Intermède en prélude aux deux dernières parties.

Qui s'émouvrait qu'il puisse exister des problèmes mathématiques irrésolus à ce jour? Après tout la liste des problèmes est illimitée et s'il en est qui semble résister aux attaques répétées, on peut toujours se rassurer en prétendant que tout échec n'est sans doute que provisoire et qu'avec plus d'astuce ou de persévérance on devrait pouvoir venir à bout de tout énoncé récalcitrant.

C'est certainement vrai si le problème posé émarge à l'algèbre ou à la géométrie élémentaire : nous verrons ultérieurement que, dans ces domaines, toute conjecture est décidable. Par contre, la situation de l'arithmétique est beaucoup moins favorable et il se trouve que quantité de problèmes demeurent non résolus dans le cadre de l'arithmétique élémentaire. La conjecture relative aux nombres premiers jumeaux est l'un de ceux là : personne ne sait, à ce jour, si l'ensemble, P_{jum}, des couples de nombres premiers jumeaux est fini ou infini. Certes, tout le monde est prêt à parier sa tête qu'il est infini mais le fait demeure qu'aucune certitude n'est acquise dans ce domaine. Si cet ensemble est infini, le programme suivant, qui prend en entrée un entier positif quelconque, n=144 dans l'exemple, s'arrête toujours sur le premier couple de jumeaux immédiatement supérieur ou égal à lui :

```
n=144;k=0;While[Not[PrimeQ[n+k]∧PrimeQ[n+k+2]],k++];Print[{n+k,n+k+2}]

{149,151}
```

Par contre, si P_{jum} est fini, il existe un entier positif, n_{seuil}, au-delà duquel plus aucun couple de premiers jumeaux n'existe d'où on déduit que ce programme ne s'arrête pas pour tout $n > n_{seuil}$. Aussi étonnant que cela puisse paraître, personne ne sait actuellement si ce programme s'arrête pour tout n. Tester l'arrêt peut fonctionner pour certaines valeurs de n pour lesquelles l'arrêt se produit plus ou moins rapidement mais que faire quand on est confronté à un calcul qui s'éternise : faut-il persévérer ou abandonner les recherches? Nous verrons que l'approche Turingienne de la calculabilité affirme en substance qu'il n'existe pas de procédure effective qui soit capable de décider l'arrêt des programmes sur l'ensemble de leurs données.

On pourrait être tenté de se tourner vers les mathématiques théoriques et essayer de démontrer, dans un cadre formel acceptable, un théorème qui affirmerait que P_{jum} est fini ou qu'il est infini. On aurait alors la réponse à la question posée de l'arrêt du programme sans avoir eu à le faire tourner une seule fois. Hélas, nous verrons dans l'approche Gödelienne des systèmes formels que le succès n'est pas davantage assuré : il existe des propositions qui sont vraies, au sens qu'il n'existe pas de contre exemple pour venir les démentir, et qui ne sont cependant pas prouvables dans le cadre de l'arithmétique classique.

Entendons-nous bien : il est parfaitement possible que dans le cas précis de la conjecture des nombres premiers jumeaux, on soit passé à côté de la solution parce qu'on a pas développé les outils théoriques adéquats et qu'un jour viendra où cette lacune sera comblée. Il n'en sera pas moins vrai que le problème générique demeurera car comme on le verra, en arithmétique mais aussi en théorie des ensembles, l'indécidabilité est impossible à éradiquer en toute généralité.

Ces deux indécidabilités, Turingienne ou Gödelienne, se présentent comme les deux faces d'une même montagne à escalader. Leur étude fait l'objet des deux exposés qui suivent.

La Tétralogique.

II) L'universalité au sens de Turing.

Les limites calculatoires imposées par le constructivisme en mathématiques.

La physique ambitionne de prédire l'évolution des systèmes inanimés qui peuplent le monde sensible et elle compte utiliser les outils mathématiques à sa disposition pour y parvenir. Partant du principe qu'il n'y a pas de physique sans physiciens, que les physiciens c'est nous et que nous sommes "naturellement" limités dans nos rapports avec le continu, il semblerait que nous ayons à nous préoccuper de veiller au respect des limites que le constructivisme impose au calcul effectif des fonctions mathématiques.

Une fonction, f, de N dans N, reçoit en entrée un entier positif, n, et lui associe, en sortie, un entier, f(n). Lorsque f(n) est défini pour tout $n \in N$, on dit que la fonction est totale. Sinon elle n'est que partielle et son domaine est un sous-ensemble strict de N.

De façon informelle, est déclarée effectivement calculable toute fonction, f(n), qu'un être humain normalement doué pour le calcul est capable d'évaluer en suivant les instructions d'un algorithme qui s'arrête au terme d'une exécution finie. C'est donc une procédure effective de calcul qui est demandée. Cette simple exigence a pour conséquence qu'un grand nombre de fonctions ne peuvent être calculées pour une raison tellement banale qu'elle confine à l'évidence : l'ensemble des procédures effectives est dénombrable alors que l'ensemble des fonctions de N dans N ne l'est pas. Un simple argument de comptage révèle qu'il n'y aura jamais assez de programmes pour calculer toutes les fonctions de N dans N.

Si la non calculabilité ne reposait que sur cette simple observation, elle n'intriguerait personne. Chacun est prêt à comprendre que la plupart des fonctions, f(n), sont aléatoires et qu'il ne faut pas s'attendre à ce qu'il existe un programme qui exhibe une logique commune à toutes les valeurs qu'elles sont susceptibles de prendre pour tout n. Toutefois, s'il est vrai que les fonctions aléatoires fournissent un fort contingent de fonctions non calculables par programme, il serait faux de croire que les deux ensembles coïncident et c'est précisément cette subtilité qui fait l'intérêt de la théorie de la non calculabilité.

Evidemment l'argument de comptage s'applique a fortiori aux fonctions de R dans R puisque leur ensemble est de cardinal, $2^{2^{\aleph_0}}$. Un fort contingent de fonctions non calculables dans R est également composé de fonctions au comportement aléatoire, même les fonctions continues dont l'ensemble est encore de cardinal, $2^{\aleph_0}$, n'y échappent pas. D'une manière générale, le calcul d'un réel, π ou e, par exemple, ne se conçoit qu'au compte-gouttes : il faut qu'une procédure effective soit capable d'en égrener les chiffres significatifs. De plus, si cela se fait par approximations successives, il est essentiel que la procédure inclue un test sûr qui renseigne sur le nombre de chiffres exacts obtenus à tous les stades du calcul.

Aucune définition de la calculabilité ne peut éviter la référence anthropique d'où toute tentative de formalisation passe obligatoirement par l'adoption d'une hypothèse de travail. C'est précisément ce que fait la "Thèse de Church-Turing" qui prétend ni plus ni moins et de façon équivalente que toute fonction calculable par un humain :

- est programmable dans un langage mu-récursif,
- appartient à la classe des fonctions mu-récursives,
- est programmable sur une machine de Turing.

Aucune des versions de cette thèse ne se démontre, elles sont chacune un acte de foi basé sur des faits expérimentaux jamais démentis. Par contre, leur équivalence se démontre.

Les limites de la programmation : programmable $\not\Rightarrow$ effectivement calculable.

Si on analyse les méthodes que nous, êtres humains, utilisons pour calculer une fonction, f(n), de N dans N, on constate que nous assimilons les étapes individuelles du processus à autant d'instructions d'un programme immuable attaché à f. Seules les valeurs numériques traitées lors des calculs change selon l'instance, n, demandée. Une première façon d'aborder le problème de la calculabilité consiste précisément à s'interroger sur l'ensemble des instructions de base que doit posséder un langage informatique calculatoirement complet. La réponse qui vient immédiatement à l'esprit consiste à exiger que soient implémentées les instructions suivantes :

lectures/écritures (read/write), affectations en mémoire (x := …), additions & multiplications d'entiers, branchements (If…Then…Else), boucles (Do[…, {i,i1,i2}]).

Un langage qui utilise ces instructions est dit primitif récursif et d'emblée, une chose est sûre : tout programme correctement écrit dans un langage primitif récursif est effectif puisqu'on a l'assurance qu'il s'arrête en affichant la bonne réponse. Considérons, à présent, l'ensemble, F, des programmes primitifs récursifs, F_j, qui prennent en entrée un entier positif quelconque, i, puis s'arrêtent en affichant l'entier, $f_j(i)$. Montrons, par un argument diagonal, qu'il est impossible que cet ensemble soit calculatoirement complet, autrement dit qu'il contienne toutes les fonctions effectivement calculables.

Cet ensemble est à coup sûr dénombrable car tout programme est assimilable à un texte de longueur finie et nous savons que l'ensemble des chaînes de caractères de longueurs finies est dénombrable. De plus, il est récursivement énumérable dans l'ordre canonique, premièrement des longueurs croissantes et à longueurs égales dans l'ordre lexicographique. Voici la liste, notée, FC, de toutes les fonctions calculées dans cet ordre canonique :

F_1 : $f_1(1), f_1(2), f_1(3), f_1(4), f_1(5), f_1(6), \ldots$ (FC)
F_2 : $f_2(1), f_2(2), f_2(3), f_2(4), f_2(5), f_2(6), \ldots$
F_3 : $f_3(1), f_3(2), f_3(3), f_3(4), f_3(5), f_3(6), \ldots$
F_4 : $f_4(1), f_4(2), f_4(3), f_4(4), f_4(5), f_4(6), \ldots$
…

Montrons, par un argument diagonal, qu'il existe beaucoup de fonctions parfaitement calculables qui ne figurent nulle part dans cette liste, par exemple, la fonction, $g(i) = f_i(i) + 1$. Celle-ci est effectivement calculable : connaissant i, il suffit pour connaître, g(i), de lire la valeur de $f_i(i)$ dans le tableau FC et de lui ajouter 1. Cependant, g(i) est absente de ce tableau qui ne peut donc certainement pas prétendre les énumérer toutes. Dans cette construction de la fonction, g, on pourrait évidemment ajouter n'importe quel entier à $f_i(i)$ de sorte que le tableau FC apparaît largement incomplet.

Il existe quantité de variantes dans la construction de fonctions absentes du tableau FC. Construisons de toute pièce une autre fonction, g(i), comme suit :

$g(1) = f_1(1)+1$
$g(2) = \text{Max}[f_1(2), f_2(2)]+1$
$g(3) = \text{Max}[f_1(3), f_2(3), f_3(3)]+1$
…

Il est facile de voir que cette fonction ne se trouve nulle part dans le tableau, FC, car elle croît plus vite que toutes les fonctions qui le meublent. Pourtant la définition de la fonction, g(i), constitue en soi une procédure parfaitement effective de son calcul.

La conclusion que l'on tire, à ce stade, est que les programmes écrits dans un langage primitif récursif ne calculent pas toutes les fonctions calculables. Il faut donc élargir le cadre primitif récursif en élargissant l'ensemble des instructions autorisées.

En autorisant l'instruction supplémentaire, *While[test,…, i]*, le cas échéant en lieu et place de, *Do[…, {i,i_1,i_2}]*, le langage passe du statut primitif récursif au statut mu-récursif et la thèse de Church-Turing pose précisément que les fonctions calculables sont exactement celles qui sont programmables dans n'importe quel langage mu-récursif. Les langages habituels de programmation, C, Pascal, Mathematica, …, sont mu-récursifs et leur équivalence calculatoire entraîne qu'il est toujours possible de traduire n'importe quel programme d'un langage de ce type vers un autre. Les langages de programmation mu-récursifs sont encore appelés langage universels (au sens de Turing).

On pourrait s'étonner que le simple élargissement de l'instruction, *Do[…, {i,i_1,i_2}],* à, *While[test,…, i],* fasse une telle différence. Il y a lieu de se demander, en particulier, ce qui empêche de reproduire à l'identique le raisonnement diagonal précédent afin de prouver qu'il subsiste des fonctions calculables qui échappent à la programmation mu-récursive. L'ensemble des programmes écrits dans un langage mu-récursif reste dénombrable et même récursivement énumérable de sorte qu'il est encore possible de construire un tableau, FC', similaire à FC. La réponse est la suivante : le tableau FC' existe bien mais il n'y a plus la moindre assurance que toutes ses cases sont occupées par un entier. Certaines cases sont, au contraire, marquées d'un carré blanc qui signifie que l'élément correspondant ne peut recevoir de valeur numérique précise parce que son calcul semble ne jamais s'arrêter. C'est la conséquence possible de l'introduction de l'instruction, *While[test,…, i]*, dont l'exécution boucle éternellement dans tous les cas où le test n'est pas sûr parce qu'il n'est vérifié à aucun passage. Nous avons vu un exemple de ce type dans l'intermède préparatoire à cette section.

F_1 : $f_1(1)$, □ , $f_1(3)$, $f_1(4)$, $f_1(5)$, $f_1(6)$, … (FC')
F_2 : $f_2(1)$, $f_2(2)$, □ , □ , □ , $f_2(6)$, …
F_3 : □ , $f_3(2)$, □ , $f_3(4)$, □ , $f_3(6)$, …
F_4 : $f_4(1)$, $f_4(2)$, $f_4(3)$, $f_4(4)$, $f_4(5)$, $f_4(6)$, …
…

Ce sont les cases vides du tableau, FC', qui rendent l'argument diagonal inutilisable. Il suffit que certaines valeurs, $f_i(i)$, ne soient pas définies, $f_3(3)$, dans l'exemple pour que la fonction , $g(i) = f_i(i) + 1$, cesse d'être calculable. A ce stade, plus rien dans l'argumentation utilisée n'interdit à la liste FC' d'être calculatoirement complète. Certes, cela ne prouve pas qu'elle est réellement complète et c'est précisément la raison d'être de la thèse de Church que d'admettre qu'il en est bien ainsi jusqu'à preuve expérimentale du contraire.

Il est tout à fait possible que certaines lignes du tableau FC' soient complètes : elles correspondent aux fonctions totales encore appelées fonctions effectivement calculables pour toutes instances, n. D'autres lignes ne le sont pas : elles correspondent aux fonctions dites partielles ou semi calculables.

En résumé, l'ensemble des fonctions programmables dans un langage primitif récursif n'épuise pas les fonctions calculables : certaines fonctions exigent un langage mu-récursif au risque de ne pas être calculable pour toutes les instances de sa variable.

Il est utile, à ce stade, de préciser le vocabulaire en usage. Le mot "calculable" fait partie intégrante de l'énoncé de la thèse de Church-Turing. Affirmer sans nuance que l'ensemble des fonctions calculables coïncide avec celui des fonctions programmables dans un langage mu-récursif, implique que l'on utilise le mot calculable dans le sens étendu "susceptible d'être calculé" sans aucune promesse que le calcul aboutisse. Si par le mot "calculable" on veut signifier une fonction dont le calcul aboutit toujours, il est préférable de le préciser en utilisant l'adverbe "effectivement" ce qui élimine toute confusion. L'ensemble des fonctions effectivement calculables coïncide alors avec l'ensemble des fonctions mu-récursives programmables avec tests sûrs. Tous les auteurs n'adhèrent pas à cet usage des mots. Certains confondent calculable avec effectivement calculable et marquent la différence en appelant semi calculables les fonctions programmables dans un langage mu-récursif. Avec ces appellations, on a qu'une fonction non calculable n'est jamais totale, qu'une fonction semi calculable pourrait ne pas l'être et qu'une fonction (effectivement) calculable l'est toujours. A l'inverse, une fonction totale est toujours (effectivement) calculable.

Lorsqu'on parle de calculabilité effective, on évoque une possibilité théorique qui ne se soucie pas d'efficacité. Certains calculs sont hors de notre portée même avec l'aide d'ordinateurs puissants car la durée du calcul ou l'espace mémoire requis dépassent toute limite raisonnable. L'efficacité des algorithmes est étudiée dans les cours de théorie de la complexité mais elle ne nous intéresse pas ici, seul compte le fait que le calcul soit théoriquement possible sur une machine certaine de s'arrêter, même si c'est dans quinze milliards d'années.

Les fonctions aléatoires sont trivialement non calculables, d'ailleurs elles ne sont même pas programmables. En effet, tout programme qui tenterait d'en décrire une serait nécessairement de longueur infinie, ne pouvant faire mieux que recopier in extenso les valeurs, f(i), attachées à chaque valeur, i, de la variable indépendante. Evidement les programmes infinis ne sont pas autorisés.

Les fonctions non aléatoires qui ne sont pas effectivement calculables sont celles pour lesquelles il arrive que leur calcul entre dans une boucle infinie. L'absence d'arrêt du programme n'a rien à voir avec une programmation défectueuse, c'est une limitation fondamentale qui frappe les mathématiques constructives.

Informellement on a donc la double inclusion stricte suivante entre les ensembles de programmes mu-récursifs infinis, finis et finis qui s'arrêtent :

$$\{\ pgm^{\infty}\ \} \supset \{\ pgm^{finis}\ \} \supset \{\ pgm^{finis} \quad qui \quad s'arrêtent\ \},$$

qu'on traduit comme suit au niveau des fonctions calculées :

$$\{\ f^{N\to N}\ \} \supset \{\ f^{N\to N} \quad semi \quad calculables (= \mu programmables)\} \supset \{\ f^{N\to N} \quad effect.\ \ calculables\ \}$$

La présentation informelle qui précède flatte l'intuition mais elle appelle une formalisation qui fut l'œuvre de Kleene. C'est à ce stade que l'on comprendra l'appellation "mu-récursive" laissée en suspens.

Formalisation de la calculabilité selon Kleene.

Ce paragraphe paraphrase formellement celui qui précède. La formalisation de Kleene, qui date de 1930, a précédé l'approche par programmes à une époque où l'informatique n'existait pas encore. On appelle primitives récursives, les fonctions de base suivantes :

- la fonction, 0
- la fonction successeur, $S(n)=n+1$
- la fonction projection qui sélectionne une variable parmi k, $\pi_i^k(n_1,\cdots,n_k)=n_i$ (cette fonction un peu marginale ne sert qu'à prélever une variable particulière quand plusieurs variables sont présentes).

Sont également primitives récursives les fonctions obtenues à partir des fonctions de base par composition ou par récursion primitive descendante du type :

$$f(n,0)=g(n)$$
$$f(n,m)=h(n,m,f(n,m-1))$$

où les fonctions, g et h, sont elles mêmes primitives récursives. On note que la forme descendante de la récurrence garantit son arrêt. Les fonctions primitives récursives sont exactement celles qui sont calculées par les langages primitifs récursifs.

Toutes les fonctions d'usage courant en mathématiques sont primitives récursives. Contentons-nous d'épingler les récurrences descendantes qui définissent les fonctions primitives récursives, Somme et Produit :

$$Somme[n,0]=n \qquad Somme[n,m]=S[Somme[n,m-1]]$$
$$Produit[n,0]=0 \qquad Produit[n,m]=Somme[n,Produit[n,m-1]]$$

En programmation, on préfère généralement les récurrences ascendantes aux récurrences descendantes. Une condition suffisante d'arrêt est cette fois que la récurrence soit effectuée un nombre fini de fois. C'est très exactement ce que fait l'instruction, *Do[…, {i,i_1,i_2}]*, commune à tous les langages primitifs récursifs.

Nous avons vu dans la section précédente l'exemple non constructif d'une fonction effectivement calculable, g(i), qui croissait plus vite que n'importe quelle fonction primitive récursive. Pour comprendre ce que cela peut signifier, intéressons-nous à la croissance asymptotique caractéristique des fonctions primitives récursives.

Il existe une hiérarchie dans l'ensemble des croissances possibles des fonctions primitives récursives lorsque leur argument croît indéfiniment. Tout dépend, en gros, du nombre de compositions qui sont présentes dans leur définition. Cette hiérarchie est indicée par la valeur de k dans la définition suivante de l'hyper exponentielle de puissance, k, que nous livrons dans une notation "horizontale" due à Knuth :

$$a\uparrow^k(n)=a\uparrow^{k-1}(a\uparrow^k(n-1))$$
$$a\uparrow^0(n)=a\times n \qquad a\uparrow^k(1)=a \qquad (k\geq 0, n\geq 1)$$

En bas de la hiérarchie, on trouve les fonctions primitives récursives qui croissent linéairement comme, $a \uparrow^0 (n) = a \times n$, puis exponentiellement comme, $a \uparrow^1 (n) = a^n$. Les récurrences qui les définissent ne font intervenir aucune composition, elles s'écrivent, respectivement, $f(n) = a + f(n-1)$ et $f(n) = af(n-1)$. Ensuite, on trouve les fonctions qui croissent comme, $a \uparrow^2 (n) = \underbrace{a^{a^{\cdot^{\cdot^{a}}}}}_{n \ fois}$, où les exponentiations sont associatives par la droite. Elles obéissent à la récurrence, $f(n) = a^{f(n-1)}$, qui comporte une composition, du type, $f(n) = h(f(n-1))$. Si on poursuit l'imbrication des compositions, on définit des fonctions primitives récursives qui croissent de plus en plus vite comme des tours d'exponentielles.

On a longtemps pensé que l'ensemble des fonctions primitives récursives coïncidait avec l'ensemble des fonctions calculables mais ce n'est pas exact. Certes, la forme descendante de la récurrence primitive apparaît clairement comme une condition suffisante d'arrêt mais le fait est qu'elle n'est pas nécessaire.

Puisque les fonctions primitives récursives sont exactement celles qui sont calculées par les langages primitifs récursifs, il doit exister des fonctions effectivement calculables qui ne sont pas primitives récursives. Le moment est venu d'en exhiber une.

Historiquement, le premier exemple fut l'œuvre de Sudan. Un exemple plus simple fut trouvé par Ackermann qui fut simplifié par Peter. Considérons la fonction suivante :

$$A[0,n] = n+1$$
$$A[m,0] = A[m-1,1] \qquad (m > 0)$$
$$A[m,n] = A[m-1,A[m,n-1]] \qquad (m > 0, n > 0)$$

Cette fonction respecte le schéma primitif récursif et à ce titre elle est effectivement calculable. Il est d'ailleurs très facile d'écrire un programme qui la calcule quels que soient m et n, peu importe que la durée de ce calcul excède tout délai raisonnable pour des valeurs des variables même pas élevées. On trouve, par exemple, les valeurs suivantes de A[m,n] aux petites valeurs de m et de n :

m\n	0	1	2	3	4	…
0	1	2	3	4	5	
1	2	3	4	5	6	
2	3	5	7	9	11	
3	5	13	29	61	125	
4	13	65533	2^{65536}-3	A[3, 2^{65536}-3]	A[3,A[4,3]]	
…						

Ce tableau donne une idée de la croissance vertigineuse de la fonction, A[m,n]. A[4,n] croît de façon hyper exponentielle en fonction de n et A[4,2] comporte déjà 19829 chiffres décimaux. La situation ne fait qu'empirer pour les valeurs de n supérieures à 2. Lorsque m>2, il est possible de proposer une forme compacte pour les valeurs de cette fonction qui utilise la notation de Knuth :

$A[m,n] = 2 \uparrow^{(m-2)} (n+3) - 3$.

Voici le détail des premières lignes du tableau correspondant aux valeurs de m>0 :

$A[1,n] = 2 + (n+3) - 3 = n + 2$
$A[2,n] = 2 \times (n+3) - 3 = 2n + 3$
$A[3,n] = 2 \uparrow (n+3) - 3 = 2^{n+3} - 3$
$A[4,n] = 2 \uparrow\uparrow (n+3) - 3 = 2 \uparrow (2 \uparrow (2 \uparrow (\cdots (2 \uparrow) \cdots)) - 3$
$A[5,n] = 2 \uparrow\uparrow\uparrow (n+3) - 3$
…

Quel que soit m fixé, la fonction, A[m,n], est primitive récursive. Par contre, la fonction diagonale, A[n,n], cesse de l'être : elle croît plus vite que toute fonction primitive récursive et il n'y a plus aucun moyen de la définir par une boucle, $Do[\cdots,\{i,1,N\}]$, présentant une limite fixe, N, connue d'avance.

Puisque les fonctions primitives récursives ne livrent qu'une partie des fonctions effectivement calculables, il y a lieu de s'interroger sur les possibilités d'élargir leur cadre. Vu le contenu de la section précédente, il est naturel de penser que cet élargissement doit se faire en direction des fonctions, dites mu-récursives, qui sont calculées par les programmes mu-récursifs. On procède comme suit.

On peut montrer les fonctions primitives récursives acceptent que figure, dans leur définition, la minimisation bornée,

$\mu\iota v \;\; i \le m : q(n,i)$,

soit la plus petite valeur de i (≤m, fixé d'avance) telle que le prédicat, q, soit vrai (0 si q n'est jamais vrai). La minimisation bornée obéit, en effet, à une récurrence primitive descendante, que nous ne reproduisons pas car elle est fastidieuse à écrire.

Les fonctions mu-récursives élargissent le cadre des fonctions primitives récursives en autorisant la minimisation non bornée,

$\mu\iota v \;\; i : q(n,i)$

soit la plus petite valeur de i telle que le prédicat q(n, i) soit vrai (0 s'il n'est jamais vrai). Ce faisant, on introduit évidemment le risque que la recherche de i ne s'arrête jamais donc que certaines fonctions ne soient pas effectivement calculables pour certaines valeurs de leur(s) argument(s). On retrouve l'argument de la section précédente qui revient à accepter les boucles, *While[test,…, i]*, dans l'écriture des programmes mu-récursifs.

En résumé, cette approche fournit une version alternative de la thèse de Church-Turing, qui assimile les fonctions (semi) calculables aux fonctions mu-récursives et les fonctions (effectivement) calculables aux fonctions mu-récursives sur prédicats sûrs.

Il n'existe aucune procédure effective qui permette de décider la sûreté d'un prédicat. Nous ne prouvons pas cette affirmation à ce stade de l'exposé mais une preuve équivalente

sera fournie lors de la discussion de l'arrêt des machines de Turing. Nous nous contentons de la rendre intuitivement plausible grâce aux exemples suivants.

Il existe des prédicats dont la sûreté ne fait aucun doute, par exemple celui-ci qui part à la recherche du plus petit entier positif qui n'est pas somme de trois carrés parfaits :

$$\mu\iota v \quad i \quad : \quad \neg\exists x,y,z : (i = x^2 + y^2 + z^2) \wedge (x \geq 0) \wedge (y \geq 0) \wedge (z \geq 0).$$

On voit qu'une recherche exhaustive, correctement programmée, s'arrête à la valeur, i = 6.

Certains prédicats ne sont pas sûrs pour des raisons évidentes, par exemple :

$$\mu\iota v \quad i \quad : \quad \exists m,n : 2i + 1 = 2m + 2n + 2$$

Dans ce cas, la recherche ne s'arrêtera jamais car aucun entier impair n'est la somme de deux entiers pairs. Un algorithme qui utiliserait un tel prédicat serait évidemment fautif comme le sont tous les programmes qui bouclent par négligence de leur auteur.

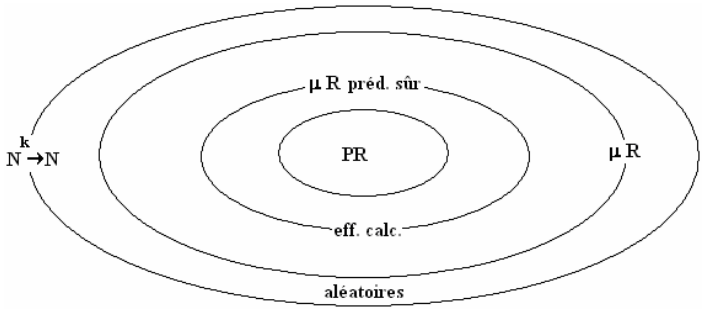
Mais il est des situations plus embarrassantes qui résultent du fait que le statut de certains prédicats n'est pas connu, tel celui-ci, dû à Goldbach, qui cherche le plus petit entier pair qui n'est pas la somme de deux nombres premiers :

$$\mu\iota v \quad i \quad : \quad \neg\exists x,y : (2i = x + y) \wedge PrimeQ[x] \wedge PrimeQ[y]$$

La conjecture de Goldbach n'est pas démontrée à ce jour et son statut reste incertain même si les présomptions penchent en faveur d'un prédicat non sûr.

En résumé, on a le schéma d'inclusions strictes suivantes :

$$\{ fonct. \} \supset \{ fonct.\ semi\ \ calculables = \mu R \} \supset$$
$$\{ fonct.\ eff.\ calculables = \mu R\ \ préd.\ sûrs \} \supset \{ fonct.\ Prim.\ Réc. \}$$



La machine de Turing.

Les mathématiciens des années 1930 se sont beaucoup investis pour trouver une machine qui serait capable d'automatiser le calcul des fonctions et la reconnaissance des langages. Ils ont progressivement construit des automates de plus en plus puissants, dans cet ordre : l'automate fini déterministe (AFD), l'automate à une pile et l'automate à bornes linéaires. Aucun ne répond cependant complètement à la question posée : ils s'avèrent impuissants devant certaines classes de fonctions ou de langages qu'un humain traite effectivement.

La solution, définitive si on accepte la thèse de Church-Turing, a été trouvée par Turing. C'est en cherchant à mimer sur le papier l'ensemble des opérations que nous effectuons mentalement lorsque nous calculons que Turing a découvert le principe de la machine qui porte son nom. Cette machine de Turing, en abrégé, MT, a été à la base de la révolution informatique. En possibilité de calcul, la MT égale l'ordinateur moderne. Le principe de son fonctionnement n'a jamais été et ne sera vraisemblablement jamais dépassé, le seul progrès attendu ne concernant que l'amélioration des performances en terme de facilité de programmation et de rapidité d'exécution.

Cette machine est constituée des éléments suivants :

- Un ruban suffisamment long, constitué d'un nombre théoriquement illimité de cases qui sont occupées par un caractère prélevé dans un alphabet donné, qui en comprend K, (a, b, c, …, 0, 1, …, et #), un caractère par case. Le caractère, #, tient lieu de « blanc ».

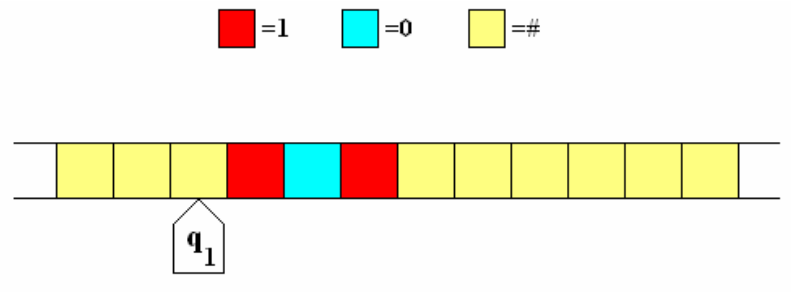
- Un processeur qui mémorise l'état interne de la machine parmi s états possibles, $\{q_1, q_2, q_3, \ldots, q_s\}$. Un de ces états, conventionnellement q_1, est l'état initial. Un état (éventuellement plusieurs), noté q_H, est un état d'arrêt. La machine n'interrompt son calcul que lorsque cet état est atteint.

- Une tête mobile de lecture-écriture, qui scanne le ruban, une case à la fois. Elle doit être capable de lire le contenu d'une case et, en fonction de son état interne, de le modifier. La lecture, l'écriture et le déplacement de la tête se font en accord avec le déroulement d'un programme qui commande l'évolution de la MT au départ de l'état initial. Ce programme consiste en une table comportant sK instructions toutes de la forme, $(q_i, \alpha) \rightarrow (q_j, \beta, G \quad ou \quad D)$, que l'on interprète comme suit.

Lorsque la MT se trouve dans l'état interne, q_i, et que sa tête lit le symbole, α, dans la case qui se trouve en face d'elle, elle bascule dans l'état interne, q_j, elle remplace α par β et, enfin, elle se déplace d'une case vers la gauche (G) ou vers le droite (D). L'opération est réitérée jusqu'à ce que le programme impose à la MT de basculer dans un état d'arrêt, q_H. Rien n'exclut que la machine n'atteigne jamais un tel état auquel cas on dit qu'elle boucle. Il existe plusieurs formes de bouclage : soit la tête se déplace toujours plus loin dans une direction sans jamais revenir en arrière, soit elle entre dans un cycle périodique soit elle fait d'incessants va-et-vient apériodiques sans jamais croiser d'instruction d'arrêt.

Il est fréquent qu'une MT ait à traiter un ensemble de données. Dans ce cas, on inscrit préalablement ces données sur le ruban et c'est sur ce même ruban qu'on récupère les résultats

in fine. Le ruban sert également aux calculs intermédiaires. Pour fixer les idées, voici une MT très rudimentaire qui double l'entier binaire, 101=5, initialement inscrit sur le ruban :



Les instructions de son programme interne s'écrivent en respectant le schéma,

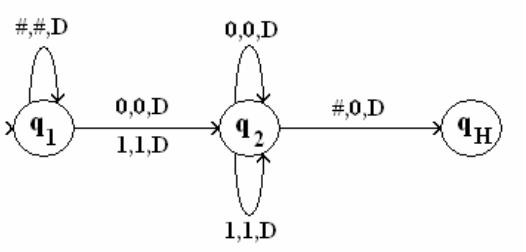
(ancien état, caractère lu) $\Rightarrow$ *(nouvel état, caractère écrit, déplacement):*

$(q_1, \#) \rightarrow (q_1, \#, D)$ $(q_1, 1) \rightarrow (q_2, 1, D)$ $(q_1, 0) \rightarrow (q_2, 0, D)$

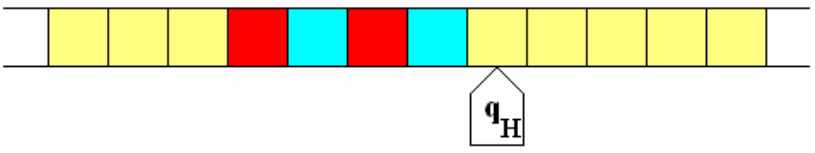
$(q_2, \#) \rightarrow (q_H, 0, D)$ $(q_2, 1) \rightarrow (q_2, 1, D)$ $(q_2, 0) \rightarrow (q_2, 0, D)$

On observe que $(q_2, \#)$ déclenche l'arrêt, H, après exécution de 0, D.

Il existe une représentation graphique assez suggestive de ce programme :



En fin d'exécution, le résultat du calcul s'affiche sur la bande de lecture comme suit :



Ce n'est qu'une question de convention de déterminer comment les données et les résultats doivent être positionnés par rapport à la tête de lecture. Il est en effet très facile de modifier la table d'instructions pour que l'encodage des données et/ou le décodage des résultats respectent une disposition différente.

Les variantes de machines de Turing.

Le modèle théorique de la machine de Turing qui vient d'être décrit est le plus répandu. Il accepte des variantes inessentielles qui n'altèrent pas son pouvoir calculatoire. On peut toujours réencoder une MT à K caractères et s états sur la base de minimum deux caractères (resp. états) au prix d'une démultiplication du nombre des états (resp. du nombre des caractères). Cet exercice de style ne présente cependant aucun intérêt pratique vu qu'en informatique on fait tout le contraire : on démultiplie le nombre des caractères (niveau software) et des états (niveau hardware) afin d'améliorer la lisibilité des programmes.

Il est toujours possible de réécrire le programme d'une MT de telle manière que les déplacements de la tête de lecture ne dépendent pas du caractère lu. Il est par contre impossible de rendre ces déplacements indépendants de l'état interne atteint par la machine à cet instant sans que cela entraîne une perte de puissance calculatoire.

On facilite la programmation d'une MT en démultipliant le nombre des bandes de lecture-écriture correctement synchronisées mais ce détail est sans importance puisque personne ne passera jamais son temps à programmer réellement une MT.

En s'en tenant à une seule bande, il est possible, sans perte de puissance calculatoire, de la considérer comme semi infinie. Cette astuce permet d'omettre l'instruction d'arrêt au niveau software pour la reporter au niveau hardware : il suffit de considérer que la MT s'arrête spontanément lorsqu'elle est sollicitée par son programme à déborder l'extrémité fixe de la bande. Si on veut poursuivre le calcul, on peut toujours ajouter une case au-delà de l'extrémité fixe et relancer la MT à partir de l'état précédemment atteint.

Il existe beaucoup de façons de programmer l'arrêt d'une MT qui ne recourent pas à une instruction, q_H, explicitée dans le programme. D'une manière générale, on peut toujours s'arranger pour que l'arrêt survienne lorsqu'une condition vient à être remplie qui a fait l'objet d'une convention prédéfinie entre le système qui calcule et l'observateur. Un ordinateur qui effectue un calcul ne s'arrête jamais au sens strict du terme, il continue son existence propre au travers d'une multitude de tâches dont l'observateur ne veut simplement rien savoir. De même un professeur qui termine un calcul devant ses élèves ne cesse en réalité pas vraiment de "calculer" et c'est par une convention quelconque qu'il les informe que le calcul est réellement achevé, peut-être en se tournant vers eux et en posant la craie.

Certaines variantes de MT sont, par contre, interdites sous peine d'en altérer le pouvoir calculatoire. Toute instruction doit impérativement comporter un déplacement de la tête de lecture vers la gauche ou vers la droite. Autrement dit, la tête ne peut jamais rester immobile. De même, il est exclu que le mouvement de la tête ne s'effectue que dans un seul sens : une MT qui serait dans ce cas ne dépasserait jamais la puissance calculatoire d'un automate fini déterministe. D'ailleurs, un automate fini déterministe peut encore être assimilé à une MT qui travaillerait sur une bande limitée.

La notation de Wolfram des machines de Turing.

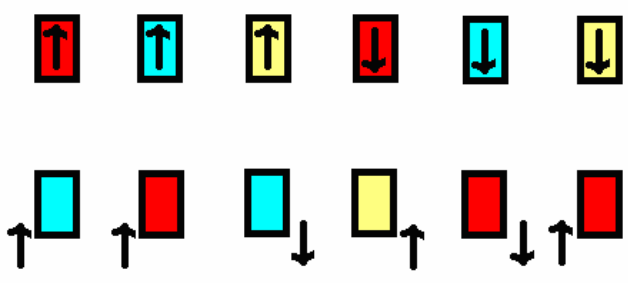
Wolfram a mis au point un schéma de numérotation des MT basé sur des codages immuables. Les s états sont numérotés de 1 à s, les K caractères sont numérotés de 0 à K-1 et les mouvements de la tête sont codés par G = -1 et D = +1. Des codages de 0 à s-1 pour les états et 0 et 1 pour les mouvements de la tête eussent sans doute été plus naturels.

Sans tenir compte des positions possibles pour la ou les instructions d'arrêt, il existe $(2sK)^{sK}$ MT distinctes travaillant sur s états et K caractères (s, K ≥ 2). Par exemple, on peut construire $12^6 = 2985984$ MT élémentaires à trois caractères et deux états.

Wolfram a proposé d'ordonner la table d'instructions de toute machine de Turing possédant s états et K caractères dans un ordre canonique qui considère en priorité les valeurs de l'indice d'état dans l'ordre ascendant puis, subsidiairement, les valeurs de l'indice des caractères dans l'ordre descendant. Une représentation graphique aide à suivre la méthode : elle consiste à représenter les caractères alphabétiques par des couleurs et les états possibles par des signets. Dans l'exemple suivant, les états sont représentés par des flèches,

↑ (q_1=1) et ↓ (q_2=2),

et les caractères sont représentés par des couleurs : ■ = 2, ■ = 1 et ■ = 0. Il convient d'y ajouter les conventions relatives à la progression de la tête, G = -1 et D = 1.



596540

Dans ce diagramme d'état, l'ordre qui prévaut à la première ligne est l'ordre canonique immuable. La deuxième ligne livre la table d'instructions de la machine de Turing en précisant pour chaque état et pour chaque caractère lu le nouvel état, le caractère écrit et le déplacement de la tête. Il y a bien $(2sK)^{sK}$ dispositions possibles.

Les instructions peuvent être traduites, comme suit, dans le format standard, $(q_i,\alpha)\rightarrow(q_j,\beta,G \quad ou \quad D)$:

```
{1,2}→{1,1,-1}    {1,1}→{1,2,-1}    {1,0}→{2,1,1}
{2,2}→{1,0,1}     {2,1}→{2,2,1}     {2,0}→{1,2,-1}    (596540)
```

Puisque l'ordre des membres de gauche est imposé, la seule information pertinente contenue dans cette table est la suite concaténée des seconds membres soit, dans cet ordre :

```
liste_instr = {1,1,-1,1,2,-1,2,1,1,1,0,1,2,2,1,1,2,-1}
```

Toute machine de Turing à deux états et trois caractères est donc représentée par une suite concaténée de six triplets. En soustrayant 1 du premier élément de chaque triplet (comme si les états étaient numérotés de 0 à s-1) et en faisant subir la transformation, $x\rightarrow(1+x)/2$ au troisième élément de chaque triplet (comme si les mouvements de la tête étaient codés par 0 et 1), cette liste se transforme comme suit :

```
liste_instr_modif = {0,1,0,0,2,0,1,1,1,0,0,1,1,2,1,0,2,0}.
```

Qu'a-t-on gagné au travers de cette manœuvre? Tout simplement que cette liste peut être considérée comme la liste des chiffres d'un nombre en base mixte {s,K,2} qui est le numéro d'ordre de cette machine de Turing. Les 2985984 MT à deux états et trois couleurs, rangées dans l'ordre d'indice croissant, possèdent les représentations modifiées suivantes :

```
MT0 = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0}
MT1 = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1}
MT2 = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0}
MT3 = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1}
MT4 = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2,0}
MT5 = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2,1}
MT6 = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0}
...
MT596540 = {0,1,0,0,2,0,1,1,1,0,0,1,1,2,1,0,2,0}
...
MT2985983 = {2,2,1,2,2,2,1,2,2,1,2,2,1,2,2,1,2,2,1}
```

L'instruction Mathematica suivante livre la table d'instructions (non modifiée) à partir du numéro, n, de la MT, et des nombres, s, d'états et, K, de caractères :

```
With[{s=2,K=3,n=596440},Flatten[MapIndexed[{1,-1}#2+{0,K}→{1,1,2}
Mod[Quotient[#1,{2K,2,1}],{s,K,2}]+{1,0,-1}&,
Partition[IntegerDigits[n,2sK,sK],K],{2}]]]
```

```
{1,2}→{1,1,-1},{1,1}→{1,2,-1},{1,0}→{2,1,1},
{2,2}→{1,0,1},{2,1}→{2,2,1},{2,0}→{1,2,-1}
```

On trouve le numéro de la MT dont on donne la table d'instructions comme suit :

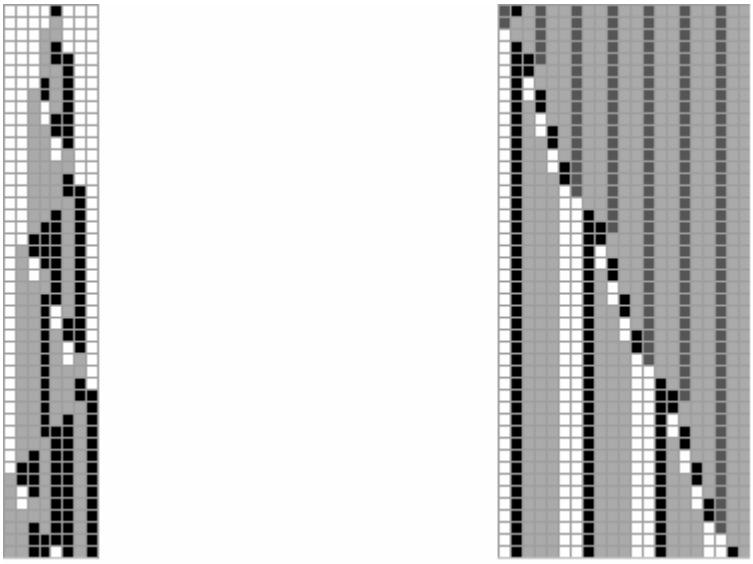
```
TMNumber[rule_,{q_,K_}]:=
FromDigits[(#[[2,3]]/.-1→0)+2(#[[2,2]])+2K(#[[2,1]]-1)&/@rule,2q K]

TMNumber[{{1,2}→{1,1,-1},{1,1}→{1,2,-1},{1,0}→{2,1,1},
{2,2}→{1,0,1},{2,1}→{2,2,1},{2,0}→{1,2,-1}},{2,3}]
```

```
596440
```

On note que le schéma de numérotation obtenu est particulier aux valeurs de s et de K considérées. On pourrait proposer une numérotation absolue des MT qui engloberait tous les cas mais ce serait fastidieux et de peu d'intérêt. Il suffit de savoir que cela est possible, autrement dit que l'ensemble des MT est effectivement récursivement énumérable.

Mathematica permet une programmation aisée de n'importe quelle MT au travers de l'instruction, TuringMachine. On peut préciser la MT visée par la liste de ses instructions ou par son numéro d'ordre accompagné des valeurs de s et de K. Il faut encore préciser les conditions initiales soit explicitement dans l'alphabet utilisé, soit sous la forme d'un entier en base, K. On peut même plonger ces conditions initiales dans un environnement périodique comme si la bande, au lieu d'être initialement vierge, avait été préalablement tapissée par un motif périodique qui accueille les données proprement dites. L'exemple qui suit concerne 45 pas d'évolution de la MT numéro 694885110, à deux états et quatre caractères, qui démarre son exécution dans l'état, 1, sur la condition initiale, '3' = 'Noir', plongée dans un environnement vierge puis périodique, qui répète le motif, {1,1,2}. A noter que Mathematica représente les caractères, 0, 1, 2, …,K-1, en niveaux de gris allant progressivement du plus clair au plus foncé, dans cet ordre.

```
ArrayPlot[Last/@TuringMachine[{694885110,2,4},{1,{{3},0}},45],
Mesh→True,PixelConstrained->True]
```

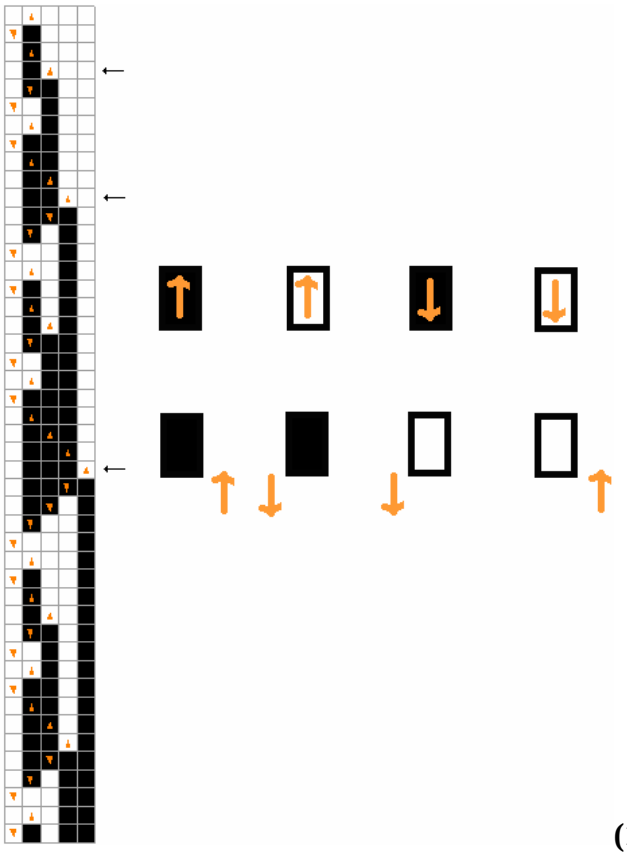


694885110

```
ArrayPlot[Last/@TuringMachine[{694885110,2,4},{1,{{3},{1,1,2}}},45],
Mesh→True,PixelConstrained->True]
```

Il est nettement plus fastidieux de programmer l'affichage de l'état de la tête à chaque pas, par exemple lors de l'évolution de la MT, 1953, à 2 états et 2 caractères :

```
{rule=1953,s=2,k=2,data={0},pos=1,step=45,tm=TuringMachine[{rule,s,k},{{1,pos},{data,0}},step]
};tm12=Transpose[Transpose[tm][[1]]][[2]];tm11=Transpose[Transpose[tm][[1]]][[1]];ArrayPlot[La
st/@tm,Mesh→True,Epilog→{Orange,Table[Rotate[Polygon[{{1/3-1+tm12[[i]],2/10+1+step-i},{2/3-1+
tm12[[i]],2/10+1+step-i},{1/2-1+tm12[[i]],2/3+1+step-i}}],-2π/s(tm11[[i]]-1)],{i,step+1}]}]
```



(1953)

On remarque l'absence d'instruction d'arrêt. La bande est initialement semi infinie vers la gauche et la tête de lecture démarre sur la case située à l'extrémité droite de la bande. On peut convenir que le calcul s'interrompt spontanément dès que la tête reçoit l'ordre de pénétrer la zone interdite. A ce moment, on peut éventuellement décider de poursuivre le calcul en ajoutant une case vierge à l'extrémité droite de la bande. Si on impose à la MT d'imprimer le contenu utile de sa bande toutes les fois que sa tête atteint une position extrême droite jamais atteinte antérieurement, on voit apparaître, dans le dernier exemple, les motifs, BNB, BNNB, BNNNB, etc, que l'on peut interpréter comme les entiers binaires de la forme générale, 2^{n+1}-2. Si on préfère ignorer les colonnes extrêmes, on voit apparaître les entiers successifs en notation unaire.

La plupart des 4096 MT à deux état et deux caractères ont un comportement inintéressant : Wolfram a montré que seules 25 d'entre elles, dont la numéro 1953, ont un comportement non trivial qui demeure cependant rudimentaire. Pour observer des MT qui effectuent un travail intéressant, il est nécessaire d'augmenter le nombre des états et/ou des caractères. Sans aller très loin dans ce sens et en cherchant bien, on a alors la surprise de rencontrer des MT qui sont capables de comportements hautement sophistiqués. Il en existe même qui sont capables d'émuler toutes les autres, on les appelle pour cette raison machines de Turing universelles, MTU.

Le problème de décision de l'arrêt des MT.

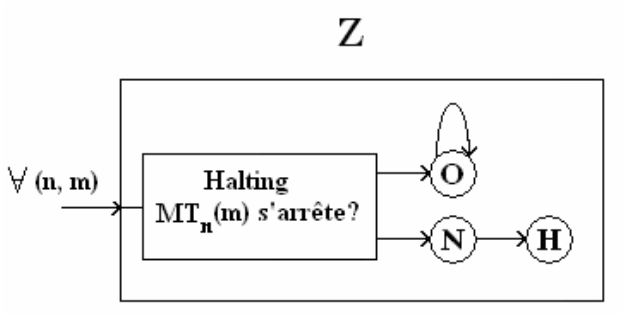
Une troisième version de la thèse de Church-Turing affirme que toute fonction (semi) calculable l'est par une MT. Il en résulte que l'ensemble des fonctions effectivement calculables coïncide avec l'ensemble des fonctions programmables sur MT qui s'arrêtent.

Qu'il existe des MT qui ne s'arrêtent pas est inévitable puisque les approches équivalentes de la calculabilité par programmes ou celle de Kleene ont révélé qu'il existait des cas de bouclages impossibles à éradiquer. Certaines MT s'arrêtent de façon tout à fait prévisible et d'autres ne s'arrêtent certainement pas mais il y a toutes les autres pour lesquelles on ne possède aucun critère de décision. On peut bien les lancer et si elles sont destinées à s'arrêter et qu'on est suffisamment patient, on finira par le découvrir mais si le calcul s'éternise, il n'existe aucun moyen de décider quand il faut abandonner tout espoir. Plus formellement, il n'existe aucune procédure effective capable de recevoir, en entrée, le numéro d'une MT et une donnée puis de décider s'il y a arrêt. On le prouve, par l'absurde, de deux manières différentes.

Première démonstration.
Nous savons comment numéroter les machines de Turing, MT_n, sans en oublier une seule. Si on alimente la $n^{ième}$ MT par la donnée, m, cela revient à lancer le programme, $MT_n(m)$. Imaginons qu'il existe un algorithme, en fait une MT, que nous appelons H(alting), qui soit capable de décider, pour tout couple, (n, m), si $MT_n(m)$ s'arrête. Concrètement, H, lit le couple, (n, m), pré inscrit sur sa bande de lecture et il termine sa propre exécution en affichant O(ui) si $MT_n(m)$ s'arrête et N(on) si $MT_n(m)$ ne s'arrête pas.

On peut alors imaginer le raffinement suivant : on fabrique une nouvelle machine, que l'on nomme, Z, en ajoutant quelques instructions à H de telle manière que Z se met à boucler si H

répond O(ui) et qu'elle s'arrête si H répond N(on). Z est aussi une MT d'où elle possède son numéro d'ordre dans l'inventaire des MT, disons n_Z. Si on alimente Z par le couple, (n_Z, m), on s'apercevoit que Z devrait s'arrêter si Z ne s'arrête pas et inversement. Dans tous les cas on a une contradiction qui invalide l'hypothèse de départ : la procédure, H(alting), n'existe pas.



Deuxième démonstration.
Il existe quantité de démonstrations équivalentes avec lesquelles on peut se sentir plus ou moins à l'aise, certaines tellement simples que l'argument de Turing a progressivement acquis ce statut d'évidence qui est le sort de tout concept dont la démonstration a été suffisamment démystifiée. Voici une autre démonstration, par l'absurde, qui est due à Chaitin.

Imaginons qu'il existe une procédure effective, H(p), qui soit capable de décider l'arrêt de n'importe quel programme, p, qui incorpore ses données ce qui est toujours possible. Commençons par faire remarquer qu'étant donné un programme, p, de longueur, K_{bits}, il est toujours possible de lui ajouter une instruction afin que le programme résultant, p', connaisse sa propre longueur, $N_{bits} > K_{bits}$. Il suffit que cette instruction soit du type, $L = N_{bits}$, où N est calculé selon la formule, $K + \ell gN = N$. Par exemple on ajouterait l'instruction, L=10013, à tout programme de longueur, $K=10^4$ bits. Construisons, à présent, le programme suivant :

- qui énumère tous les programmes dans l'ordre des longueurs croissantes jusqu'à l'entier, 1000N (ou 10000N peu importe),

- qui filtre tous ces programmes avec l'aide de la procédure, H, ne retenant que ceux qui s'arrêtent,

- qui est capable de lancer un à un ces programmes et de mémorise leur réponse, un entier binaire, dans une liste de résultats,

- enfin qui imprime le plus petit entier qui ne fait pas partie de cette liste.

Quand ce corps de programme est écrit, on en calcule la longueur, K, et on ajoute l'instruction, L=N, où N est calculé selon la formule, $K + \ell gN = N$. Appelons, Z, ce nouveau programme et montrons à quelle contradiction il mène : il est de longueur, N, il s'arrête à coup sûr et il imprime un entier qui ne fait pas partie de l'ensemble des entiers qu'impriment les programmes de longueurs inférieures à 1000N. La contradiction saute aux yeux puisque la longueur de Z, à savoir N, est largement inférieure à 1000N. La conclusion qui s'impose c'est que la procédure, H(p), n'existe pas.

Indécidabilité de l'élégance d'un programme.

Le théorème suivant, dû à Chaitin, permet de comprendre l'indécidabilité de l'arrêt sous un angle différent :

Il est impossible de décider l'élégance d'un programme.

Par définition un programme est élégant s'il s'arrête en imprimant une suite de résultats qu'aucun programme plus court que lui n'est capable d'imprimer. La démonstration de ce théorème se fait par l'absurde en posant l'existence d'un programme, E, capable de tester l'élégance d'un programme donné.

Fabriquons de toute pièce le programme, F, suivant : il comporte une instruction qui lui donne une valeur entière, N, dont il se sert pour énumérer tous les programmes, $P_{k>N}$, de longueur, k, supérieure à N, qu'il fait tourner. F utilise la procédure, E, pour sélectionner le premier programme élégant qui se présente. Il est assuré d'en trouver un car l'ensemble des programmes élégants est infini. Quand il l'a trouvé, disons pour k=j, il ne va pas plus loin et il se contente d'imprimer la suite que P_j aurait imprimée. Ce programme, F, possède sa propre longueur, disons, N_F. Réactivons, à présent, le programme, F, en y inscrivant, N_F+1 (ou +2, +3, … peu importe), à la place de N. F va nécessairement imprimer la même suite qu'un certain programme certifié élégant mais cela est impossible puisque ce programme serait nécessairement plus long que F! La conclusion qui s'impose est que la procédure E n'existe pas.

Chaitin a fait remarquer que l'inexistence d'un décideur d'élégance livre une preuve alternative de l'inexistence d'un testeur de l'arrêt des machines de Turing. De fait, si l'arrêt était décidable, il suffirait de passer en revue tous les programmes dans l'ordre canonique des longueurs croissantes, de ne retenir que ceux qui s'arrêtent, de les faire tourner en notant le résultat qu'ils impriment après quoi il serait facile de retenir le plus court de ceux qui produisent ce résultat. On aurait fabriqué de toutes pièces un testeur d'élégance ce qui a été démontré impossible à réaliser. Ce théorème possède une autre conséquence importante :

La complexité algorithmique, dite de Kolmogorov, d'une suite n'est en toute généralité pas calculable par une procédure effective.

Rappelons que la complexité algorithmique d'une suite est la longueur du plus court programme capable de la calculer sans perte. Si cette grandeur était calculable, on se trouverait en contradiction immédiate avec le théorème d'inexistence d'un décideur d'élégance pour les programmes.

L'entropie d'un système physique étant assimilable à la complexité algorithmique de la suite qui l'encode complètement, il en résulte que cette fonction n'est pas calculable en toute généralité.

Les causes de non calculabilité des fonctions sont multiples : soit elles n'obéissent à aucun schéma programmable, soit qu'elles croissent plus vite que n'importe quelle fonction calculable soit, enfin, elles sont sujettes au problème de l'arrêt,.

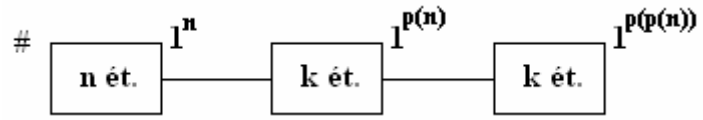
Exemples de fonctions non calculables.

L'activité du castor affairé. Un corollaire de l'indécidabilité du problème de l'arrêt est qu'il existe des fonctions non calculables. Il serait intéressant d'en expliciter une. En 1962, Rado a suggéré le problème suivant, dit « du castor affairé ». Soit l'ensemble des MT à n états, dont un état explicite d'arrêt, et deux caractères, 0 et 1. Ces MT travaillent sur une bande initialement vierge donc initialisée à {0}. S(n) est définie comme étant le plus grand nombre de transitions qu'une de ces MT peut subir avant de s'arrêter. Cette fonction, S(n), n'est pas calculable. Cela signifie qu'il n'existe pas de MT capable d'afficher, en un temps fini, la valeur de S(n) quand on lui donne la valeur de n sur sa bande de lecture. En effet, si cette MT particulière existait, le problème général de l'arrêt des MT deviendrait décidable car il suffirait de tester chaque MT soumise au test de l'arrêt pendant un nombre de pas au moins égal à S(n). Seules les premières valeurs de la fonction, S(n), sont connues soit, S(2)=6, S(3)=21, S(4)=107. La MT à 4 états et 2 caractères la plus active porte le numéro, 1736633119. Personne actuellement ne connaît la valeur de S(5) sauf qu'elle est au moins égale à 47176870. La valeur de S(6) vaut au moins $3.002\ 10^{1730}$!

La productivité du castor affairé. Le même problème révèle l'existence d'une autre fonction non calculable en rapport avec le nombre maximum de symboles '1' qu'une MT à n états est capable d'aligner consécutivement sur la bande au moment où elle s'arrête. Plus précisément, considérons toutes les MT à n états qui démarrent sur une bande initialement vierge. Trois cas sont possibles : les cas où une MT ne s'arrête pas ou ceux où elle s'arrête en imprimant une suite quelconque mais mélangée de '0' et de '1' ne nous intéressent pas. Seuls nous intéressent les cas où la MT s'arrête après avoir imprimé une suite ininterrompue de $p_i(n)$ '1'. Appelons p(n) la valeur fournie par la MT la plus productive, $p(n) = max_i\ p_i(n)$. Montrons, sans entrer dans trop de détails, que la fonction, p(n), n'est pas calculable, au sens que la thèse de Church prête à ce terme à savoir qu'il n'existe pas de MT_k, à k états, qui lit en entrée l'entier n, calcule et puis s'arrête en imprimant p(n). Nous nous contentons d'esquisser la démonstration qui se fait en 6 étapes. Les relations qui suivent sont faciles à comprendre :

$p(1) = 1$	(Un inventaire vite fait des machines à un seul état livre ce résultat)
$p(n) \geq n$	(Il existe une MT à n états assez triviale qui écrit n '1')
$p(n+1) \geq p(n)$	(Evident)
$p(n+11) \geq 2n$	(On peut construire une MT à 11 états qui double le nombre de '1')

Le cœur de la démonstration est l'inégalité suivante, $p(n+2k) \geq p(p(n))$. On l'obtient en mettant en série, sur une bande initialement vierge, trois MT particulières : une MT, à n états, qui imprime n '1' , suivie par deux MT_k supposées exister.



La productivité de la machine résultante vaut certainement au moins, p(p(n)). Voici à présent vers quelle contradiction cela nous entraîne. De, $p(n+1) \geq p(n)$, on déduit l'inférence, $i > j \Rightarrow p(i) > p(j)$, qui entraîne, par contra position : $p(j) \geq p(i) \Rightarrow j \geq i$.

Quel que soit n, on peut dès lors écrire la séquence suivante :

$$p(n+2k) \geq p(p(n)) \Rightarrow n+2k \geq p(n) \Rightarrow n+11+2k \geq p(n+11) \Rightarrow n+11+2k \geq 2n$$

une impossibilité manifeste puisque k est un entier fixé et que n peut être arbitrairement grand.

Seules sont connues les premières valeurs de p(n), à savoir, p(2)=4, p(3)=5, p(4)=13. Au-delà, on sait seulement que, p(5) ≥4098 et p(6) ≥$1.29\ 10^{865}$.

En fait les fonctions, S(n) et p(n), croissent plus vite que n'importe quelle fonction calculable donnée d'avance. Insistons sur le fait que lorsqu'on affirme que les fonctions S(n) et p(n) ne sont pas calculables, on n'exclut absolument pas que certaines valeurs soient trouvables par un programme qui s'arrête après avoir inventorié tous les cas possibles, les exemples, n=2,3,4, en sont la preuve. Mais il arrivera que pour certaines valeurs de n, certaines machines de Turing ne s'arrêteront pas empêchant de poursuivre l'inventaire.

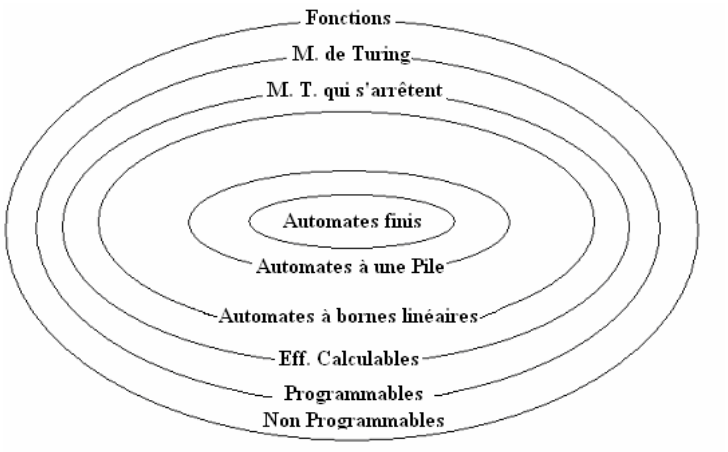
Le calcul des réels. Le calcul des réels se pose comme suit : on demande de construire une MT qui calcule sur bande initialement vierge les décimales de $\sqrt{2}$ ou de π et les imprime au compte gouttes sur sa bande. Une variante serait de demander de construire une MT qui imprime les n premières décimales du nombre demandé à partir de la donnée n.

Les réels que l'on rencontre dans la pratique mathématique courante sont calculables. Tous les réels ne sont cependant pas calculables. En voici un dont le caractère exotique apparaît d'emblée : le réel, compris entre 0 et 1, dont le $n^{ième}$ chiffre vaut 1 si la $n^{ième}$ MT (dans la numérotation de votre choix) s'arrête sur une bande vierge et 0 si elle ne s'arrête pas, n'est pas calculable. Cette absence de calculabilité est clairement reliée à l'indécidabilité du problème de l'arrêt des MT.

Un simple argument de comptage permet de se rendre compte qu'il doit exister beaucoup de réels non calculables : l'ensemble des réels est non dénombrable alors que l'ensemble des MT l'est. Il n'y a donc pas assez de MT pour calculer tous les réels. On peut renverser la proposition et comprendre que les MT définissent précisément les réels effectivement calculables. Tel est le cas des rationnels bien sûr mais aussi d'un grand nombre d'irrationnels pour lesquels il existe une procédure algorithmique qui condense la totalité de l'information nécessaire au calcul de leurs décimales. De tels nombres sont dits algorithmiquement compressibles. Les nombres algorithmiquement incompressibles ou encore aléatoires au sens de Kolmogorov ne sont certainement pas calculables : il n'existe pas d'autre procédure pour les calculer que d'inclure la succession de leurs décimales dans le programme ce qui rend celui-ci de longueur infinie et donc inacceptable. Autrement dit, un gros contingent de réels non calculables est constitué de nombres dont les décimales binaires ont été tirées à pile ou face. Cependant un argument diagonal maintes fois utilisé permet de voir qu'il existe des réels non calculables et non aléatoires, tel celui dont la $k^{ième}$ décimale vaut 3 si la $k^{ième}$ MT travaillant sur une bande vierge imprime un $k^{ième}$ chiffre différent de 3 (ou n'imprime pas de $k^{ième}$ chiffre) et n'importe quel chiffre autre que 3 sinon.

La hiérarchie des automates.

L'adoption de la thèse de Church-Turing situe la MT au sommet de la hiérarchie des automates. Les automates subalternes sont limités en puissance calculatoire du fait d'un accès limité à une mémoire restreinte. Cette mémoire peut être finie comme dans les automates finis déterministes. Elle peut se limiter à une pile unique qui accepte l'empilement et le dépilement d'un caractère à la fois comme c'est le cas dans l'automate à une pile (on montre qu'un automate à plusieurs piles est équivalent à une MT). Elle peut accepter une mémoire potentiellement infinie mais dont le taux d'occupation ne peut pas croître plus vite que linéairement en fonction de la taille des données comme dans l'automate à mémoire bornée. En comparaison, la machine de Turing dispose d'une mémoire potentiellement infinie et sans restriction d'accès.



On pourrait aborder le problème de la calculabilité par d'autres versants encore, les machines à registre, les systèmes de Post, etc, avec à la clef un nouvel énoncé de la thèse de Church-Turing mais le fait est que tous ces systèmes s'avèreraient de puissances calculatoires équivalentes.

En résumé, les versions toutes équivalentes de la thèse de Church-Turing s'énoncent comme suit :

Les fonctions (semi) calculables (synonyme : mu-récursives) sont les fonctions programmables soit dans un langage mu-récursif ou de façon équivalente sur une MT.

Les fonctions (effectivement) calculables sont les fonctions programmables soit dans un langage mu-récursif sur tests sûrs ou de façon équivalente sur une MT qui s'arrête.

Insistons sur le fait que cette thèse n'est pas démontrable : elle sert uniquement à briser la circularité qui résulterait de l'idée que ce qui est calculable doit l'être ni plus ni moins par un humain et inversement. Par contre, rien n'empêche qu'elle soit réfutable par quelqu'un qui proposerait un nouveau mode de calcul effectivement plus puissant que la machine de Turing. Personne, actuellement, ne croit à cette éventualité.

Emulation calculatoire entre programmes : le seuil d'universalité.

La thèse de Church-Turing pose que les machines de Turing programment tout ce qui est programmable et que celles qui s'arrêtent calculent tout ce qui est effectivement calculable. D'autres systèmes, que nous définirons plus loin, revendiquent le même statut, par exemple, les machines à registres (MR), les automates cellulaires (AC) et les systèmes arithmétiques (SA). L'équivalence entre ces systèmes repose sur le principe d'émulation.

On dit qu'un système calculatoire, A, émule un autre système, B, lorsqu'il existe un encodage effectif des conditions initiales de A qui livre des conditions initiales valables pour B telles qu'au terme de l'évolution de B, il est possible de décoder l'état final de B en terme de l'état final de A. Ce procédé différé permet de connaître l'état final de A sans avoir procédé à l'exécution de son évolution. Un système peut très bien en émuler un autre sans que la réciproque soit vraie, il est tout simplement plus puissant que lui.

Lorsqu'un système est capable d'émuler n'importe quel autre système, on dit qu'il est universel au sens de Turing (synonyme : calculatoirement universel). En particulier, nous appellerons machine de Turing universelle, en abrégé MTU, toute machine de Turing qui est capable d'émuler toutes les autres y compris elle-même. Tous les systèmes universels au sens de Turing sont équivalents quant à leurs possibilités de calculs, seule la vitesse de calcul peut varier d'un système à l'autre dans des conditions parfois dramatiques tel un ralentissement (hyper)exponentiel. L'encodage émulatoire d'un système universel s'appelle sa programmation.

Tous les systèmes calculatoirement universels sont sujets au problème de l'indécidabilité de l'arrêt. Ils sont d'ailleurs sujets à quantité d'autres types d'indécidabilités tant il est vrai que toute décision relative au comportement à plus ou moins long terme de n'importe quel système universel tend à être indécidable. Il est par exemple indécidable de savoir s'il existe une configuration initiale des données qui mène une MT vers un état final prescrit d'avance ou encore qui la fait obligatoirement emprunter ou éviter une instruction donnée, etc. L'inverse n'est pas vrai : il existe des systèmes sujets à tel type d'indécidabilité et qui ne sont pas universels pour autant. La sujétion au problème de l'arrêt est donc une condition nécessaire mais non suffisante d'universalité.

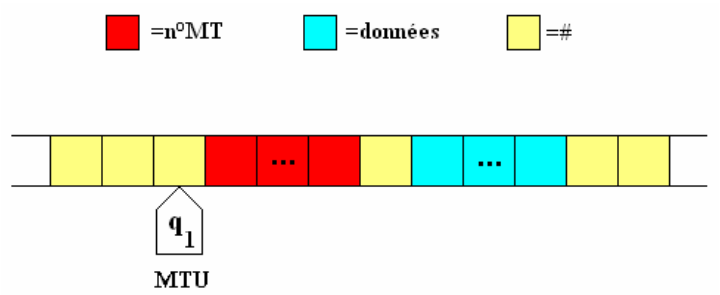
On démontre en théorie des langages que toute question générique relative aux langages réguliers est nécessairement décidable. Cela cesse d'être vrai pour les langages hors contextes qui ne sont pourtant pas situés bien haut dans la hiérarchie de Chomsky : les automates à une pile qui reconnaissent ces langages ne sont pas universels au sens de Turing. On peut traduire autrement cette observation en affirmant que l'universalité calculatoire est plus fondamentale que l'indécidabilité et il ne faut pas chercher ailleurs la raison pour laquelle le titre de cet exposé s'y réfère.

On prouve l'universalité calculatoire d'un système en montrant qu'il est capable d'émuler un système déjà reconnu comme universel. Ceci n'a évidemment de sens que si on peut faire état d'un système universel pouvant servir de modèle et c’est cette tâche qui va nous occuper à présent.

La machine de Turing universelle.

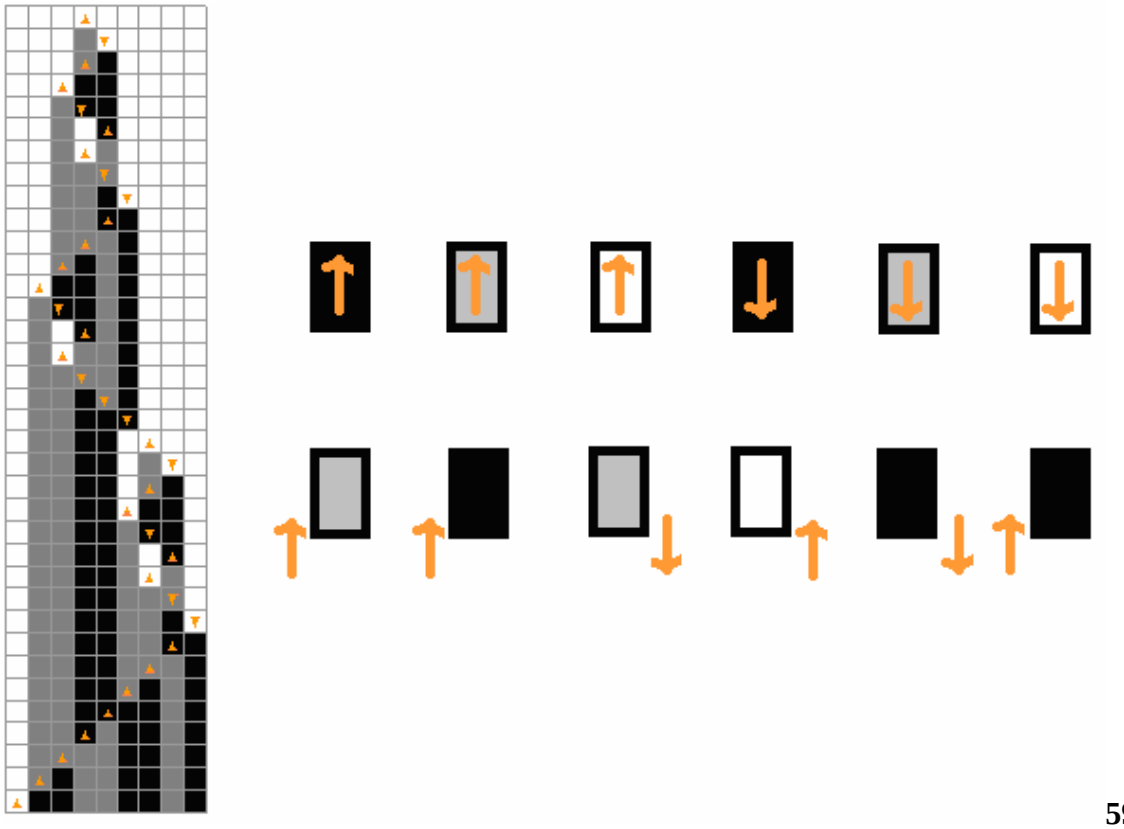
Il n'est pas évident a priori qu'il existe une machine de Turing universelle. La plupart des machines de Turing sont dédicacées à des tâches particulières, l'addition ou la multiplication de deux entiers binaires, la recherche d'une chaîne spécifique de caractères, etc. Une MTU, par contre, est capable de calculer tout ce qui est calculable. Il y a autant de différences entre une MT particulière et une MTU qu'entre une calculette ou un agenda électronique et un ordinateur.

Pour fonctionner correctement, une MTU doit être capable d'émuler n'importe quelle MT. Elle doit pouvoir lire, sur sa bande, le code d'instructions de la MT particulière qu'elle veut émuler et les données qui correspondent aux instances demandées. Ces deux parties, qui incarnent le software, doivent être séparées d'une manière ou d'une autre sur la bande afin que la machine universelle puisse travailler sans ambiguïté en reconnaissant les points de départ et de fin du programme et des données. On peut aussi décider d'incorporer les données au programme comme cela se fait parfois en pratique informatique.



On prouve l'existence d'une MTU en la construisant de toutes pièces. Dessiner les plans d'une MTU est fastidieux. On trouvera dans l'ouvrage de référence de Feynman (Feynman Lectures on Computation) le détail de la construction, due à Von Neuman, d'une MTU comprenant 23 états et 8 caractères. Cette MTU, consiste, en fait, en un assemblage minutieux de plusieurs MT individuelles qui se consacrent chacune à une tâche bien définie, localiser une chaîne de caractère sur la bande, la recopier à une adresse imposée, concaténer deux chaînes, effacer une chaîne, effectuer un test logique etc. L'ensemble complet travaille alors comme un hardware d'ordinateur.

Il a fallu attendre 1962 pour que Minsky parvienne à construire une MTU raisonnablement plus simple, sur base de 7 états et 4 caractères. Par la suite, on a pu montrer que des machines encore plus simples existaient et le record incontesté de concision est actuellement détenu par la MTU numéro 8679752795626 à 2 états et 5 caractères. Wolfram a même conjecturé que la MT numéro 596440, à 2 états et 3 caractères, pourrait battre ce record. La figure ci-dessous en montre l'évolution sur une bande initialement vierge. Aux dernières nouvelles, il semblerait que Alex Smith ait démontré cette conjecture dans une version faible dont la signification est précisée dans la section suivante. Ceci tend à prouver que le seuil d'universalité des MT est situé extraordinairement bas. La concision d'une MTU a cependant comme toujours un prix en terme de complexité que l'on paye au moment de sa programmation.



596440

La machine de Turing universelle est l'ancêtre sur papier de l'ordinateur moderne. Il est remarquable que son architecture primitive, due à von Neumann, ait traversé les décennies qui ont suivi sans altérations majeures. Signalons quand même qu'il existe des architectures, dites parallèles, qui s'inspirent d'un principe différent de celui de von Neumann.

Universalités calculatoires fortes et faibles.

En informatique théorique, la MTU incarne le principe de l'universalité calculatoire. Le pré encodage du programme et des données éventuelles sur sa bande de lecture écriture est de longueur finie. Vu que cette bande est potentiellement infinie, au moins dans une direction, cela implique généralement que la partie non utilisée de la bande soit initialement vierge. Lorsque toutes ces conditions sont remplies on parle d'universalité forte.

Rien n'empêche que l'on plonge les données et le programme dans un environnement non vierge mais il y a des limites à ce que l'on peut faire à ce sujet. On peut certainement tapisser le reste de la bande par un motif périodique. La périodicité fait que lorsqu'il s'avère nécessaire d'allonger la bande en cours de calcul parce que la tête s'aventure de plus en plus loin, on sait au moins ce qui doit être pré écrit sur ce supplément de bande avant de le mettre en service. Lorsqu'on recourt à ce procédé, on parle d'universalité faible.

En théorie, on pourrait même faire plus compliqué. Rien ne devrait s'opposer à ce qu'on tapisse la partie de bande non utilisée initialement par un motif non répétitif pourvu qu'il soit effectivement calculable. Toute adjonction ultérieure de morceaux de bande

exigerait qu'on calcule les symboles à y pré inscrire au fur et à mesure des besoins avec l'aide d'un automate externe, cela peut paraître compliqué mais c'est certainement faisable.

Il ne faut pas attacher d'importance excessive aux qualificatifs "forte" et "faible" : il s'agit dans tous les cas d'une véritable universalité calculatoire même si, dans le second cas, un travail préparatoire supplémentaire est nécessaire dans l'initialisation de la bande.

On pourrait se demander pour quelles raisons on considère des universalités non fortes. Le problème toujours en discussion ne concerne actuellement que les systèmes universels de très petites dimensions pour lesquels aucune preuve d'universalité forte n'est connue. C'est en fait le cas de la démonstration de Smith concernant la MTU 596440, à 2 états et 3 caractères, qui ne prouve que son universalité faible.

Disons enfin quelques mots de ce qui n'est pas autorisé lors du pré encodage de la bande. Il est interdit de tapisser la partie de la bande non initialement utilisée à partir d'un motif qui ne serait pas effectivement calculable car cela impliquerait qu'on se trouverait tôt ou tard dans l'incapacité de connaître le caractère à pré inscrire sur les morceaux de bande que l'on serait amené à ajouter en cours de calcul. L'indécidabilité de l'arrêt de la MT ainsi construite ne serait plus spontanée, il se pourrait qu'elle résulte de la non calculabilité de cet encodage.

A fortiori il est interdit de tapisser la bande à partir d'un motif aléatoire. Cette remarque peut paraître ridicule mais elle fixe, une fois pour toute, la différence qui existe entre l'universalité "intelligente", au sens de Turing, et l'universalité "triviale", au sens de Borel. Borel a montré que la plupart des réels présentent cette particularité que n'importe quelle suite finie de chiffres donnée d'avance, aussi longue soit elle, apparaît quelque part dans la succession de ses décimales. C'est évident dans le cas du nombre de Chambertown, dont les décimales successives égrènent la suite des entiers naturels, sous la forme :

C=0.1234567891011121314…

mais cela reste vrai des nombres, dits universels au sens de Borel, dont l'ensemble est infini et non dénombrable.

Dans un certain sens, le résultat de n'importe quel calcul (ou d'ailleurs de n'importe quel texte ou œuvre d'art codée en binaire) se trouve caché quelque part dans la succession des chiffres de n'importe quel réel universel au sens de Borel. Toutefois, cette propriété est inutilisable car à quoi cela sert-il de savoir que la réponse à un calcul se trouve encodé quelque part si on ne sait pas où précisément ? L'absence de constructivisme est particulièrement interpellant dans les exemples de ce genre. Quelle confiance peut-on développer dans de tels nombres « universels » qui donnent l'illusion de contenir tous les savoirs passés, présents et futurs ? Ces réels ne sont même pas nommables par une formule finie rédigée en français puisque ces formules forment un ensemble dénombrable. Ils sont pourtant infiniment denses sur l'axe réel.

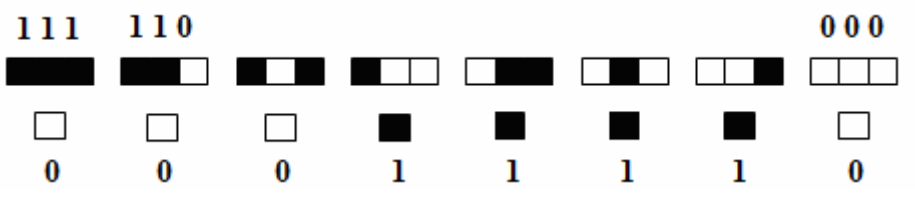
En comparaison, l'universalité, forte ou faible, au sens de Turing est beaucoup plus exigeante puisqu'elle est conçue de telle manière que toute réponse à un calcul figure obligatoirement dans un domaine de la bande de lecture écriture parfaitement spécifié à l'avance. Nous passons à présent en revue quelques systèmes universels au sens de Turing.

Les automates cellulaires et la règle universelle 110.

Les automates cellulaires sont des systèmes discrets doués d'un pouvoir calculatoire éventuellement universel. Ils ont été étudiés intensivement par Wolfram qui n'hésite pas à les présenter comme une alternative possible, discrète et constructive, à l'analyse infinitésimale. Les succès enregistrés dans les applications à la physique restent toutefois actuellement confinés à la mécanique des fluides. Rappelons qu'en mécanique des fluides traditionnelle, on discrétise un ensemble d'équations aux dérivées partielles héritées de la physique classique ce qui revient de fait à remplacer l'espace-temps continu par un automate cellulaire. Le point de vue défendu par Wolfram revient à penser qu'il pourrait être intéressant de rechercher de nouveaux automates dédicacés aux problèmes posés sans transiter par la discrétisation d'équations différentielles.

Les automates cellulaires traitent un réseau discret, à n dimensions, constitué de cellules adjacentes qui interagissent entre voisines en respectant un ensemble de règles imposées. Chaque cellule porte un indice qui permet de la repérer sans ambiguïté et se trouve dans un état codé par un symbole choisi parmi s valeurs distinctes. Une horloge commande la transition d'état que subissent simultanément toutes les cellules du réseau en fonction de l'état actuel de chacune et de celui de ses voisines qui sont concernées. Les automates dits élémentaires sont les plus simples de tous : ils sont unidimensionnels, ils possèdent s=2 symboles et ils ne font intervenir que les deux voisins immédiats soit, r (= 1), de chaque côté de la cellule incriminée.

Il existe, $s^{s^{2r+1}}$, automates à une dimension, s symboles et 2r voisins et, par conséquent, $2^{2^3} = 256$ automates élémentaires. On peut numéroter ceux-ci systématiquement en adoptant la numérotation de Wolfram que l'on illustre sur l'exemple de la règle élémentaire, 30 = 00011110_{bin}. Un code immuable des couleurs, (Noir = 1, Blanc = 0), permet de représenter graphiquement les règles de transition de l'automate :



La ligne supérieure range les cellules mères dans un ordre immuable qui égrène les codes binaires des entiers dans l'ordre descendant. La ligne inférieure précise l'état de la cellule fille en fonction de son état antérieur et de celui de ses deux voisines immédiates. Les 256 automates élémentaires que l'on peut construire sur ce modèle ne sont pas tous distincts car certaines règles sont équivalentes soit par inversion des symboles 0 et 1, soit par réflexion gauche droite ou enfin par combinaison des deux. Par exemple, la règle 30 est équivalente dans l'ordre aux règles, 135, 86 et 149. Au total il ne subsiste que 88 règles distinctes dont voici l'évolution d'abord sur conditions initiales particulières, une case noire entourée de cases blanches, puis sur conditions initiales pseudo aléatoires.

{0, 255, 0, 255}

{1, 127, 1, 127}

{2, 191, 16, 247}

{3, 63, 17, 119}

{4, 223, 4, 223}

{5, 95, 5, 95}

{6, 159, 20, 215}

{7, 31, 21, 87}

{8, 239, 64, 253}

{9, 111, 65, 125}

{10, 175, 80, 245}

{11, 47, 81, 117}

{12, 207, 68, 221}

{13, 79, 69, 93}

{14, 143, 84, 213}

{15, 15, 85, 85}

{18, 183, 18, 183}

{19, 55, 19, 55}

{22, 151, 22, 151}

{23, 23, 23, 23}

{24, 231, 66, 189}

{25, 103, 67, 61}

{26, 167, 82, 181}

{27, 39, 83, 53}

{28, 199, 70, 157}

{29, 71, 71, 29}

{30, 135, 86, 149}

{32, 251, 32, 251}

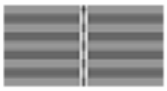
{33, 123, 33, 123}

{34, 187, 48, 243}

{35, 59, 49, 115}

{36, 219, 36, 219}

{37, 91, 37, 91}



{38, 155, 52, 211}

{40, 235, 96, 249}

{41, 107, 97, 121}

{42, 171, 112, 241}

{43, 43, 113, 113}

{44, 203, 100, 217}

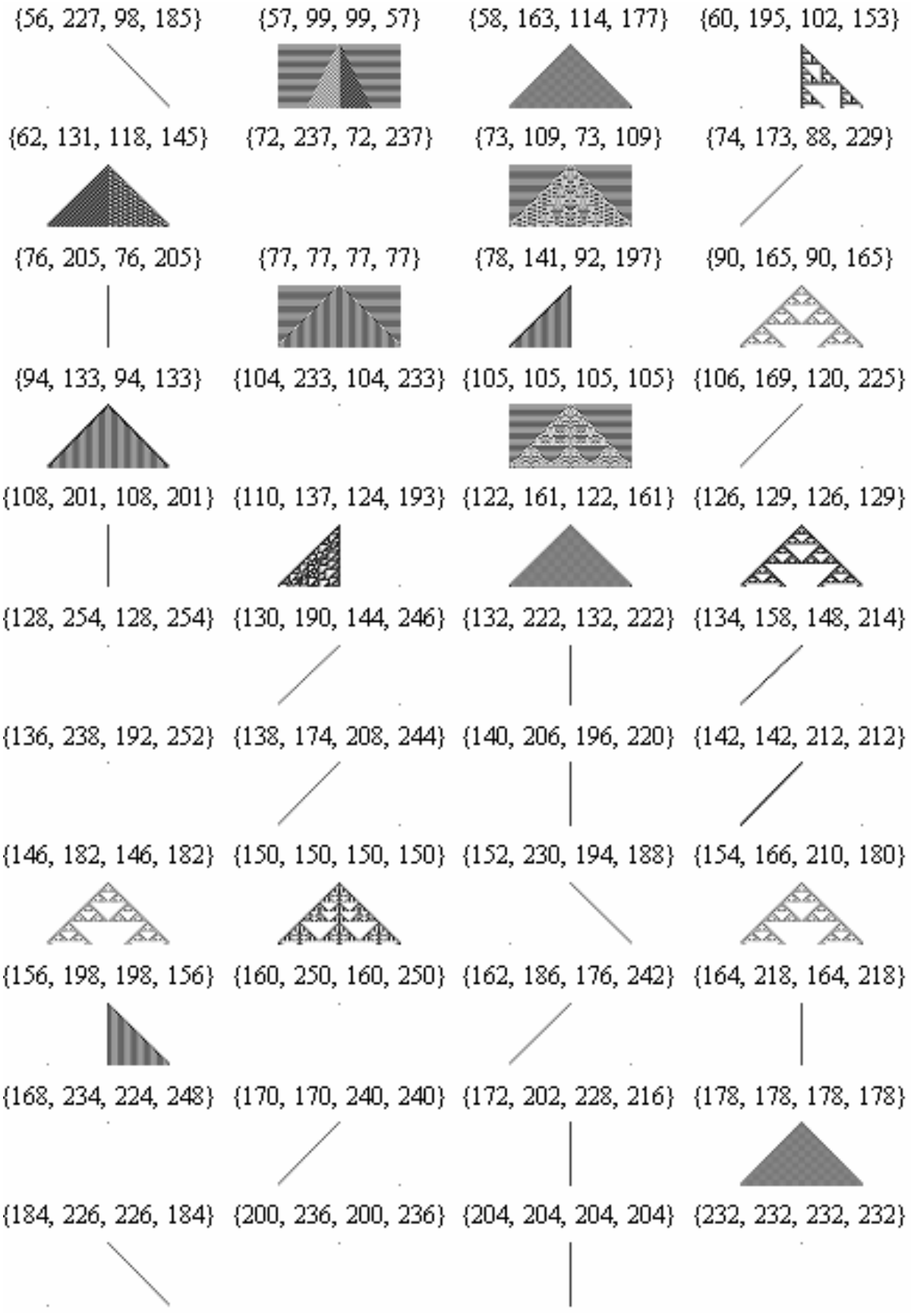
{45, 75, 101, 89}

{46, 139, 116, 209}

{50, 179, 50, 179}

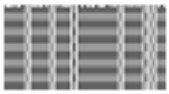
{51, 51, 51, 51}

{54, 147, 54, 147}

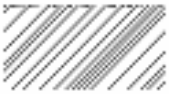

{56, 227, 98, 185}
{57, 99, 99, 57}
{58, 163, 114, 177}
{60, 195, 102, 153}
{62, 131, 118, 145}
{72, 237, 72, 237}
{73, 109, 73, 109}
{74, 173, 88, 229}
{76, 205, 76, 205}
{77, 77, 77, 77}
{78, 141, 92, 197}
{90, 165, 90, 165}
{94, 133, 94, 133}
{104, 233, 104, 233}
{105, 105, 105, 105}
{106, 169, 120, 225}
{108, 201, 108, 201}
{110, 137, 124, 193}
{122, 161, 122, 161}
{126, 129, 126, 129}
{128, 254, 128, 254}
{130, 190, 144, 246}
{132, 222, 132, 222}
{134, 158, 148, 214}
{136, 238, 192, 252}
{138, 174, 208, 244}
{140, 206, 196, 220}
{142, 142, 212, 212}
{146, 182, 146, 182}
{150, 150, 150, 150}
{152, 230, 194, 188}
{154, 166, 210, 180}
{156, 198, 198, 156}
{160, 250, 160, 250}
{162, 186, 176, 242}
{164, 218, 164, 218}
{168, 234, 224, 248}
{170, 170, 240, 240}
{172, 202, 228, 216}
{178, 178, 178, 178}
{184, 226, 226, 184}
{200, 236, 200, 236}
{204, 204, 204, 204}
{232, 232, 232, 232}

{0, 255, 0, 255}

{1, 127, 1, 127}



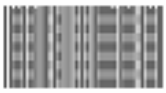
{2, 191, 16, 247}



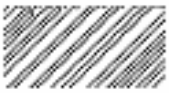
{3, 63, 17, 119}

{4, 223, 4, 223}

{5, 95, 5, 95}



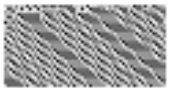
{6, 159, 20, 215}



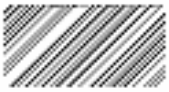
{7, 31, 21, 87}

{8, 239, 64, 253}

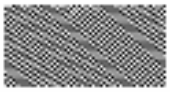
{9, 111, 65, 125}



{10, 175, 80, 245}



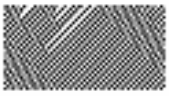
{11, 47, 81, 117}



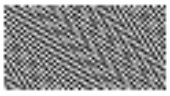
{12, 207, 68, 221}

{13, 79, 69, 93}

{14, 143, 84, 213}



{15, 15, 85, 85}

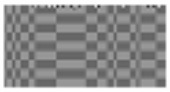


{18, 183, 18, 183}

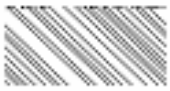
{19, 55, 19, 55}

{22, 151, 22, 151}

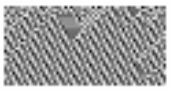
{23, 23, 23, 23}



{24, 231, 66, 189}



{25, 103, 67, 61}



{26, 167, 82, 181}

{27, 39, 83, 53}

{28, 199, 70, 157}

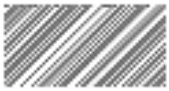
{29, 71, 71, 29}

{30, 135, 86, 149}

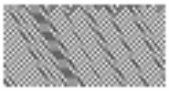
{32, 251, 32, 251}

{33, 123, 33, 123}

{34, 187, 48, 243}

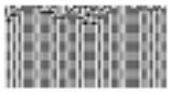


{35, 59, 49, 115}

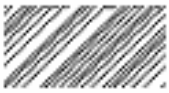


{36, 219, 36, 219}

{37, 91, 37, 91}



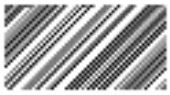
{38, 155, 52, 211}



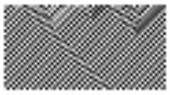
{40, 235, 96, 249}

{41, 107, 97, 121}

{42, 171, 112, 241}



{43, 43, 113, 113}



{44, 203, 100, 217}

{45, 75, 101, 89}

{46, 139, 116, 209}

{50, 179, 50, 179}

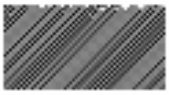
{51, 51, 51, 51}

{54, 147, 54, 147}

{56, 227, 98, 185}

{57, 99, 99, 57}

{58, 163, 114, 177}



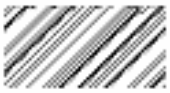
{60, 195, 102, 153}

{62, 131, 118, 145}

{72, 237, 72, 237}

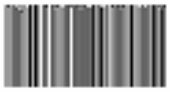
{73, 109, 73, 109}

{74, 173, 88, 229}



{76, 205, 76, 205}

{77, 77, 77, 77}



{78, 141, 92, 197}

{90, 165, 90, 165}

{94, 133, 94, 133}

{104, 233, 104, 233}

{105, 105, 105, 105}

{106, 169, 120, 225}

{108, 201, 108, 201}

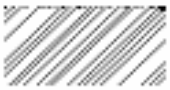
{110, 137, 124, 193}

{122, 161, 122, 161}

{126, 129, 126, 129}

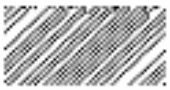
{128, 254, 128, 254}

{130, 190, 144, 246}



{132, 222, 132, 222}

{134, 158, 148, 214}



{136, 238, 192, 252}

{138, 174, 208, 244}

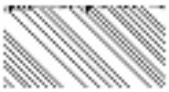
{140, 206, 196, 220}

{142, 142, 212, 212}

{146, 182, 146, 182}

{150, 150, 150, 150}

{152, 230, 194, 188}

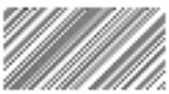


{154, 166, 210, 180}

{156, 198, 198, 156}

{160, 250, 160, 250}

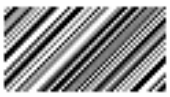
{162, 186, 176, 242}



{164, 218, 164, 218}

{168, 234, 224, 248}

{170, 170, 240, 240}



{172, 202, 228, 216}

{178, 178, 178, 178}

{184, 226, 226, 184}

{200, 236, 200, 236}

{204, 204, 204, 204}

{232, 232, 232, 232}

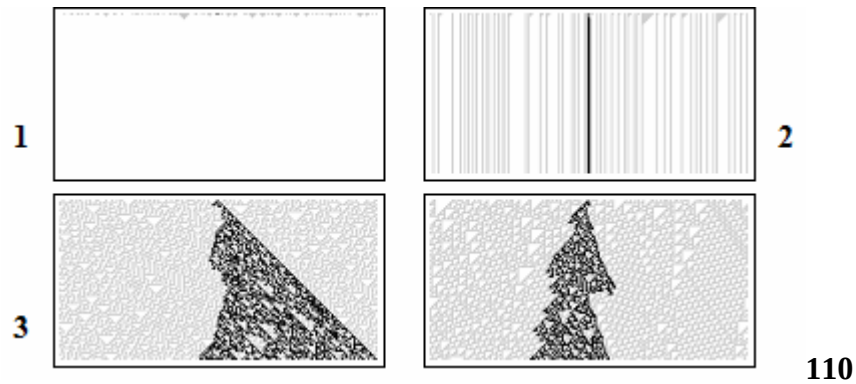
L’examen de ces figures révèle une extrême diversité de comportements, en particulier dans le cas général de conditions initiales pseudo aléatoires :

- Certains automates, 0, 8, 32, …, dits de classe 1, ne se départissent jamais d'un comportement rudimentaire : ils convergent vers un état uniforme, blanc ou noir.

- D'autres automates, 1, 2, 3, 4, …, dits de classe 2, tendent plus ou moins rapidement vers un état attracteur figé mais non uniforme.

- Une troisième classe d'automates, 22, 30, …, dits de classe 3, évoluent de façon erratique. Wolfram a d'ailleurs tiré parti du comportement chaotique de la règle, 30, pour en faire le générateur de nombres pseudo aléatoires du logiciel Mathematica. Il franchit de fait les tests statistiques les plus exigeants.

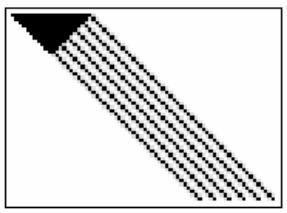
On met en évidence la différence de comportement de ces classes d’automates en perturbant localement leurs conditions initiales et en regardant ce qui change dans leur évolution. Dans la classe 1 (automate 128 dans l’exemple représenté ci-dessous), la perturbation s’éteint rapidement comme cela se produit lorsqu’on perturbe le mouvement d’un pendule avec frottements. Dans la classe 2 (tel l’automate 140), la perturbation se maintient sans amplification comme cela se produirait si on pouvait perturber le mouvement d’une planète autour de son étoile. Les systèmes de classe 3 (tel l’automate 30) sont chaotiques : ils présentent une nette sensibilité aux perturbations des conditions initiales. Il suffit d'altérer un seul bit de celles-ci pour que la perturbation se propage rapidement de cellules en cellules. L’automate 110, qui est cité en quatrième exemple, est situé à la frontière qui sépare les classes 2 et 3 : nous allons voir que des glisseurs y assurent une propagation contrôlée de la perturbation. Cette propriété remarquable est d'ailleurs en relation avec l'universalité calculatoire de l'automate 110.



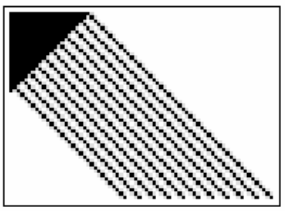
On pourrait considérer des automates plus complexes, travaillant sur plus de deux caractères ou prenant en considération des interactions avec des voisins plus lointains que les voisins immédiats et la classification précédente subsisterait. Elle ne diffère pas en substance de celle que l'on fait à propos des systèmes physiques qui peuvent évoluer vers des attracteurs simples du type points fixes ou cycles limites ou encore évoluer de façon chaotique.

Aucun automate appartenant à l'une des classes, 1, 2 ou 3, ne peut prétendre à l'universalité calculatoire. C'est évident en ce qui concerne les automates des classes 1 et 2 qui ne remplissent que de tâches très particulières. Par exemple, les automates, 144 et 152, effectuent le calcul des fonctions, Ceil[n/4] et Ceil[n/2], comme dans l'exemple, n=22 :

```
ArrayPlot[CellularAutomaton[règle,{Table[1,{n,22}],0},50]]
```



Règle 144



Règle 152

L'encodage de la donnée sur la première ligne, ici 22, et le décodage des réponses, respectivement 6 et 11, sont suffisamment clairs sur les figures pour qu'aucun commentaire ne soit nécessaire. En modifiant l'encodage et/ou le décodage, on pourrait sans doute utiliser les mêmes automates pour calculer quelques autres fonctions simples mais le fait demeurerait qu'ils ne seront jamais capables d'universalité.

Aucun automate de classe 3 ne peut probablement prétendre à l'universalité calculatoire mais pour une raison très différente : ils ont un comportement tellement erratique qu'il ne sera jamais possible de recueillir la solution cherchée dans une zone prévisible de leur espace mémoire. Ces systèmes peuvent bien être universels au sens de Borel, ils ne le seront pas au sens de Turing : la solution à n'importe quel problème peut se trouver, par hasard, encodée quelque part dans le schéma d'évolution de l'automate mais il ne sera jamais possible de savoir où précisément.

Certains automates cellulaires peuvent-ils prétendre à l'universalité? S'ils existent, ils se situent nécessairement à la frontière qui sépare les classes 2 et 3, suffisamment complexes pour que l'information puisse se propager au sein de la mémoire et pas trop complexes afin que cette propagation reste contrôlable. La réponse à la question posée est positive. Toute MT à s états et k caractères peut être émulée par un automate cellulaire possédant k(s+1) symboles qui fait interagir les voisins immédiats (r =1). Ses règles sont données par :

```
TMToCA[rules_,k_:2]:=Flatten[{Map[g[#,k]&,rules],{_,x_,_}→x}]
g[{s_,a_}→{sp_,ap_,d_},k_]:={If[d==1,Identity,Reverse]
                                    [{ks+a,x_,_}]→k sp+x,{_,k s+a,_}→ap}
```

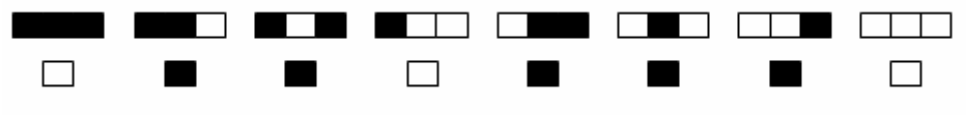
Il suffit d'appliquer la fonction TMToCA à une MT dont on sait qu'elle est universelle, par exemple la MTU numéro 596440 à 2 états et 3 symboles, pour trouver un automate cellulaire capable de l'émuler. L'automate cellulaire correspondant est alors nécessairement universel lui aussi, dans l'exemple il comporte 9 symboles et porte sur les voisins directs (r=1). Les règles de cet automate unidimensionnel sont les suivantes (x et x$ sont des variables muettes et _ remplace n'importe lequel des 9 symboles de l'automate) :

```
TMToCA[{{1,2}→{1,1,-1},{1,1}→{1,2,-1},{1,0}→{2,1,1},{2,2}→{1,0,1},
{2,1}→{2,2,1},{2,0}→{1,2,-1}},3]
```

```
{{_,x$_,2+ks}→3+x$,{_,5,_}→1,{_,x$_,1+ks}→3+x$,{_,4,_}→2,{ks,x$_,_}→6+x$,
{_,3,_}→1,{2+ks,x$_,_}→3+x$,{_,8,_}→0,{1+ks,x$_,_}→6+x$,{_,7,_}→2,
{_,x$_,ks}→3+x$,{_,6,_}→2,{_,x_,_}→x}
```

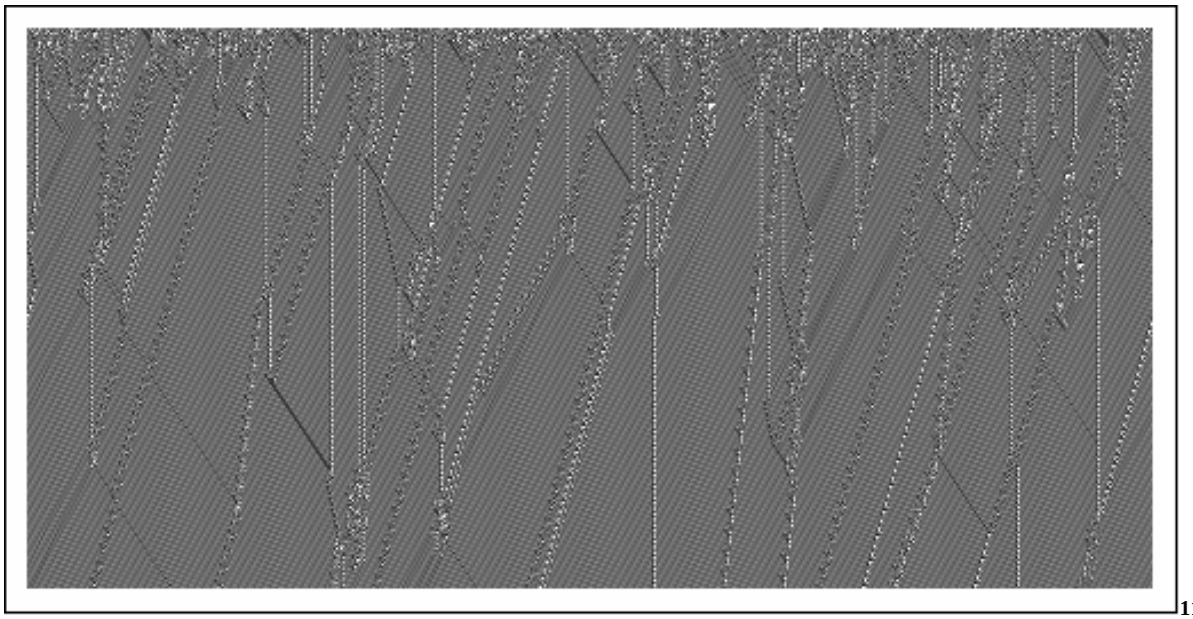
Pour être complet, il faut encore préciser le mode d'encodage des conditions initiales de l'automate. L'instruction, TMToCA, est ainsi conçue, qu'une condition initiale de l'automate cellulaire constituée d'une seule cellule de couleur entourée de '0' correspond à une MT positionnée dans son état initial, q_1, avec une bande vierge. La fonction inverse, CAToTM, qui livre une MT capable d'émuler un automate cellulaire donné existe également.

Il existe des automates plus concis que celui qui vient d'être proposé, à 9 symboles et (1+1) voisins. Cook a, en effet, démontré l'universalité faible de l'automate élémentaire, 110, à s = 2 symboles et (1+1) voisins! Il est probablement le seul parmi les automates élémentaires à posséder cette propriété. Il est tellement simple qu'il souffre inévitablement d'un défaut qui le rend inutilisable en pratique : l'encodage des données d'un problème quelconque et le décodage du résultat sont sans doute hors d'atteinte.



110

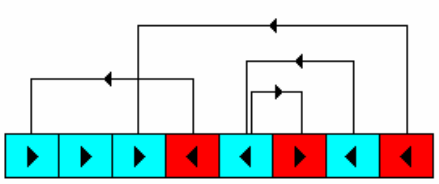
Le graphique suivant montre l'évolution de l'automate 110 sur des conditions initiales quelconques. On voit apparaître un comportement qui est à la frontière des classes deux et trois. Certaines structures, qu'on appelle glisseurs, se déplacent permettant les transferts et les échanges d'information lorsqu'elles entrent en collision. Cette observation est fondamentale car pour qu'un système calculatoire puisse prétendre à l'universalité il faut non seulement qu'il soit capable d'émuler toutes les portes logiques, $\wedge,\vee,\overline{\wedge},\cdots$, mais il faut encore que l'information circule de façon contrôlée au sein de l'automate afin de pouvoir adresser le contenu de n'importe quel groupe de cellules, de les recopier puis de les retrouver le moment venu. Il va de soi que le graphique ci-dessous porte le temps en ordonnée descendante. L'automate proprement dit est unidimensionnel et chaque coup d'horloge réactualise l'état de ses cellules. En Mathematica, l'instruction Manipulate[] permet de visualiser cette évolution.



110

Les machines à registres universelles.

Plus proches d'un langage machine moderne, les automates à registres fonctionnent sur base d'un très petit nombre d'instructions élémentaires. Dans une version minimaliste due à Wolfram, ils fonctionnent sur deux états seulement et ils n'utilisent que deux instructions, "Increment", et "Jump-Decrement". Voici, pour l'exemple, le diagramme et le programme d'une machine à, nr = 2, registres (bleu et rouge) et, $n_{instr} = 8$, huit instructions.



```
Programme : {i[1],i[1],i[1],d[2,1],d[1,6],i[2],d[1,5],d[2,3]},
```

Ces notations signifient :

i[s] = "incrémenter (= +1) le contenu du registre, s, et passer à l'instruction suivante"
d[s,instr] = "si le contenu du registre, s, est positif, le décrémenter (= -1) et sauter à l'instruction, instr, sinon le laisser à 0 et passer à l'instruction suivante ".

Initialement, les registres contiennent chacun un entier positif, éventuellement nul. Un pointeur est positionné initialement sur une instruction de départ, généralement la première et la machine suit alors les indications du programme. Ce programme ne comprend que deux types d'instructions, qui incrémentent ou décrémentent le contenu du registre qu'indique le pointeur. Toute incrémentation est suivie d'un passage automatique à l'instruction suivante tandis que toute décrémentation s'accompagne d'un saut que le programme spécifie explicitement. Les registres n'acceptent pas d'entiers négatifs : lorsqu'une instruction "décrément" est appliquée à un registre qui est déjà à 0, elle est tout simplement ignorée et le programme passe à l'instruction suivante. Ce dernier point est essentiel car c'est lui qui ouvre la porte à l'absence d'arrêt du calcul. On peut décréter l'arrêt lorsque plus aucune instruction n'est applicable ou encore lorsque le système entre dans un état fixe invariable. C'est bien ce qui se passe avec les systèmes d'exploitation qui ne contiennent pas d'instruction d'arrêt à proprement parler : c'est l'état invariable qui lorsqu'il est détecté est retourné comme valeur de sortie.

La machine citée en exemple ne possède que deux registres. Elle pourrait en contenir davantage mais le fait est que les machines à plusieurs registres ne sont pas fondamentalement plus puissantes que celles qui n'en comportent que deux, elles sont simplement plus rapides et plus faciles à programmer. Bien qu'elle ne soit pas universelle, la machine citée en exemple exhibe un comportement non trivial. Pour fixer les idées, voici le début de l'évolution du contenu de ses deux registres :

```
{0,0},{1,0},{2,0},{3,0},{3,0},{2,0},{2,1},{1,1},{0,1},{0,2},
{0,2},{0,1},{1,1},{1,0},{2,0},{3,0},{4,0},{4,0},{3,0},{3,1},…
```

Nos ancêtres du néolithique auraient parfaitement pu être les inventeurs de la machine à registres. Deux (ou plusieurs) caisses de couleurs différentes, initialement vides, et un tas illimité de cailloux leur auraient suffi. L'opération d'incrément consiste à ajouter un caillou dans la bonne caisse et celle de décrément consiste à en enlever un puis, dans les deux cas, à lire quelle instruction le programme prévoit pour la suite. Lorsqu' aucun caillou n'est présent dans la caisse concernée, on se contente de passer à l'instruction suivante du programme. On arrête le calcul lorsque le programme ne propose plus aucune instruction ou quand le système entre dans un état fixe. Nos ancêtres auraient ainsi appris à extraire la racine carrée par défaut d'un entier, n, en utilisant le programme suivant portant sur trois caisses initialement vides sauf la première qui contient n cailloux :

```
{d[1,4],i[1],d[1,15],i[2],d[1,6],d[1,11],i[1],d[2,7],d[3,7],d[1,15],d[3,4],
i[3],d[2,12],d[3,4]}.
```

Lorsque n=10, par exemple, l'évolution du nombre de cailloux dans les trois caisses est la suivante, qui converge bien vers un point fixe qui est la solution cherchée, 3 :

```
{10,0,0},{9,0,0},{9,1,0},{8,1,0},{7,1,0},{7,1,0},{7,1,1},{7,0,1},{7,0,2},{7
,0,2},{7,0,1},{7,1,1},{6,1,1},{5,1,1},{5,1,0},{5,2,0},{4,2,0},{3,2,0},{3,2,
0},{3,2,1},{3,1,1},{3,1,2},{3,0,2},{3,0,3},{3,0,3},{3,0,2},{3,1,2},{2,1,2},
{1,1,2},{1,1,1},{1,2,1},{0,2,1},{0,2,1},{1,2,1},{1,1,1},{2,1,1},{2,0,1},{3,
0,1},{3,0,1},{3,0,0},{4,0,0},{4,0,0},{4,0,0},{3,0,0},{3,0,0},{3,0,0},… .
```

Toute MT à s états et 2 couleurs (= caractères) peut être émulée par une machine à 3 registres. On obtient les règles de cette dernière à partir de celles de la MT grâce à l'instruction Mathematica, TMToRM, détaillée ci-dessous. Une instruction similaire existe lorsque le nombre de caractères de la MT excède 2 mais elle est plus compliquée et nous ne la reproduisons pas. De toutes façons, nous savons qu'il existe des MTU à deux caractères d'où il résulte qu'il est possible de construire une MR universelle. On voit que le jeu des instructions tout juste nécessaires pour prétendre à l'universalité, "Increment" et "Jump-Decrement", est incroyablement mince.

```
TMToRM[rules_]:=Module[{segs,adrs},segs=Map[TMCompile,rules];
adrs=Thread[Map[First,rules]→Drop[FoldList[Plus,1,Map[Length,segs]],-1]];
MapIndexed[#1/. {dr[r_,n_]:→d[r,n+First[#2]],dm[r_,z_]:→d[r,z/. adrs]}&,
                                                  Flatten[segs]]]
TMCompile[_→z:{_,_,1}]:=f[z,{1,2}]
TMCompile[_→z:{_,_,-1}]:=f[z,{2,1}]
f[{s_,a_,_},{ra_,rb_}]:=Flatten[{i[3],dr[ra,-1],dr[3,3],i[ra],i[ra],dr[3,-
2],If[a==1,i[ra],{}],i[3],dr[rb,5],i[rb],dr[3,1],dr[rb,1],dm[rb,{s,0}],dr[r
b,-6],i[rb],dr[3,-1],dr[rb,1],dm[rb,{s,1}]}]
```

Voici l'exemple de la machine à registres qui émule la MT 1971 à 2 états et 2 caractères :

```
TMToRM[{{1,1}→{1,1,1}, {1,0}→{2,1,-1}, {2,1}→{2,1,-1}, {2,0}→{1,1,1}}]
```

```
{i[3],d[1,1],d[3,6],i[1],i[1],d[3,4],i[1],i[3],d[2,14],i[2],d[3,10],d[2,13]
,d[2,19],d[2,8],i[2],d[3,15],d[2,18],d[2,1],i[3],d[2,19],d[3,24],i[2],i[2],
d[3,22],i[2],i[3],d[1,32],i[1],d[3,28],d[1,31],d[1,55],d[1,26],i[1],d[3,33]
,d[1,36],d[1,37],i[3],d[2,37],d[3,42],i[2],i[2],d[3,40],i[2],i[3],d[1,50],i
[1],d[3,46],d[1,49],d[1,55],d[1,44],i[1],d[3,51],d[1,54],d[1,37],i[3],d[1,5
5],d[3,60],i[1],i[1],d[3,58],i[1],i[3],d[2,68],i[2],d[3,64],d[2,67],d[2,19]
,d[2,62],i[2],d[3,69],d[2,72],d[2,1]}
```

L'initialisation de la MR doit spécifier la position initiale du pointeur et le contenu des trois registres. Dans le cas particulier de l'émulation d'une MT qui démarre dans l'état, 1, sur une bande initialement vierge, cette initialisation est notée, {1,{0,0,0}}. Si la bande de la MT n'est pas vierge, l'initialisation est un peu plus compliquée : on la trouvera dans l'ouvrage de Wolfram (page 1114).

Les systèmes arithmétiques universels.

Un système arithmétique est défini comme suit. On se donne un entier, p, et une table d'instructions qui associe une opération arithmétique élémentaire à tout entier, k, tel que, $0 \le k < p$. Voici l'exemple, particulièrement simple d'un système arithmétique ne possédant que, p = 2, instructions :

$$0 : x \to 3x/2 \qquad 1 : x \to (3x+1)/2$$

Dans Mathematica, les expressions du type $x \to 3x/2$ ou $x \to (3x+1)/2$ sont souvent abréviées dans une syntaxe ésotérique : 3#/2& et (3#+1)/2&. Le lecteur qui voudrait essayer les programmes qui suivent ne s'étonnera pas de les rencontrer.

Le principe du fonctionnement du système arithmétique est simple. On part d'un entier positif auquel on applique celle des instructions qui porte le numéro égal au reste de sa division par p. On poursuit cette manœuvre itérativement. L'exemple suivant aide à comprendre, qui part de l'entier, 1. Le programme calcule la suite des dix premiers itérés :

```
NestList[If[EvenQ[#],3#/2,(3#+1)/2]&,1,10]
```

```
{1,2,3,5,8,12,18,27,41,62,93, …}
```

Il existe une connexion entre les systèmes arithmétiques et les machines à registres. Dans l'exemple précédent, il suffit de considérer le contenu du deuxième registre lorsque le contenu du premier retombe de 1 à 0 pour retrouver la suite précédente. Ce n'est nullement une coïncidence. Etant donnée une machine à nr registres et n_{instr} instructions, on trouve le système arithmétique qui l'émule grâce à l'instruction, RMToAS, que l'on expérimente sur une MR particulière (Prime[n] est le $n^{ième}$ entier premier) :

```
RMToAS[prog_,nr_]:=
With[{ninstr=Length[prog],g=Product[Prime[j],{j,nr}]},{ninstr g,
Sort[Flatten[MapIndexed[With[{n=First[#2]-1},#1/.{i[r_]:>Table[n+j ninstr →
(1+n+Prime[r](-n+#)&),{j,0,g-1}],d[r_,k_]:>Table[n+j ninstr →
If[Mod[j,Prime[r]]==0,-1+k+(-n+#)/Prime[r]&,#+1&],{j,0,g-1}]}]&,prog]]}]
```

```
RMToAS[{i[1],d[2,1],i[2],d[1,3],d[2,1]},2]
```

```
{p=30,
{0,5,10,15,20,25} ->2n+1
{1,16} ->(n-1)/3
{2,7,12,17,22,27} ->3n-3
{3,13,23} ->(n+1)/2
{4,19} ->(n-4)/3
{6,8,9,11,14,18,21,24,26,28,29} ->n+1}
```

Ce système arithmétique possède, $p = n_{instr} \prod_{k=1}^{nr} Prime[k] = 5 \times 2 \times 3 = 30$ instructions que l'on a numérotées de 0 à 29 et que l'on a regroupées pour gagner un peu de place. Si la machine à registres démarre à l'instruction, n_0, 1 la plupart du temps, avec la liste d'entiers, reg, dans ses nr registres, le système arithmétique démarre avec l'entier,

$$n_{init} = n_0 + \text{Product[Prime[i]\^{}reg[[i]],\{i,nr\}]}\, n_{instr} - 1.$$

On calcule l'évolution du système arithmétique en calculant le reste de la division de n_{init} par p puis en lui appliquant l'instruction qui porte le numéro d'ordre correspondant à ce reste. On poursuit le processus itérativement. On programme le calcul comme suit :

```
ASEvolveList[{p_,rules_},init_,t_]:=NestList[(Mod[#,p]/. rules)[#]&,init,t]
```

On peut décoder l'évolution du système arithmétique à chaque pas et trouver l'état correspondant de la machine à registre, sous la forme habituelle, {instr,{reg}}, grâce à l'instruction :

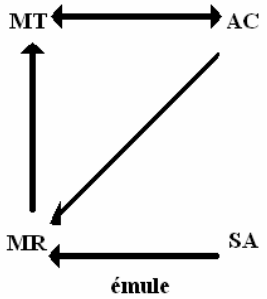
```
{Mod[m,p]+1,Map[Last,FactorInteger[Product[Prime[i],{i,nr}]
Quotient[m,p]]]-1}
```

Elle décode le nombre obtenu sous la forme, $i + n_{instr} \prod_{k=1}^{nr} Prime[k]^{reg[[k]]} - 1$, qui permet de retrouver la position, i, du pointeur de la MR, ainsi que le contenu, reg[[k]], de chaque registre. L'arrêt peut être imposé lorsque ce nombre est tel que, $(reg[[nr]] = 0) \wedge (i = nr)$, ce qui correspond à l'arrêt naturel de la MR qui a épuisé ses instructions.

Il suffit d'appliquer la procédure décrite à une MR universelle pour en déduire la table d'instructions d'un SA universel. En conclusion, toute suite calculable, donc toute fonction totale de N dans N, peut l'être par un SA particulier correctement initialisé ou si l'on préfère, par un SA universel. Qu'il existe des systèmes arithmétiques universels peut surprendre mais à y regarder de plus près, on s'apercevra que toutes les opérations qu'effectue un ordinateur s'y ramènent. Nous verrons dans la dernière partie que ce dernier point est en relation directe avec le fait que l'arithmétique correctement axiomatisée est également universelle dans un autre sens attribué à Gödel.

Emulations entre systèmes universels.

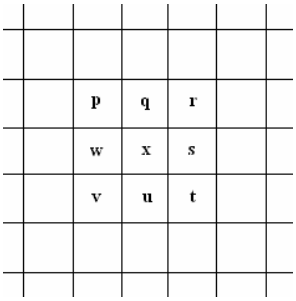
L'existence des fonctions TMToCA, RMToCA, CAToTM, TMToRM et RMToAS peut être reportée sur le diagramme d'émulation suivant :



L'existence d'une MTU implique l'existence d'un automate cellulaire universel, ACU, et d'une machine à registre universelle, MRU, qui implique, à sont tour, l'existence d'un système arithmétique universel, SAU.

Les systèmes universels exotiques.

Le jeu de la vie (Life).
Il existe des automates universels multidimensionnels. L'automate cellulaire bidimensionnel universel le plus étudié est le jeu de la vie (Life) de Conway.



Les règles de l'automate sont simples : initialement toutes les cellules du réseau sont dans l'état OFF (=0) sauf quelques unes qui sont ON (=1). A chaque coup d'horloge, une cellule OFF le reste sauf si 3 de ses huit voisines sont ON et une cellule ON ne le reste que si 2 ou 3 de ses huit voisines sont ON. La position exacte des cellules ON qui entourent la cellule considérée n'a aucune importance, seule leur nombre compte. Un automate qui met toutes ses cellules voisines sur un pied d'égalité est dit totalistique. Parmi tous les automates totalistiques, une numérotation due à Wolfram confère le numéro 224 à Life :

```
GameOfLife={224,{2,{{2,2,2},{2,1,2},{2,2,2}}},{1,1}};
```

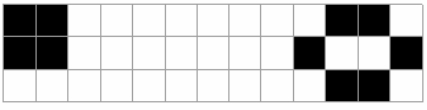
Il est possible d'écrire une fonction logique qui calcule à tout instant le nouvel état, x, d'une cellule donnée. La nouvelle valeur de x est donnée par :

$$x_{new} = (sum == 3) \vee ((sum == 2) \wedge (x_{old} == 1)) \qquad (sum = p + q + r + s + t + u + v + w)$$

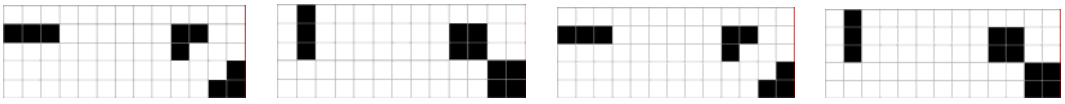
Une littérature abondante a été consacrée à Life. La version 6 de Mathematica inclut un arsenal complet d'instructions qui permettent d'en étudier le comportement dans des conditions de confort optimal. Un logiciel freeware très bien développé existe, nommé Golly.

Certaines structures uniquement composées de cellules isolées s'évanouissent trivialement dès le premier coup d'horloge. D'autres, plus compliquées, finissent par s'évanouir après des temps plus longs. Enfin, d'autres ne s'évanouissent pas : elles oscillent ou elles croissent ou elles se répliquent, etc, tous les cas de figures existent comme il sied à une système universel. Voici deux structures autonomes invariables, que l'on a regroupées pour gagner de la place :

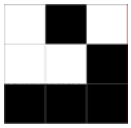
```
Fixe1Fixe2={{1,1},{1,2},{2,1},{2,2},{1,11},{1,12},{2,10},{2,13},{3,11},{3,1
2}};
Manipulate[ArrayPlot[CellularAutomaton[GameOfLife,{SparseArray[Fixe1Fixe2-
>1],0},300][[k]],Mesh→True],{k,1,10,1}]
```



Voici trois pas d'évolution de deux structures autonomes périodiques (de période 2), que l'on a également regroupées sur un même damier :

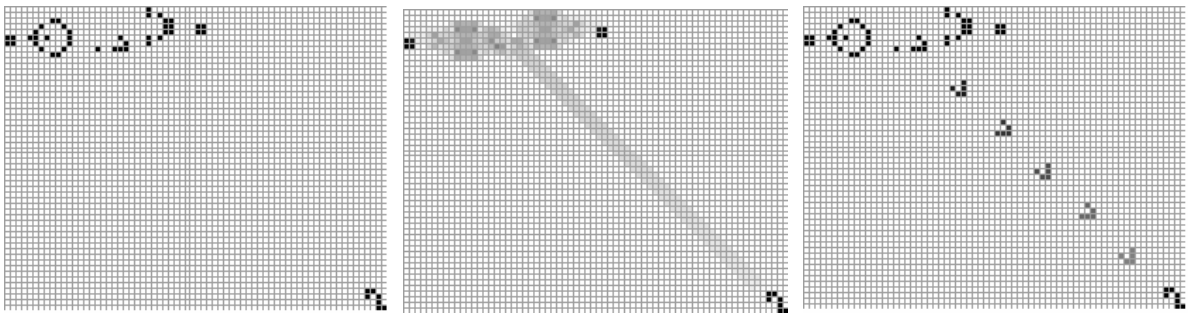


D'autres oscillateurs existent dont la période est supérieure à 2. Certaines structures, appelées glisseurs, se déplacent à vitesse constante dans le plan du réseau. Le glisseur le plus simple comporte 5 cellules correctement agencées :



Les figures suivantes montrent dans leur coin supérieur gauche un oscillateur de période 30, baptisé canon parce qu'il émet un glisseur tous les 30 coups d'horloge. Dans le coin inférieur droit, on a placé une structure invariable, baptisée absorbeur, parce qu'il attend que les glisseurs émis à la file indienne l'atteignent après quoi il les absorbe. La figure de gauche affiche l'état initial du système, celle du milieu affiche en niveaux de gris le taux moyen d'occupation des cellules du réseau et celle de droite est une vue stroboscopique de cette évolution moyenne prise à la fréquence 1/30. Mathematica permet également de suivre les évolutions temporelles dans Life grâce à l'instruction Manipulate décrite dans la documentation.

```
EaterplusGun={{1,26},{2,26},{2,28},{3,9},{3,10},{3,29},{3,30},{4,7},{4,11},
{4,29},{4,30},{4,35},{4,36},{5,6},{5,12},{5,29},{5,30},{5,35},{5,36},{6,1},
{6,2},{6,5},{6,6},{6,8},{6,12},{6,21},{6,26},{6,28},{7,1},{7,2},{7,6},{7,12
},{7,22},{7,26},{8,7},{8,11},{8,17},{8,20},{8,21},{8,22},{9,9},{9,10},{51,5
1+14},{51,52+14},{52,51+14},{52,53+14},{53,53+14},{54,53+14},{54,54+14}};

ArrayPlot[CellularAutomaton[GameOfLife,{SparseArray[EaterplusGun
->1],0},300][[k]],Mesh→True]

ArrayPlot[Mean[CellularAutomaton[GameOfLife,{SparseArray[EaterplusGun
->1],0},300]],Mesh->True]

ArrayPlot[Mean[CellularAutomaton[GameOfLife,{SparseArray[EaterplusGun
->1],0},300][[Table[k,{k,1,300,30}]]]],Mesh->True]
```



Les glisseurs sont l'ingrédient de base qui permet l'universalité au sens de Turing au travers des échanges contrôlés de l'information entre les diverses sous structures de l'automate. Ils étaient déjà présents dans l'automate 110 sous une forme moins directement apparente vu le caractère unidimensionnel de cette règle. Plusieurs canons correctement agencés génèrent des glisseurs susceptibles d'entrer en interaction constructive ou destructive. Il est possible de construire sur ces bases à peine esquissées des structures qui simulent toutes les portes logiques nécessaires à la construction d'un système calculatoirement universel. Des amateurs passionnés perfectionnent sans cesse les structures produites et des travaux récents de Chapman proposent une construction effective d'une machine à registre universelle.

Puisque ce système est universel, il est fatalement soumis à l'indécidabilité du problème de l'arrêt. Alors que l'indécidabilité qui frappe les automates universels unidimensionels, en particulier la règle 110, ne peut concerner que leur comportement à long terme, celle qui frappe les automates multi dimensionnels concerne toutes les propriétés non triviales de l'automate. Par exemple, dans le cas de Life, étant donnée un motif initial quelconque de cellules 'ON', il n'existe pas de MT capable de prédire s'il va déboucher sur une extinction, sur une structure figée ou périodique, sur une réplication ou sur une évolution ininterrompue.

Fractran.
Fractran est un système arithmétique universel anecdotique qui fonctionne selon un principe original également dû à Conway. Il est particulièrement simple : on se donne, en entrée, un entier, N_0, et une liste finie de fractions rationnelles, $f_1, f_2, \cdots, f_k$, auxquels on fait subir le traitement itératif suivant :

- multiplier N_0 par les f_i dans l'ordre où ils se présentent jusqu'à tomber sur un entier, N_1, qui prend la place de N_0
- recommencer la même opération à partir de N_1
- poursuivre tant que la manœuvre réussit. Lorsqu'elle échoue parce qu'aucun produit n'est entier, arrêter.

L'exemple suivant est célèbre :

$$N_0 = 2 \qquad f_i = \{\frac{17}{91}, \frac{78}{85}, \frac{19}{51}, \frac{23}{38}, \frac{29}{33}, \frac{77}{29}, \frac{95}{23}, \frac{77}{19}, \frac{1}{17}, \frac{11}{13}, \frac{13}{11}, \frac{15}{2}, \frac{1}{7}, \frac{55}{1}\}$$

engendre la suite, 2, 15, 825, 725, 2275, 425, 390, 330, 290, 770, …, qui fonctionne, à sa manière, comme un énumérateur de nombres premiers : tous les termes de la suite qui sont puissances de deux affichent un exposant premier, soit dans l'ordre, $\{2^2, 2^3, 2^5, 2^7, 2^{11}, 2^{13}, 2^{17}, \cdots\}$. Ici encore on ne se préoccupe pas de la rapidité du calcul qui en l'occurrence nécessite environ 8200 itérations pour n'obtenir ces 7 nombres premiers.

Toute suite qui liste les valeurs prises par une fonction calculable de N vers N peut être programmée de cette façon à partir d'un entier, N_0, et d'une liste finie de fractions. On note que la présence de l'instruction "Tant Que" confronte cette méthode de calcul au problème de l'arrêt comme il se doit pour un système qui revendique l'universalité calculatoire.

Fractran est difficile à programmer. A part l'exemple cité, dû à Conway, on ne connaît que très peu de programmes effectuant une tâche non triviale.

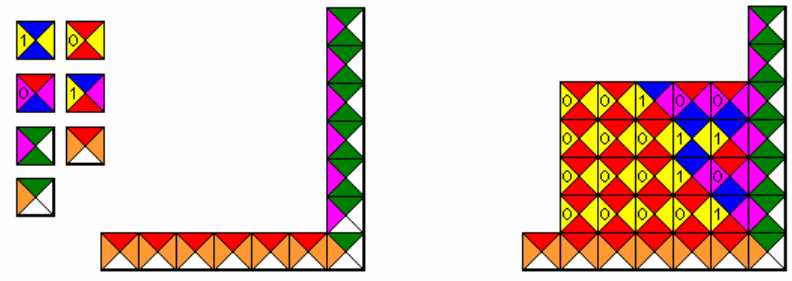
Les tuiles de Wang.
Une classe célèbre de problèmes indécidables concerne le pavage du plan. Rappelons que étant donné un jeu de tuiles en nombres illimités, il n'existe pas de MTU capable de décider s'il est possible de paver le plan au delà de toute limite avec ces tuiles.

L'origine de ce problème provient du fait, pas vraiment évident, qu'il existe des pavages non périodiques du plan dont les premiers furent découverts par Penrose. En effet, avec un pavage périodique, il est facile de deviner si l'on a quelques chances d'aboutir. Par contre plus rien n'est clair si le pavage est apériodique.

L'indécidabilité de ce problème laisse espérer que l'on puisse construire un système calculatoire universel sur le principe du pavage. C'est effectivement ce que Wang a réalisé de la manière suivante. On se propose de paver un quart de plan infini à l'aide de tuiles. Pour ce faire, on dispose d'une réserve illimitée de tuiles qui respectent un des k gabarits autorisés. Certaines tuiles sont exclusivement réservées aux bords du puzzle : on distingue les tuiles qui incarnent le programme et qui s'alignent obligatoirement selon l'axe vertical et celles qui incarnent les données à disposer le long de l'axe horizontal. L'orientation est imposée.

A l'aide d'un codage binaire par exemple, on marque les tuiles qui vont devoir s'adapter dans le champ ainsi défini en respectant les règles du jeu de dominos. On attend des motifs qui apparaissent dans le puzzle qu'ils calculent la fonction définie par le programme. Un exemple, emprunté à Winfree, fera mieux comprendre comment le système fonctionne.

On se propose de construire un compteur binaire. Pour ce faire, on se donne un jeu de sept tuiles différentes dont quatre seront destinées à paver le champ. Celles-ci sont marquées par un des symboles binaires, 0 ou 1. Les trois autres pièces sont respectivement la pièce d'angle, les pièces définissant le programme et les pièces définissant les données. Les bords dessinés en trait gras permettent leur identification sans ambiguïté.



On constate qu'une fois les bords mis en place, le puzzle ne possède qu'une seule solution qui fait apparaître le comptage binaire demandé.

On peut généraliser cet exemple et calculer de cette manière tout ce qui est calculable. On peut convenir qu'une procédure de construction s'arrête lorsque plus aucune pièce du puzzle ne peut être ajoutée. Evidemment l'arrêt est indécidable en toute généralité.

Il se pourrait que l'unicité de la solution ne soit pas garantie ce qui rendrait le système non déterministe. Loin d'être un inconvénient, cette circonstance est au contraire un atout dans le cas où on envisagerait un calcul massivement parallèle. Cette remarque prend tout son poids quand on réalise que ce modèle est proche du principe de l'ordinateur ADN. Bien que celui-ci ne soit pas plus opérationnel que l'ordinateur quantique, des recherches intensives sont déployées partout dans le monde pour tenter de progresser dans cette direction.

Cette architecture d'un genre nouveau entend tirer parti du fait que les brins complémentaires d'ADN, enroulés en double hélice, ne peuvent s'assembler que si les bases, A, C, T et G, qui jouent le rôle de tuiles, se présentent les unes aux autres dans la bonne configuration, l'adénine ne s'appariant qu'à la thymine (A-T) et la guanine à la cytosine (G-C). On peut en théorie synthétiser et manipuler des chaînes d'ADN au moyens d'enzymes et isoler des brins qui présentent des successions connues de bases.

Les premiers essais d'utilisation du concept ADN à des fins calculatoires ont été l'œuvre d' Adleman en 1994. Il a montré comment résoudre le problème du voyageur de commerce sur base moléculaire et il a illustré la méthode sur l'exemple simple de 7 villes à visiter le long d'un trajet de longueur minimale. Ce problème dont on pense qu'il n'est pas soluble en un temps raisonnable par un ordinateur classique dès l'instant où le nombre de villes excède la centaine pourrait trouver ici une réponse rapide. L'idée consiste à coder chaque ville par une séquence de disons huit bases, par exemple, Marseille = 'AGTTAGCA', Lyon='GAAACTAG', etc, puis à coder chaque trajet reliant deux villes par la séquence complémentaire de celle qui sélectionne les quatre dernières bases d'une ville et les quatre premières de l'autre, soit, Marseille-Lyon='TCGTCTTT'. Lorsque les séquences ont été bien choisies afin d'éviter toute ambiguïté, il "suffit" de mettre en présence une immense quantité de brins reprenant toutes les villes et trajets possibles. Après agitation prolongée, on peut espérer que les brins vont se recombiner comme le feraient des Lego. On n'est sûr que d'une chose : chaque chaîne reconstituée correspond à un trajet possible quoique éventuellement incomplet. Une technique appropriée de filtrage permet de sélectionner l'assemblage le plus long qui, en théorie et avec un peu de chance, correspond à la solution cherchée.

D'autres courants de recherche existent dans le même domaine. Une avancée récente (Stojanovic et Stefanovic, 2002), implémente un automate cellulaire, baptisé Maya, dans une boîte à compartiments, chacun étant rempli d'une préparation enzymatique bien choisie qui est capable de réagir avec des fragments spécifiques d'ADN. En choisissant bien les préparations, on peut montrer qu'il est possible d'émuler, de cette manière, les différentes portes logiques. En travaillant sur une boîte carrée à 9 compartiments, les auteurs ont réussi à programmer le jeu de Morpion. A condition de le laisser commencer, Maya ne perd jamais. Il commence par choisir la cellule centrale, comme la théorie le recommande, puis le joueur humain lui indique sa réponse en ajoutant dans chacune des 8 cellules restantes une solution contenant un fragment d'ADN particulier, celui qui code le coup qu'il veut jouer, parmi les huit possibles. La réaction chimique entre l'ADN et la préparation présente dans la cellule adéquate y produit une légère fluorescence qui indique à Maya le coup qu'il doit jouer à son tour.

L'émulation calculatoire entre systèmes physiques.

Le principe de l'émulation calculatoire s'étend sans surprise aux systèmes physiques. Cette extension repose sur les observations suivantes.

Lorsqu'un système inanimé non chaotique est préparé dans un état initial donné et qu'on se propose de connaître ses états ultérieurs, deux stratégies sont envisageables. La première est expérimentale et elle consiste simplement à observer l'évolution naturelle du système puis à *mesurer* les états ultérieurs demandés. Cette stratégie opère en temps réel et la précision des résultats est entièrement dépendante de la précision des mesures effectuées. Elle repose, en définitive, sur le principe que,

"Tout système physique calcule sa propre évolution en temps réel".

L'autre stratégie est théorique, elle consiste à *calculer* en temps virtuel l'évolution du système avec l'aide d'un système auxiliaire qui est capable d'en mimer le comportement. Cette fois, la précision des résultats est entièrement dépendante de la précision avec laquelle on assure l'état initial. L'hypothèse faite que le système n'est pas chaotique garantit qu'aucune instabilité n'affecte dramatiquement la conduite des calculs.

Cette stratégie en temps virtuel est envisageable de deux manières distinctes : l'une est basée sur l'émulation spécifique par un système particulier, calculatoirement équivalent au système donné, et l'autre a recours à un système capable d'émulation universelle.

1- Le principe d'émulation physique spécifique.

L'émulation physique définit le principe de l'équivalence de classes :

"Deux systèmes appartiennent à une même classe d'équivalence calculatoire s'ils sont capable de s'émuler mutuellement".

Par exemple, deux pendules sans frottements, l'un de torsion et l'autre simple, peuvent s'entre émuler car ils sont tous les deux de classe deux. On sait qu'un pendule de torsion sans frottement, partant au repos de l'élongation, $\pi/3$, oscille selon la loi analytique,

$$\theta_{osc} = \frac{\pi}{3} cos(\frac{2\pi}{T_{osc}} t)$$

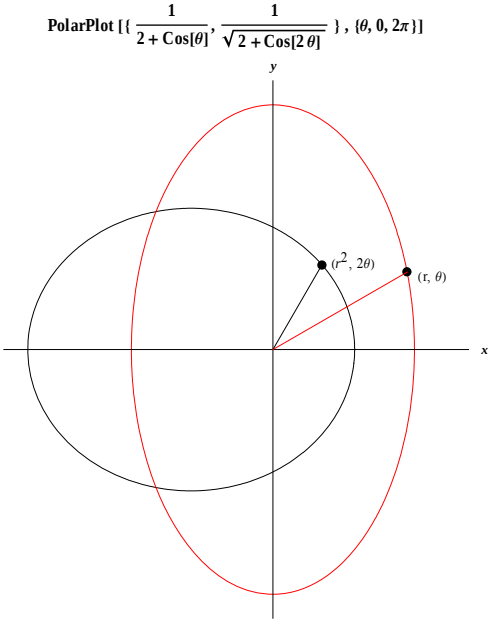
Un pendule simple, partant des mêmes conditions initiales, évolue selon une loi un peu plus compliquée à écrire du fait qu'elle fait intervenir des fonctions elliptiques, F et am :

$$\theta_{pend} = 2am[F(\frac{\pi}{6},4)(\frac{4t}{T_{pend}}+1),4]$$

Chaque pendule est, en fait, émulable par l'autre pris dans n'importe quelle configuration initiale. Il suffit, en effet, d'éliminer le temps entre les équations horaires pour connaître la position de l'un quand on connaît celle de l'autre. Ici, pour faire simple, on a choisi des conditions initiales similaires (vitesse initiale nulle) et des périodes identiques :

$$\theta_{pend} = 2am[F(\frac{\pi}{6},4)(\frac{2}{\pi}\arccos\frac{3\theta_{osc}}{\pi}+1),4] \quad \Leftrightarrow \quad \theta_{osc} = \frac{\pi}{3}\cos\left[\frac{\pi}{2}\frac{F(\frac{\theta_{pend}}{2},4)-F(\frac{\pi}{6},4)}{F(\frac{\pi}{6},4)}\right]$$

En ne quittant pas la classe deux, on peut aussi entre émuler deux systèmes hamiltoniens portant sur le même nombre de variables en cherchant une transformation canonique qui fait passer de l'un à l'autre. L'exemple classique est celui de l'émulation réciproque d'un oscillateur harmonique isotrope et d'un problème de Képler avec un centre fixe. La figure suivante montre, à une échelle arbitraire, les trajectoires correspondantes en coordonnées polaires, (r, θ) pour l'oscillateur harmonique (courbe rouge) et, (ρ, φ), pour le problème Képlérien (courbe noire). Il est bien connu que les trajectoires sont des ellipses rapportées soit au centre (oscillateur harmonique) soit à un foyer (mouvement Képlérien). On passe d'une trajectoire à l'autre par le changement de variables, $\rho = r^2$, $\varphi = 2\theta$.



Mouvement Képlérien. Oscillateur harmonique isotrope.

Il est théoriquement possible de généraliser le principe de l'émulation spécifique à des systèmes présentant des nombres différents de degrés de liberté mais les codages qui entrelacent les chiffres significatifs correspondant à des dimensions différentes sont plus artificiels.

Le principe d'émulation spécifique est d'un emploi plutôt rare. Personne ne passe volontiers son temps à construire un système qui n'aurait d'autre fonction que celle d'en émuler un autre alors qu'il existe des systèmes qui sont capables d'émulation universelle. C'est pourtant ce que l'on fait dans une certaine mesure lorsqu'on utilise une calculette de poche : celle-ci suffit à calculer l'évolution des systèmes simples, tels un oscillateur harmonique, puisque les fonctions trigonométriques y sont incorporées. La démocratisation de l'ordinateur calculatoirement universel condamne cependant à terme ces auxiliaires dépassés.

2- Le principe d'émulation physique universelle.

" L'ordinateur moderne, pourvu d'une mémoire potentiellement illimitée, est un système physique capable d'émuler, en temps virtuel, le comportement de n'importe quel système physique particulier. "

Les systèmes physiques qui sont capables d'émuler tous les autres sont également dits universels au sens de Turing. L'inventaire des systèmes physiques possédant cette propriété n'est pas fait. A ce jour, aucun système "naturel" n'a été démontré universel. L'ordinateur est le seul exemple connu mais sa construction requiert l'intervention de l'intelligence humaine et il n'est pas connu si l'intelligence artificielle peut rivaliser avec elle. En s'évadant du domaine de la physique pour celui de la biologie, il est clair que le cerveau humain en pleine possession de ses moyens et assisté d'une quantité potentiellement illimitée de papier brouillon, est spontanément universel. Il est équivalent à une MTU alors que, par comparaison, le cerveau d'un Bonobo est sans doute équivalent à une MT non universelle quoique d'un bon niveau.

Le principe d'irréductibilité calculatoire.

Le but avoué de toute théorie physique est de prédire l'évolution des systèmes inanimés sans avoir à en suivre l'évolution en temps réel. Cela revient à trouver une formule compacte qui court-circuite le calcul détaillé de cette évolution en donnant immédiatement accès à la réponse finale. C'est bien ce que la mécanique classique réussit lorsqu'elle prédit la position des astres dans le ciel. Mais les systèmes astronomiques non chaotiques sont assimilables à des systèmes de classe 2 et il ne faut pas chercher ailleurs les raisons de ce succès.

Toute prédiction qui ambitionne de trouver une formule compacte nécessite de disposer d'un système de calcul plus puissant que le système qu'on cherche à prédire. Les scientifiques ont toujours pensé que cela ne posait pas de problème de principe et que tout système pouvait toujours être analysé dans un cadre plus général. Mais c'est précisément cette idée que combat le principe d'irréductibilité calculatoire :

"Il est inutile d'espérer court-circuiter le calcul de l'évolution d'un système qui atteint le seuil d'universalité calculatoire car aucun système calculatoire n'est plus puissant qu'un système qui a atteint le seuil d'universalité".

En résumé on distingue trois comportements des systèmes sur le long terme :

- Les systèmes de classes un et deux sont couramment étudiés précisément parce qu'ils sont solubles au terme d'un raccourci calculatoire. A ce titre, ils sont totalement prédictibles.

- Les systèmes de classe trois ne sont universels qu'au sens de Borel. Ils ne sont pas prédictibles, en toute généralité, pour cause de sensibilité aux conditions initiales. Il "suffirait" cependant de connaître leurs conditions initiales avec une précision infinie pour les rendre prédictibles. C'est très exactement ce qui se passerait si ces conditions initiales étaient

des réels constructibles, $\sqrt{2}$, par exemple. Dans de tels cas, Mathematica, pourvu d'une mémoire illimitée, serait parfaitement capable de calculer leur évolution exacte jusqu'au temps, t, quelconque mais donné. Le seul prix serait un allongement probablement au moins exponentiel de l'espace mémoire requis et du temps de calcul. Par exemple, le calcul des trajectoires dans un billard convexe dont la frontière est une courbe algébrique peut toujours être effectué en restant dans le domaine des nombres algébriques, c'est une conséquence de l'existence d'une procédure de décision pour la géométrie élémentaire. Evidemment, les réels constructibles ne forment qu'un ensemble de mesure nulle dans l'ensemble des réels et c'est précisément parce qu'on a pris l'habitude de travailler sur l'ensemble des réels que le problème de la sensibilité aux conditions initiales se pose avec tant d'acuité.

- Les systèmes universels au sens de Turing se situent à la frontière qui sépare la classe deux de la classe trois. Ces systèmes sont soumis à une imprédictibilité essentielle d'un genre nouveau : même sur conditions initiales parfaitement précisées, il n'existe aucune procédure effective capable de décider leur comportement à long terme. Pour ces systèmes, il n'existe, en toute généralité, aucun raccourci calculatoire possible et il n'est d'autre ressource que de les laisser évoluer en temps réel puis d'en mesurer l'état final. Les systèmes universels ne peuvent pas davantage s'auto court-circuiter. Insistons une dernière fois sur le fait que l'évolution de ces systèmes est parfaitement prédictible pour un grand nombre de conditions initiales, c'est uniquement le problème général qui est frappé d'indécidabilité.