autres solutions.

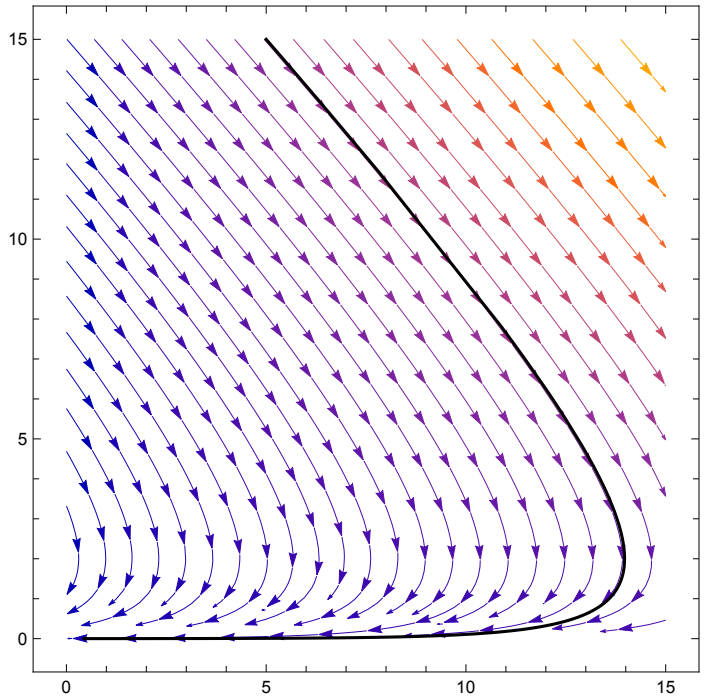
In[•]:=
```
ClearAll["Global`*"]
```
efface tout
```
equation1 = s'[t] == -0.1 s[t] * i[t];
equation2 = i'[t] == 0.1 s[t] * i[t] - 0.2 * i[t];
equation3 = r'[t] == 0.2 * i[t];
sol = NDSolve[{equation1, equation2,
```
résous numériquement équation différentielle
```
   equation3, s[0] == 15, i[0] == 5, r[0] == 2}, {s, i, r}, {t, 20}]
Plot[Evaluate[{s[t], i[t], r[t]} /. First[sol]], {t, 0, 20},
```
tracé évalue premier
```
 PlotLegends → {"sain", "infecté", "guéri"}]
```
légendes de tracé
```
champ = ParametricPlot[Evaluate[{i[t], s[t]} /. sol],
```
tracé paramétrique évalue
```
   {t, 0, 20}, PlotStyle → {Black}];
```
style de tracé noir
```
solution = StreamPlot[{0.1 s i - 0.2 i, -0.1 s i}, {i, 0, 15}, {s, 0, 15}];
```
tracé de lignes de courant
```
Show[{solution, champ}]
```
montre

Out[•]=
```
{{s → InterpolatingFunction[ Domain: {{0., 20. }}
                             Output: scalar ],
 i → InterpolatingFunction[ Domain: {{0., 20. }}
                             Output: scalar ],
 r → InterpolatingFunction[ Domain: {{0., 20. }}
                             Output: scalar ]}}
```

Out[•]=



Out[•]=



Le lecteur est encouragé à jouer avec ce modèle en modifiant les valeurs des stocks initiaux, puis des paramètres. Les modèles relatifs à la pandémie de la Covid 19 comportent d'autres éléments, notamment le temps d'incubation et/ou la quarantaine. Par exemple, le modèle SEI2HR introduit deux populations infectées, légèrement (I1) ou fortement (I2), et les lits d'hôpitaux (H).

Les systèmes d'ER sont généralement plus intuitifs.

Un système d'ER : le modèle proie-prédateur

La résolution du système d'ER proie-prédateur s'apparente à la résolution d'un système d'EDO. Mais

généralement, la résolution se fait avec la fonction **RecurrenceTable**[] dans laquelle on positionne les équations (equat1 et equat2), les valeurs initiales des stocks (150 et 200), la liste des stocks (popa et popb), et la durée de la simulation (50). L'évolution des deux populations est affichée avec la fonction **ListPlot**[]. Puis l'espace des phases est dessiné avec la fonction **Graphics**[].

In[•]:=
```
ClearAll["Global`*"]
```
efface tout

```
equat1 = popa[n] == 0.7 popa[n - 1] + 0.002 popa[n - 1] × popb[n - 1]
equat2 = popb[n] == 1.2 popb[n - 1] - 0.001 popa[n - 1] × popb[n - 1]
sol = RecurrenceTable[
```
table de récurrence

```
    {equat1, equat2, popa[0] == 150, popb[0] == 200}, {popa, popb}, {n, 0, 50}];
sollist = Transpose[sol];
```
transpose

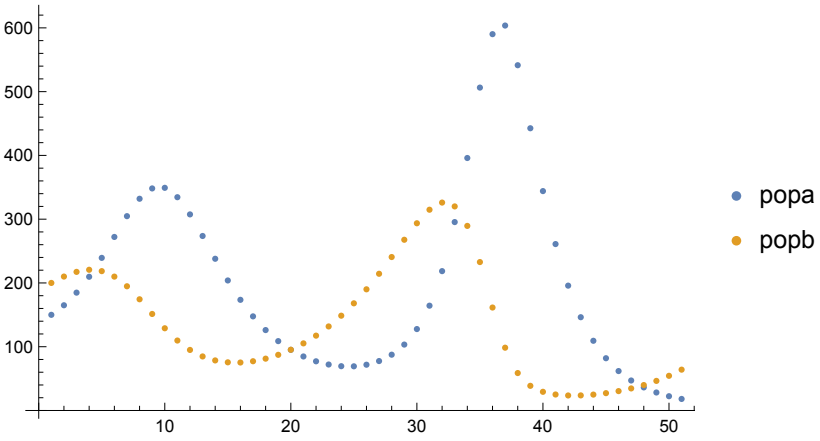
```
ListPlot[sollist, PlotLegends → {"popa", "popb"}]
```
tracé de liste légendes de tracé

```
Graphics[Line[sol]]
```
graphique ligne

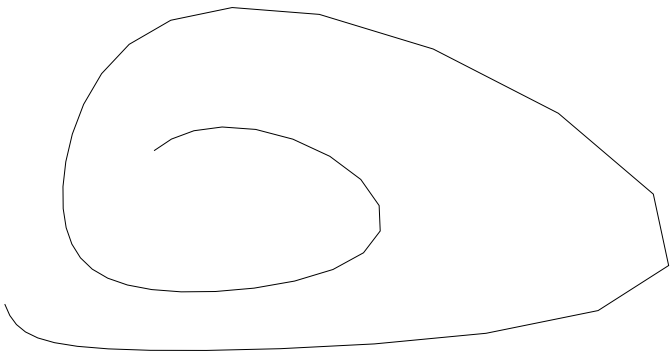
Out[•]=
$$\text{popa}[n] == 0.7\,\text{popa}[-1+n] + 0.002\,\text{popa}[-1+n] \times \text{popb}[-1+n]$$

Out[•]=
$$\text{popb}[n] == 1.2\,\text{popb}[-1+n] - 0.001\,\text{popa}[-1+n] \times \text{popb}[-1+n]$$

Out[•]=



Out[•]=



Le lecteur est encouragé à introduire une troisième équation dans ce modèle. Dans tous ces modèles d'EDO ou d'ER l'espace est absent. Très utile pour comprendre, ces modèles s'avèrent imparfaits, voire erronés, quand les contraintes spatiales sont à prendre en compte. Le géographe doit alors employer des modèles d'EDP. La résolution d'un système d'EDP comporte les mêmes étapes que la résolution d'un système d'EDP. Mais, avec la version 12.2 Mathematica offre de nouvelles potentialités pour le chercheur. Nous les utiliserons dans le notebook consacré à cette seule forme de modélisation. Dans ce notebook,

nous proposons seulement un modèles d'EDP programmé avec une version antérieure (12.0).

Un modèle simple d'EDP : le modèle de Burger

Le modèle de Burger simule le double mouvement de diffusion et d'advection d'une population qui demeure stable, sans croissance. La première instruction donne une valeur constante à la viscosité imposée par l'espace. La deuxième résout le système. La première solution, qui correspond à la situation initiale sera la première affichée. Dans l'instruction **NDSolve[]** est placée l'équation, où u représente la population étudiée, x et y les deux dimensions spatiales, D représente la dérivée partielle de u par rapport au temps, t, et aux deux dimensions spatiales, x et y. Puis, toujours à l'intérieur de l'instruction **NDSolve[]**, on entre la répartition initiale de la population au temps t0, quatre conditions bordières égales dans ce cas, la variable qui est analysée, u dans ce modèle, puis le temps d'intégration, qui varie de 1 à 4, et enfin l'espace qui est un carré dans cet exemple. Le lecteur attentif remarque qu'à l'origine la population est concentrée dans un coin du carré. Le résultat est encore une fonction d'interpolation. La dernière instruction, **Manipulate[]**, permet de disposer des solutions visuelles en trois dimensions suivant le temps accordé à l'advection et à la diffusion. En faisant coulisser le disque sur la ligne temps du graphique on observe comment évolue ce système.

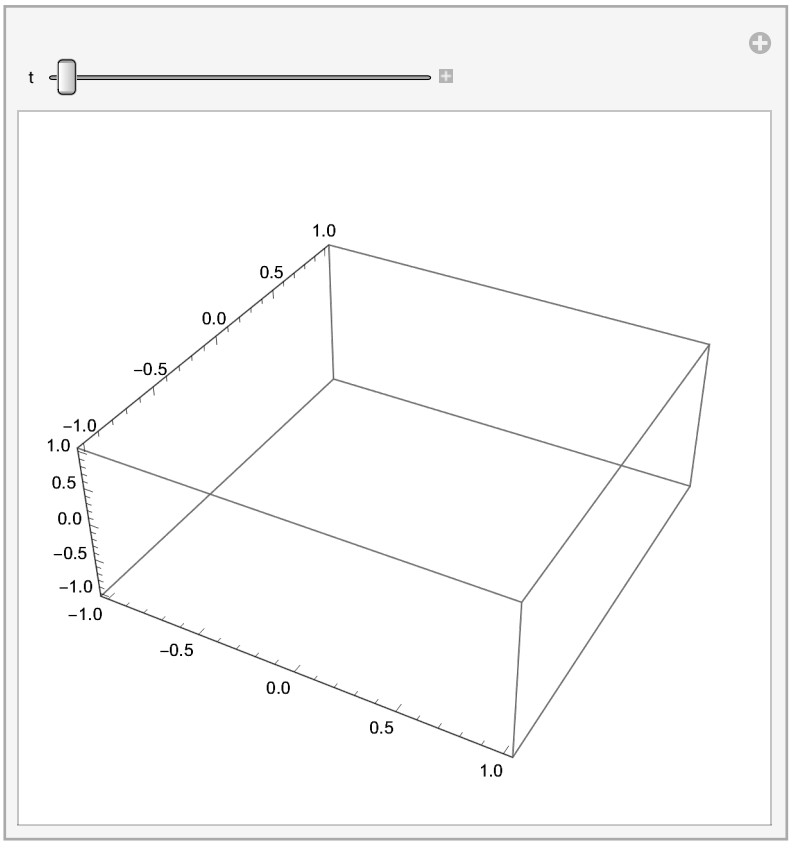
In[•]:=
```
ClearAll["Global`*"]
```
efface tout
```
viscosite = 0.09;
sol1 = First[
```
premier
```
  NDSolve[{D[u[t, x, y], t] == viscosite * (D[u[t, x, y], x, x] + D[u[t, x, y], y, y]) -
```
résous nu··· dérivée d dérivée d dérivée d
```
      u[t, x, y] (2 * D[u[t, x, y], x] - D[u[t, x, y], y]),
```
dérivée d dérivée d
```
    u[0, x, y] == Exp[-(x^2 + y^2)], u[t, -4, y] == u[t, 4, y],
```
exponentielle
```
    u[t, x, -4] == u[t, x, 4]}, u, {t, 0, 4}, {x, -4, 4}, {y, -4, 4}]]
Manipulate[Plot3D[u[t, x, y] /. sol1, {x, 0, 4}, {y, -4, 0}, PlotRange → All],
```
manipule tracé 3D zone de tracé tout
```
 {t, 0, 4, 1}]
```

Out[•]=
```
{u → InterpolatingFunction[
```

Domain: {{0., 4. }, {−4., 4. }, {−4., 4. }}
Output: scalar
```
]}
```

Les données ne sont pas dans le notebook. Stocker –les maintenant »

Out[•]=



••• ReplaceAll : {sol1 } is neither a list of replacement rules nor a valid dispatch table, and so cannot be used for replacing.

••• ReplaceAll : {sol1 } is neither a list of replacement rules nor a valid dispatch table, and so cannot be used for replacing.

••• ReplaceAll : {sol1 } is neither a list of replacement rules nor a valid dispatch table, and so cannot be used for replacing.

••• General : Further output of ReplaceAll::reps will be suppressed during this calculation.

Comme dans les exercices précédents, le lecteur peut modifier tel ou tel élément de ce modèle, par exemple réduire ou augmenter la viscosité.

Par ailleurs, la version 12.2 donne accès à de nombreux outils très puissants pour construire des modèles de simulation d'un système d'EDP. Ils seront présentés dans le notabook consacré ultérieurement à ce type de simulation.

Les micro-modèles de simulation avec Mathematica

Le concepteur du logiciel, Stephen Wolfram, était à l'origine un physicien très fortement investi dans la recherche sur les automates cellulaires. Cette approche est donc un des points fort du logiciel Mathematica. Comme pour les macro-modèles, nous consacrerons un notebook à cette famille de micro-modèles. Ce notebook n'est qu'une très brève introduction. Un automate cellulaire applique de façon itérative, pendant un laps de temps fixé, une règle ou fonction de transition à un ensemble de cellules, des lieux en géographie. Trois conditions très générales sont donc nécessaire : une règle liant une cellule aux cellules voisines, une configuration initiale, et la durée de simulation. Il existe des milliers de règles, et d'innombrables configurations, un réseau, des ensembles de points, un maillage, etc. L'aide à la fonction **Cellu-**

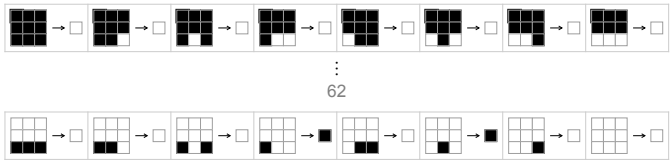
larAutomaton[] témoigne de cette gigantesque diversité. Les exemples présentés ci-dessous sont volontairement simples. Commençons par la fonction **RulePlot**[] qui permet au géographe de visualiser le fonctionnement d'un règle avant de l'utiliser.

Visualisons quelques règles avec la fonction RulePlot[]

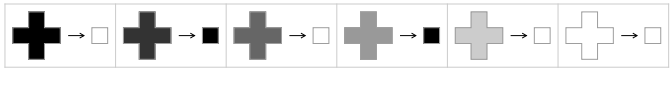
Les cinq lignes d'instructions ci-dessous affichent la règle 20 appliquée à un espace de dimension 2. La première ligne affiche la règle sans aucune précision. La ligne 2 indique qu'il s'agit toujours de la règle 20, mais pour un automate totalistique ou sommatif, donc lié à la somme du contenu de la cellule et des contenus des cellules voisines, avec un voisinage de Von Newmann. La ligne 3 retient la même règle, seul change le voisinage qui est celui de Moore. La ligne 4 affiche toujours la même règle, mais les options sont explicitées sous forme d'une association. Enfin, la ligne 5 affiche le résultat d'un ensemble de substitution, donc avec deux règles souples.

In[ ]:=
```
RulePlot[CellularAutomaton[{20, 2, {1, 1}}]]
RulePlot[CellularAutomaton[{20, {2, {{0, 1, 0}, {1, 1, 1}, {0, 1, 0}}}, {1, 1}}]]
RulePlot[CellularAutomaton[{20, {2, {{1, 1, 1}, {1, 1, 1}, {1, 1, 1}}}, {1, 1}}]]
RulePlot[CellularAutomaton[<|
    "OuterTotalisticCode" → 20, "Dimension" → 2, "Neighborhood" → 5|>]]
RulePlot[SubstitutionSystem[{0 → {{0, 0, 0}, {0, 0, 0}},
    1 → {{1, 0, 0}, {1, 1, 1}}}]]
```

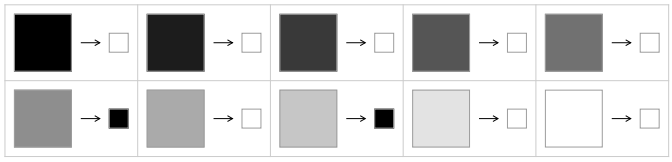
Out[ ]=



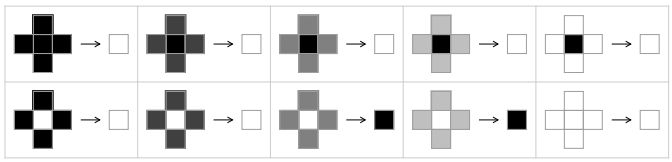
Out[ ]=



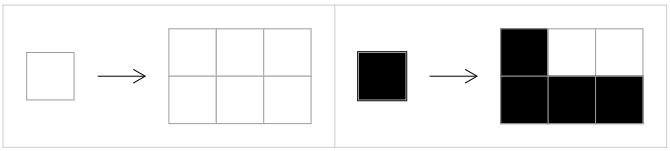
Out[ ]=



Out[ ]=



Out[ ]=



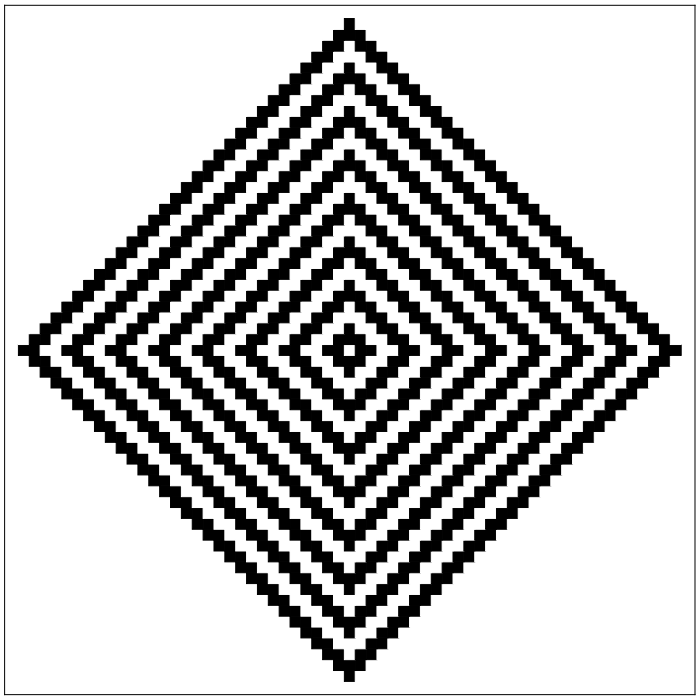
Visualiser quelques évolutions d'automates cellulaires

Pour simuler l'évolution d'un AC une seule instruction suffit, **CellularAutomaton**[*règle, initialisation, temps*]. Puis, les divers résultats sont affichés avec la fonction **ArrayPlot**[] **ou MatrixPlot**[]. Dans les exercices ci-dessous, dans les 3 premières visualisations, seule la configuration finale, après {{{30}}} itérations est affichée. Pour avoir toutes les configurations successives, nous les enregistrons d'abord dans le fichier *evol*, puis nous choisissons les configurations de départ et d'arrivées. Avec la fonction **Table**[], nous obtenons les configurations qui correspondent aux itérations 3, 4, et 5. Dans ces exercices, sauf le premier, la condition initiale, *initialisation*, est une matrice de 30 x 30 cellules, dont seule la cellule centrale est noire. Enfin, la dernière ligne calcule l'entropie à chaque étape de l'évolution et en donne une représentation graphique.

In[ ]:=
```
init = CenterArray[{30, 30}];
```
centre tableau
```
Print["Solution avec une condition initiale implicite {{{1}},0}"]
```
imprime
```
ArrayPlot[
```
tracé de tableau
```
 CellularAutomaton[<|"TotalisticCode" → 30, "Dimension" → 2, "Neighborhood" → 5|>,
```
automate cellulaire
```
  {{{1}}, 0}, {{{30}}}]]
Print["Solution avec une condition initiale explicite definie dans init "]
```
imprime
```
ArrayPlot[
```
tracé de tableau
```
 CellularAutomaton[<|"TotalisticCode" → 30,
```
automate cellulaire
```
   "Dimension" → 2, "Neighborhood" → 5|>, init, {{{30}}}]]
Print["Solution avec une autre regle une condition initiale explicite
```
imprime
```
   plusieurs sorties et calcul des entropies successives"]
ArrayPlot[CellularAutomaton[<|"Dimension" → 2, "GrowthCases" → {1, 2}|>,
```
tracé de tabl··· automate cellulaire
```
  init, {{{30}}}]]
evol = CellularAutomaton[<|"Dimension" → 2, "GrowthCases" → {1, 2}|>, init, 15];
```
automate cellulaire
```
{ArrayPlot[evol[[1]]], ArrayPlot[evol[[15]]]}
```
tracé de tableau tracé de tableau
```
Table[ArrayPlot[evol[[i]]], {i, 3, 5}]
```
table tracé de tableau
```
Print["représentation graphique des entropies successives"]
```
imprime
```
ListPlot[Table[Entropy[evol[[i]]], {i, 1, 15}]]
```
tracé de liste table entropie

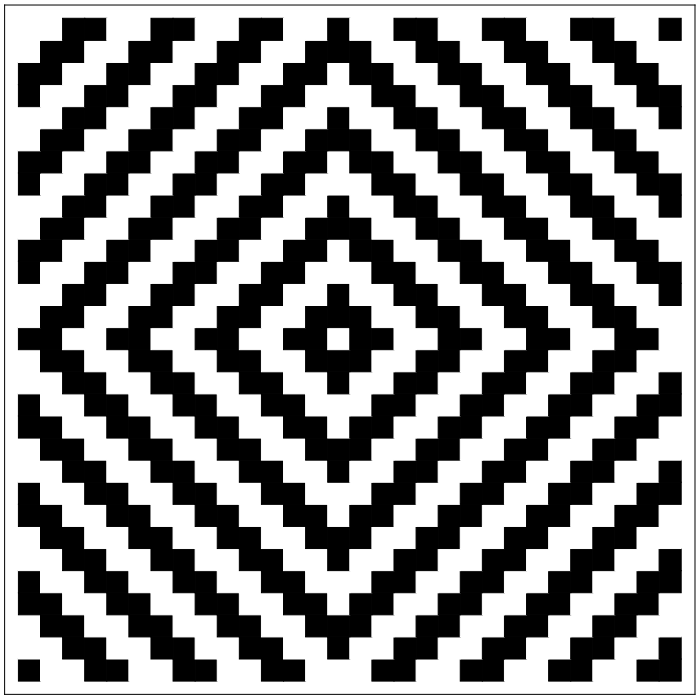
```
Solution avec une condition initiale implicite {{{1}},0}
```

Out[ ]=



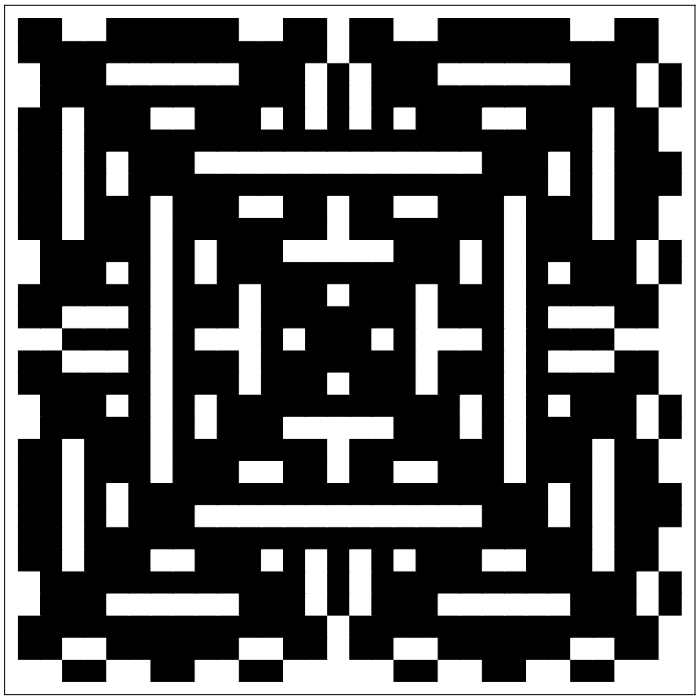
Solution avec une condition initiale explicite definie dans init

Out[ ]=

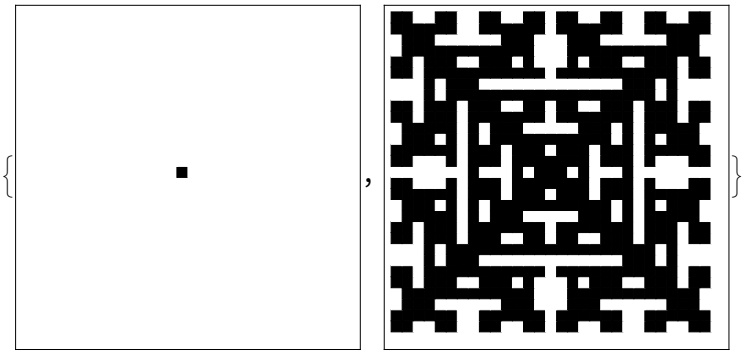


Solution avec une autre regle une condition initiale
explicite plusieurs sorties et calcul des entropies successives

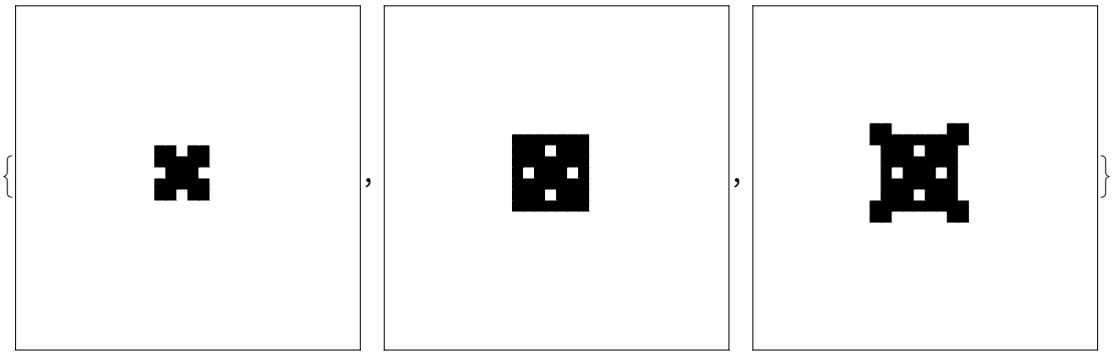
Out[ ]=



Out[ ]=

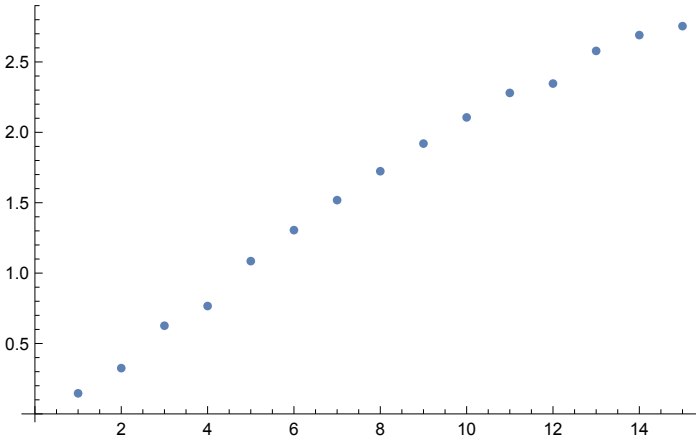


Out[ ]=



représentation graphique des entropies successives

Out[ ]=



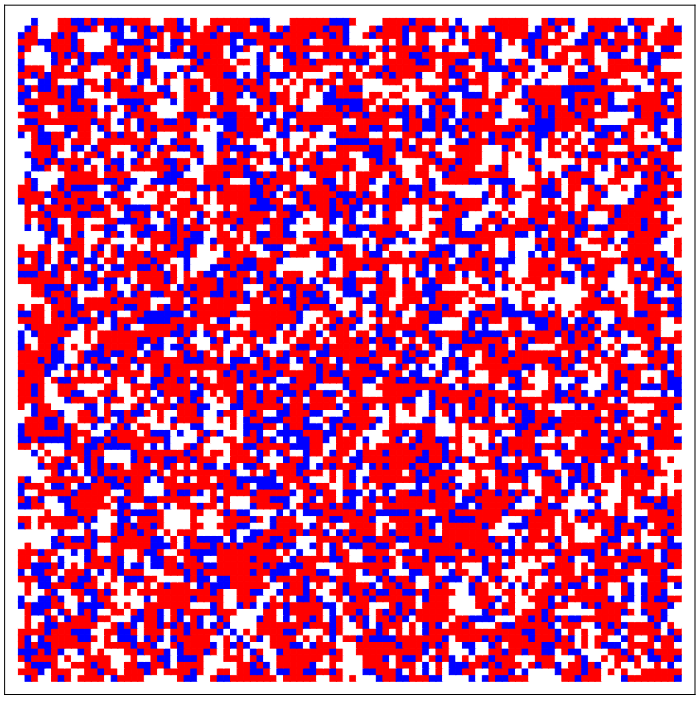
Dans une recherche il serait nécessaire de multiplier les analyses sur les résultats obtenus. Mais, dans cette introduction envisageons des AC plus complexes.

Des automates cellulaires plus complexes

Il est facile de considérer un AC avec, non pas uniquement 2 états, mais plusieurs états. Par exemple, avec 3 états (0,1, et 2), toujours en 2D, les lignes suivantes donnent la configuration à la cinquantième itération. La couleur blanche correspond à l'état 0.

In[ ]:=
```
ArrayPlot[CellularAutomaton[{679458, {3, 1}, {1, 1}},
 RandomInteger[1, {100, 100}], {{{50}}}], ColorRules → {1 → Red, 2 → Blue}]
```
tracé de tabl··· automate cellulaire
nombre entier aléatoire règles de couleur rouge bleu

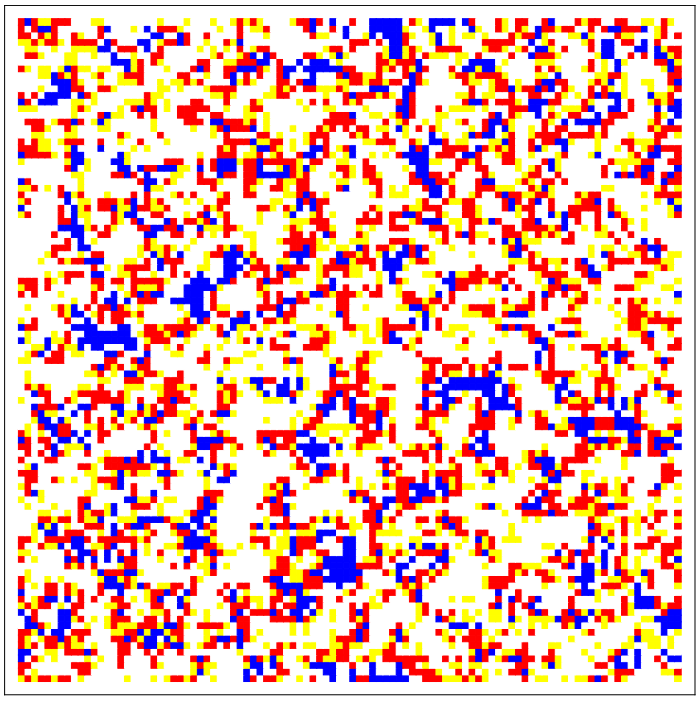
Out[ ]=



ou un AC avec 4 états :

In[ ]:=
```
ArrayPlot[CellularAutomaton[
 {12 679 458, {4, 1}, {1, 1}}, RandomInteger[1, {100, 100}], {{{50}}}],
 ColorRules → {1 → Red, 2 → Blue, 3 → Yellow}]
```

Out[ ]=

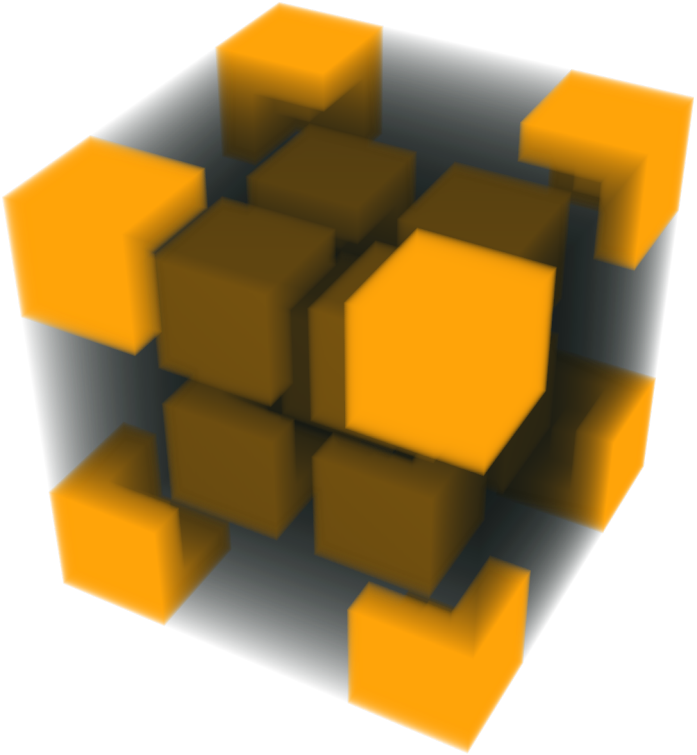


Dans ce cas, il faut se reporter à l'aide pour comprendre ces règles où les états sont indiqués par des couleurs. Il est aussi possible de travailler en trois dimensions :

```
In[ ]:= Image3D[CellularAutomaton[
          <|"GrowthCases" → {1}, "Dimension" → 3, "Neighborhood" → "Moore"|>,
          {{{{1}}}, 0}, {{{5}}}]]
```

image 3D automate cellulaire

Out[ ]=



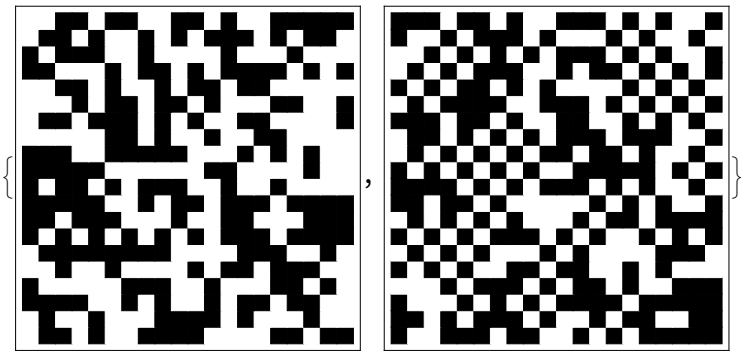
Mais généralement, les conditions initiales correspondent à une configuration plus compliquée d'états, et le chercheur peut construire une règle qui répond à sa problématique, puis l'appliquer avec un AC hexagonal. Dans l'exemple ci-dessous la condition initiale est une matrice de 0 et 1 tirés au sort, et le programme applique la règle originale suivante : un cellule passe de la valeur 0 à la valeur 1 si une ou trois cellules voisines ont la valeur 1; sa valeur est maintenue à 1 si elle a zéro, une, ou deux cellules voisines égales à 1, autrement elle prend la valeur 0. Le programme affiche la situation aux pas de temps 1 et 15.

```
In[ ]:= evol = CellularAutomaton[<|
          "Dimension" → 2, "GrowthSurvivalCases" → {{1, 3}, {0, 1, 2}},
          "Neighborhood" → "VonNeumann"|>, RandomInteger[1, {20, 20}], 15];
        {ArrayPlot[evol[[1]]], ArrayPlot[evol[[15]]]}
```

automate cellulaire nombre entier aléatoire

tracé de tableau tracé de tableau

Out[ ]=

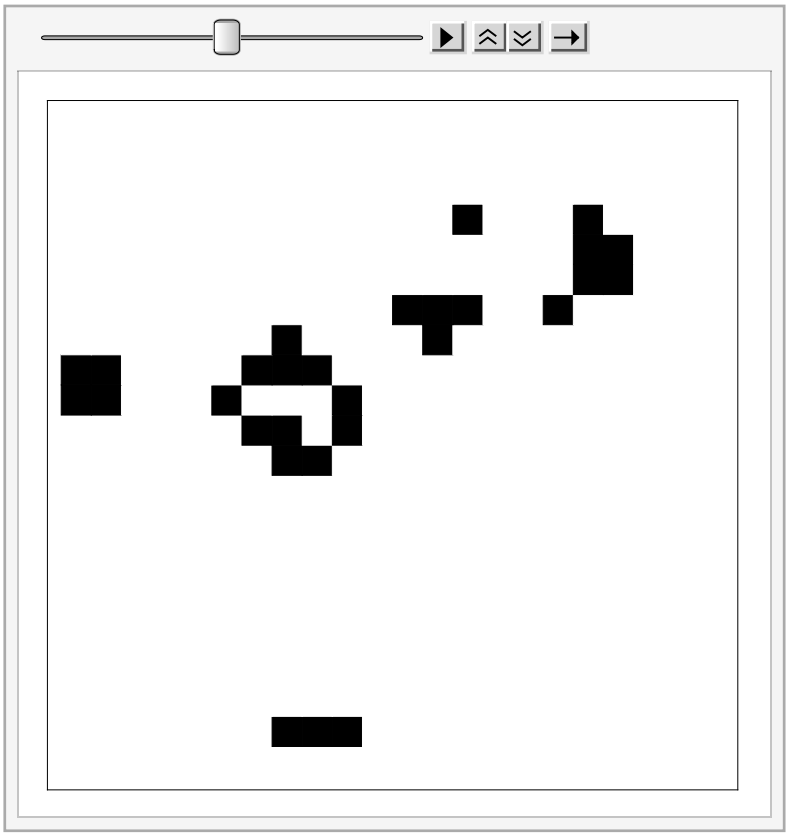


Et pour le plaisir voici le célèbre jeu de la vie animé,grâce à la fonction **ListAnimate**[]. RELANCER CETTE

CELLULE D'INSTRUCTION POUR UN FONCTIONNEMENT CORRECT. Et cliquer sur le signe II de la fenêtre pour arrêter l'animation.

In[ ]:=
```
ListAnimate[
 anime liste
  ArrayPlot /@ CellularAutomaton[{224, {2, {{2, 2, 2}, {2, 1, 2}, {2, 2, 2}}}, {1, 1}},
   tracé de tableau   automate cellulaire
    {RandomInteger[1, {9, 9}], 0}, 90]]
     nombre entier aléatoire
```

Out[ ]=



En parcourant les Demonstration Projects, le chercheur peut se rendre compte de la très grande variété des applications (percolation, diffusion d'épidémie entre villes, feux de forêts, diffusion dans un réseau, etc) que nous aborderons dans un notebook consacré à ce type de simulation.

Mais dans ce type de simulation, les cellules ne bougent pas. Elles peuvent seulement changer d'état. Pour prendre en compte les déplacements, quelle que soit leur nature, il est nécessaire de procéder à une simulation par SMA, qui nécessite un peu de programmation, ou de retenir un processus de chemin stochastique.

Simuler avec les processus stochastiques

Certains processus stochastique en 2D simulent le mouvement d'un objet, un oiseau dans le ciel, un citadin qui va à son travail, un touriste qui visite une cité balnéaire. Ces mouvements élémentaires, essentiels, sont de trois types : la diffusion ou déplacement sur de courtes distances, l'advection qui est un mouvement plus long, tel le vol d'un avion ou le transport par TGV, et enfin la turbulence, repérable dans les rivières ou l'atmosphère, mais aussi dans une foule prise de panique. Surtout, ces mouvements élémentaires se combinent et donnent une infinie variété à la "réalité" observée. Les professeurs d'université se

déplacent sur de courtes distances dans leur laboratoire, mais de temps en temps, ils courent le monde lors de congrès et colloques. Or, comme les AC, les processus stochastiques offrent une très grande souplesse pour prendre en compte ces contraintes du réel. Eux aussi méritent un long notebook. Dans cette introduction, nous présentons les grands traits de ces outils de simulation, dont certains furent déjà abordé dans le traitement des séries temporelles.

Les modèles simples de marche aléatoire et de mouvement brownien

Commençons par deux modèles simples qui servent généralement d'introduction pour comprendre ce type de simulation. La fonction **RandomWalk**[] simule un processus discret dans le temps et l'espace eux aussi discrets appelé marche au hasard ou promenade aléatoire. Ces pas ne sont pas corrélés et les mathématiciens parlent de propriété de Markov. À chaque pas est sélectionné de manière aléatoire la direction et la longueur du pas. Ci-dessous la même instruction pour simuler deux chemins aléatoires différents. Le premier avec des probabilités égales à 0.5, donc identiques dans tous les sens. Le second plus direct vers l'Est (probabilité = 0.55 et non pas 0.50). Et les résultats des deux simulations sont illustrées avec les fonctions **ListLinePlot**[] ou **Graphics**[].

In[•]:=
```
SeedRandom[1234];
```
amorçage aléatoire

```
data2d = RandomFunction[RandomWalkProcess[0.5], {0, 10^3}, 2]
```
fonction aléatoire processus de marche aléatoire

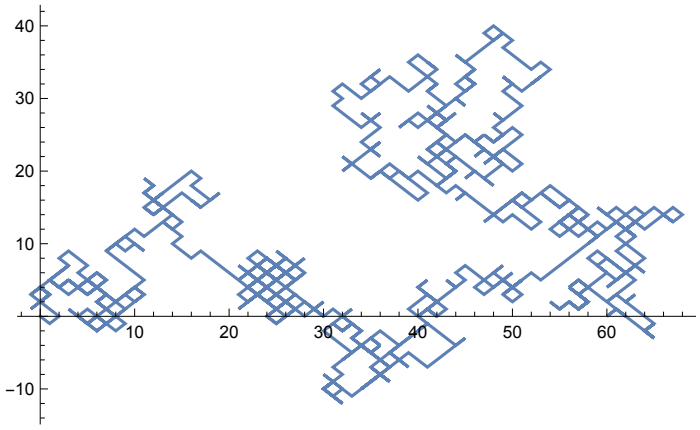
```
ListLinePlot[Transpose[data2d["ValueList"]]]
```
tracé de ligne de l·· transpose

```
data2d = RandomFunction[RandomWalkProcess[0.55], {0, 10^3}, 2]
```
fonction aléatoire processus de marche aléatoire

```
Graphics[Line[Transpose@data2d["ValueList"]], AspectRatio → Automatic]
```
graphique ligne transpose rapport d'aspect automatique

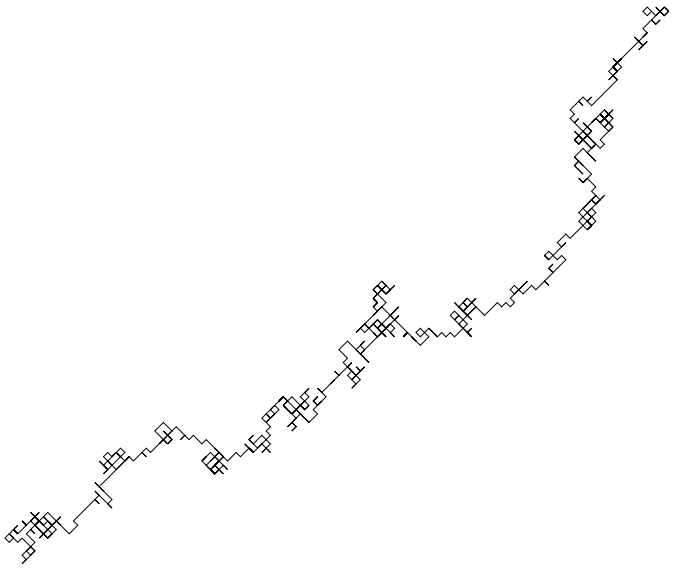
Out[•]= TemporalData[Time: 0 to 1000 Data points: 2002 Paths: 2]

Out[•]=



Out[•]= TemporalData[Time: 0 to 1000 Data points: 2002 Paths: 2]

Out[ ]=



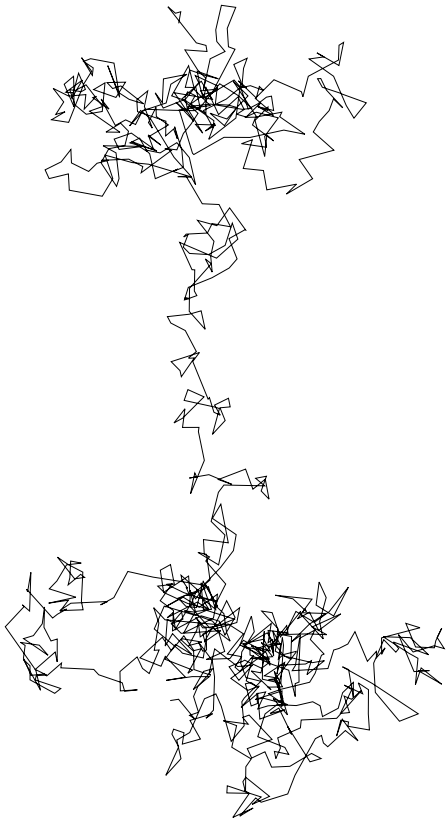
Le mouvement brownien ou processus de Wiener est l'équivalent du chemin aléatoire, mais avec un temps et un espace continu. Ce modèle est simulé avec la fonction **WienerProcess**[***a, b***]. Où ***a*** est la dérive et ***b*** la volatilité. Ci-dessous un premier exemple avec une dérive nulle et une volatilité égale à 1 :

In[ ]:=
```
SeedRandom[103];
```
amorçage aléatoire

```
sample = RandomFunction[WienerProcess[], {0, 1, .001}, 2]["ValueList"];
```
fonction aléatoire processus de Wiener

```
Graphics[Line[Transpose@sample]]
```
graphique ligne transpose

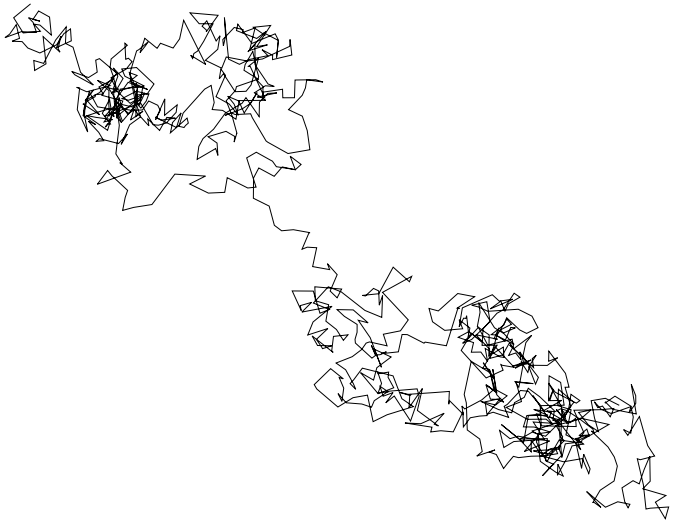
Out[ ]=



Puis en donnant d'autres valeurs à la dérive et à la volatilité on obtient un nouveau chemin :

```
In[•]:= sample = RandomFunction[WienerProcess[0.6, 0.5], {0, 1, .001}, 2]["ValueList"];
        fonction aléatoire   processus de Wiener

        Graphics[Line[Transpose@sample]]
        graphique   ligne   transpose
```



Out[•]=

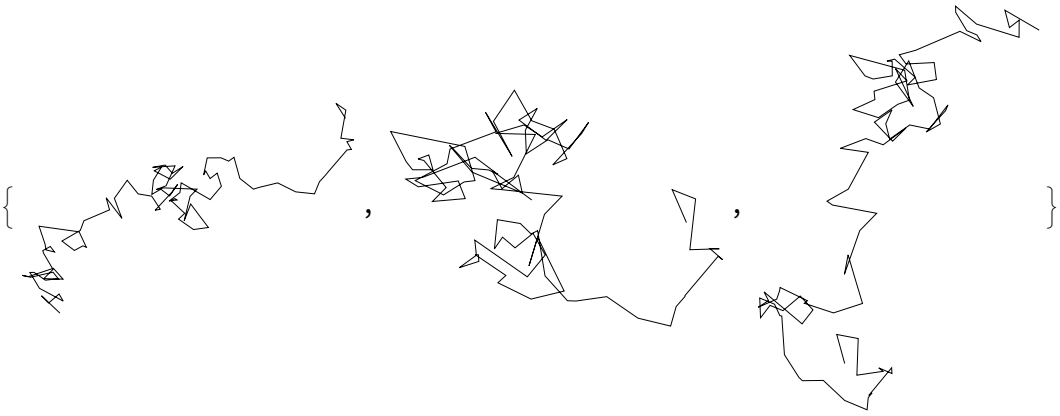
Enfin, il est possible de voir comment intervient le changement de la dérive avec le petit programme ci-dessous qui donne 3 valeurs à cette dérive (-2, 0, et 2) et la valeur 1 à la volatilité :

```
In[•]:= ClearAll["Global`*"]
        efface tout

        sample[μ_] := (SeedRandom[14];
                       amorçage aléatoire

          RandomFunction[WienerProcess[μ, 1], {0, 1, .01}, 2])
          fonction aléatoire   processus de Wiener

        Graphics[Line[Transpose[sample[#]["ValueList"]]]] & /@ {-2, 0, 2}
        graphique   ligne   transpose
```

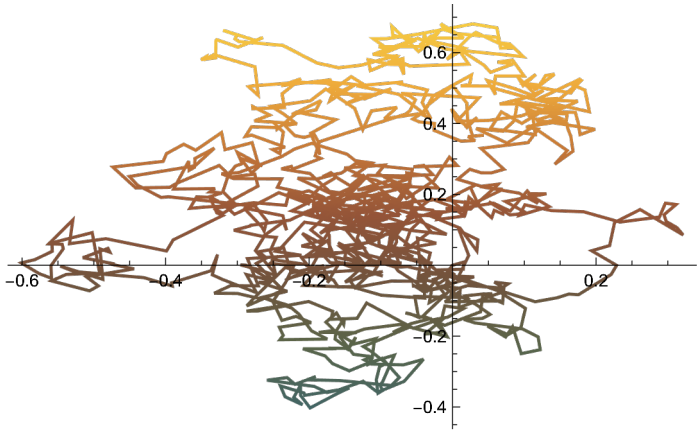


Out[•]=

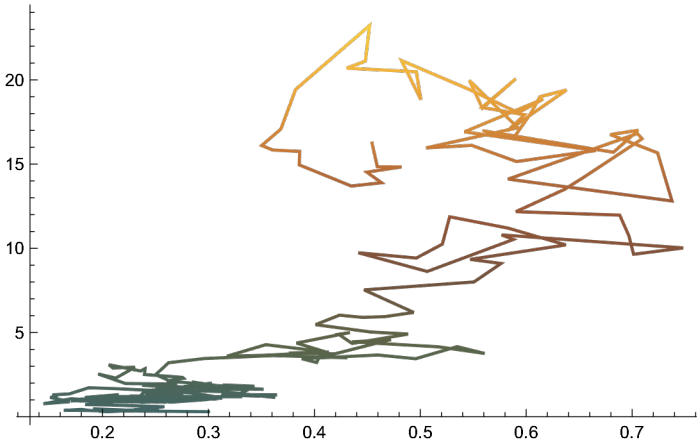
Cependant, pour mieux prendre en compte les contraintes de terrain, le chercheur dispose de processus mieux adapté. Les premiers complexifient le mouvement brownien. Ce sont les processus de pont brownien, qui encadre l'évolution entre deux valeurs, et de mouvement géométrique brownien, qui part d'une valeur initiale. Leur simulation s'effectue avec les fonctions **BrownianBridgeProcess**[] et **GeometricBrownianMotionProcess**[]. Le programme ci-dessous simule successivement les deux processus :

In[ ]:=
```
SeedRandom[104];
```
amorçage aléatoire
```
proc1 = BrownianBridgeProcess[];
```
processus de pont brownien
```
sample = RandomFunction[proc1, {0, 1, 0.001}, 2]["ValueList"];
```
fonction aléatoire
```
ListLinePlot[Transpose@sample, ColorFunction → "FallColors"]
```
tracé de ligne de l·· transpose fonction de couleur
```
proc2 = GeometricBrownianMotionProcess[1, 1, .3];
```
processus de mouvement brownien géométrique
```
SeedRandom[104];
```
amorçage aléatoire
```
sample = RandomFunction[proc2, {0, 3, 0.01}, 2]["ValueList"];
```
fonction aléatoire
```
ListLinePlot[Transpose@sample, ColorFunction → "FallColors"]
```
tracé de ligne de l·· transpose fonction de couleur

Out[ ]=



Out[ ]=

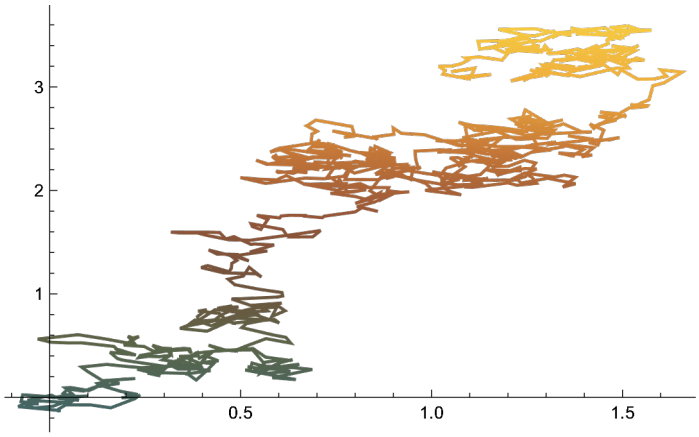


Simuler des processus fractals

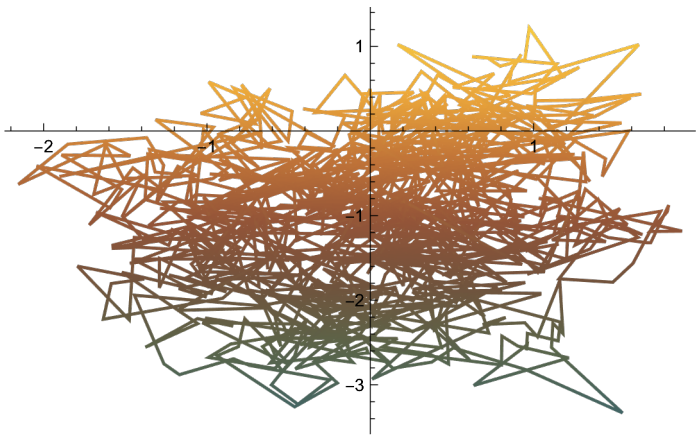
Un grand nombre d'organisations spatio-temporelles sont fractales. Le géographe, qui utilise Mathematica, dispose alors de deux fonctions pour simuler les processus fractals les plus fréquemment observés. La première, **FractionalBrownianMotionProcess[]**, généralise le processus brownien en utilisant comme paramètre la dérive, la volatilité auquel il ajoute le coefficient de Hurst, qui est employé dans le calcul de la dimension fractale. La seconde, **FractionalGaussianNoiseProcess[]**, simule un processus de bruit gaussien fractal et nécessite les trois mêmes paramètres. Notons que ces processus ont déjà été évoqués dans le traitement des chroniques.

In[•]:=
```
SeedRandom[153];
```
amorçage aléatoire
```
sample = RandomFunction[
```
fonction aléatoire
```
    FractionalBrownianMotionProcess[.45], {0, 1, .001}, 2]["ValueList"];
```
processus de mouvement brownien fractionnaire
```
ListLinePlot[Transpose@sample, ColorFunction → "FallColors"]
```
tracé de ligne de l·· transpose fonction de couleur
```
sample = RandomFunction[
```
fonction aléatoire
```
     FractionalGaussianNoiseProcess[.2], {0, 1, .001}, 2]["ValueList"];
```
processus de bruit gaussien fractionnaire
```
ListLinePlot[Transpose@sample, ColorFunction → "FallColors"]
```
tracé de ligne de l·· transpose fonction de couleur

Out[•]=



Out[•]=



Comme dans le cas du traitement de séries temporelles, quand le géographe dispose de données, il peut rechercher le processus sous-jacent, avec les fonctions **EstimatedProcess**[] et **FindProcessParameters**[].

Conclusion

Rappelons d'abord que pour valider ces modèles, le géographe doit disposer de données, pour comparer les résultats d'une simulation avec ces données recueillies. Par ailleurs, ce notebook est une simple introduction, dans laquelle l'espace sous-jacent est un espace plan, lisse sans diversité ni contrainte. Il conviendra de travailler sur des espaces plus "réalistes" dans les futurs notebook.

Bibliographie

Sur les EDP

Lionel Roques, 2013, *Modèles de réaction-diffusion pour l'écologie spatiale*, Èditions Quae.
R. st. Cantrell and Ch. Cosner, 2003 , *Spatial Ecology via Reacttion-Diffusion Equations*, Wiley.

Sur les AC

André Ourednik, 2005, *La géographie cellulaire*, Institut de Géographie, Lausanne.
Jean-Philippe Rennard, 2002, *Vie artificielle*, Vuibert informatique.

Sur les processus stochastiques

David O'Sullivan et G. L. W. Perry, 2013, *Spatial Simulation*, Wiley-Blackwell.