

Modules complémentaires à MATLAB

MATLAB Compiler : Transcription de M-files en langage C - Mise au point de MEX-files - Mise au point de source C au code C ANSI ou K&R indépendant de MATLAB.

Neural Network Toolbox : Réseaux associatifs, à rétropropagation, de Hopfield, de Kohonen, à auto-organisation, à base radiale et autres réseaux - Fonctions compétitives, limites, linéaires, de transfert sigmoïde - Architectures récurrentes, retroactives - Fonctions d'analyse des performances et graphes - Taille de réseaux illimitée - Blocs 'réseaux de neurones' sous SIMUUNK.

Control System Toolbox : Techniques modernes et classiques - Temps discret et continu - Modèles sous forme de fonctions de transfert et d'équations d'état - Interconnexion système - Transformation entre modèles - Réduction de modèles - Réponse fréquentielle: Bode, Nyquist, Nichols, valeurs singulières - Réponses temporelles pour une impulsion, un échelon, un signal quelconque - Lieu des racines, placement des pôles, commande optimale quadratique.

System Identification Toolbox : MA, AR, ARMA, Box-Jenkins - Modèles sous forme de fonctions de transfert et d'équation d'état - Techniques récursives et non récursives - Validation de modèle - Sélection de l'ordre manuel ou automatique - Traitement du signal.

Signal Processing Toolbox : Conception de filtres numériques et analogiques, implémentation, et simulation de réponse - Analyse et estimation spectrales, FFT, DCT et autres transformées - Modélisation paramétrique - Traitement de signaux multi-échantillonnés - Modulation et démodulation.

Image processing Toolbox : Filtrage et conception de filtres 2-D - Amélioration et restauration d'images - Opérations morphologiques, géométriques, et coloration - Transformations 2-D - Analyse d'images et statistiques.

Optimization Toolbox : Programmation linéaire et programmation quadratique - Calcul du minimum ou du maximum d'une fonction - Moindres carrés non linéaires - Problèmes semi-infinis et minimax - Optimisation multi-objectif - Optimisation sous contraintes - Solution d'équations.

Symbolic Math Toolbox : Calcul symbolique, algèbre linéaire, et résolution d'équations - Simplification d'expressions - Fonctions spéciales - Précision arithmétique variable - Accès au noyau Maple, y compris à la bibliothèque de base d'algèbre linéaire - L'"Extended Toolbox" ajoute la programmation et toutes les fonctions.

Statistics Toolbox : Bêta, binomiale, chi-deux, Poisson, et autres distributions - Génération de nombres aléatoires - Boxplots, probabilité, quantile-quantile, et autres graphiques - Régression, polynômes, et analyse de la variance - Statistiques descriptives - Tests d'hypothèse - Outils d'analyse commandables graphiquement .

Math Library : Contient la librairie C des routines du noyau MATLAB - Est utilisable en standalone (sans MATLAB) ou en liaison avec le compilateur MATLAB.

Nonlinear Control Design Blockset : Mise au point automatique de lois de commandes pour les systèmes non linéaires - Intégration complète aux schémas-blocs de SIMULINK - Editeur graphique pour les spécifications des paramètres de performance temporelle - Solution d'équations non linéaires.

DSP Blockset : Bibliothèque de blocs SIMULINK permettant la conception, la simulation et la mise au point de systèmes en traitement du signal - Plus de 100 blocs permettant la prise en compte dans les modèles SIMULINK des opérations élémentaires de traitement du signal (buffers, FFI, etc.), arithmétique complexe, mathématiques vectorielles et statistiques, fenêtrage, conception de filtres et filtrage, génération et visualisation de signaux... - Compatibilité avec le Real-Time Workshop pour la génération de code C et l'implantation sur cible - Complémentaire à la Signal Processing Toolbox de MATLAB.

Real-Time Workshop : Génération automatique de code C à partir de schémas-blocs SIMULINK en temps discret, temps continu ou systèmes hybrides - Code optimisé pour une exécution rapide, portable lisible et bien documenté - Tableau de commande API - Choix de paramètres: paramètres en ligne ou vectoriels, code C

ANSI ou K&R, simulation en temps réel ou fixes, validation de code automatisée, algorithmes d'intégration - Code C généré utilisable avec pratiquement tout système temps réel du commerce ou propriétaire.

SIMULINK Accelerator : Accélère les simulations sous SIMULINK en transformant vos modèles SIMULINK en fichiers exécutables (nécessite le compilateur WATCOM C) Supporte les systèmes en temps continu ou discret, les systèmes hybrides et multiéchantillonnés - Permet de changer les paramètres avant ou en cours de simulation, sans nouvelle compilation - Compare la structure de votre modèle SIMULINK avec le dernier modèle compilé pour savoir si une nouvelle compilation est nécessaire - Supporte tous les algorithmes d'intégration de SIMULINK.

Fixed-Point Blockset : Bibliothèque de blocs SIMULINK permettant l'utilisation de composants en virgule fixe. Les blocs fournis incluent : opérations arithmétiques, gains et constantes, conversions virgule flottante/virgule fixe, tables d'interpolation 1 D et 2D, opérations logiques, relationnels, conversion/saturation, retards... Compatible avec le RealTime Workshop pour la génération de code C et l'implantation sur cible.

LMI Control Toolbox : Fonctions d'optimisation pour la résolution d'inégalités matricielles - Fonctions spécialisées pour l'analyse et la conception de contrôleurs par une approche LMI.

Model Predictive Control Toolbox : Conception et tests de commandes adaptatives - Inclut les principales approches pour la spécification du problème d'optimisation et l'estimation d'état.

Quantitative Feedback Theory Toolbox : Conception pratique de systèmes bouclés robustes - Prise en compte des spécifications de performances et des incertitudes sur le procédé dans le domaine fréquentiel - Interface Utilisateurs Graphique complète pour la formulation de contrôleurs simples et d'ordre peu élevé, qui remplissent les critères spécifiés.

Robust Control Toolbox : Synthèse optimale LQG/LTR - Synthèse optimale H2 et H(infini) - Réduction de modèles par valeurs singulières - Factorisation spectrale.

Mu-Analysis and Synthesis Toolbox : Synthèse et analyse Mu - Synthèse optimale H2 et H(infini) - Théorie de l'analyse et de la synthèse mu.

Frequency Domain System Identification Toolbox : Identification de systèmes dynamiques linéaires à partir de mesures de l'excitation et de la réponse fréquentielle.

MMLE3 Identification Toolbox : Identification de modèles d'états à partir de données d'entrée-sortie - Algorithmes avancés mis au point à la NASA

Hi-Spec Toolbox : Estimation de spectres d'ordre supérieur - Approches conventionnelles et paramétriques - Algorithmes avancés en traitement du signal.

Spline Toolbox : Polynômes par morceaux et B-splines - Manipulation et construction de Splines - Fitting et lissage de courbes - différenciation de fonctions, intégration et évaluation.

Fuzzy Logic : Conception de systèmes flous - Apprentissage neuroflou, adaptatif - Classification - Systèmes d'inférences de type SUGENO - Génération de code C - Blocs Logique Floue sous SIMULINK.

Chemometrics Toolbox : Développement et évaluation de matrices de calibration - Développement de méthodes d'analyse quantitative - Régression linéaire multiple - Analyse en composantes principales - Moindres carrés partiels.

PDE Toolbox : Résolution de systèmes d'équations aux dérivées partielles en 2-D par les méthodes des éléments finis - Equations elliptiques, paraboliques, hyperboliques, aux valeurs propres - Equations à coefficients complexes, constants ou variables en fonction de la position et de la solution - Conditions limites de Neumann, Dirichlet ou mixtes - Interface Utilisateurs évoluée permettant de traiter les applications standard suivantes: mécanique des structures, électrostatique, magnétostatique, électromagnétisme, milieux conducteurs, transferts de chaleur, diffusion - On dispose, en particulier, d'un meilleur adaptatif, d'un solveur non-linéaire, d'outils de traitement des matrices creuses et de solveurs d'équations différentielles adaptés.

NAG Foundation Toolbox : Plus de 200 fonctions NAG (fonctions de la NAG Foundation Library) intégrées et utilisables depuis MATLAB - Nombreux sujets couverts: optimisation, équations différentielles ordinaires

et aux dérivées partielles, intégration numérique, statistiques, etc. - Nom de fonctions et syntaxes d'appel dérivés de ceux utilisés dans NAG.

Financial Toolbox : Fournit un ensemble robuste de fonctions financières indispensables à l'analyse quantitative et financière - Les applications comprennent la détermination du prix des titres, le calcul d'intérêts et de taux, l'analyse des produits dérivés et l'optimisation de portefeuilles.

Structural Dynamics Toolbox : Identification modale à partir de données expérimentales - Analyse de modèles aux éléments finis définis provenant d'outils tels que NASTRAN... - Recalage de ces modèles par rapport à des données expérimentales - Interface Utilisateurs Graphique permettant la visualisation et l'analyse de fonctions de réponses fréquentielles, l'identification de modes normaux et complexes, la visualisation et l'estimation des déformations de structures.

Comment compiler un programme MATLAB

1- Introduction

2- Pourquoi compiler des fonctions MATLAB ?

3 - Création d'une fonction MATLAB compilée 'MEX-files'

- a) Mécanisme de base
- b) Compilation simultanée de plusieurs fonctions
- c) Compilation d'une fonction SIMULINK

4 - Création d'une application externe indépendante de MATLAB.

- a) Mécanisme de base
- b) Génération de programmes compilés C et C++
- c) Génération de programmes C pour utilisation dans une autre application C.
- d) Génération de fonctions objets

5 - Limitations et Restrictions

1- Introduction

Le compilateur MATLAB est un produit assez jeune qui évolue assez vite. La version actuelle est la version 2.0 . Le compilateur génère un code source C ou C++ à partir d'une fonction MATLAB. Ce code C est ensuite compilé afin d'obtenir:

- soit une fonction MATLAB compilée ('MEX-file').
- soit une application externe qui puisse fonctionner sans MATLAB.

2- Pourquoi compiler des fonctions MATLAB ?

Il y a au de bonnes raisons pour compiler une fonction MATLAB:

- Pour augmenter la vitesse
- Pour créer une application indépendante de MATLAB.

Un code C compilé tourne plus vite que son équivalent en MATLAB car :

- Un programme compilé tourne plus vite qu'un programme interprété.
- Un code C contient des données de types plus simple. Pour MATLAB toutes les données sont des matrices (des tableaux).
- MATLAB vérifie la taille des tableaux à chaque affectation d'élément ce que l'on peut éviter de faire en C.

- MATLAB doit ré-allouer de la mémoire en cours d'exécution la ou en C on peut l'éviter.

La compilation d'une fonction MATLAB n'apportera pas forcément de gain si :

- la fonction MATLAB est fortement vectorisée.
- la fonction MATLAB passe beaucoup de temps à utiliser des fonctions internes MATLAB mathématiques (très rapide) et graphiques (très lent).

La compilation apportera des gains de temps si la fonction MATLAB :

- contient des boucles (for,while).
- contient des variables que le compilateur traduit en 'real' ou en 'integer'.
- ne travaille que sur des réelles.

3 - Création d'une fonction MATLAB compilée 'MEX-files'

a) Mécanisme de base

Le compilateur MATLAB '**mcc**' transforme une fonction MATLAB en code source C qui est ensuite compilé par l'outil '**mbuild**'. Celui-ci génère un code compilé de la fonction (avec comme extension '.mexalp') qui sera automatiquement utilisé à la place de la fonction MATLAB correspondante.

Soit la fonction MATLAB suivante '**carrel**' qui va nous servir d'exemple:

```
function mat = carrel(n);
% calcul test
for i=1:n, mat(i)=sqrt(i)*i; end
```

Pour compiler on tape sous MATLAB :

```
>> mcc -x carrel.m
```

On obtient un code source C 'carrel.c' et un exécutable 'carrel.mexalp' qui sera utilisé à la place de 'carrel.m'. Ces deux fichiers se trouvent dans le répertoire courant. L'utilisation de l'outil 'mbuild' est transparent pour l'utilisateur.

Pour lancer la fonction on tape par exemple :

```
>> carrel(100)
```

Si l'on désire des informations sur la compilation l'option '**-v**' permet d'avoir toutes les informations en cours de la compilation.

```
>> mcc -vx carrel.m
```

b) Compilation simultanée de plusieurs fonctions

Si une fonction fait appel à une ou plusieurs autres fonctions que vous avez écrites, vous pouvez les compiler en même temps. Attention, il faut que la fonction principale soit la première citée parmi les arguments pour le compilateur. Par exemple :

```
>> mcc -x fonction-principale.m fonction1.m fonction2.m
```

On génère un fonction compilée 'fonction-principale.mexalp', il n'y a pas de 'fonction1.mexalp' et 'fonction2.mexalp'

c) Compilation d'une fonction SIMULINK

Pour compiler on tape sous MATLAB :

```
>> mcc -S mafonction.m
```

On obtient un exécutable 'mafonction.mexalp' utilisable seulement MATLAB.

4 - Création d'une application externe indépendante de MATLAB.

a) Mécanisme de base

Si l'on invoque le compilateur MATLAB avec la bonne option '**mcc -c**', celui-ci génère un code C qui peut être intégré dans vos propres applications C.

Après la compilation de votre application C à l'aide de l'outil 'mbuild' les différents programmes objet sont liés avec les bibliothèques suivantes :

- 'MATLAB Math built-In Library' qui contient la version compilée de la plupart des routines mathématiques internes de MATLAB.
- 'MATLAB Math Toolbox Library' qui contient la version compilée des routines mathématiques externes de MATLAB (script).
- 'MATLAB Compiler Library' qui contient des routines spéciales pour la manipulation de certaines structures de données.
- 'ANSI C Math library' bibliothèque mathématique standard fournie par le constructeur de la machine.

Vous obtenez alors un code exécutable sur toutes les plateformes COMPAQ/DEC du Centre. Attention ! avant de lancer votre programme sous UNIX il faut définir le chemin où le programme trouvera les bibliothèques Matlab par la commande:

```
export LD_LIBRARY_PATH=/usr/local/matlab531/extern/lib/alpha
```

Voyons un exemple, prenons la fonction 'carre2.m'.

```
function mat=carre2;  
% calcul test  
n=10000;  
for i=1:n, mat(i)=sqrt(i)*i; end
```

b) Génération de programmes compilés C et C++

Pour générer un code exécutable C qui ne contient pas d'appel à des fonctions graphiques mais seulement de calcul il suffit de taper la commande:

```
>> mcc -m carre2.m
```

De même pour générer un code C++ on tape sous Matlab:

```
>> mcc -p carre2.m
```

c) Génération de programmes C pour utilisation dans une autre application C.

Pour générer le code C approprié on tape :

```
>> mcc -t -L C carre2.m
```

On obtient un programme C 'carre2.c' qui peut être intégré dans un programme C externe.

Si on a écrit un programme C 'main.c' qui utilise la fonction 'carre2' on compilera le tout en tapant sous UNIX:

```
$ mbuild -o monprog monprog.c carre2.c
```

ou sous MATLAB par la commande

```
>> mcc -T link.exe monprog.c carre2.c
```

On peut maintenant exécuter le programme 'monprog' sur n'importe quel DEC/ALPHA du Centre.

d) Génération de fonctions objets

Le généré de fonctions objets '.o' est intéressante à plusieurs titres :

- La première raison est logique on ne peut pas générer un exécutable en compilant une fonction qui n'a pas de programme principale.
- Si on développe une application assez importante, il est possible de ne compiler que les objets modifiés d'où un gain de temps à la compilation.
- On peut ne fournir à un autre utilisateur que les fonctions objets ce qui assure une bonne confidentialité (c'est du binaire)

Voici un exemple pour obtenir des fonctions objets :

Ici on compile la fonction `carre1` qui génère un programme C et un fichier objet. Puis on génère le programme C principal qui utilise cette fonction (juste pour éviter d'écrire le programme C) et on le compile ce qui donne un autre fichier objet

```
>> mcc -v -t -L C carre2 # Generation carre2.c
>> mcc -v -T compile:exe carre2.c # Generation carre2.o
>> mcc -v -W main -L C carre2 # Generation carre2_main.c
>> mcc -v -T compile:exe carre2_main.c # Generation carre2_main.o
```

Puis on génère l'exécutable à partir des fichiers objets :

```
>> mcc -T link:exe carre2.o carre2_main.o
```

ou sous UNIX par la commande :

```
$ mbuild -o carre2.o carre2_main.o
```

5 - Limitations et Restrictions du compilateur version 2.0

Il y a quelques restrictions qui font que le compilateur MATLAB ne peut pas transformer tous les codes MATLAB en code C.

Le compilateur MATLAB ne peut pas compiler :

- un script MATLAB, il faut que ce soit une fonction MATLAB, souvent il suffit de rajouter 'fonction f=nomfonction' en première ligne.
- un code contenant 'eval','input'.
- un code qui utilise les objets MATLAB.

a) Restrictions supplémentaires concernant les applications externes

Les applications externes ne peuvent accéder :

- aux fonctions de debug MATLAB comme 'dbclear'.
- aux fonctions graphiques comme 'surf','plot','get' et 'set' (bibliothèque non disponible pour le moment)
- une application ne peut s'exécuter que sur une plateforme du même type, en l'occurrence DEC/COMPAQ soit les machines Berg, Webern et Schoenberg.

b) Restrictions levées par la version 2 du compilateur

Le compilateur version 1.2 avait des restrictions très contraignantes qui sont supprimées, qui sont :

- l'utilisation de load et save
- l'utilisation des fonctions SIMULINK
- l'optimisation du code est automatique et n'a plus besoin d'être précisé par des options.

6 - Documentation

Pour plus de renseignements tapez sous MATLAB '**help mcc**' et '**help mbuild**'

Utilisation de la Boîte à outils Matlab Réseaux de Neurones

1- Généralités

- a) Mécanisme de base
- b) Création et apprentissage d'un réseau
- c) Simulation (ou activation) d'un réseau

2- Différents types de réseaux prédéfinis

- a) Apprentissage compétitif (non-supervisé, parfois appelé VQ)
- b) LVQ (apprentissage compétitif supervisé)
- c) Cartes auto-organisatrices
- d) Perceptron multicouches avec apprentissage par rétro-propagation
- e) Réseau probabiliste
- f) Réseaux à fonctions radiales de base (FRB)
- d) Réseaux récurrents

1 - Généralités

a) Génération de données de test

La fonction `nngenc` génère des points distribués aléatoirement selon des courbes gaussiennes. Ces données permettent de tester rapidement des algorithmes de classification.

Exemple d'utilisation (voir la démo d'apprentissage compétitif `democ1`) :

```
x = [0 1; 0 1];
clusters = 8; points = 10; std_dev = 0.05;
P = nngenc(X,clusters,points,std_dev);
plot(P(1,:),P(2,:),'+r');
```

La fonction `nngenc` génère dans cet exemple 8 distributions gaussiennes de 10 points chacune en 2 dimension (cette dernière information est programmée au moment de la définition de X). L'écart-type σ des gaussiennes est 0,05. Les centres des gaussiennes sont compris entre 0 et 1.

Remarque : la fonction `nngenc` est non documentée dans Matlab.

Structure Matlab des réseaux

Il s'agit d'une structure hiérarchique (certains membres de la structure de base sont eux-mêmes des structures) relativement complexe. Elle peut être visualisée lors de la création du réseaux ou plus tard. Exemple :

```
net=newc(...paramètres...)
```

→ pas de point-virgule donc affichage

Autre solution :

```
net=newc(...paramètres...) ;
net
```

b) Création et apprentissage d'un réseau

La fonction de création d'un réseau est spécifique au modèle de réseau utilisé (`newc`, `newlvq`, etc).

Il existe 2 types d'apprentissage :

- Incrémental : fonction `adapt`
- Par paquets : fonction `train`

Apprentissage incrémental (en-ligne, *on-line*) : les poids sont modifiés à chaque présentation d'une entrée

Apprentissage par paquets (hors-ligne, *off-line, batch mode*) : les poids sont modifiés uniquement après présentation de toutes les entrées.

c) *Simulation (ou activation) d'un réseau*

`a=sim(net, p) ;`

où `net` est le pointeur retourné par une fonction de création de réseau.

2 - Différents types de réseaux pré-définis

a) *Apprentissage compétitif (non-supervisé, parfois appelé VQ)*

Création d'un réseau pour l'apprentissage compétitif : fonction `newc`

Exemple : création d'un réseau de 2 neurones à 2 entrées :

`net=newc([0 1 ; 0 1], 2) ;`

Le 1^{er} argument indique la plage de variation des 2 entrées.

Dans le cas d'un grand nombre d'entrées... ?

Apprentissage de base (règle de Kohonen) : `learnk`

Apprentissage avec paramètre de conscience : `learncon`

L'apprentissage est lancé soit par la fonction `adapt` (incrémental) soit par `train` (par paquets). Ces 2 fonctions auront le même effet.

Paramètres par défaut :

Les poids sont initialisés à la moitié de la plage d'entrée

Les biais sont initialisés avec la fonction `initcon`, qui initialise les biais en vue de l'utilisation du mode d'apprentissage avec paramètre de conscience.

Fonction d'apprentissage : règle de Kohonen : `learnk`

Démo : `democ1`

b) *LVQ (apprentissage compétitif supervisé)*

Création d'un réseau pour le LVQ : `newlvq`

Liste des fonctions et démo : `help lvq`

Démo : `demolvq1`

Rappel : algorithmes de base VQ et LVQ :

-application d'une entrée

-recherche du neurone vainqueur

-modification des poids

(cas VQ : rapprochement de l'exemple

cas LVQ : rapprochement ou éloignement selon que le neurone vainqueur est le neurone désiré ou non)

Dans Matlab le réseau pour le LVQ est programmé au moyen de 2 couches :

- une couche de compétition pour la détermination du neurone vainqueur (la sortie de ce dernier est mise à 1, celle des autres neurones à 0)
- une couche de neurones linéaires pour la classification

La 2^e couche comporte un neurone par classe, la 1^{ère} couche comporte un neurone par sous-classe.

L'apprentissage modifie les poids de la première couche. Les neurones de la 2^e couche combinent certaines sorties des neurones de la 1^{ère} au moyen de fonctions OU.

Exemple

10 vecteurs d'entrée appartenant à 2 classes. Les vecteurs d'entrées sont disposés de telle façon qu'il faut au moins 4 sous-classes. Chaque classe sera l'union de 2 sous-classes.

Ces informations se traduisent donc par 4 neurones dans la 1^{ère} couche et 2 dans la 2^e.

Points choisis, en dimension 2 (matrice 2 lignes, 10 colonnes) :

```
P=[-3 -2 -2 0 0 0 0 +2 +2 +3 ; 0 +1 -1 +2 +1 -1 -2 +1 -1 0] ;
```

Cibles (indices de classe de chaque point) :

```
Tc=[1 1 1 2 2 2 2 1 1 1] ;
```

Création du réseau

```
net=newlvq((minmax(P), 4, [.6 .4]));
```

1^{er} argument : plage de variation des entrées (affichage par `minmax(P)`)

2^e argument : nombre de neurones dans la couche de compétition

3^e argument : pourcentage d'appartenance des entrées pour chacune des 2 classes (l'information du nombre de classes fait partie de cet argument)

Par défaut les poids de la 1^{ère} couche sont initialisés avec la fonction `midpoint`. Celle-ci initialise à la moitié de la plage de variation : ici à 0. On le vérifie en entrant :

```
net.IW{1,1}
>ans=
    0 0
    0 0
    0 0
    0 0
```

Les poids entre la 1^{ère} et la 2^e couche sont fixes. Affichage :

```
net.LW{2,1}
>ans=
    1 1 0 0
    0 0 1 1
```

Activation du réseau (simulation dans la terminologie de Matlab)

```
Y=sim(net,P)
```

La fonction `sim` retourne le résultat sous forme de 10 vecteurs de dimension 2 : un 0 représente un neurone inactif, un 1 un neurone actif.

A cette première activation toutes les entrées sont classées dans la 1^{ère} classe. En effet l'apprentissage n'a pas encore eu lieu.

Pour convertir ces vecteurs en indices (l'indice 1 correspondra au 1^{er} neurone actif et l'indice 2 au 2^e :

```
Y2=vec2ind(Y);
```

Affiche 10 fois le chiffre 1.

Apprentissage

On commence par fixer des paramètres d'apprentissage :

```
net.trainParam.epochs=1000;      %100 par défaut  
net.trainParam.lr=0.05 ;        %pas de valeur par défaut
```

puis on lance l'apprentissage :

```
net=train(net,P,T) ;
```

Affichage des poids résultats :

```
net.IW{1,1}
```

Affichage des résultats de classification :

```
Y=sim(net,P)
```

Affichage graphique des entrées et des poids :

Généralisation (=activation avec des données non-apprises) :

On choisit une entrée proche d'un vecteur d'apprentissage :

```
vec=[0 ; 0.5];  
Y=sim(net, vec);  
Yc1=vec2ind(Y)  
>Yc1=2
```

Ce vecteur d'entrée a donc été classé dans la 2^e classe, ce qui est correct.

Conditions à respecter pour l'apprentissage compétitif :

La couche de compétition doit posséder suffisamment de neurones, et chaque neurone de classe être connecté à suffisamment de neurones de la couche de compétition.

c) Cartes auto-organisatrices

Création du réseau

Elle est commandée par la fonction `newsom`. Le premier argument de cette fonction contient les bornes des entrées, et le second le nombre de neurones du réseau ainsi que leur arrangement. Par exemple :

```
net=newsom([0 2 ; 0 1], [2 3]) ;
```

créé un réseau de 2 x 3 neurones, dont les entrées à 2 dimensions peuvent être comprises entre 0 et 2 pour la 1^{ère} dimension et 0 et 1 pour la 2^e.

Types de voisinages

Il existe 3 types de topologies de réseaux :

- grille rectangulaire (`gridtop`)
- hexagonale (`hextop`)
- aléatoire (`randtop`)

Par défaut, le type sélectionné est le type hexagonal. Pour modifier ce paramètre, il suffit de forcer le 3^e paramètre de la fonction `newsom`. Par exemple pour forcer le type `gridtop` :

```
newsom([0 2 ; 0 1], [2 3], 'gridtop') ;
```

Principe de l'apprentissage en 2 phases

La phase d'organisation globale (*ordering phase*) s'étend sur un nombre d'étapes égal à `LP.order_steps` ; la valeur par défaut est 1000. Le taux d'apprentissage est égal au paramètre `LP.order_lr`, dont la valeur par défaut est 0,9, et décroît au cours de l'apprentissage jusqu'à la valeur définie par `LP.tune_lr`, égale à 0,02 par défaut. La distance de voisinage est initialement égale à la distance maximale entre 2 neurones, puis décroît au cours de l'apprentissage jusqu'à la valeur définie par `LP.tune_nd`, égale à 1 par défaut.

La phase de réglage fin (*tuning phase*) prend le relais de la phase d'organisation, avec comme taille de voisinage la valeur définie par `LP.tune_nd`. Le taux d'apprentissage commence avec la valeur définie par `LP.tune_lr`, puis décroît beaucoup plus lentement que dans la phase précédente. Le nombre de cycles pour cette phase doit être beaucoup plus grande que dans la phase précédente.

Pour modifier ces paramètres, afficher la syntaxe de la fonction `newsom` à l'aide de la ligne de commande `help newsom`.

Lancement de l'apprentissage

L'apprentissage est commandé par l'appel de la fonction `train`. L'apprentissage de Kohonen est programmé par l'utilisation de la fonction `learnsom`. La modification des poids intervient après chaque présentation d'une nouvelle entrée, indépendamment du type de fonction d'apprentissage sélectionné : `trainwb1` (apprentissage en mode *batch*) ou `adaptwb` (apprentissage en mode instantané).

L'apprentissage affecte le neurone vainqueur et ses voisins de la façon suivante : les poids du vainqueur sont modifiés avec le taux d'apprentissage défini, et ses voisins sont modifiés avec un taux d'apprentissage égal à la moitié de celui-ci.

Affichage de l'état du réseau

La fonction `plotsom`, avec comme argument un réseau déjà créé, permet d'afficher les relations de voisinage entre les neurones sous forme de grille.

Démos

Demosm1

Demosm2

d) Perceptron multicouches avec apprentissage par rétro-propagation

La fonction de création d'un réseau est `newff`, pour feed-forward (car dans ces réseaux l'activité ne se propage que de l'entrée vers la sortie). Exemple de syntaxe :

```
net=newff([-1 2 ; 0 5], [3,1], {'tansig', 'purelin'}, 'traingd') ;
```

Cette commande crée le réseau et initialise ses poids.

4^e argument : algorithme d'apprentissage. Différents types :

`learngd` : apprentissage par descente de gradient

`learngdm` : version avec moment de `learngd`

`traingd` : même chose que `learngd` mais en mode off-ligne (*batch learning*)

`traingdm` : version avec moment de `traingd`

e) Réseau probabiliste

La fonction `newpnn` crée un réseau probabiliste. Il s'agit d'un réseau composé de 2 couches :

- la première est une couche de neurones à fonctions radiales de base (FRB), ce qui signifie 2 choses :
 - ∅ les neurones calculent la distance entre un vecteur d'entrée et leur vecteur poids ;

Ø leur fonction de transfert est une gaussienne

- la seconde est une couche de compétition : elle prend les sorties de la couche à FRB, remplace la sortie maximale de cette couche par un 1 et les autres par un 0.

Cette méthode fonctionne en apprentissage supervisé, c'est à dire que l'apprentissage nécessite comme information un ensemble de vecteurs d'entrée associés avec une sortie désirée du réseau.

La fonction `newpnn` crée dans la 1^{ère} couche 1 neurone par exemple d'apprentissage.

Ce réseau est appelé probabiliste car il peut être vu comme l'implémentation de la méthode des fenêtres de Parzen, qui consiste à centrer une gaussienne sur chaque exemple d'apprentissage. Chaque classe est alors constituée par une somme de gaussiennes dont l'amplitude et l'étalement reste à déterminer.

Exemple

Considérons le tableau P à 2 lignes et 7 colonnes suivant (il est en effet transposé par le ') :

```
P=[0 0;1 1;0 3;1 4;3 1;4 1;4 3]'
```

puis un vecteur comportant les indices des classes désirées (=cibles) :

```
Tc=[1 1 2 2 3 3 3] ;
```

La fonction `newpnn` a besoin d'une matrice avec des 1 aux bons endroits. Cela est réalisé avec la fonction `ind2vec` :

```
T=ind2vec(Tc)
```

qui donne :

```
T=
```

```
(1,1)    1
(1,2)    1
(2,3)    1
```

etc.

qui signifie qu'il y a dans la matrice T des 1 aux positions (1,1), (1,2), (2,3)...

Création du réseau et simulation :

```
net=newpnn(P,T) ;
```

```
Y=sim(net,P) ;
```

```
Yc=vec2ind(Y)
```

affiche :

```
Yc=  1    1    2    2    3    3    3
```

Les 6 vecteurs d'apprentissage sont donc correctement classés. L'étape suivante serait de tester le réseau avec des vecteurs non-appris.

Démo

```
demopnn1
```

Remarque : Il n'y a pas de phase d'apprentissage comme dans la plupart des autres réseaux de neurones. Ici le calcul des paramètres du réseau est effectué au moment de la création du réseau.

f) Réseaux à fonctions radiales de base (FRB)

Il existe 2 types de réseaux, créés par `newrbe` et `newrb`.

- `newrbe`

Cette fonction crée un réseau à fonctions radiales de base possédant autant de neurones dans la première couche (neurones de codage) que d'exemples d'entrée.

Exemple d'utilisation :

```
net=newrbe(P, T, spread);
```

P : exemples d'apprentissage, sous forme de colonnes dans une matrice

T : sorties désirées sous forme de vecteur dont la dimension est le nombre de classes

spread : paramètre d'étendue des gaussiennes

Le principal inconvénient de cette méthode est qu'elle nécessite un neurone par vecteur d'apprentissage, ce qui peut devenir coûteux en mémoire et en temps de traitement pour les grosses bases d'apprentissage.

- **newrb**

Crée un réseau à construction progressive. Les neurones sont créés un par un, en fonction d'une erreur de sortie : à chaque itération le réseau choisit l'exemple d'entrée qui minimise cette erreur. Ce vecteur est alors copié dans les poids d'un neurone à FRB, et devient alors un représentant d'une classe. Cette opération est répétée jusqu'à ce que l'erreur de sortie passe en dessous du seuil demandé.

Exemple d'utilisation :

```
net=newrbe(P, T, goal, spread);
```

Les paramètres **P**, **T** et **spread** sont les mêmes que pour **newrbe**.

Le paramètre **goal** est un seuil indiqué par l'utilisateur pour l'erreur de sortie, en deça duquel la construction du réseau s'arrête.

g) Réseaux récurrents

Réseau de Hopfield

Fonction de création et de détermination des poids (apprentissage) : **newhop**

Entrées : cibles (points d'équilibre, entrées à mémoriser, à apprendre) ; matrice **T** de ces vecteurs

Sorties : les poids et les seuils

Utilisation : **net=newhop(T)** ;