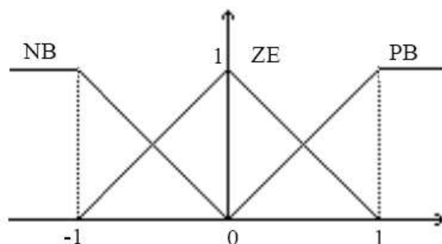
Chapitre 8

Applications en intelligence artificielle

Beaucoup d'exemples des chapitres précédents traitent déjà d'intelligence artificielle, résolution de problèmes, planification, systèmes-experts, traitement de la langue naturelle… Nous présentons ici d'autres thèmes, la logique floue avec l'algorithme de Sugeno pour la commande floue, des algorithmes d'évolution liés au hasard et un réseau de neurones à une couche cachée.

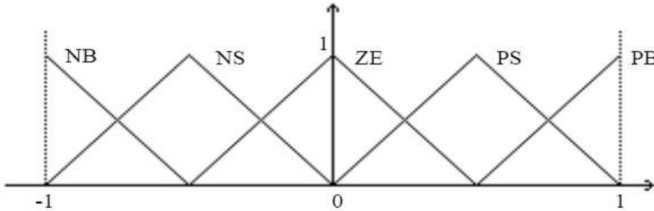
LA COMMANDE FLOUE

Pour représenter graduellement le fait de vérifier des prédicats vagues tel que « être jeune », « être grand », « *X* et *Y* sont similaires », « *X* est nettement plus petit que *Y* », etc., on se sert d'ensembles flous. Un ensemble flou est la donnée d'une fonction d'appartenance, c'est-à-dire une fonction définie sur l'univers dont on parle, ce qui signifie là où sont définies les variables par exemple, une température, l'âge d'une personne, la vitesse d'un train sur une voie, et à valeurs dans [0, 1]. Ainsi « vitesse élevée » pourrait être réalisée au maximum avec le degré 1 passé 120 km/h, pas du tout réalisée avec le degré 0 en dessous de 90 km/h et graduellement de façon continue et linéaire entre les deux. Une solution commode consiste à prendre de telles fonctions d'appartenance par un jeu de contraction-dilatation, qui vont se ramener toujours à l'intervalle [–1, 1] dans lequel on notera *NB*, *ZE*, *PB* les appellations négatif, zéro, positif.



Au vu de ces schémas, les clauses déterminant le niveau d'appartenance *D* d'une variable *X* à ces trois prédicats flous sont évidentes en Prolog. Naturellement, il est possible, et les applications ne manquent pas, de définir d'autres prédicats,

en plus grand nombre et avec des formes différentes construits avec des courbes gaussiennes ou autres. Généralement il s'agit de *PB* = *positive big*, *NS* = *negative small*, etc.



Nous définissons en conformité avec la première famille de prédicats, *positif, negatif* et *zero*.

```
positif(X, 1) :- 1 < X, !.
positif(X, 0) :- X < 0, !.
positif(X, X).
negatif(X, 1) :- X < –1, !.
negatif(X, 0) :- 0 < X, !.
negatif(X, D) :- D is –X.
zero(X, 0) :- (X < –1; 1 < X), !.
zero(X, D) :- D is 1 – abs(X).
```

Exemples

positif(0.8, *D*). → *D* = 0.8
positif(–0.3, *D*). → *D* = 0
negatif(–0.3, *D*). → *D* = 0.3
negatif(–3.14, *D*). → *D* = 1
zero(–0.1, *D*). → *D* = 0.9

La commande floue consiste à formaliser des règles que l'on pourrait énoncer pour décrire des situations types telles que « si la vitesse est élevée et le feu rouge proche, alors freiner fortement ». En donnant des sens précis à de tels prédicats, en ajustant certains paramètres et en comparant différents jeux de règles, il se trouve que le mécanisme de la commande floue exposé ci-dessous rend des services là où l'automatique classique le fait plus difficilement.
L'algorithme de Sugeno s'exprime très simplement. Il s'agit de régler la valeur d'une grandeur physique *U* destinée à atteindre ou maintenir une « consigne » dans le langage de l'automatique. Pour simplifier, supposons qu'avec deux entrées *X* et *Y* relevées en temps réel, quelques règles du type « si *X* est *A* et *Y*

est B alors $U = c$ » suffisent à décrire des situations approximatives rendant compte de certains états du problème. Dans la mesure où A et B sont des prédicats flous et où il y a chevauchement des règles, un petit nombre de règles peut traduire la décision U à prendre par consensus.
Plus précisément, supposons trois règles :

si X est *zéro* et Y est *négatif* alors $U = 1$
si X est *zéro* et Y est *zéro* alors $U = 2$
si X est *positif* et Y est *positif* alors $U = 3$

Chacun des prédicats flous *negatif, zero, positif* est plus ou moins vérifié par les deux entrées X et Y. Moyennant une transformation, *positif* peut vouloir dire « vitesse élevée » ou par exemple *zero* peut signifier à la fois « faible vitesse » ou « accélération plus ou moins nulle » si X mesure la vitesse, Y mesurant l'accélération-décélération.
La conjonction « et » sera interprétée comme la plus petite valeur entre les deux degrés de satisfaction des prédicats, ce qui donnera un degré de satisfaction de la règle. C'est la conjonction de Zadeh. L'algorithme consiste alors à ce que la sortie U soit définie comme la moyenne des conclusions des règles, pondérée par les niveaux de satisfaction de ces règles.
Par exemple la première règle va déterminer le degré DX d'appartenance de X à l'ensemble flou *zero,* puis de même, le degré DY de satisfaction de Y au prédicat *negatif,* enfin M sera la conjonction par *min* de ces deux degrés. M sera donc le degré de satisfaction de la règle pour les entrées X et Y.
Les trois règles auront ainsi $R1$, $R2$ et $R3$ comme degrés, coefficients qui serviront à calculer la moyenne pondérée U, constituant la sortie du contrôleur.

Avec le prédicat *min* déjà connu, on pourra programmer :

```
min(A, B, A) :- A < B, !.
min(_, B; B).

regle1(X, Y, 1, M) :- zero(X, DX), negatif(Y, DY), min(DX, DY, M).
regle2(X, Y, 2, M) :- zero(X, DX), zero(Y, DY), min(DX, DY, M).
regle3(X, Y, 3, M) :- positif(X, DX), positif(Y, DY), min(DX, DY, M).

commande(X, Y, U) :- regle1(X, Y, U1, R1), regle2(X, Y, U2, R2),
                     regle3(X, Y, U3, R3),
                     moyenne(U1, R1, U2, R2, U3, R3, U).

moyenne(U1, R1, U2, R2, U3, R3, U)
                     :- U is (U1*R1 + U2*R2 + U3*R3) / (R1 + R2 + R3).
```

Comme on peut le vérifier, si $X = 0.2$ et $Y = -0.6$, la première règle est vérifiée à 0.6, la seconde à 0.4 et la troisième à 0, la moyenne pondérée des conclusions est donc 1.4, dans le second cas, pour la même entrée, X et $Y = 0.5$, alors U est (2*0.5 + 3*0.2) / (0.5 + 0.2).

```
commande(0.2, –0.6, U). → U = 1.4
commande(0.2, 0.5, U). → U = 2.285714285714286
```

Naturellement, les systèmes opérationnels possèdent un peu plus de règles, mais souvent assez peu. Par ailleurs, s'il y avait davantage de règles, il serait possible de procéder à une programmation plus élégante, notamment en utilisant la moyenne des valeurs dans une liste.
L'algorithme de Sugeno est contemporain de celui de Mamdani où les conclusions sont données par des prédicats flous, ce qui, informatiquement, est un tout petit peu plus difficile à formaliser et à implémenter.

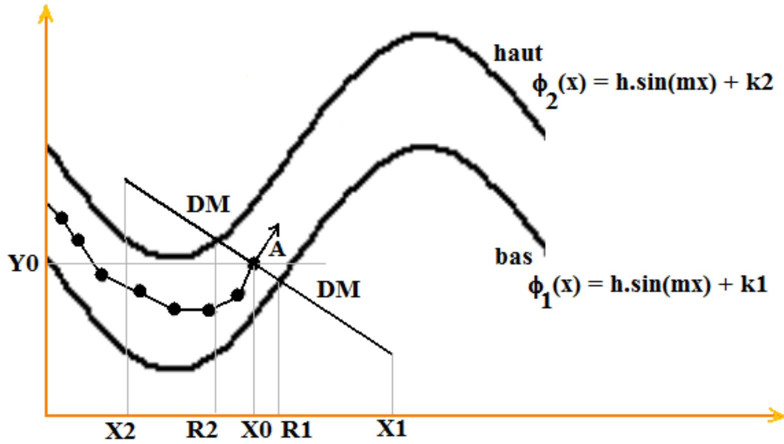
La philosophie de ces algorithmes est que pour des applications pratiques telles un mélangeur de douche où on doit régler deux débits pour maintenir une température de consigne, le chevauchement de règles floues fait qu'il n'y a pas de conflit entre règles mais une « moyenne » en sortie, autrement dit, un consensus. Par ailleurs, la mise au point de ces systèmes et leur rapidité, les rendent plus faciles et robustes que les systèmes d'automatique classique, dans des applications difficilement mathématisables.
Voir [Louis Gacôgne, *Eléments de logique floue*, Hermès, 1997]

Application à un suivi de route par un robot simulé

On va simuler un robot muni de deux capteurs de distances dans son cheminement dans un couloir sinusoïdal.
Nommons *bas* et *haut* deux courbes dont les équations sont du type :

$$\phi(x) = h*\sin(mx) + k$$

A chaque étape du trajet, le robot se trouve en une position ($X0$, $Y0$) avec une direction mesurée par l'angle A. Le robot doit évaluer les deux distances qui le séparent des parois (les deux courbes), ces distances étant mesurées perpendiculairement à sa direction.
Notons $R1$ et $R2$ les abscisses des intersections de la normale du robot avec les courbes du bas et du haut.



Grâce à un paramètre *DM* représentant la distance maximale de reconnaissance, les distances du robot aux deux courbes perpendiculairement à sa direction sont ramenées à [0, 1], ce sont les deux entrées *E*1 et *E*2 d'un contrôleur flou que l'on peut ici prendre très simple.

$$zero(E1) \to DA = \pi/6$$
$$zero(E2) \to DA = -\pi/6$$

Cela signifie que si on se rapproche trop de la courbe inférieure, il est nécessaire de braquer vers la gauche, la seconde règle traduisant la situation symétrique.
Le contrôleur donne en sortie la variation d'angle *DA* à donner à la direction *A* du robot, d'où le « programme principal » dont la première clause indique l'échec si le robot n'est plus sur la route entre les deux courbes, la seconde le succès s'il est arrivé et la troisième le pas à faire. On considère la fin d'un trajet avec succès si l'abscisse dépasse 500.

```
trajet(X, Y, _, _, _, _, _, _, _)
        :- bas(X, F1), haut(X, F2), (Y < F1 ; F2 < Y), write(' echec '), !.
trajet(X,_, _, _, _, _, _, _, _) :- X > 500, write(' succes '), !.
trajet(X0, Y0, A, P, H, M, K1, K2, DM)
        :- X1 is X0 + DM*cos(A – 1.57), intersec(X1, X0, Y0, A, H, M, K2, R1),
        X2 is X0 + DM*cos(A + 1.57), intersec(X2, X0, Y0, A, H, M, K2, R2),
        E1 is abs((R1 – X0) / (DM*sin(A))), E2 is abs((X0 – R2) / (DM*sin(A))),
        commande(E1, E2, DA), NA is A + DA,
        NX is X0 + P*cos(NA), NY is Y0 + P*sin(NA),
        line(X0, Y0, NX, NY), trajet(NX, NY, NA, P, H, M, K1, K2).
```

```
commande(E1, E2, U) :- zero(E1, D1), zero(E2, D2), moyenne(D1, D2, U).

moyenne(D1, D2, 0) :- 0 is D1 + D2, !.
        % cas où les parois sont hors de portée des capteurs
moyenne(D1, D2, U) :- U is (D1 – D2) / (0.5236*(D1 + D2)).
        % les conclusions sont Dπ/6
```

Dans cette programmation descendante, l'essentiel, c'est-à-dire le prédicat trajet, fait appel au prédicat commande qui, ici, peut être considérablement simplifié, et au problème du calcul de l'intersection d'une droite et une courbe qui nécessite un plus long développement :

Au point courant, la normale à la trajectoire du robot a pour équation :

$$(X - X0)\cos(A) + (Y - Y0)\sin(A) = 0$$

Trouver une intersection entre cette droite et la courbe d'équation $Y = h\sin(mX) + k$ va se faire par dichotomie sur un intervalle $[X0, X1]$ pour celle du bas et $[X0, X2]$ pour celle du haut, avec naturellement des orientations variables et le cas particulier où $A = 0$.
Simulant un capteur de distance d'une reconnaissance maximale DM, on prendra par exemple ici $DM = 30$, mesuré en pixels, on part avec :

$$X1 = X0 + DM\cos(A - \pi/2) \text{ et } X2 = X0 + DM\cos(A + \pi/2)$$

Le problème de l'intersection de la droite normale avec l'une des courbes revient à trouver dans un intervalle donné $[X0, Xi]$ la racine R, si elle existe, de l'équation

$$F(R) = (R - X0)\cos(A) + (h\sin(mR) + k - Y0)\sin(A) = 0$$

Pour connaître ce R qui annule, disons à 0.001 près cette équation, nous utilisons la méthode de dichotomie, il faut réitérer une segmentation en deux de l'intervalle noté $[Xi, Xj]$ en tenant compte du signe des valeurs de F à ses extrémités que l'on appellera Fi, Fj, d'où le prédicat *intersec* qui appelle *dicho* avec toutes les valeurs destinées à calculer les valeurs de F aux extrémités de l'intervalle, lequel *dicho* appellera *dichobis* avec ces mêmes valeurs ainsi que celles C, Fc du centre de l'intervalle.
La dichotomie se fondant alors sur un raisonnement de signe pour aller chercher la racine R soit à gauche de C, soit à droite :

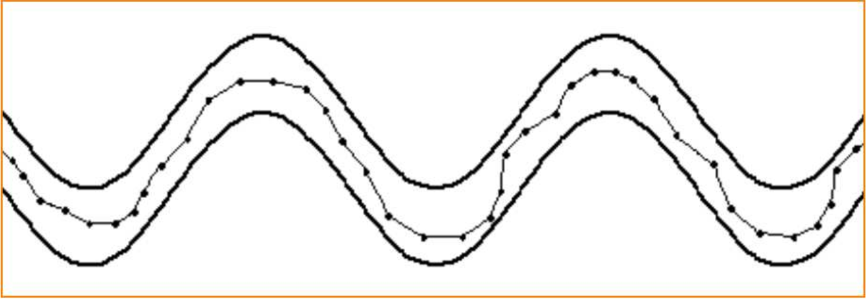
```
intersec(Xi, X0, Y0, A, H, M, K, R)
        :- F is (Xi – X0)*cos(A) + (H*sin(M*Xi) + K – Y0)*sin(A),
        dicho(Xi, F, X0, Y0, X0, Y0, A, H, M, K, R).

dicho(R, F, _, _, X0, Y0, A, H, M, K, R) :- abs(F) < 0.001, !.
dicho(Xi, Fi, Xj, Fj, X0, Y0, A, H, M, K, R) :- C is (Xi + Xj)/2,
        Fc is (C – X0)*cos(A) + (H*sin(M*C) + K – Y0)*sin(A),
        dichobis(Xi, Fi, C, Fc, Xj, Fj, X0, Y0, A, H, M, K, R).

dichobis(Xi, Fi, C, Fc, Xj, Fj, X0, Y0, A, H, M, K, R) :- Fc*Fj > 0, !,
        dicho(Xi, Fi, C, Fc, X0, Y0, A, H, M, K, R). % cas de la première moitié

dichobis(Xi, Fi, C, Fc, Xj, Fj, X0, Y0, A, H, M, K, R) :-
        dicho(C, Fc, Xj, Fj, X0, Y0, A, H, M, K, R). % cas de la seconde moitié
```

A présent, pour faire fonctionner le robot, il faut, comme au chapitre 6 ouvrir une fenêtre graphique (les différentes versions de Prolog possèdent différentes méthodes)

```
ouvrir :- draw(1, 'dessin'(50, 10, 200, 500), _).
fermer :- draw(2,_,_).     % fermeture de la fenêtre graphique

line(X1, Y1, X2, Y2) :- draw(3, line(X1, Y1, X2, Y2),_).

haut(X, Y) :- Y is 40*sin(0.04*X) + 50.
bas(X, Y) :- Y is 40*sin(0.04*X) + 70.
```

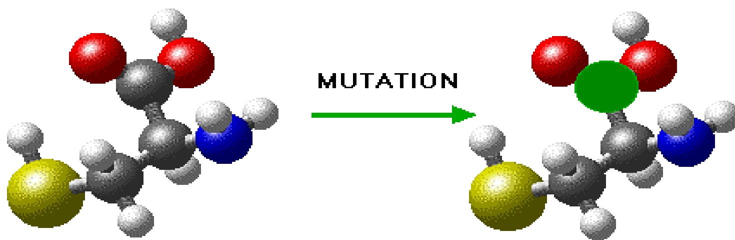
Pour le départ, il faut lancer le trajet avec des coordonnées de départ $X0 = 10$, $Y0 = 100$, une orientation $A = -\pi/4$, un pas, disons de $P = 10$ pixels, et les paramètres des deux courbes constituant les parois du couloir $H = 40$, $M = 0.04$, $K1 = 50$, $K2 = 70$.
Il faut bien entendu tracer les deux courbes *haut* et *bas* avec des paramètres indicatifs, de plus le réglage du paramètre $DM = 30$ n'est pas facile, mais la sensibilité du contrôleur flou n'a heureusement pratiquement que ce paramètre à régler.

```
| trajet(10, 100, –0.78, 10, 40, 0.04, 50, 70, 30).
```

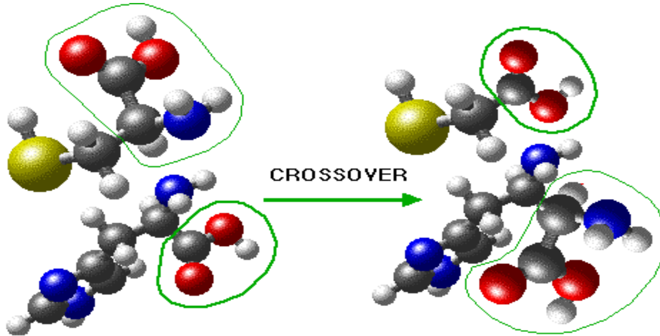


UN ALGORITHME D'EVOLUTION

L'évolution artificielle est un domaine de l'intelligence artificielle où, en s'inspirant de l'évolution naturelle des espèces vivantes, on en déduit des heuristiques assez efficaces pour optimiser une fonction *f* n'ayant pas de « bonnes propriétés mathématiques ». Il s'agit, dans tout ce qui suit, de trouver le minimum global d'une fonction positive *f* définie sur un domaine tel que l'intervalle $[a, b]^{dim}$. Le principe est de partir d'une population aléatoire d'un certain nombre μ de points (appelés chromosomes ou individus) dans ce domaine.
De générations en générations, des individus de cette population sont sélectionnés, croisés par des « crossover », et produisent donc des « enfants » avec des « mutations », lesquels sont évalués suivant leur valeur relativement à *f* et triés.



Ces algorithmes s'écartent plus ou moins de l'évolution naturelle, depuis « l'algorithme génétique standard », jusqu'aux « stratégies d'évolution » de part les différents opérateurs génétiques utilisés et les différentes options de mise à jour.
Celui que nous développons ci-dessous est dit « steady-state genetic algorithm » (SSGA), il est simple et efficace.



L'algorithme SSGA(μ, τ, ξ)

Dans le but de trouver le minimum de *f* sur un domaine, une population initiale de μ points pris au hasard est formée. Chacun d'entre eux est évalué suivant sa valeur pour *f*, ils sont ensuite triés du meilleur au pire, donc dans l'ordre croissant vis-à-vis de *f*.
D'une génération à l'autre, chaque individu produit un enfant qui sera évalué pour *f*. Cet enfant est produit par un des opérateurs génétiques tiré au hasard dans la liste énoncée plus bas.
Les τ meilleurs enfants et ξ individus tirés au hasard (grâce au prédicat *mig*) remplace les μ–τ–ξ pires parents. Après cette mise à jour, la nouvelle population contient de nouveau μ individus qui sont triés.
L'arrêt peut se faire si le meilleur individu *x* possède une valeur $f(x) < \varepsilon$.

L'idée initiale des algorithmes génétiques était d'opérer avec un codage binaire des individus. Dans cet esprit, une mutation, qui est une modification d'un seul chiffre binaire, peut être interprétée aussi bien comme une exploration du voisinage de l'individu que comme une exploration du domaine entier en brouillant tout grâce à ce codage.
Pour notre part, nous allons représenter une solution à un problème quelconque par une liste de *M* chiffres et dans tout ce qui suit, une population sera une liste de μ listes, chacune de la forme $(f(x), x)$ où *x* est un individu codé par une liste et sera précédée de sa valeur suivant *f*. Ainsi, si lors de la recherche du minimum de *f* sur [0, 1], [2, 8, 6, 4, 0, 7, 1, 1, 5] est le développement décimal du réel $x = 0.286407115$, la liste [$f(0.286407115)$, 2, 8, 6, 4, 0, 7, 1, 1, 5] sera un élément de la population parmi d'autres.
Les opérateurs génétiques utilisés ici sont la migration fournissant un individu au hasard, la mutation qui consiste à modifier l'un des chiffres du codage, la

transposition consistant à inverser deux parties préfixe et suffixe d'un individu, le *crossover0* qui prend les chiffres au hasard tantôt du père, tantôt de la mère et les croisements (*crossover1, crossover2*) à un ou deux sites qui consistent à produire un enfant par des segments de codage du père et de la mère.

On utilise le prédicat *random(A, B, X)* produisant un entier *X* entre *A* et *B* non compris.

Un « individu » ou « chromosome » est une liste de chiffres, ce peut naturellement peut être tout autre chose, mais nous nous en tiendrons à cette représentation et dans toute la suite ce sera une liste de *M* chiffres et une population sera une liste de μ individus.
Afin de manipuler les individus, il est nécessaire de reprendre les prédicats *tete, queue, rang*, ainsi que *hasard* qui permet de tirer un élément *X* dans une liste *L.*

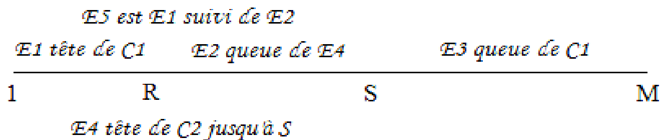
```
app(X, [X / _]).
app(X, [_ / L]) :- app(X, L).

len([], 0).
len([_ / L], N) :- len(L, M), N is M + 1.

tete(_, 0, []) :- !.
tete([], _, []) :- !.
tete([X / L], N, [X / T]) :- M is N – 1, tete(L, M, T).

queue(L, 0, L) :- !.
queue([], _, []) :- !.
queue([_ / L], N, Q) :- M is N – 1, queue(L, M, Q).

rg(X, 0, [X / _]).
rg(X, M, [_ / L]) :- rg(X, K, L), M is K + 1.

conc([], L, L).
conc([X / L], M, [X / N]) :- conc(L, M, N).

hasard(L, X) :- len(L, N), random(0, N, R), rg(X, R, L).
```

Les relations *mig, mut, transpo, cross0, cross1, cross2* permettent respectivement de construire un individu complètement nouveau, de le muter (c'est-à-dire modifier l'un de ses chiffres), de transposer une tête et la queue correspondante du codage, enfin de produire un enfant par croisement avec un autre individu.

Dans tous ces prédicats, le paramètre *M* désigne le nombre de chiffres composant un individu, et *C* (chromosome) un individu.
Le prédicat *construc* permet de construire une population aléatoire de *MU* individus composés chacun de *M* chiffres.

```
mig(0, []) :- !.
mig(M, [G | C]) :- hasard([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], G), N is M – 1, mig(N, C).

construc(_, 0, []) :- !.
construc(M, MU, [C | P]) :- mig(M, C), N is MU – 1, construc(M, N, P).

mut(M, C, CM) :- random(1, M, R), tete(C, R, T), queue(C, R, [_ | Q]),
                 hasard([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], G), conc(T, [G | Q], CM).

transpo(M, C, CT) :- random(2, M, R), tete(C, R, T),
                 queue(C, R, Q), conc(Q, T, CT).
transpo(M, C, CT) :- random(2, M, R), tete(C, R, T),
                 queue(C, R, Q), conc(Q, T, CT).

cross1(M, C1, C2, E) :- random(2, M, R), tete(C1, R, E1),
                 queue(C2, R, E2), conc(E1, E2, E).

cross0([], [], []). % croisement uniforme en tirant à pile ou face chaque gène
cross0([G1 | C1], [_ | C2], [G1 | E])
                 :- random(0, 2, X), 0 < X, !, cross0(C1, C2, E).
cross0([_ | C1], [G2 | C2], [G2 | E]) :- cross0(C1, C2, E).

cross2(M, C1, C2, E) :- random(2, M, R), random(R, M, S),
                 tete(C1, R, E1), queue(C1, S, E3), tete(C2, S, E4),
                 queue(E4, R, E2), conc(E1, E2, E5), conc(E5, E3, E).
```

On peut voir que le crossover à un site est analogue à une transposition et que pour le « crossover0 », on tire à pile ou face pour choisir le gène tantôt du père, tantôt de la mère.
Le crossover à deux sites consiste à tirer au hasard deux indices *R* et *S*, prendre le début et la fin du premier parent *C*1 et le segment délimité par ces deux indices, dans l'autre parent *C*2. Ce découpage et les paramètres utilisés sont illustrés par la figure ci-dessous.



mig(5, *C*). → *C* = [0, 5, 1, 8, 5] % *C* est un chromosome tout neuf
mig(5, *C*). → *C* = [4, 3, 8, 8, 7] % en est un autre
construc(3, 5, *P*). → *P* = [[0, 5, 1], [8, 5, 4], [3, 8, 8], [7, 1, 8], [7, 5, 3]]
construc(4, 7, *P*). % construit 7 individus de longueur 4
→ *P* = [[0, 5, 1, 8], [5, 4, 3, 8], [8, 7, 1, 8], [7, 5, 3, 0], [0, 3, 1, 1],
[9, 4, 1, 0], [0, 3, 5, 5]]

Maintenant, les opérateurs génétiques donnent par exemple :

mut(5, [0, 1, 2, 3, 4], *X*). → *X* = [0, 1, 2, 6, 4]
transpo(5, [0, 1, 2, 3, 4], *X*). → *X* = [3, 4, 0, 1, 2]
cross0([a, b, c, d, e, f, g], [0, 1, 2, 3, 4, 5, 6], *X*).
→ *X* = [0, 1, c, 3, 4, f, 6]
cross1(7, [a, b, c, d, e, f, g], [0, 1, 2, 3, 4, 5, 6], *X*).
→ *X* = [a, b, c, d, 4, 5, 6]
cross2(7, [a, b, c, d, e, f, g], [0, 1, 2, 3, 4, 5, 6], *X*).
→ *X* = [a, b, c, 3, 4, f, g]
cross2(9, [a, b, c, d, e, f, g, h, i], [0, 1, 2, 3, 4, 5, 6, 7, 8], *X*).
→ *X* = [a, b, 2, 3, 4, f, g, h, i]

L'évaluation pour une fonction *f* ne peut se faire en passant *f* en paramètre, ce qui est possible dans un langage fonctionnel comme Lisp ou Caml. La fonction à optimiser *f* sera redéfinie dans les trois exemples qui suivent.

Pour une population *P* d'individus, *PE* est la population des individus évalués, nous convenons de mettre la valeur *f(i)* d'un individu *i* en première position de la liste ce qui simplifie les écritures.

Un premier exemple avec la somme des chiffres

Ainsi, par exemple, si *f* est la simple somme (*som)* des chiffres composant un individu *i*, on a un optimum de 0 à atteindre, et par exemple

som([2, 2, 8, 5, 3, 0, 1], V). → *V = 2*

Le prédicat *evaluer* doit appliquer *f* aux différents individus d'une population et placer cette valeur au début de chacun d'entre eux.
Le tri par insertion est repris pour trier la population en considérant que ce sont les valeurs mises en tête des individus qui doivent être prises en compte pour un tri en ordre croissant.

```
som([], 0) :- !.
som([X | I], V) :- som(I, S), V is S + X.

evaluer([], []).
evaluer([I | P], [[V | I] | PE]) :- som(I, V), evaluer(P, PE).

insert(I, [], [I]).
insert([X | I], [[Y | J] | L], [[X | I], [Y | J] | L]) :- X < Y, !.

tri([], []).
tri([I | P], R) :- tri(P, PT), insert(I, PT, R).
```

```
som([3, 3, 0, 1], V). → V = 7
som([2, 2, 8, 5, 3, 0, 1], V) . → V = 21
evaluer([[2, 3, 4], [5, 0, 1], [3, 2, 1]], PE).
→ PE = [[9, 2, 3, 4], [6, 5, 0, 1], [6, 3, 2, 1]]

tri([[4, a, s, d], [6, j, k, l], [0, a, z, e, r, t, y], [2, q, s, d, f]], PT).
→  PT = [[0, a, z, e, r, t, y], [2, q, s, d, f], [4, a, s, d], [6, j, k, l]]
```

A présent, on est en mesure de construire la population fille à partir de la population parentale. Pour ce faire, chaque parent doit produire un enfant en tirant au sort un opérateur génétique, ce qui permet par la suite d'étendre le programme à d'autres opérateurs. Nous faisons ici le choix de faire une mutation une fois sur deux en tirant à pile ou face, sinon un des trois cross-overs, en choisissant au hasard un autre parent dans la population restante. Si cette dernière est vide, alors nous choisissons une transposition. Mais ces choix peuvent être complètement remis en cause au vu des clauses *choix*. On verra quelques exemples pour $M = 10$, il faut bien noter que la population père est déjà évaluée, celle des enfants ne l'étant que par la suite.
Le prédicat *enfants* relie un entier M (longueur des gènes), une population père et une population des enfants.

```
enfants(_, [], []).
enfants(M, [[_ | I] | P], [E | PE]) :- choix(M, I, P, E), enfants(M, P, PE).

choix(M, I, _, E) :- random(0, 2, X), X < 1, mut(M, I, E).
choix(M, I, [], E) :- transpo(M, I, E), !.
choix(_, I, P, E) :- random(0, 3, X), X < 1, hasard(P, [_ | J]), cross0(I, J, E).
choix(M, I, P, E) :- random(0, 2, X), X < 1,
        hasard(P, [_ | J]), cross1(M, I, J, E), !.
choix(M, I, P, E) :- hasard(P, [_ | J]), cross2(M, I, J, E).
```

enfants(10, [[31, 0, 7, 8, 4, 1, 2, 2, 0, 3, 4],
[25, 4, 7, 3, 1, 2, 2, 0, 5, 0, 1], [17, 2, 1, 0, 6, 3, 0, 1, 1, 2, 1]], *E*)
→ *E* = [[0, 0, 8, 4, 1, 2, 2, 0, 3, 4], [2, 1, 0, 1, 2, 2, 0, 1, 0, 1],
[1, 2, 1, 0, 6, 3, 0, 1, 1, 2]]

enfants(10, [[31, 0, 7, 8, 4, 1, 2, 2, 0, 3, 4],
[25, 4, 7, 3, 1, 2, 2, 0, 5, 0, 1], [17, 2, 1, 0, 6, 3, 0, 1, 1, 2, 1]], *E*)
→ *E* = [[4, 7, 3, 4, 1, 2, 2, 0, 0, 1], [4, 7, 3, 1, 2, 2, 0, 1, 0, 1],
[3, 0, 1, 1, 2, 1, 2, 1, 0, 6]]

Nous arrivons maintenant au cœur de l'algorithme, à savoir la mise à jour de la population à chaque génération. Pour cela, la population précédente déjà triée des pères va produire le même nombre d'enfants, ceux-ci devront être évalués, puis triés, la génération d'après étant formée par les τ meilleurs d'entre eux et ξ nouveaux individus aléatoires devant être évalués. Ces nouveaux individus remplacent les $\tau + \xi$ pires parents avec la condition $0 < \tau$ et $\tau + \xi < \mu$, de telle sorte que la nouvelle population ait le même effectif.

```
generation(M, MU, TAU, KSI, P1, P2)
    :- enfants(M, P1, PE), evaluer(PE, PEE),
    tri(PEE, PET), tete(PET, TAU, NE),
    construc(M, KSI, NI), evaluer(NI, NIE),
    NBP is MU – TAU – KSI, tete(P1, NBP, NP),
    conc(NE, NIE, P3), conc(P3, NP, P4), tri(P4, P2).
```

generation(10, 3, 1, 1, [[17, 2, 1, 0, 6, 3, 0, 1, 1, 2, 1],
[25, 4, 7, 3, 1, 2, 2, 0, 5, 0, 1], [31, 0, 7, 8, 4, 1, 2, 2, 0, 3, 4]], *PE*).
→ *PE* = [[16, 2, 0, 0, 6, 3, 0, 1, 1, 2, 1], [17, 2, 1, 0, 6, 3, 0, 1, 1, 2, 1],
[44, 5, 6, 3, 6, 8, 5, 5, 0, 4, 2]]

Revoyons maintenant toutes les étapes pour un codage de longueur $M = 3$, une taille $\mu = 7$, un taux de renouvellement $\tau = 2$ et un taux de migration $\xi = 1$. D'abord la construction d'une population initiale, puis son évaluation, le tri et une première génération où les améliorations sont déjà constatées.

construc(3, 7, *P*).
→ *P* = [[8, 6, 6], [5, 7, 7], [9, 4, 3], [1, 9, 8], [4, 1, 8], [7, 0, 4], [9, 0, 1]]

evaluer([[8, 6, 6], [5, 7, 7], [9, 4, 3], [1, 9, 8], [4, 1, 8], [7, 0, 4],
[9, 0, 1]], *PE*).
→ *PE* = [[20, 8, 6, 6], [19, 5, 7, 7], [16, 9, 4, 3], [18, 1, 9, 8],
[13, 4, 1, 8], [11, 7, 0, 4], [10, 9, 0, 1]]

```
tri([[20, 8, 6, 6], [19, 5, 7, 7], [16, 9, 4, 3], [18, 1, 9, 8], [13, 4, 1, 8],
        [11, 7, 0, 4], [10, 9, 0, 1]], PT).
→ PT = [[10, 9, 0, 1], [11, 7, 0, 4], [13, 4, 1, 8], [16, 9, 4, 3],
        [18, 1, 9, 8], [19, 5, 7, 7], [20, 8, 6, 6]]

generation(3, 7, 2, 1, [[10, 9, 0, 1], [11, 7, 0, 4], [13, 4, 1, 8],
        [16, 9, 4, 3], [18, 1, 9, 8], [19, 5, 7, 7], [20, 8, 6, 6]], NP).
→ NP = [[7, 3, 3, 1], [10, 9, 0, 1], [11, 7, 0, 4], [12, 7, 0, 5],
        [12, 1, 3, 8], [13, 4, 1, 8], [16, 9, 4, 3]]
```

Il reste à enchaîner les générations grâce à un prédicat *evol* qui sera vrai dès qu'une population aura son meilleur élément i obtenu avec une valeur $f(i) < \varepsilon$. Cependant, comme ces algorithmes sont stochastiques, il est prudent de se fixer une borne maximale de générations ou mieux, une borne maximale *MAX* en nombre d'évaluations de la fonction à optimiser. C'est ce dernier critère qui permet de comparer ces algorithmes et notamment de fixer empiriquement leurs paramètres.

Dans le prédicat *generation,* une population *pop*, pour la fonction f à minimiser, un taux *tau* de renouvellement et un taux *ksi* de nouveaux individus, on produit la génération suivante évaluée et triée. Cette nouvelle population est constituée par une option très simple pour poursuivre l'exploration tout en approfondissant l'exploitation de l'optimum provisoire en décidant d'un taux ξ d'exploration.

Ainsi, à chaque génération, dans la population de μ individus, les τ meilleurs enfants et un nombre ξ de nouveaux individus aléatoires, remplacent les $\tau + \xi$ pires parents avec la condition $0 < \tau$ et $\tau + \xi < \mu$.

Le prédicat principal *evol* devra renvoyer le meilleur individu suivant f au bout de *NV* évaluations avec une population initiale.

Ainsi par exemple on peut avoir :

```
evol(0, 0.0001, 500, 3, 7, 3, 1, P)
→ 87 avec la population [[0, 0, 0, 0], [1, 1, 0, 0], [1, 1, 0, 0], [1, 0, 0, 1],
        [4, 4, 0, 0], [4, 1, 3, 0], [11, 8, 3, 0]]
```

```
evol(0, EPS, MAX, M, MU, TAU, KSI, _)
        :- construc(M, MU, PI), evaluer(PI, PE), tri(PE, P),
        evol(MU, EPS, MAX, M, MU, TAU, KSI, P).
        % départ de l'évolution avec NV = 0
```

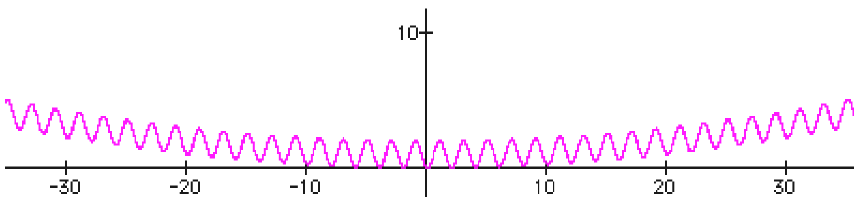
```
evol(NV, EPS, MAX, _, _, _, _, [[V | I] | _])
	:- (V < EPS ; MAX < NV), !, write('valeur '),
	write(V), write(' obtenue pour '), write(I), write(' avec '), write(NV),
	write(' évaluations '). % fin de l'évolution

evol(NV1, EPS, MAX, M, MU, TAU, KSI, P) :-
	generation(M, MU, TAU, KSI, P, PE), NV2 is NV1 + MU + KSI,
	evol(NV2, EPS, MAX, M, MU, TAU, KSI, PE). % cas général
```

> Exemple pour la fonction *som* qui calcule simplement la somme des entiers d'un individu (attention, la dispersion est très grande) :
>
> evol(0, 1, 500, 3, 7, 2, 1, []).
> → valeur 0 obtenue pour [0, 0, 0] avec 167 évaluations

D'autres essais donnent une très grande dispersion des résultats, c'est pourquoi on pourra compléter en construisant un prédicat *moyenne* qui lancera 100 processus d'évolution en renvoyant la moyenne du nombre d'évaluations de f pour atteindre ε.
Les expériences à faire seraient plutôt pour $\mu = 3$ à 12, pour $\tau = 1$ à $\mu - 2$, et pour $\xi = 1$ à $\mu - \tau - 1$, fournissant des résultats optimaux en général pour $\xi = 1$ et τ entre $\mu/4$ et $\mu/2$.

Exemple d'application à la recherche du minimum global de la fonction de Rastrigin $F_R(x) = 0.01[x^2 + 2 - 2\cos(2\pi x)]$ en dimension 1

Cette fonction difficile à minimiser à cause de ses multiples minimums locaux est donnée ici en dimension 1. On voit qu'il s'agit en gros d'une sinusoïde inscrite sur une parabole (les coefficients peuvent être différents) et qu'elle possède un minimum local pour chaque entier, le véritable minimum étant en 0. Les plus proches valant 0.01, on peut prendre cette valeur pour ε.



Pour une fonction de [*a, b*] dans **R**, on peut par exemple coder par la liste des décimales, puis par dilatation obtenir un réel dans [a, b]. Ainsi, par exemple, la liste [7, 5] représente 0,75 qui dans l'intervalle [1, 5] représente la valeur 4, c'est-à-dire la valeur située aux trois quarts de l'intervalle.
Ici, par exemple, si x est dans [−10, 10], la liste [3, 3, 3, 3, 3, 3, 3, 3, 3] représente 1/3 dans l'intervalle [0, 1] et donc un prédicat *decode* se chargera de passer de la liste des décimales [3, 4, 5, 6, 7] au réel 0.34567.
On lui donne le départ avec $K = 0.1$ et $R = 0$, le résultat est dans [0, 1].

On a également besoin en général d'un prédicat réalisant la dilatation de [0, 1] dans [a, b], ici il est dans la formule 20*R – 10 si on se limite à [−10, 10].
Le prédicat *ras* réalise la fonction de Rastrigin, mais entre un individu codé comme liste de chiffres et sa valeur réelle.

```
decode([], _, R, R).
decode([X | L], K, S, R) :- KS is 0.1*K, SS is S + K*X, decode(L, KS, SS, R).

ras(I, V) :- decode(I, 0.1, 0, R), X is 20*R – 10,
        V is (X*X + 2 – 2*cos(6.28*X)) / 100.
```

```
decode([3, 4, 5, 6, 7], 0.1, 0, R). → R = 0.34567
decode([1, 2, 3, 4, 5, 6, 7, 8, 9], 0.1, 0, R). → R = 0.123456789
ras([9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9], V).
→ V = 1.00001 car la liste correspond à 1 dilaté en 10
ras([0], V). → V = 1.00001
ras([5], V). → V = 0.0 car la liste correspond au réel 0
ras([7, 5], V). → V = 0.250002 car la liste correspond au réel 5
```

On s'aperçoit de petites erreurs, ce qui est normal. Maintenant, pour chercher à optimiser cette fonction dont on connaît d'avance le minimum 0, il suffit de changer le prédicat *evaluer* :

```
evaluer([], []).
evaluer([I | P], [[V | I] | PE]) :- ras(I, V), evaluer(P, PE).
```

Variante à l'algorithme avec un taux d'élimination variable SSGA(μ, τ, π)

Le grand défaut des algorithmes évolutionnaires est le risque de concentrer la population à proximité d'un minimum local, c'est pourquoi une population trop homogène doit être évitée. Afin de favoriser une population hétérogène, on peut la parcourir de deux en deux, en éliminant, dans le cas d'une similarité à définir, le plus mauvais des deux individus au profit d'un nouveau arbitrairement créé. Cette similitude peut être définie en pourcentage de gènes communs en section commençante. Par exemple, pour une population triée *pop* d'individus, tous de longueur *M*, le prédicat *elim* renvoie la même population où les individus proches au-delà du seuil π sont remplacés par des migrations. Le prédicat *evol* est alors redéfini en éliminant dans la génération suivante triée, et en re-triant après coup.

L'expérience a montré qu'en termes de nombre d'appels moyen de *f* pour obtenir son optimum à ε près, ce second algorithme est meilleur que le premier.

La relation de similitude entre deux listes de chiffres de longueur *M* est ici tout simplement, exprimé en pourcentage, la longueur du plus grand préfixe commun rapporté à la longueur *M*. Pour plus de simplicité, le paramètre *M* est passé directement.
On l'exprime en comptant le nombre *K* d'éléments *X* communs consécutifs dans les deux chaînes *I* et *J*.

```
prox(I, J, M, P) :- proxbis(I, J, M, 0, P).

proxbis([X | I], [X | J], M, K1, R) :- K2 is K1 + 1, !, proxbis(I, J, M, K2, R).
proxbis(_, _, M, K, R) :- R is 100*K / M.
```

> Exemples
> prox([a, b, c, d, e, f, g, h, i, j, m], [a, b, c, d, h, g, k, i, j, m], 10, *P*).
> → *P* = 40.0
> prox([a, b, c, d, e, f, g, h, i, j, m], [a, b, c, e, h, g, k, i, j, m], 10, *P*).
> → *P* = 30.0

Nous prenons le parti de ne pas comparer tous les individus deux à deux, mais de les prendre dans l'ordre décroissant du pire au meilleur. Ainsi, le prédicat *elim* va compter le nombre *NV* de nouvelle évaluations de *f*, donc le départ du balayage de la population se fera avec *NV* = 0.

Le total des nouvelles évaluations au cours de l'élimination est récupéré par le paramètre *TV*. Les autres paramètres sont l'ancienne population *AP* en cours de balayage, la nouvelle *NP* en cours de construction, le nombre de gènes *M* et le seuil *PI*. Le résultat est *R*.
La première clause de *elim* signifie que lorsqu'il n'y a plus qu'un élément à regarder, on le conserve ainsi que tous les individus conservés ou créés, et le résultat *R* est délivré.
La seconde clause de ce même prédicat indique que si deux individus *I* et *J* sont proches (*J* étant le meilleur), alors *J* est conservé et *I* remplacé par un migrant aléatoire *NI* dont la valeur est calculée.
La troisième de *elim* clause indique, si les individus *I* et *J* ne sont pas semblables au seuil *PI* près, d'avancer d'un cran.

```
inv(L, LR) :- invbis(L, [], LR).        % Inverse d'une liste
invbis([], R, R).
invbis([X | L], T, R) :- invbis(L, [X | T], R).

elim(TV, TV, [I], NP, _, _, R) :- tri([I | NP], R).
        % cas du dernier individu I (le meilleur)
elim(NV, TV, [[_ | I], [VJ | J] | AP], NP, M, PI, R)
        :- prox(I, J, M, P), P > PI, !, V is NV + 1, mig(M, NI),
        evaluer([NI], [NVI]),
        elim(V, TV, [[VJ | J] | AP], [NVI | NP], M, PI, R).
elim(NV, TV, [I, J | AP], NP, M, PI, R)
        :- elim(NV, TV, [J | AP], [I | NP], M, PI, R).
```

Exemples avec la première fonction *som* pour une liste ordonnée du pire au meilleur avec des taux d'élimination de 2/3 puis 1/3 :
elim(0, *V*, [[14, 8, 3, 3], [13, 8, 3, 2], [12, 8, 3, 1], [6, 2, 2, 2],
[5, 2, 2, 1], [4, 2, 1, 1]], [], 3, 60, *R*).
→ *R* = [[4, 2, 1, 1], [5, 2, 2, 1], [11, 6, 3, 2], [12, 8, 3, 1],
[17, 8, 2, 7], [26, 8, 9, 9]] *V* = 3
elim(0, *V*, [[14, 8, 3, 3], [13, 8, 3, 2], [12, 8, 3, 1], [6, 2, 2, 2],
[5, 2, 2, 1], [4, 2, 1, 1]], [], 3, 30, *R*).
→ *R* = [[4, 2, 1, 1], [12, 8, 3, 1], [13, 4, 2, 7], [13, 7, 1, 5],
[17, 4, 4, 9], [21, 6, 8, 7]] *V* = 4

Il faut maintenant redéfinir *generation* et *evol*. Dans la première clause, la population père *P*1 produit les enfants *PE* qui sont évalués ; en en prenant les *TAU* premiers, on a les nouveaux enfants *NE*, les nouveaux pères étant appelés *NP*. Ensemble, ils forment une nouvelle génération *P*3, laquelle est soumise à un tri *P*4, inversée *P*5 et une élimination.

```
generation(NV, M, MU, TAU, PI, P1, P2)
    :- enfants(M, P1, PE), evaluer(PE, PEE),
    tri(PEE, PET), tete(PET, TAU, NE), P is MU − TAU, tete(P1, P, NP),
    conc(NE, NP, P3), tri(P3, P4), inv(P4, P5),
    elim(0, K, P5, [ ], M, PI, P2), NV is MU + K.
```

L'exemple suivant pour une population de 6 individus montre que les deux meilleurs pères ont été conservés, qu'il y a eu 6 fils produits puis deux nouveaux individus créés par élimination des fils trop ressemblants, avec un seuil de 30%, d'où *NV* = 8. Naturellement, les résultats sont différents à chaque tirage, ainsi le second exemple n'a rien éliminé mais trouvé un fils différent des meilleurs pères quoique de performance aussi bonne.

> generation(*NV*, 3, 6, 2, 30, [[3, 0, 0, 3], [4, 2, 1, 1], [11, 1, 1, 9],
> [12, 8, 3, 1], [15, 7, 5, 3], [16, 7, 1, 8]], NP).
> → *NP* = [[3, 0, 0, 3], [4, 2, 1, 1], [7, 0, 6, 1], [9, 6, 0, 3],
> [11, 1, 1, 9], [12, 8, 3, 1]]   *NV* = 8
>
> generation(*NV*, 3, 6, 2, 30, [[3, 0, 0, 3], [4, 2, 1, 1], [11, 1, 1, 9],
> [12, 8, 3, 1], [15, 7, 5, 3], [16, 7, 1, 8]], *NP*).
> → *NP* = [[3, 1, 1, 1], [3, 0, 0, 3], [4, 2, 1, 1], [9, 0, 0, 9],
> [11, 1, 1, 9], [12, 8, 3, 1]]   *NV* = 6

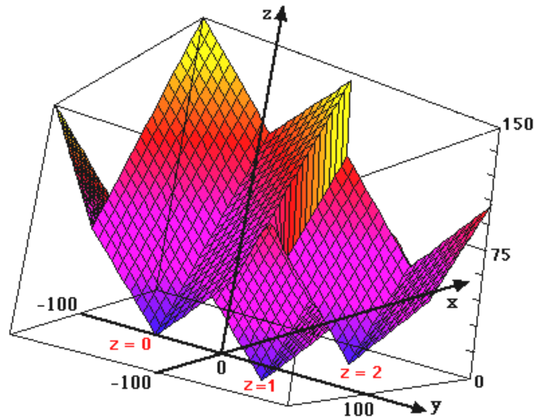
Concernant l'évolution, nous partons comme précédemment d'une population initiale aléatoire, évaluée, puis triée *P*. Il y a donc très peu de changement dans l'écriture des clauses.

```
evol(0, EPS, MAX, M, MU, TAU, PI, _)
    :- construc(M, MU, P0), evaluer(P0, PE), tri(PE, P),
    evol(MU, EPS, MAX, M, MU, TAU, PI, P).
    % départ de l'évolution avec NV = 0
evol(NV, EPS, MAX, _, _, _, _, [[V | I] | _])
    :- (V < EPS ; MAX < NV), !, write('valeur '),
    write(V), write(' obtenue pour '), write(I), write(' avec '), write(NV),
    write(' évaluations '). % fin de l'évolution
evol(NV1, EPS, MAX, M, MU, TAU, PI, P)
    :- generation(K, M, MU, TAU, PI, P, PE), NV2 is NV1 + K,
    evol(NV2, EPS, MAX, M, MU, TAU, PI, PE). % cas général
```

> Exemple avec la fonction *som*
> evol(0, 1, 1000, 3, 7, 3, 30, *P*).
> → valeur 0 obtenue pour [0, 0, 0] avec 127 évaluations

Application à la recherche du minimum global de la fonction « *tripod* » de dimension 2

Cette fonction, définie sur [–100, 100], est très rapide à calculer mais pas facile à optimiser à cause de ses trois bassins d'attraction. Elle constitue donc un bon test pour comparer différents algorithmes évolutionnaires.



tripod(x, y) = si $y < 0$ alors $|x| + |y + 50|$
sinon si $x < 0$ alors $1 + |x + 50| + |y - 50|$
sinon $2 + |x - 50| + |y - 50|$

Nous proposons donc d'utiliser la même représentation des individus en listes de chiffres, car avec cette représentation tout est plus simple. Nous prenons un codage discret des couples de [–100, 100] en choisissant $M = 4$; un individu est un quadruplet (a, b, c, d) représentant $(x, y) = (20a + 2b - 100, 20c + 2d - 100)$ de telle façon que les trois couples suivants aient les valeurs des minima :

$(x, y) = (0, -50)$, soit tripod (5, 0, 2, 5) = 0
$(x, y) = (-50, 50)$, tripod (2, 5, 7, 5) = 1
$(x, y) = (50, 50)$, tripod (7, 5, 7, 5) = 2

```
tripod([A, B, C, D], Z) :- X is 20*A + 2*B – 100,
        Y is 20*C + 2*D – 100, tripodbis(X, Y, Z).
tripodbis(X, Y, Z) :- Y < 0, Z is abs(X) + abs(Y + 50), !.
tripodbis(X, Y, Z) :- X < 0, Z is 1 + abs(X + 50) + abs(Y – 50), !.
tripodbis(X, Y, Z) :- Z is 2 + abs(X – 50) + abs(Y – 50).
```

```
evaluer([], []).
evaluer([I | P], [[V | I] | PE]) :- tripod(I, V), evaluer(P, PE).
```

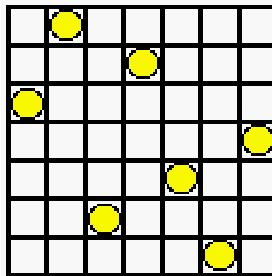
> tripod([5, 0, 2, 5], V). → V = 0
> tripod([2, 5, 7, 5], V). → V = 1
> tripod([7, 5, 7, 5], V). → V = 2
> evol(0, 1, 1000, 4, 7, 3, 30, P).
> → valeur 0 obtenue pour P = [5, 0, 2, 5] avec 201 évaluations

Remarque : en Prolog, si la valeur absolue n'est pas prédéfinie, il est facile de la refaire :

```
abs(X, Y) :- X < 0, Y is –X, !.
abs(X, X).
```

Application au problème des reines de Gauss

Pour ce problème, déjà vu au chapitre 4, un damier $M*M$ où M reines sont placées, chacune sur une colonne, sera représenté par la liste des numéros de lignes occupées par ces reines. En se limitant à $M < 10$ (mais même au-delà) la représentation précédente peut convenir.
Ainsi, par exemple, la liste [2, 0, 5, 1, 4, 6, 3], les lignes étant comptées à partir de 0, représente la solution pour $M = 7$ figurée par le dessin ci-dessous.



La fonction que l'on veut minimiser est le nombre de paires de reines en position de prise, en arrivant à une disposition pour laquelle la fonction *gauss* s'annule, l'évolution de la population s'arrêtera.
La difficulté est précisément d'écrire cette fonction. Pour une liste de numéros de lignes S, le nombre de paires de reines en position de prise sera V, son

décompte commence par 0 et est augmenté d'une unité chaque fois qu'en avançant dans la lecture de la liste, la première reine peut prendre une autre de la liste restante.
Plus précisément, si *C* est le numéro de la reine courante, dans la liste *LR* des reines suivantes, on compte leurs rangs *K* à partir de 1 et dès que la *K*-ième reine suivante possède la valeur *C* ou bien *C* – *K* ou encore *C* + *K* elle peut être prise. Il faut noter que l'utilisation de la relation *is* effectue un calcul et vérifie une égalité, le test utilisant une disjonction (;) de trois propositions.

```
gauss(S, V) :- gaussbis(S, 0, V).
gaussbis([], V, V).
% cas où toute la liste est parcourue, la réponse V est la somme trouvée
gaussbis([C | LR], SP, V) :- prises(C, 1, LR, 0, NP),
        NS is SP + NP, gaussbis(LR, NS, V).

prises(_, _, [], NP, NP). % la liste est épuisée, NP est le nombre de prises
prises(C, K, [D | LC], SP, NP)
        :- (C = D; C is D – K; C is D + K), !,
        NN is SP + 1, NK is K + 1, prises(C, NK, LC, NN, NP).
        % cas d'une prise possible, on continue la lecture du damier
prises(C, K, [D | LC], SP, NP) :- NK is K + 1, prises(C, NK, LC, SP, NP).
```

```
gauss([2, 0, 5, 1, 4, 6, 3], V). → V = 0
gauss([0, 0, 6, 8, 1, 5, 7, 2, 4], V). → V = 2
gauss([3, 0, 6, 8, 1, 5, 7, 2, 4], V). → V = 0
```

Pour utiliser tout ce qui précède, il reste à faire une petite modification sur les productions par migration ou mutation, car pour ce problème particulier, un individu de taille *M* ne doit avoir ses gènes que parmi des nombres entre les valeurs 0 et *M* – 1.

```
mig(0, []) :- !.
mig(M, [G | C]) :- random(0, M, G), N is M – 1, mig(N, C).

mut(M, C, CM) :- random(1, M, R), tete(C, R, T), queue(C, R, [_ | Q]),
        random(0, M, G), conc(T, [G | Q], CM).

evaluer([], []).
evaluer([I | P], [[V | I] | PE]) :- gauss(I, V), evaluer(P, PE).
```

construc(7, 4, *P*).
→ *P* = [[5, 0, 3, 0, 1, 1, 0], [3, 4, 0, 2, 2, 1, 0], [4, 2, 4, 0, 2, 1, 0],
[0, 1, 4, 0, 0, 0, 0]]

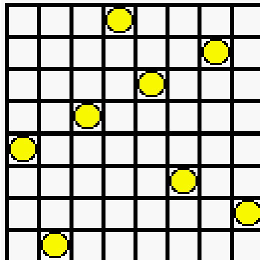
En fixant un maximum d'appels de la fonction gauss à 1000, voici quelques solutions pour *M* de 5 à 8 :

evol(0, 1, 1000, 5, 7, 3, 30, *P*). →
valeur 0 obtenue pour [2, 4, 1, 3, 0] avec 121 évaluations ;
valeur 0 obtenue pour [2, 0, 3, 1, 4] avec 68 évaluations ;
valeur 0 obtenue pour [0, 2, 4, 1, 3] avec 205 évaluations ;
valeur 0 obtenue pour [0, 2, 4, 1, 3] avec 612 évaluations…

evol(0, 1, 5000, 6, 7, 3, 30, *P*). →
valeur 0 obtenue pour [1, 3, 5, 0, 2, 4] avec 711 évaluations ;
valeur 0 obtenue pour [2, 5, 1, 4, 0, 3] avec 1855 évaluations…

evol(0, 1, 5000, 7, 5, 2, 30, *P*). →
valeur 0 obtenue pour [5, 2, 0, 3, 6, 4, 1] avec 165 évaluations ;
valeur 0 obtenue pour [5, 2, 6, 3, 0, 4, 1] avec 226 évaluations ;
valeur 0 obtenue pour [4, 1, 5, 2, 6, 3, 0] avec 793 évaluations…

evol(0, 1, 5000, 8, 5, 2, 30, *P*). →
valeur 0 obtenue pour [4, 7, 3, 0, 2, 5, 1, 6] avec 2504 évaluations…
Cette dernière solution étant :

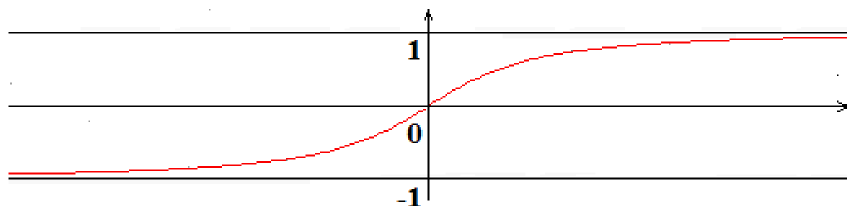


RESEAUX DE NEURONES

Dans ce domaine de l'intelligence artificielle appelé « connexionisme », les réseaux de neurones sont des schématisations de cerveaux simplifiés, dont on postule qu'ils sont constitués d'un très grand nombre d'unités simples en interaction.

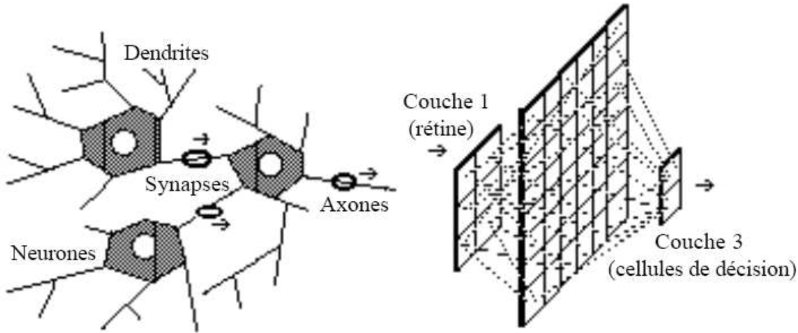
Plusieurs modèles existent utilisés en reconnaissance des formes, mais l'un des modèles les plus séduisants et les plus utilisés est celui du réseau à couches. Dans chacune des couches, chaque neurone est relié à ceux de la couche précédente dont il reçoit les informations et à chaque neurone de la couche suivante à qui il transmet des informations, mais il n'est pas relié aux autres neurones de sa propre couche.

On considère que chaque neurone reçoit par ses « dendrites » une certaine activation électrique dont une somme pondérée constitue l'entrée e (un certain poids $w_{i,j}$ sera affecté à la liaison entre les neurones i et j).
Par suite, le neurone passe à un certain « état » $s = f(e)$ qui sera la mesure de sa sortie portée par son « axone ».
De plus, f peut être modélisée par une simple fonction de seuil ou par une fonction du type *atan* ou *th* de façon à se ramener à l'intervalle [–1, 1].
Rappelons que $th(x) = (\exp(2x) - 1) / (\exp(2x) + 1)$ réalise une contraction bijective de R sur l'intervalle ouvert de –1 à 1 :



L'axone transmet ensuite cette valeur par l'intermédiaire de « synapses » aux dendrites d'autres neurones. (Les poids mesurent en fait l'efficacité des synapses.)

L'apprentissage du réseau consiste en une modification de ces poids au fur et à mesure des expériences, c'est-à-dire de la confrontation entre le vecteur sortant de la dernière couche et celui qui est attendu en fonction d'un vecteur d'entrée.



Algorithme de rétropropagation

Dans un réseau multicouche, chaque neurone n, à l'intérieur, ayant $e = \sum w_{i,n} s_i$ pour entrée, où i parcourt les neurones de la couche précédente, aura une sortie $s = f(e)$.
Lorsque l'on propose le vecteur $X = (x_1, x_2, ..., x_m)$ à la première couche, la dernière restitue le vecteur $S = (s_1,..., s_p)$ alors qu'on attend $Y = (y_1,..., y_p)$ comme réponse.
Le but de cet algorithme est d'exprimer l'erreur quadratique $E = \sum_{1 \leq i \leq p}(y_i - s_i)^2$ et de chercher à la minimiser en modifiant chaque poids w suivant l'influence qu'il a sur E : $w(t + dt) - w(t) = -\mu.\partial E/\partial w$ où µ est un coefficient positif, le « pas » du gradient.

En effet, si cette dérivée est nulle ou faible, cela veut dire que l'erreur E dépend peu de ce poids là w et donc qu'il n'y a pas lieu de le modifier. Si elle est positive et importante, cela signifie au contraire que E croît avec w, donc on diminue w d'autant.
On cherche donc à exprimer ce gradient qui est le vecteur formé par tous les $\partial E/\partial w$.

Plaçons nous entre le neurone i d'une couche et un neurone n fixé, en notant i l'indice parcourant la couche précédente et j celui de la couche suivante. Si d_n est la dérivée partielle de E par rapport à e, on calcule une règle d'apprentissage qui est à chaque présentation d'exemple X, Y, de mesurer la sortie S, l'erreur E, et de modifier chaque poids $w_{i,j}$ en le remplaçant par $w_{i,j} - \mu d_j s_i$ avec $d_n = (\sum d_j w_{n,j}) f'(e_n)$ pour les indices j en aval de n, sauf si n est en sortie auquel cas on a plutôt $d_n = 2(s_n - Y_n) f'(e_n)$. Il y a rétro-propagation de l'erreur commise dans la mesure où les modifications vont devoir être effectuées de la dernière couche

vers la première. Ce processus est répété sur plusieurs exemples jusqu'à ce qu'il y ait convergence suivant un seuil fixé. Voir Rumelhart D. Jordan M.I., *Internal world models and supervised learning,* Proceedings of the eight international workshop on machine learning, p.70-74, 1991.

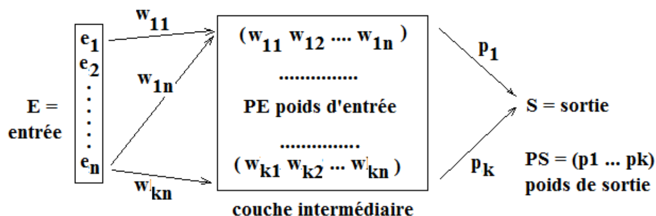
Les résultats expérimentaux sont laborieux mais efficaces dans toutes sortes d'applications concrètes en classification. Il se peut que des exemples présentés au début influencent trop le réseau et que des exemples présentés tardivement ne puissent être appris facilement.

Mais ce qui arrive aussi souvent est que le réseau a tendance à se conformer au dernier exemple présenté. C'est pourquoi on peut modifier la procédure d'apprentissage en présentant successivement tous les exemples avec une seule rétropropagation à chaque fois et si l'erreur est supérieure au seuil au moins une fois au cours de ce balayage, on refait un balayage complet.

Des résultats intéressants ont été obtenus surtout en reconnaissance des formes, prédiction de consonnes ou voyelles dans un mot, etc., mais le problème du choix des « bons » exemples et des paramètres du réseau (nombre de neurones par couches) reste entier.

Nous allons programmer en Prolog un réseau à trois couches.
La couche d'entrée accepte une liste de n nombres, la couche cachée contient k neurones et il n'y aura qu'une sortie, un résultat numérique dans [–1, 1].
Pour suivre le schéma et le programme, notons E une entrée, c'est-à-dire une liste (e_1, e_2, … , e_n) de n nombres. Comme chaque neurone d'entrée est relié à chacun des neurones de la couche cachée, nous allons noter (w_1, w_2, … ,w_n) la liste des poids entre tous les neurones d'entrée et le premier de la couche cachée.
De même, (w_{k1}, w_{k2}, … , w_{kn}) représentera la liste des poids entre tous les neurones d'entrée et le dernier des k neurones de la couche cachée. PE désigne la liste de ces listes.

Soit PS la liste des poids (p_1, p_2, … , p_k) entre les k neurones cachés et le neurone de sortie.



Propagation

Tout d'abord, pour initialiser une liste de *N* nombres réels entre –1 et 1, puis une liste de *K* telles listes, nous définissons les prédicats d'initialisation. La valeur des poids est arbitrairement dans cet intervalle de départ, par la suite, avec le jeu des modifications, elle peut être n'importe quel réel.
La fonction de *transfert* choisie est *th*, sa dérivée est notée *derivee*.
Il est nécessaire de définir le produit scalaire *pdscal* de deux vecteurs et *multscal* la multiplication d'un scalaire par un vecteur.

```
init1(0, []) :- !.
init1(N, [Z | V]) :- random(–100, 100, X), Z is X/100, M is N – 1, init1(M, V).

init2(0, _, []) :- !.
init2(K, N, [U | V]) :- init1(N, U), P is  K – 1, init2(P, N, V).

transfert(X, Y) :- Z is exp(2*X), Y is (Z – 1)/(Z + 1).
        % est la fonction tangeante hyperbolique

pdscal([], [], 0).
pdscal([U1 | U], [V1 | V], R) :- pdscal(U, V, P), R is P + U1*V1.

interm(_, [], []).
interm(E, [LP | PE], [SN | LS]) :- pdscal(E, LP, EN),
        transfert(EN, SN), interm(E, PE, LS).

propag(E, PE, PS, S) :- interm(E, PE, LS), pdscal(LS, PS, DN),
        transfert(DN, S).

essai(K, N, E) :- init2(K, N, PE), init1(K, PS), propag(E, PE, PS, S).
```

Le prédicat *interm* est une fonction qui calcule la liste des sorties de la couche cachée, appelée *LS*, en réalisant le produit scalaire du « vecteur » d'entrée *E* avec *LP* les poids correspondant à chaque neurone de cette couche. Chacun d'entre eux a donc une entrée numérique *EN* dont la fonction de transfert donne une sortie *SN*. La fonction de propagation consiste alors à effectuer le produit scalaire de ce « vecteur » avec celui *PS* des poids de sortie, puis en composant avec la fonction de transfert, d'obtenir la sortie numérique *S*, résultat de la propagation de l'entrée *E*.

Le prédicat *essai* n'a d'intérêt que pour vérifier le fonctionnement d'un aller simple.

Exemples

```
init1(5, L). → L = [–1.0, 0.12, –0.62, 0.61, 0.16]
init1(5, L). → L = [–0.05, –0.3, 0.79, 0.64, 0.49]

init2(2, 5, L). → L = [[–0.66, 0.71, 0.42, 0.02, –0.4],
          [–0.98, –0.82, –0.28, –0.71, –0.67]]
init2(3, 5, L).
→ L = [[–0.25, 0.06, 0.14, 0.2, 0.21], [–0.67, 0.32, –0.1, –0.29, –0.89],
          [0.2, 0.56, 0.6, 0.03, –0.4]]

transfert(–15, Y). → Y = –0.99999999999981293
transfert(–1, Y). → Y = –0.76159415595576485
transfert(0, Y). → Y = 0.0
transfert(0.5, Y). →Y = 0.46211715726000974
transfert(15, Y). → Y = 0.99999999999981282
transfert(50, Y). → Y = 1.0

pdscal([2, 3, 4], [3, 2, 5], X). → X = 32
pdscal([1, 2, 3, 4], [4, 3, 2, 1], X). → X = 20

essai(5, 3, [5, 2, 1], S). → S = –0.96046987030423536
```

Nous aurons besoin en outre de la dérivée de la fonction de transfert, du calcul des dérivées sur tous les éléments d'une liste et de la multiplication scalaire d'un vecteur par le nombre *M* et enfin de la somme des éléments d'une liste.

```
derivee(X, Y) :- Z is exp(2*X), Y is 4*Z / ((1 + Z)*(1 + Z)).

vecteurderive([], []).
vecteurderive([E | LE], [ DE | LDE])
        :- derivee(E, DE), vecteurderive(LE, LDE).

multscal(_, [], []).
multscal(M, [X | U], [Y | V]) :- Y is M*X, multscal(M, U, V).

add([], [], []). % addition de deux vecteurs
add([X | U], [Y | V], [Z | W]) :- Z is X + Y, add(U, V, W).

som([], 0).  % calcule la somme des éléments d'une liste
som([X | L], S) :- som(L, SP), S is SP + X.
```

derivee(0, D). → D = 1.0
derivee(–3, D). → D = 0.01
derivee(1, D). → D = 0.42
vecteurderive([–3, –1, 0, 2, 7], V).
→ V = [0.01, 0.42, 1.0, 0.070651, 0.000003326]
multscal(2, [2, 6, 3, 0, 7, 1], V). → V = [4, 12, 6, 0, 14, 2]
add([2, 4, 0, 3, 1], [–1, 1, 3, 0, 5], V). → V = [1, 5, 3, 3, 6]
som([2, 6, 3, 0, 7, 1], S). → S = 19

Rétropropagation

Un exemple est la donnée d'une entrée X et de sa sortie numérique attendue y. Pour un vecteur d'entrée $X = (x_1, x_2, \dots, x_n)$, le calcul de la propagation de X donne une sortie s, l'erreur est donc $s - y$.
La démarche est donc d'initialiser un réseau, c'est-à-dire les listes de poids *PE* et *PS*. Pour un exemple (X, Y) :
- calculer les k entrées $e_i = \sum_{j=1..k} w_{ji} x_j$ dans la liste *EI* de la couche intermédiaire
- en déduire les k sorties s_i (liste *SI*)
- calculer la valeur $E = \sum p_i s_i$ de l'entrée du dernier neurone
- en déduite la sortie $S = f(E)$
- calculer la dérivée $D = 2(s - y)f'(E)$
- modifier *PS* par $p_i \leftarrow p_i - \mu D s_i$
- modifier *PE* par le calcul de $d_i = (\sum d p_j) f'(e_i)$ puis $w_{ji} \leftarrow w_{ji} - \mu x_j d_i$
- recommencer, en affichant l'erreur et en décidant d'un arrêt de ces allers-retours.

Pour cela, nous devons revoir le prédicat *interm* afin d'isoler les valeurs de *EI* et *SI*, utiliser les prédicats *transfert* et sa dérivée *derivee*, ainsi que *modif1* tel que *modif1(U, M, V, W)* calcule le vecteur W comme $U - MV$ où M est un nombre réel. Le prédicat *modif2* applique *modif1* sur une liste de listes.

```
entrees(_, [], []).
entrees(X, [LP | PE], [E | EI]) :- pdscal(X, LP, E), entrees(X, PE, EI).

sorties([], []).
sorties([E | EI], [S | SI]) :- transfert(E, S), sorties(EI, SI).

modif1([], _, _, []).
modif1([X | U], M, [Y | V], [Z | W]) :- Z is X – M*Y, modif1(U, M, V, W).
```

```
modif2([], _, _, []).
modif2([U | LU], X, [D | LD], [V | LV])
        :- modif1(U, D, X, V), modif2(LU, X, LD, LV).
```

Le prédicat *ar* (aller-retour) pour un réseau de poids *PE* et *PS* et un couple-exemple (*X*, *Y*) va calculer les nouveaux poids *NPE*, *NPS*.
Dans cette longue exposition de la séquence de calculs à faire, *EI* et *SI* désignent les listes d'entrées et sorties de la couche cachée, *E* et *S* l'entrée et la sortie du dernier neurone, *DE* et *DEI* les dérivées de ces entrées et *DI* la liste des coefficients (le μ est dedans) tels que *j* de 1 à *k*, on fasse $w_{ji} \leftarrow w_{ji} - x_j d_i$ pour *i* de 1 à *n*.

```
ar(MU, X, Y, PE, PS, ER, NPE, NPS) :- entrees(X, PE, EI), sorties(EI, SI),
        pdscal(PS, SI, E), transfert(E, S), derivee(E, DE),
        write('erreur '), ER is abs(S – Y), write(ER), nl,
        D is 2*(S − Y)*DE, MUD is MU*D,
        modif1(PS, MUD, SI, NPS),
        som(NPS, SP), M is MU*SP*D, vecteurderive(EI, DEI),
        multscal(M, DEI, DI),
        modif2(PE, X, DI, NPE).
```

```
entrees([2, 1, 3], [[1, 2, 2], [3, 0, 1]], E).
→ E = [10, 9] % pour N = 3 entrées et K = 2

sorties([–1, 2, 0, 3], LS).
→ LS = [–0.761594156, 0.96402758, 0.0, 0.995]

modif1([1, 2, 3], 2, [5, 1, 4], V). → V = [–9, 0, –5]
Un exemple que l'on peut vérifier pour N = 3 et K = 2
modif2([[1, 2, 3], [5, 0, 2]], [2, 3, 4], [–1, 2], V).
→ V = [[3, 5, 7],[1, –6, –6]]

ar(7, [2, 11, 0], 0.25, [[12, –2, 3], [1, –14, 2]], [2, –3], ER, NPE, NPS).
→ erreur 0.749895153749749l3
ER = 0.74989515374974913
NPE = [[12.000311029479496, –1.9982893378627682, 3.0],
        [1.0, –14.0, 2.0]]
NPS = [1.997877839765432, –2.9977986519489401]

% on constate des modifications minimes des poids
```

Maintenant, dans l'espoir de réduire l'erreur, il faut enchaîner ces allers-retours. Un programme qui réalise cela doit avoir comme arguments de départ les nombres *N* de neurones d'entrée et *K* de neurones cachés, et la valeur ε de l'erreur acceptée.
On relancera le cycle des propagations–rétropropagations tant que l'erreur n'est pas en valeur absolue inférieure à ε. Avant de l'écrire, l'essentiel sera une boucle.

```
boucle(MU, X, Y, PE, PS, EPS)
        :- ar(MU, X, Y, PE, PS, ER, _, _), ER < EPS,
        write('fini avec le réseau '), write(PE), write(PS), !.

boucle(MU, X, Y, PE, PS, EPS)
        :- ar(MU, X, Y, PE, PS, ER, NPE, NPS),
        boucle(MU, X, Y, NPE, NPS, EPS).

reseau(N, K, MU, X, Y, EPS)
        :- init2(K, N, PE), init1(K, PS), boucle(MU, X, Y, PE, PS, EPS).
```

```
Exemple
boucle(1, [–0.2, 1, 2], 0.5, [[1.2, –0.2, 0.03], [–0.1, –1.4, 0.02]],
        [0.2, –0.03], 0.1). →
erreur 0.54635807428125882
erreur 0.54635807428125882
erreur 0.32424440047343117
erreur 0.32424440047343117
erreur 0.1390279337277347
erreur 0.1390279337277347
erreur 0.095071663155207542
fini avec le réseau [[1.4227, –1.3135, –2.197],
        [–0.039, –1.7, –0.59]], [0.177, –0.61]

% la convergence est loin d'être toujours aussi rapide
```

Application à la classification

Nous prenons un problème très simple où les deux exemples fournis seront censés représenter deux extrêmes uniformément à zéro ou à un, ainsi le premier vecteur $X1 = [0, 0, 0, 0, 0]$ doit-il renvoyer la sortie $y1 = -1$ et le second vecteur $X2 = [1, 1, 1, 1, 1]$ doit renvoyer $y2 = 1$.
On choisit $N = 5$ et $K = 7$.

La programmation n'est pas très élégante en dédoublant les opérations à faire sur les exemples, on peut naturellement mettre les exemples dans une liste.
De même que le réseau pourrait avoir plusieurs couches et la rétropropagation se faire de façon plus rationnelle de couche en couche, mais outre le fait que ce simple « perceptron » peut suffire, moyennant une couche cachée comportant beaucoup de neurones, la mise au point est très délicate.
En effet le pas μ n'est pas facile à régler, il peut en étant trop fort, faire osciller l'erreur dans une sorte de période, sans jamais converger. Trop faible, les modifications restent très minimes.

A cela s'ajoute un problème de fond qui est que l'apprentissage sur un exemple, puis sur un autre risque de désapprendre les exemples qui précèdent. C'est pourquoi on peut choisir de cumuler les erreurs sur une propagation appliquée à tous les exemples (apprentissage par lots), puis une rétropropagation unique sur le cumul de l'erreur.

Néanmoins l'apprentissage est très long et laborieux.

```
arbis(MU, X1, Y1, X2, Y2, PE, PS, ER, NPE, NPS)
    :- entrees(X1, PE, EI1), sorties(EI1, SI1),
    pdscal(PS, SI1, E1), transfert(E1, S1), derivee(E1, DE1),
    entrees(X2, PE, EI2), sorties(EI2, SI2),
    pdscal(PS, SI2, E2), transfert(E2, S2), derivee(E2, DE2),
    write('erreur '), ER is abs(S1 – Y1) + abs(S2 – Y2), write(ER), nl,
    D is 2*(S1 + S2 – Y1 – Y2)*(DE1 + DE2), MUD is MU*D,
    add(SI1, SI2, SI),
    modif1(PS, MUD, SI, NPS), som(NPS, SP), M is MU*SP*D,
    vecteurderive(EI1, DEI1), vecteurderive(EI2, DEI2),
    add(DEI1, DEI2, DEI), multscal(M, DEI, DI), add(X1, X2, X),
    modif2(PE, X, DI, NPE).

bouclebis(MU, X1, Y1, X2, Y2, PE, PS, EPS) :-
    arbis(MU, X1, Y1, X2, Y2, PE, PS, ER, _, _), ER < EPS,
    write('fini avec le réseau '), write(PE), write(PS), !.
bouclebis(MU, X1, Y1, X2, Y2, PE, PS, EPS) :-
    arbis(MU, X1, Y1, X2, Y2, PE, PS, ER, NPE, NPS),
    bouclebis(MU, X1, Y1, X2, Y2, NPE, NPS, EPS).

reseau(N, K, MU, X1, Y1, X2, Y2, EPS)
    :- init2(K, N, PE), init1(K, N, PS),
    bouclebis(MU, X1, Y1, X2, Y2, PE, PS, EPS).
```

```
reseau(5, 7, 7, [0, 0, 0, 0, 0], –1, [1, 1, 1, 1, 1], 1, 0.5). →
donne le réseau
[[0.190864061301686, –0.229135938698313, –0.649135938698313,
–0.499135938698313, –0.0991359386983135], [0.631945679626780,
0.0619456796267802, 1.00194567962678, 0.53194567962678,
–0.27805432037321], [–0.516207614213241, 0.463792385786759,
0.133792385786759, –0.726207614213240, 0.543792385786759],
[0.281659801056644, –0.428340198943355, 1.17165980105664,
–0.43834019894335, 1.09165980105664], [0.679689727329535,
–0.100310272670464, 0.149689727329535, –0.560310272670464,
1.22968972732953], [0.450799130222028, –0.609200869777971,
0.84079913022202, 0.730799130222028, 0.250799130222028],
[–0.55221869903766, –0.582218699037669, 0.69778130096233,
0.0177813009623306, –0.622218699037669]]

[–0.604170010910256, 1.08645259389529, –0.65457598212530,
–0.275380262188636, 0.0639819442336674, 0.914199503825363,
–0.0735242312258354]
```

En affectant ce réseau sous le nom de résultat, on définit l'unique clause *resultat* on demandera de vérifier ce prédicat *resultat*(*PE, PS*). puis en relançant des propagations, on obtient des résultats moyens (en généralisation) mais pas si mal pour une première approche. Ils mesurent très grossièrement entre –1 et 1, le poids total du vecteur entre –5 et 5.

```
resultat(PE, PS), propag([1, 1, 1, 1, 1], PE, PS, Y).
→ Y = 0.9782044
resultat(PE, PS), propag([1, 0, 1, 0, 1], PE, PS, Y).
→  Y = 0.2312076
resultat(PE, PS), propag([0, 0.5, 0.1, 1, 1], PE, PS, Y).
→  Y = 0.003754
resultat(PE, PS), propag([–0.6, 0, –0.5, –0.7, –1], PE, PS, Y).
→  Y = –0.4185303
resultat(PE, PS), propag([–0.5, –0.7, –1, –1, –1], PE, PS, Y).
→  Y = –0.673114
resultat(PE, PS), propag([–1, –1, –1, –1, –1], PE, PS, Y).
→  Y = –0.9578021
```

*

*   *