eBook Gratuit

APPRENEZ

MATLAB Language

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#matlab

Table des matières

À propos 1
Chapitre 1: Démarrer avec MATLAB Language 2
Versions 2
Voir aussi: Historique des versions de MATLAB sur Wikipedia 3
Examples 4
Bonjour le monde 4
Matrices et tableaux 4
Matrices d'indexation et tableaux 6
Indexation des indices 6
Indexation linéaire 7
Indexation logique 9
Plus sur l'indexation 10
Vous aider 12
Lecture d'entrée et écriture 13
Réseaux de cellules 14
Scripts et fonctions 16
Types de données 17
Fonctions anonymes et poignées de fonction 20
Les bases 20
L'opérateur @ 21
Fonctions anonymes personnalisées 21
Fonctions anonymes d'une variable 21
Fonctions anonymes de plus d'une variable 22
Paramétrage des fonctions anonymes 22
Les arguments d'entrée d'une fonction anonyme ne font pas référence à des variables d'espa 23
Les fonctions anonymes sont stockées dans des variables 23
Utilisation avancée 23
Passer des poignées de fonction à d'autres fonctions 24
Utilisation de fonctions bsxfun , cellfun et similaires avec des fonctions anonymes 24
Chapitre 2: Applications financières 26

Examples....26

Random Walk....26

Mouvement brownien géométrique univarié....26

Chapitre 3: Astuces utiles....29

Examples....29

Fonctions utiles sur les cellules et les matrices....29

Préférences de pliage du code....32

Extraire des données de figure....34

Programmation fonctionnelle à l'aide de fonctions anonymes....36

Enregistrer plusieurs figures dans le même fichier .fig....36

Blocs de commentaires....37

Chapitre 4: Conditions....39

Syntaxe....39

Paramètres....39

Examples....39

SI condition....39

Condition IF-ELSE....40

Condition IF-ELSEIF....40

Conditions imbriquées....42

Chapitre 5: Contrôle de la coloration des sous-parcelles dans Matlab....44

Introduction....44

Remarques....44

Examples....44

Comment c'est fait....44

Chapitre 6: Décompositions matricielles....46

Syntaxe....46

Examples....46

Décomposition de Cholesky....46

Décomposition QR....46

Décomposition LU....47

Décomposition de schur....48

Décomposition en valeur singulière....49

Chapitre 7: Définir les opérations 50
Syntaxe 50
Paramètres 50
Examples 50
Opérations élémentaires 50
Chapitre 8: Dessin 52
Examples 52
Des cercles 52
Flèches 54
Ellipse 57
Polygone (s) 58
Polygone unique 58
Plusieurs polygones 59
Tracé de pseudo 4D 59
Dessin rapide 64
Chapitre 9: Erreurs communes et erreurs 66
Examples 66
Ne nommez pas de variable avec un nom de fonction existant 66
Ce que vous voyez n'est pas ce que vous obtenez: char vs cellstring dans la fenêtre de com 66
Les opérateurs de transposition 67
Fonction ou méthode non définie pour les arguments d'entrée de type Y 68
Cette fonction a été introduite après votre version actuelle de MATLAB 68
Vous n'avez pas cette boîte à outils! 69
MATLAB ne peut pas localiser la fonction 70
Soyez conscient de l'imprécision en virgule flottante 70
Exemples: comparaison de points flottants effectuée WRONG: 70
Exemple: comparaison de virgule flottante effectuée à DROITE: 71
Lectures complémentaires: 71
Pas assez d'arguments en entrée 71
Méthode n ° 1 - via l'invite de commande 72
Méthode n ° 2 - Interactif via l'éditeur 72

Attention aux changements de taille de tableau 73
Erreurs d'incompatibilité de dimension 74
L'utilisation de "i" ou "j" comme unité imaginaire, index de boucle ou variable commune. 74
Recommandation 74
Défaut 74
Les utiliser comme une variable (pour les indices de boucle ou autre variable) 75
En les utilisant comme unité imaginaire: 76
Pièges 76
Utiliser `length` pour les tableaux multidimensionnels 77
Chapitre 10: Fonctionnalités non documentées 79
Remarques 79
Examples 79
Fonctions d'aide compatibles C ++ 79
Tracés de lignes 2D à code couleur avec données de couleur en troisième dimension 79
Marqueurs semi-transparents dans les lignes et les nuages de points 80
Tracés de contour - Personnaliser les étiquettes de texte 82
Ajout / ajout d'entrées à une légende existante 84
Scatter plot jitter 85
Chapitre 11: Fonctions de documentation 87
Remarques 87
Examples 87
Documentation de fonction simple 87
Documentation de fonction locale 87
Obtenir une signature de fonction 88
Documenter une fonction avec un exemple de script 88
Chapitre 12: Graphiques: Tracés de lignes 2D 92
Syntaxe 92
Paramètres 92
Remarques 92
Examples 92
Plusieurs lignes dans une seule parcelle 92
Split line avec NaN 93

Ordre personnalisé de couleurs et de lignes 94
Chapitre 13: Graphiques: Transformations 2D et 3D 98
Examples 98
Transformations 2D 98
Chapitre 14: Initialisation de matrices ou de tableaux 101
Introduction 101
Syntaxe 101
Paramètres 101
Remarques 101
Examples 101
Créer une matrice de 0 101
Créer une matrice de 1 102
Créer une matrice d'identité 102
Chapitre 15: Interfaces utilisateur MATLAB 103
Examples 103
Passage de données autour de l'interface utilisateur 103
guidata 103
setappdata / getappdata 103
UserData 104
Fonctions imbriquées 105
Arguments de saisie explicites 105
Faire un bouton dans votre interface utilisateur qui interrompt l'exécution du rappel 106
Passer des données en utilisant la structure "handles" 107
Problèmes de performances lors du transfert de données autour de l'interface utilisateur 108
Chapitre 16: Interpolation avec MATLAB 111
Syntaxe 111
Examples 111
Interpolation par morceaux en 2 dimensions 111
Interpolation par morceaux 1 dimension 114
Interpolation polynomiale 120
Chapitre 17: Introduction à l'API MEX 124

Examples 124
Vérifier le nombre d'entrées / sorties dans un fichier MEX C ++ 124
testinputs.cpp 124
Entrez une chaîne, modifiez-la en C et affichez-la 125
stringIO.cpp 125
Passer une matrice 3D de MATLAB à C 126
matrixIn.cpp 126
Passer une structure par noms de champs 128
structIn.c 128
Chapitre 18: L'intégration 130
Examples 130
Intégrale, intégrale2, intégrale3 130
Chapitre 19: Le débogage 132
Syntaxe 132
Paramètres 132
Examples 132
Travailler avec des points d'arrêt 132
Définition 132
Points d'arrêt dans MATLAB 132
Motivation 133
Types de points d'arrêt 133
Placer les points d'arrêt 133
Désactivation et réactivation des points d'arrêt 134
Suppression des points d'arrêt 134
Reprise de l'exécution 134
Débogage du code Java appelé par MATLAB 135
Vue d'ensemble 135
MATLAB fin 135
Les fenêtres: 135
Fin du débogueur 136
IntelliJ IDEA 136

Chapitre 20: Lecture de gros fichiers 139

Examples 139

textescan 139

Chaînes de date et d'heure vers un tableau numérique rapide 139

Chapitre 21: Les fonctions 141

Examples 141

Exemple de base 141

Plusieurs sorties 141

nargin, nargout 142

Chapitre 22: Meilleures pratiques MATLAB 144

Remarques 144

Examples 144

Gardez les lignes courtes 144

Indentez le code correctement 144

Utiliser assert 145

Éviter les boucles 146

Exemples 146

Créer un nom unique pour un fichier temporaire 146

Utiliser les attributs valides 148

Opérateur de commentaire de bloc 153

Chapitre 23: Multithreading 155

Examples 155

Utiliser parfor pour paralléliser une boucle 155

Quand utiliser parfor 155

Exécuter des commandes en parallèle à l'aide d'une instruction "Single Program, Multiple D 156

Utilisation de la commande batch pour effectuer divers calculs en parallèle 157

Chapitre 24: Performance et Benchmarking 158

Remarques 158

Examples 158

Identification des goulots d'étranglement des performances à l'aide du profileur 158

Comparer le temps d'exécution de plusieurs fonctions 161

C'est bien d'être célibataire! 162

Aperçu: 162
Conversion de variables dans un script en une précision / type / classe autre que celle pa 163
Mises en garde et pièges: 163
Voir également: 164
réorganiser un tableau ND peut améliorer les performances globales 164
L'importance de la préallocation 166
Chapitre 25: Pour les boucles 169
Remarques 169
Itérer sur le vecteur de colonne 169
Modifier la variable d'itération 169
Performance du cas particulier d' a:b dans le côté droit 169
Examples 170
Boucle 1 à n 170
Itérer sur des éléments de vecteur 170
Itérer sur des colonnes de matrice 172
Boucle sur les index 172
Boucles imbriquées 172
Remarque: Bizarrement, les mêmes boucles imbriquées. 174
Chapitre 26: Programmation orientée objet 175
Examples 175
Définir une classe 175
Exemple de classe: 175
Classes valeur vs poignée 176
Héritage des classes et des classes abstraites 177
Constructeurs 181
Chapitre 27: Solveurs des équations différentielles ordinaires (ODE) 184
Examples 184
Exemple pour odeset 184
Chapitre 28: Traitement d'image 186
Examples 186
Image de base I / O 186

Récupérer des images sur Internet....186
Filtrage à l'aide d'une FFT 2D....186
Filtrage d'image....187
Propriétés de mesure des régions connectées....189
Chapitre 29: Transformées de Fourier et Transformées de Fourier Inverse....192
Syntaxe....192
Paramètres....192
Remarques....193
Examples....193
Implémenter une simple transformation de Fourier dans Matlab....193
Transformées de Fourier Inverse....194
Images et FT multidimensionnelles....196
Zéro remplissage....196
Trucs, astuces, 3D et au-delà....200
Chapitre 30: Utilisation de fonctions avec sortie logique....202
Examples....202
Tous et tout avec des tableaux vides....202
Chapitre 31: Utilisation de la fonction `accumarray ()`....203
Introduction....203
Syntaxe....203
Paramètres....203
Remarques....203
Références :....204
Examples....204
Recherche de la valeur maximale parmi les éléments regroupés par un autre vecteur....204
Appliquer un filtre aux patchs d'image et définir chaque pixel comme la moyenne du résulta....205
Chapitre 32: Utiliser des ports série....206
Introduction....206
Paramètres....206
Examples....206
Créer un port série sur Mac / Linux / Windows....207
Lecture depuis le port série....207

Fermer un port série même perdu, supprimé ou écrasé 207
Écriture sur le port série 207
Choisir votre mode de communication 208
Mode 1: synchrone (maître / esclave) 208
Mode 2: Asynchrone 209
Mode 3: Streaming (temps réel) 210
Traitement automatique des données reçues d'un port série 211
Chapitre 33: Utilitaires de programmation 213
Examples 213
Minuterie simple dans MATLAB 213
Chapitre 34: Vectorisation 214
Examples 214
Opérations élémentaires 214
Somme, méchant, prod & co 214
Utilisation de bsxfun 215
Syntaxe 216
Remarques 217
Masquage logique 217
Extension implicite de la matrice (diffusion) [R2016b] 218
Exemples de tailles compatibles: 219
Exemples de tailles incompatibles: 219
Obtenir la valeur d'une fonction de deux arguments ou plus 220
Crédits 222

À propos

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: matlab-language

It is an unofficial and free MATLAB Language ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official MATLAB Language.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapitre 1: Démarrer avec MATLAB Language

Versions

Version	Libération	Date de sortie
1.0		1984-01-01
2		1986-01-01
3		1987-01-01
3.5		1990-01-01
4		1992-01-01
4.2c		1994-01-01
5.0	Volume 8	1996-12-01
5.1	Volume 9	1997-05-01
5.1.1	R9.1	1997-05-02
5.2	R10	1998-03-01
5.2.1	R10.1	1998-03-02
5.3	R11	1999-01-01
5.3.1	R11.1	1999-11-01
6,0	R12	2000-11-01
6.1	R12.1	2001-06-01
6,5	R13	2002-06-01
6.5.1	R13SP2	2003-01-01
6.5.2	R13SP2	2003-01-02
7	R14	2006-06-01
7.0.4	R14SP1	2004-10-01
7.1	R14SP3	2005-08-01

Version	Libération	Date de sortie
7.2	R2006a	2006-03-01
7.3	R2006b	2006-09-01
7.4	R2007a	2007-03-01
7.5	R2007b	2007-09-01
7.6	R2008a	2008-03-01
7.7	R2008b	2008-09-01
7.8	R2009a	2009-03-01
7,9	R2009b	2009-09-01
7.10	R2010a	2010-03-01
7.11	R2010b	2010-09-01
7.12	R2011a	2011-03-01
7.13	R2011b	2011-09-01
7.14	R2012a	2012-03-01
8.0	R2012b	2012-09-01
8.1	R2013a	2013-03-01
8.2	R2013b	2013-09-01
8.3	R2014a	2014-03-01
8.4	R2014b	2014-09-01
8.5	R2015a	2015-03-01
8.6	R2015b	2015-09-01
9.0	R2016a	2016-03-01
9.1	R2016b	2016-09-14
9.2	R2017a	2017-03-08

Voir aussi: Historique des versions de MATLAB sur Wikipedia .

Examples

Bonjour le monde

Ouvrez un nouveau document vierge dans l'éditeur MATLAB (dans les versions récentes de MATLAB, faites-le en sélectionnant l'onglet Accueil de la barre d'outils et en cliquant sur Nouveau script). Le raccourci clavier par défaut pour créer un nouveau script est `Ctrl-n` .

Alternativement, en tapant `edit myscriptname.m` ouvrira le fichier `myscriptname.m` pour le modifier, ou proposera de créer le fichier s'il n'existe pas sur le chemin MATLAB.

Dans l'éditeur, tapez ce qui suit:

```
disp('Hello, World!');
```

Sélectionnez l'onglet Editeur de la barre d'outils, puis cliquez sur Enregistrer sous. Enregistrez le document dans un fichier du répertoire en cours appelé `helloworld.m` . L'enregistrement d'un fichier sans titre fera apparaître une boîte de dialogue pour nommer le fichier.

Dans la fenêtre de commande MATLAB, tapez ce qui suit:

```
>> helloworld
```

Vous devriez voir la réponse suivante dans la fenêtre de commande MATLAB:

```
Hello, World!
```

Nous voyons que dans la fenêtre de commande, nous pouvons taper les noms des fonctions ou des fichiers de script que nous avons écrits, ou qui sont livrés avec MATLAB, pour les exécuter.

Ici, nous avons lancé le script "helloworld". Notez que taper l'extension (`.m`) est inutile. Les instructions contenues dans le fichier de script sont exécutées par MATLAB, imprimant ici "Hello, World!" en utilisant la fonction `disp` .

Les fichiers de script peuvent être écrits de cette manière pour enregistrer une série de commandes pour une réutilisation ultérieure.

Matrices et tableaux

Dans MATLAB, le type de données le plus élémentaire est le tableau numérique. Il peut s'agir d'un scalaire, d'un vecteur 1-D, d'une matrice 2D ou d'un tableau multidimensionnel ND.

```
% a 1-by-1 scalar value
x = 1;
```

Pour créer un vecteur de ligne, entrez les éléments entre crochets, séparés par des espaces ou des virgules:

```
% a 1-by-4 row vector
v = [1, 2, 3, 4];
v = [1 2 3 4];
```

Pour créer un vecteur de colonne, séparez les éléments par des points-virgules:

```
% a 4-by-1 column vector
v = [1; 2; 3; 4];
```

Pour créer une matrice, nous entrons les lignes comme précédemment séparées par des points-virgules:

```
% a 2 row-by-4 column matrix
M = [1 2 3 4; 5 6 7 8];

% a 4 row-by-2 column matrix
M = [1 2; ...
     4 5; ...
     6 7; ...
     8 9];
```

Notez que vous ne pouvez pas créer une matrice avec une taille de ligne / colonne inégale. Toutes les lignes doivent avoir la même longueur et toutes les colonnes doivent avoir la même longueur:

```
% an unequal row / column matrix
M = [1 2 3 ; 4 5 6 7]; % This is not valid and will return an error

% another unequal row / column matrix
M = [1 2 3; ...
     4   5; ...
     6 7 8; ...
     9   10];     % This is not valid and will return an error
```

Pour transposer un vecteur ou une matrice, nous utilisons le `.'` -operator, ou le `'` opérateur de prendre son conjugué hermitien, qui est le conjugué complexe de sa transposée. Pour les vraies matrices, ces deux sont les mêmes:

```
% create a row vector and transpose it into a column vector
v = [1 2 3 4].';              % v is equal to [1; 2; 3; 4];

% create a 2-by-4 matrix and transpose it to get a 4-by-2 matrix
M = [1 2 3 4; 5 6 7 8].';     % M is equal to [1 5; 2 6; 3 7; 4 8]

% transpose a vector or matrix stored as a variable
A = [1 2; 3 4];
B = A.';                      % B is equal to [1 3; 2 4]
```

Pour les tableaux de plus de deux dimensions, il n'y a pas de syntaxe de langage directe pour les entrer littéralement. Au lieu de cela, nous devons utiliser des fonctions pour les construire (telles que des `ones` , des `zeros` , des `rand`) ou en manipulant d'autres tableaux (en utilisant des fonctions telles que `cat` , `reshape` , `permute`). Quelques exemples:

```
% a 5-by-2-by-4-by-3 array (4-dimensions)
arr = ones(5, 2, 4, 3);

% a 2-by-3-by-2 array (3-dimensions)
arr = cat(3, [1 2 3; 4 5 6], [7 8 9; 0 1 2]);

% a 5-by-4-by-3-by-2 (4-dimensions)
arr = reshape(1:120, [5 4 3 2]);
```

Matrices d'indexation et tableaux

MATLAB permet plusieurs méthodes pour indexer (accéder) des éléments de matrices et de tableaux:

- **Indexation des indices** - où vous spécifiez séparément la position des éléments que vous souhaitez dans chaque dimension de la matrice.
- **Indexation linéaire** - où la matrice est traitée comme un vecteur, quelles que soient ses dimensions. Cela signifie que vous spécifiez chaque position dans la matrice avec un seul numéro.
- **Indexation logique** - où vous utilisez une matrice logique (et une matrice de valeurs `true` et `false`) avec les dimensions identiques de la matrice que vous essayez d'indexer en tant que masque pour spécifier la valeur à renvoyer.

Ces trois méthodes sont maintenant expliquées plus en détail à l' aide de la matrice 3 par 3 suivant `M` comme exemple:

```
>> M = magic(3)

ans =

       8     1     6
       3     5     7
       4     9     2
```

Indexation des indices

La méthode la plus simple pour accéder à un élément consiste à spécifier son index de colonne de ligne. Par exemple, accéder à l'élément sur la deuxième ligne et la troisième colonne:

```
>> M(2, 3)

ans =

      7
```

Le nombre d'indices fournis correspond exactement au nombre de dimensions `M` (deux dans cet exemple).

Notez que l'ordre des indices est identique à la convention mathématique: l'index de ligne est le premier. De plus, les indices MATLAB **commencent par** `1` et **non par** `0` comme la plupart des langages de programmation.

Vous pouvez indexer plusieurs éléments à la fois en passant un vecteur pour chaque coordonnée au lieu d'un seul nombre. Par exemple, pour obtenir la deuxième ligne entière, nous pouvons spécifier que nous voulons les première, deuxième et troisième colonnes:

```
>> M(2, [1,2,3])

ans =

       3     5     7
```

Dans MATLAB, le vecteur `[1,2,3]` est plus facilement créé en utilisant l'opérateur deux-points, c'est-à-dire `1:3` . Vous pouvez également l'utiliser pour l'indexation. Pour sélectionner toute une ligne (ou colonne), Matlab fournit un raccourci en vous permettant de spécifier simplement `:` . Par exemple, le code suivant renverra également la deuxième ligne entière

```
>> M(2, :)

ans =

       3     5     7
```

MATLAB fournit également un raccourci pour spécifier le dernier élément d'une dimension sous la forme du mot clé `end` . Le mot-clé `end` fonctionnera exactement comme s'il s'agissait du numéro du dernier élément de cette dimension. Donc, si vous voulez toutes les colonnes de la colonne `2` à la dernière colonne, vous pouvez utiliser ce qui suit:

```
>> M(2, 2:end)

ans =

       5     7
```

L'indexation des indices peut être restrictive car elle ne permet pas d'extraire des valeurs uniques de différentes colonnes et lignes; il va extraire la combinaison de toutes les lignes et colonnes.

```
>> M([2,3], [1,3])
ans =

       3     7
       4     2
```

Par exemple, l'indexation des indices ne peut extraire que les éléments `M(2,1)` ou `M(3,3)` . Pour ce faire, il faut envisager l'indexation linéaire.

Indexation linéaire

MATLAB vous permet de traiter les tableaux à n dimensions comme des tableaux à une dimension lorsque vous indexez en utilisant une seule dimension. Vous pouvez accéder directement au premier élément:

```
>> M(1)

ans =

       8
```

Notez que les tableaux sont stockés en ordre majeur dans MATLAB, ce qui signifie que vous accédez aux éléments en commençant par les colonnes. Donc `M(2)` est le deuxième élément de la première colonne qui est `3` et `M(4)` sera le premier élément de la deuxième colonne, c'est-à-dire

```
>> M(4)

ans =

        1
```

Il existe des fonctions intégrées dans MATLAB pour convertir les indices en indice en indices linéaires, et inversement: `sub2ind` et `ind2sub` respectivement. Vous pouvez convertir manuellement les indices (`r` , `c`) en un index linéaire en

```
idx = r + (c-1)*size(M,1)
```

Pour comprendre cela, si nous sommes dans la première colonne, l'index linéaire sera simplement l'indice de ligne. La formule ci-dessus est vraie pour cela car pour `c == 1` , `(c-1) == 0` . Dans les colonnes suivantes, l'index linéaire est le numéro de ligne plus toutes les lignes des colonnes précédentes.

Notez que le mot-clé `end` s'applique toujours et fait maintenant référence au tout dernier élément du tableau, à savoir `M(end) == M(end, end) == 2` .

Vous pouvez également indexer plusieurs éléments à l'aide de l'indexation linéaire. Notez que si vous faites cela, la matrice retournée aura la même forme que la matrice des vecteurs d'index.

`M(2:4)` renvoie un vecteur de ligne car `2:4` représente le vecteur de ligne `[2,3,4]` :

```
>> M(2:4)

ans =

        3     4     1
```

Comme autre exemple, `M([1,2;3,4])` renvoie une matrice 2 par 2 car `[1,2;3,4]` est également une matrice 2 par 2. Voir le code ci-dessous pour vous convaincre:

```
>> M([1,2;3,4])

ans =

       8         3
       4         1
```

Notez que l'indexation avec `:` alone renverra *toujours* un vecteur de colonne:

```
>> M(:)

ans =

        8
        3
        4
        1
        5
        9
        6
        7
        2
```

Cet exemple illustre également l'ordre dans lequel MATLAB renvoie des éléments lors de l'utilisation de l'indexation linéaire.

Indexation logique

La troisième méthode d'indexation consiste à utiliser une matrice logique, c'est-à-dire une matrice contenant uniquement `false` valeurs `true` ou `false` , comme masque pour filtrer les éléments que vous ne voulez pas. Par exemple, si nous voulons trouver tous les éléments de `M` supérieurs à `5` nous pouvons utiliser la matrice logique

```
>> M > 5

ans =

    1     0     1
    0     0     1
    0     1     0
```

pour indexer `M` et renvoyer uniquement les valeurs supérieures à `5` comme suit:

```
>> M(M > 5)

ans =

        8
        9
        6
        7
```

Si vous vouliez que ces numéros restent en place (c.-à-d. Gardez la forme de la matrice), vous pouvez attribuer au complément logique

```
>> M(~(M > 5)) = NaN

ans =

    8      NaN    6
    NaN    NaN    7
```

```
    NaN    9      Nan
```

Nous pouvons réduire les blocs de code complexes contenant `for` instructions `if` et `for` en utilisant l'indexation logique.

Prenez le non vectorisé (déjà raccourci en une seule boucle en utilisant l'indexation linéaire):

```
for elem = 1:numel(M)
  if M(elem) > 5
    M(elem) = M(elem) - 2;
  end
end
```

Cela peut être raccourci au code suivant à l'aide de l'indexation logique:

```
idx = M > 5;
M(idx) = M(idx) - 2;
```

Ou même plus court:

```
M(M > 5) = M(M > 5) - 2;
```

Plus sur l'indexation

Matrices de dimension supérieure

Toutes les méthodes mentionnées ci-dessus se généralisent en n-dimensions. Si nous utilisons la matrice tridimensionnelle `M3 = rand(3,3,3)` comme exemple, vous pouvez accéder à toutes les lignes et colonnes de la deuxième tranche de la troisième dimension en écrivant

```
>> M(:,:,2)
```

Vous pouvez accéder au premier élément de la deuxième tranche en utilisant l'indexation linéaire. L'indexation linéaire ne se déplacera que sur la deuxième tranche après toutes les lignes et toutes les colonnes de la première tranche. Donc, l'index linéaire pour cet élément est

```
>> M(size(M,1)*size(M,2)+1)
```

En fait, dans MATLAB, *chaque* matrice est une dimension n: il se trouve que la taille de la plupart des autres dimensions n est une. Donc, si `a = 2` alors `a(1) == 2` (comme on peut s'attendre), *mais aussi* `a(1, 1) == 2` , comme le fait `a(1, 1, 1) == 2` , `a(1, 1, 1, ..., 1) == 2` et ainsi de suite. Ces dimensions "supplémentaires" (de taille `1`) sont appelées *cotes singleton* . La commande `squeeze` les supprimera, et on pourra utiliser `permute` pour échanger l'ordre des dimensions (et introduire des dimensions singleton si nécessaire).

Une matrice à n dimensions peut également être indexée à l'aide d'un m indices (où m <= n). La règle est que les premiers indices m-1 se comportent normalement, tandis que le dernier indice (m'th) référence les dimensions restantes (n-m + 1), tout comme un index linéaire ferait référence

à une dimension (n-m + 1). tableau. Voici un exemple:

```
>> M = reshape(1:24,[2,3,4]);
>> M(1,1)
ans =
     1
>> M(1,10)
ans =
    19
>> M(:,:)
ans =
     1     3     5     7     9    11    13    15    17    19    21    23
     2     4     6     8    10    12    14    16    18    20    22    24
```

Retour des plages d'éléments

Avec l'indexation des indices, si vous spécifiez plusieurs éléments dans plusieurs dimensions, MATLAB renvoie chaque paire de coordonnées possible. Par exemple, si vous essayez M ([1,2], [1,3]), MATLAB renverra `M(1,1)` et `M(2,3)` mais retournera **également** `M(1,3)` et `M(2,1)` . Cela peut sembler peu intuitif lorsque vous recherchez les éléments d'une liste de paires de coordonnées, mais considérez l'exemple d'une matrice plus grande, `A = rand(20)` (note `A` maintenant `20` - par `20`), où vous voulez obtenir la quadrant supérieur droit. Dans ce cas, au lieu d'avoir à spécifier chaque paire de coordonnées dans ce quadrant (et ce serait `100` paires), il vous suffit de spécifier les `10` lignes et les `10` colonnes que vous souhaitez. `A(1:10, 11:end)` . *Trancher* une matrice comme celle-ci est beaucoup plus courant que de demander une liste de paires de coordonnées.

Si vous souhaitez obtenir une liste de paires de coordonnées, la solution la plus simple consiste à convertir en indexation linéaire. Considérez le problème où vous avez un vecteur d'index de colonne que vous souhaitez retourner, où chaque ligne du vecteur contient le numéro de colonne que vous souhaitez renvoyer pour la ligne *correspondante* de la matrice. Par exemple

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

Donc, dans ce cas, vous voulez récupérer les éléments en `(1,3)` , `(2,2)` et `(3,1)` . Donc, en utilisant l'indexation linéaire:

```
>> colIdx = [3;2;1];
>> rowIdx = 1:length(colIdx);
>> idx = sub2ind(size(M), rowIdx, colIdx);
>> M(idx)

ans =

        6     5     4
```

Renvoyer un élément plusieurs fois

Avec l’indexation linéaire et en indice, vous pouvez également renvoyer plusieurs fois un élément en répétant son index

```
>> M([1,1,1,2,2,2])
```

```
ans =

     8     8     8     3     3     3
```

Vous pouvez l'utiliser pour dupliquer des lignes et des colonnes entières, par exemple, pour répéter la première ligne et la dernière colonne.

```
>> M([1, 1:end], [1:end, end])

ans =

     8     1     6     6
     8     1     6     6
     3     5     7     7
     4     9     2     2
```

Pour plus d'informations, voir ici .

Vous aider

MATLAB est livré avec de nombreux scripts et fonctions intégrés qui vont de la simple multiplication aux boîtes à outils de reconnaissance d'images. Pour obtenir des informations sur une fonction que vous souhaitez utiliser, tapez: `help functionname` dans la ligne de commande. Prenons la fonction d' `help` comme exemple.

L'information sur la façon de l'utiliser peut être obtenue en tapant:

```
>> help help
```

dans la fenêtre de commande. Cela renverra des informations sur l'utilisation de l' `help` de la fonction. Si les informations que vous recherchez ne sont toujours pas claires, vous pouvez essayer la page de **documentation** de la fonction. Tapez simplement:

```
>> doc help
```

dans la fenêtre de commande. Cela ouvrira la documentation navigable sur la page d' `help` aux fonctions fournissant toutes les informations dont vous avez besoin pour comprendre comment fonctionne l'aide.

Cette procédure fonctionne pour toutes les fonctions et tous les symboles intégrés.

Lorsque vous développez vos propres fonctions, vous pouvez leur laisser leur propre section d'aide en ajoutant des commentaires en haut du fichier de fonction ou juste après la déclaration de fonction.

Exemple pour une fonction simple `multiplyby2` enregistrée dans le fichier `multiplyby2.m`

```
function [prod]=multiplyby2(num)
% function MULTIPLYBY2 accepts a numeric matrix NUM and returns output PROD
% such that all numbers are multiplied by 2
```

```
    prod=num*2;
end
```

ou

```
% function MULTIPLYBY2 accepts a numeric matrix NUM and returns output PROD
% such that all numbers are multiplied by 2

function [prod]=multiplyby2(num)
    prod=num*2;
end
```

Ceci est très utile lorsque vous prenez votre code semaines / mois / années après l'avoir écrit.

La fonction d' `help` et de `doc` fournit de nombreuses informations, apprendre à utiliser ces fonctionnalités vous aidera à progresser rapidement et à utiliser efficacement MATLAB.

Lecture d'entrée et écriture

Comme tous les langages de programmation, Matlab est conçu pour lire et écrire dans une grande variété de formats. La bibliothèque native prend en charge un grand nombre de formats Texte, Image, Vidéo, Audio, Données avec plus de formats inclus dans chaque mise à jour de version. Cochez ici pour voir la liste complète des formats de fichiers pris en charge et leur fonction.

Avant d'essayer de charger votre fichier, vous devez vous demander ce que vous voulez que les données deviennent et comment vous attendez que l'ordinateur organise les données pour vous. Supposons que vous avez un fichier txt / csv au format suivant:

```
Fruit,TotalUnits,UnitsLeftAfterSale,SellingPricePerUnit
Apples,200,67,$0.14
Bananas,300,172,$0.11
Pineapple,50,12,$1.74
```

Nous pouvons voir que la première colonne est dans le format de chaînes, tandis que la deuxième, troisième, est numérique, la dernière colonne est sous la forme de devise. Disons que nous voulons trouver combien de revenus nous avons fait aujourd'hui en utilisant Matlab et que nous voulons d'abord charger dans ce fichier txt / csv. Après avoir vérifié le lien, nous pouvons voir que les types de fichiers txt String et Numeric sont gérés par `textscan` . Nous pourrions donc essayer:

```
fileID = fopen('dir/test.txt'); %Load file from dir
C = textscan(fileID,'%s %f %f %s','Delimiter',',','HeaderLines',1); %Parse in the txt/csv
```

où `%s` suggère que l'élément est un type String, `%f` suggère que l'élément est un type Float et que le fichier est délimité par ",". L'option HeaderLines demande à Matlab d'ignorer les N premières lignes tandis que le 1 immédiatement après signifie de sauter la première ligne (la ligne d'en-tête).

Maintenant, C est la donnée que nous avons chargée et qui se présente sous la forme d'un

tableau de 4 cellules, chacune contenant la colonne de données du fichier txt / csv.

Donc, nous voulons d'abord calculer combien de fruits nous avons vendus aujourd'hui en soustrayant la troisième colonne de la deuxième colonne, cela peut être fait par:

```
sold = C{2} - C{3}; %C{2} gives the elements inside the second cell (or the second column)
```

Maintenant, nous voulons multiplier ce vecteur par le prix par unité, donc nous devons d'abord convertir cette colonne de chaînes en une colonne de nombres, puis la convertir en une matrice numérique à l'aide de `cell2mat` de Matlab. du signe "$", il y a plusieurs façons de le faire. La manière la plus directe consiste à utiliser une simple expression régulière:

```
D = cellfun(@(x)(str2num(regexprep(x, '\$',''))), C{4}, 'UniformOutput', false);%cellfun
allows us to avoid looping through each element in the cell.
```

Ou vous pouvez utiliser une boucle:

```
for t=1:size(C{4},1)
   D{t} = str2num(regexprep(C{4}{t}, '\$',''));
end

E = cell2mat(D)% converts the cell array into a Matrix
```

La fonction `str2num` transforme la chaîne qui a les signes "$" en types numériques et `cell2mat` transforme la cellule des éléments numériques en une matrice de nombres

Maintenant, nous pouvons multiplier les unités vendues par le coût par unité:

```
revenue = sold .* E; %element-wise product is denoted by .* in Matlab

totalrevenue = sum(revenue);
```

Réseaux de cellules

Les éléments de la même classe peuvent souvent être concaténés en tableaux (à quelques rares exceptions près, par exemple les descripteurs de fonctions). Les scalaires numériques, par défaut de classe `double` , peuvent être stockés dans une matrice.

```
>> A = [1, -2, 3.14, 4/5, 5^6; pi, inf, 7/0, nan, log(0)]
A =
   1.0e+04 *
    0.0001   -0.0002    0.0003    0.0001    1.5625
    0.0003       Inf       Inf       NaN      -Inf
```

Les caractères, qui sont de classe `char` dans MATLAB, peuvent également être stockés dans un tableau en utilisant une syntaxe similaire. Un tel tableau est similaire à une chaîne dans de nombreux autres langages de programmation.

```
>> s = ['MATLAB ','is ','fun']
```

```
s =
MATLAB is fun
```

Notez que bien que les deux utilisent des crochets `[` et `]` , les classes de résultats sont différentes. Par conséquent, les opérations pouvant être effectuées sur eux sont également différentes.

```
>> whos
  Name      Size            Bytes  Class     Attributes

  A         2x5                80  double
  s         1x13               26  char
```

En fait, le tableau `s` n'est pas un tableau des chaînes `'MATLAB '` , `'is '` et `'fun'` , il ne s'agit que d'une chaîne - un tableau de 13 caractères. Vous obtiendriez les mêmes résultats s'il était défini par l'un des éléments suivants:

```
>> s = ['MAT','LAB ','is f','u','n'];
>> s = ['M','A','T','L','A','B,' ','i','s',' ','f','u','n'];
```

Un vecteur MATLAB standard ne vous permet pas de stocker un mélange de variables de différentes classes ou de plusieurs chaînes différentes. C'est là que le tableau de `cell` est utile. Il s'agit d'un tableau de cellules pouvant contenir chacun un objet MATLAB, dont la classe peut être différente dans chaque cellule si nécessaire. Utilisez des accolades `{` et `}` autour des éléments à stocker dans un tableau de cellules.

```
>> C = {A; s}
C =
    [2x5 double]
    'MATLAB is fun'
>> whos C
  Name      Size            Bytes  Class    Attributes

  C         2x1               330  cell
```

Les objets MATLAB standard de toutes les classes peuvent être stockés ensemble dans un tableau de cellules. Notez que les tableaux de cellules nécessitent plus de mémoire pour stocker leur contenu.

L'accès au contenu d'une cellule se fait à l'aide d'accolades `{` et `}` .

```
>> C{1}
ans =
   1.0e+04 *
    0.0001   -0.0002    0.0003    0.0001    1.5625
    0.0003       Inf       Inf       NaN      -Inf
```

Notez que `C(1)` est différent de `C{1}` . Alors que le dernier retourne le contenu de la cellule (et a par exemple la classe `double` in out), le premier retourne un tableau de cellules qui est un sous-tableau de `C` De même, si `D` était un tableau de 10 par 5 cellules, alors `D(4:8,1:3)` renverrait un sous-tableau de `D` dont la taille est de 5 par 3 et dont la classe est la `cell` . Et la syntaxe `C{1:2}` n'a pas un seul objet retourné, mais il renvoie 2 objets différents (similaire à une fonction MATLAB avec

plusieurs valeurs de retour):

```
>> [x,y] = C{1:2}
x =
                        1                        -2                        3.14
0.8                   15625
          3.14159265358979                      Inf                        Inf
NaN                    -Inf
y =
MATLAB is fun
```

Scripts et fonctions

Le code MATLAB peut être enregistré dans les fichiers m pour être réutilisé. Les fichiers m ont l'extension `.m` qui est automatiquement associée à MATLAB. Un fichier m peut contenir un script ou des fonctions.

Scripts

Les scripts sont simplement des fichiers de programme qui exécutent une série de commandes MATLAB dans un ordre prédéfini.

Les scripts n'acceptent pas les entrées et les scripts ne renvoient pas de sortie. Fonctionnellement, les scripts équivalent à taper des commandes directement dans la fenêtre de commande MATLAB et à pouvoir les rejouer.

Un exemple de script:

```
length = 10;
width = 3;
area = length * width;
```

Ce script définira la `length` , la `width` et la `area` dans l'espace de travail actuel avec la valeur `10` , `3` et `30` respectivement.

Comme indiqué précédemment, le script ci-dessus est fonctionnellement équivalent à taper les mêmes commandes directement dans la fenêtre de commande.

```
>> length = 10;
>> width = 3;
>> area = length * width;
```

Les fonctions

Les fonctions, par rapport aux scripts, sont beaucoup plus flexibles et extensibles. Contrairement aux scripts, les fonctions peuvent accepter les entrées et les renvoyer à l'appelant. Une fonction a son propre espace de travail, cela signifie que les opérations internes des fonctions ne modifieront pas les variables de l'appelant.

Toutes les fonctions sont définies avec le même format d’en-tête:

```
function [output] = myFunctionName(input)
```

Le mot-clé `function` commence chaque en-tête de fonction. La liste des sorties suit. La liste des sorties peut également être une liste de variables séparées par des virgules à renvoyer.

```
function [a, b, c] = myFunctionName(input)
```

Suivant est le nom de la fonction qui sera utilisée pour l'appel. C'est généralement le même nom que le nom de fichier. Par exemple, nous enregistrerions cette fonction sous le nom `myFunctionName.m` .

Le nom de la fonction suit la liste des entrées. Comme les sorties, il peut également s'agir d'une liste séparée par des virgules.

```
function [a, b, c] = myFunctionName(x, y, z)
```

Nous pouvons réécrire l'exemple de script précédent en tant que fonction réutilisable comme suit:

```
function [area] = calcRecArea(length, width)
   area = length * width;
end
```

Nous pouvons appeler des fonctions à partir d'autres fonctions, ou même à partir de fichiers de script. Voici un exemple de notre fonction ci-dessus utilisée dans un fichier script.

```
l = 100;
w = 20;
a = calcRecArea(l, w);
```

Comme précédemment, nous créons `l` , `w` et `a` dans l'espace de travail avec les valeurs de `100` , `20` et `2000` respectivement.

Types de données

Il existe 16 types de données fondamentaux , ou classes, dans MATLAB. Chacune de ces classes est sous la forme d'une matrice ou d'un tableau. À l'exception des descripteurs de fonctions, cette matrice ou ce tableau a une taille minimale de 0 par 0 et peut devenir un tableau à n dimensions de n'importe quelle taille. Un handle de fonction est toujours scalaire (1 par 1).

Un moment important dans MATLAB est que vous n'avez pas besoin d'utiliser une déclaration de type ou des instructions de dimension par défaut. Lorsque vous définissez une nouvelle variable, MATLAB la crée automatiquement et alloue un espace mémoire approprié.

Exemple:

```
a = 123;
b = [1 2 3];
c = '123';
```

```
>> whos
  Name       Size             Bytes  Class     Attributes

  a          1x1                  8  double
  b          1x3                 24  double
  c          1x3                  6  char
```

Si la variable existe déjà, MATLAB remplace les données d'origine par une nouvelle et alloue un nouvel espace de stockage si nécessaire.

Types de données fondamentaux

Les types de données fondamentaux sont: numeric, `logical` , `char` , `cell` , `struct` , `table` et `function_handle` .

Types de données numériques :

- Nombre à virgule flottante (*par défaut*)

  MATLAB représente des nombres à virgule flottante au format double précision ou simple précision. La valeur par défaut est la double précision, mais vous pouvez créer n'importe quelle précision simple avec une fonction de conversion simple:

```
a = 1.23;
b = single(a);

>> whos
  Name       Size             Bytes  Class     Attributes

  a          1x1                  8  double
  b          1x1                  4  single
```

- Entiers

  MATLAB a quatre classes d'entiers signés et quatre non signés. Les types signés vous permettent de travailler avec des entiers négatifs aussi bien que positifs, mais ne peuvent pas représenter une plage de nombres aussi large que les types non signés, car un bit est utilisé pour désigner un signe positif ou négatif pour le nombre. Les types non signés vous donnent une plus grande gamme de nombres, mais ces nombres ne peuvent être que zéro ou positifs.

  MATLAB prend en charge le stockage à 1, 2, 4 et 8 octets pour les données entières. Vous pouvez économiser de la mémoire et du temps d'exécution pour vos programmes si vous utilisez le plus petit type d'entier adapté à vos données. Par exemple, vous n'avez pas besoin d'un entier 32 bits pour stocker la valeur 100.

```
a = int32(100);
b = int8(100);

>> whos
  Name       Size             Bytes  Class    Attributes
```

```
  a          1x1                   4  int32
  b          1x1                   1  int8
```

Pour stocker des données sous la forme d'un nombre entier, vous devez convertir du type double au type entier souhaité. Si le nombre converti en entier a une partie fractionnaire, MATLAB arrondit à l'entier le plus proche. Si la partie fractionnaire est exactement égale à `0.5` , MATLAB choisit, à partir des deux entiers également voisins, celui pour lequel la valeur absolue est la plus grande.

```
a  = int16(456);
```

- `char`

  Les tableaux de caractères fournissent un stockage pour les données de texte dans MATLAB. Conformément à la terminologie de programmation traditionnelle, un tableau (séquence) de caractères est défini comme une chaîne. Il n'y a pas de type de chaîne explicite dans les versions commerciales de MATLAB.

- logique: les valeurs logiques de 1 ou 0 représentent respectivement true et false. Utilisez pour les conditions relationnelles et l'indexation de tableau. Parce qu'il est juste VRAI ou FAUX, il a une taille de 1 octet.

```
a = logical(1);
```

- structure. Un tableau de structure est un type de données qui regroupe des variables de différents types de données à l'aide de conteneurs de données appelés *champs* . Chaque champ peut contenir n'importe quel type de données. Accéder aux données dans une structure en utilisant la notation par points de la forme structName.fieldName.

```
field1 = 'first';
field2 = 'second';
value1 = [1 2 3 4 5];
value2 = 'sometext';
s = struct(field1,value1,field2,value2);
```

Pour accéder à value1, chacune des syntaxes suivantes est équivalente

```
s.first or s.(field1) or s.('first')
```

Nous pouvons explicitement accéder à un champ dont nous savons qu’il existera avec la première méthode, ou bien passer une chaîne ou créer une chaîne pour accéder au champ dans le deuxième exemple. Le troisième exemple montre que la notation par points est une chaîne, qui est la même que celle stockée dans la variable field1.

- Les variables de table peuvent être de tailles et de types de données différents, mais toutes les variables doivent avoir le même nombre de lignes.

```
Age = [15 25 54]';
```

```
Height = [176 190 165]';
Name = {'Mike', 'Pete', 'Steeve'}';
T = table(Name,Age, Height);
```

- cellule. C'est un type de données MATLAB très utile: le tableau de cellules est un tableau dont chaque élément peut avoir un type et une taille de données différents. C'est un instrument très puissant pour manipuler les données comme vous le souhaitez.

  ```
  a = { [1 2 3], 56, 'art'};
  ```

  ou

  ```
  a = cell(3);
  ```

- la fonction gère stocke un pointeur sur une fonction (par exemple, une fonction anonyme). Cela vous permet de passer une fonction à une autre fonction ou d'appeler des fonctions locales en dehors de la fonction principale.

Il y a beaucoup d'instruments à `str2double` avec chaque type de données et également des fonctions de conversion de type de données intégrées (`str2double` , `table2cell`).

Types de données supplémentaires

Plusieurs types de données supplémentaires sont utiles dans certains cas spécifiques. Elles sont:

- Date et heure: tableaux représentant les dates, l'heure et la durée. `datetime('now')` retourne `21-Jul-2016 16:30:16` .
- Tableaux catégoriels: il s'agit d'un type de données permettant de stocker des données avec des valeurs provenant d'un ensemble de catégories discrètes. Utile pour stocker des données non numériques (mémoire efficace). Peut être utilisé dans un tableau pour sélectionner des groupes de lignes.

  ```
  a = categorical({'a' 'b' 'c'});
  ```

- Les conteneurs de cartes sont une structure de données qui a la capacité unique d'indexer non seulement par le biais de toutes les valeurs numériques scalaires, mais également par le biais du vecteur de caractères. Les indices dans les éléments d'une carte sont appelés clés. Ces clés, ainsi que les valeurs de données qui leur sont associées, sont stockées dans la carte.
- Les séries chronologiques sont des vecteurs de données échantillonnés dans le temps, dans l'ordre, souvent à intervalles réguliers. Il est utile de stocker les données connectées avec des horodatages et de nombreuses méthodes utiles à utiliser.

Fonctions anonymes et poignées de fonction

Les bases

Les fonctions anonymes sont un outil puissant du langage MATLAB. Ce sont des fonctions qui existent localement, à savoir: dans l'espace de travail actuel. Cependant, ils n'existent pas sur le chemin MATLAB comme le ferait une fonction normale, par exemple dans un fichier m. C'est pourquoi ils sont appelés anonymes, bien qu'ils puissent avoir un nom comme une variable dans l'espace de travail.

L'opérateur `@`

Utilisez l'opérateur `@` pour créer des fonctions anonymes et des descripteurs de fonctions. Par exemple, pour créer un handle vers la fonction `sin` (sinus) et l'utiliser comme `f` :

```
>> f = @sin
f =
    @sin
```

Maintenant, `f` est un handle vers la fonction `sin` . Tout comme (dans la vraie vie) une poignée de porte est un moyen d'utiliser une porte, une poignée de fonction est un moyen d'utiliser une fonction. Pour utiliser `f` , les arguments lui sont transmis comme s'il s'agissait de la fonction `sin` :

```
>> f(pi/2)
ans =
     1
```

`f` accepte tous les arguments en entrée que la fonction `sin` accepte. Si `sin` était une fonction qui accepte des arguments d'entrée nuls (ce qui n'est pas le cas, mais que d'autres le font, par exemple la fonction `peaks`), `f()` serait utilisé pour l'appeler sans argument d'entrée.

Fonctions anonymes personnalisées

Fonctions anonymes d'une variable

Il n'est évidemment pas utile de créer un handle pour une fonction existante, comme `sin` dans l'exemple ci-dessus. C'est un peu redondant dans cet exemple. Cependant, il est utile de créer des fonctions anonymes qui effectuent des tâches personnalisées qui, autrement, devraient être répétées plusieurs fois ou créer une fonction distincte pour. Comme exemple d'une fonction anonyme personnalisée qui accepte une variable en entrée, additionnez le carré sinus et le cosinus d'un signal:

```
>> f = @(x) sin(x)+cos(x).^2
f =
    @(x)sin(x)+cos(x).^2
```

Maintenant, `f` accepte un argument appelé `x` . Cela a été spécifié en utilisant des parenthèses `(...)` directement après l'opérateur `@` . `f` maintenant est une fonction anonyme de `x` : `f(x)` . Il est

utilisé en passant une valeur de `x` à `f` :

```
>> f(pi)
ans =
    1.0000
```

Un vecteur de valeurs ou une variable peut également être transmis à `f` , du moment qu'ils sont utilisés de manière valide dans `f` :

```
>> f(1:3) % pass a vector to f
ans =
    1.1334    1.0825    1.1212
>> n = 5:7;
>> f(n) % pass n to f
ans =
   -0.8785    0.6425    1.2254
```

Fonctions anonymes de plus d'une variable

De la même manière, des fonctions anonymes peuvent être créées pour accepter plusieurs variables. Un exemple de fonction anonyme qui accepte trois variables:

```
>> f = @(x,y,z) x.^2 + y.^2 - z.^2
f =
    @(x,y,z)x.^2+y.^2-z.^2
>> f(2,3,4)
ans =
    -3
```

Paramétrage des fonctions anonymes

Les variables de l'espace de travail peuvent être utilisées dans la définition des fonctions anonymes. Ceci s'appelle le paramétrage. Par exemple, pour utiliser une constante `c = 2` dans une fonction anonyme:

```
>> c = 2;
>> f = @(x) c*x
f =
    @(x)c*x
>> f(3)
ans =
     6
```

`f(3)` utilisé la variable `c` comme paramètre pour multiplier avec le `x` fourni. Notez que si la valeur de `c` est définie sur quelque chose de différent à ce stade, alors `f(3)` est appelé, le résultat **ne** serait **pas** différent. La valeur de `c` est la valeur *au moment de la création* de la fonction anonyme:

```
>> c = 2;
>> f = @(x) c*x;
>> f(3)
ans =
     6
```

```
>> c = 3;
>> f(3)
ans =
     6
```

Les arguments d'entrée d'une fonction anonyme ne font pas référence à des variables d'espace de travail

Notez que l'utilisation du nom des variables dans l'espace de travail comme l'un des arguments d'entrée d'une fonction anonyme (c'est-à-dire, l'utilisation de `@(...)`) n'utilisera **pas** les valeurs de ces variables. Au lieu de cela, ils sont traités comme des variables différentes dans la portée de la fonction anonyme, à savoir: la fonction anonyme a son espace de travail privé où les variables d'entrée ne font jamais référence aux variables de l'espace de travail principal. L'espace de travail principal et l'espace de travail de la fonction anonyme ne connaissent pas le contenu de l'autre. Un exemple pour illustrer ceci:

```
>> x = 3 % x in main workspace
x =
     3
>> f = @(x) x+1; % here x refers to a private x variable
>> f(5)
ans =
     6
>> x
x =
     3
```

La valeur de `x` dans l'espace de travail principal n'est pas utilisée dans `f` . En outre, dans l'espace de travail principal, `x` n'a pas été modifié. Dans le cadre de `f` , les noms de variable entre parenthèses après l'opérateur `@` sont indépendants des variables principales de l'espace de travail.

Les fonctions anonymes sont stockées dans des variables

Une fonction anonyme (ou, plus précisément, le handle de fonction pointant sur une fonction anonyme) est stockée comme toute autre valeur dans l'espace de travail actuel: Dans une variable (comme nous l'avons fait ci-dessus), dans un tableau de cellules (`{@(x)x.^2,@(x)x+1}`), ou même dans une propriété (comme `h.ButtonDownFcn` pour les graphiques interactifs). Cela signifie que la fonction anonyme peut être traitée comme toute autre valeur. Lorsque vous le stockez dans une variable, il porte un nom dans l’espace de travail actuel et peut être modifié et effacé comme les variables contenant des numéros.

En d'autres termes: Un handle de fonction (que ce soit dans la forme `@sin` ou pour une fonction anonyme) est simplement une valeur qui peut être stockée dans une variable, comme une matrice numérique peut l'être.

Utilisation avancée

Passer des poignées de fonction à d'autres fonctions

Les descripteurs de fonctions étant traités comme des variables, ils peuvent être transmis à des fonctions qui acceptent les descripteurs de fonctions en tant qu'arguments d'entrée.

Un exemple: Une fonction est créée dans un fichier m qui accepte un handle de fonction et un nombre scalaire. Il appelle ensuite le handle de fonction en lui passant `3` , puis ajoute le numéro scalaire au résultat. Le résultat est renvoyé.

Contenu de `funHandleDemo.m` :

```
function y = funHandleDemo(fun,x)
y = fun(3);
y = y + x;
```

Enregistrez-le quelque part sur le chemin, par exemple dans le dossier actuel de MATLAB. Maintenant, `funHandleDemo` peut être utilisé comme suit, par exemple:

```
>> f = @(x) x^2; % an anonymous function
>> y = funHandleDemo(f,10) % pass f and a scalar to funHandleDemo
y =
    19
```

Le handle d'une autre fonction existante peut être transmis à `funHandleDemo` :

```
>> y = funHandleDemo(@sin,-5)
y =
   -4.8589
```

Notez que `@sin` était un moyen rapide d'accéder à la fonction `sin` sans la stocker d'abord dans une variable en utilisant `f = @sin` .

Utilisation de fonctions `bsxfun` , `cellfun` et similaires avec des fonctions anonymes

MATLAB a des fonctions intégrées qui acceptent les fonctions anonymes en entrée. C'est un moyen d'effectuer de nombreux calculs avec un nombre minimal de lignes de code. Par exemple, `bsxfun` , qui effectue des opérations binaires élément par élément, c'est-à-dire qu'il applique une fonction sur deux vecteurs ou matrices d'une manière élément par élément. Normalement, cela nécessiterait l'utilisation de `for` -loops, qui nécessite souvent une préallocation pour la vitesse. En utilisant `bsxfun` ce processus est accéléré. L'exemple suivant illustre ceci en utilisant `tic` et `toc` , deux fonctions qui peuvent être utilisées pour chronométrer la durée du code. Il calcule la différence de chaque élément de matrice par rapport à la moyenne de la colonne de matrice.

```
A = rand(50); % 50-by-50 matrix of random values between 0 and 1

% method 1: slow and lots of lines of code
tic
```

```
meanA = mean(A); % mean of every matrix column: a row vector
% pre-allocate result for speed, remove this for even worse performance
result = zeros(size(A));
for j = 1:size(A,1)
    result(j,:) = A(j,:) - meanA;
end
toc
clear result % make sure method 2 creates its own result

% method 2: fast and only one line of code
tic
result = bsxfun(@minus,A,mean(A));
toc
```

L'exécution de l'exemple ci-dessus se traduit par deux sorties:

```
Elapsed time is 0.015153 seconds.
Elapsed time is 0.007884 seconds.
```

Ces lignes proviennent des fonctions `toc` , qui impriment le temps écoulé depuis le dernier appel à la fonction `tic` .

L'appel `bsxfun` applique la fonction dans le premier argument d'entrée aux deux autres arguments d'entrée. `@minus` est un nom long pour la même opération que ferait le signe moins. Une fonction ou un descripteur anonyme (`@`) différent de toute autre fonction aurait pu être spécifié, à condition qu'il accepte `A` et `mean(A)` comme entrées pour générer un résultat significatif.

Surtout pour les grandes quantités de données dans les grandes matrices, `bsxfun` peut accélérer les choses. Cela rend aussi le code plus propre, bien qu'il puisse être plus difficile à interpréter pour les personnes qui ne connaissent pas MATLAB ou `bsxfun` . (Notez que dans MATLAB R2016a et versions ultérieures, de nombreuses opérations qui utilisaient auparavant `bsxfun` n'en ont plus besoin; `A-mean(A)` fonctionne directement et peut dans certains cas être encore plus rapide.)

Lire Démarrer avec MATLAB Language en ligne:
https://riptutorial.com/fr/matlab/topic/235/demarrer-avec-matlab-language

Chapitre 2: Applications financières

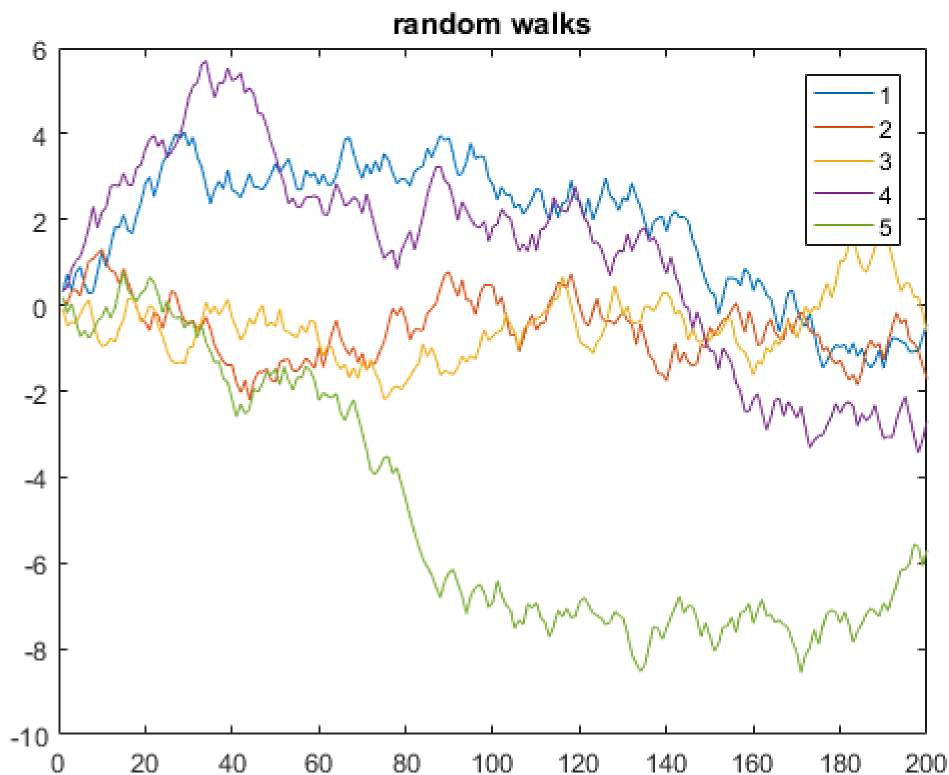
Examples

Random Walk

Voici un exemple qui affiche 5 marches aléatoires unidimensionnelles de 200 étapes:

```
y = cumsum(rand(200,5) - 0.5);

plot(y)
legend('1', '2', '3', '4', '5')
title('random walks')
```

Dans le code ci-dessus, `y` est une matrice de 5 colonnes, chacune de longueur 200. Étant donné que `x` est omis, la valeur par défaut est le nombre de lignes de `y` (équivalent à l'utilisation de `x=1:200` comme axe des x). De cette manière, la fonction de `plot` trace plusieurs vecteurs y sur le même vecteur x, chacun utilisant automatiquement une couleur différente.



Mouvement brownien géométrique univarié

La dynamique du mouvement brownien géométrique (GBM) est décrite par l'équation différentielle stochastique suivante:

$$dS_t = \mu S_t \, dt + \sigma S_t \, dW_t$$

Je peux utiliser la solution **exacte** pour le SDE

$$S_t = S_0 \exp\left(\left(\mu - \frac{\sigma^2}{2}\right) t + \sigma W_t\right)$$

générer des chemins qui suivent un GBM.

Compte tenu des paramètres quotidiens pour une simulation d'un an

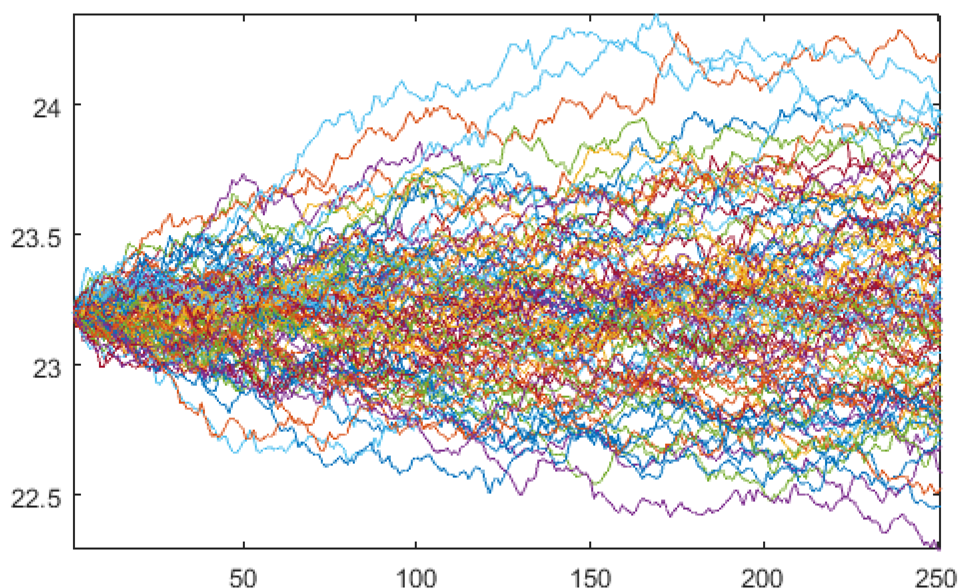
```
mu     = 0.08/250;
sigma  = 0.25/sqrt(250);
dt     = 1/250;
npaths = 100;
nsteps = 250;
S0     = 23.2;
```

on peut obtenir le mouvement brownien (BM) `W` partir de 0 et l'utiliser pour obtenir le GBM à partir de `S0`

```
% BM
epsilon = randn(nsteps, npaths);
W       = [zeros(1,npaths); sqrt(dt)*cumsum(epsilon)];

% GBM
t = (0:nsteps)'*dt;
Y = bsxfun(@plus, (mu-0.5*sigma.^2)*t, sigma*W);
Y = S0*exp(Y);
```

Qui produit les chemins

```
plot(Y)
```



Lire Applications financières en ligne: https://riptutorial.com/fr/matlab/topic/1197/applications-

financieres

Chapitre 3: Astuces utiles

Examples

Fonctions utiles sur les cellules et les matrices

Cet exemple simple fournit une explication sur certaines fonctions que j'ai trouvées extrêmement utiles depuis que j'ai commencé à utiliser MATLAB: `cellfun` , `arrayfun` . L'idée est de prendre un tableau ou une variable de classe de cellules, de parcourir tous ses éléments et d'appliquer une fonction dédiée sur chaque élément. Une fonction appliquée peut être anonyme, ce qui est généralement le cas, ou toute fonction régulière définie dans un fichier * .m.

Commençons par un problème simple et disons que nous devons trouver une liste de fichiers * .mat en fonction du dossier. Pour cet exemple, commençons par créer des fichiers * .mat dans un dossier en cours:

```
for n=1:10; save(sprintf('mymatfile%d.mat',n)); end
```

Après l'exécution du code, il devrait y avoir 10 nouveaux fichiers avec l'extension * .mat. Si nous exécutons une commande pour répertorier tous les fichiers * .mat, tels que:

```
mydir = dir('*.mat');
```

nous devrions avoir un tableau d'éléments d'une structure dir; MATLAB devrait donner un résultat similaire à celui-ci:

```
10x1 struct array with fields:
    name
    date
    bytes
    isdir
    datenum
```

Comme vous pouvez le voir, chaque élément de ce tableau est une structure avec deux champs. Toutes les informations sont en effet importantes pour chaque fichier mais dans 99%, je suis plutôt intéressé par les noms de fichiers et rien d'autre. Pour extraire des informations d'un tableau de structure, j'ai créé une fonction locale qui impliquait la création de variables temporelles de taille correcte, de boucles, d'extraction d'un nom de chaque élément et de sauvegarde dans la variable créée. Un moyen beaucoup plus facile d'obtenir exactement le même résultat est d'utiliser l'une des fonctions susmentionnées:

```
mydirlist = arrayfun(@(x) x.name, dir('*.mat'), 'UniformOutput', false)
mydirlist =
    'mymatfile1.mat'
    'mymatfile10.mat'
    'mymatfile2.mat'
    'mymatfile3.mat'
```

```
    'mymatfile4.mat'
    'mymatfile5.mat'
    'mymatfile6.mat'
    'mymatfile7.mat'
    'mymatfile8.mat'
    'mymatfile9.mat'
```

Comment fonctionne cette fonction? Il faut généralement deux paramètres: un handle de fonction comme premier paramètre et un tableau. Une fonction fonctionnera alors sur chaque élément d'un tableau donné. Les troisième et quatrième paramètres sont facultatifs mais importants. Si nous savons qu'une sortie ne sera pas régulière, elle doit être enregistrée dans la cellule. Ce doit être un paramètre de mise en évidence `false` à `UniformOutput` . Par défaut, cette fonction tente de renvoyer une sortie régulière telle qu'un vecteur de nombres. Par exemple, extrayons des informations sur la quantité d'espace disque prise par chaque fichier en octets:

```
mydirbytes = arrayfun(@(x) x.bytes, dir('*.mat'))
mydirbytes =
       34560
       34560
       34560
       34560
       34560
       34560
       34560
       34560
       34560
       34560
```

ou kilo-octets:

```
mydirbytes = arrayfun(@(x) x.bytes/1024, dir('*.mat'))
mydirbytes =
   33.7500
   33.7500
   33.7500
   33.7500
   33.7500
   33.7500
   33.7500
   33.7500
   33.7500
   33.7500
```

Cette fois, la sortie est un vecteur régulier de double. `UniformOutput` été défini sur `true` par défaut.

`cellfun` est une fonction similaire. La différence entre cette fonction et `arrayfun` est que `cellfun` opère sur des variables de classe de cellules. Si nous souhaitons extraire uniquement les noms à partir d’une liste de noms de fichiers dans une cellule "mydirlist", il suffit de lancer cette fonction comme suit:

```
mydirnames = cellfun(@(x) x(1:end-4), mydirlist, 'UniformOutput', false)
mydirnames =
    'mymatfile1'
    'mymatfile10'
```

```
    'mymatfile2'
    'mymatfile3'
    'mymatfile4'
    'mymatfile5'
    'mymatfile6'
    'mymatfile7'
    'mymatfile8'
    'mymatfile9'
```

De même, comme une sortie n'est pas un vecteur de nombres régulier, une sortie doit être enregistrée dans une variable de cellule.

Dans l'exemple ci-dessous, je combine deux fonctions en une et renvoie uniquement une liste de noms de fichiers sans extension:

```
cellfun(@(x) x(1:end-4), arrayfun(@(x) x.name, dir('*.mat'), 'UniformOutput', false),
'UniformOutput', false)
ans =
    'mymatfile1'
    'mymatfile10'
    'mymatfile2'
    'mymatfile3'
    'mymatfile4'
    'mymatfile5'
    'mymatfile6'
    'mymatfile7'
    'mymatfile8'
    'mymatfile9'
```

C'est fou mais très possible parce que `arrayfun` retourne une cellule à laquelle on attend l'entrée de `cellfun` ; Une autre remarque est que nous pouvons forcer l'une de ces fonctions à retourner des résultats dans une variable de cellule en définissant `UniformOutput` sur false. Nous pouvons toujours obtenir des résultats dans une cellule. Nous ne pourrons peut-être pas obtenir de résultats dans un vecteur régulier.

Il y a une autre fonction similaire qui opère sur les champs une structure: `structfun` . Je ne l'ai pas particulièrement trouvé aussi utile que les deux autres, mais cela brillerait dans certaines situations. Si par exemple on aimerait savoir quels champs sont numériques ou non numériques, le code suivant peut donner la réponse:

```
structfun(@(x) ischar(x), mydir(1))
```

Le premier et le second champ d'une structure de répertoire sont de type char. Par conséquent, la sortie est la suivante:

```
 1
 1
 0
 0
 0
```

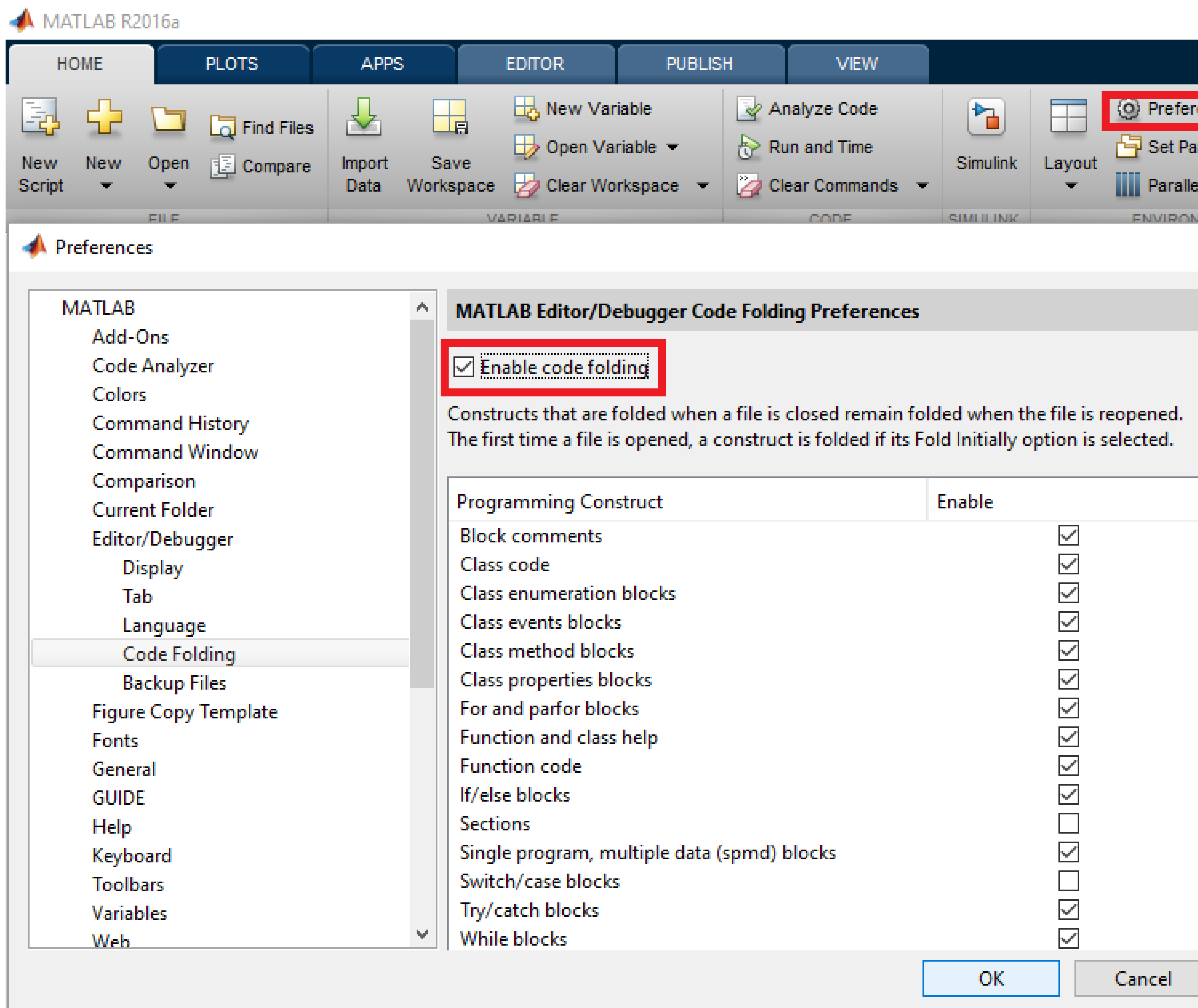
En outre, la sortie est un vecteur logique de `true` / `false` . Par conséquent, il est régulier et peut

être enregistré dans un vecteur; pas besoin d'utiliser une classe de cellules.

Préférences de pliage du code

Il est possible de modifier la préférence de pliage du code en fonction de vos besoins. Ainsi, le pliage de code peut être défini sur activer / impossible pour des constructions spécifiques (ex: `if block` , `for loop` , `Sections` ...).

Pour modifier les préférences de pliage, allez dans Préférences -> Pliage de code:



Ensuite, vous pouvez choisir quelle partie du code peut être pliée.

Des informations:

- Notez que vous pouvez également développer ou réduire tout le code d'un fichier en plaçant votre curseur n'importe où dans le fichier, en cliquant avec le bouton droit, puis en sélectionnant Pliage du code> Développer tout ou Plier le code> Plier tout dans le menu

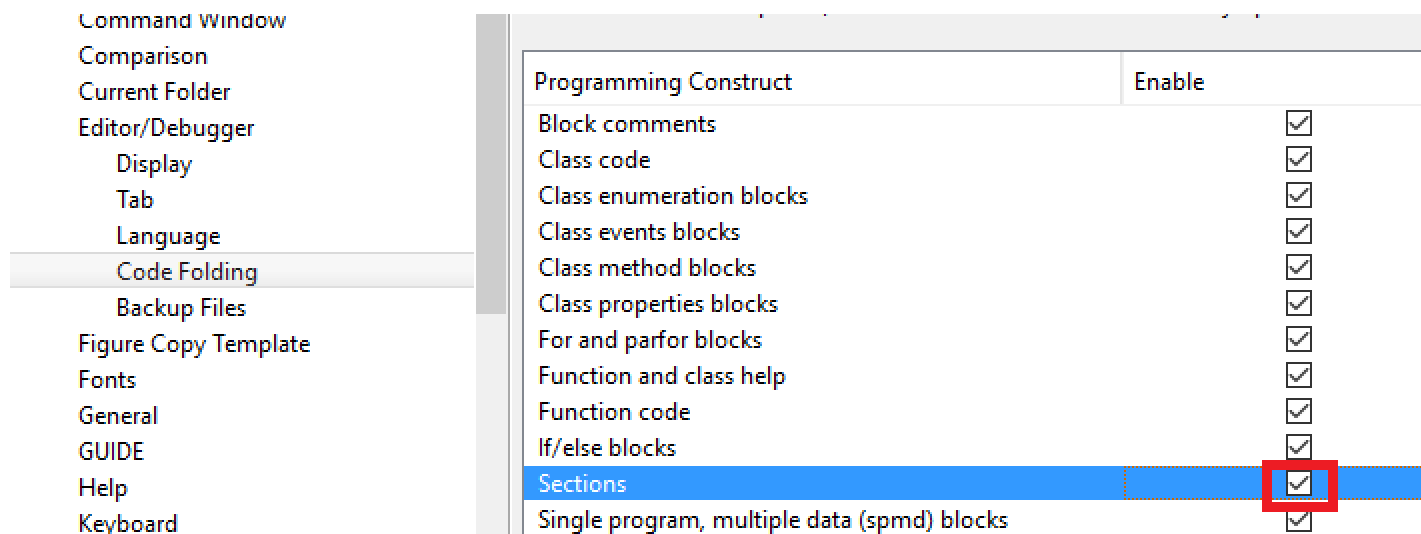
contextuel.

- Notez que le pliage est persistant, en ce sens qu'une partie du code qui a été développé / réduit conserve son statut après que Matlab ou que le fichier m ait été fermé et ré-ouvert.

Exemple: Pour activer le pliage des sections:

Une option intéressante est de permettre de plier des sections. Les sections sont délimitées par deux signes de pourcentage (`%%`).

Exemple: Pour l'activer, cochez la case "Sections":

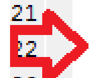


Ensuite, au lieu de voir un long code source similaire à:

```
3   %% LOAD
4
5           %code to load data
6           %           ...
7           %           ...
8           %           ...
9           %           ...
10          %           ...
11          %           ...
12
13  %% TREAT
14          % code to run the model
15          %           ...
16          %           ...
17          %           ...
18          %           ...
19          %           ...
20          %           ...
21
22  %% OUTPUT
23
24          % code to output results
25          %           ...
26          %           ...
27          %           ...
28          %           ...
29          %           ...
30          %           ...
31
```

Vous pourrez plier des sections pour avoir un aperçu général de votre code:

```
1
2
3   ⊞ %% LOAD %%...%%
12
13  ⊞ %% TREAT %%...%%
21
22  ⊟ %% OUTPUT
23
24          % code to output results
25          %           ...
26          %           ...
27          %           ...
28          %           ...
29          %           ...
30          %           ...
```



Extraire des données de figure

À quelques occasions, j'ai eu un personnage intéressant que j'ai enregistré mais j'ai perdu l'accès à ses données. Cet exemple montre comment obtenir des informations à partir d'une figure.

Les fonctions clés sont findobj et get . findobj renvoie un gestionnaire à un objet avec les attributs

ou propriétés de l'objet, tels que `Type` ou `Color` , etc. Une fois qu'un objet ligne a été trouvé, get peut renvoyer toute valeur contenue dans les propriétés. Il s'avère que les objets `Line` contiennent toutes les données dans les propriétés suivantes: `XData` , `YData` et `ZData` ; le dernier est généralement 0 sauf si un chiffre contient un tracé 3D.

Le code suivant crée un exemple qui montre deux lignes une fonction sin et un seuil et une légende

```
t = (0:1/10:1-1/10)';
y = sin(2*pi*t);
plot(t,y);
hold on;
plot([0 0.9],[0 0], 'k-');
hold off;
legend({'sin' 'threshold'});
```

La première utilisation de findobj renvoie deux gestionnaires aux deux lignes:

```
findobj(gcf, 'Type', 'Line')
ans =
  2x1 Line array:

  Line    (threshold)
  Line    (sin)
```

Pour limiter le résultat, findobj peut également utiliser une combinaison d’opérateurs logiques `-and` , `-or` ou de noms de propriétés. Par exemple, je peux trouver un objet ligne dont `DiplayName` est `sin` et lire ses `XData` et `YData` .

```
lineh = findobj(gcf, 'Type', 'Line', '-and', 'DisplayName', 'sin');
xdata = get(lineh, 'XData');
ydata = get(lineh, 'YData');
```

et vérifiez si les données sont égales.

```
isequal(t(:),xdata(:))
ans =
     1
isequal(y(:),ydata(:))
ans =
     1
```

De même, je peux affiner mes résultats en excluant la ligne noire (seuil):

```
lineh = findobj(gcf, 'Type', 'Line', '-not', 'Color', 'k');
xdata = get(lineh, 'XData');
ydata = get(lineh, 'YData');
```

et le dernier contrôle confirme que les données extraites de ce chiffre sont identiques:

```
isequal(t(:),xdata(:))
ans =
```

```
     1
isequal(y(:),ydata(:))
ans =
     1
```

Programmation fonctionnelle à l'aide de fonctions anonymes

Les fonctions anonymes peuvent être utilisées pour la programmation fonctionnelle. Le principal problème à résoudre est qu'il n'existe pas de méthode native pour ancrer une récursivité, mais cela peut toujours être implémenté sur une seule ligne:

```
if_ = @(bool, tf) tf{2-bool}();
```

Cette fonction accepte une valeur booléenne et un tableau de cellules de deux fonctions. La première de ces fonctions est évaluée si la valeur booléenne est vraie et la seconde si la valeur booléenne est fausse. Nous pouvons facilement écrire la fonction factorielle maintenant:

```
fac = @(n,f) if_(n>1, {@()n*f(n-1,f), @()1});
```

Le problème ici est que nous ne pouvons pas appeler directement un appel récursif, car la fonction n'est pas encore affectée à une variable lorsque le côté droit est évalué. Nous pouvons cependant compléter cette étape en écrivant

```
factorial_ = @(n)fac(n,fac);
```

Maintenant, `@(n)fac(n,fac)` évalue récursivement la fonction factorielle. Une autre façon de le faire dans la programmation fonctionnelle en utilisant un combinateur y, qui peut également être facilement implémenté:

```
y_ = @(f)@(n)f(n,f);
```

Avec cet outil, la fonction factorielle est encore plus courte:

```
factorial_ = y_(fac);
```

Ou directement:

```
factorial_ = y_(@(n,f) if_(n>1, {@()n*f(n-1,f), @()1}));
```

Enregistrer plusieurs figures dans le même fichier .fig

En plaçant plusieurs descripteurs de figures dans un tableau graphique, plusieurs figures peuvent être enregistrées dans le même fichier .fig

```
h(1) = figure;
scatter(rand(1,100),rand(1,100));
```

```
h(2) = figure;
scatter(rand(1,100),rand(1,100));

h(3) = figure;
scatter(rand(1,100),rand(1,100));

savefig(h,'ThreeRandomScatterplots.fig');
close(h);
```

Cela crée 3 diagrammes de dispersion de données aléatoires, chaque partie du tableau graphique h. Ensuite, le tableau graphique peut être enregistré en utilisant savefig comme avec un chiffre normal, mais avec le handle du tableau graphique comme argument supplémentaire.

Une remarque intéressante est que les chiffres ont tendance à rester disposés de la même manière qu'ils ont été sauvegardés lorsque vous les ouvrez.

Blocs de commentaires

Si vous souhaitez commenter une partie de votre code, des blocs de commentaires peuvent être utiles. Le bloc de commentaires commence par `%{` dans une nouvelle ligne et se termine par `%}` dans une autre nouvelle ligne:

```
a = 10;
b = 3;
%{
c = a*b;
d = a-b;
%}
```

Cela vous permet de plier les sections que vous avez commentées pour rendre le code plus propre et compact.

Ces blocs sont également utiles pour activer / désactiver des parties de votre code. Tout ce que vous avez à faire pour décommenter le bloc est d'ajouter un autre `%` avant qu'il ne se manifeste:

```
a = 10;
b = 3;
%%{ <-- another % over here
c = a*b;
d = a-b;
%}
```

Parfois, vous souhaitez commenter une section du code, mais sans affecter son retrait:

```
for k = 1:a
    b = b*k;
    c = c-b;
    d = d*c;
    disp(b)
end
```

Habituellement, lorsque vous marquez un bloc de code et appuyez sur Ctrl + r pour le commenter

(en ajoutant automatiquement le `%` à toutes les lignes, lorsque vous appuyez plus tard sur Ctrl + i pour obtenir une indentation automatique, le bloc de code place, et déplacé trop à droite:

```
for k = 1:a
    b = b*k;
    %     c = c-b;
    %     d = d*c;
    disp(b)
end
```

Une façon de résoudre ce problème consiste à utiliser des blocs de commentaires, afin que la partie interne du bloc reste correctement en retrait:

```
for k = 1:a
    b = b*k;
    %{
    c = c-b;
    d = d*c;
    %}
    disp(b)
end
```

Lire Astuces utiles en ligne: https://riptutorial.com/fr/matlab/topic/4179/astuces-utiles

Chapitre 4: Conditions

Syntaxe

- si *expression* ... fin
- si *expression* ... sinon ... fin
- if *expression* ... elseif *expression* ... end
- if *expression* ... elseif *expression* ... else ... end

Paramètres

Paramètre	La description
expression	une expression qui a un sens logique

Examples

SI condition

Les conditions sont une partie fondamentale de presque toutes les parties du code. Ils sont utilisés pour exécuter certaines parties du code uniquement dans certaines situations, mais pas dans d’autres. Regardons la syntaxe de base:

```
a = 5;
if a > 10    % this condition is not fulfilled, so nothing will happen
    disp('OK')
end

if a < 10    % this condition is fulfilled, so the statements between the if...end are
executed
    disp('Not OK')
end
```

Sortie:

```
Not OK
```

Dans cet exemple, nous voyons que le `if` est composé de 2 parties: la condition et le code à exécuter si la condition est vraie. Le code est tout écrit après la condition et avant la `end` de celle-ci `if` . La première condition n'a pas été remplie et le code à l'intérieur n'a pas été exécuté.

Voici un autre exemple:

```
a = 5;
if a ~= a+1        % "~=" means "not equal to"
    disp('It''s true!') % we use two apostrophes to tell MATLAB that the ' is part of the
```

```
string
end
```

La condition ci-dessus sera toujours vraie et affichera le résultat `It's true!` .

On peut aussi écrire:

```
a = 5;
if a == a+1    % "==" means "is equal to", it is NOT the assignment ("=") operator
    disp('Equal')
end
```

Cette fois-ci, la condition est toujours fausse, donc nous n'obtiendrons jamais la sortie `Equal` .

Les conditions qui sont toujours vraies ou fausses ne sont pas très utiles, car si elles sont toujours fausses, nous pouvons simplement supprimer cette partie du code, et si elles sont toujours vraies, la condition n'est pas nécessaire.

Condition IF-ELSE

Dans certains cas, nous voulons exécuter un code alternatif si la condition est fausse, pour cela nous utilisons la partie facultative `else` :

```
a = 20;
if a < 10
    disp('a smaller than 10')
else
    disp('a bigger than 10')
end
```

Ici, nous voyons cela parce `a` n'est pas plus petit que `10` la deuxième partie du code, après que l' `else` soit exécuté et que nous obtenons la sortie `a bigger than 10` . Maintenant, regardons un autre essai:

```
a = 10;
if a > 10
    disp('a bigger than 10')
else
    disp('a smaller than 10')
end
```

Dans cet exemple montre que nous n'avons pas vérifié si `a` est en effet inférieur à 10, et que nous obtenons un message erroné car la condition ne vérifie que l'expression telle quelle, et TOUS les cas qui ne sont pas égaux à true (`a = 10`) deuxième partie à exécuter.

Ce type d'erreur est un piège très courant pour les programmeurs débutants et expérimentés, en particulier lorsque les conditions deviennent complexes et doivent toujours être prises en compte

Condition IF-ELSEIF

En utilisant `else` nous pouvons effectuer certaines tâches lorsque la condition n'est pas satisfaite.

Mais que faire si nous voulons vérifier une seconde condition au cas où la première serait fausse. Nous pouvons le faire de cette façon:

```
a = 9;
if mod(a,2)==0   % MOD - modulo operation, return the remainder after division of 'a' by 2
    disp('a is even')
else
    if mod(a,3)==0
        disp('3 is a divisor of a')
    end
end

OUTPUT:
3 is a divisor of a
```

Ceci est aussi appelé "condition imbriquée" , mais ici nous avons un cas particulier qui peut améliorer la lisibilité du code et réduire les risques d'erreur - nous pouvons écrire:

```
a = 9;
if mod(a,2)==0
    disp('a is even')
elseif mod(a,3)==0
    disp('3 is a divisor of a')
end

OUTPUT:
3 is a divisor of a
```

en utilisant le `elseif` nous pouvons vérifier une autre expression dans le même bloc de condition, et cela ne se limite pas à un essai:

```
a = 25;
if mod(a,2)==0
    disp('a is even')
elseif mod(a,3)==0
    disp('3 is a divisor of a')
elseif mod(a,5)==0
    disp('5 is a divisor of a')
end

OUTPUT:
5 is a divisor of a
```

Des précautions supplémentaires doivent être prises lors du choix d'utiliser `elseif` dans une ligne, car **un seul** d'entre eux sera exécuté à partir de tout le bloc `if` to `end` . Donc, dans notre exemple, si nous voulons afficher tous les diviseurs d' `a` (à partir de ceux que nous vérifions explicitement), l'exemple ci-dessus ne sera pas bon:

```
a = 15;
if mod(a,2)==0
    disp('a is even')
elseif mod(a,3)==0
    disp('3 is a divisor of a')
elseif mod(a,5)==0
    disp('5 is a divisor of a')
```

```
end

OUTPUT:
3 is a divisor of a
```

non seulement 3, mais aussi 5 est un diviseur de 15, mais la partie qui vérifie la division par 5 n'est pas atteinte si l'une des expressions ci-dessus était vraie.

Enfin, nous pouvons ajouter un `else` (et **un seul)** après toutes les `elseif` conditions d'exécution d' un code lorsque aucune des conditions ci - dessus sont remplies:

```
a = 11;
if mod(a,2)==0
    disp('a is even')
elseif mod(a,3)==0
    disp('3 is a divisor of a')
elseif mod(a,5)==0
    disp('5 is a divisor of a')
else
    disp('2, 3 and 5 are not divisors of a')
end

OUTPUT:
2, 3 and 5 are not divisors of a
```

Conditions imbriquées

Lorsque nous utilisons une condition dans une autre condition, nous disons que les conditions sont "imbriquées". Un cas particulier de conditions imbriquées est donné par l'option `elseif` , mais il existe de nombreuses autres façons d'utiliser des conditions imbriquées. Examinons le code suivant:

```
a = 2;
if mod(a,2)==0    % MOD - modulo operation, return the remainder after division of 'a' by 2
    disp('a is even')
    if mod(a,3)==0
        disp('3 is a divisor of a')
        if mod(a,5)==0
            disp('5 is a divisor of a')
        end
    end
else
    disp('a is odd')
end
```

Pour `a=2` , le résultat sera `a is even` , ce qui est correct. Pour `a=3` , le résultat sera `a is odd` , ce qui est également correct, mais rate la vérification si 3 est un diviseur de `a` . C'est parce que les conditions sont imbriquées, donc **seulement** si la première est `true` , que nous passons à la valeur interne, et si `a` est impair, aucune des conditions internes n'est vérifiée. Ceci est quelque peu opposé à l'utilisation de `elseif` où seulement si la première condition est `false` que la suivante. Et si on vérifiait la division par 5? seul un nombre ayant 6 comme diviseur (à la fois 2 et 3) sera vérifié pour la division par 5, et nous pouvons tester et voir que pour `a=30` la sortie est:

```
a is even
3 is a divisor of a
5 is a divisor of a
```

Nous devrions également remarquer deux choses:

1. La position de la `end` au bon endroit pour chaque `if` est crucial pour l'ensemble des conditions de travail comme prévu, de sorte que l' indentation est plus qu'une bonne recommandation ici.
2. La position de l'instruction `else` est également cruciale, car nous devons savoir `if` (et il pourrait y en avoir plusieurs) nous voulons faire quelque chose si cette expression est `false` .

Regardons un autre exemple:

```
for a = 5:10    % the FOR loop execute all the code within it for every a from 5 to 10
    ch = num2str(a);    % NUM2STR converts the integer a to a character
    if mod(a,2)==0
        if mod(a,3)==0
            disp(['3 is a divisor of ' ch])
        elseif  mod(a,4)==0
            disp(['4 is a divisor of ' ch])
        else
            disp([ch ' is even'])
        end
    elseif mod(a,3)==0
        disp(['3 is a divisor of ' ch])

    else
        disp([ch ' is odd'])
    end
end
```

Et le résultat sera:

```
5 is odd
3 is a divisor of 6
7 is odd
4 is a divisor of 8
3 is a divisor of 9
10 is even
```

nous voyons que nous n’avons que 6 lignes pour 6 chiffres, car les conditions sont imbriquées de manière à ne garantir qu’une impression par numéro. un nombre n'est même pas il n'y a aucun point à vérifier si 4 est l'un de ses diviseurs.

Lire Conditions en ligne: https://riptutorial.com/fr/matlab/topic/3806/conditions

Chapitre 5: Contrôle de la coloration des sous-parcelles dans Matlab

Introduction

Comme je me débattais plus d'une fois, et que le Web n'était pas très clair sur ce qu'il fallait faire, j'ai décidé de prendre ce qui se passait, d'ajouter un peu de mien pour expliquer comment créer des sous-intrigues comportant sont mis à l'échelle en fonction de cela.

Je l'ai testé avec la dernière version de Matlab, mais je suis sûr que cela fonctionnera dans les anciennes versions.

Remarques

La seule chose que vous devez faire vous-même est le positionnement de la barre de couleur (si vous voulez l'afficher). Cela dépendra du nombre de graphiques que vous avez et de l'orientation de la barre.

La position et la taille sont définies à l'aide de 4 paramètres - x_start, y_start, x_width, y_width. Le tracé est généralement mis à l'échelle en unités normalisées, de sorte que le coin inférieur gauche correspond à (0,0) et le coin supérieur droit à (1,1).

Examples

Comment c'est fait

Il s'agit d'un code simple qui crée 6 sous-tracés 3d et, à la fin, synchronise la couleur affichée dans chacun d'eux.

```
c_fin = [0,0];
[X,Y] = meshgrid(1:0.1:10,1:0.1:10);

figure; hold on;
for i = 1 : 6
    Z(:,:,i) = i * (sin(X) + cos(Y));

    ax(i) = subplot(3,2,i); hold on; grid on;
    surf(X, Y, Z(:,:,i));
    view(-26,30);
    colormap('jet');
    ca = caxis;
    c_fin = [min(c_fin(1),ca(1)), max(c_fin(2),ca(2))];
end

%%you can stop here to see how it looks before we color-manipulate

c = colorbar('eastoutside');
c.Label.String = 'Units';
```

```
set(c, 'Position', [0.9, 0.11, 0.03, 0.815]); %%you may want to play with these values
pause(2); %%need this to allow the last image to resize itself before changing its axes
for i = 1 : 6
    pos=get(ax(i), 'Position');
    axes(ax(i));
    set(ax(i), 'Position', [pos(1) pos(2) 0.85*pos(3) pos(4)]);
    set(ax(i),'Clim', c_fin); %%this is where the magic happens
end
```

Lire Contrôle de la coloration des sous-parcelles dans Matlab en ligne: https://riptutorial.com/fr/matlab/topic/10913/controle-de-la-coloration-des-sous-parcelles-dans-matlab

Chapitre 6: Décompositions matricielles

Syntaxe

1. R = chol (A);
2. [L, U] = lu (A);
3. R = qr (A);
4. T = schur (A);
5. [U, S, V] = svd (A);

Examples

Décomposition de Cholesky

La décomposition de Cholesky est une méthode pour décomposer une matrice hermitienne définie positive en une matrice triangulaire supérieure et sa transposition. Il peut être utilisé pour résoudre des systèmes d'équations linéaires et est environ deux fois plus rapide que la décomposition LU.

```
A = [4 12 -16
    12 37 -43
    -16 -43 98];
R = chol(A);
```

Cela renvoie la matrice triangulaire supérieure. Le plus bas est obtenu par transposition.

```
L = R';
```

Nous pouvons enfin vérifier si la décomposition était correcte.

```
b = (A == L*R);
```

Décomposition QR

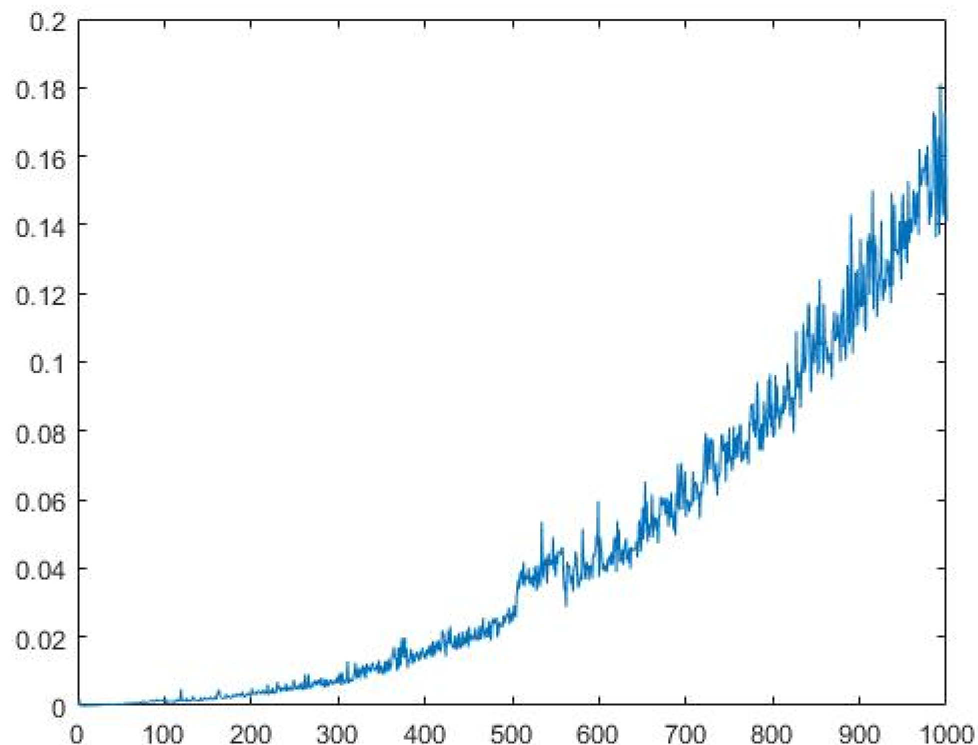
Cette méthode décomposera une matrice en une matrice triangulaire supérieure et une matrice orthogonale.

```
A = [4 12 -16
    12 37 -43
    -16 -43 98];
R = qr(A);
```

Cela retournera la matrice triangulaire supérieure tandis que les suivantes renverront les deux matrices.

```
[Q,R] = qr(A);
```

Le tracé suivant affichera le temps d'exécution de `qr` dépendant de la racine carrée des éléments



de la matrice.

Décomposition LU

Ici, une matrice sera décomposée en une matrice triangulaire supérieure et une matrice triangulaire inférieure. Souvent, il sera utilisé pour augmenter la performance et la stabilité (si cela est fait avec la permutation) de l'élimination de Gauß.

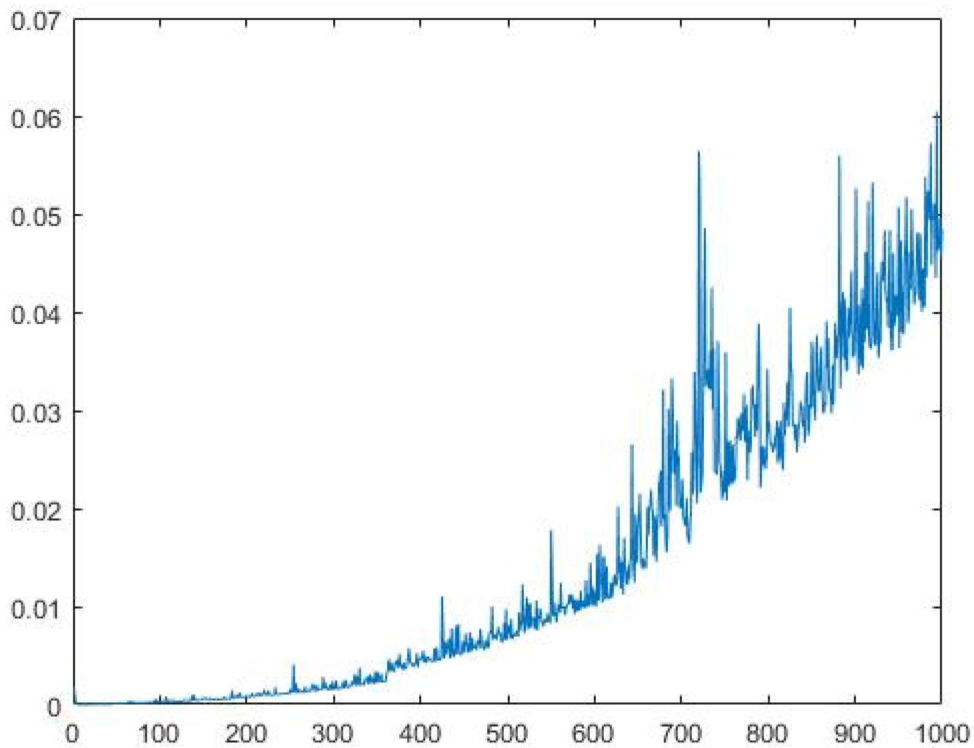
Cependant, très souvent cette méthode ne fonctionne pas ou mal car elle n'est pas stable. Par exemple

```
A = [8 1 6
     3 5 7
     4 9 2];
[L,U] = lu(A);
```

Il suffit d'ajouter une matrice de permutation telle que PA = LU:

```
[L,U,P]=lu(A);
```

Dans ce qui suit, nous allons maintenant tracer le runtime de `lu 'dépendant de la racine carrée des éléments de la matrice.

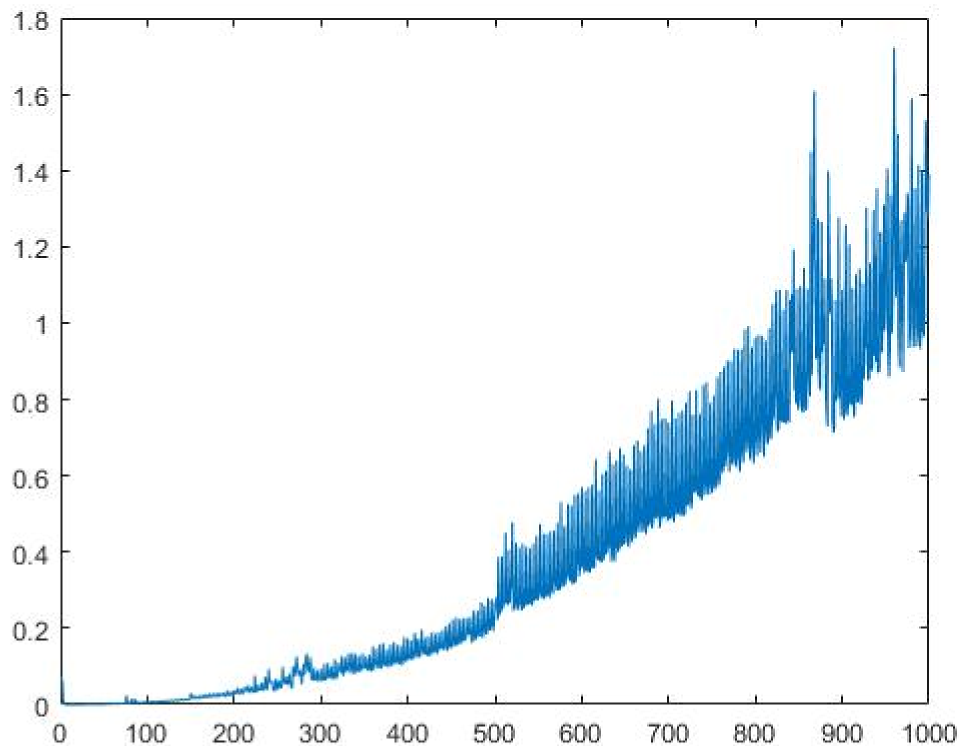


Décomposition de schur

Si A est une matrice complexe et quadratique, il existe un Q unitaire tel que Q * AQ = T = D + N avec D étant la matrice diagonale composée des valeurs propres et N étant strictement tridiagonal supérieur.

```
A = [3 6 1
    23 13 1
    0 3 4];
T = schur(A);
```

Nous `schur` également le temps d'exécution de `schur` dépendant de la racine carrée des éléments de la matrice:



Décomposition en valeur singulière

Étant donné un m fois n matrice A avec n plus grand que m. La décomposition en valeurs singulières

```
[U,S,V] = svd(A);
```

calcule les matrices U, S, V.

La matrice U est constituée des vecteurs propres singuliers de gauche qui sont les vecteurs propres de `A*A.'` tandis que V est constitué des valeurs propres singulières droites qui sont les vecteurs propres de `A.'*A` La matrice `S` a les racines carrées des valeurs propres de `A*A.'` et `A.'*A` sur sa diagonale.

Si m est plus grand que n on peut utiliser

```
[U,S,V] = svd(A,'econ');
```

effectuer une décomposition en valeur singulière économique.

Lire Décompositions matricielles en ligne: https://riptutorial.com/fr/matlab/topic/6163/decompositions-matricielles

Chapitre 7: Définir les opérations

Syntaxe

1. C = union (A, B);
2. C = intersection (A, B);
3. C = setdiff (A, B);
4. a = membre (A, x);

Paramètres

Paramètre	Détails
UN B	ensembles, éventuellement matrices ou vecteurs
X	élément possible d'un ensemble

Examples

Opérations élémentaires

Il est possible d'effectuer des opérations élémentaires avec Matlab. Supposons que nous ayons donné deux vecteurs ou tableaux

```
A = randi([0 10],1,5);
B = randi([-1 9], 1,5);
```

et nous voulons trouver tous les éléments qui sont en `A` et en `B` Pour cela nous pouvons utiliser

```
C = intersect(A,B);
```

`C` comprendra tous les nombres qui font partie de `A` et une partie de `B` Si nous voulons aussi trouver la position de ces éléments, nous appelons

```
[C,pos] = intersect(A,B);
```

`pos` est la position de ces éléments tels que `C == A(pos)` .

Une autre opération de base est l'union de deux ensembles

```
D = union(A,B);
```

Herby contient `D` tous les éléments de `A` et `B`

Notez que `A` et `B` sont traités comme des ensembles, ce qui signifie que peu importe la fréquence avec laquelle un élément fait partie de `A` ou de `B` Pour clarifier cela, on peut vérifier `D == union(D,C)` .

Si nous voulons obtenir les données qui sont dans «A» mais pas dans «B», nous pouvons utiliser la fonction suivante

```
E = setdiff(A,B);
```

Nous voulons noter à nouveau que ce sont des ensembles tels que l'instruction suivante contient `D == union(E,B)` .

Supposons que nous voulons vérifier si

```
x = randi([-10 10],1,1);
```

est un élément de `A` ou `B` nous pouvons exécuter la commande

```
a = ismember(A,x);
b = ismember(B,x);
```

Si `a==1` alors `x` est un élément de `A` et `x` un élément est `a==0` . La même chose vaut pour `B` Si `a==1 && b==1` `x` est aussi un élément de `C` Si `a == 1 || b == 1` `x` est un élément de `D` et si `a == 1 || b == 0` c'est aussi un élément de `E`

Lire Définir les opérations en ligne: https://riptutorial.com/fr/matlab/topic/3242/definir-les-operations

Chapitre 8: Dessin

Examples

Des cercles

L'option la plus simple pour dessiner un cercle est évidemment la fonction `rectangle` .

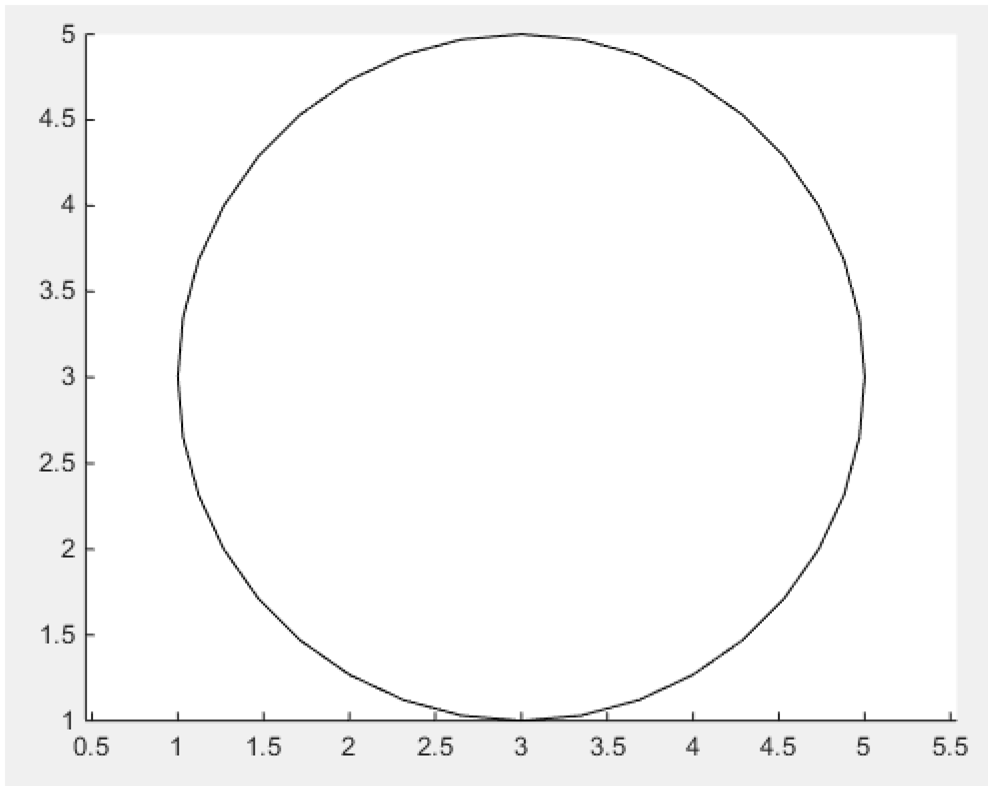
```
%// radius
r = 2;

%// center
c = [3 3];

pos = [c-r 2*r 2*r];
rectangle('Position',pos,'Curvature',[1 1])
axis equal
```

mais la courbure du rectangle doit être réglée sur **1** !

Le vecteur de `position` définit le rectangle, les deux premières valeurs `x` et `y` sont le coin inférieur gauche du rectangle. Les deux dernières valeurs définissent la largeur et la hauteur du rectangle.

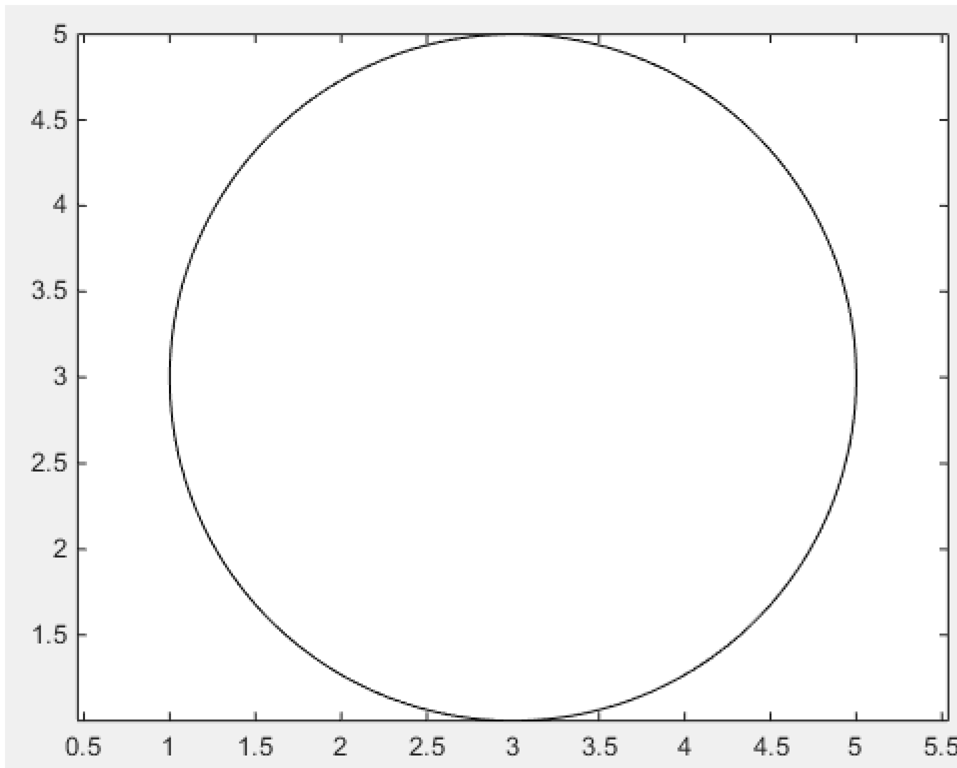
```
pos = [ [x y] width height ]
```

Le *coin* inférieur gauche du cercle - oui, ce cercle a des coins, mais imaginaires - est le **centre** `c = [3 3]` **moins le rayon** `r = 2` qui est `[xy] = [1 1]` . **La largeur** et la **hauteur** sont égales au **diamètre** du cercle, donc `width = 2*r; height = width;`



Dans le cas où la finesse de la solution ci-dessus n'est pas suffisante, il n'y a pas moyen de contourner le moyen évident de dessiner un cercle réel en utilisant des **fonctions trigonométriques** .

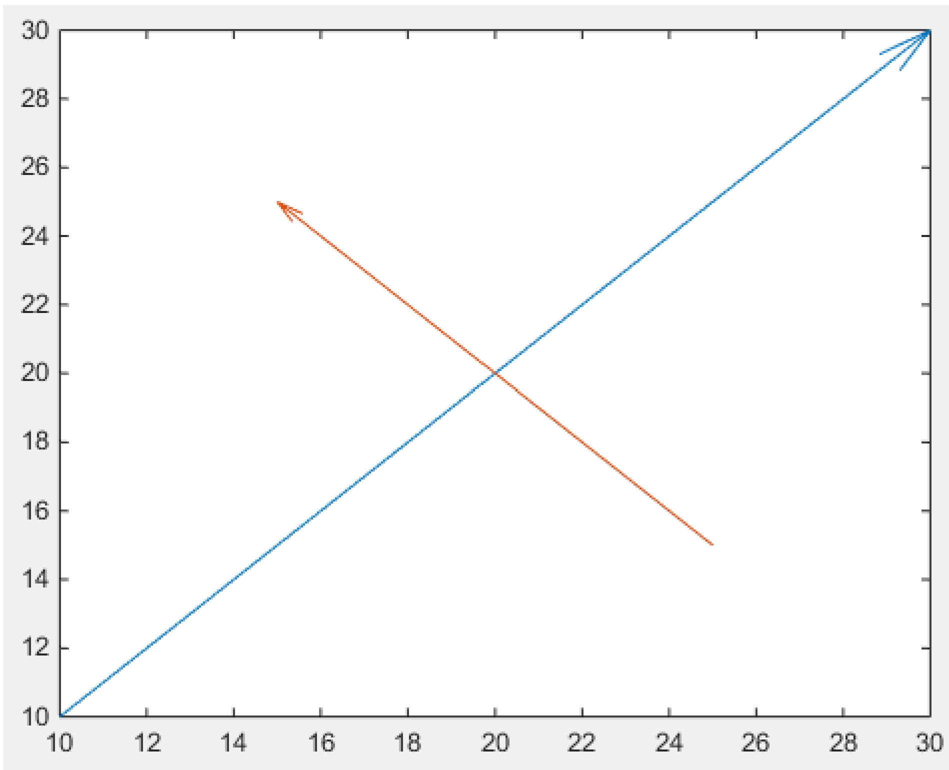
```
%// number of points
n = 1000;

%// running variable
t = linspace(0,2*pi,n);

x = c(1) + r*sin(t);
y = c(2) + r*cos(t);

%// draw line
line(x,y)

%// or draw polygon if you want to fill it with color
%// fill(x,y,[1,1,1])
axis equal
```



Flèches

Tout d'abord, on peut utiliser le `quiver` , où l'on n'a pas à traiter avec des unités de chiffres normalisées peu maniables en utilisant des `annotation`

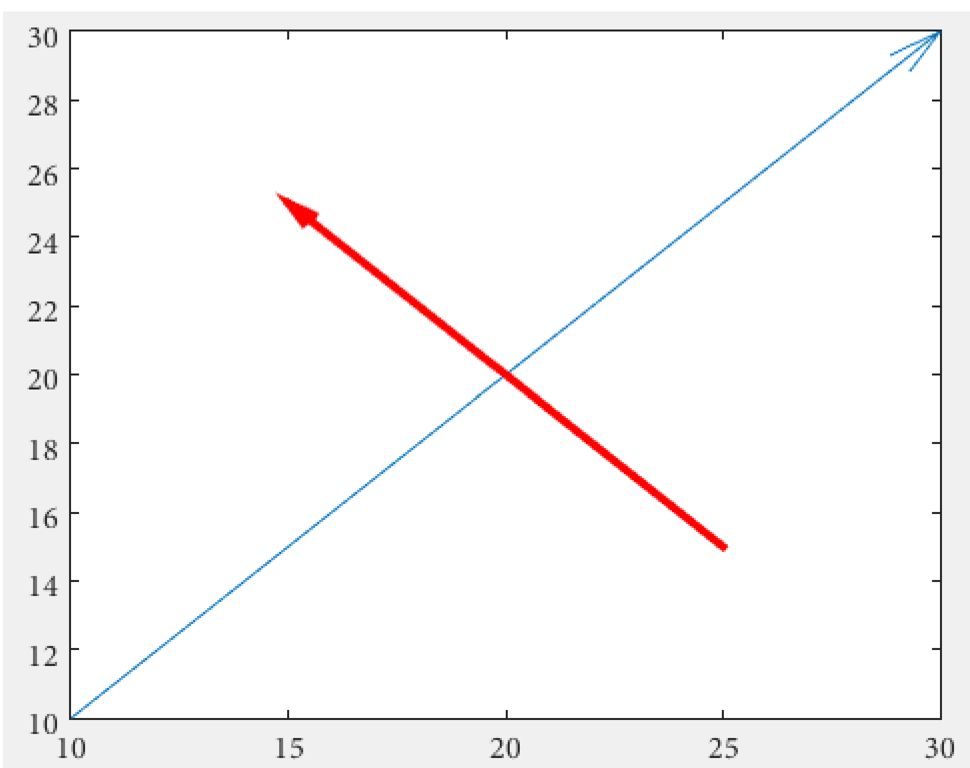
```
drawArrow = @(x,y) quiver( x(1),y(1),x(2)-x(1),y(2)-y(1),0 )

x1 = [10 30];
y1 = [10 30];

drawArrow(x1,y1); hold on

x2 = [25 15];
y2 = [15 25];

drawArrow(x2,y2)
```



Important est le cinquième argument de `quiver` : 0 qui désactive une mise à l'échelle par défaut, car cette fonction est généralement utilisée pour tracer des champs de vecteurs. (ou utilisez la paire de valeurs de propriétés `'AutoScale','off'`)

On peut également ajouter des fonctionnalités supplémentaires:

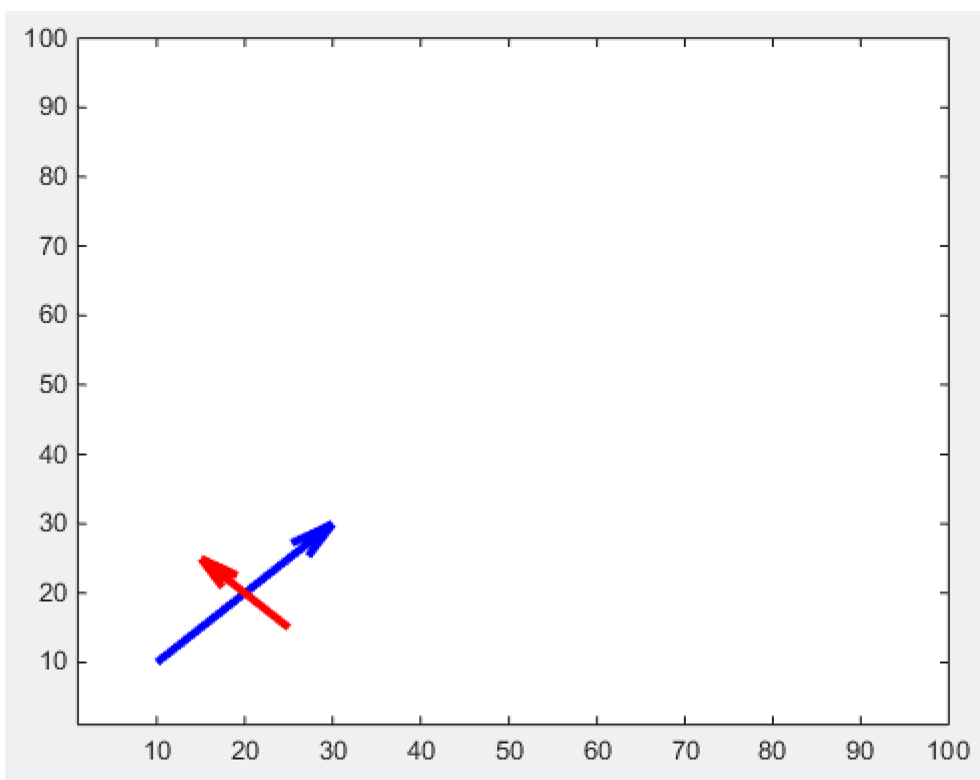
```
drawArrow = @(x,y,varargin) quiver( x(1),y(1),x(2)-x(1),y(2)-y(1),0, varargin{:} )
drawArrow(x1,y1); hold on
drawArrow(x2,y2,'linewidth',3,'color','r')
```



Si des pointes de flèches différentes sont souhaitées, il faut utiliser des annotations (cette réponse peut être utile). Comment puis-je modifier le style de la flèche dans le diagramme de carquois?

La taille de la tête de flèche peut être `'MaxHeadSize'` avec la propriété `'MaxHeadSize'` . Ce n'est pas cohérent malheureusement. Les limites des axes doivent être définies ultérieurement.

```
x1 = [10 30];
y1 = [10 30];
drawArrow(x1,y1,{'MaxHeadSize',0.8,'Color','b','LineWidth',3}); hold on

x2 = [25 15];
y2 = [15 25];
drawArrow(x2,y2,{'MaxHeadSize',10,'Color','r','LineWidth',3}); hold on

xlim([1, 100])
ylim([1, 100])
```

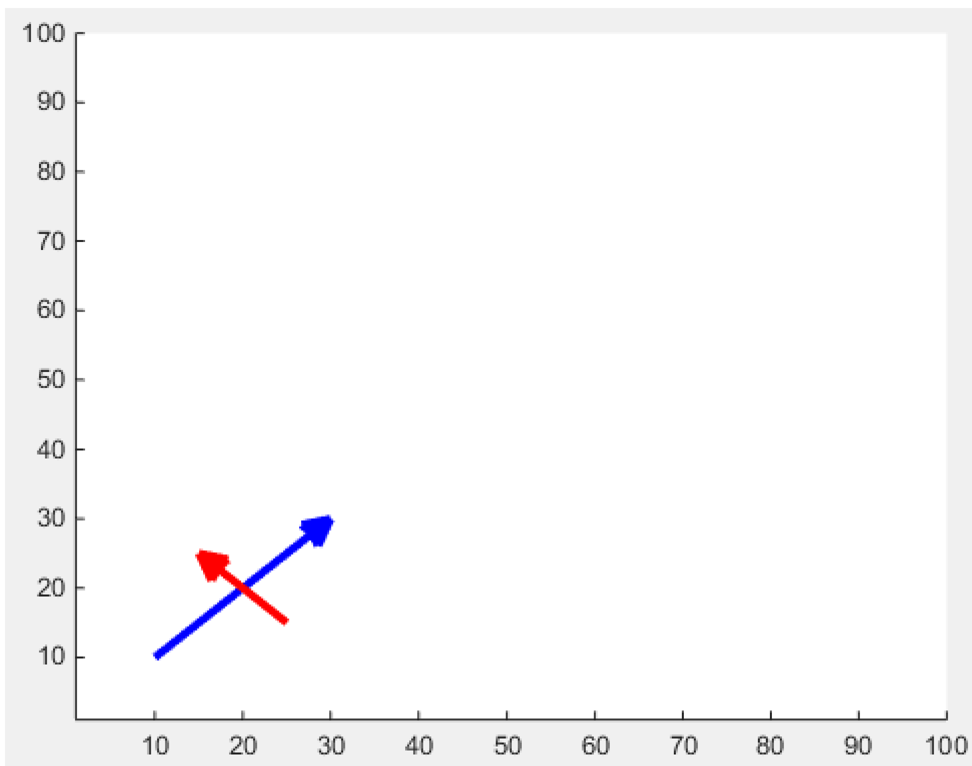


Il y a un autre ajustement pour les têtes de flèche réglables:

```
function [ h ] = drawArrow( x,y,xlimits,ylimits,props )

xlim(xlimits)
ylim(ylimits)

h = annotation('arrow');
set(h,'parent', gca, ...
    'position', [x(1),y(1),x(2)-x(1),y(2)-y(1)], ...
    'HeadLength', 10, 'HeadWidth', 10, 'HeadStyle', 'cback1', ...
    props{:} );

end
```

que vous pouvez appeler depuis votre script comme suit:

```
drawArrow(x1,y1,[1, 100],[1, 100],{'Color','b','LineWidth',3}); hold on
drawArrow(x2,y2,[1, 100],[1, 100],{'Color','r','LineWidth',3}); hold on
```



Ellipse

Pour tracer une ellipse, vous pouvez utiliser son équation . Une ellipse a un axe majeur et un axe mineur. Nous voulons également pouvoir tracer l'ellipse sur différents points centraux. Nous écrivons donc une fonction dont les entrées et les sorties sont:

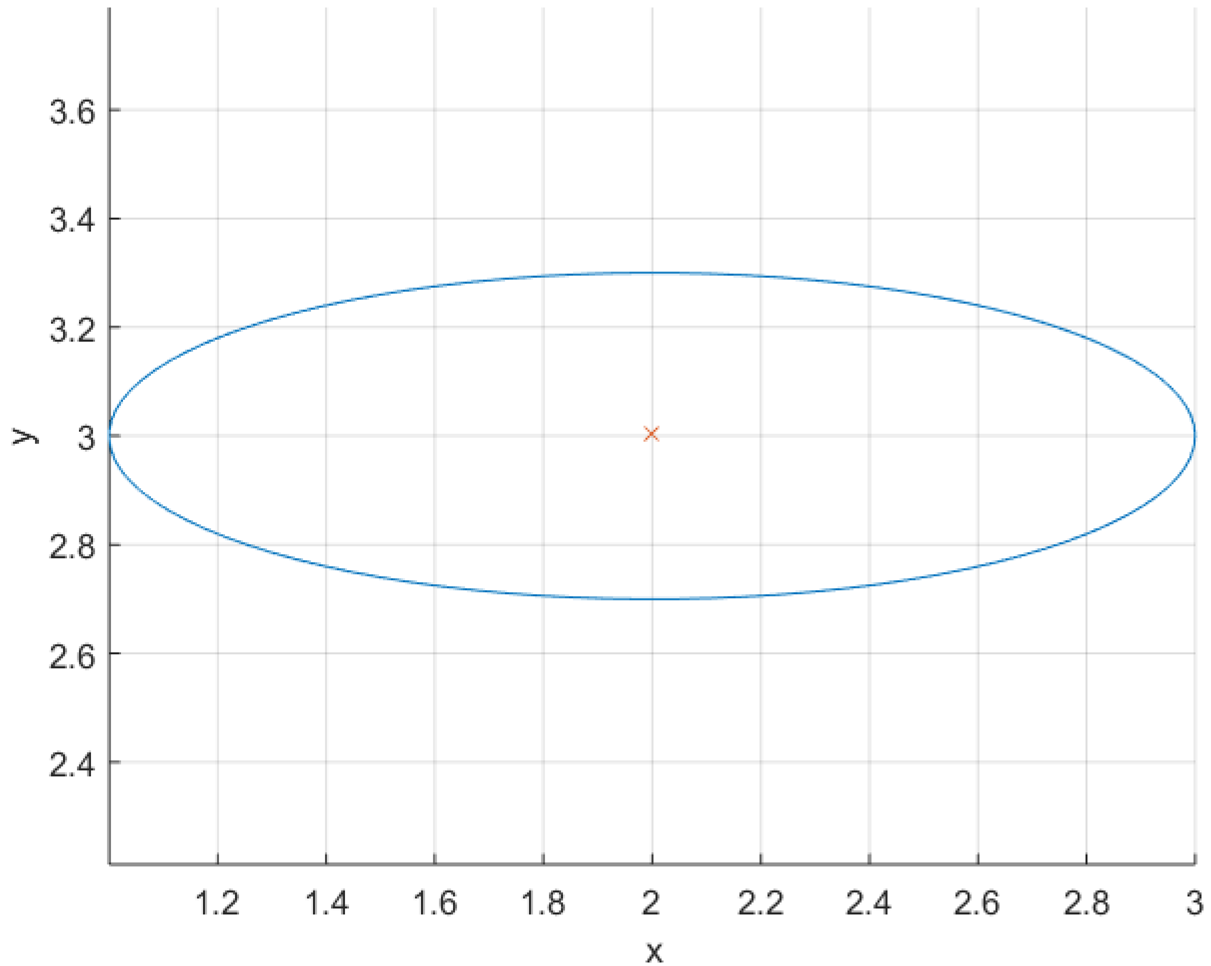
```
Inputs:
    r1,r2: major and minor axis respectively
    C: center of the ellipse (cx,cy)
Output:
    [x,y]: points on the circumference of the ellipse
```

Vous pouvez utiliser la fonction suivante pour obtenir les points sur une ellipse, puis tracer ces points.

```
function [x,y] = getEllipse(r1,r2,C)
beta = linspace(0,2*pi,100);
x = r1*cos(beta) - r2*sin(beta);
y = r1*cos(beta) + r2*sin(beta);
x = x + C(1,1);
y = y + C(1,2);
end
```

Exmaple:

```
[x,y] = getEllipse(1,0.3,[2 3]);
plot(x,y);
```

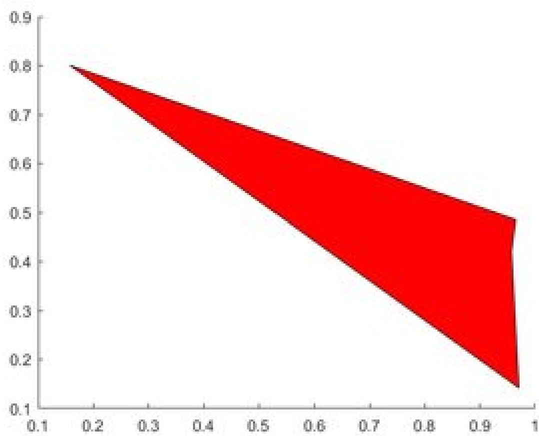


Polygone (s)

Créez des vecteurs pour contenir les emplacements x et y des sommets, insérez-les dans le `patch` .

Polygone unique

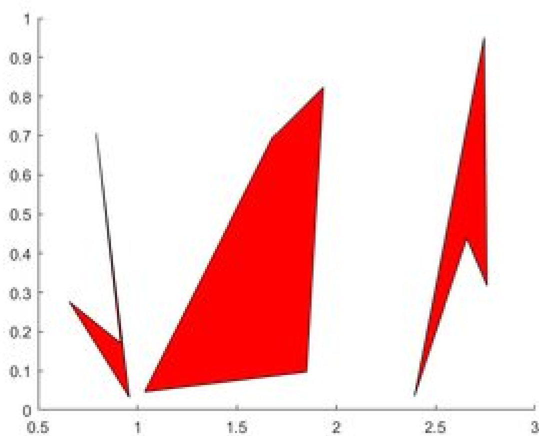
```
X=rand(1,4); Y=rand(1,4);
h=patch(X,Y,'red');
```



Plusieurs polygones

Les sommets de chaque polygone occupent une colonne de chacun des `X` , `Y`

```
X=rand(4,3); Y=rand(4,3);
for i=2:3
  X(:,i)=X(:,i)+(i-1); % create horizontal offsets for visibility
end

h=patch(X,Y,'red');
```



Tracé de pseudo 4D

Une matrice `(mxn)` peut être représentée par une surface en utilisant le surf ;

La couleur de la surface est automatiquement définie en fonction des valeurs de la matrice `(mxn)` . Si la palette de couleurs n'est pas spécifiée, celle par défaut est appliquée.

Une barre de couleur peut être ajoutée pour afficher la palette de couleurs actuelle et indiquer le mappage des valeurs de données dans la palette de couleurs.

Dans l'exemple suivant, la matrice `z (mxn)` est générée par la fonction:

```
z=x.*y.*sin(x).*cos(y);
```

sur l'intervalle `[-pi,pi]` . Les valeurs `x` et `y` peuvent être générées à l'aide de la fonction meshgrid et la surface est rendue comme suit:

```
% Create a Figure
figure
% Generate the `x` and `y` values in the interval `[-pi,pi]`
[x,y] = meshgrid([-pi:.2:pi],[-pi:.2:pi]);
% Evaluate the function over the selected interval
z=x.*y.*sin(x).*cos(y);
% Use surf to plot the surface
S=surf(x,y,z);
xlabel('X Axis');
ylabel('Y Axis');
zlabel('Z Axis');
grid minor
colormap('hot')
colorbar
```

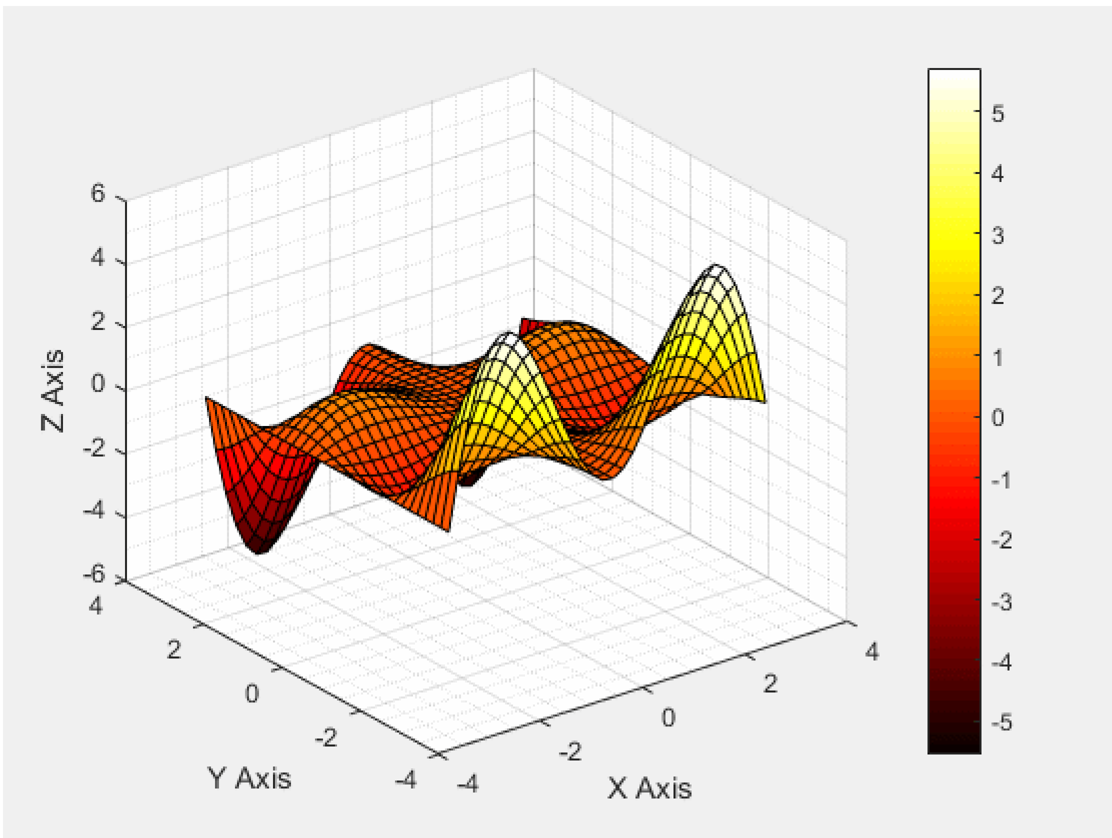


Figure 1

Maintenant, il se peut que des informations supplémentaires soient liées aux valeurs de la matrice `z` et qu'elles soient stockées dans une autre matrice `(mxn)`

Il est possible d'ajouter ces informations supplémentaires sur le tracé en modifiant la couleur de la surface.

Cela permet d'avoir un peu de 4D plot: à la représentation 3D de la surface générée par la première matrice `(mxn)` , la quatrième dimension sera représentée par les données contenues

dans la deuxième matrice `(mxn)` .

Il est possible de créer un tel complot en appelant `surf` avec 4 entrées:

```
surf(x,y,z,C)
```

où le paramètre `C` est la seconde matrice (qui doit avoir la même taille de `z`) et sert à définir la couleur de la surface.

Dans l'exemple suivant, la matrice `C` est générée par la fonction:

```
C=10*sin(0.5*(x.^2.+y.^2))*33;
```

sur l'intervalle `[-pi,pi]`

La surface générée par `C` est

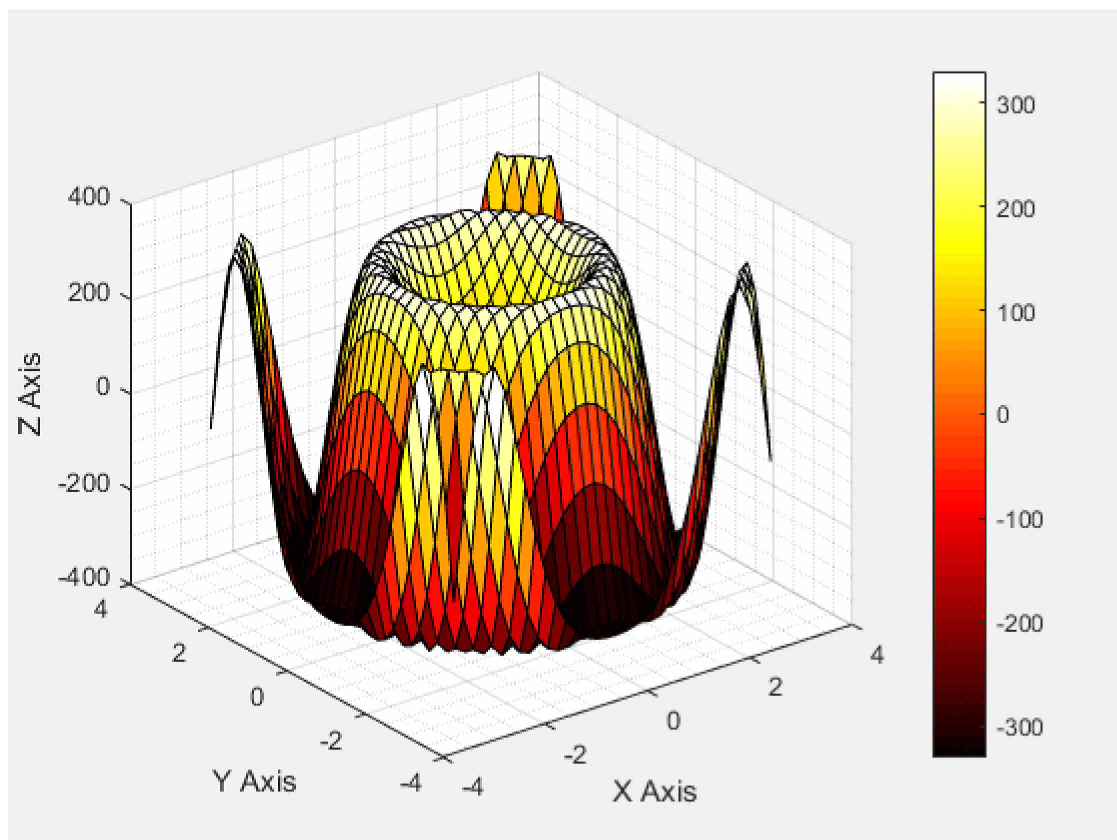


Figure 2

Maintenant, nous pouvons appeler `surf` avec quatre entrées:

```
figure
surf(x,y,z,C)
% shading interp
xlabel('X Axis');
ylabel('Y Axis');
zlabel('Z Axis');
grid minor
colormap('hot')
```

```
colorbar
```

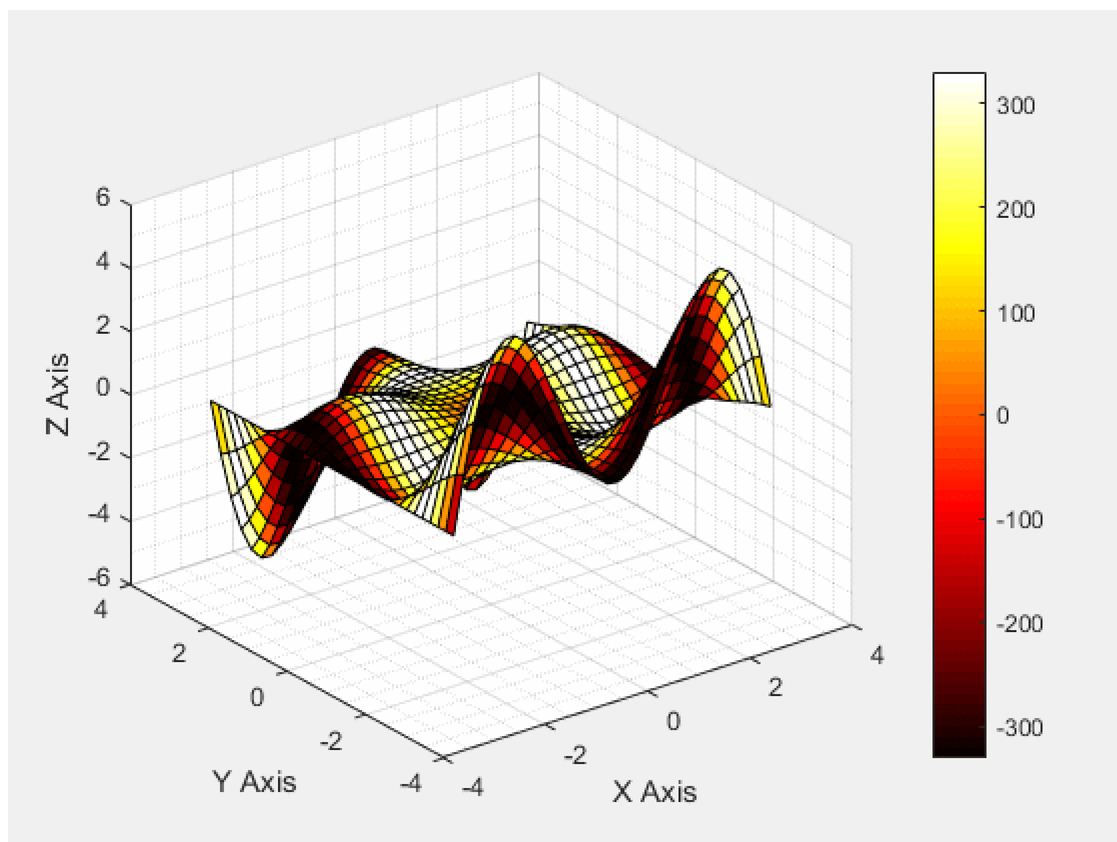


figure 3

En comparant les figures 1 et 3, on peut noter que:

- la forme de la surface correspond aux valeurs `z` (la première matrice `(mxn)`)
- la couleur de la surface (et son étendue, donnée par la barre de couleur) correspond aux valeurs `C` (la première matrice `(mxn)`)

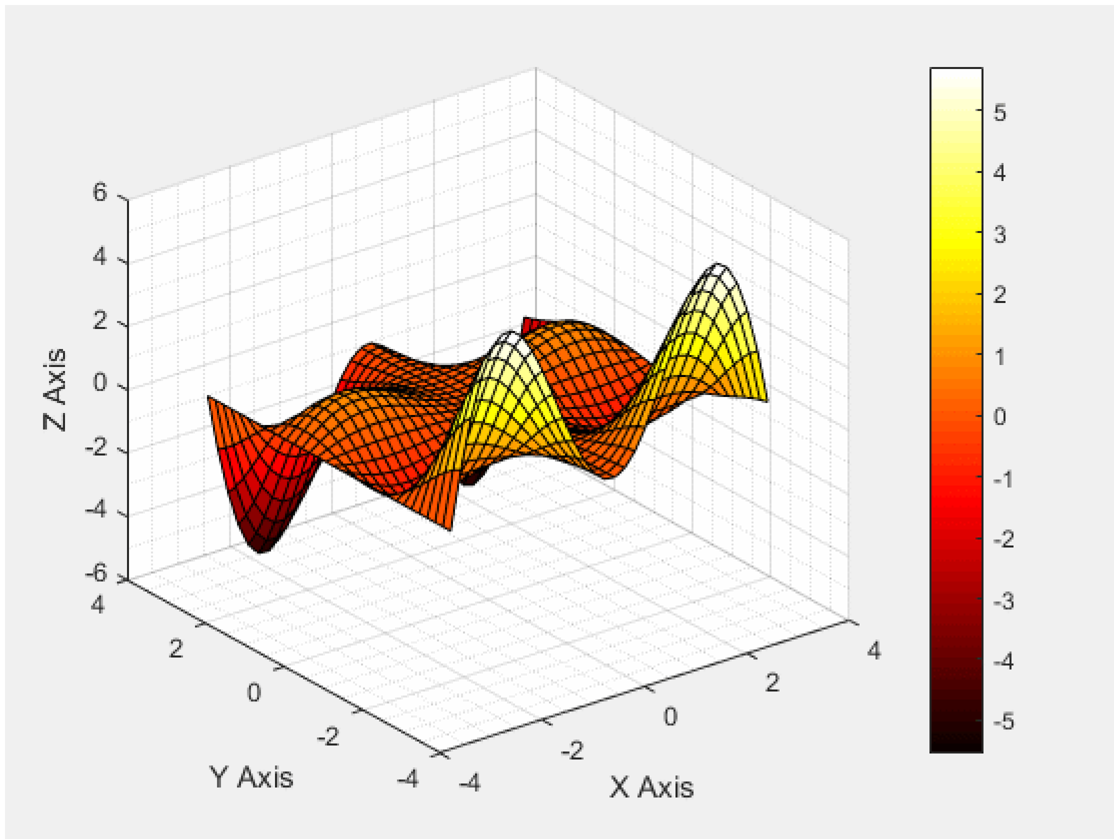
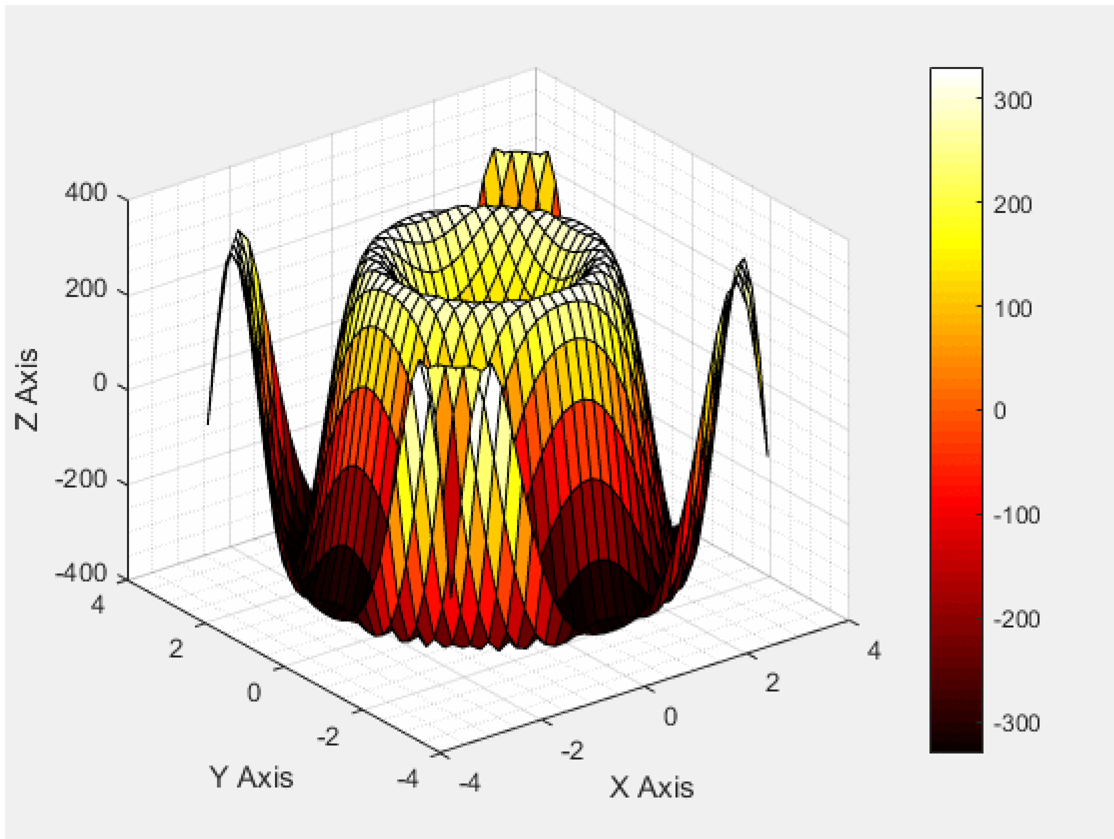


Figure 4

Bien entendu, il est possible de permuter `z` et `C` dans le tracé pour avoir la forme de la surface donnée par la matrice `C` et la couleur donnée par la matrice `z` :

```
figure
surf(x,y,C,z)
% shading interp
xlabel('X Axis');
ylabel('Y Axis');
zlabel('Z Axis');
grid minor
colormap('hot')
colorbar
```

et comparer la figure 2 avec la figure 4



Dessin rapide

Il y a trois manières principales de faire un tracé séquentiel ou des animations: `plot(x,y)` , `set(h , 'XData' , y, 'YData' , y)` et ligne `animatedline` . Si vous voulez que votre animation soit fluide, vous avez besoin d'un dessin efficace et les trois méthodes ne sont pas équivalentes.

```
% Plot a sin with increasing phase shift in 500 steps
x = linspace(0 , 2*pi , 100);

figure
tic
for thetha = linspace(0 , 10*pi , 500)
    y = sin(x + thetha);
    plot(x,y)
    drawnow
end
toc
```

Je reçois `5.278172 seconds` . La fonction de tracé supprime et recrée essentiellement l'objet ligne à chaque fois. Un moyen plus efficace de mettre à jour un tracé consiste à utiliser les propriétés `XData` et `YData` de l'objet `Line` .

```
tic
h = [];    % Handle of line object
for thetha = linspace(0 , 10*pi , 500)
    y = sin(x + thetha);

    if isempty(h)
        % If Line still does not exist, create it
        h = plot(x,y);
```

```
    else
        % If Line exists, update it
        set(h , 'YData' , y)
    end
    drawnow
end
toc
```

Maintenant, j'obtiens `2.741996 seconds` , beaucoup mieux!

`animatedline` est une fonction relativement nouvelle, introduite en 2014b. Voyons comment ça se passe:

```
tic
h = animatedline;
for thetha = linspace(0 , 10*pi , 500)
    y = sin(x + thetha);
    clearpoints(h)
    addpoints(h , x , y)
    drawnow
end
toc
```

`3.360569 seconds` , pas aussi bon que la mise à jour d'un tracé existant, mais toujours meilleur que le `plot(x,y)` .

Bien sûr, si vous devez tracer une seule ligne, comme dans cet exemple, les trois méthodes sont presque équivalentes et donnent des animations fluides. Mais si vous avez des tracés plus complexes, la mise à jour des objets `Line` existants fera une différence.

Lire Dessin en ligne: https://riptutorial.com/fr/matlab/topic/3978/dessin

Chapitre 9: Erreurs communes et erreurs

Examples

Ne nommez pas de variable avec un nom de fonction existant

Il existe déjà une fonction `sum()` . Par conséquent, si nous nommons une variable du même nom

```
sum = 1+3;
```

et si nous essayons d'utiliser la fonction alors que la variable existe toujours dans l'espace de travail

```
A = rand(2);
sum(A,1)
```

nous aurons l' **erreur** cryptique:

```
Subscript indices must either be real positive integers or logicals.
```

`clear()` abord la variable `clear()` puis utiliser la fonction

```
clear sum

sum(A,1)
ans =
       1.0826       1.0279
```

Comment pouvons-nous vérifier si une fonction existe déjà pour éviter ce conflit?

Utilisez `which()` avec le drapeau `-all` :

```
which sum -all
sum is a variable.
built-in (C:\Program Files\MATLAB\R2016a\toolbox\matlab\datafun\@double\sum)    % Shadowed
double method
...
```

Cette sortie nous indique que `sum` est d'abord une variable et que les méthodes (fonctions) suivantes sont masquées, c'est-à-dire que MATLAB essaiera d'abord d'appliquer notre syntaxe à la variable, plutôt que d'utiliser la méthode.

Ce que vous voyez n'est pas ce que vous obtenez: char vs cellstring dans la fenêtre de commande

Ceci est un exemple de base destiné aux nouveaux utilisateurs. Il ne se concentre pas sur

l'explication de la différence entre le `char` et les `cellstring` .

Il se peut que vous vouliez vous débarrasser du `'` dans vos chaînes", bien que vous ne les ayez jamais ajoutées. En fait, ce sont des *artefacts* que la **fenêtre de commande** utilise pour distinguer certains types.

Une chaîne imprimera

```
s = 'dsadasd'
s =
dsadasd
```

Une chaîne de caractères s'imprimera

```
c = {'dsadasd'};
c =
    'dsadasd'
```

Notez comment les **guillemets simples** et **l'empreinte** sont des artefacts pour nous signaler que `c` est un `cellstring` plutôt que d' un `char` . La chaîne est en fait contenue dans la cellule, c.-à-d.

```
c{1}
ans =
dsadasd
```

Les opérateurs de transposition

- `.'` est la manière correcte de **transposer** un vecteur ou une matrice dans MATLAB.
- `'` est la manière correcte de prendre la **transposée complexe conjuguée** (alias conjugué Hermitien) d'un vecteur ou d'une matrice dans MATLAB.

Notez que pour la transposition `.'` , il y a une **période** devant l'apostrophe. Ceci est conforme à la syntaxe des autres opérations élémentaires dans MATLAB: `*` multiplie les *matrices,*. `.*` Multiplie les *éléments des matrices* ensemble. Les deux commandes sont très similaires, mais conceptuellement très distinctes. Comme les autres commandes MATLAB, ces opérateurs sont des "sucres syntaxiques" qui sont transformés en un appel de fonction "correct" à l'exécution. Tout comme `==` devient une évaluation de la fonction eq , pensez à `.'` comme raccourci pour `transpose` . Si vous ne vouliez écrire que `'` (sans le point), vous utilisez en fait la commande `ctranspose` , qui calcule la transposition complexe conjuguée , également connue sous le nom de conjugué hermitien , souvent utilisée en physique. Tant que le vecteur ou la matrice transposés ont une valeur réelle, les deux opérateurs produisent le même résultat. Mais dès que nous traitons des nombres complexes , nous rencontrerons inévitablement des problèmes si nous n'utilisons pas le raccourci "correct". Ce qui est "correct" dépend de votre application.

Prenons l'exemple suivant d'une matrice `C` contenant des nombres complexes:

```
>> C = [1i, 2; 3*1i, 4]
C =
```

```
   0.0000 + 1.0000i   2.0000 + 0.0000i
   0.0000 + 3.0000i   4.0000 + 0.0000i
```

Prenons la *transposition en* utilisant la sténographie `.'` (avec la période). La sortie est comme prévu, la forme transposée de `C`

```
>> C.'
ans =
   0.0000 + 1.0000i   0.0000 + 3.0000i
   2.0000 + 0.0000i   4.0000 + 0.0000i
```

Maintenant, utilisons `'` (sans le point). Nous voyons que, outre la transposition, les valeurs complexes ont également été transformées en leurs *conjugués complexes* .

```
>> C'
ans =
   0.0000 - 1.0000i   0.0000 - 3.0000i
   2.0000 + 0.0000i   4.0000 + 0.0000i
```

En résumé, si vous souhaitez calculer le conjugué hermitien, le conjugué complexe transpose, puis utilisez `'` (sans le point). Si vous voulez simplement calculer la transposition sans conjuguer les valeurs, utilisez `.'` (avec la période).

Fonction ou méthode non définie pour les arguments d'entrée de type Y

Il s'agit de la manière très longue de MATLAB de dire qu'il ne peut pas trouver la fonction que vous essayez d'appeler. Vous pouvez obtenir cette erreur pour plusieurs raisons:

Cette fonction a été introduite *après* votre version actuelle de MATLAB

La documentation en ligne MATLAB fournit une fonctionnalité très intéressante qui vous permet de déterminer dans quelle version une fonction donnée a été introduite. Il se trouve en bas à gauche de chaque page de la documentation:

More About

▾ Tips

- The behavior of `histcounts` is similar to that of the `discre` which bin each element belongs to (without counting).

- Replace Discouraged Instances of hist and histc

See Also

`discretize` | `histcounts2` | `histogram` | `histogram2`

Introduced in R2014b

Comparez cette version avec votre propre version actuelle (`ver`) pour déterminer si cette fonction est disponible dans votre version particulière. Si ce n'est pas le cas, essayez de rechercher les versions archivées de la documentation pour trouver une alternative appropriée dans votre version.

Vous n'avez pas cette boîte à outils!

L'installation de base MATLAB a un grand nombre de fonctions; Cependant, des fonctionnalités plus spécialisées sont regroupées dans des boîtes à outils et vendues séparément par Mathworks. La documentation de *toutes les* boîtes à outils est visible, que vous ayez ou non la boîte à outils. Assurez-vous de vérifier si vous avez la boîte à outils appropriée.

Pour vérifier à quelle boîte à outils appartient une fonction donnée, regardez en haut à gauche de la documentation en ligne pour voir si une boîte à outils spécifique est mentionnée.

Documentation

CONTENTS Close

- All Products
- Image Processing Toolbox
- Display and Exploration
- Basic Display

- Image Processing Toolbox
- Geometric Transformation, Spatial Referencing, and Image Registration
- Spatial Referencing

- Image Processing Toolbox
- GPU Computing

imshow

Display image

Syntax

```
imshow(I)
imshow(I,RI)

imshow(X,map)
imshow(X,RX,map)

imshow(filename)
```

Vous pouvez ensuite déterminer les boîtes à outils que votre version de MATLAB a installées en émettant la commande `ver` qui affichera une liste de toutes les boîtes à outils installées.

Si vous ne disposez pas de cette boîte à outils et que vous souhaitez utiliser cette fonction, vous devrez acheter une licence pour cette boîte à outils particulière dans The Mathworks.

MATLAB ne peut pas localiser la fonction

Si MATLAB ne trouve toujours pas votre fonction, alors il doit s'agir d'une fonction définie par l'utilisateur. Il est possible qu'il réside dans un autre répertoire et que ce répertoire soit ajouté au chemin de recherche de votre code à exécuter. Vous pouvez vérifier si MATLAB peut localiser votre fonction en utilisant `which` qui devrait renvoyer le chemin vers le fichier source.

Soyez conscient de l'imprécision en virgule flottante

Les nombres à virgule flottante ne peuvent pas représenter tous les nombres réels. C'est ce qu'on appelle l'imprécision en virgule flottante.

Il y a une infinité de nombres en virgule flottante et ils peuvent être infiniment longs (par exemple, `π`), donc pouvoir les représenter parfaitement exigerait une quantité infinie de mémoire. Voyant que c'était un problème, une représentation spéciale pour le stockage "réel" dans l'ordinateur a été conçue, la norme IEEE 754 . En bref, il décrit comment les ordinateurs stockent ce type de nombres, avec un exposant et une mantisse, comme,

```
floatnum = sign * 2^exponent * mantissa
```

Avec une quantité limitée de bits pour chacun de ces éléments, seule une précision finie peut être obtenue. Plus le nombre est petit, plus l'écart entre les nombres possibles est faible (et vice versa!). Vous pouvez essayer vos vrais chiffres dans cette démonstration en ligne .

Soyez conscient de ce comportement et essayez d'éviter toute comparaison de points flottants et leur utilisation comme conditions d'arrêt dans les boucles. Voir ci-dessous deux exemples:

Exemples: comparaison de points flottants effectuée WRONG:

```
>> 0.1 + 0.1 + 0.1  == 0.3

ans =

  logical

   0
```

Il est peu pratique d'utiliser la comparaison en virgule flottante, comme le montre l'exemple précédent. Vous pouvez le surmonter en prenant la valeur absolue de leur différence et en la comparant à un niveau de tolérance (faible).

Vous trouverez ci-dessous un autre exemple où un nombre à virgule flottante est utilisé comme

condition d'arrêt dans une boucle while: **

```
k = 0.1;
while k <= 0.3
  disp(num2str(k));
  k = k + 0.1;
end

% --- Output: ---
0.1
0.2
```

Il manque la dernière boucle attendue (`0.3 <= 0.3`).

Exemple: comparaison de virgule flottante effectuée à DROITE:

```
x = 0.1 + 0.1 + 0.1;
y = 0.3;
tolerance = 1e-10; % A "good enough" tolerance for this case.

if ( abs( x - y ) <= tolerance )
  disp('x == y');
else
  disp('x ~= y');
end

% --- Output: ---
x == y
```

Plusieurs choses à noter:

- Comme prévu, `x` et `y` sont maintenant traités comme des équivalents.
- Dans l'exemple ci-dessus, le choix de la tolérance a été fait arbitrairement. Ainsi, la valeur choisie peut ne pas convenir à tous les cas (en particulier lorsque vous travaillez avec des nombres beaucoup plus faibles). Le choix *intelligent de* la limite peut être fait en utilisant la fonction `eps` , c'est-à-dire `N*eps(max(x,y))` , où `N` est un nombre spécifique au problème. Un choix raisonnable pour `N` , qui est également assez permissif, est `1E2` (même si, dans le problème ci-dessus, `N=1` suffirait également).

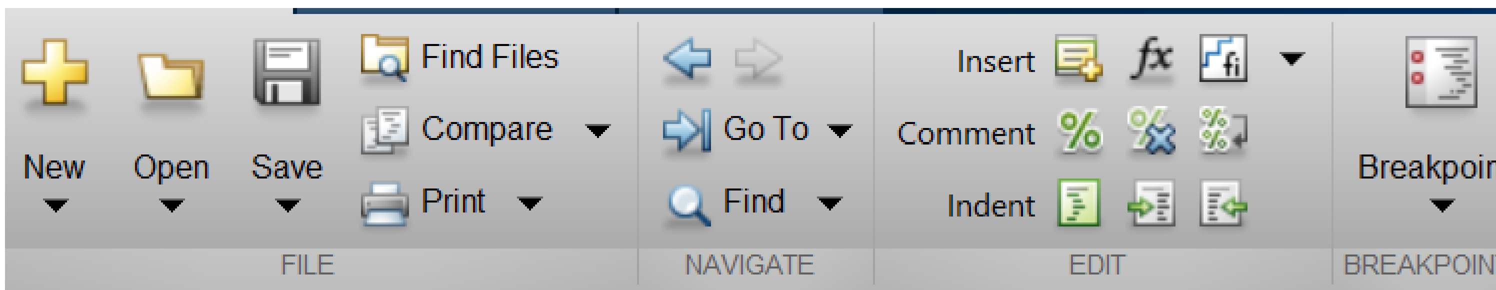
Lectures complémentaires:

Voir ces questions pour plus d'informations sur l'imprécision en virgule flottante:

- Pourquoi 24.0000 n'est-il pas égal à 24.0000 dans MATLAB?
- Le calcul en virgule flottante est-il cassé?

Pas assez d'arguments en entrée

Souvent, les développeurs MATLAB débutants utiliseront l'éditeur de MATLAB pour écrire et éditer du code, en particulier des fonctions personnalisées avec des entrées et des sorties. Un bouton *Exécuter* en haut est disponible dans les versions récentes de MATLAB:



Une fois que le développeur a fini avec le code, ils sont souvent tentés d'appuyer sur le bouton *Exécuter* . Pour certaines fonctions, cela fonctionnera correctement, mais pour d'autres, ils recevront une erreur " `Not enough input arguments` et seront perplexes quant à la raison de l'erreur.

La raison pour laquelle cette erreur peut ne pas se produire est que vous avez écrit un script MATLAB ou une fonction qui n'accepte aucun argument d'entrée. L'utilisation du bouton *Exécuter* permet d'exécuter un script de test ou d'exécuter une fonction en supposant qu'aucun argument d'entrée ne soit utilisé. Si votre fonction nécessite des arguments d'entrée, l'erreur `Not enough input arguments` se produira lorsque vous aurez écrit une fonction qui attend que des entrées entrent dans la fonction. Par conséquent, vous ne pouvez pas vous attendre à ce que la fonction s'exécute en appuyant simplement sur le bouton *Exécuter* .

Pour démontrer ce problème, supposons que nous ayons une fonction `mult` qui multiplie simplement deux matrices ensemble:

```
function C = mult(A, B)
    C = A * B;
end
```

Dans les versions récentes de MATLAB, si vous avez écrit cette fonction et que vous avez appuyé sur le bouton *Exécuter* , cela vous donnera l'erreur attendue:

```
>> mult
Not enough input arguments.

Error in mult (line 2)
    C = A * B;
```

Il existe deux manières de résoudre ce problème:

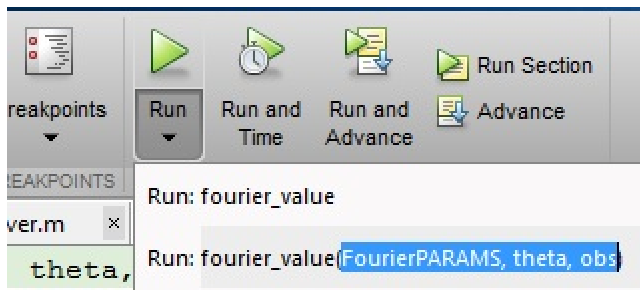
Méthode n ° 1 - via l'invite de commande

Créez simplement les entrées dont vous avez besoin dans l'invite de commandes, puis exécutez la fonction en utilisant les entrées que vous avez créées:

```
A = rand(5,5);
B = rand(5,5);
C = mult(A,B);
```

Méthode n ° 2 - Interactif via l'éditeur

Sous le bouton *Exécuter* , il y a une flèche noire foncée. Si vous cliquez sur cette flèche, vous pouvez spécifier les variables que vous souhaitez obtenir dans l'espace de travail MATLAB en saisissant la manière dont vous souhaitez appeler la fonction exactement comme vous l'avez vu dans la méthode n ° 1. Assurez-vous que les variables que vous spécifiez dans la fonction existent dans l'espace de travail MATLAB:



Attention aux changements de taille de tableau

Certaines opérations courantes dans MATLAB, telles que la **différenciation** ou l' **intégration** , produisent des résultats dont la quantité d'éléments est différente de celle des données d'entrée. Ce fait peut facilement être ignoré, ce qui entraînerait généralement des erreurs, comme les `Matrix dimensions must agree` . Prenons l'exemple suivant:

```
t = 0:0.1:10;          % Declaring a time vector
y = sin(t);            % Declaring a function

dy_dt = diff(y);       % calculates dy/dt for y = sin(t)
```

Disons que nous voulons tracer ces résultats. Nous examinons les tailles de tableau et voyons:

```
size(y) is 1x101
size(t) is 1x101
```

Mais:

```
size(dy_dt) is 1x100
```

Le tableau est un élément plus court!

Maintenant, imaginez que vous avez des données de mesure de positions dans le temps et que vous voulez calculer *jerk (t)* , vous obtiendrez un tableau 3 éléments de moins que le tableau de temps (parce que le jerk est la position différenciée 3 fois).

```
vel = diff(y);         % calculates velocity vel=dy/dt for y = sin(t)  size(vel)=1x100
acc = diff(vel);       % calculates acceleration acc=d(vel)/dt          size(acc)=1x99
jerk = diff(acc);      % calculates jerk jerk=d(acc)/dt                 size(jerk)=1x98
```

Et puis les opérations comme:

```
x = jerk .* t;          % multiplies jerk and t element wise
```

les erreurs de retour, car les dimensions de la matrice ne sont pas d'accord.

Pour calculer les opérations comme ci-dessus, vous devez ajuster la plus grande taille de tableau pour l'adapter à la plus petite. Vous pouvez également exécuter une régression (`polyfit`) avec vos données pour obtenir un polynôme pour vos données.

Erreurs d'incompatibilité de dimension

Les erreurs d' **incompatibilité de dimension** apparaissent généralement lorsque:

- Ne pas prêter attention à la forme des variables renvoyées par les appels de fonction / méthode. Dans de nombreuses fonctions MATLAB intégrées, les matrices sont converties en vecteurs pour accélérer les calculs, et la variable renvoyée peut toujours être un vecteur plutôt que la matrice attendue. Ceci est également un scénario courant lorsque le masquage logique est impliqué.
- Utilisation de tailles de tableau incompatibles lors de l'appel d' une extension de tableau implicite .

L'utilisation de "i" ou "j" comme unité imaginaire, index de boucle ou variable commune.

Recommandation

Comme les symboles `i` et `j` peuvent représenter des choses significativement différentes dans MATLAB, leur utilisation en tant qu'indices de boucle a divisé la communauté d'utilisateurs MATLAB depuis des âges. Bien que certaines raisons de performance historique puissent aider à équilibrer l'équilibre, ce n'est plus le cas et maintenant, le choix repose entièrement sur vous et sur les pratiques de codage que vous choisissez de suivre.

Les recommandations officielles actuelles de Mathworks sont les suivantes:

- Puisque `i` est une fonction, il peut être remplacé et utilisé comme variable. Cependant, il est préférable d'éviter d'utiliser `i` et `j` pour les noms de variables si vous avez l'intention de les utiliser en arithmétique complexe.
- Pour la rapidité et la robustesse améliorée de l'arithmétique complexe, utilisez `1i` et `1j` au lieu de `i` et `j` .

Défaut

Dans MATLAB, par défaut, les lettres `i` et `j` sont `function` noms de `function` intégrés, qui font tous deux référence à l'unité imaginaire dans le domaine complexe.

Donc, par défaut, `i = j = sqrt(-1)` .

```
>> i
ans =
   0.0000 + 1.0000i
>> j
ans =
   0.0000 + 1.0000i
```

et comme vous devez vous attendre:

```
>> i^2
ans =
    -1
```

Les utiliser comme une variable (pour les indices de boucle ou autre variable)

MATLAB permet d'utiliser le nom de la fonction intégrée en tant que variable standard. Dans ce cas, le symbole utilisé ne pointera plus vers la fonction intégrée mais vers votre propre variable définie par l'utilisateur. Cependant, cette pratique n'est généralement pas recommandée car elle peut entraîner de la confusion, un débogage difficile et une maintenance (*voir un autre exemple:* *do-not-name-a-variable-with-an-existing-function-name*).

Si vous êtes très pédant dans le respect des conventions et des meilleures pratiques, vous éviterez de les utiliser comme index de boucle dans ce langage. Cependant, il est autorisé par le compilateur et parfaitement fonctionnel. Vous pouvez donc choisir de conserver les anciennes habitudes et de les utiliser comme itérateurs de boucle.

```
>> A = nan(2,3);
>> for i=1:2        % perfectly legal loop construction
       for j = 1:3
        A(i, j) = 10 * i + j;
       end
   end
```

Notez que les indices de boucle ne sortent pas de la portée à la fin de la boucle, ils conservent donc leur nouvelle valeur.

```
>> [ i ; j ]
ans =
     2
     3
```

Si vous les utilisez comme variables, assurez-vous **qu'elles sont initialisées** avant leur utilisation. Dans la boucle ci-dessus, MATLAB les initialise automatiquement lorsqu'il prépare la boucle, mais s'il n'est pas initialisé correctement, vous pouvez rapidement voir que vous pouvez introduire par inadvertance `complex` nombres `complex` dans votre résultat.

Si plus tard, vous devez annuler l'observation de la fonction intégrée (= par exemple, si vous voulez que `i` et `j` représentent à nouveau l'unité imaginaire), vous pouvez `clear` les variables:

```
>> clear i j
```

Vous comprenez maintenant la réservation Mathworks à propos de leur utilisation en tant qu'indices de boucle *si vous souhaitez les utiliser en arithmétique complexe* . Votre code serait truffé d'initialisations variables et de commandes `clear` , le meilleur moyen de confondre le programmeur le plus sérieux (*oui vous êtes là! ...*) et les accidents de programme en attente.

Si aucune arithmétique complexe n'est attendue, l'utilisation de `i` et `j` est parfaitement fonctionnelle et il n'y a pas de pénalité de performance.

En les utilisant comme unité imaginaire:

Si votre code doit traiter `complex` nombres `complex` , alors `i` et `j` seront certainement utiles. Cependant, pour des raisons de désambiguïsation et même pour les performances, il est recommandé d'utiliser la forme complète au lieu de la syntaxe abrégée. La forme complète est `1i` (ou `1j`).

```
>> [ i ; j ; 1i ; 1j]
ans =
   0.0000 + 1.0000i
   0.0000 + 1.0000i
   0.0000 + 1.0000i
   0.0000 + 1.0000i
```

Ils représentent la même valeur `sqrt(-1)` , mais la dernière forme:

- est plus explicite, de manière sémantique.
- est plus maintenable (quelqu'un qui regarde votre code plus tard n'aura pas à lire le code pour savoir si `i` ou `j` était une variable ou l'unité imaginaire).
- est plus rapide (source: Mathworks).

Notez que la syntaxe complète `1i` est valide avec tout nombre précédant le symbole:

```
>> a = 3 + 7.8j
a =
   3.0000 + 7.8000i
```

C'est la seule fonction avec laquelle vous pouvez coller un numéro sans opérateur entre eux.

Pièges

Bien que leur utilisation comme *unité imaginaire* **OU** *variable* soit parfaitement légale, voici juste un petit exemple de la façon dont cela pourrait être déroutant si les deux usages étaient

mélangés:

Surpassons `i` et en faisons une variable:

```
>> i=3
i =
     3
```

Maintenant, `i` est une *variable* (contenant la valeur `3`), mais nous ne remplaçons que la notation *abrégée* de l'unité imaginaire, la forme complète est toujours interprétée correctement:

```
>> 3i
ans =
   0.0000 + 3.0000i
```

Ce qui nous permet maintenant de construire les formulations les plus obscures. Je vous laisse évaluer la lisibilité de toutes les constructions suivantes:

```
>> [ i ; 3i ; 3*i ; i+3i ; i+3*i ]
ans =
   3.0000 + 0.0000i
   0.0000 + 3.0000i
   9.0000 + 0.0000i
   3.0000 + 3.0000i
  12.0000 + 0.0000i
```

Comme vous pouvez le voir, chaque valeur du tableau ci-dessus renvoie un résultat différent. Bien que chaque résultat soit valide (à condition que ce soit l'intention initiale), la plupart d'entre vous admettront qu'il serait bon de lire un code contenant de telles constructions.

Utiliser `length` pour les tableaux multidimensionnels

Une erreur courante des codeurs MATLAB consiste à utiliser la fonction `length` pour les matrices (par opposition aux **vecteurs** auxquels elle est destinée). La fonction `length` , mentionnée dans sa documentation , " *renvoie la longueur de la plus grande dimension du tableau* " de l'entrée.

Pour les vecteurs, la valeur de retour de `length` a deux significations différentes:

1. Le nombre total d'éléments dans le vecteur.
2. La plus grande dimension du vecteur.

Contrairement aux vecteurs, les valeurs ci-dessus ne seraient pas égales pour les tableaux de plus d'une dimension non-singleton (c'est-à-dire dont la taille est supérieure à `1`). C'est pourquoi l'utilisation de la `length` pour les matrices est ambiguë. Au lieu de cela, l'utilisation de l'une des fonctions suivantes est encouragée, même lorsque vous travaillez avec des vecteurs, pour que l'intention du code soit parfaitement claire:

1. `size(A)` - renvoie un vecteur de ligne dont les éléments contiennent la quantité d'éléments le long de la dimension correspondante de `A`
2. `numel(A)` - renvoie le nombre d'éléments dans `A` Équivalent à `prod(size(A))` .

3. `ndims(A)` - renvoie le nombre de dimensions du tableau `A` Équivalent à `numel(size(A))` .

Ceci est particulièrement important lors de l'écriture de fonctions de bibliothèque vectorisées "à l'épreuve du futur", dont les entrées ne sont pas connues à l'avance et peuvent avoir différentes tailles et formes.

Lire Erreurs communes et erreurs en ligne: https://riptutorial.com/fr/matlab/topic/973/erreurs-communes-et-erreurs

Chapitre 10: Fonctionnalités non documentées

Remarques

- L'utilisation de fonctionnalités non documentées est considérée comme une pratique risquée [1] , car ces fonctionnalités peuvent changer sans préavis ou simplement fonctionner différemment sur différentes versions de MATLAB. Pour cette raison, il est conseillé d'utiliser des techniques de programmation défensives telles que la mise en place de blocs de code non documentés dans `try/catch` blocs `try/catch` avec des solutions de secours documentées.

Examples

Fonctions d'aide compatibles C ++

L'utilisation de **Matlab Coder** refuse parfois l'utilisation de certaines fonctions très courantes, si elles ne sont pas compatibles avec C ++. Il existe relativement souvent **des fonctions auxiliaires non documentées** , qui peuvent être utilisées comme substituts.

Voici une liste complète des fonctions prises en charge. .

Et en suivant une série d'alternatives, pour les fonctions non prises en charge:

Les fonctions `sprintf` et `sprintfc` sont assez similaires, les anciens renvoie un *tableau de caractères,* ce dernier une *chaîne cellulaire:*

```
str = sprintf('%i',x)    % returns '5' for x = 5
str = sprintfc('%i',x)   % returns {'5'} for x = 5
```

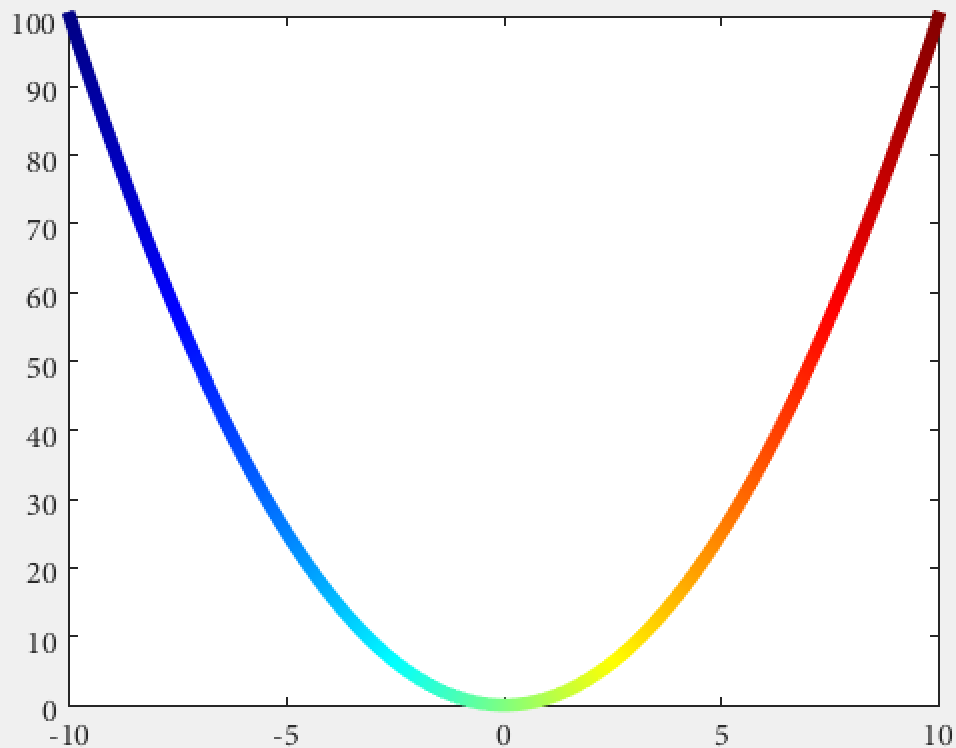
Cependant, `sprintfc` est compatible avec C ++ pris en charge par Matlab Coder, mais pas `sprintf` .

Tracés de lignes 2D à code couleur avec données de couleur en troisième dimension

Dans les versions de MATLAB antérieures à **R2014b** , utilisant l'ancien moteur graphique HG1, il n'était pas évident de créer des tracés de lignes 2D avec code couleur . Avec la sortie du nouveau moteur graphique HG2, une nouvelle **fonctionnalité non documentée a été introduite par Yair Altman** :

```
n = 100;
x = linspace(-10,10,n); y = x.^2;
p = plot(x,y,'r', 'LineWidth',5);
```

```
% modified jet-colormap
cd = [uint8(jet(n)*255) uint8(ones(n,1))].';

drawnow
set(p.Edge, 'ColorBinding','interpolated', 'ColorData',cd)
```



Marqueurs semi-transparents dans les lignes et les nuages de points

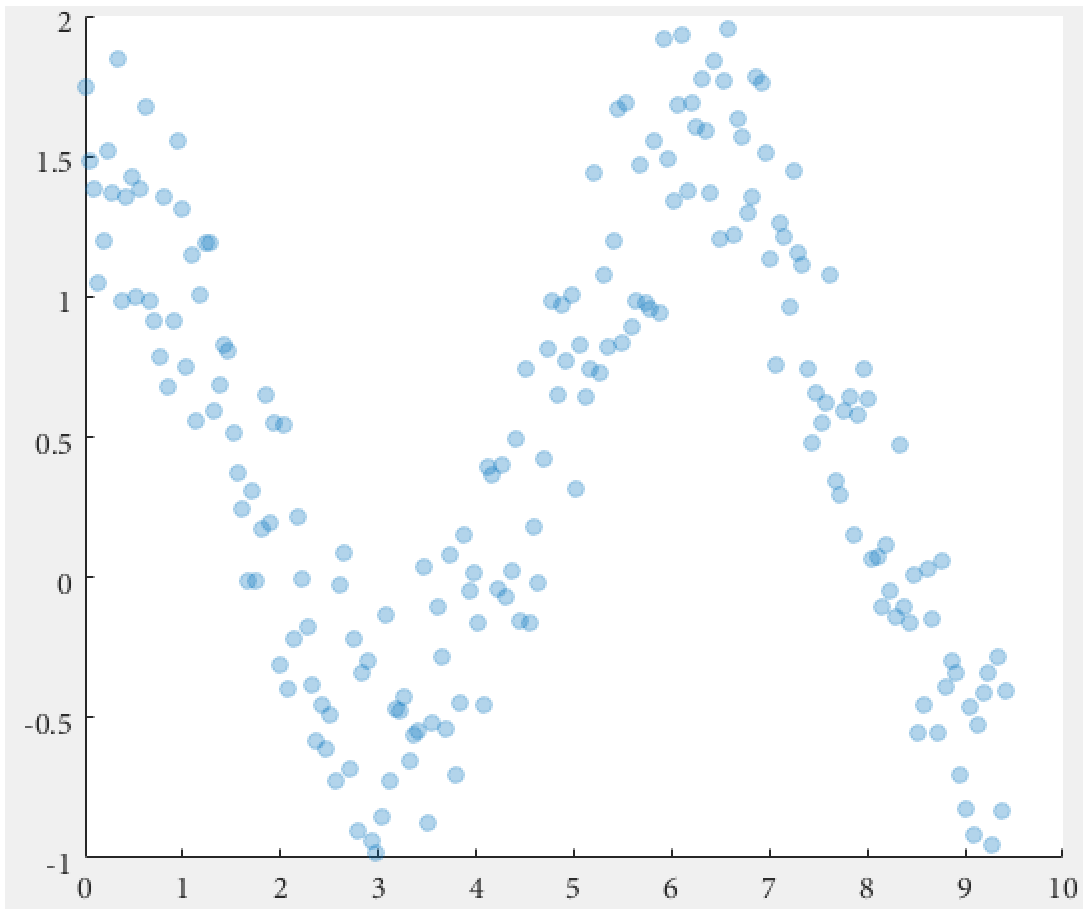
Depuis **Matlab R2014b,** il est facilement possible d'obtenir des marqueurs semi-transparents pour les tracés de lignes et de dispersion en utilisant **les fonctionnalités non documentées introduites par Yair Altman** .

L'idée de base est d'obtenir le handle caché des marqueurs et d'appliquer une valeur **<1** pour la dernière valeur dans `EdgeColorData` pour obtenir la transparence souhaitée.

Ici nous allons pour la `scatter` :

```
%// example data
x = linspace(0,3*pi,200);
y = cos(x) + rand(1,200);

%// plot scatter, get handle
h = scatter(x,y);
drawnow; %// important

%// get marker handle
hMarkers = h.MarkerHandle;

%// get current edge and face color
edgeColor = hMarkers.EdgeColorData
faceColor = hMarkers.FaceColorData
```

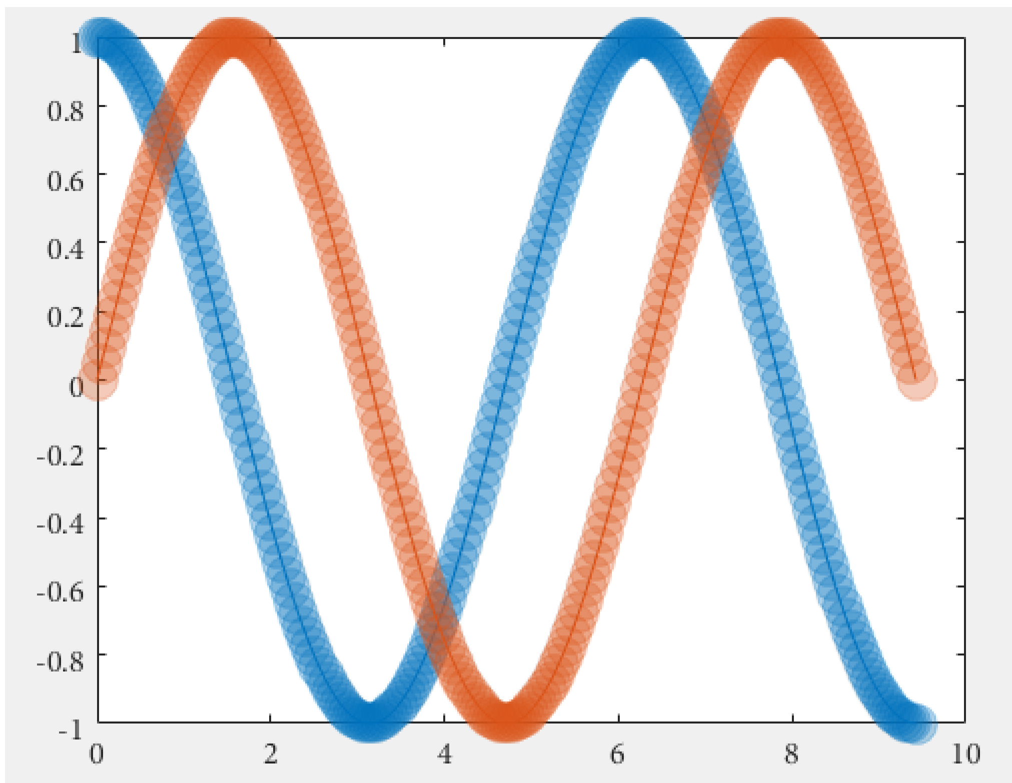
```
%// set face color to the same as edge color
faceColor = edgeColor;

%// opacity
opa = 0.3;

%// set marker edge and face color
hMarkers.EdgeColorData = uint8( [edgeColor(1:3); 255*opa] );
hMarkers.FaceColorData = uint8( [faceColor(1:3); 255*opa] );
```



et pour un `plot` ligne

```
%// example data
x = linspace(0,3*pi,200);
y1 = cos(x);
y2 = sin(x);

%// plot scatter, get handle
h1 = plot(x,y1,'o-','MarkerSize',15); hold on
h2 = plot(x,y2,'o-','MarkerSize',15);
drawnow; %// important

%// get marker handle
h1Markers = h1.MarkerHandle;
h2Markers = h2.MarkerHandle;

%// get current edge and face color
edgeColor1 = h1Markers.EdgeColorData;
edgeColor2 = h2Markers.EdgeColorData;
```

```
%// set face color to the same as edge color
faceColor1 = edgeColor1;
faceColor2 = edgeColor2;

%// opacity
opa = 0.3;

%// set marker edge and face color
h1Markers.EdgeColorData = uint8( [edgeColor1(1:3); 255*opa] );
h1Markers.FaceColorData = uint8( [faceColor1(1:3); 255*opa] );
h2Markers.EdgeColorData = uint8( [edgeColor2(1:3); 255*opa] );
h2Markers.FaceColorData = uint8( [faceColor2(1:3); 255*opa] );
```



Les poignées de marqueur utilisées pour la manipulation sont créées avec la figure. La commande `drawnow` assure la création de la figure avant l'appel des commandes suivantes et évite les erreurs en cas de retard.

Tracés de contour - Personnaliser les étiquettes de texte

Lorsque vous affichez des étiquettes sur les contours, Matlab ne vous permet pas de contrôler le format des nombres, par exemple pour passer à la notation scientifique.

Les objets texte individuels sont des objets texte normaux, mais la manière dont vous les obtenez n'est pas documentée. Vous y accédez à partir de la propriété `TextPrims` du `TextPrims` de contour.

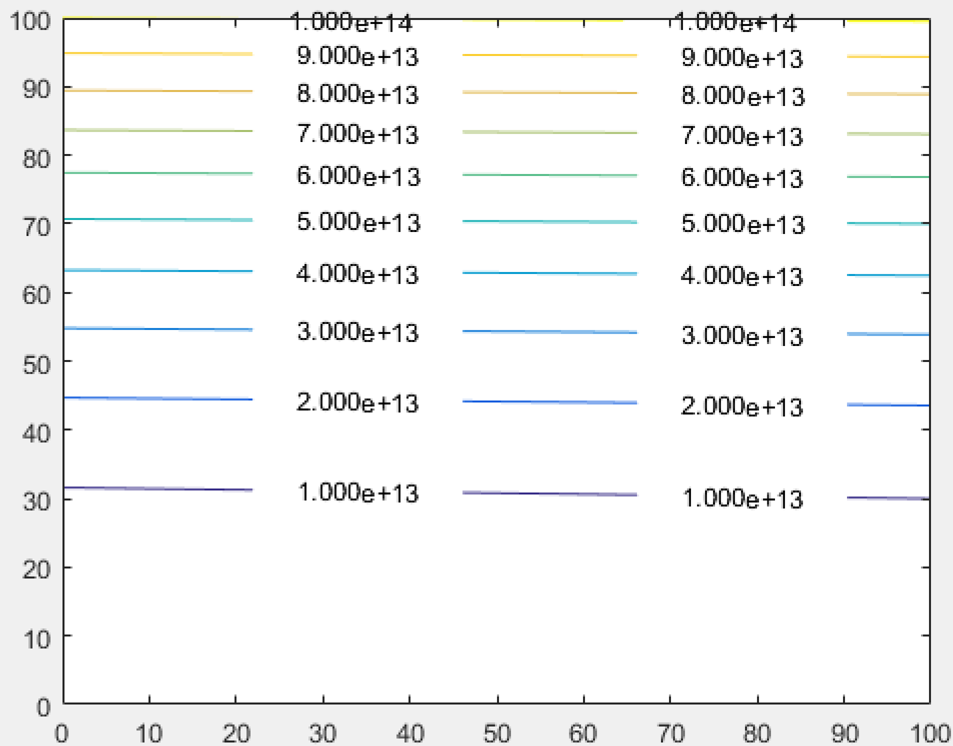
```
figure
[X,Y]=meshgrid(0:100,0:100);
Z=(X+Y.^2)*1e10;
[C,h]=contour(X,Y,Z);
h.ShowText='on';
drawnow();
txt = get(h,'TextPrims');
```

```
  v = str2double(get(txt,'String'));
  for ii=1:length(v)
    set(txt(ii),'String',sprintf('%0.3e',v(ii)))
  end
```

Remarque : vous devez ajouter une commande `drawnow` pour forcer Matlab à dessiner les contours. Le nombre et l'emplacement des objets txt ne sont déterminés que lorsque les contours sont réellement dessinés, afin que les objets texte ne soient créés qu'à ce moment-là.

Le fait que les objets txt soient créés lorsque les contours sont dessinés signifie qu'ils sont recalculés chaque fois que le tracé est redessiné (par exemple, redimensionnement de la figure). Pour gérer cela, vous devez écouter l' `undocumented event MarkedClean` :

```
function customiseContour
  figure
  [X,Y]=meshgrid(0:100,0:100);
  Z=(X+Y.^2)*1e10;
  [C,h]=contour(X,Y,Z);
  h.ShowText='on';
  % add a listener and call your new format function
  addlistener(h,'MarkedClean',@(a,b)ReFormatText(a))
end
function ReFormatText(h)
  % get all the text items from the contour
  t = get(h,'TextPrims');
  for ii=1:length(t)
    % get the current value (Matlab changes this back when it
    %   redraws the plot)
    v = str2double(get(t(ii),'String'));
    % Update with the format you want - scientific for example
    set(t(ii),'String',sprintf('%0.3e',v));
  end
end
```



Exemple testé avec Matlab r2015b sous Windows

Ajout / ajout d'entrées à une légende existante

Les légendes existantes peuvent être difficiles à gérer. Par exemple, si votre tracé comporte deux lignes, mais qu'une seule d'entre elles comporte une entrée de légende et que cela doit rester ainsi, l'ajout d'une troisième ligne avec une entrée de légende peut s'avérer difficile. Exemple:

```
figure
hold on
fplot(@sin)
fplot(@cos)
legend sin  % Add only a legend entry for sin
hTan = fplot(@tan);  % Make sure to get the handle, hTan, to the graphics object you want to
add to the legend
```

Maintenant, pour ajouter une entrée de légende pour `tan` , mais pas pour `cos` , aucune des lignes suivantes ne fera l'affaire. ils échouent tous d'une manière ou d'une autre:

```
legend tangent  % Replaces current legend -> fail
legend -DynamicLegend  % Undocumented feature, adds 'cos', which shouldn't be added -> fail
legend sine tangent  % Sets cos DisplayName to 'tangent' -> fail
legend sine '' tangent  % Sets cos DisplayName, albeit empty -> fail
legend(f)
```

Heureusement, une propriété de légende non documentée appelée `PlotChildren` garde la trace des enfants de la figure parent [1] . Ainsi, la voie à suivre est de définir explicitement les enfants de la légende à travers sa propriété `PlotChildren` comme suit:

```
hTan.DisplayName = 'tangent';  % Set the graphics object's display name
l = legend;
l.PlotChildren(end + 1) = hTan;  % Append the graphics handle to legend's plot children
```

La légende est mise à jour automatiquement si un objet est ajouté ou supprimé de sa propriété `PlotChildren` .

[1] En effet: figure. Vous pouvez ajouter un enfant de n'importe quel personnage avec la propriété `DisplayName` à une légende de la figure, par exemple à partir d'une sous-intrigue différente. En effet, une légende en elle-même est essentiellement un objet de type axe.

Testé sur MATLAB R2016b

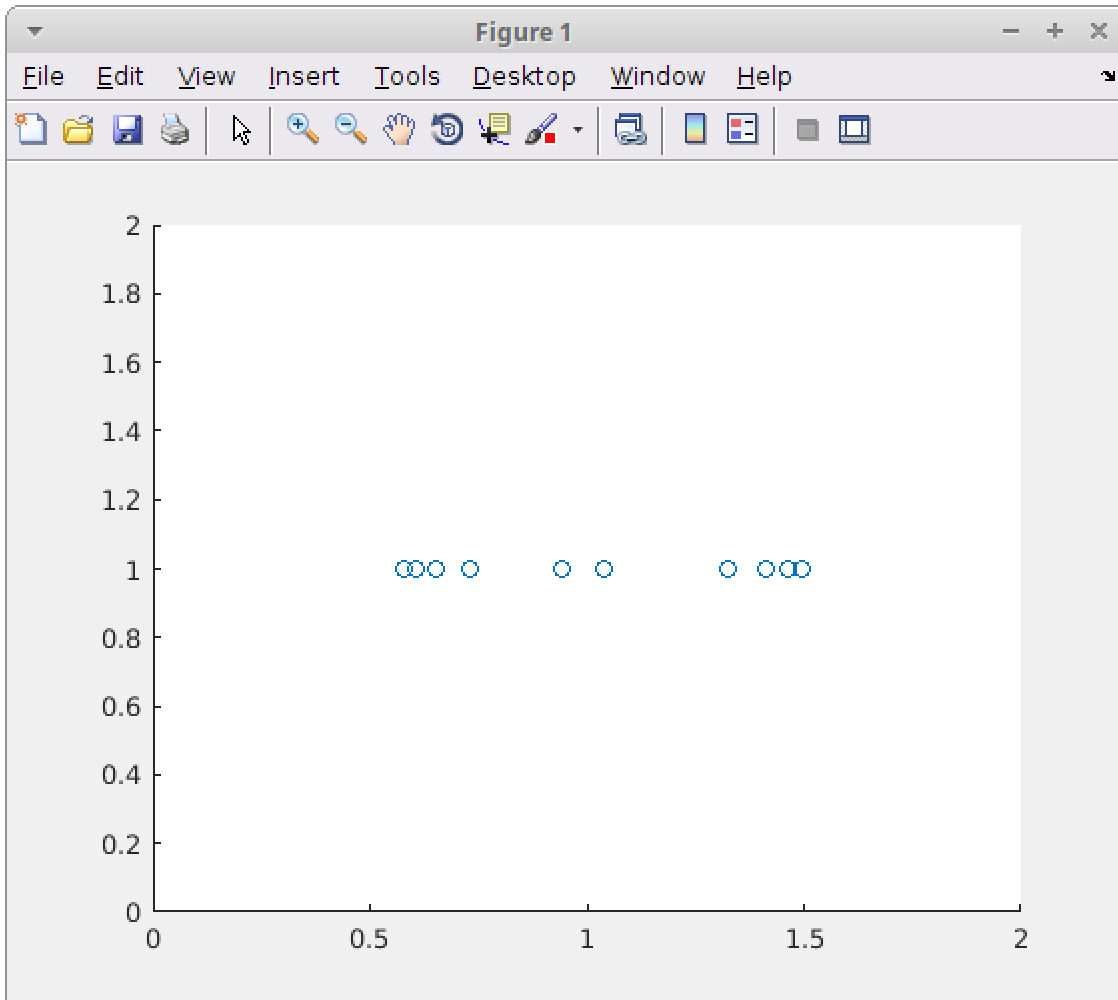
Scatter plot jitter

La fonction `scatter` a deux propriétés non documentées `'jitter'` et `'jitterAmount'` qui permettent de faire pivoter les données sur l'axe des x uniquement. Cela remonte à Matlab 7.1 (2005), et peut-être plus tôt.

Pour activer cette fonctionnalité, définissez la propriété `'jitter'` sur `'on'` et définissez la propriété `'jitterAmount'` sur la valeur absolue souhaitée (la valeur par défaut est `0.2`).

Ceci est très utile lorsque vous souhaitez visualiser des données qui se chevauchent, par exemple:

```
scatter(ones(1,10), ones(1,10), 'jitter', 'on', 'jitterAmount', 0.5);
```



En savoir plus sur Undocumented Matlab

Lire Fonctionnalités non documentées en ligne: https://riptutorial.com/fr/matlab/topic/2383/fonctionnalites-non-documentees

Chapitre 11: Fonctions de documentation

Remarques

- Le texte d'aide peut être situé avant ou après la ligne de `function` , tant qu'il n'y a pas de code entre la ligne de fonction et le début du texte d'aide.
- La mise en majuscule du nom de la fonction ne marque que le nom et n'est pas obligatoire.
- Si une ligne est précédée de `See also` , tous les noms de la ligne correspondant au nom d'une classe ou d'une fonction sur le chemin de recherche seront automatiquement liés à la documentation de cette classe / fonction.
  - Les fonctions globales peuvent être désignées ici en précédant leur nom par un `\` . Sinon, les noms vont d'abord essayer de résoudre les fonctions membres.
- Les hyperliens de la forme `<a href="matlab:web('url')">Name</a>` sont autorisés.

Examples

Documentation de fonction simple

```
function output = mymult(a, b)
% MYMULT Multiply two numbers.
%   output = MYMULT(a, b) multiplies a and b.
%
%   See also fft, foo, sin.
%
%   For more information, see <a href="matlab:web('https://google.com')">Google</a>.
    output = a * b;
end
```

`help mymult` fournit alors:

> **mymult** Multiplie deux nombres.
>
> output = **mymult** (a, b) multiplie a et b.
>
> Voir aussi, foo, péché.
>
> Pour plus d'informations, voir Google .

`fft` et `sin` automatiquement `fft` à leur texte d'aide respectif, et Google est un lien vers google.com . `foo` ne liera aucune documentation dans ce cas, tant qu'il n'y a pas de fonction / classe documentée par le nom de `foo` sur le chemin de recherche.

Documentation de fonction locale

Dans cet exemple, la documentation de la fonction locale `baz` (définie dans `foo.m`) est accessible soit par le lien résultant dans `help foo` , soit directement via `help foo>baz` .

```
function bar = foo
%This is documentation for FOO.
% See also foo>baz

% This wont be printed, because there is a line without % on it.
end

function baz
% This is documentation for BAZ.
end
```

Obtenir une signature de fonction

Il est souvent utile que MATLAB imprime la 1ère ligne d'une fonction, car celle-ci contient généralement la signature de la fonction, y compris les entrées et les sorties:

```
dbtype <functionName> 1
```

Exemple:

```
>> dbtype fit 1

1  function [fitobj,goodness,output,warnstr,errstr,convmsg] =
fit(xdatain,ydatain,fittypeobj,varargin)
```

Documenter une fonction avec un exemple de script

Pour documenter une fonction, il est souvent utile d'avoir un exemple de script qui utilise votre fonction. La fonction de publication dans Matlab peut ensuite être utilisée pour générer un fichier d'aide contenant des images, du code, des liens, etc. intégrés. La syntaxe de documentation de votre code peut être trouvée ici .

La fonction Cette fonction utilise une FFT "corrigée" dans Matlab.

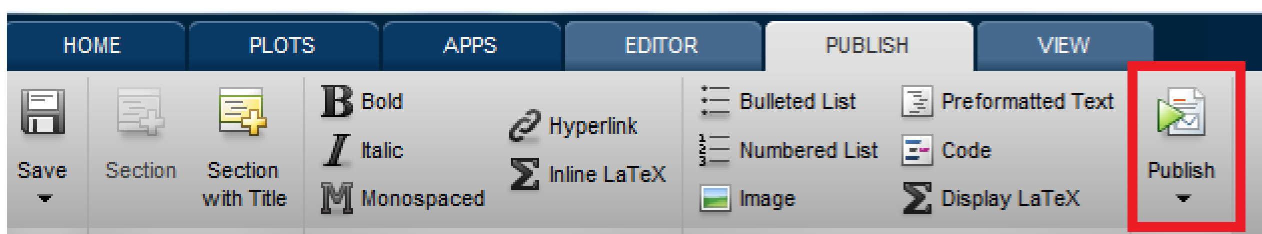
```
function out_sig = myfft(in_sig)

out_sig = fftshift(fft(ifftshift(in_sig)));

end
```

L'exemple de script Il s'agit d'un script distinct qui explique les entrées, les sorties et donne un exemple expliquant pourquoi la correction est nécessaire. Merci à Wu, Kan, l'auteur original de cette fonction.

```
%% myfft
% This function uses the "proper" fft in matlab. Note that the fft needs to
% be multiplied by dt to have physical significance.
% For a full description of why the FFT should be taken like this, refer
% to: Why_use_fftshift(fft(fftshift(x)))__in_Matlab.pdf included in the
% help folder of this code. Additional information can be found:
% <https://www.mathworks.com/matlabcentral/fileexchange/25473-why-use-fftshift-fft-fftshift-x-
---in-matlab-instead-of-fft-x-->
```

```
%
%% Inputs
% *in_sig* - 1D signal
%
%% Outputs
% *out_sig* - corrected FFT of *in_sig*
%
%% Examples
% Generate a signal with an analytical solution. The analytical solution is
% then compared to the fft then to myfft. This example is a modified
% version given by Wu, Kan given in the link aboce.
%%
% Set parameters
fs = 500;           %sampling frequency
dt = 1/fs;          %time step
T=1;                %total time window
t = -T/2:dt:T/2-dt; %time grids
df = 1/T;           %freq step
Fmax = 1/2/dt;      %freq window
f=-Fmax:df:Fmax-df; %freq grids, not used in our examples, could be used by plot(f, X)
%%
% Generate Gaussian curve
Bx = 10; A = sqrt(log(2))/(2*pi*Bx);    %Characteristics of Gaussian curve
x = exp(-t.^2/2/A^2);                   %Create Gaussian Curve
%%
% Generate Analytical solution
Xan = A*sqrt(2*pi)*exp(-2*pi^2*f.^2*A^2); %X(f), real part of the analytical Fourier transform
of x(t)

%%
% Take FFT and corrected FFT then compare
Xfft = dt *fftshift(fft(x));    %FFT
Xfinal = dt * myfft(x);         %Corrected FFT
hold on
plot(f,Xan);
plot(f,real(Xfft));
plot(f,real(Xfinal),'ro');
title('Comparison of Corrected and Uncorrected FFT');
legend('Analytical Solution','Uncorrected FFT','Corrected FFT');
xlabel('Frequency'); ylabel('Amplitude');
DT = max(f) - min(f);
xlim([-DT/4,DT/4]);
```

L' option de publication se trouve sous l'onglet "Publier", mis en évidence dans l'image Documentation de fonction simple ci-dessous.



Matlab exécutera le script et enregistrera les images affichées, ainsi que le texte généré par la ligne de commande. La sortie peut être enregistrée dans différents types de formats, y compris HTML, Latex et PDF.

La sortie du script d'exemple ci-dessus peut être vue dans l'image ci-dessous.

myfft

This function uses the "proper" fft in matlab. Note that the fft needs to be multiplied by dt to have physical significance. For a full description of why the FFT should be taken like this, refer to: Why_use_fftshift(fft(fftshift(x)))__in_Matlab.pdf included in the help folder of this code. Additional information can be found: https://www.mathworks.com/matlabcentral/fileexchange/25473-why-use-fftshift-fft-fftshift-x----in-matlab-instead-of-fft-x--

Contents

- Inputs
- Outputs
- Examples

Inputs

in_sig - 1D signal

Outputs

out_sig - corrected FFT of **in_sig**

Examples

Generate a signal with an analytical solution. The analytical solution is then compared to the fft then to myfft. This example is a modified version given by Wu, Kan given in the link aboce.

Set parameters

```
fs = 500;           %sampling frequency
dt = 1/fs;          %time step
T=1;                %total time window
t = -T/2:dt:T/2-dt; %time grids
df = 1/T;           %freq step
Fmax = 1/2/dt;      %freq window
f=-Fmax:df:Fmax-df; %freq grids, not used in our examples, could be used by plot(f, X)
```

Generate Gaussian curve

```
Bx = 10; A = sqrt(log(2))/(2*pi*Bx);    %Characteristics of Gaussian curve
x = exp(-t.^2/2/A^2);                   %Create Gaussian Curve
```

Generate Analytical solution

```
Xan = A*sqrt(2*pi)*exp(-2*pi^2*f.^2*A^2); %X(f), real part of the analytical Fourier transform of x(t)
```

Take FFT and corrected FFT then compare

```
Xfft = dt *fftshift(fft(x));    %FFT
Xfinal = dt * myfft(x);         %Corrected FFT
hold on
plot(f,Xan);
plot(f,real(Xfft));
plot(f,real(Xfinal),'ro');
title('Comparison of Corrected and Uncorrected FFT');
legend('Analytical Solution','Uncorrected FFT','Corrected FFT');
xlabel('Frequency'); ylabel('Amplitude');
DT = max(f) - min(f);
xlim([-DT/4,DT/4]);
```


Comparison of Corrected and Uncorrected FFT

0.04

https://riptutorial.com/fr/matlab/topic/1253/fonctions-de-documentation

Chapitre 12: Graphiques: Tracés de lignes 2D

Syntaxe

- parcelle (Y)
- plot (Y, LineSpec)
- tracé (X, Y)
- plot (X, Y, LineSpec)
- tracé (X1, Y1, X2, Y2, ..., Xn, Yn)
- tracé (X1, Y1, LineSpec1, X2, Y2, LineSpec2, ..., Xn, Yn, LineSpecn)
- plot (___, nom, valeur)
- h = tracé (___)

Paramètres

Paramètre	Détails
X	x-values
Y	valeurs y
LineSpec	Style de trait, symbole de marqueur et couleur, spécifié sous forme de chaîne
Nom, Valeur	Paires facultatives d'arguments de valeur de nom pour personnaliser les propriétés de ligne
h	manipuler à un objet graphique

Remarques

http://www.mathworks.com/help/matlab/ref/plot.html

Examples

Plusieurs lignes dans une seule parcelle

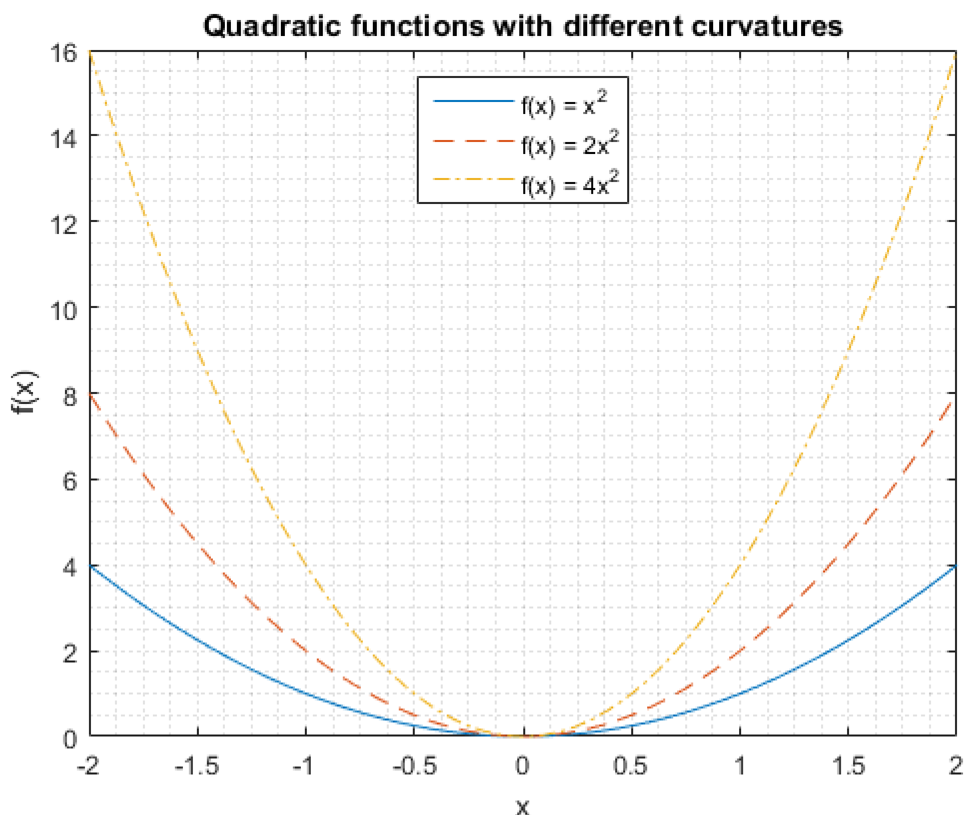
Dans cet exemple, nous allons tracer plusieurs lignes sur un seul axe. De plus, nous choisissons une apparence différente pour les lignes et créons une légende.

```
% create sample data
x = linspace(-2,2,100);                  % 100 linearly spaced points from -2 to 2
y1 = x.^2;
y2 = 2*x.^2;
y3 = 4*x.^2;

% create plot
figure;                                  % open new figure
plot(x,y1, x,y2,'--', x,y3,'-.');        % plot lines
grid minor;                              % add minor grid
title('Quadratic functions with different curvatures');
xlabel('x');
ylabel('f(x)');
legend('f(x) = x^2', 'f(x) = 2x^2', 'f(x) = 4x^2', 'Location','North');
```

Dans l'exemple ci-dessus, nous avons tracé les lignes avec une seule commande `plot` . Une alternative consiste à utiliser des commandes séparées pour chaque ligne. Nous devons *tenir* le contenu d'une figure en `hold on` la dernière avant d'ajouter la deuxième ligne. Sinon, les lignes précédemment tracées disparaîtront de la figure. Pour créer le même tracé que ci-dessus, nous pouvons utiliser les commandes suivantes:

```
figure; hold on;
plot(x,y1);
plot(x,y2,'--');
plot(x,y3,'-.');
```

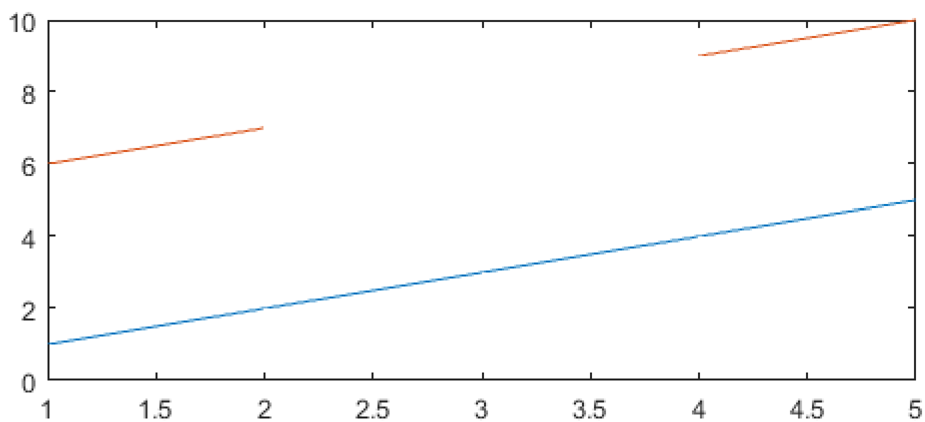
La figure résultante ressemble à ceci dans les deux cas:



Split line avec NaN

Entrelacez vos valeurs `y` ou `x` avec `NaNs`

```
x = [1:5; 6:10]';

x(3,2) = NaN
x =
     1     6
     2     7
     3   NaN
     4     9
     5    10

plot(x)
```

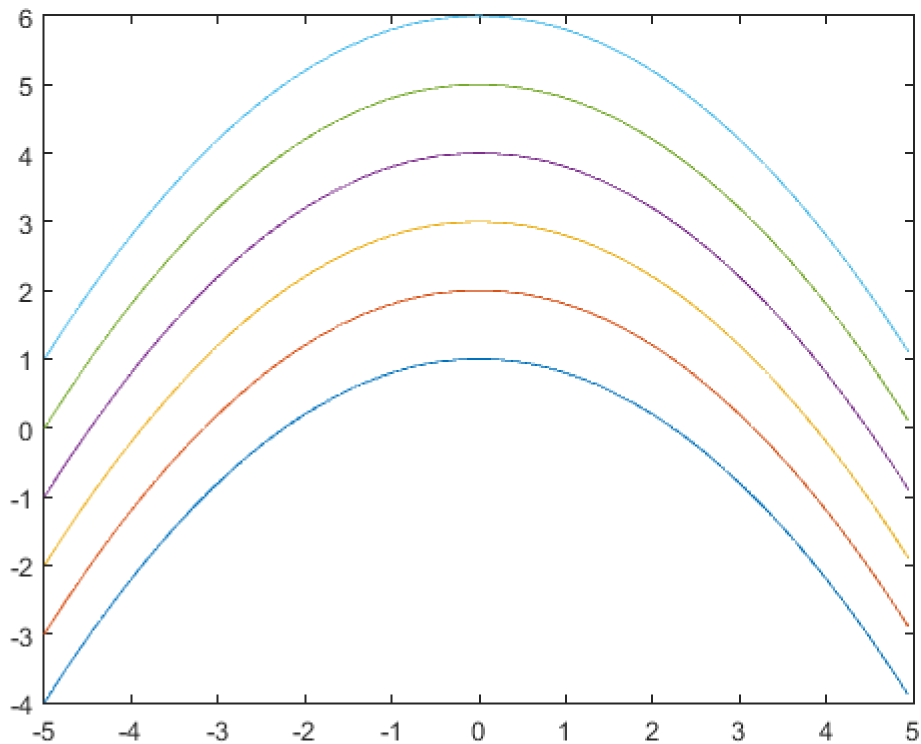


Ordre personnalisé de couleurs et de lignes

Dans MATLAB, nous pouvons définir de nouveaux ordres personnalisés *par défaut* , tels qu'un ordre de couleur et un ordre de style de ligne. Cela signifie que les nouvelles commandes seront appliquées à tout chiffre créé après l'application de ces paramètres. Les nouveaux paramètres restent jusqu'à la fermeture de la session MATLAB ou de nouveaux paramètres.

Ordre des couleurs et des lignes par défaut

Par défaut, MATLAB utilise deux couleurs différentes et uniquement un style de trait plein. Par conséquent, si le `plot` est appelé pour dessiner plusieurs lignes, MATLAB alterne à travers un ordre de couleurs pour dessiner des lignes de couleurs différentes.



Nous pouvons obtenir l'ordre des couleurs par défaut en appelant `get` avec un handle global `0` suivi de cet attribut `DefaultAxesColorOrder` :

```
>> get(0, 'DefaultAxesColorOrder')
ans =
         0    0.4470    0.7410
    0.8500    0.3250    0.0980
    0.9290    0.6940    0.1250
    0.4940    0.1840    0.5560
    0.4660    0.6740    0.1880
    0.3010    0.7450    0.9330
    0.6350    0.0780    0.1840
```

Ordre de couleur et de style de trait personnalisé

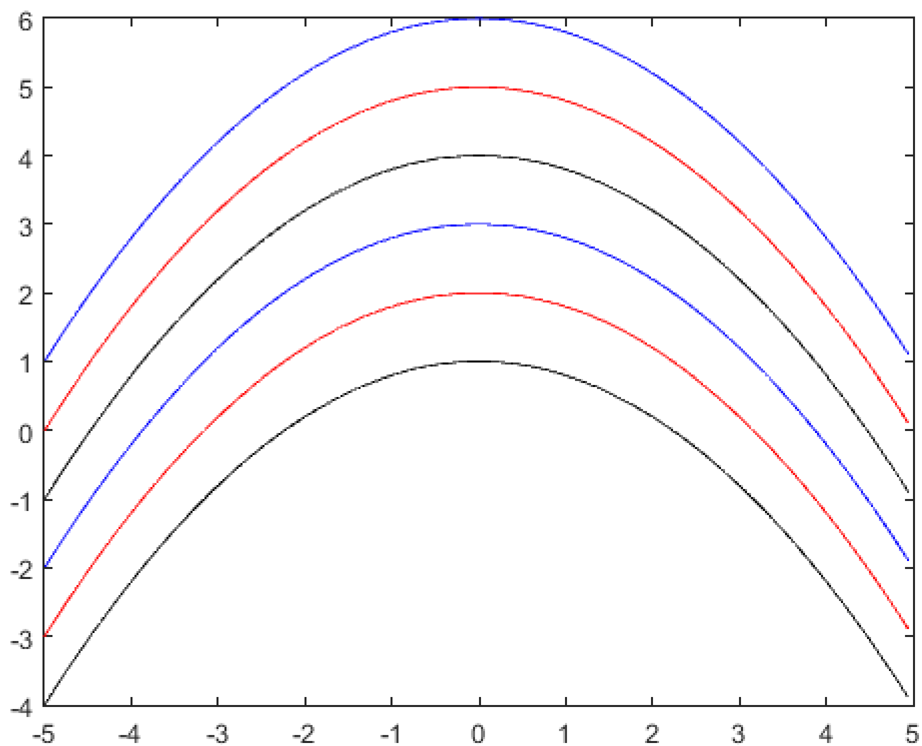
Une fois que nous avons décidé de définir un ordre de couleurs et un style de ligne personnalisés, MATLAB doit alterner les deux. Le premier changement appliqué par MATLAB est une couleur. Lorsque toutes les couleurs sont épuisées, MATLAB applique le style de ligne suivant dans un ordre de style de ligne défini et définit un index de couleur sur 1. Cela signifie que MATLAB recommencera à alterner toutes les couleurs, mais en utilisant le style de ligne suivant dans son ordre. Lorsque tous les styles de lignes et les couleurs sont épuisés, il est évident que MATLAB commence à utiliser le premier style et la première couleur.

Pour cet exemple, j'ai défini un vecteur d'entrée et une fonction anonyme pour faciliter le traçage des figures:

```
F = @(a,x) bsxfun(@plus, -0.2*x(:).^2, a);
x = (-5:5/100:5-5/100)';
```

Pour définir une nouvelle couleur ou un nouveau style de ligne, nous appelons `set` fonction `set` avec un handle global `0` suivi d'un attribut `DefaultAxesXXXXXXX` ; `XXXXXXX` peut être `ColorOrder` ou `LineStyleOrder` . La commande suivante définit un nouvel ordre de couleur en noir, rouge et bleu, respectivement:

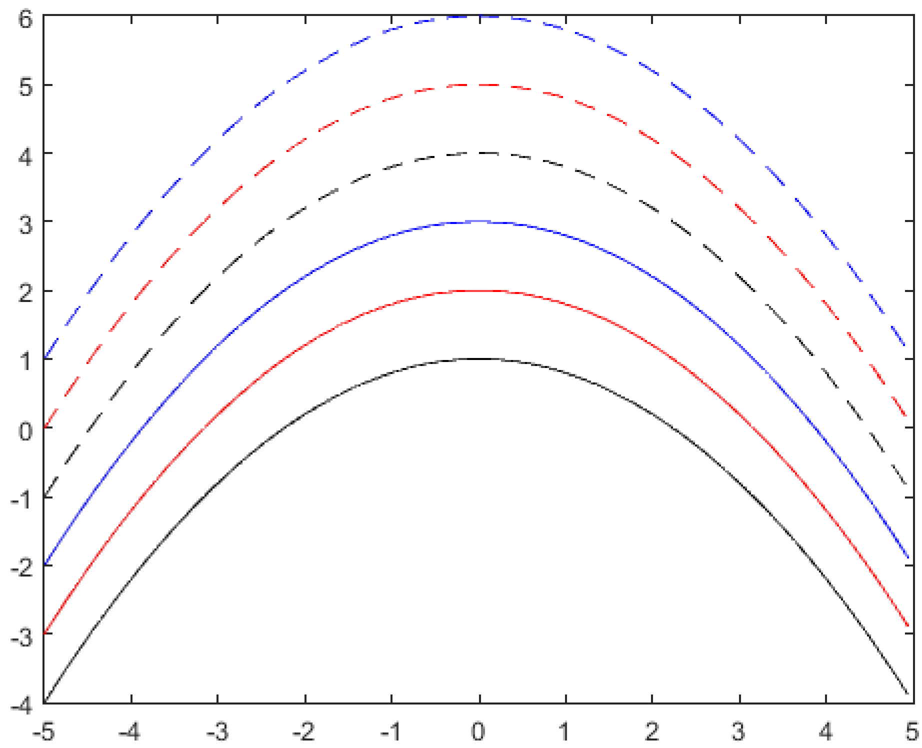
```
set(0, 'DefaultAxesColorOrder', [0 0 0; 1 0 0; 0 0 1]);
plot(x, F([1 2 3 4 5 6],x));
```



Comme vous pouvez le voir, MATLAB alterne uniquement avec les couleurs car l'ordre des lignes est défini par défaut sur une ligne continue. Lorsqu'un jeu de couleurs est épuisé, MATLAB démarre à partir de la première couleur dans l'ordre des couleurs.

Les commandes suivantes définissent les ordres de couleur et de style de ligne:

```
set(0, 'DefaultAxesColorOrder', [0 0 0; 1 0 0; 0 0 1]);
set(0, 'DefaultAxesLineStyleOrder', {'-' '--'});
plot(x, F([1 2 3 4 5 6],x));
```



Maintenant, MATLAB alterne à travers différentes couleurs et différents styles de lignes en utilisant la couleur comme attribut le plus fréquent.

Lire Graphiques: Tracés de lignes 2D en ligne: https://riptutorial.com/fr/matlab/topic/426/graphiques--traces-de-lignes-2d

Chapitre 13: Graphiques: Transformations 2D et 3D

Examples

Transformations 2D

Dans cet exemple, nous allons prendre une ligne en forme de sqaure tracée en `line` et effectuer des transformations sur celle-ci. Ensuite, nous allons utiliser les mêmes transformations mais dans un ordre différent et voir comment cela influence les résultats.

Nous ouvrons d'abord une figure et définissons des paramètres initiaux (coordonnées de points carrés et paramètres de transformation)

```
%Open figure and create axis
Figureh=figure('NumberTitle','off','Name','Transformation Example',...
    'Position',[200 200 700 700]); %bg is set to red so we know that we can only see the axes
Axesh=axes('XLim',[-8 8],'YLim',[-8,8]);

 %Initializing Variables
    square=[-0.5 -0.5;-0.5 0.5;0.5 0.5;0.5 -0.5]; %represented by its vertices
    Sx=0.5;
    Sy=2;
    Tx=2;
    Ty=2;
    teta=pi/4;
```

Nous construisons ensuite les matrices de transformation (échelle, rotation et traduction):

```
%Generate Transformation Matrix
S=makehgtform('scale',[Sx Sy 1]);
R=makehgtform('zrotate',teta);
T=makehgtform('translate',[Tx Ty 0]);
```

Ensuite, nous traçons le suare bleu:

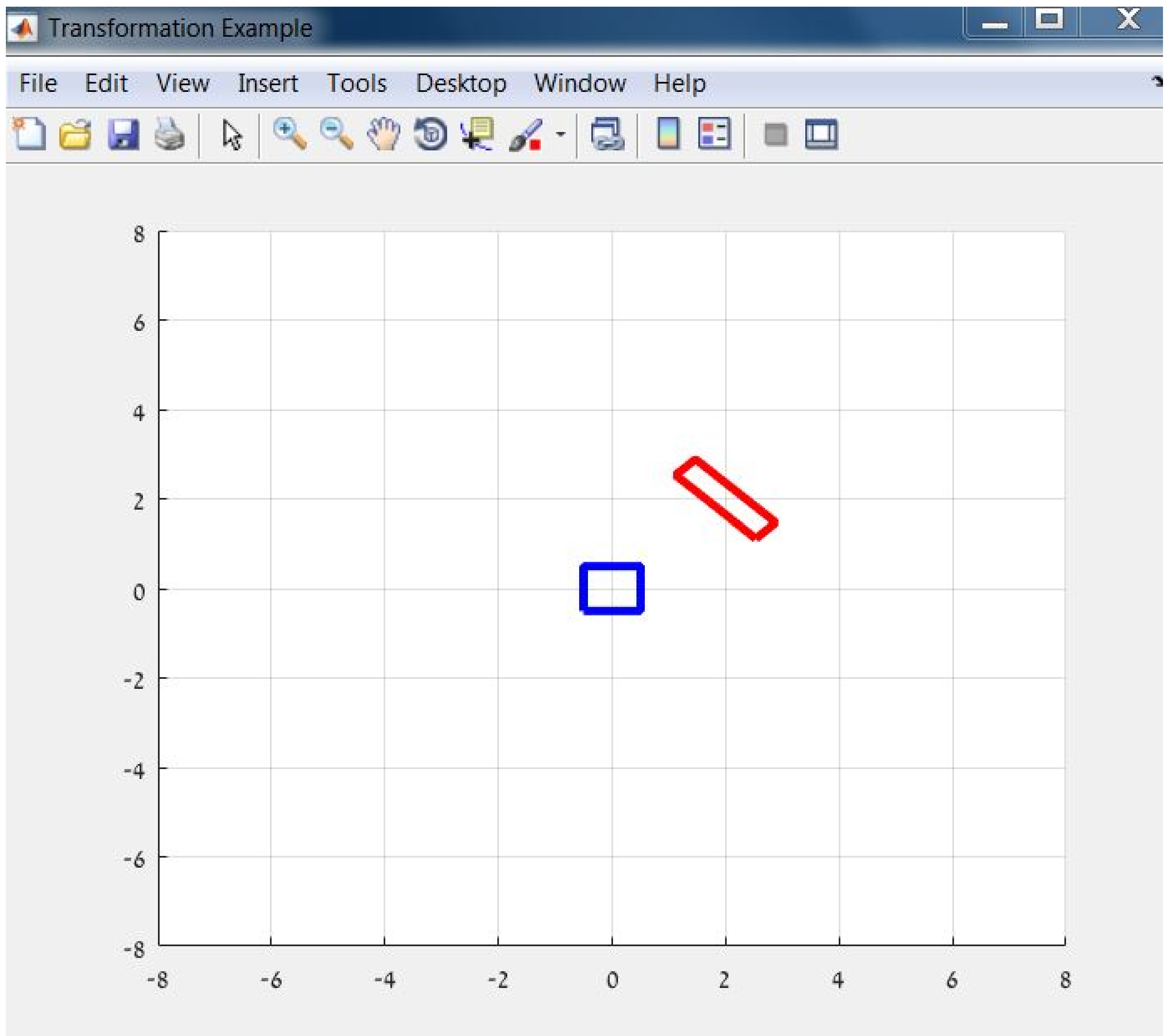
```
%% Plotting the original Blue Square
OriginalSQ=line([square(:,1);square(1,1)],[square(:,2);square(1,2)],'Color','b','LineWidth',3);

grid on;    % Applying grid on the figure
hold all;   % Holding all Following  graphs to current axes
```

Ensuite, nous allons le tracer à nouveau dans une couleur différente (rouge) et appliquer les transformations:

```
%% Plotting the Red Square
%Calculate rectangle vertices
HrectTRS=T*R*S;
RedSQ=line([square(:,1);square(1,1)],[square(:,2);square(1,2)],'Color','r','LineWidth',3);
```

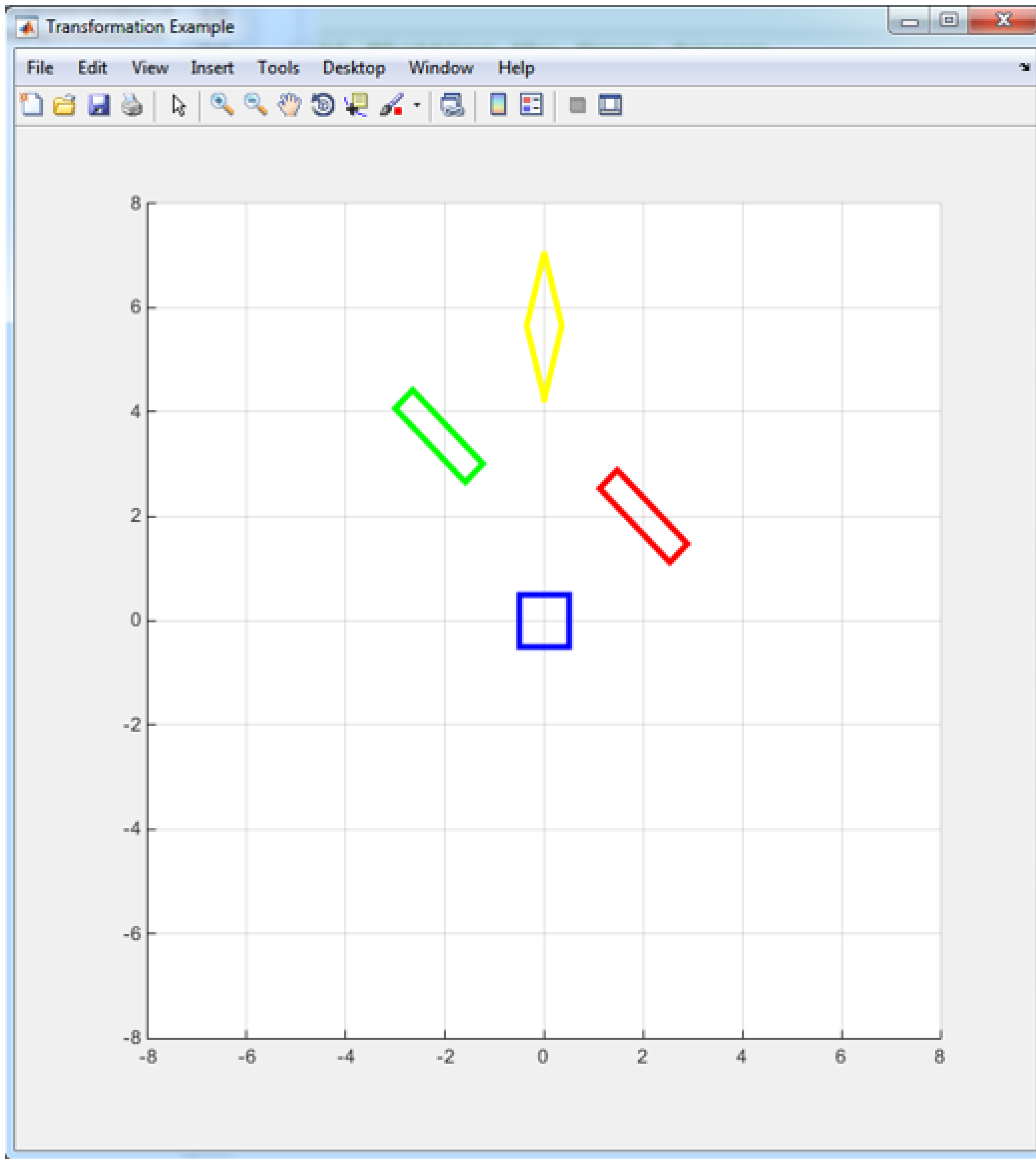
```
%transformation of the axes
AxesTransformation=hgtransform('Parent',gca,'matrix',HrectTRS);
%seting the line to be a child of transformed axes
set(RedSQ,'Parent',AxesTransformation);
```

Le résultat devrait ressembler à ceci:



Voyons maintenant ce qui se passe quand on change l'ordre de transformation:

```
%% Plotting the Green Square
HrectRST=R*S*T;
GreenSQ=line([square(:,1);square(1,1)],[square(:,2);square(1,2)],'Color','g','LineWidth',3);
AxesTransformation=hgtransform('Parent',gca,'matrix',HrectRST);
set(GreenSQ,'Parent',AxesTransformation);

%% Plotting the Yellow Square
HrectSRT=S*R*T;
YellowSQ=line([square(:,1);square(1,1)],[square(:,2);square(1,2)],'Color','y','LineWidth',3);
AxesTransformation=hgtransform('Parent',gca,'matrix',HrectSRT);
set(YellowSQ,'Parent',AxesTransformation);
```



Lire Graphiques: Transformations 2D et 3D en ligne: https://riptutorial.com/fr/matlab/topic/7418/graphiques--transformations-2d-et-3d

Chapitre 14: Initialisation de matrices ou de tableaux

Introduction

Matlab a trois fonctions importantes pour créer des matrices et définir leurs éléments sur des zéros, des uns ou la matrice d'identité. (La matrice d'identité en a une sur la diagonale principale et des zéros ailleurs.)

Syntaxe

- Z = zéros (sz, type de données, type de tableau)
- X = ceux (sz, type de données)
- I = oeil (sz, type de données)

Paramètres

Paramètre	Détails
sz	n (pour une matrice nxn)
sz	n, m (pour une matrice nxm)
sz	m, n, ..., k (pour une matrice m par n-by -...- by-k)
Type de données	'double' (par défaut), 'single', 'int8', 'uint8', 'int16', 'uint16', 'int32', 'uint32', 'int64' ou 'uint64'
type de tableau	'distribué'
type de tableau	«codistribué»
type de tableau	'gpuArray'

Remarques

Ces fonctions créeront une matrice de doubles, par défaut.

Examples

Créer une matrice de 0

```
z1 = zeros(5); % Create a 5-by-5 matrix of zeroes
z2 = zeros(2,3); % Create a 2-by-3 matrix
```

Créer une matrice de 1

```
o1 = ones(5); % Create a 5-by-5 matrix of ones
o2 = ones(1,3); % Create a 1-by-3 matrix / vector of size 3
```

Créer une matrice d'identité

```
i1 = eye(3); % Create a 3-by-3 identity matrix
i2 = eye(5,6); % Create a 5-by-6 identity matrix
```

Lire Initialisation de matrices ou de tableaux en ligne:
https://riptutorial.com/fr/matlab/topic/8049/initialisation-de-matrices-ou-de-tableaux

Chapitre 15: Interfaces utilisateur MATLAB

Examples

Passage de données autour de l'interface utilisateur

La plupart des interfaces utilisateur avancées exigent que l'utilisateur puisse transmettre des informations entre les différentes fonctions constituant une interface utilisateur. MATLAB dispose de plusieurs méthodes différentes pour le faire.

`guidata`

L' environnement de développement GUI (GUIDE) de MATLAB préfère utiliser une `struct` nommée `handles` pour transmettre des données entre les rappels. Cette `struct` contient tous les descripteurs graphiques des différents composants de l'interface utilisateur, ainsi que des données spécifiées par l'utilisateur. Si vous n'utilisez pas un rappel créé GUIDE-qui passe automatiquement `handles` , vous pouvez récupérer la valeur actuelle à l' aide `guidata`

```
% hObject is a graphics handle to any UI component in your GUI
handles = guidata(hObject);
```

Si vous souhaitez modifier une valeur stockée dans cette structure de données, vous pouvez la modifier, mais vous devez la stocker dans `hObject` pour que les modifications soient visibles par d'autres rappels. Vous pouvez le stocker en spécifiant un second argument d'entrée à `guidata` .

```
% Update the value
handles.myValue = 2;

% Save changes
guidata(hObject, handles)
```

La valeur de `hObject` n'a pas d'importance tant qu'il s'agit d'un composant d'interface utilisateur *dans la même* `figure` car les données sont finalement stockées dans la figure contenant `hObject` .

Meilleur pour:

- Stockage de la structure des `handles` dans laquelle vous pouvez stocker toutes les poignées de vos composants d'interface graphique.
- Stocker les "petites" autres variables auxquelles la plupart des callbacks doivent accéder.

Non recommandé pour :

- Stockage de variables volumineuses auxquelles tous les callbacks et sous-fonctions ne doivent pas accéder (utilisez `setappdata` / `getappdata` pour ceux-ci).

`setappdata` / `getappdata`

À l' `guidata` approche `guidata` , vous pouvez utiliser `setappdata` et `getappdata` pour stocker et récupérer des valeurs dans un handle graphique. L'avantage de l'utilisation de ces méthodes est que vous ne pouvez récupérer *que la valeur souhaitée* plutôt qu'une `struct` entière contenant *toutes* les données stockées. Il est similaire à un magasin de clé / valeur.

Pour stocker des données dans un objet graphique

```
% Create some data you would like to store
myvalue = 2

% Store it using the key 'mykey'
setappdata(hObject, 'mykey', myvalue)
```

Et pour récupérer cette même valeur depuis un rappel différent

```
value = getappdata(hObject, 'mykey');
```

Note: Si aucune valeur n'a été stockée avant d'appeler `getappdata` , cela retournera un tableau vide (`[]`).

Semblable à `guidata` , les données sont stockées dans la figure contenant `hObject` .

Meilleur pour:

- Stockage de variables volumineuses auxquelles tous les callbacks et sous-fonctions ne doivent pas accéder.

`UserData`

Chaque descripteur graphique a une propriété spéciale, `UserData` qui peut contenir toutes les données souhaitées. Il pourrait contenir un tableau de cellules, une `struct` ou même un scalaire. Vous pouvez tirer parti de cette propriété et stocker toutes les données que vous souhaitez associer à une poignée graphique donnée dans ce champ. Vous pouvez enregistrer et récupérer la valeur à l'aide des méthodes `get` / `set` standard pour les objets graphiques ou la notation par points si vous utilisez R2014b ou une version plus récente.

```
% Create some data to store
mydata = {1, 2, 3};

% Store it within the UserData property
set(hObject, 'UserData', mydata)

% Of if you're using R2014b or newer:
% hObject.UserData = mydata;
```

Ensuite, à partir d'un autre rappel, vous pouvez récupérer ces données:

```
their_data = get(hObject, 'UserData');

% Or if you're using R2014b or newer:
% their_data = hObject.UserData;
```

Meilleur pour:

- Stockage de variables avec une portée limitée (variables susceptibles d'être utilisées uniquement par l'objet dans lequel elles sont stockées, ou objets ayant une relation directe avec celui-ci).

Fonctions imbriquées

Dans MATLAB, une fonction imbriquée peut lire et modifier toute variable définie dans la fonction parente. De cette façon, si vous spécifiez un rappel comme une fonction imbriquée, il peut récupérer et modifier toutes les données stockées dans la fonction principale.

```
function mygui()
    hButton = uicontrol('String', 'Click Me', 'Callback', @callback);

    % Create a counter to keep track of the number of times the button is clicked
    nClicks = 0;

    % Callback function is nested and can therefore read and modify nClicks
    function callback(source, event)
        % Increment the number of clicks
        nClicks = nClicks + 1;

        % Print the number of clicks so far
        fprintf('Number of clicks: %d\n', nClicks);
    end
end
```

Meilleur pour:

- Petites interfaces graphiques simples. (pour un prototypage rapide, ne pas avoir à implémenter les `guidata` et / ou `set/getappdata`).

Non recommandé pour :

- Interfaces graphiques moyennes, grandes ou complexes.
- Interface graphique créée avec `GUIDE` .

Arguments de saisie explicites

Si vous devez envoyer des données à une fonction de rappel sans avoir à modifier les données dans le rappel, vous pouvez toujours envisager de transmettre les données au rappel en utilisant

une définition de rappel soigneusement conçue.

Vous pouvez utiliser une fonction anonyme qui ajoute des entrées

```
% Create some data to send to mycallback
data = [1, 2, 3];

% Pass data as a third input to mycallback
set(hObject, 'Callback', @(source, event)mycallback(source, event, data))
```

Vous pouvez également utiliser la syntaxe du tableau de cellules pour spécifier un rappel, en spécifiant à nouveau des entrées supplémentaires.

```
set(hObject, 'Callback', {@mycallback, data})
```

Meilleur pour:

- Lorsque le rappel a besoin de `data` pour effectuer certaines opérations, la variable de `data` n'a pas besoin d'être modifiée et enregistrée dans un nouvel état.

Faire un bouton dans votre interface utilisateur qui interrompt l'exécution du rappel

Parfois, nous aimerions suspendre l'exécution du code pour inspecter l'état de l'application (voir Débogage). Lorsque vous exécutez du code via l'éditeur MATLAB, vous pouvez utiliser le bouton "Pause" de l'interface utilisateur ou en appuyant sur `Ctrl` + `c` (sous Windows). Cependant, lorsqu'un calcul a été lancé à partir d'une interface graphique (via le rappel de certains `uicontrol`), cette méthode ne fonctionne plus, et le rappel doit être *interrompu* via un autre rappel. Voici une démonstration de ce principe:

```
function interruptibleUI
dbclear in interruptibleUI % reset breakpoints in this file
figure('Position',[400,500,329,160]);

uicontrol('Style', 'pushbutton',...
          'String', 'Compute',...
          'Position', [24 55 131 63],...
          'Callback', @longComputation,...
          'Interruptible','on'); % 'on' by default anyway

uicontrol('Style', 'pushbutton',...
          'String', 'Pause #1',...
          'Position', [180 87 131 63],...
          'Callback', @interrupt1);

uicontrol('Style', 'pushbutton',...
          'String', 'Pause #2',...
          'Position', [180 12 131 63],...
          'Callback', @interrupt2);

end
```

```
function longComputation(src,event)
  superSecretVar = rand(1);
  pause(15);
  print('done!'); % we'll use this to determine if code kept running "in the background".
end

function interrupt1(src,event) % depending on where you want to stop
  dbstop in interruptibleUI at 27 % will stop after print('done!');
  dbstop in interruptibleUI at 32 % will stop after **this** line.
end

function interrupt2(src,event) % method 2
  keyboard;
  dbup; % this will need to be executed manually once the code stops on the previous line.
end
```

Pour vous assurer que vous comprenez cet exemple, procédez comme suit:

1. Collez le code ci-dessus dans un nouveau fichier appelé et enregistrez-le en tant que `interruptibleUI.m` , de sorte que le code démarre sur la **toute première ligne** du fichier (cela est important pour que la 1ère méthode fonctionne).
2. Exécutez le script.
3. Cliquez sur `Calculer` et, peu après, cliquez sur `Pause # 1` ou sur `Pause # 2` .
4. Assurez-vous de pouvoir trouver la valeur de `superSecretVar` .

Passer des données en utilisant la structure "handles"

Voici un exemple d'interface graphique de base avec deux boutons qui modifient une valeur stockée dans la structure des `handles` l'interface graphique.

```
function gui_passing_data()
    % A basic GUI with two buttons to show a simple use of the 'handles'
    % structure in GUI building

    %  Create a new figure.
    f = figure();

    % Retrieve the handles structure
    handles = guidata(f);

    % Store the figure handle
    handles.figure = f;

    % Create an edit box and two buttons (plus and minus),
    % and store their handles for future use
    handles.hedit  = uicontrol('Style','edit','Position',[10,200,60,20] , 'Enable',
'Inactive');

    handles.hbutton_plus  = uicontrol('Style','pushbutton','String','+',...
           'Position',[80,200,60,20] , 'Callback' , @ButtonPress);

    handles.hbutton_minus  = uicontrol('Style','pushbutton','String','-',...
           'Position',[150,200,60,20] , 'Callback' , @ButtonPress);

    % Define an initial value, store it in the handles structure and show
    % it in the Edit box
```

```
    handles.value = 1;
    set(handles.hedit , 'String' , num2str(handles.value))

    % Store handles
    guidata(f, handles);



function  ButtonPress(hObject, eventdata)
    % A button was pressed
    % Retrieve the handles
    handles = guidata(hObject);

    % Determine which button was pressed; hObject is the calling object
    switch(get(hObject , 'String'))
        case '+'
            % Add 1 to the value
            handles.value = handles.value + 1;
            set(handles.hedit , 'String', num2str(handles.value))
        case '-'
            % Substract 1 from the value
            handles.value = handles.value - 1;
    end

    % Display the new value
    set(handles.hedit , 'String', num2str(handles.value))

    % Store handles
    guidata(hObject, handles);
```

Pour tester l'exemple, enregistrez-le dans un fichier nommé `gui_passing_data.m` et lancez-le avec F5. Veuillez noter que dans un cas aussi simple, vous n'avez même pas besoin de stocker la valeur dans la structure des descripteurs car vous pouvez y accéder directement à partir de la propriété `String` de la zone d'édition.

Problèmes de performances lors du transfert de données autour de l'interface utilisateur

Deux techniques principales permettent de transmettre des données entre les fonctions d'interface graphique et les rappels: setappdata / getappdata et guidata (pour en savoir plus). Le premier devrait être utilisé pour les plus grandes variables car il est plus efficace dans le temps. L'exemple suivant teste l'efficacité des deux méthodes.

Une interface graphique avec un simple bouton est créée et une grande variable (10000x10000 double) est stockée à la fois avec guidata et avec setappdata. Le bouton recharge et stocke la variable en utilisant les deux méthodes tout en chronométrant leur exécution. Le temps d'exécution et le pourcentage d'amélioration à l'aide de setappdata sont affichés dans la fenêtre de commande.

```
function gui_passing_data_performance()
    % A basic GUI with a button to show performance difference between
    % guidata and setappdata

    %  Create a new figure.
    f = figure('Units' , 'normalized');
```

```
    % Retrieve the handles structure
    handles = guidata(f);

    % Store the figure handle
    handles.figure = f;

    handles.hbutton =
uicontrol('Style','pushbutton','String','Calculate','units','normalized',...
               'Position',[0.4 , 0.45 , 0.2 , 0.1] , 'Callback' , @ButtonPress);

    % Create an uninteresting large array
    data = zeros(10000);

    % Store it in appdata
    setappdata(handles.figure , 'data' , data);

    % Store it in handles
    handles.data = data;

    % Save handles
    guidata(f, handles);



function  ButtonPress(hObject, eventdata)

    % Calculate the time difference when using guidata and appdata
    t_handles = timeit(@use_handles);
    t_appdata = timeit(@use_appdata);

    % Absolute and percentage difference
    t_diff = t_handles - t_appdata;
    t_perc = round(t_diff / t_handles * 100);

    disp(['Difference: ' num2str(t_diff) ' ms / ' num2str(t_perc) ' %'])




function  use_appdata()

    % Retrieve the data from appdata
    data = getappdata(gcf , 'data');

    % Do something with data %

    % Store the value again
    setappdata(gcf , 'data' , data);


function use_handles()

    % Retrieve the data from handles
    handles = guidata(gcf);
    data = handles.data;

    % Do something with data %

    % Store it back in the handles
    handles.data = data;
```

```
    guidata(gcf, handles);
```

Sur mon Xeon W3530@2.80 GHz, j'obtiens une `Difference: 0.00018957 ms / 73 %` , en utilisant getappdata / setappdata, j'obtiens une amélioration des performances de 73%! Notez que le résultat ne change pas si une double variable 10x10 est utilisée. Cependant, le résultat changera si `handles` contient de nombreux champs avec de grandes données.

Lire Interfaces utilisateur MATLAB en ligne: https://riptutorial.com/fr/matlab/topic/2883/interfaces-utilisateur-matlab

Chapitre 16: Interpolation avec MATLAB

Syntaxe

1. zy = interp1 (x, y);
2. zy = interp1 (x, y, 'méthode');
3. zy = interp1 (x, y, 'méthode', 'extrapolation');
4. zy = interp1 (x, y, zx);
5. zy = interp1 (x, y, zx, 'méthode');
6. zy = interp1 (x, y, zx, 'méthode', 'extrapolation');

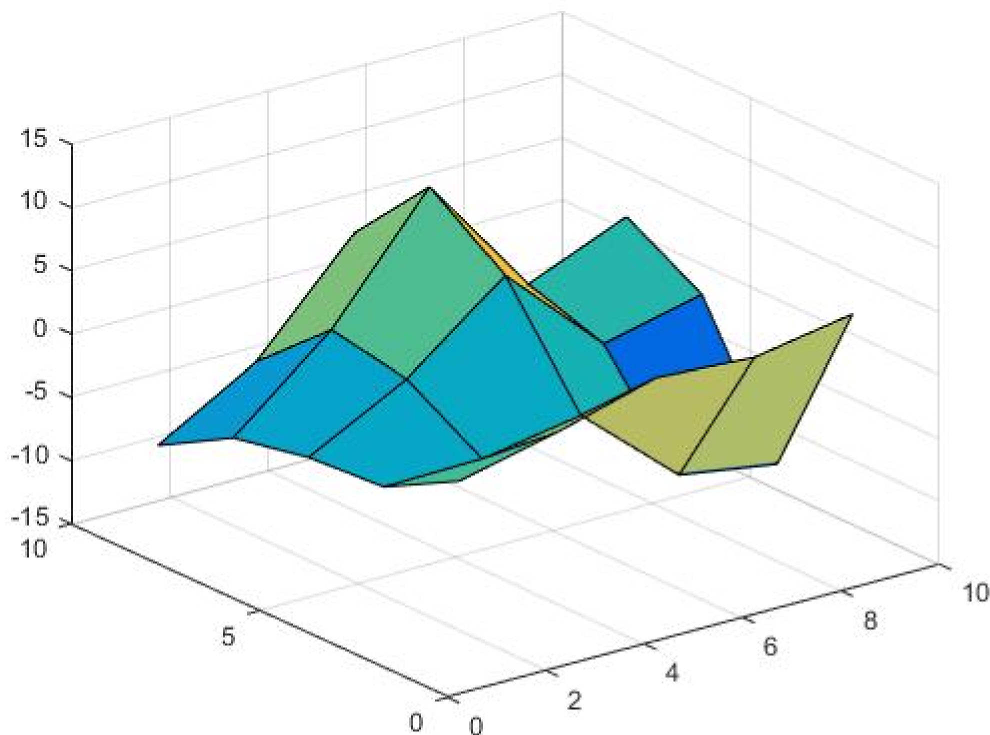
Examples

Interpolation par morceaux en 2 dimensions

Nous initialisons les données:

```
[X,Y] = meshgrid(1:2:10);
Z = X.*cos(Y) - Y.*sin(X);
```

La surface ressemble à la suivante.

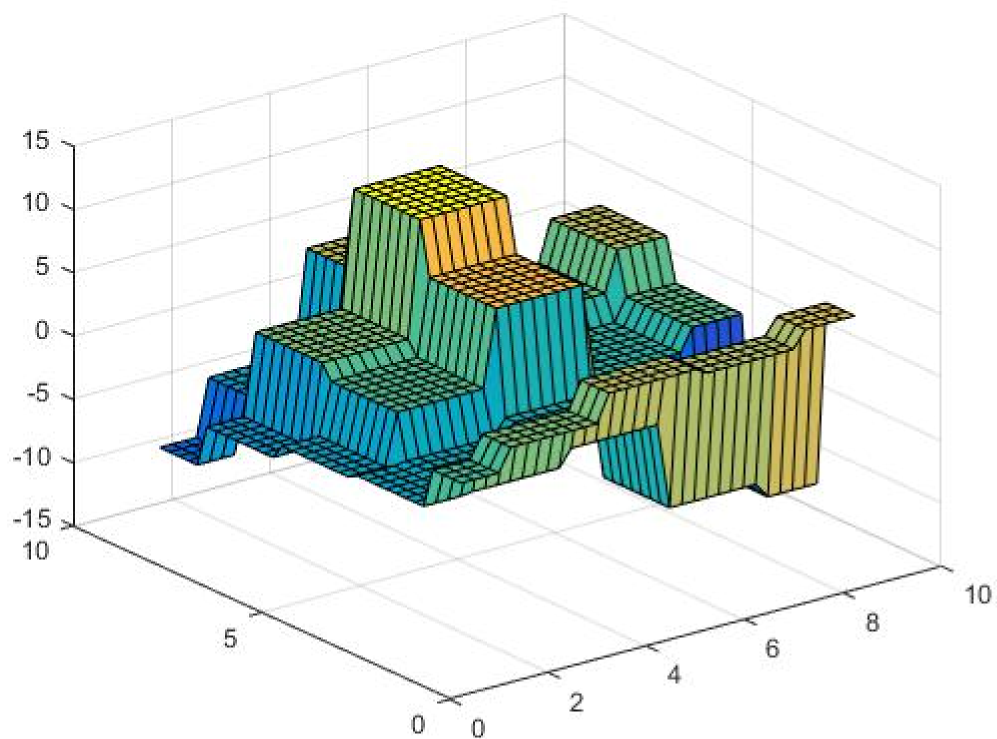


Maintenant, nous définissons les points où nous voulons interpoler:

```
[Vx,Vy] = meshgrid(1:0.25:10);
```

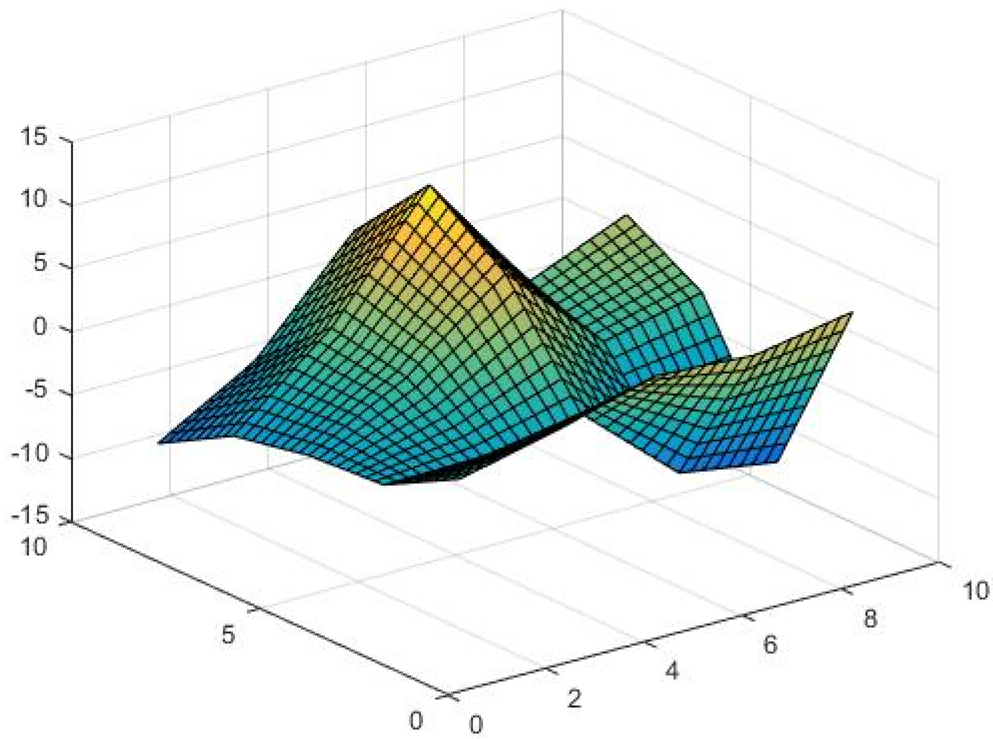
Nous pouvons maintenant effectuer l'interpolation la plus proche,

```
Vz = interp2(X,Y,Z,Vx,Vy,'nearest');
```



interpolation linéaire,

```
Vz = interp2(X,Y,Z,Vx,Vy,'linear');
```

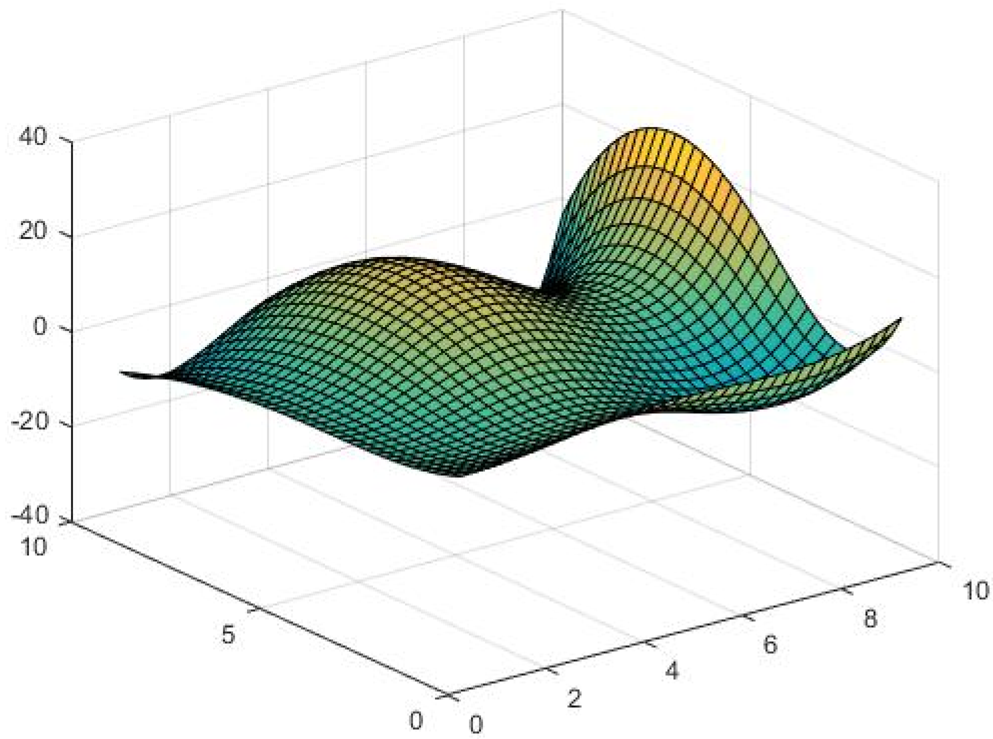


interpolation cubique

```
Vz = interp2(X,Y,Z,Vx,Vy,'cubic');
```

ou interpolation spline:

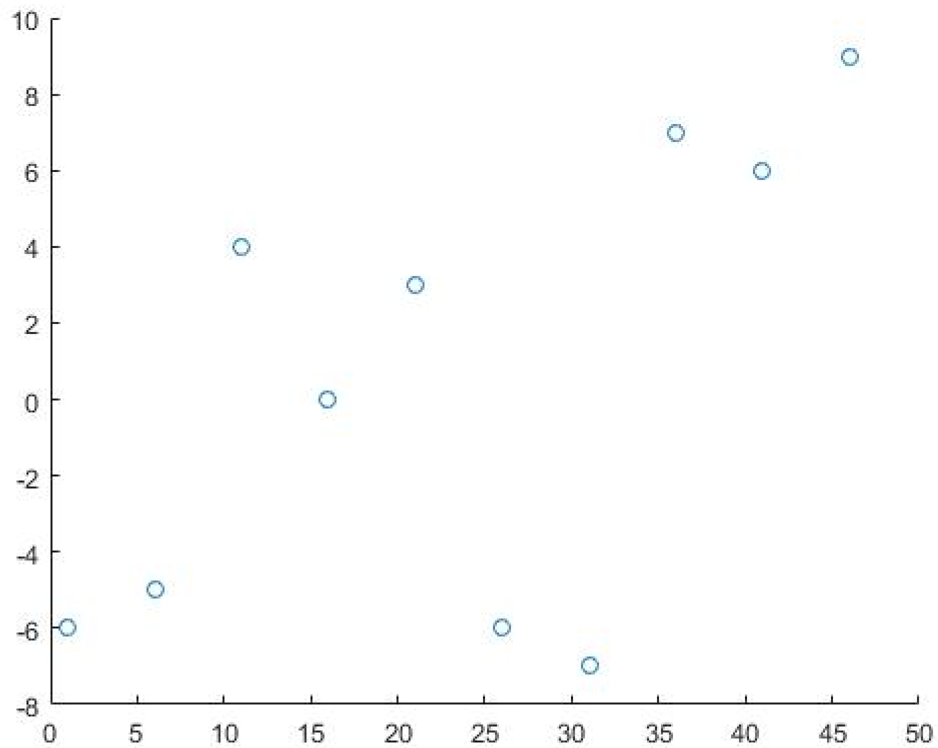
```
Vz = interp2(X,Y,Z,Vx,Vy,'spline');
```



Interpolation par morceaux 1 dimension

Nous utiliserons les données suivantes:

```
x = 1:5:50;
y = randi([-10 10],1,10);
```

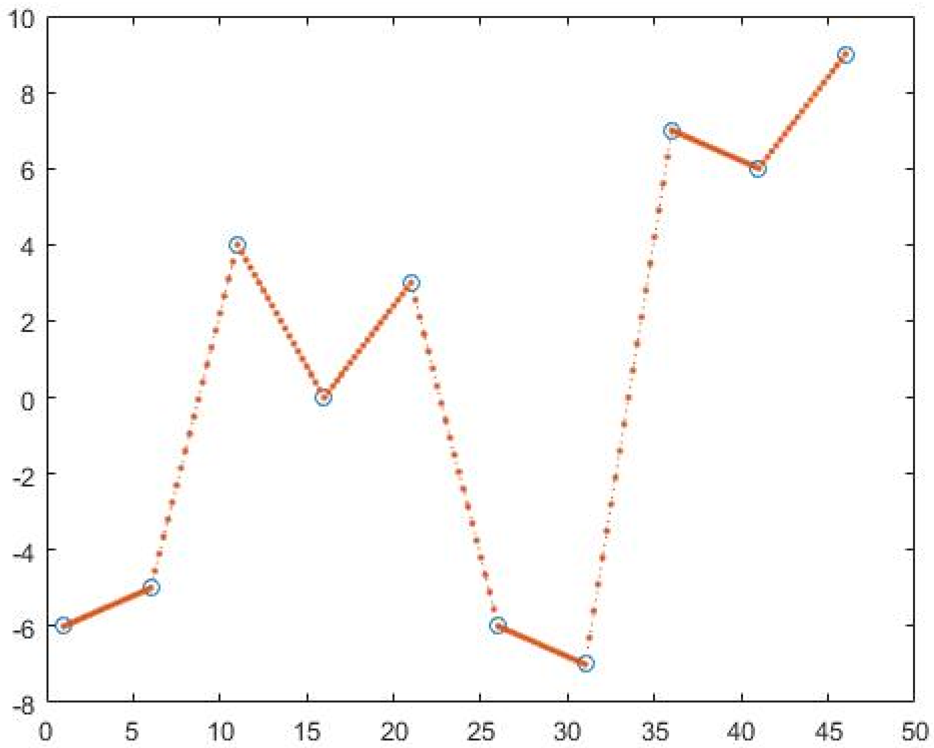


Ici, `x` et `y` sont les coordonnées des points de données et `z` sont les points pour lesquels nous avons besoin d'informations.

```
z = 0:0.25:50;
```

Une méthode permettant de trouver les valeurs y de z est l'interpolation linéaire par morceaux.

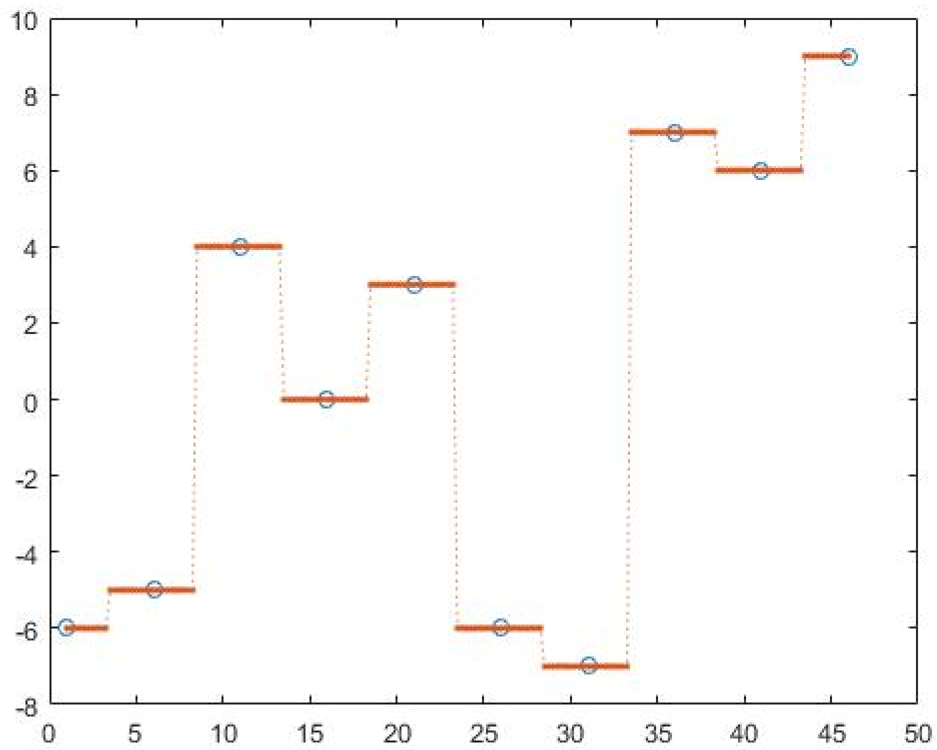
```
z_y = interp1(x,y,z,'linear');
```



On calcule ici la ligne entre deux points adjacents et obtient `z_y` en supposant que le point serait un élément de ces lignes.

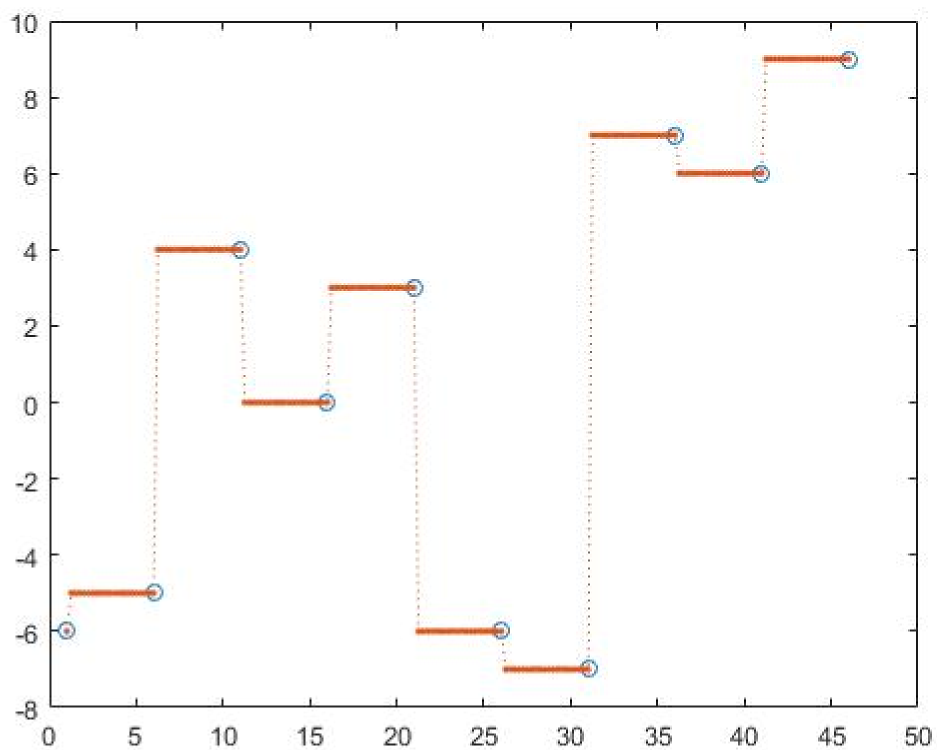
`interp1` fournit d'autres options comme l'interpolation la plus proche,

```
z_y = interp1(x,y,z, 'nearest');
```



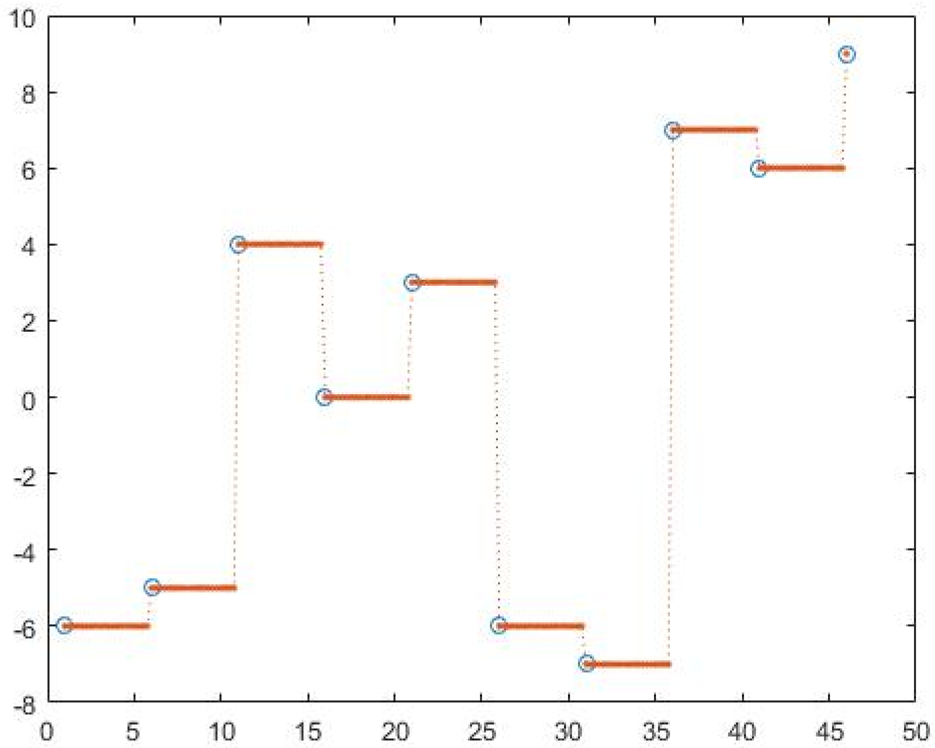
interpolation suivante,

```
z_y = interp1(x,y,z, 'next');
```



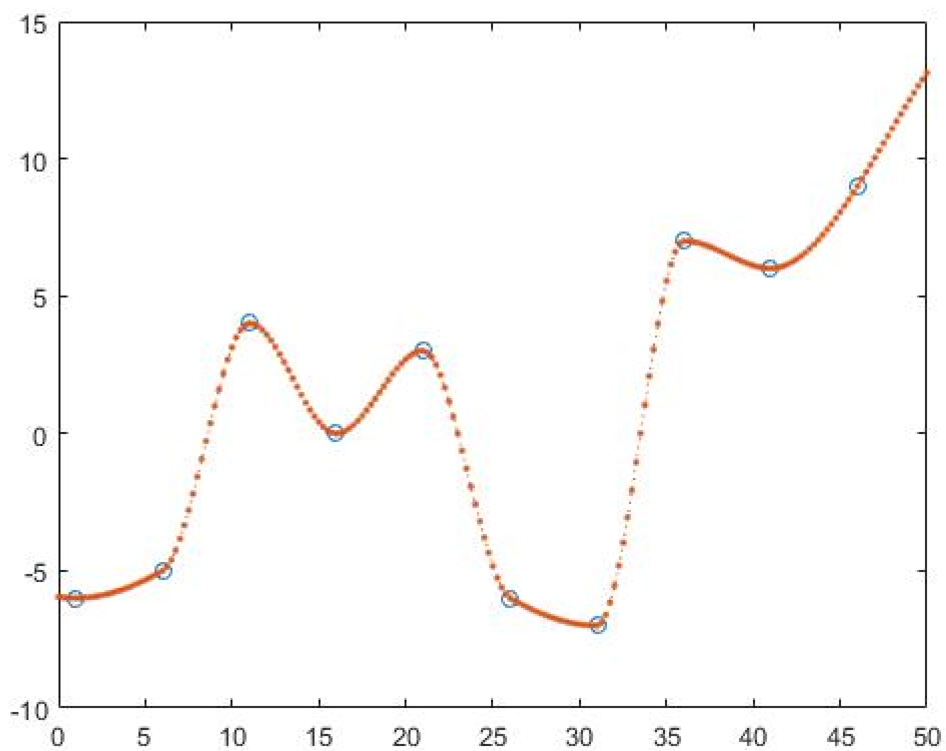
interpolation précédente,

```
z_y = interp1(x,y,z, 'previous');
```

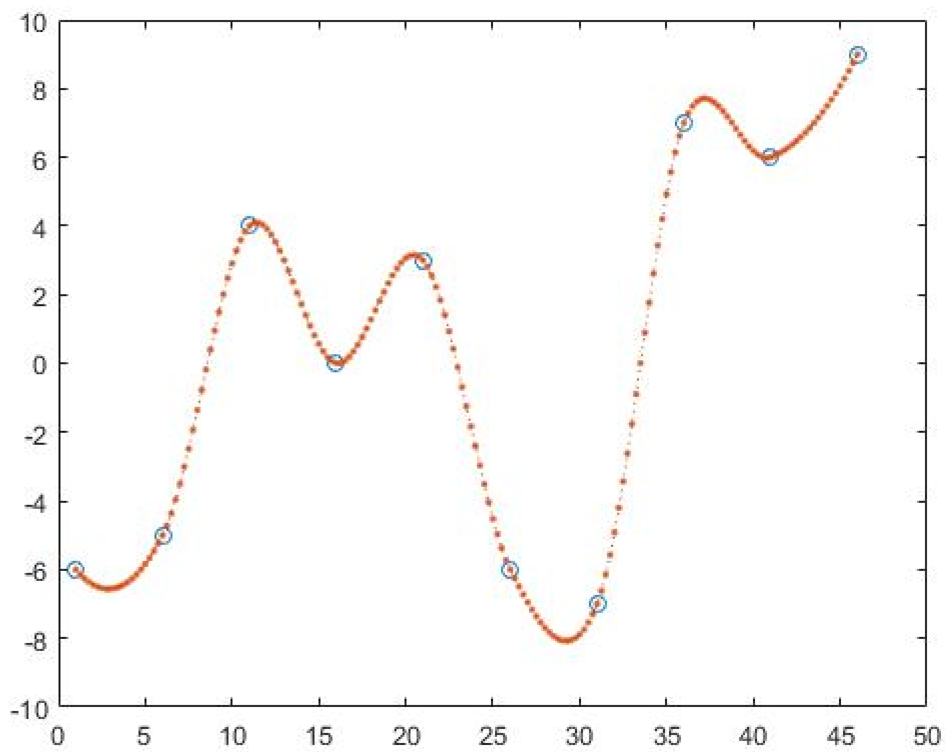


Interpolation cubique par morceaux préservant la forme,

```
z_y = interp1(x,y,z, 'pchip');
```

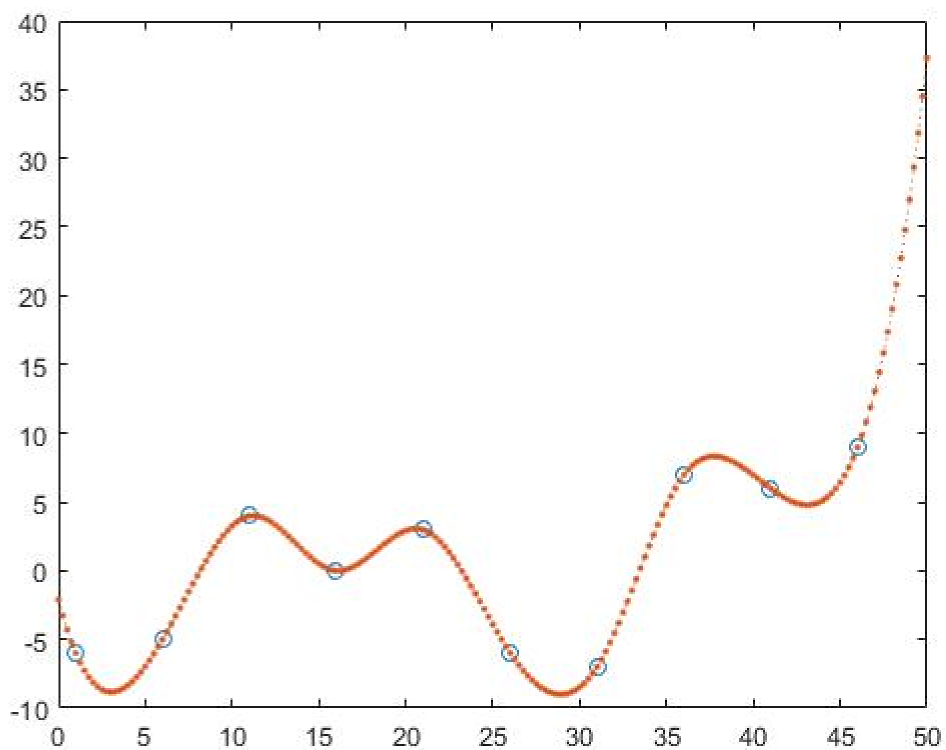


convolution cubique, z_y = interp1 (x, y, z, 'v5cubic');



et interpolation spline

```
z_y = interp1(x,y,z, 'spline');
```



Les interpolations constantes par morceaux sont les plus proches, les suivantes et les suivantes.

Interpolation polynomiale

Nous initialisons les données que nous voulons interpoler:

```
x = 0:0.5:10;
y = sin(x/2);
```

Cela signifie que la fonction sous-jacente pour les données dans l'intervalle [0,10] est sinusoïdale. Maintenant, les coefficients des polynômes approximatifs sont calculés:

```
p1 = polyfit(x,y,1);
p2 = polyfit(x,y,2);
p3 = polyfit(x,y,3);
p5 = polyfit(x,y,5);
p10 = polyfit(x,y,10);
```

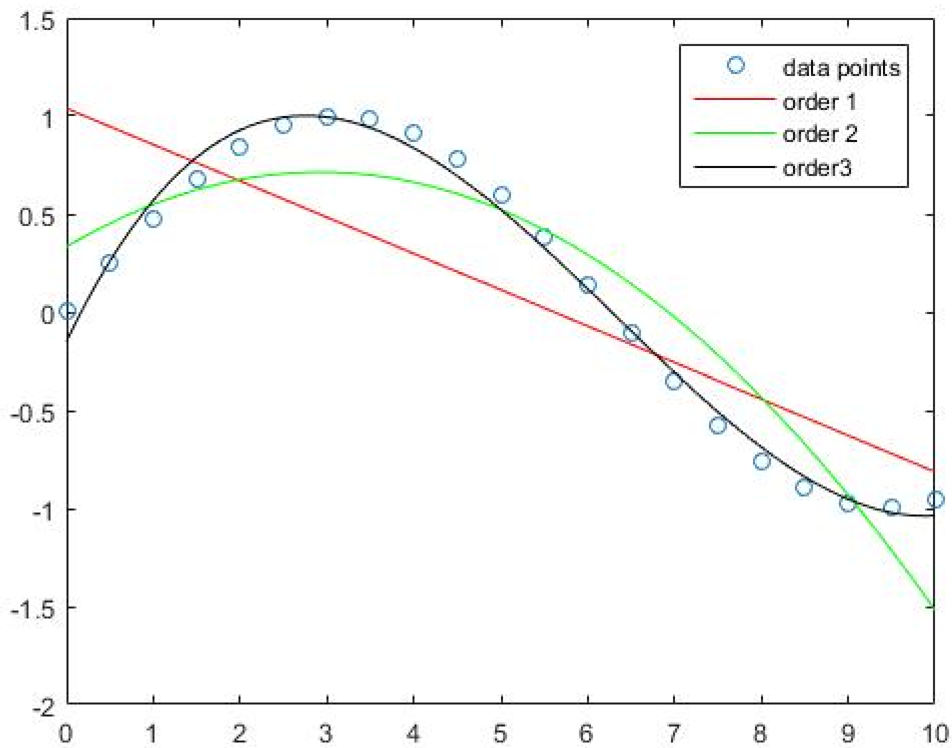
Voici `x` la valeur x et `y` la valeur y de nos points de données et le troisième nombre est l'ordre / degré du polynôme. Nous définissons maintenant la grille que nous voulons calculer notre fonction d'interpolation sur:

```
zx = 0:0.1:10;
```

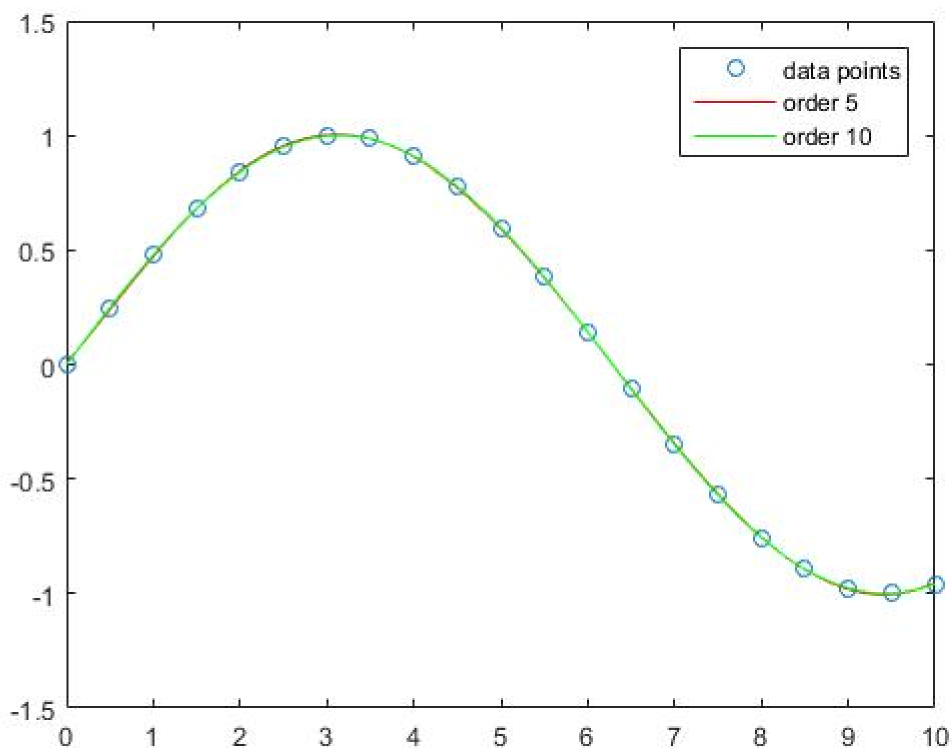
et calculer les valeurs y:

```
zy1 = polyval(p1,zx);
zy2 = polyval(p2,zx);
zy3 = polyval(p3,zx);
zy5 = polyval(p5,zx);
zy10 = polyval(p10,zx);
```

On peut voir que l'erreur d'approximation de l'échantillon diminue lorsque le degré du polynôme augmente.



Alors que l'approximation de la ligne droite dans cet exemple comporte des erreurs plus importantes, le polynôme d'ordre 3 estime que la fonction sinusale est relativement bonne dans cet intervalle.



L’interpolation avec les polynômes de l’ordre 5 et de l’ordre 10 n’a presque aucune erreur d’approbation.

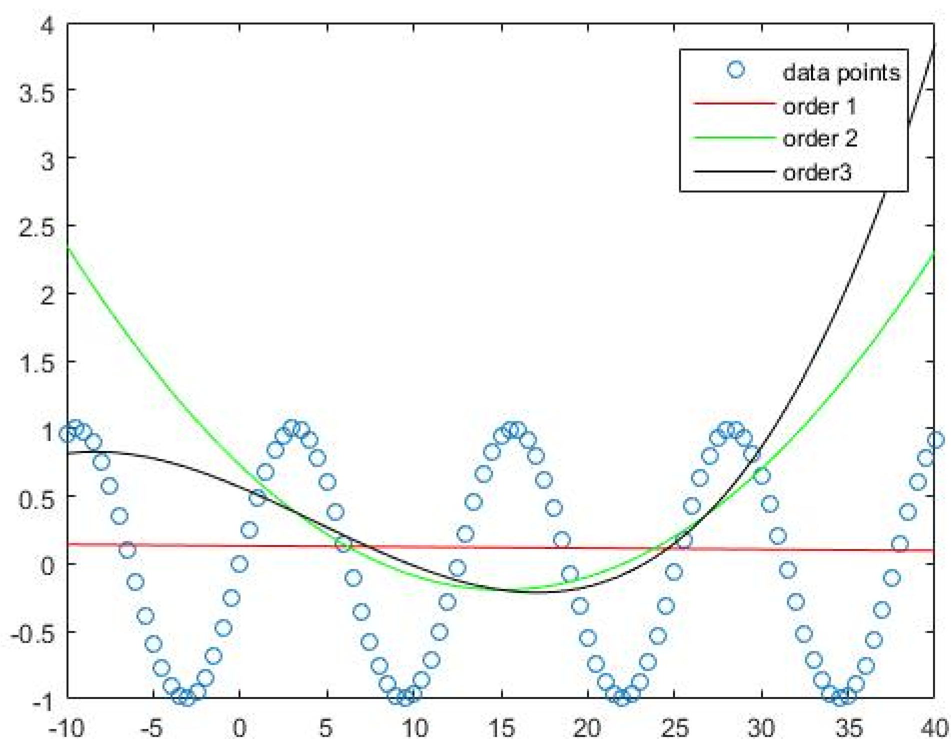
Cependant, si l'on considère les performances hors échantillon, on constate que les commandes trop élevées ont tendance à suréquiper et donc à produire de mauvais résultats. Nous fixons

```
zx = -10:0.1:40;
p10 = polyfit(X,Y,10);
p20 = polyfit(X,Y,20);
```

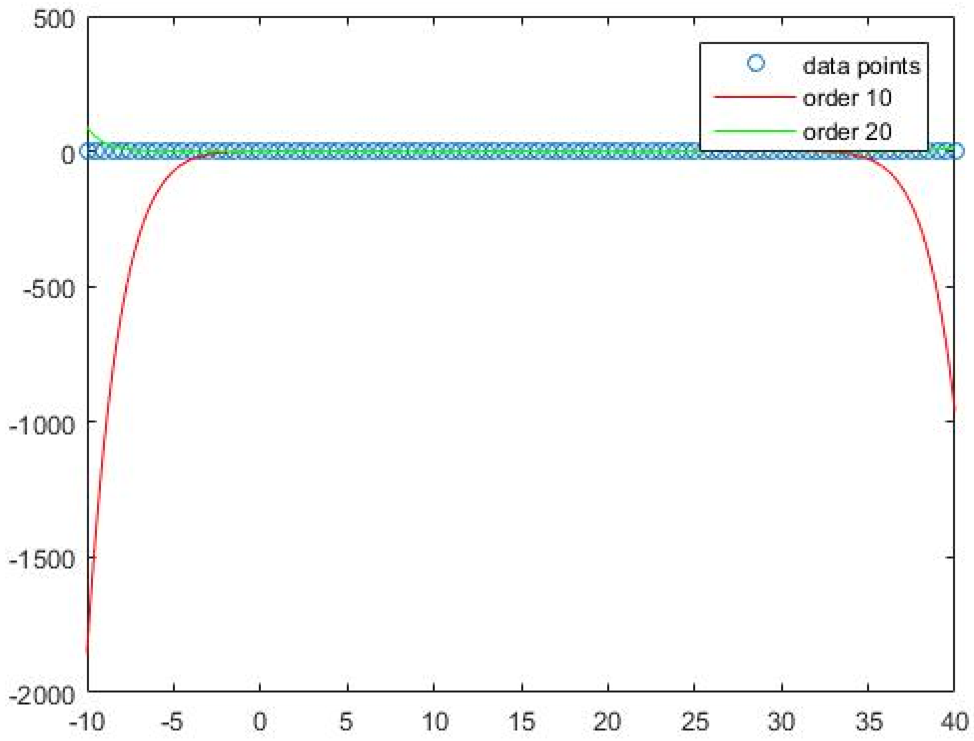
et

```
zy10 = polyval(p10,zx);
zy20 = polyval(p20,zx);
```

Si nous regardons l'intrigue, nous voyons que la performance hors échantillon est la meilleure pour la commande 1



et continue d'empirer avec un degré croissant.



Lire Interpolation avec MATLAB en ligne: https://riptutorial.com/fr/matlab/topic/6997/interpolation-avec-matlab

Chapitre 17: Introduction à l'API MEX

Examples

Vérifier le nombre d'entrées / sorties dans un fichier MEX C ++

Dans cet exemple, nous allons écrire un programme de base qui vérifie le nombre d'entrées et de sorties passées à une fonction MEX.

Pour commencer, nous devons créer un fichier C ++ implémentant la "passerelle MEX". C'est la fonction exécutée lorsque le fichier est appelé depuis MATLAB.

testinputs.cpp

```
// MathWorks provided header file
#include "mex.h"

// gateway function
void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
    // This function will error if number of inputs its not 3 or 4
    // This function will error if number of outputs is more than 1

    // Check inputs:
    if (nrhs < 3 || nrhs > 4) {
        mexErrMsgIdAndTxt("Testinputs:ErrorIdIn",
            "Invalid number of inputs to MEX file.");
    }

    // Check outputs:
    if (nlhs > 1) {
        mexErrMsgIdAndTxt("Testinputs:ErrorIdOut",
                "Invalid number of outputs to MEX file.");
    }
}
```

Tout d'abord, nous incluons l'en `mex.h` tête `mex.h` qui contient les définitions de toutes les fonctions et types de données requis pour fonctionner avec l'API MEX. Ensuite, nous implémentons la fonction `mexFunction` comme indiqué, où sa signature ne doit pas changer, indépendamment des entrées / sorties réellement utilisées. Les paramètres de la fonction sont les suivants:

- `nlhs` : Nombre de sorties demandées.
- `*plhs[]` : Tableau contenant toutes les sorties au format API MEX.
- `nrhs` : Nombre d'entrées passées.
- `*prhs[]` : Tableau contenant toutes les entrées au format API MEX.

Ensuite, nous vérifions le nombre d’arguments entrées / sorties, et si la validation échoue, une erreur est lancée avec la fonction `mexErrMsgIdAndTxt` (on attend un `somename:iD` identifiant de format `somename:iD` , un simple "ID" ne fonctionnera pas).

Une fois le fichier compilé en `mex testinputs.cpp` , la fonction peut être appelée dans MATLAB comme:

```
>> testinputs(2,3)
Error using testinputs. Invalid number of inputs to MEX file.

>> testinputs(2,3,5)

>> [~,~] = testinputs(2,3,3)
Error using testinputs. Invalid number of outputs to MEX file.
```

Entrez une chaîne, modifiez-la en C et affichez-la

Dans cet exemple, nous illustrons la manipulation de chaînes dans MATLAB MEX. Nous allons créer une fonction MEX qui accepte une chaîne en entrée de MATLAB, copier les données dans C-string, les modifier et les reconvertir en `mxArray` renvoyées du côté MATLAB.

L'objectif principal de cet exemple est de montrer comment les chaînes peuvent être converties en C / C ++ à partir de MATLAB et vice versa.

stringIO.cpp

```
#include "mex.h"
#include <cstring>

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
    // check number of arguments
    if (nrhs != 1 || nlhs > 1) {
        mexErrMsgIdAndTxt("StringIO:WrongNumArgs", "Wrong number of arguments.");
    }

    // check if input is a string
    if (mxIsChar(prhs[0])) {
        mexErrMsgIdAndTxt("StringIO:TypeError", "Input is not a string");
    }

    // copy characters data from mxArray to a C-style string (null-terminated)
    char *str = mxArrayToString(prhs[0]);

    // manipulate the string in some way
    if (strcmp("theOneString", str) == 0) {
        str[0] = 'T';  // capitalize first letter
    } else {
        str[0] = ' ';  // do something else?
    }

    // return the new modified string
    plhs[0] = mxCreateString(str);

    // free allocated memory
    mxFree(str);
}
```

Les fonctions pertinentes dans cet exemple sont les suivantes:

- `mxIsChar` pour tester si un `mxArray` est de type `mxCHAR` .
- `mxArrayToString` pour copier les données d'une chaîne `mxArray` dans un tampon `char *` .
- `mxCreateString` pour créer une chaîne `mxArray` partir d'un caractère `char*` .

En guise de remarque, si vous voulez seulement lire la chaîne et ne pas la modifier, n'oubliez pas de la déclarer en tant que `const char*` pour la rapidité et la robustesse.

Enfin, une fois compilé, nous pouvons l'appeler depuis MATLAB comme:

```
>> mex stringIO.cpp

>> strOut = stringIO('theOneString')
strOut =
TheOneString

>> strOut = stringIO('somethingelse')
strOut=
omethingelse
```

Passer une matrice 3D de MATLAB à C

Dans cet exemple, nous illustrons comment prendre une double matrice 3D de type réel à partir de MATLAB et la passer à un tableau C `double*` .

Les principaux objectifs de cet exemple montrent comment obtenir des données à partir de baies MATLAB MEX et mettre en évidence certains petits détails dans le stockage et la gestion de la matrice.

matrixIn.cpp

```
#include "mex.h"

void mexFunction(int  nlhs , mxArray *plhs[],
        int nrhs, mxArray const *prhs[]){
   // check amount of inputs
   if (nrhs!=1) {
        mexErrMsgIdAndTxt("matrixIn:InvalidInput", "Invalid number of inputs to MEX file.");
    }

   // check type of input
   if( !mxIsDouble(prhs[0]) || mxIsComplex(prhs[0])){
        mexErrMsgIdAndTxt("matrixIn:InvalidType",  "Input matrix must be a double, non-complex
array.");
   }

   // extract the data
   double const * const matrixAux= static_cast<double const *>(mxGetData(prhs[0]));

   // Get matrix size
   const mwSize *sizeInputMatrix= mxGetDimensions(prhs[0]);

   // allocate array in C. Note: its 1D array, not 3D even if our input is 3D
```

```
    double*  matrixInC= (double*)malloc(sizeInputMatrix[0] *sizeInputMatrix[1]
*sizeInputMatrix[2]* sizeof(double));


    // MATLAB is column major, not row major (as C). We need to reorder the numbers
    // Basically permutes dimensions

    // NOTE: the ordering of the loops is optimized for fastest memory access!
    // This improves the speed in about 300%

     const int size0 = sizeInputMatrix[0]; // Const makes compiler optimization kick in
     const int size1 = sizeInputMatrix[1];
     const int size2 = sizeInputMatrix[2];

     for (int j = 0; j < size2; j++)
     {
         int jOffset = j*size0*size1; // this saves re-computation time
         for (int k = 0; k < size0; k++)
         {
             int kOffset = k*size1; // this saves re-computation time
             for (int i = 0; i < size1; i++)
             {
                 int iOffset = i*size0;
                 matrixInC[i + jOffset + kOffset] = matrixAux[iOffset + jOffset + k];
             }
         }
     }

     // we are done!

     // Use your C matrix here

     // free memory
     free(matrixInC);
     return;
}
```

Les concepts pertinents à prendre en compte:

- Les matrices MATLAB sont toutes 1D en mémoire, quel que soit le nombre de dimensions utilisées dans MATLAB. Cela est également vrai pour la plupart (sinon la totalité) représentation de la matrice principale dans les bibliothèques C / C ++, ce qui permet une optimisation et une exécution plus rapide.
- Vous devez copier explicitement les matrices de MATLAB vers C dans une boucle.
- Les matrices MATLAB sont stockées dans l'ordre des colonnes, comme dans Fortran, mais C / C ++ et la plupart des langages modernes sont majeurs. Il est important de permuter la matrice d'entrée, sinon les données seront complètement différentes.

Les fonctions pertinentes dans cet exemple sont les suivantes:

- `mxIsDouble` vérifie si l'entrée est de type `double` .
- `mxIsComplex` vérifie si l'entrée est réelle ou imaginaire.
- `mxGetData` renvoie un pointeur sur les données réelles du tableau d'entrée. `NULL` s'il n'y a pas de données réelles.

- `mxGetDimensions` renvoie un pointeur sur un tableau `mwSize` , avec la taille de la dimension dans chaque index.

Passer une structure par noms de champs

Cet exemple montre comment lire des entrées de structure de différents types dans MATLAB et les transmettre à des variables de type équivalent C.

Bien qu'il soit possible et facile de comprendre à partir de l'exemple comment charger des champs par des nombres, cela est obtenu en comparant les noms de champs aux chaînes. Ainsi, les champs de la structure peuvent être adressés par leurs noms de champs et les variables peuvent être lues par C.

structIn.c

```
#include "mex.h"
#include <string.h> // strcmp


void mexFunction (int nlhs, mxArray *plhs[],
                  int nrhs, const mxArray *prhs[])
{
  // helpers
  double* double_ptr;
  unsigned int i; // loop variable

  // to be read variables
  bool optimal;
  int randomseed;
  unsigned int desiredNodes;

  if (!mxIsStruct(prhs[0])) {
    mexErrMsgTxt("First argument has to be a parameter struct!");
  }
  for (i=0; i<mxGetNumberOfFields(prhs[0]); i++) {
    if (0==strcmp(mxGetFieldNameByNumber(prhs[0],i),"randomseed")) {
      mxArray *p = mxGetFieldByNumber(prhs[0],0,i);
      randomseed = *mxGetPr(p);
    }
    if (0==strcmp(mxGetFieldNameByNumber(prhs[0],i),"optimal")) {
      mxArray *p = mxGetFieldByNumber(prhs[0],0,i);
      optimal = (bool)*mxGetPr(p);
    }
    if (0==strcmp(mxGetFieldNameByNumber(prhs[0],i),"numNodes")) {
      mxArray *p = mxGetFieldByNumber(prhs[0],0,i);
      desiredNodes = *mxGetPr(p);
    }
  }
}
```

La boucle sur `i` s'exécute sur tous les champs de la structure, tandis que les parties `if(0==strcmp)` comparent le nom du champ matlab à la chaîne donnée. S'il s'agit d'une correspondance, la valeur correspondante est extraite dans une variable C.

Lire Introduction à l'API MEX en ligne: https://riptutorial.com/fr/matlab/topic/680/introduction-a-l-api-mex

Chapitre 18: L'intégration

Examples

Intégrale, intégrale2, intégrale3

1 dimension

Intégrer une fonction unidimensionnelle

```
f = @(x) sin(x).^3 + 1;
```

dans la gamme

```
xmin = 2;
xmax = 8;
```

on peut appeler la fonction

```
q = integral(f,xmin,xmax);
```

il est également possible de définir des limites pour les erreurs relatives et absolues

```
q = integral(f,xmin,xmax, 'RelTol',10e-6, 'AbsTol',10-4);
```

2 dimensions

Si l'on veut intégrer une fonction bidimensionnelle

```
f = @(x,y) sin(x).^y ;
```

dans la gamme

```
xmin = 2;
xmax = 8;
ymin = 1;
ymax = 4;
```

on appelle la fonction

```
q = integral2(f,xmin,xmax,ymin,ymax);
```

Comme dans l'autre cas, il est possible de limiter les tolérances

```
q = integral2(f,xmin,xmax,ymin,ymax, 'RelTol',10e-6, 'AbsTol',10-4);
```

3 dimensions

Intégrer une fonction tridimensionnelle

```
f = @(x,y,z) sin(x).^y - cos(z) ;
```

dans la gamme

```
xmin = 2;
xmax = 8;
ymin = 1;
ymax = 4;
zmin = 6;
zmax = 13;
```

est effectué en appelant

```
q = integral3(f,xmin,xmax,ymin,ymax, zmin, zmax);
```

Encore une fois, il est possible de limiter les tolérances

```
q = integral3(f,xmin,xmax,ymin,ymax, zmin, zmax, 'RelTol',10e-6, 'AbsTol',10-4);
```

Lire L'intégration en ligne: https://riptutorial.com/fr/matlab/topic/7022/l-integration

Chapitre 19: Le débogage

Syntaxe

- dbstop dans le fichier à l'emplacement si expression

Paramètres

Paramètre	Détails
fichier	Nom du fichier `.m` (sans extension), par exemple `fit` . Ce paramètre est *(obligatoire)* sauf si vous définissez des types de points d'arrêt conditionnels spéciaux tels que `dbstop if error` ou `dbstop if naninf` .
emplacement	Numéro de ligne où le point d'arrêt doit être placé. Si la ligne spécifiée ne contient pas de code exécutable, le point d'arrêt sera placé sur la première ligne valide **après** celle spécifiée.
expression	Toute expression ou combinaison de celles-ci qui donne une valeur booléenne. Exemples: `ind == 1` , `nargin < 4 && isdir('Q:\')` .

Examples

Travailler avec des points d'arrêt

Définition

> *En développement logiciel, un **point d'arrêt** est un **point** d'arrêt ou de pause intentionnel dans un programme, mis en place à des fins de débogage.*
>
> *Plus généralement, un point d'arrêt est un moyen d'acquérir des connaissances sur un programme pendant son exécution. Pendant l'interruption, le programmeur inspecte l'environnement de test (registres généraux, mémoire, journaux, fichiers, etc.) pour savoir si le programme fonctionne comme prévu. En pratique, un point d'arrêt consiste en une ou plusieurs conditions qui déterminent à quel moment l'exécution d'un programme doit être interrompue.*
>
> -Wikipédia

Points d'arrêt dans MATLAB

Motivation

Dans MATLAB, lorsque l'exécution s'interrompt à un point d'arrêt, les variables existant dans l'espace de travail actuel (ou *portée*) ou l'un des espaces de travail appelants peuvent être inspectés (et généralement modifiés).

Types de points d'arrêt

MATLAB permet aux utilisateurs de placer deux types de points d'arrêt dans les fichiers `.m` :

- *Points d'* arrêt standard (ou " *illimités* ") (affichés en rouge) - suspendre l'exécution à chaque fois que la ligne marquée est atteinte.
- Points d'arrêt "conditionnels" (affichés en jaune) - interrompre l'exécution à chaque fois que la ligne marquée est atteinte ET que la condition définie dans le point d'arrêt est évaluée comme `true` .

Placer les points d'arrêt

Les deux types de points d'arrêt peuvent être créés de plusieurs manières:

- En utilisant l'interface graphique de l'éditeur MATLAB, en cliquant avec le bouton droit sur la ligne horizontale située à côté du numéro de ligne.

- En utilisant la commande `dbstop` :

```
% Create an unrestricted breakpoint:
dbstop in file at location
% Create a conditional breakpoint:
dbstop in file at location if expression

% Examples and special cases:
dbstop in fit at 99 % Standard unrestricted breakpoint.

dbstop in fit at 99 if nargin==3 % Standard conditional breakpoint.

dbstop if error % This special type of breakpoint is not limited to a specific file, and
                % will trigger *whenever* an error is encountered in "debuggable" code.

dbstop in file % This will create an unrestricted breakpoint on the first executable line
               % of "file".
```

```
dbstop if naninf % This special breakpoint will trigger whenever a computation result
                 % contains either a NaN (indicates a division by 0) or an Inf
```

- Utilisation des raccourcis clavier: la clé par défaut pour créer un point d'arrêt standard sous Windows est `F12` ; la clé par défaut pour les points d'arrêt conditionnels n'est pas *définie* .

Désactivation et réactivation des points d'arrêt

Désactiver un point d'arrêt pour l'ignorer temporairement: les points d'arrêt désactivés ne suspendent pas l'exécution. La désactivation d'un point d'arrêt peut se faire de plusieurs manières:

- Faites un clic droit sur le cercle rouge / jaune du point d'arrêt> Désactiver le point d'arrêt.
- Clic gauche sur un point d'arrêt conditionnel (jaune).
- Dans l'onglet Editeur> Points d'arrêt> Activer \ Désactiver.

Suppression des points d'arrêt

Tous les points d'arrêt restent dans un fichier jusqu'à leur suppression, manuellement ou automatiquement. Les points d'arrêt sont effacés *automatiquement* à la fin de la session MATLAB (c'est-à-dire la fin du programme). La suppression manuelle des points d'arrêt se fait de l'une des manières suivantes:

- En utilisant la commande `dbclear` :

```
dbclear all
dbclear in file
dbclear in file at location
dbclear if condition
```

- Clic gauche sur une icône de point d'arrêt standard ou une icône de point d'arrêt conditionnel désactivé.

- Clic droit sur n'importe quel point d'arrêt> Effacer le point d'arrêt.

- Dans l'onglet Editeur> Points d'arrêt> Effacer tout.

- Dans les versions antérieures à R2015b de MATLAB, utilisez la commande `clear` .

Reprise de l'exécution

Lorsque l'exécution est suspendue à un point d'arrêt, il existe deux manières de continuer à exécuter le programme:

- Exécutez la ligne en cours et faites une nouvelle pause avant la ligne suivante.

  `F10` [1] dans l'éditeur, `dbstep` dans la fenêtre de commande, "Step" dans le ruban> Editor> DEBUG.

- Exécutez jusqu'au prochain point d'arrêt (s'il n'y a plus de points d'arrêt, l'exécution se poursuit jusqu'à la fin du programme).

  `F12` [1] dans l'éditeur, `dbcont` dans la fenêtre de commande, "Continuer" dans le ruban> éditeur> DEBUG.

---

[1] - par défaut sous Windows.

Débogage du code Java appelé par MATLAB

Vue d'ensemble

Pour déboguer les classes Java appelées lors de l'exécution de MATLAB, il est nécessaire d'effectuer deux étapes:

1. Exécutez MATLAB en mode de débogage JVM.
2. Attachez un débogueur Java au processus MATLAB.

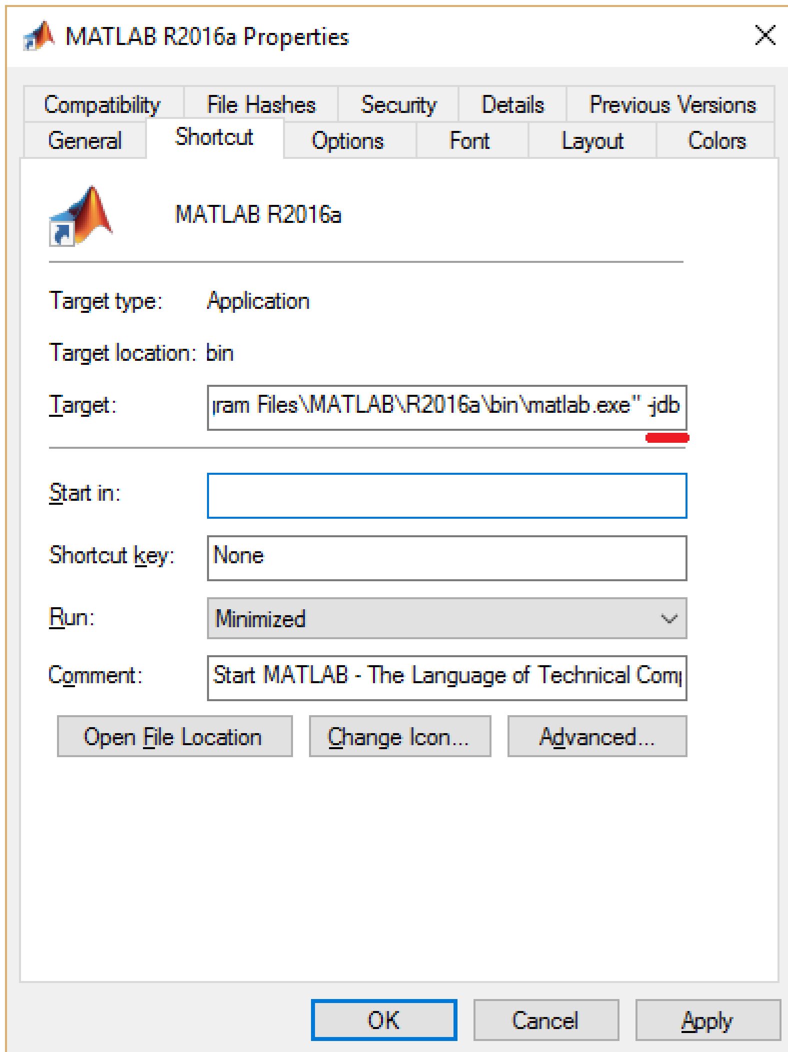
Lorsque MATLAB est démarré en mode de débogage JVM, le message suivant apparaît dans la fenêtre de commande:

```
JVM is being started with debugging enabled.
Use "jdb -connect com.sun.jdi.SocketAttach:port=4444" to attach debugger.
```

MATLAB fin

Les fenêtres:

Créez un raccourci vers l'exécutable MATLAB (`matlab.exe`) et ajoutez l'indicateur `-jdb` à la fin, comme indiqué ci-dessous:



Lors de l'exécution de MATLAB à l'aide de ce raccourci, le débogage JVM sera activé.

`java.opts` fichier `java.opts` peut également être créé / mis à jour. Ce fichier est stocké dans "matlab-root \ bin \ arch", où "matlab-root" est le directoy d'installation de MATLAB et "arch" est l'architecture (par exemple "win32").

Les éléments suivants doivent être ajoutés dans le fichier:

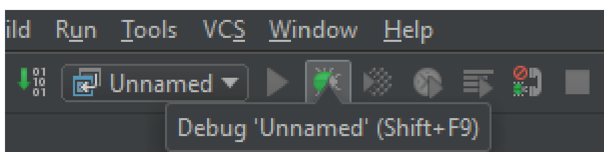
```
-Xdebug
-Xrunjdwp:transport=dt_socket,address=1044,server=y,suspend=n
```

Fin du débogueur

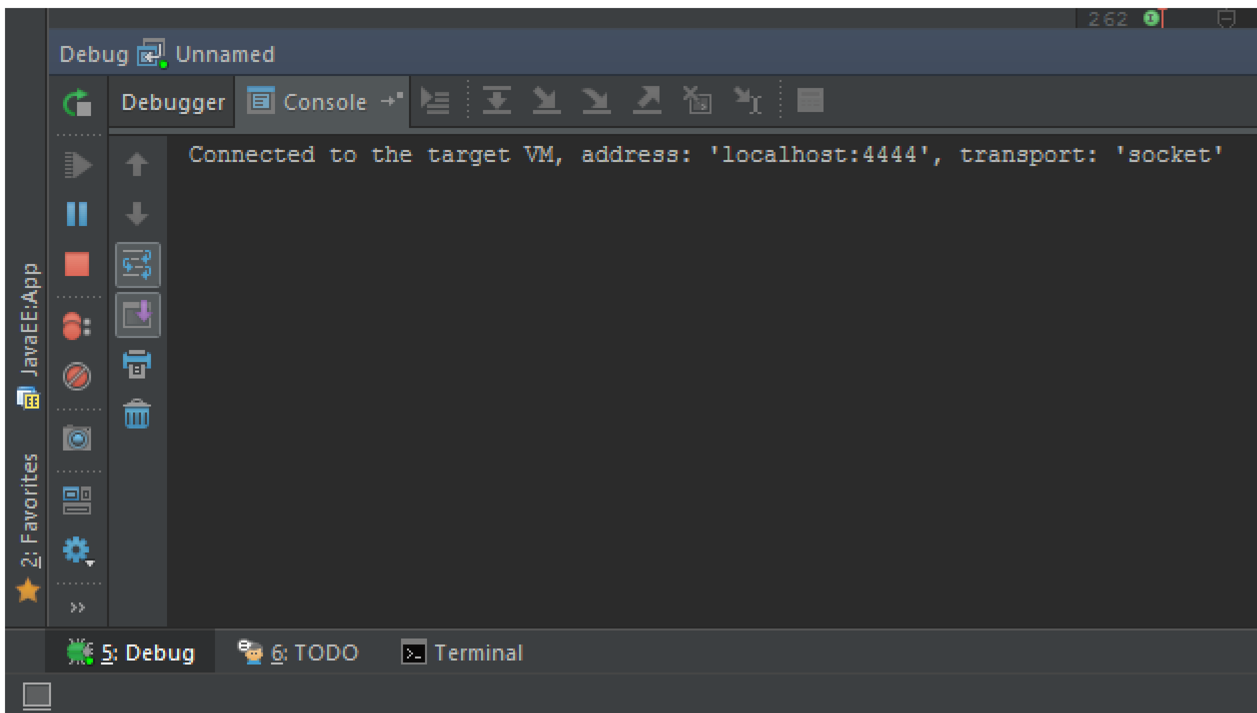
IntelliJ IDEA

La connexion de ce débogueur nécessite la création d'une configuration de "débogage distant" avec le port exposé par MATLAB:

Ensuite, le débogueur est démarré:



Si tout fonctionne comme prévu, le message suivant apparaîtra dans la console:



Lire Le débogage en ligne: https://riptutorial.com/fr/matlab/topic/1045/le-debogage

Chapitre 20: Lecture de gros fichiers

Examples

textescan

Supposons que vous ayez formaté des données dans un fichier texte ou une chaîne volumineuse, par exemple

```
Data,2015-09-16,15:41:52;781,780.000000,0.0034,2.2345
Data,2015-09-16,15:41:52;791,790.000000,0.1255,96.5948
Data,2015-09-16,15:41:52;801,800.000000,1.5123,0.0043
```

on peut utiliser `textscan` pour le lire assez rapidement. Pour cela, obtenez un identifiant de fichier du fichier texte avec `fopen` :

```
fid = fopen('path/to/myfile');
```

Supposons pour les données de cet exemple que nous souhaitons ignorer la première colonne "Données", lire la date et l'heure sous forme de chaînes et lire le reste des colonnes en double, c.-à-d.

```
 Data   ,  2015-09-16  , 15:41:52;801  , 800.000000  , 1.5123  ,  0.0043
ignore      string         string          double       double      double
```

Pour ce faire, appelez:

```
data = textscan(fid,'%*s %s %s %f %f %f','Delimiter',',');
```

L'astérisque dans `%*s` signifie "ignorer cette colonne". `%s` signifie "interpréter comme une chaîne". `%f` signifie "interpréter comme des doubles (flottants)". Enfin, `'Delimiter',','` indique que toutes les virgules doivent être interprétées comme le délimiteur entre chaque colonne.

Pour résumer:

```
fid = fopen('path/to/myfile');
data = textscan(fid,'%*s %s %s %f %f %f','Delimiter',',');
```

`data` contiennent maintenant un tableau de cellules avec chaque colonne dans une cellule.

Chaînes de date et d'heure vers un tableau numérique rapide

La conversion des chaînes de date et d'heure en tableaux numériques peut s'effectuer avec le `datenum` , bien que la lecture d'un fichier de données volumineux puisse prendre jusqu'à la moitié du temps.

Considérons les données dans l'exemple **Textscan** . En utilisant à nouveau les Texscan et en interprétant la date et l'heure comme des nombres entiers, ils peuvent être rapidement convertis en un tableau numérique.

Une ligne dans l'exemple de données serait interprétée comme suit:

```
 Data , 2015  - 09  -  16  ,  15  :  41  :  52  ;  801 , 800.000000 , 1.5123 , 0.0043
ignore double double double double double double double    double     double   double
```

qui sera lu comme:

```
fid = fopen('path/to/myfile');
data = textscan(fid,'%*s %f %f %f %f %f %f %f %f %f %f','Delimiter',',-:;');
fclose(fid);
```

À présent:

```
y = data{1};          % year
m = data{2};          % month
d = data{3};          % day
H = data{4};          % hours
M = data{5};          % minutes
S = data{6};          % seconds
F = data{7};          % milliseconds

% Translation from month to days
ms = [0,31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334];

n = length(y);        % Number of elements
Time = zeros(n,1);    % Declare numeric time array

% Algorithm for calculating numeric time array
for k = 1:n
    Time(k) = y(k)*365 + ms(m(k)) + d(k) + floor(y(k)/4)...
              - floor(y(k)/100) + floor(y(k)/400) + (mod(y(k),4)~=0)...
              - (mod(y(k),100)~=0) + (mod(y(k),400)~=0)...
              + (H(k)*3600 + M(k)*60 + S(k) + F(k)/1000)/86400 + 1;
end
```

En utilisant le `datenum` sur 566 678 éléments, il fallait 6,626570 secondes, alors que la méthode ci-dessus nécessitait 0,048334 seconde, soit 0,73% du temps pour le `datenum` ou ~ 137 fois plus vite.

Lire Lecture de gros fichiers en ligne: https://riptutorial.com/fr/matlab/topic/9023/lecture-de-gros-fichiers

Chapitre 21: Les fonctions

Examples

Exemple de base

Le script MATLAB suivant montre comment définir et appeler une fonction de base:

myFun.m :

```
function [out1] = myFun(arg0, arg1)
    out1 = arg0 + arg1;
end
```

terminal :

```
>> res = myFun(10, 20)

res =

    30
```

Plusieurs sorties

Le script MATLAB suivant montre comment renvoyer plusieurs sorties en une seule fonction:

myFun.m :

```
function [out1, out2, out3] = myFun(arg0, arg1)
    out1 = arg0 + arg1;
    out2 = arg0 * arg1;
    out3 = arg0 - arg1;
end
```

terminal :

```
>> [res1, res2, res3] = myFun(10, 20)

res1 =

        30

res2 =

        200

res3 =
        -10
```

Cependant, MATLAB ne renverra que la première valeur lorsqu'il est assigné à une seule variable

```
    >> res = myFun(10, 20)

    res =

            30
```

L'exemple suivant montre comment obtenir une sortie spécifique

```
    >> [~, res] = myFun(10, 20)

    res =

            200
```

nargin, nargout

Dans le corps d'une fonction, `nargin` et `nargout` indiquent respectivement le nombre réel d'entrées et de sorties fournies dans l'appel.

Nous pouvons par exemple contrôler l'exécution d'une fonction en fonction du nombre d'entrées fournies.

myVector.m :

```
function [res] = myVector(a, b, c)
    % Roughly emulates the colon operator

    switch nargin
        case 1
            res = [0:a];
        case 2
            res = [a:b];
        case 3
            res = [a:b:c];
        otherwise
            error('Wrong number of params');
    end
end
```

Terminal:

```
>> myVector(10)

ans =

    0     1     2     3     4     5     6     7     8     9    10

>> myVector(10, 20)

ans =

   10    11    12    13    14    15    16    17    18    19    20

>> myVector(10, 2, 20)
```

```
ans =

   10    12    14    16    18    20
```

De la même manière, nous pouvons contrôler l'exécution d'une fonction en fonction du nombre de paramètres de sortie.

myIntegerDivision :

```
function [qt, rm] = myIntegerDivision(a, b)
    qt = floor(a / b);

    if nargout == 2
        rm = rem(a, b);
    end
end
```

terminal :

```
>> q = myIntegerDivision(10, 7)

q = 1

>> [q, r] = myIntegerDivision(10, 7)

q = 1
r = 3
```

Lire Les fonctions en ligne: https://riptutorial.com/fr/matlab/topic/5659/les-fonctions

Chapitre 22: Meilleures pratiques MATLAB

Remarques

Cette rubrique affiche les meilleures pratiques que la communauté a apprises au fil du temps.

Examples

Gardez les lignes courtes

Utilisez le caractère de continuation (points de suspension) ... pour continuer la longue déclaration.

Exemple:

```
MyFunc( parameter1,parameter2,parameter3,parameter4, parameter5, parameter6,parameter7,
parameter8, parameter9)
```

peut être remplacé par:

```
MyFunc( parameter1, ...
        parameter2, ...
        parameter3, ...
        parameter4, ...
        parameter5, ...
        parameter6, ...
        parameter7, ...
        parameter8, ...
        parameter9)
```

Indentez le code correctement

Une indentation correcte donne non seulement un aspect esthétique, mais augmente également la lisibilité du code.

Par exemple, considérez le code suivant:

```
%no need to understand the code, just give it a look
n = 2;
bf = false;
while n>1
for ii = 1:n
for jj = 1:n
if ii+jj>30
bf = true;
break
end
end
if bf
break
```

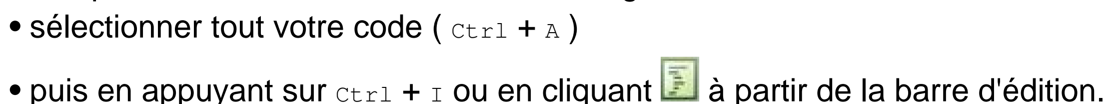
```
end
end
if bf
break
end
n = n + 1;
end
```

Comme vous pouvez le voir, vous devez regarder attentivement pour voir quelle boucle et `if` instructions se terminent où.
Avec l'indentation intelligente, vous aurez ce look:

```
n = 2;
bf = false;
while n>1
    for ii = 1:n
        for jj = 1:n
            if ii+jj>30
                bf = true;
                break
            end
        end
        if bf
            break
        end
    end
    if bf
        break
    end
    n = n + 1;
end
```

Cela indique clairement le début et la fin de l'instruction loop / `if` .

Vous pouvez faire une mise en retrait intelligente en:

- sélectionner tout votre code (`Ctrl` + `A`)
- puis en appuyant sur `Ctrl` + `I` ou en cliquant  à partir de la barre d'édition.

Utiliser assert

Matlab permet à certaines erreurs très triviales de passer inaperçues, ce qui peut entraîner une erreur beaucoup plus tard dans la course, rendant le débogage difficile. Si vous **assumez** quelque chose à propos de vos variables, **validez-** le.

```
function out1 = get_cell_value_at_index(scalar1,cell2)
```

```
assert(isscalar(scalar1),'1st input must be a scalar')
assert(iscell(cell2),'2nd input must be a cell array')

assert(numel(cell2) >= scalar1),'2nd input must have more elements than the value of the 1st
input')
assert(~isempty(cell2{scalar1}),'2nd input at location is empty')

out1 = cell2{scalar1};
```

Éviter les boucles

La plupart du temps, les boucles coûtent cher avec Matlab. Votre code sera plus rapide si vous utilisez la vectorisation. Cela rend souvent votre code plus modulaire, facilement modifiable et plus facile à déboguer. L'inconvénient majeur est que vous devez prendre le temps de planifier les structures de données et que les erreurs de dimension sont plus faciles à trouver.

Exemples

N'écris pas

```
for t=0:0.1:2*pi
    R(end+1)=cos(t);
end
```

mais

```
t=0:0.1:2*pi;
R=cos(t)
```

N'écris pas

```
for i=1:n
    for j=1:m
        c(i,j)=a(i)+2*b(j);
    end
end
```

Mais quelque chose de similaire à

```
c=repmat(a.',1,m)+2*repmat(b,n,1)
```

Pour plus de détails, voir vectorisation

Créer un nom unique pour un fichier temporaire

Lors du codage d'un script ou d'une fonction, il peut arriver qu'un ou plusieurs fichiers temporaires soient nécessaires pour, par exemple, stocker des données.

Afin d'éviter d'écraser un fichier existant ou d'occulter une fonction MATLAB, la fonction

tempname peut être utilisée pour générer un **nom unique** pour un fichier temporaire dans le dossier temporaire du système.

```
my_temp_file=tempname
```

Le nom de fichier est généré sans l'extension; il peut être ajouté en concaténant l'extension souhaitée au nom généré par `tempname`

```
my_temp_file_with_ext=[tempname '.txt']
```

La localisation du dossier temporaire du système peut être récupérée en calant la fonction tempdir .

Si, pendant l'exécution de la fonction / script, le fichier temporaire n'est plus nécessaire, il peut être supprimé en utilisant la fonction delete

Depuis `delete` ne demande pas de confirmation, il pourrait être utile de mettre `on` la possibilité de déplacer le fichier à supprimer dans le `recycle` dossier.

Cela peut être fait en utilisant la fonction recycle de cette façon:

```
recycle('on')
```

Dans l'exemple suivant, une utilisation possible des fonctions `tempname` , `delete` et `recycle` est proposée.

```
%
% Create some example data
%
theta=0:.1:2*pi;
x=cos(theta);
y=sin(theta);
%
% Generate the temporary filename
%
my_temp_file=[tempname '.mat'];
%
% Split the filename (path, name, extension) and display them in a message box
[tmp_file_path,tmp_file_name, tmp_file_ext]=fileparts(my_temp_file)
uiwait(msgbox(sprintf('Path= %s\nName= %s\nExt= %s', ...
              tmp_file_path,tmp_file_name,tmp_file_ext),'TEMPORARY FILE'))
%
% Save the varaibles in a temporary file
%
save(my_temp_file,'x','y','theta')
%
% Load the varaibles from the temporary file
%
load(my_temp_file)
%
% Set the reclycle option on
%
recycle('on')
%
```

```
% Delete the temporary file
%
delete(my_temp_file)
```

Caveat

Le nom de fichier temporaire est généré à l'aide de la méthode `java.util.UUID.randomUUID` (randomUUID).

Si MATLAB est exécuté sans JVM, le nom de fichier temporaire est généré en utilisant `matlab.internal.timing.timing` basé sur le compteur et l'heure du processeur. Dans ce cas, le nom de fichier temporaire n'est pas garanti unique.

Utiliser les attributs valides

La fonction validateattributes peut être utilisée pour valider un tableau par rapport à un ensemble de spécifications

Il peut donc être utilisé pour valider l'entrée fournie à une fonction.

Dans l'exemple suivant, la fonction `test_validateattributes` requiert trois entrées

```
function test_validateattributes(input_1,input_2,input_3)
```

Les spécifications d'entrée sont les suivantes:

- array_1:
  - classe: double
  - taille: [3,2]
  - valeurs: les éléments ne doivent pas être NaN
- char_array:
  - classe: char
  - value: la chaîne ne doit pas être vide
- tableau_3
  - classe: double
  - taille: [5 1]
  - valeurs: les éléments doivent être réels

Pour valider les trois entrées, la fonction `validateattributes` peut être appelée avec la syntaxe suivante:

```
validateattributes(A,classes,attributes,funcName,varName,argIndex)
```

où:

- `A` est le tableau à vider

- `classes` : est le `type` du tableau (par exemple, `single` , `double` , `logical`)
- `attributes` : sont les `attributes` auxquels le tableau en entrée doit correspondre (par exemple, `[3,2], size` pour spécifier la taille du tableau, `nonnan` pour spécifier que le tableau ne doit pas avoir de valeurs NaN)
- `funcName` : est le nom de la fonction dans laquelle la validation a lieu. Cet argument est utilisé dans la génération du message d'erreur (le cas échéant)
- `varName` : est le nom du tableau en cours de validation. Cet argument est utilisé dans la génération du message d'erreur (le cas échéant)
- `argIndex` : est la position du tableau inpurt dans la liste des entrées. Cet argument est utilisé dans la génération du message d'erreur (le cas échéant)

Si une ou plusieurs entrées ne correspondent pas à la spécification, un message d'erreur est généré.

En cas de plusieurs entrées non valides, la validation s'arrête lorsque la première incompatibilité est trouvée.

C'est la `function test_validateattributes` dans laquelle la validation d'entrée a été implémentée.

Étant donné que la fonction nécessite trois entrées, une première vérification du nombre d'entrées fournies est effectuée à l'aide de la fonction nargin .

```
function test_validateattributes(array_1,char_array_1,array_3)
%
% Check for the number of expected input: if the number of input is less
% than the require, the function exits with an error message
%
if(nargin ~= 3)
   error('Error: TEST_VALIDATEATTRIBUTES requires 3 input, found %d',nargin)
end
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Definition of the expected characteristics of the first input %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% INPUT #1 name (only used in the generation of the error message)
%
input_1_name='array_1';
%
% INPUT #1 position (only used in the generation of the error message)
%
input_1_position=1;
%
% Expected CLASS of the first input MUST BE "double"
%
input_1_class={'double'};
%
% Expected ATTRIBUTES of the first input
%   SIZE: MUST BE [3,2]
%
input_1_size_attribute='size';
input_1_size=[3,2];
%
%   VALUE CHECK: the element MUST BE NOT NaN
%
```

```
input_1_value_type='nonnan';
%
% Build the INPUT 1 attributes
%
input_1_attributes={input_1_size_attribute,input_1_size, ...
                    input_1_value_type};
%
% CHECK THE VALIDITY OF THE FIRST INPUT
%
validateattributes(array_1, ...
                   input_1_class,input_1_attributes,'', ...
                   input_1_name,input_1_position);

%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Definition of the expected characteristics of the second input %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% INPUT #1 name (only used in the generation of the error message)
%
input_2_name='char_array_1';
%
% INPUT #2 position (only used in the generation of the error message)
%
input_2_position=2;
%
% Expected CLASS of the first input MUST BE "string"
%
input_2_class={'char'};
%
%   VALUE CHECK: the element must be not NaN
%
input_2_size_attribute='nonempty';
%
% Build the INPUT 2 attributes
%
input_2_attributes={input_2_size_attribute};
%
% CHECK THE VALIDITY OF THE SECOND INPUT
%
validateattributes(char_array_1, ...
                   input_2_class,input_2_attributes,'', ...
                   input_2_name,input_2_position);
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Definition of the expected characteristics of the third input %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% INPUT #3 name (only used in the generation of the error message)
%
input_3_name='array_3';
%
% INPUT #3 position (only used in the generation of the error message)
%
input_3_position=3;
%
% Expected CLASS of the first input MUST BE "double"
%
input_3_class={'double'};
%
% Expected ATTRIBUTES of the first input
```

```
%   SIZE: must be [5]
input_3_size_attribute='size';
input_3_size=[5 1];
%   VALUE CHECK: the elements must be real
input_3_value_type='real';
%
% Build the INPUT 3 attributes
%
input_3_attributes={input_3_size_attribute,input_3_size, ...
                    input_3_value_type};
%
% CHECK THE VALIDITY OF THE FIRST INPUT
%
validateattributes(array_3, ...
                   input_3_class,input_3_attributes,'', ...
                   input_3_name,input_3_position);

disp('All the three input are OK')
```

Le script suivant peut être utilisé pour tester l'implémentation de la procédure de validation.

Il génère les trois entrées requises et, au hasard, les rend non valides.

```
%
% Generate the first input
%
n_rows=randi([2 3],1);
n_cols=2;
input_1=randi([20 30],n_rows,n_cols);
%
% Generate the second input
%
if(rand > 0.5)
   input_2='This is a string';
else
   input_2='';
end
%
% Generate the third input
%
input_3=acos(rand(5,1)*1.2);
%
% Call the test_validateattributes function with the above generated input
%
input_1
input_2
input_3
%
test_validateattributes(input_1,input_2,input_3)
```

Voici quelques exemples de mauvaises entrées détectées par la fonction `validateattributes` :

Mauvaise entrée

```
input_1 =

    23    22
    26    28
```

```
input_2 =

     ''

input_3 =

   0.0000 + 0.4455i
   1.2420 + 0.0000i
   0.4063 + 0.0000i
   1.3424 + 0.0000i
   1.2186 + 0.0000i

Error using test_validateattributes (line 44)
Expected input number 1, array_1, to be of size 3x2 when it is actually
size 2x2.
```

Mauvaise entrée

```
input_1 =

    22    24
    21    25
    26    27

input_2 =

This is a string

input_3 =

   1.1371 + 0.0000i
   0.6528 + 0.0000i
   1.0479 + 0.0000i
   0.0000 + 0.1435i
   0.0316 + 0.0000i

Error using test_validateattributes (line 109)
Expected input number 3, array_3, to be real.
```

Entrée valide

```
input_1 =

    20    25
    25    28
    24    23

input_2 =

This is a string

input_3 =

    0.9696
    1.5279
    1.3581
    0.5234
    0.9665
```

```
All the three input are OK
```

Opérateur de commentaire de bloc

Il est recommandé d'ajouter des commentaires décrivant le code. Il est utile pour les autres et même pour le codeur lorsqu'il est renvoyé plus tard. Une seule ligne peut être commentée en utilisant le symbole `%` ou en utilisant le `Ctrl+R` clavier `Ctrl+R` Pour supprimer le commentaire d'une ligne précédemment commentée, supprimez le symbole `%` ou utilisez le raccourci `Crtl+T`

Bien que le commentaire d'un bloc de code puisse être effectué en ajoutant un symbole `%` au début de chaque ligne, les nouvelles versions de MATLAB (après 2015a) vous permettent d'utiliser l' **opérateur de commentaire par bloc** `%{ code %}` . Cet opérateur augmente la lisibilité du code. Il peut être utilisé à la fois pour les commentaires sur le code et pour la documentation d'aide sur les fonctions. Le bloc peut être **plié** et **déplié** pour améliorer la lisibilité du code.

```
1   function y = myFunction(x)
2   %{
3   myFunction  Binary Singleton Expansion Function
4   y = myFunction(x) applies the element-by-element binary operation
5   specified by the function handle FUNC to arrays A and B, with impli
6   expansion enabled.
7   %}
8
9   %% Compute z(x, y) = x.*sin(y) on a grid:
10  % x = 1:10;
11  y = x.';
12
13  %{
14  z = zeros(numel(x),numel(y));
15  for ii=1:numel(x)
16      for jj=1:numel(y)
17          z(ii,jj) = x(ii)*sin(y(jj));
18      end
19  end
20  %}
21
22  z = bsxfun(@(x, y) x.*sin(y), x, y);
23  y = y + z;
24
25  end
```

Comme on peut le voir, les opérateurs `%{` et `%}` doivent apparaître seuls sur les lignes. N'incluez aucun autre texte sur ces lignes.

```
function y = myFunction(x)
%{
myFunction  Binary Singleton Expansion Function
y = myFunction(x) applies the element-by-element binary operation
specified by the function handle FUNC to arrays A and B, with implicit
expansion enabled.
%}

%%  Compute z(x, y) = x.*sin(y) on a grid:
% x = 1:10;
y = x.';

%{
z = zeros(numel(x),numel(y));
for ii=1:numel(x)
    for jj=1:numel(y)
        z(ii,jj) = x(ii)*sin(y(jj));
    end
end
%}

z = bsxfun(@(x, y) x.*sin(y), x, y);
y = y + z;

end
```

Lire Meilleures pratiques MATLAB en ligne: https://riptutorial.com/fr/matlab/topic/2887/meilleures-pratiques-matlab

Chapitre 23: Multithreading

Examples

Utiliser parfor pour paralléliser une boucle

Vous pouvez utiliser `parfor` pour exécuter les itérations d'une boucle en parallèle:

Exemple:

```
poolobj = parpool(2);       % Open a parallel pool with 2 workers

s = 0;                      % Performing some parallel Computations
parfor i=0:9
    s = s + 1;
end
disp(s)                     % Outputs '10'

delete(poolobj);            % Close the parallel pool
```

Remarque: `parfor` ne peut pas être imbriqué directement. Pour `parfor` imbriquer, utilisez une fonction dans `parfor` et ajoutez un second `parfor` dans cette fonction.

Exemple:

```
parfor i = 1:n
[op] = fun_name(ip);
end

function [op] = fun_name(ip)
parfor j = 1:length(ip)
% Some Computation
end
```

Quand utiliser parfor

Fondamentalement, `parfor` est recommandé dans deux cas: beaucoup d'itérations dans votre boucle (comme `1e10`), ou si chaque itération prend beaucoup de temps (par exemple, `eig(magic(1e4))`). Dans le second cas, vous pouvez envisager d'utiliser `spmd` . La raison `parfor` est plus lente qu'une `for` boucle pour les courtes distances ou itérations rapides est la surcharge nécessaire pour gérer correctement tous les travailleurs, plutôt que de faire tout le calcul.

En outre, de nombreuses fonctions `parfor` une fonction multi-threading implicite , ce qui `parfor` boucle `parfor` moins efficace lors de l'utilisation de ces fonctions qu'une boucle série `for` , car tous les cœurs sont déjà utilisés. `parfor` sera en fait un inconvénient dans ce cas, car il est `parfor` la surcharge tout en étant aussi parallèle que la fonction que vous essayez d'utiliser.

Prenons l'exemple suivant pour voir le comportement de `for` par opposition à celui de `parfor` . Ouvrez d'abord le pool parallèle si vous ne l'avez pas déjà fait:

```
gcp; % Opens a parallel pool using your current settings
```

Ensuite, exécutez quelques grandes boucles:

```
n = 1000; % Iteration number
EigenValues = cell(n,1); % Prepare to store the data
Time = zeros(n,1);
for ii = 1:n
tic
    EigenValues{ii,1} = eig(magic(1e3)); % Might want to lower the magic if it takes too long
Time(ii,1) = toc; % Collect time after each iteration
end

figure; % Create a plot of results
plot(1:n,t)
title 'Time per iteration'
ylabel 'Time [s]'
xlabel 'Iteration number[-]';
```

Ensuite, faites la même chose avec `parfor` au lieu de `for` . Vous remarquerez que le temps moyen par itération augmente. `parfor` cependant que le `parfor` utilisé tous les travailleurs disponibles, donc le temps total (`sum(Time)`) doit être divisé par le nombre de cœurs de votre ordinateur.

Ainsi, alors que le temps de faire chaque itération séparée augmente en utilisant `parfor` qui concerne l'utilisation de `for` , le temps total diminue considérablement.

Exécuter des commandes en parallèle à l'aide d'une instruction "Single Program, Multiple Data" (SPMD)

Contrairement à un parallèle for-loop (`parfor`), qui prend les itérations d'une boucle et les répartit entre plusieurs threads, un seul programme, une `spmd` multiple data (`spmd`) prend une série de commandes et les distribue à **tous** les threads, de sorte que chaque thread effectue la commande et stocke les résultats. Considère ceci:

```
poolobj = parpool(2);    % open a parallel pool with two workers

spmd
    q = rand(3);         % each thread generates a unique 3x3 array of random numbers
end

q{1}             % q is called like a cell vector
q{2}             % the values stored in each thread may be accessed by their index

delete(poolobj)  % if the pool is closed, then the data in q will no longer be accessible
```

Il est important de noter que chaque thread est accessible pendant le bloc `spmd` par son index de thread (également appelé index de laboratoire ou `labindex`):

```
poolobj = parpool(2);        % open a parallel pool with two workers

spmd
    q = rand(labindex + 1);  % each thread generates a unique array of random numbers
end
```

```
size(q{1})                  % the size of q{1} is 2x2
size(q{2})                  % the size of q{2} is 3x3

delete(poolobj)             % q is no longer accessible
```

Dans les deux exemples, `q` est un objet composite qui peut être initialisé avec la commande `q = Composite()` . Il est important de noter que les objets composites ne sont accessibles que lorsque le pool est en cours d'exécution.

Utilisation de la commande batch pour effectuer divers calculs en parallèle

Pour utiliser le multithreading dans MATLAB, vous pouvez utiliser la commande `batch` . Notez que la boîte à outils Parallel Computing doit être installée.

Pour un script long, par exemple,

```
for ii=1:1e8
   A(ii)=sin(ii*2*pi/1e8);
end
```

pour l'exécuter en mode batch, utilisez les éléments suivants:

```
job=batch("da")
```

ce qui permet à MATLAB de s'exécuter en mode batch et permet d'utiliser MATLAB entre-temps pour faire d'autres choses, telles que l'ajout de processus par lots.

Pour récupérer les résultats après avoir terminé le travail et charger le tableau `A` dans l'espace de travail:

```
load(job, 'A')
```

Enfin, ouvrez le "moniteur de travail" à partir de *Accueil* → *Environnement* → *Parallèle* → *Surveiller les travaux* et supprimez le travail via:

```
delete(job)
```

Pour charger une fonction pour le traitement par lots, utilisez simplement cette instruction où `fcn` est le nom de la fonction, `N` est le nombre de tableaux de sortie et `x1` , `...` , `xn` sont des tableaux d'entrée:

```
 j=batch(fcn, N, {x1, x2, ..., xn})
```

Lire Multithreading en ligne: https://riptutorial.com/fr/matlab/topic/4378/multithreading

Chapitre 24: Performance et Benchmarking

Remarques

- Le code de profilage est un moyen d'éviter la pratique redoutée de «l' optimisation prématurée » en concentrant le développeur sur les parties du code qui justifient *réellement* les efforts d'optimisation.
- Article de documentation MATLAB intitulé " Mesurer les performances de votre programme ".

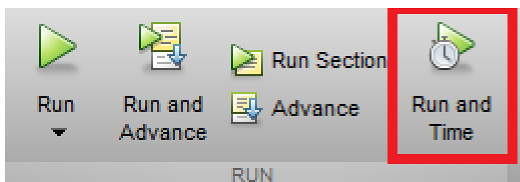
Examples

Identification des goulots d'étranglement des performances à l'aide du profileur

MATLAB Profiler est un outil de profilage logiciel du code MATLAB. En utilisant le profileur, il est possible d'obtenir une représentation visuelle du temps d'exécution et de la consommation de mémoire.

L'exécution du profileur peut se faire de deux manières:

- En cliquant sur le bouton "Run and Time" dans l'interface graphique de MATLAB tout en ayant un fichier `.m` ouvert dans l'éditeur (ajouté dans **R2012b**).



- Par programme, en utilisant:

```
profile on
<some code we want to test>
profile off
```

Voici un exemple de code et le résultat de son profilage:

```
function docTest

for ind1 = 1:100
  [~] = var(...
            sum(...
                randn(1000)));
end

spy
```

Profiler

File Edit Debug Window Help

Start Profiling Run this code: Profile time: 2 s

Profile Summary

Generated 21-Jul-2016 18:07:13 using performance time.

Function Name	Calls	**Total Time**	Self Time*	Total Time Plot (dark band = self time)
docTest	1	1.997 s	1.469 s	
spy	1	0.504 s	0.231 s	
newplot	2	0.251 s	0.219 s	
newplot>ObserveAxesNextPlot	2	0.028 s	0.004 s	
var	100	0.024 s	0.024 s	
cla	2	0.024 s	0.018 s	
xlabel	1	0.019 s	0.019 s	
graphics\private\clo	2	0.005 s	0.005 s	
gobjects	4	0.004 s	0.004 s	
spy>defaultspy	1	0.002 s	0.002 s	
graphics\private\claNotify	2	0.001 s	0.001 s	
ishold	2	0.001 s	0.001 s	
newplot>ObserveFigureNextPlot	2	0.001 s	0.001 s	
graph2d\private\labelcheck	1	0.000 s	0.000 s	
int2str	1	0.000 s	0.000 s	

Self time is the time spent in a function excluding the time spent in its child functions. Self time also includes overhead resulting from the process of profiling.

De ce qui précède, nous apprenons que la fonction d' `spy` prend environ 25% du temps d'exécution total. Dans le cas du "code réel", une fonction qui prend un tel temps d'exécution serait un bon candidat pour l'optimisation, par opposition aux fonctions analogues à `var` et `cla` dont l'optimisation devrait être évitée.

De plus, il est possible de cliquer sur les entrées de la colonne *Nom de* la *fonction* pour voir une ventilation détaillée du temps d'exécution de cette entrée. Voici l'exemple de cliquer sur `spy` :

Profiler

File Edit Debug Window Help

Start Profiling Run this code: Profile time: 2 s

spy (Calls: 1, Time: 0.504 s)

Generated 21-Jul-2016 18:36:32 using performance time.
function in file E:\Program Files\MATLAB\R2016a\toolbox\matlab\sparfun\spy.m
Copy to new window for comparing multiple runs

Refresh

☑ Show parent functions ☑ Show busy lines ☑ Show child functions
☑ Show Code Analyzer results ☑ Show file coverage ☑ Show function listing

Parents (calling functions)

Function Name	Function Type	Calls
docTest	function	1

Lines where the most time was spent

Line Number	Code	Calls	Total Time	% Time	Time Plot
17	`cax = newplot;`	1	0.257 s	51.1%	
53	`pos = get(gca,'position');`	1	0.196 s	38.8%	
64	`xlabel(['nz = ' int2str(nnz(S)...`	1	0.025 s	5.0%	
62	`'linestyle',linestyle,'color',...`	1	0.019 s	3.7%	
45	`if nargin < 1, S = defaults...`	1	0.002 s	0.3%	
All other lines			0.005 s	1.0%	
Totals			0.504 s	100%	

Il est également possible de profiler la consommation de mémoire en exécutant `profile('-memory')` avant d'exécuter le profileur.

Profiler

File Edit Debug Window Help

Start Profiling Run this code:

spy (Calls: 1, Time: 0.282 s, 9316.00 Kb, 0.00 Kb, 9036.00 Kb)

Generated 21-Jul-2016 18:51:42 using performance time.

function in file E:\Program Files\MATLAB\R2016a\toolbox\matlab\sparfun\spy.m

Copy to new window for comparing multiple runs

Refresh

☐ Show parent functions ☑ Show busy lines ☐ Show child functions

☐ Show Code Analyzer results ☐ Show file coverage ☐ Show function listing

Sort busy lines and graph according to time

Lines where the most time was spent

Line Number	Code	Calls	Total Time	Allocated Memory	Freed Memory	Peak Memory	% Time
64	`xlabel(['nz = ' int2str(nnz(S)...`	1	0.222 s	9036.00 Kb	0.00 Kb	9036.00 Kb	78.9%
17	`cax = newplot;`	1	0.047 s	280.00 Kb	0.00 Kb	280.00 Kb	16.8%
62	`'linestyle',linestyle,'color',...`	1	0.007 s	0.00 Kb	0.00 Kb	0.00 Kb	2.4%
53	`pos = get(gca,'position');`	1	0.002 s	0.00 Kb	0.00 Kb	0.00 Kb	0.8%
45	`if nargin < 1, S = defaults...`	1	0.001 s	0.00 Kb	0.00 Kb	0.00 Kb	0.3%
All other lines			0.003 s	0.00 Kb	0.00 Kb	9036.00 Kb	0.9%
Totals			0.282 s	9316.00 Kb	0.00 Kb	9036.00 Kb	100%

Comparer le temps d'exécution de plusieurs fonctions

La combinaison très répandue de `tic` et de `toc` peut donner une idée approximative du temps d'exécution d'une fonction ou d'extraits de code.

Pour comparer plusieurs fonctions, il ne faut pas l'utiliser. Pourquoi? Il est presque impossible de fournir *des conditions égales* pour tous les extraits de code à comparer dans un script utilisant la solution ci-dessus. Les fonctions partagent peut-être le même espace de fonction et les mêmes variables communes, de sorte que les fonctions appelées et les extraits de code ultérieurs tirent déjà parti des variables et fonctions précédemment initialisées. En outre, il n'y a pas d'aperçu si le compilateur JIT traiterait ces fragments appelés ultérieurement de la même manière.

La fonction dédiée aux benchmarks est le `timeit` . L'exemple suivant illustre son utilisation.

Il y a le tableau `A` et la matrice `B` Il convient de déterminer quelle ligne de `B` est la plus similaire à `A`

en comptant le nombre d'éléments différents.

```
function t = bench()
    A = [0 1 1 1 0 0];
    B = perms(A);

    % functions to compare
    fcns = {
        @() compare1(A,B);
        @() compare2(A,B);
        @() compare3(A,B);
        @() compare4(A,B);
    };

    % timeit
    t = cellfun(@timeit, fcns);
end

function Z = compare1(A,B)
    Z = sum(  bsxfun(@eq,  A,B) , 2);
end
function Z = compare2(A,B)
    Z = sum(bsxfun(@xor, A, B),2);
end
function Z = compare3(A,B)
    A = logical(A);
    Z = sum(B(:,~A),2) + sum(~B(:,A),2);
end
function Z = compare4(A,B)
     Z = pdist2( A, B, 'hamming', 'Smallest', 1 );
end
```

Ce mode de référence a été vu pour la première fois dans cette réponse .

C'est bien d'être célibataire!

Aperçu:

Le type de données par défaut pour les tableaux numériques dans MATLAB est `double` . `double` est une représentation à virgule flottante des nombres , et ce format prend 8 octets (ou 64 bits) par valeur. Dans **certains** cas, où, par exemple, ne traiter que des nombres entiers ou lorsque l’instabilité numérique n’est pas un problème imminent, une telle résolution peut ne pas être requise. Pour cette raison, il est conseillé de prendre en compte les avantages d'une `single` précision (ou d'autres types appropriés):

- Temps d'exécution plus rapide (particulièrement visible sur les GPU).
- La moitié de la consommation de mémoire: peut réussir lorsque le `double` échoue en raison d'une erreur de mémoire insuffisante; plus compact lors du stockage en fichiers.

La conversion d' une variable à partir de tout type de données pris en charge à `single` est effectuée en utilisant:

```
sing_var = single(var);
```

Certaines fonctions couramment utilisées (telles que les `zeros` , les `eye` , les `ones` , etc.) qui génèrent `double` valeurs `double` par défaut permettent de spécifier le type / la classe de la sortie.

Conversion de variables dans un script en une précision / type / classe autre que celle par défaut:

Depuis juillet 2016, il n'existe aucun moyen documenté de modifier le type de données MATLAB *par défaut* du `double` .

Dans MATLAB, les nouvelles variables imitent généralement les types de données des variables utilisées lors de leur création. Pour illustrer cela, considérons l'exemple suivant:

```
A = magic(3);
B = diag(A);
C = 20*B;
>> whos C
  Name      Size            Bytes  Class     Attributes
  C         3x1                24  double
```

```
A = single(magic(3)); % A is converted to "single"
B = diag(A);
C = B*double(20);     % The stricter type, which in this case is "single", prevails
D = single(size(C));  % It is generally advised to cast to the desired type explicitly.
>> whos C
  Name      Size            Bytes  Class     Attributes
  C         3x1                12  single
```

Ainsi, il peut sembler suffisant de lancer / convertir plusieurs variables initiales pour que le changement imprègne le code, mais cela est **déconseillé** (voir *Mises en garde et pièges* ci-dessous).

Mises en garde et pièges:

1. Les conversions répétées sont **déconseillées en** raison de l'introduction de bruit numérique (lors du passage d'un `single` à un `double`) ou de la perte d'informations (lors du passage du `double` au `single` ou entre certains types d'entiers), par exemple:

   ```
   double(single(1.2)) == double(1.2)
   ans =
        0
   ```

   Cela peut être atténué en utilisant `typecast` . Voir aussi Soyez conscient de l'imprécision en virgule flottante .

2. Le recours exclusif au typage implicite des données (c.-à-d. Ce que MATLAB suppose que

le type de sortie d'un calcul devrait être) est **déconseillé en** raison de plusieurs effets indésirables qui pourraient survenir:

- *Perte d'information* : lorsqu'un `double` résultat est attendu, mais qu'une combinaison négligente d'opérandes `single` et `double` produit `single` précision `single` .
- *Consommation de mémoire élevée de manière inattendue* : lorsqu'un `single` résultat est attendu mais qu'un calcul imprudent entraîne une `double` sortie.
- *Une surcharge inutile lors de l'utilisation de GPU* : lors du mélange de types `gpuArray` (c'est-à-dire les variables stockées dans VRAM) avec des variables non `gpuArray` (c'est-à-dire *généralement* stockées dans la RAM), les données devront être transférées avant que le calcul puisse être effectué. Cette opération prend du temps et peut être très visible dans les calculs répétitifs.
- *Erreurs lors du mélange de types à virgule flottante avec des types entiers* : les fonctions telles que `mtimes` (`*`) ne sont pas définies pour les entrées mixtes de types entiers et à virgule flottante - et seront des erreurs. Les fonctions comme les `times` (`.*`) Ne sont pas du tout définies pour les entrées de type entier - et seront à nouveau erronées.

```
>> ones(3,3,'int32')*ones(3,3,'int32')
Error using  *
MTIMES is not fully supported for integer classes. At least one input must be
scalar.

>> ones(3,3,'int32').*ones(3,3,'double')
Error using  .*
Integers can only be combined with integers of the same class, or scalar doubles.
```

Pour une meilleure lisibilité du code et un risque réduit de types indésirables, une approche défensive est **conseillée** , dans laquelle les variables sont *explicitement* converties au type souhaité.

Voir également:

- Documentation MATLAB: Nombres à virgule flottante .
- Article technique de Mathworks: Meilleures pratiques pour convertir le code MATLAB en points fixes .

réorganiser un tableau ND peut améliorer les performances globales

Dans certains cas, nous devons appliquer des fonctions à un ensemble de tableaux ND. Regardons cet exemple simple.

```
A(:,:,1) = [1 2; 4 5];
A(:,:,2) = [11 22; 44 55];
B(:,:,1) = [7 8; 1 2];
B(:,:,2) = [77 88; 11 22];
```

```
A =

ans(:,:,1) =

   1   2
   4   5

ans(:,:,2) =

   11   22
   44   55

>> B
B =

ans(:,:,1) =

   7   8
   1   2

ans(:,:,2) =

   77   88
   11   22
```

Les deux matrices sont en 3D, disons que nous devons calculer ce qui suit:

```
result= zeros(2,2);
...
for k = 1:2
   result(i,j) = result(i,j) + abs( A(i,j,k) - B(i,j,k) );
...

if k is very large, this for-loop can be a bottleneck since MATLAB order the data in a column
major fashion. So a better way to compute "result" could be:

% trying to exploit the column major ordering
Aprime = reshape(permute(A,[3,1,2]), [2,4]);
Bprime = reshape(permute(B,[3,1,2]), [2,4]);


>> Aprime
Aprime =

    1    4    2    5
   11   44   22   55

>> Bprime
Bprime =

    7    1    8    2
   77   11   88   22
```

Maintenant, nous remplaçons la boucle ci-dessus comme suit:

```
result= zeros(2,2);
....
temp = abs(Aprime - Bprime);
```

```
for k = 1:2
    result(i,j) = result(i,j) + temp(k, i+2*(j-1));
...
```

Nous avons réorganisé les données pour pouvoir exploiter la mémoire cache. La permutation et la remise en forme peuvent être coûteuses, mais lorsque vous travaillez avec des baies ND volumineuses, le coût de calcul lié à ces opérations est bien inférieur à celui des baies non organisées.

L'importance de la préallocation

Les tableaux dans MATLAB sont des blocs continus en mémoire, alloués et libérés automatiquement par MATLAB. MATLAB masque les opérations de gestion de la mémoire telles que le redimensionnement d'un tableau derrière une syntaxe facile à utiliser:

```
a = 1:4

a =

     1     2     3     4

a(5) = 10  % or alternatively a = [a, 10]

a =

     1     2     3     4    10
```

Il est important de comprendre que ce qui précède n'est pas une opération triviale, `a(5) = 10` obligera MATLAB à allouer un nouveau bloc de mémoire de taille 5, à copier les 4 premiers numéros et à définir le 5 à 10. C'est une opération `O(numel(a))` , et non pas `O(1)` .

Considérer ce qui suit:

```
clear all
n=12345678;
a=0;
tic
for i = 2:n
    a(i) = sqrt(a(i-1)) + i;
end
toc

Elapsed time is 3.004213 seconds.
```

`a` est réallouée `n` fois dans cette boucle (à l'exclusion de certaines optimisations entreprises par MATLAB)! Notez que MATLAB nous avertit:

> "La variable 'a' semble changer de taille à chaque itération de boucle. Envisagez de pré-allouer la vitesse."

Que se passe-t-il lorsque nous pré-allouons?

```
a=zeros(1,n);
tic
for i = 2:n
    a(i) = sqrt(a(i-1)) + i;
end
toc

Elapsed time is 0.410531 seconds.
```

Nous pouvons voir que le temps d'exécution est réduit d'un ordre de grandeur.

Méthodes de préallocation:

MATLAB fournit diverses fonctions pour l'allocation de vecteurs et de matrices, en fonction des besoins spécifiques de l'utilisateur. Ceux-ci incluent: les `zeros` , les `ones` , les `nan` , l' `eye` , `true` etc.

```
a = zeros(3)       % Allocates a 3-by-3 matrix initialized to 0
a =

     0     0     0
     0     0     0
     0     0     0

a = zeros(3, 2)     % Allocates a 3-by-2 matrix initialized to 0
a =

     0     0
     0     0
     0     0

a = ones(2, 3, 2)      % Allocates a 3 dimensional array (2-by-3-by-2) initialized to 1
a(:,:,1) =

     1     1     1
     1     1     1


a(:,:,2) =

     1     1     1
     1     1     1

a = ones(1, 3) * 7  % Allocates a row vector of length 3 initialized to 7
a =

     7     7     7
```

Un type de données peut également être spécifié:

```
a = zeros(2, 1, 'uint8');  % allocates an array of type uint8
```

Il est également facile de cloner la taille d'un tableau existant:

```
a = ones(3, 4);       % a is a 3-by-4 matrix of 1's
b = zeros(size(a));  % b is a 3-by-4 matrix of 0's
```

Et cloner le type:

```
a = ones(3, 4, 'single');        % a is a 3-by-4 matrix of type single
b = zeros(2, 'like', a);         % b is a 2-by-2 matrix of type single
```

Notez que "like" clone également la *complexité* et la *rareté* .

La préallocation est implicitement obtenue en utilisant n'importe quelle fonction qui renvoie un tableau de la taille finale requise, tel que `rand` , `gallery` , `kron` , `bsxfun` , `colon` et bien d'autres. Par exemple, un moyen courant d'allouer des vecteurs avec des éléments variant de façon linéaire consiste à utiliser l'opérateur deux-points (avec la variante [1] ou 2 opérandes):

```
a = 1:3
a =

     1     2     3

a = 2:-3:-4
a =

     2    -1    -4
```

Les tableaux de cellules peuvent être alloués à l'aide de la fonction `cell()` de la même manière que les `zeros()` .

```
a = cell(2,3)
a =

    []    []    []
    []    []    []
```

Notez que les tableaux de cellules fonctionnent en conservant des pointeurs vers les emplacements en mémoire du contenu de la cellule. Ainsi, toutes les astuces de préallocation s'appliquent également aux éléments individuels du tableau de cellules.

Lectures complémentaires:

- Documentation officielle MATLAB sur " **Mémoire préallouée** ".
- Documentation officielle MATLAB sur " **Comment MATLAB alloue la mémoire** ".
- Performances de préallocation sur **matlab non documenté** .
- Comprendre la préallocation de matrice sur **Loren sur l'art de MATLAB**

Lire Performance et Benchmarking en ligne: https://riptutorial.com/fr/matlab/topic/1141/performance-et-benchmarking

Chapitre 25: Pour les boucles

Remarques

Itérer sur le vecteur de colonne

Une source commune de bogues est d'essayer de faire une boucle sur les éléments d'un vecteur de colonne. Un vecteur de colonne est traité comme une matrice avec une colonne. (Il n'y a en fait aucune distinction dans Matlab.) La boucle `for` s'exécute une fois avec la variable de boucle définie sur la colonne.

```
% Prints once: [3, 1]
my_vector = [1; 2; 3];
for i = my_vector
    display(size(i))
end
```

Modifier la variable d'itération

La modification de la variable d'itération modifie sa valeur pour l'itération en cours, mais n'a aucune incidence sur sa valeur dans les itérations suivantes.

```
% Prints 1, 2, 3, 4, 5
for i = 1:5
    display(i)
    i = 5; % Fail at trying to terminate the loop
end
```

Performance du cas particulier d' `a:b` dans le côté droit

L'exemple de base traite `1:n` comme une instance normale de création d'un vecteur de ligne, puis itère dessus. Pour des raisons de performances, Matlab traite en fait tout `a:b` ou `a:c:b` particulier en ne créant pas entièrement le vecteur de ligne, mais en créant chaque élément un par un.

Cela peut être détecté en modifiant légèrement la syntaxe.

```
% Loops forever
for i = 1:1e50
end
```

```
% Crashes immediately
for i = [1:1e50]
```

```
end
```

Examples

Boucle 1 à n

Le cas le plus simple consiste simplement à préparer une tâche pour un nombre de fois déterminé. Disons que nous voulons afficher les nombres entre 1 et n, nous pouvons écrire:

```
n = 5;
for k = 1:n
    display(k)
end
```

La boucle exécutera les instructions internes, tout entre le `for` et le `end` , pour `n` fois (5 dans cet exemple):

```
1
2
3
4
5
```

Voici un autre exemple:

```
n = 5;
for k = 1:n
    disp(n-k+1:-1:1) % DISP uses more "clean" way to print on the screen
end
```

Cette fois, nous utilisons à la fois le `n` et le `k` dans la boucle pour créer un affichage "imbriqué":

```
 5     4     3     2     1

 4     3     2     1

 3     2     1

 2     1

 1
```

Itérer sur des éléments de vecteur

Le côté droit de l'affectation dans une boucle `for` peut être un vecteur de ligne. Le côté gauche de l'affectation peut être n'importe quel nom de variable valide. La boucle `for` assigne un élément différent de ce vecteur à la variable à chaque exécution.

```
other_row_vector = [4, 3, 5, 1, 2];
for any_name = other_row_vector
```

```
    display(any_name)
end
```

La sortie afficherait

```
4
3
5
1
2
```

(La version `1:n` est un cas normal, car dans Matlab `1:n` est juste la syntaxe pour construire un vecteur de ligne de `[1, 2, ..., n]` .)

Par conséquent, les deux blocs de code suivants sont identiques:

```
A = [1 2 3 4 5];
for x = A
  disp(x);
end
```

et

```
for x = 1:5
  disp(x);
end
```

Et ce qui suit sont identiques:

```
A = [1 3 5 7 9];
for x = A
  disp(x);
end
```

et

```
for x = 1:2:9
  disp(x);
end
```

Tout vecteur de ligne fera l'affaire. Ils n'ont pas besoin d'être des chiffres.

```
my_characters = 'abcde';
for my_char = my_characters
    disp(my_char)
end
```

va sortir

```
a
b
```

```
c
d
e
```

Itérer sur des colonnes de matrice

Si la partie droite de l'affectation est une matrice, la colonne se voit attribuer dans chaque itération les colonnes suivantes de cette matrice.

```
some_matrix = [1, 2, 3; 4, 5, 6]; % 2 by 3 matrix
for some_column = some_matrix
    display(some_column)
end
```

(La version vectorielle de ligne est un cas normal car, dans Matlab, un vecteur de ligne est simplement une matrice dont les colonnes sont de taille 1.)

La sortie afficherait

```
1
4
2
5
3
6
```

c'est-à-dire chaque colonne de la matrice itérée affichée, chaque colonne imprimée à chaque appel d' `display` .

Boucle sur les index

```
my_vector = [0, 2, 1, 3, 9];
for i = 1:numel(my_vector)
    my_vector(i) = my_vector(i) + 1;
end
```

Les opérations les plus simples réalisées avec `for` boucles peuvent être effectuées plus rapidement et plus facilement grâce aux opérations vectorisées. Par exemple, la boucle ci-dessus peut être remplacée par `my_vector = my_vector + 1` .

Boucles imbriquées

Les boucles peuvent être imbriquées pour préformer une tâche itérée dans une autre tâche itérée. Considérons les boucles suivantes:

```
ch = 'abc';
m = 3;
for c = ch
    for k = 1:m
        disp([c num2str(k)]) % NUM2STR converts the number stored in k to a charachter,
                             % so it can be concataneted with the letter in c
```

```
    end
end
```

nous utilisons 2 itérateurs pour afficher toutes les combinaisons d'éléments `abc` et `1:m` , ce qui donne:

```
a1
a2
a3
b1
b2
b3
c1
c2
c3
```

Nous pouvons également utiliser des boucles imbriquées pour combiner des tâches à effectuer à chaque fois et des tâches à effectuer une fois dans plusieurs itérations:

```
N = 10;
n = 3;
a1 = 0; % the first element in Fibonacci series
a2 = 1; % the secound element in Fibonacci series
for j = 1:N
    for k = 1:n
        an = a1 + a2; % compute the next element in Fibonacci series
        a1 = a2;      % save the previous element for the next iteration
        a2 = an;      % save ht new element for the next iteration
    end
    disp(an) % display every n'th element
end
```

Nous voulons ici pour calculer toutes les séries de Fibonacci , mais pour afficher uniquement le `n` ième élément à chaque fois, donc nous obtenons

```
   3
   13
   55
   233
   987
   4181
   17711
   75025
   317811
   1346269
```

Une autre chose que nous pouvons faire est d'utiliser le premier itérateur (externe) dans la boucle interne. Voici un autre exemple:

```
N = 12;
gap = [1 2 3 4 6];
for j = gap
    for k = 1:j:N
        fprintf('%d ',k) % FPRINTF prints the number k proceeding to the next the line
    end
```

```
    fprintf('\n')        % go to the next line
end
```

Cette fois, nous utilisons la boucle imbriquée pour formater la sortie et freiner la ligne uniquement lorsqu'un nouvel espace (`j`) entre les éléments a été introduit. Nous parcourons la largeur de l'intervalle dans la boucle externe et l'utilisons dans la boucle interne pour parcourir le vecteur:

```
1 2 3 4 5 6 7 8 9 10 11 12
1 3 5 7 9 11
1 4 7 10
1 5 9
1 7
```

Remarque: Bizarrement, les mêmes boucles imbriquées.

Ce n'est pas quelque chose que vous verrez dans d'autres environnements de programmation. Je l'ai rencontré il y a quelques années et je ne comprenais pas pourquoi cela se passait, mais après avoir travaillé avec MATLAB pendant un certain temps, j'ai pu le comprendre. Regardez l'extrait de code ci-dessous:

```
for x = 1:10
    for x = 1:10
        fprintf('%d,', x);
    end
    fprintf('\n');
end
```

vous ne vous attendriez pas à ce que cela fonctionne correctement, mais en produisant la sortie suivante:

```
1,2,3,4,5,6,7,8,9,10,
1,2,3,4,5,6,7,8,9,10,
1,2,3,4,5,6,7,8,9,10,
1,2,3,4,5,6,7,8,9,10,
1,2,3,4,5,6,7,8,9,10,
1,2,3,4,5,6,7,8,9,10,
1,2,3,4,5,6,7,8,9,10,
1,2,3,4,5,6,7,8,9,10,
1,2,3,4,5,6,7,8,9,10,
1,2,3,4,5,6,7,8,9,10,
```

La raison en est que, comme avec tout le reste dans MATLAB, le compteur `x` est également une matrice - un vecteur pour être précis. En tant que tel, `x` n'est qu'une référence à un «tableau» (une structure de mémoire consécutive cohérente) qui est correctement référencé avec chaque boucle conséquente (imbriquée ou non). Le fait que la boucle imbriquée utilise le même identifiant ne fait aucune différence sur la façon dont les valeurs de ce tableau sont référencées. Le seul problème est que, dans la boucle imbriquée, le `x` externe est masqué par le `x` imbriqué (local) et ne peut donc pas être référencé. Cependant, la fonctionnalité de la structure de boucle imbriquée reste intacte.

Lire Pour les boucles en ligne: https://riptutorial.com/fr/matlab/topic/1927/pour-les-boucles

Chapitre 26: Programmation orientée objet

Examples

Définir une classe

Une classe peut être définie en utilisant `classdef` dans un fichier `.m` portant le même nom que la classe. Le fichier peut contenir le bloc `classdef` ... `end` et les fonctions locales à utiliser dans les méthodes de classe.

La définition de classe MATLAB la plus générale a la structure suivante:

```
classdef (ClassAttribute = expression, ...) ClassName < ParentClass1 & ParentClass2 & ...

   properties (PropertyAttributes)
      PropertyName
   end

   methods (MethodAttributes)
      function obj = methodName(obj,arg2,...)
         ...
      end
   end

   events (EventAttributes)
      EventName
   end

   enumeration
      EnumName
   end

end
```

Documentation MATLAB: attributs de classe, attributs de propriété, attributs de méthode, attributs d' événement , restrictions de classe d'énumération .

Exemple de classe:

Une classe appelée `Car` peut être définie dans le fichier `Car.m` comme

```
classdef Car < handle % handle class so properties persist
    properties
        make
        model
        mileage = 0;
    end

    methods
        function obj = Car(make, model)
            obj.make = make;
            obj.model = model;
```

```
        end
        function drive(obj, milesDriven)
            obj.mileage = obj.mileage + milesDriven;
        end
    end
end
```

Notez que le constructeur est une méthode avec le même nom que la classe. <Un constructeur est une méthode spéciale d'une classe ou d'une structure dans la programmation orientée objet qui initialise un objet de ce type. Un constructeur est une méthode d'instance qui porte généralement le même nom que la classe et peut être utilisée pour définir les valeurs des membres d'un objet, soit par défaut, soit en fonction de valeurs définies par l'utilisateur.

Une instance de cette classe peut être créée en appelant le constructeur;

```
>> myCar = Car('Ford', 'Mustang'); //creating an instance of car class
```

L' appel de la `drive` méthode incrémenter le kilométrage

```
>> myCar.mileage

    ans =
            0

>> myCar.drive(450);

>> myCar.mileage

   ans =
            450
```

Classes valeur vs poignée

Les classes dans MATLAB sont divisées en deux catégories principales: les classes de valeur et les classes de handle. La principale différence est que lors de la copie d'une instance d'une classe de valeur, les données sous-jacentes sont copiées dans la nouvelle instance, tandis que pour les classes de descripteurs, la nouvelle instance pointe sur les données d'origine. Une classe peut être définie comme un handle en héritant de la classe de `handle` .

```
classdef valueClass
    properties
        data
    end
end
```

et

```
classdef handleClass < handle
    properties
        data
    end
end
```

puis

```
>> v1 = valueClass;
>> v1.data = 5;
>> v2 = v1;
>> v2.data = 7;
>> v1.data
ans =
     5

>> h1 = handleClass;
>> h1.data = 5;
>> h2 = h1;
>> h2.data = 7;
>> h1.data
ans =
     7
```

Héritage des classes et des classes abstraites

Déni de responsabilité: les exemples présentés ici ont uniquement pour but de montrer l'utilisation de classes abstraites et de l'héritage et ne sont pas nécessairement utiles. En outre, il n'y a aucune chose aussi polymorphe dans MATLAB et par conséquent l'utilisation de classes abstraites est limitée. Cet exemple montre qui doit créer une classe, hérite d'une autre classe et applique une classe abstraite pour définir une interface commune.

L'utilisation de classes abstraites est plutôt limitée dans MATLAB, mais elle peut toujours être utile à quelques reprises.

Disons que nous voulons un enregistreur de messages. Nous pourrions créer une classe similaire à celle ci-dessous:

```
classdef ScreenLogger
    properties(Access=protected)
        scrh;
    end

    methods
        function obj = ScreenLogger(screenhandler)
            obj.scrh = screenhandler;
        end

        function LogMessage(obj, varargin)
            if ~isempty(varargin)
                varargin{1} = num2str(varargin{1});
                fprintf(obj.scrh, '%s\n', sprintf(varargin{:}));
            end
        end
    end
end
```

Propriétés et méthodes

En résumé, les propriétés contiennent un état d'objet tandis que les méthodes sont comme une interface et définissent des actions sur des objets.

La propriété `scrh` est protégée. C'est pourquoi il doit être initialisé dans un constructeur. Il existe d'autres méthodes (getters) pour accéder à cette propriété, mais cela ne correspond pas à cet exemple. Les propriétés et les méthodes peuvent être accessibles via une variable qui contient une référence à un objet en utilisant la notation par points suivie du nom d'une méthode ou d'une propriété:

```
mylogger = ScreenLogger(1);                         % OK
mylogger.LogMessage('My %s %d message', 'very', 1); % OK
mylogger.scrh = 2;                                  % ERROR!!! Access denied
```

Les propriétés et les méthodes peuvent être publiques, privées ou protégées. Dans ce cas, protected signifie que je pourrai accéder à `scrh` partir d'une classe héritée mais pas de l'extérieur. Par défaut, toutes les propriétés et méthodes sont publiques. Par conséquent, `LogMessage()` peut être librement utilisé en dehors de la définition de classe. En outre, `LogMessage` définit une interface, ce qui signifie que nous devons appeler ce message lorsque nous voulons qu'un objet enregistre nos messages personnalisés.

Application

Disons que j'ai un script où j'utilise mon enregistreur:

```
clc;
% ... a code
logger = ScreenLogger(1);
% ... a code
logger.LogMessage('something');
% ... a code
logger.LogMessage('something');
% ... a code
logger.LogMessage('something');
% ... a code
logger.LogMessage('something');
```

Si j'ai plusieurs endroits où j'utilise le même enregistreur et que je veux ensuite le transformer en quelque chose de plus sophistiqué, comme écrire un message dans un fichier, je devrai créer un autre objet:

```
classdef DeepLogger
    properties(SetAccess=protected)
        FileName
    end
    methods
        function obj = DeepLogger(filename)
            obj.FileName = filename;
        end

        function LogMessage(obj, varargin)
            if ~isempty(varargin)
                varargin{1} = num2str(varargin{1});
                fid = fopen(obj.fullfname, 'a+t');
                fprintf(fid, '%s\n', sprintf(varargin{:}));
                fclose(fid);
            end
        end
```

```
    end
end
```

et changez simplement une ligne de code en ceci:

```
clc;
% ... a code
logger = DeepLogger('mymessages.log');
```

La méthode ci-dessus ouvre simplement un fichier, ajoute un message à la fin du fichier et le ferme. Pour le moment, pour être cohérent avec mon interface, je dois me souvenir que le nom d'une méthode est `LogMessage()` mais que cela pourrait également être tout autre chose. MATLAB peut forcer le développeur à conserver le même nom en utilisant des classes abstraites. Disons que nous définissons une interface commune pour tout enregistreur:

```
classdef MessageLogger
    methods(Abstract=true)
        LogMessage(obj, varargin);
    end
end
```

Maintenant, si `ScreenLogger` et `DeepLogger` héritent tous deux de cette classe, MATLAB générera une erreur si `LogMessage()` n'est pas défini. Les classes abstraites aident à créer des classes similaires pouvant utiliser la même interface.

Pour cette raison, je vais faire un changement légèrement différent. Je vais supposer que DeepLogger fera les deux messages de journalisation sur un écran et dans un fichier en même temps. Étant `ScreenLogger` que `ScreenLogger` enregistre déjà les messages à l'écran, je vais hériter de `DeepLogger` de `ScreenLoggger` pour éviter les répétitions. `ScreenLogger` ne change pas du tout en dehors de la première ligne:

```
classdef ScreenLogger < MessageLogger
// the rest of previous code
```

Cependant, `DeepLogger` besoin de plus de modifications dans la méthode `LogMessage` :

```
classdef DeepLogger < MessageLogger & ScreenLogger
    properties(SetAccess=protected)
        FileName
        Path
    end
    methods
        function obj = DeepLogger(screenhandler, filename)
            [path,filen,ext] = fileparts(filename);
            obj.FileName = [filen ext];
            pbj.Path     = pathn;
            obj = obj@ScreenLogger(screenhandler);
        end
        function LogMessage(obj, varargin)
            if ~isempty(varargin)
                varargin{1} = num2str(varargin{1});
                LogMessage@ScreenLogger(obj, varargin{:});
```

```
                fid = fopen(obj.fullfname, 'a+t');
                fprintf(fid, '%s\n', sprintf(varargin{:}));
                fclose(fid);
            end
        end
    end
end
```

Tout d'abord, j'initialise simplement les propriétés dans le constructeur. Deuxièmement, comme cette classe hérite de `ScreenLogger` je dois également initialiser cet objet parrent. Cette ligne est d'autant plus importante que le constructeur de `ScreenLogger` nécessite un paramètre pour l'initialisation de son propre objet. Cette ligne:

```
obj = obj@ScreenLogger(screenhandler);
```

il suffit de dire "Appelle le consortium de ScreenLogger et l'initialise avec un gestionnaire d'écran". Il convient de noter ici que j'ai défini `scrh` comme protégé. Par conséquent, je pourrais également accéder à cette propriété à partir de `DeepLogger` . Si la propriété a été définie comme privée. La seule façon de l'initialiser serait d'utiliser le consœur.

Un autre changement concerne les `methods` section. Pour éviter la répétition, j'appelle `LogMessage()` depuis une classe parente pour enregistrer un message sur un écran. Si je devais changer quoi que ce soit pour améliorer la journalisation de l'écran, je dois maintenant le faire au même endroit. Le code de repos est identique à celui de `DeepLogger` .

Parce que cette classe hérite également d'une classe abstraite `MessageLogger` je devais m'assurer que `LogMessage()` dans `DeepLogger` était également défini. Hériter de `MessageLogger` est un peu difficile ici. Je pense que cela rend la redéfinition de `LogMessage` obligatoire - à mon avis.

En ce qui concerne le code où un enregistreur est appliqué, grâce à une interface commune dans les classes, je peux assurer que cette ligne dans le code entier ne posera aucun problème. Les mêmes messages seront enregistrés à l'écran, mais le code écrira ces messages dans un fichier.

```
clc;
% ... a code
logger = DeepLogger(1, 'mylogfile.log');
% ... a code
logger.LogMessage('something');
% ... a code
logger.LogMessage('something');
% ... a code
logger.LogMessage('something');
% ... a code
logger.LogMessage('something');
```

J'espère que ces exemples expliquent l'utilisation des classes, l'utilisation de l'héritage et l'utilisation de classes abstraites.

PS La solution au problème ci-dessus est l'une des nombreuses. Une autre solution, moins complexe, consisterait à faire de `ScreenLoger` un composant d'un autre enregistreur comme `FileLogger`

etc. `ScreenLogger` serait conservé dans l'une des propriétés. Son `LogMessage` appelle simplement `LogMessage` du `ScreenLogger` et affiche le texte sur un écran. J'ai choisi une approche plus complexe pour montrer comment les classes fonctionnent dans MATLAB. L'exemple de code ci-dessous:

```
classdef DeepLogger < MessageLogger
    properties(SetAccess=protected)
        FileName
        Path
        ScrLogger
    end
    methods
        function obj = DeepLogger(screenhandler, filename)
            [path,filen,ext] = fileparts(filename);
            obj.FileName     = [filen ext];
            obj.Path         = pathn;
            obj.ScrLogger    = ScreenLogger(screenhandler);
        end
        function LogMessage(obj, varargin)
            if ~isempty(varargin)
                varargin{1} = num2str(varargin{1});
                obj.LogMessage(obj.ScrLogger, varargin{:}); % <-------- thechange here
                fid = fopen(obj.fullfname, 'a+t');
                fprintf(fid, '%s\n', sprintf(varargin{:}));
                fclose(fid);
            end
        end
    end
end
```

Constructeurs

Un constructeur est une méthode spéciale dans une classe appelée lorsqu'une instance d'un objet est créée. C'est une fonction MATLAB standard qui accepte les paramètres d'entrée, mais elle doit également respecter certaines règles .

Les constructeurs ne sont pas requis car MATLAB en crée un par défaut. En pratique, cependant, il s'agit d'un endroit pour définir un état d'un objet. Par exemple, les propriétés peuvent être restreintes en spécifiant des attributs . Ensuite, un constructeur peut initaliser de telles propriétés par défaut ou des valeurs définies par l'utilisateur qui peuvent en fait être envoyées par des paramètres d'entrée d'un constructeur.

Appeler un constructeur d'une classe simple

Ceci est une classe simple `Person` .

```
classdef Person
    properties
        name
        surname
        address
    end

    methods
        function obj = Person(name,surname,address)
            obj.name = name;
```

```
            obj.surname = surname;
            obj.address = address;
        end
    end
end
```

Le nom d'un constructeur est le même que le nom d'une classe. Par conséquent, les constructeurs sont appelés par le nom de leur classe. Une `Person` classe peut être créée comme suit:

```
>> p = Person('John','Smith','London')
p =
  Person with properties:

       name: 'John'
    surname: 'Smith'
    address: 'London'
```

Appeler un constructeur d'une classe enfant

Les classes peuvent être héritées des classes parentes si elles partagent des propriétés ou des méthodes communes. Lorsqu'une classe est héritée d'une autre, il est probable qu'un constructeur d'une classe parente doit être appelé.

Un `Member` classe hérite d'une classe `Person` car `Member` utilise les mêmes propriétés que la classe Person mais ajoute également le `payment` à sa définition.

```
classdef Member < Person
    properties
        payment
    end

    methods
        function obj = Member(name,surname,address,payment)
            obj = obj@Person(name,surname,address);
            obj.payment = payment;
        end
    end
end
```

De même que pour la classe `Person` , `Member` est créé en appelant son constructeur:

```
>> m = Member('Adam','Woodcock','Manchester',20)
m =
  Member with properties:

    payment: 20
       name: 'Adam'
    surname: 'Woodcock'
    address: 'Manchester'
```

Un constructeur de `Person` nécessite trois paramètres d'entrée. `Member` doit respecter ce fait et donc appeler un constructeur de la classe `Person` avec trois paramètres. Il est rempli par la ligne:

```
obj = obj@Person(name,surname,address);
```

L'exemple ci-dessus montre le cas où une classe enfant a besoin d'informations pour sa classe parente. C'est pourquoi un constructeur de `Member` requiert quatre paramètres: trois pour sa classe parente et un pour lui-même.

Lire Programmation orientée objet en ligne: https://riptutorial.com/fr/matlab/topic/1028/programmation-orientee-objet

Chapitre 27: Solveurs des équations différentielles ordinaires (ODE)

Examples

Exemple pour odeset

Tout d'abord, nous initialisons notre problème de valeur initiale que nous voulons résoudre.

```
odefun = @(t,y) cos(y).^2*sin(t);
tspan = [0 16*pi];
y0=1;
```

Nous utilisons ensuite la fonction ode45 sans aucune option spécifiée pour résoudre ce problème. Pour le comparer plus tard, nous traçons la trajectoire.

```
[t,y] = ode45(odefun, tspan, y0);
plot(t,y,'-o');
```

Nous établissons maintenant une limite étroite et une limite absolue absolue de tolérance à notre problème.

```
options = odeset('RelTol',1e-2,'AbsTol',1e-2);
[t,y] = ode45(odefun, tspan, y0, options);
plot(t,y,'-o');
```

Nous fixons une limite absolue et étroite de tolérance absolue.

```
options = odeset('RelTol',1e-7,'AbsTol',1e-2);
[t,y] = ode45(odefun, tspan, y0, options);
plot(t,y,'-o');
```

Nous fixons une limite absolue et étroite de tolérance absolue. Comme dans les exemples précédents avec des limites de tolérance étroites, la trajectoire est complètement différente de la première sans option spécifique.

```
options = odeset('RelTol',1e-2,'AbsTol',1e-7);
[t,y] = ode45(odefun, tspan, y0, options);
plot(t,y,'-o');
```

Nous fixons une limite absolue et stricte de tolérance absolue. La comparaison du résultat avec l’autre graphique souligne les erreurs de calcul avec des limites de tolérance étroites.

```
options = odeset('RelTol',1e-7,'AbsTol',1e-7);
[t,y] = ode45(odefun, tspan, y0, options);
plot(t,y,'-o');
```

Les éléments suivants devraient démontrer le compromis entre précision et exécution.

```
tic;
options = odeset('RelTol',1e-7,'AbsTol',1e-7);
[t,y] = ode45(odefun, tspan, y0, options);
time1 = toc;
plot(t,y,'-o');
```

Pour comparaison, nous resserrons la limite de tolérance pour les erreurs absolues et relatives. Nous pouvons maintenant voir que sans un gain de précision important, il faudra beaucoup plus de temps pour résoudre notre problème de valeur initiale.

```
tic;
options = odeset('RelTol',1e-13,'AbsTol',1e-13);
[t,y] = ode45(odefun, tspan, y0, options);
time2 = toc;
plot(t,y,'-o');
```

Lire Solveurs des équations différentielles ordinaires (ODE) en ligne: https://riptutorial.com/fr/matlab/topic/6079/solveurs-des-equations-differentielles-ordinaires--ode-

Chapitre 28: Traitement d'image

Examples

Image de base I / O

```
>> img = imread('football.jpg');
```

Utilisez `imread` pour lire les fichiers image dans une matrice dans MATLAB.
Une fois que vous `imread` une image, celle-ci est stockée dans un tableau ND en mémoire:

```
>> size(img)
ans =
 256   320     3
```

L'image `'football.jpg'` comporte 256 lignes et 320 colonnes et comporte 3 canaux de couleur: rouge, vert et bleu.

Vous pouvez maintenant le refléter:

```
>> mirrored = img(:, end:-1:1, :);  %// like mirroring any ND-array in Matlab
```

Et enfin, écrivez-le en tant qu'image en utilisant `imwrite` :

```
>> imwrite(mirrored, 'mirrored_football.jpg');
```

Récupérer des images sur Internet

Tant que vous avez une connexion Internet, vous pouvez lire les images d'un lien hypertexte

```
I=imread('https://cdn.sstatic.net/Sites/stackoverflow/company/img/logos/so/so-logo.png');
```

Filtrage à l'aide d'une FFT 2D

Comme pour les signaux 1D, il est possible de filtrer les images en appliquant une transformation de Fourier, en multipliant avec un filtre dans le domaine fréquentiel et en transformant à nouveau le domaine spatial. Voici comment appliquer des filtres passe-haut ou passe-bas à une image avec Matlab:

Soit l' `image` l' `image` originale non filtrée, voici comment calculer son FFT 2D:

```
ft = fftshift(fft2(image));
```

Maintenant, pour exclure une partie du spectre, il faut définir ses valeurs de pixel sur 0. La fréquence spatiale contenue dans l'image d'origine est mappée du centre vers les bords (après

avoir utilisé `fftshift`). Pour exclure les basses fréquences, nous allons définir la zone circulaire centrale sur 0.

Voici comment générer un masque binaire en forme de disque de rayon `D` utilisant la fonction intégrée:

```
[x y ~] = size(ft);
D = 20;
mask = fspecial('disk', D) == 0;
mask = imresize(padarray(mask, [floor((x/2)-D) floor((y/2)-D)], 1, 'both'), [x y]);
```

Le masquage de l'image dans le domaine fréquentiel peut être effectué en multipliant le FFT au niveau du point par le masque binaire obtenu ci-dessus:

```
masked_ft = ft .* mask;
```

Maintenant, calculons l'inverse FFT:

```
filtered_image = ifft2(ifftshift(masked_ft), 'symmetric');
```

Les hautes fréquences d'une image sont les arêtes vives, ce filtre passe-haut peut donc être utilisé pour affiner les images.

Filtrage d'image

Disons que vous avez une image `rgbImg` , par exemple, lisez en utilisant `imread` .

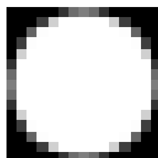
```
>> rgbImg = imread('pears.png');
>> figure, imshow(rgbImg), title('Original Image')
```

Original Image

Utilisez `fspecial` pour créer un filtre 2D:

```
>> h = fspecial('disk', 7);
>> figure, imshow(h, []), title('Filter')
```

Filter



Utilisez `imfilter` pour appliquer le filtre sur l'image:

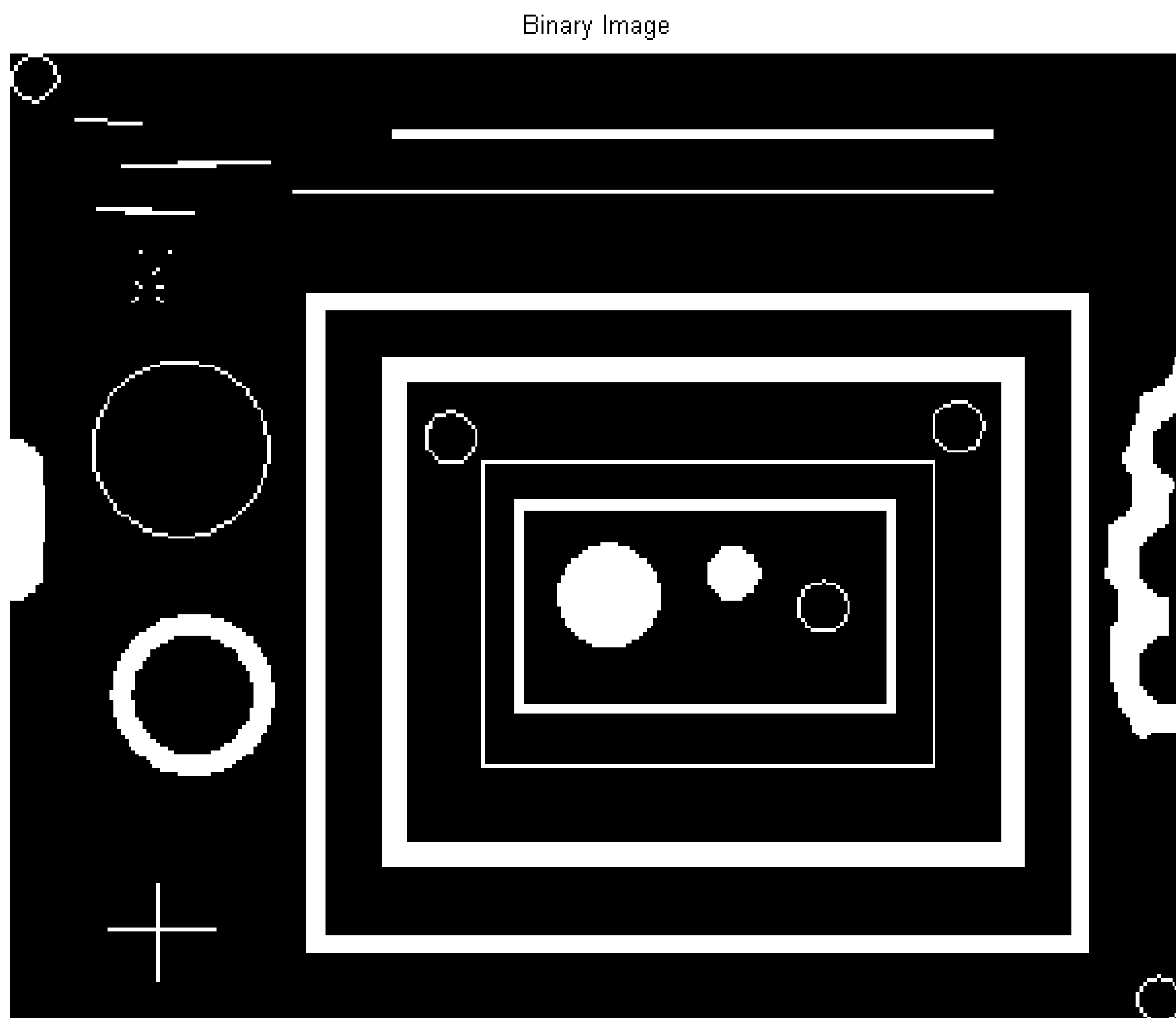
```
>> filteredRgbImg = imfilter(rgbImg, h);
>> figure, imshow(filteredRgbImg), title('Filtered Image')
```

Filtered Image

Propriétés de mesure des régions connectées

À partir d'une image binaire, `bwImg` , qui contient un certain nombre d'objets connectés.

```
>> bwImg = imread('blobs.png');
>> figure, imshow(bwImg), title('Binary Image')
```



Pour mesurer les propriétés (par exemple, la surface, le centroïde, etc.) de chaque objet de l'image, utilisez les `regionprops` :

```
>> stats = regionprops(bwImg, 'Area', 'Centroid');
```

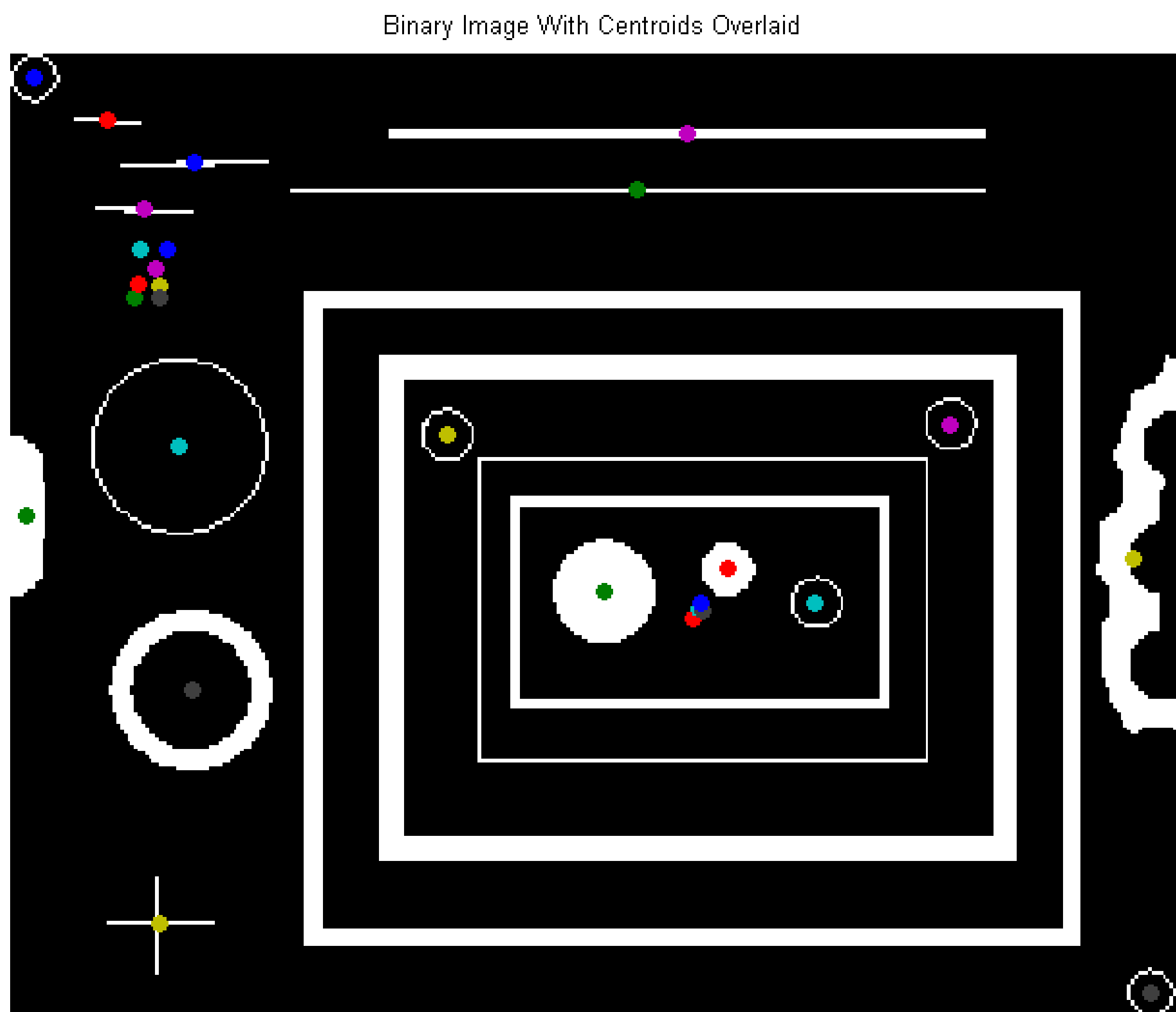
`stats` est un tableau struct qui contient une structure pour chaque objet de l'image. L'accès à une propriété mesurée d'un objet est simple. Par exemple, pour afficher la zone du premier objet, simplement,

```
>> stats(1).Area

ans =

    35
```

Visualisez les centroïdes d'objet en les superposant sur l'image d'origine.

```
>> figure, imshow(bwImg), title('Binary Image With Centroids Overlaid')
>> hold on
>> for i = 1:size(stats)
scatter(stats(i).Centroid(1), stats(i).Centroid(2), 'filled');
```

```
end
```



Lire Traitement d'image en ligne: https://riptutorial.com/fr/matlab/topic/3274/traitement-d-image

Chapitre 29: Transformées de Fourier et Transformées de Fourier Inverse

Syntaxe

1. Y = fft (X)% calcule la FFT de Vector ou Matrix X en utilisant une longueur de transformation par défaut de 256 (à confirmer pour la version)
2. Y = fft (X, n)% calcule la FFT de X en utilisant n comme longueur de transformation, n doit être un nombre basé sur 2 puissances. Si la longueur de X est inférieure à n, Matlab compresse automatiquement X avec des zéros tels que la longueur (X) = n
3. Y = fft (X, n, dim)% calcule la FFT de X en utilisant n comme longueur de transformation le long de la dimension dim (peut être égale à 1 ou 2 pour l'horizontale ou la verticale, respectivement)
4. Y = fft2 (X)% Calculer la FFT 2D de X
5. Y = fftn (X, dim)% Calculer la FFT dimensionnelle dimensionnelle de X, par rapport au vecteur de dimensions dim.
6. y = ifft (X)% calcule l'inverse de FFT de X (qui est une matrice / vecteur de nombres) en utilisant la longueur de transformation par défaut de 256
7. y = ifft (X, n)% calcule l'IFFT de X en utilisant n comme longueur de transformation
8. y = ifft (X, n, dim)% calcule l'IFFT de X en utilisant n comme longueur de transformation sur la dimension dim (peut être 1 ou 2 pour l'horizontale ou la verticale, respectivement)
9. y = ifft (X, n, dim, 'symétrique')% L'option Symétrique permet à ifft de traiter X comme conjugué symétrique le long de la dimension active. Cette option est utile lorsque X n'est pas exactement conjugué symétrique, simplement en raison d'une erreur d'arrondi.
10. y = ifft2 (X)% Calculer l'inverse 2D ft de X
11. y = ifftn (X, dim)% Calculer l'inverse dim dimensionnel fft de X.

Paramètres

Paramètre	La description
X	Ceci est votre signal Time-Domain d'entrée, il devrait être un vecteur de chiffres.
n	c'est le paramètre NFFT connu sous le nom de longueur de transformation, pensez-y comme étant la résolution de votre résultat FFT, il DOIT être un

Paramètre	La description
	nombre qui est une puissance de 2 (c.-à-d. 64 128 256 ... 2 ^ N)
faible	c'est la dimension sur laquelle vous voulez calculer la FFT, utilisez 1 si vous voulez calculer votre FFT dans la direction horizontale et 2 si vous voulez calculer votre FFT dans la direction verticale - Notez que ce paramètre est généralement laissé vide, car la fonction est capable de détecter la direction de votre vecteur.

Remarques

Matlab FFT est un processus très parallèle capable de gérer de grandes quantités de données. Il peut également utiliser le GPU à un énorme avantage.

```
ifft(fft(X)) = X
```

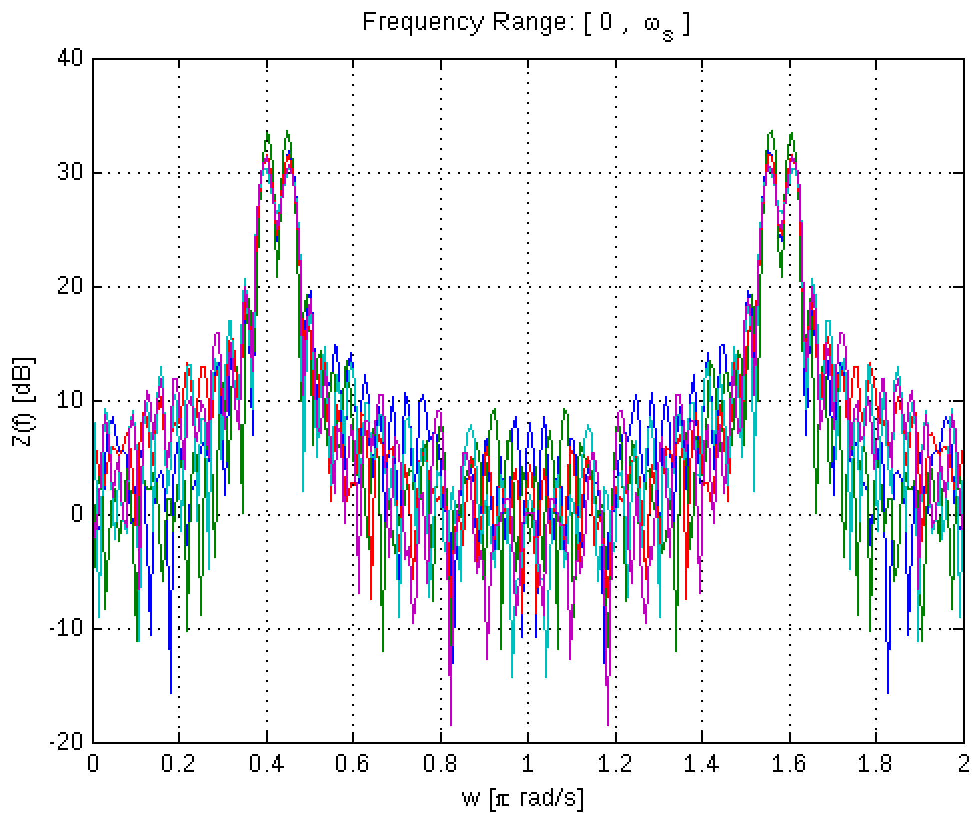
L'instruction ci-dessus est vraie si les erreurs d'arrondi sont omises.

Examples

Implémenter une simple transformation de Fourier dans Matlab

La transformation de Fourier est probablement la première leçon de traitement du signal numérique, son application est omniprésente et constitue un outil puissant pour analyser des données (dans tous les secteurs) ou des signaux. Matlab dispose d'un ensemble de boîtes à outils puissantes pour la transformation de Fourier. Dans cet exemple, nous utiliserons la transformée de Fourier pour analyser un signal sinusoïdal de base et générer ce que l'on appelle parfois un périodogramme à l'aide de la FFT:

```
%Signal Generation
A1=10;                % Amplitude 1
A2=10;                % Amplitude 2
w1=2*pi*0.2;          % Angular frequency 1
w2=2*pi*0.225;        % Angular frequency 2
Ts=1;                 % Sampling time
N=64;                 % Number of process samples to be generated
K=5;                  % Number of independent process realizations
sgm=1;                % Standard deviation of the noise
n=repmat([0:N-1].',1,K);              % Generate resolution
phi1=repmat(rand(1,K)*2*pi,N,1);      % Random phase matrix 1
phi2=repmat(rand(1,K)*2*pi,N,1);      % Random phase matrix 2
x=A1*sin(w1*n*Ts+phi1)+A2*sin(w2*n*Ts+phi2)+sgm*randn(N,K);   % Resulting Signal

NFFT=256;            % FFT length
F=fft(x,NFFT);       % Fast Fourier Transform Result
Z=1/N*abs(F).^2;     % Convert FFT result into a Periodogram
```



Notez que la Transformée de Fourier Discrète est implémentée par Fast Fourier Transform (fft) dans Matlab, les deux donneront le même résultat, mais FFT est une implémentation rapide de DFT.

```
figure
w=linspace(0,2,NFFT);
plot(w,10*log10(Z)),grid;
xlabel('w [\pi rad/s]')
ylabel('Z(f) [dB]')
title('Frequency Range: [ 0 ,  \omega_s ]')
```

Transformées de Fourier Inverse

L'un des principaux avantages de la transformation de Fourier est sa capacité à revenir au domaine temporel sans perdre d'informations. Considérons le même Signal que nous avons utilisé dans l'exemple précédent:

```
A1=10;                % Amplitude 1
A2=10;                % Amplitude 2
w1=2*pi*0.2;          % Angular frequency 1
w2=2*pi*0.225;        % Angular frequency 2
Ts=1;                 % Sampling time
N=64;                 % Number of process samples to be generated
K=1;                  % Number of independent process realizations
sgm=1;                % Standard deviation of the noise
n=repmat([0:N-1].',1,K);             % Generate resolution
phi1=repmat(rand(1,K)*2*pi,N,1);     % Random phase matrix 1
phi2=repmat(rand(1,K)*2*pi,N,1);     % Random phase matrix 2
x=A1*sin(w1*n*Ts+phi1)+A2*sin(w2*n*Ts+phi2)+sgm*randn(N,K);   % Resulting Signal
```

```
NFFT=256;             % FFT length
F=fft(x,NFFT);        % FFT result of time domain signal
```

Si nous ouvrons `F` dans Matlab, nous constaterons que c'est une matrice de nombres complexes, une partie réelle et une partie imaginaire. Par définition, afin de récupérer le signal Time Domain d'origine, nous avons besoin à la fois du Real (qui représente la variation de l'amplitude) et de l'Imaginary (qui représente la variation de phase).

```
TD = ifft(F,NFFT);   %Returns the Inverse of F in Time Domain
```

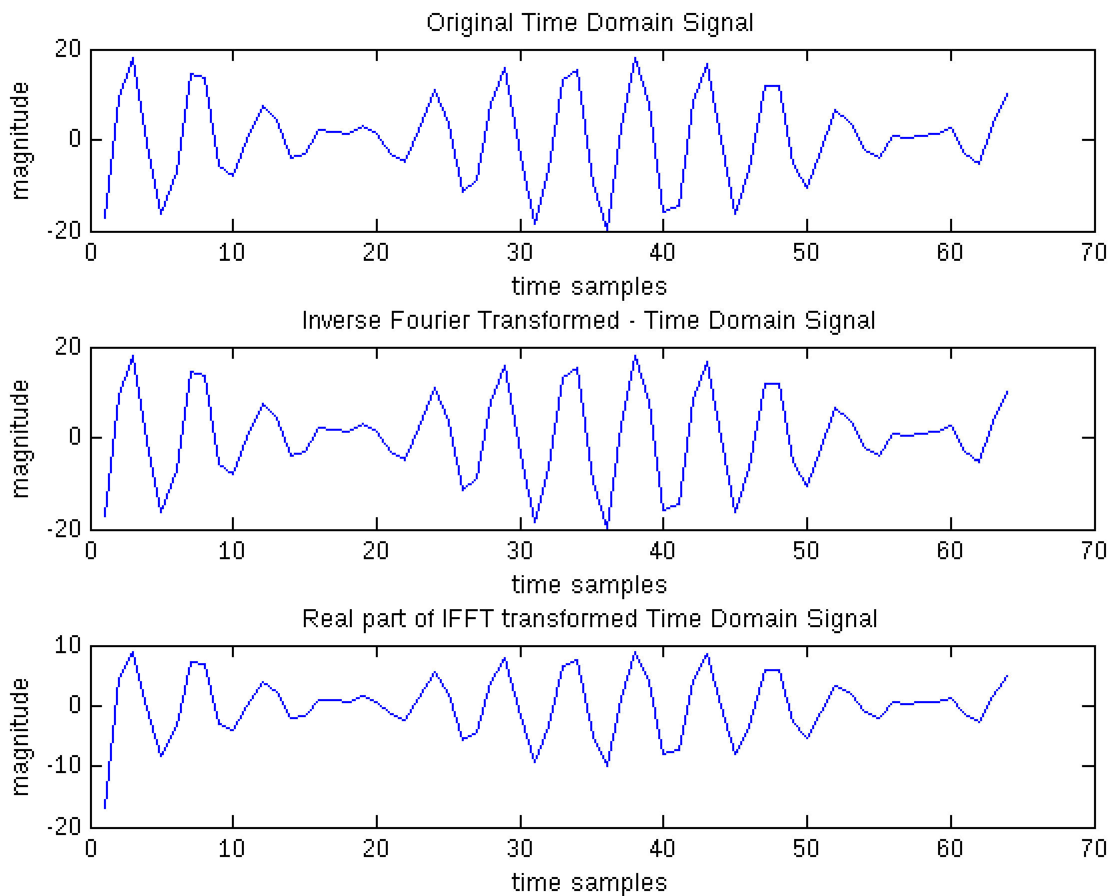
Notez que TD renvoyé aurait la longueur 256 car nous avons défini NFFT sur 256, mais la longueur de x est seulement 64, donc Matlab compilera les zéros à la fin de la transformation TD. Ainsi, par exemple, si NFFT était 1024 et que la longueur était 64, le TD retourné serait 64 + 960 zéros. Notez également qu'en raison de l'arrondi à virgule flottante, vous pourriez obtenir quelque chose comme 3.1 * 10e-20 mais pour des raisons générales: Pour tout X, ifft (fft (X)) est égal à X dans l'erreur d'arrondi.

Disons un instant qu'après la transformation, nous avons fait quelque chose et ne sommes restés qu'avec la partie REAL de la FFT:

```
R = real(F);          %Give the Real Part of the FFT
TDR = ifft(R,NFFT);   %Give the Time Domain of the Real Part of the FFT
```

Cela signifie que nous perdons la partie imaginaire de notre FFT et que, par conséquent, nous perdons des informations dans ce processus inverse. Pour préserver l'original sans perdre d'informations, vous devez toujours conserver la partie imaginaire de la FFT en utilisant `imag` et appliquer vos fonctions à la fois à la pièce réelle et à la pièce.

```
figure
subplot(3,1,1)
plot(x);xlabel('time samples');ylabel('magnitude');title('Original Time Domain Signal')
subplot(3,1,2)
plot(TD(1:64));xlabel('time samples');ylabel('magnitude');title('Inverse Fourier Transformed -
Time Domain Signal')
subplot(3,1,3)
plot(TDR(1:64));xlabel('time samples');ylabel('magnitude');title('Real part of IFFT
transformed Time Domain Signal')
```



Images et FT multidimensionnelles

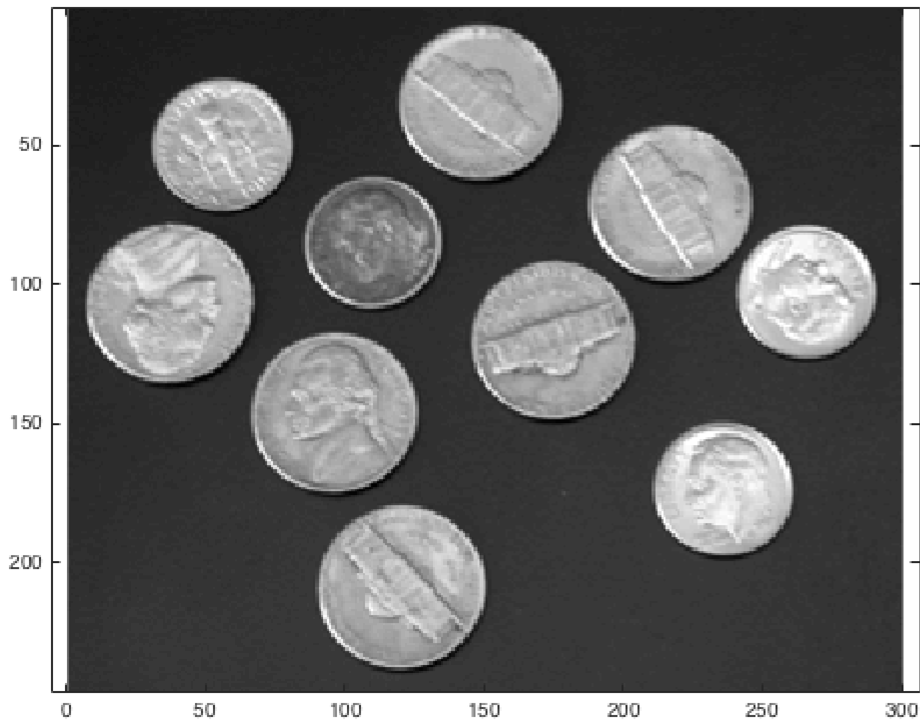
Dans l'imagerie médicale, la spectroscopie, le traitement d'images, la cryptographie et d'autres domaines de la science et de l'ingénierie, on souhaite souvent calculer des transformées de Fourier multidimensionnelles. Ceci est assez simple dans Matlab: les images (multidimensionnelles) ne sont que des matrices à n dimensions, après tout, les transformées de Fourier sont des opérateurs linéaires: l'une juste transforme Fourier selon d'autres dimensions. Matlab fournit `fft2` et `ifft2` pour le faire en `fftn` , ou `fftn` en n-dimensions.

Un écueil potentiel est que la transformée de Fourier des images est généralement représentée "centrée sur l'ordre", c'est-à-dire avec l'origine de l'espace k au milieu de l'image. Matlab fournit la commande `fftshift` pour échanger de manière `fftshift` l'emplacement des composants DC de la transformée de Fourier. Cette notation permet de simplifier considérablement les techniques courantes de traitement des images, dont l'une est illustrée ci-dessous.

Zéro remplissage

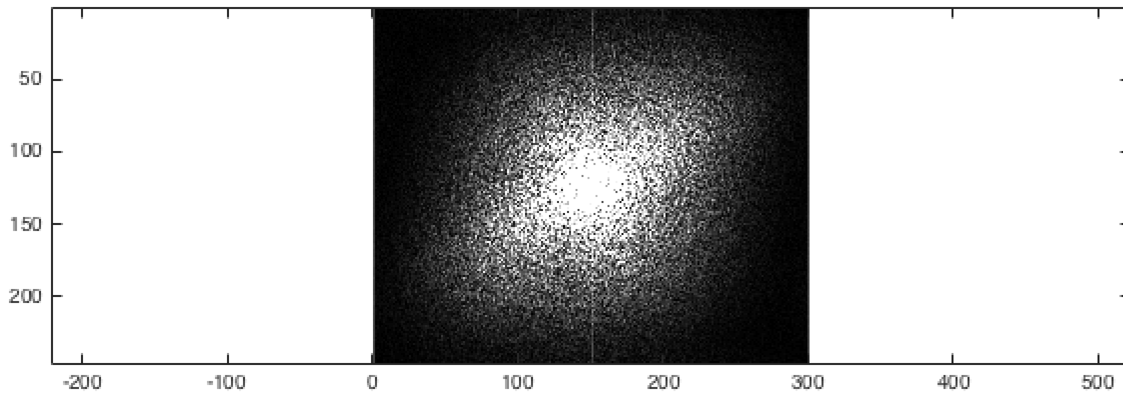
Une façon "rapide et sale" d’interpoler une petite image à une plus grande taille est de la transformer en Fourier, de placer des zéros dans la transformée de Fourier, puis de prendre la transformation inverse. Cela interpole efficacement chaque pixel avec une fonction de base en forme de sinc et est couramment utilisé pour mettre à niveau les données d'imagerie médicale à basse résolution. Commençons par charger un exemple d'image intégré

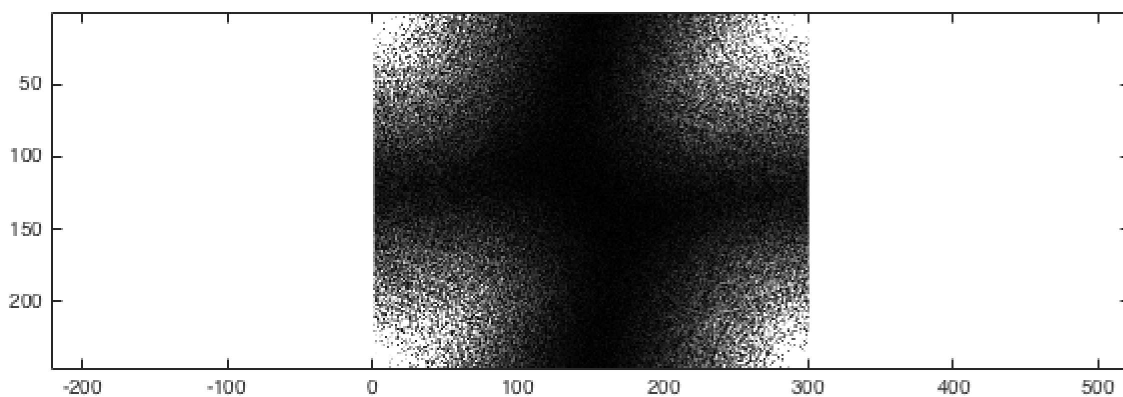
```
%Load example image
I=imread('coins.png'); %Load example data -- coins.png is builtin to Matlab
I=double(I); %Convert to double precision -- imread returns integers
imageSize = size(I); % I is a 246 x 300 2D image

%Display it
imagesc(I); colormap gray; axis equal;
%imagesc displays images scaled to maximum intensity
```



Nous pouvons maintenant obtenir la transformée de Fourier de I. Pour illustrer ce `fftshift` fait `fftshift` , comparons les deux méthodes:

```
% Fourier transform
%Obtain the centric- and non-centric ordered Fourier transform of I
k=fftshift(fft2(fftshift(I)));
kwrong=fft2(I);

%Just for the sake of comparison, show the magnitude of both transforms:
figure; subplot(2,1,1);
imagesc(abs(k),[0 1e4]); colormap gray; axis equal;
subplot(2,1,2);
imagesc(abs(kwrong),[0 1e4]); colormap gray; axis equal;
%(The second argument to imagesc sets the colour axis to make the difference clear).
```

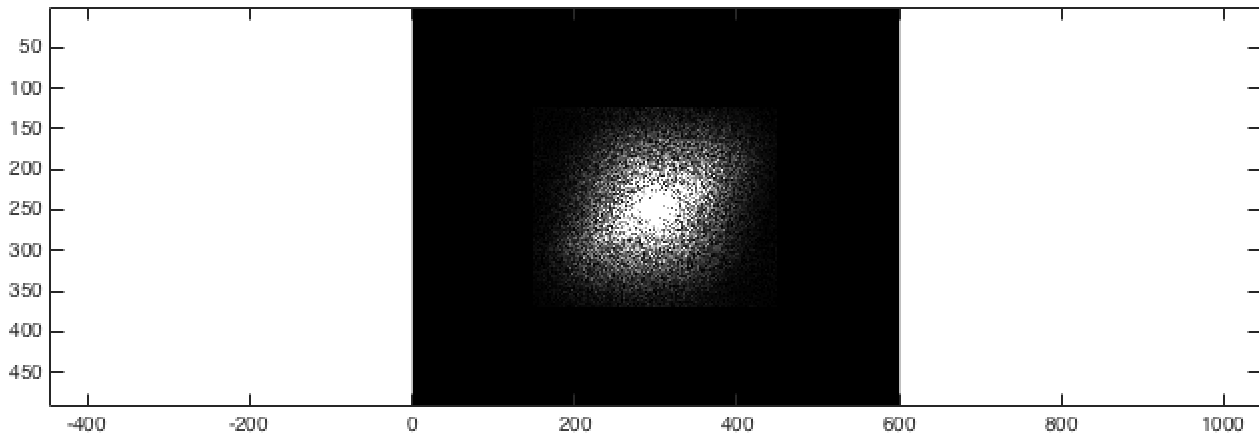


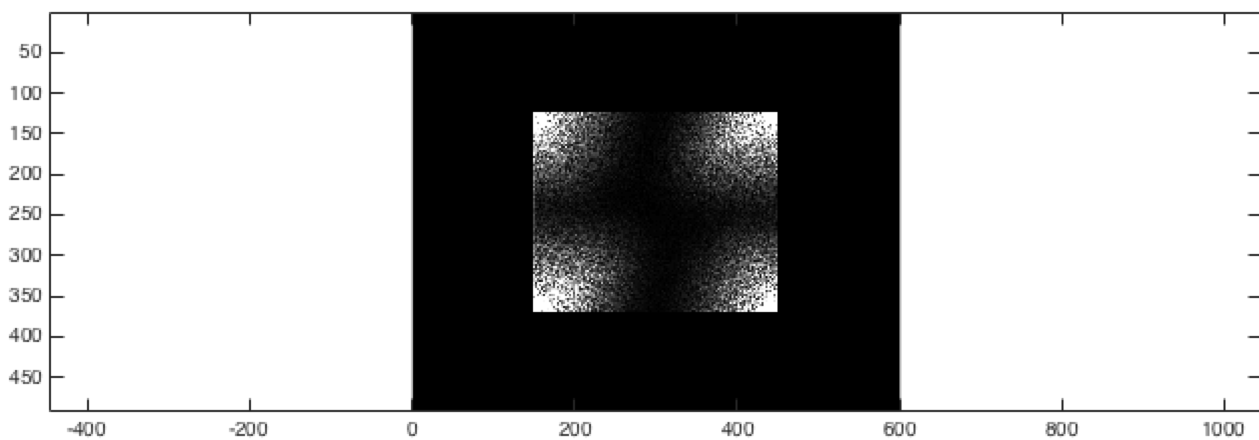


Nous avons maintenant obtenu le FT 2D d'un exemple d'image. Pour le remplir à zéro, nous voulons prendre chaque espace k, remplir les bords avec des zéros, puis prendre la transformation inverse:

```
%Zero fill
kzf = zeros(imageSize .* 2); %Generate a  492x600 empty array to put the result in
kzf(end/4:3*end/4-1,end/4:3*end/4-1) = k; %Put k in the middle
kzfwrong = zeros(imageSize .* 2); %Generate a  492x600 empty array to put the result in
kzfwrong(end/4:3*end/4-1,end/4:3*end/4-1) = kwrong; %Put k in the middle

%Show the differences again
%Just for the sake of comparison, show the magnitude of both transforms:
figure; subplot(2,1,1);
imagesc(abs(kzf),[0 1e4]); colormap gray; axis equal;
subplot(2,1,2);
imagesc(abs(kzfwrong),[0 1e4]); colormap gray; axis equal;
%(The second argument to imagesc sets the colour axis to make the difference clear).
```

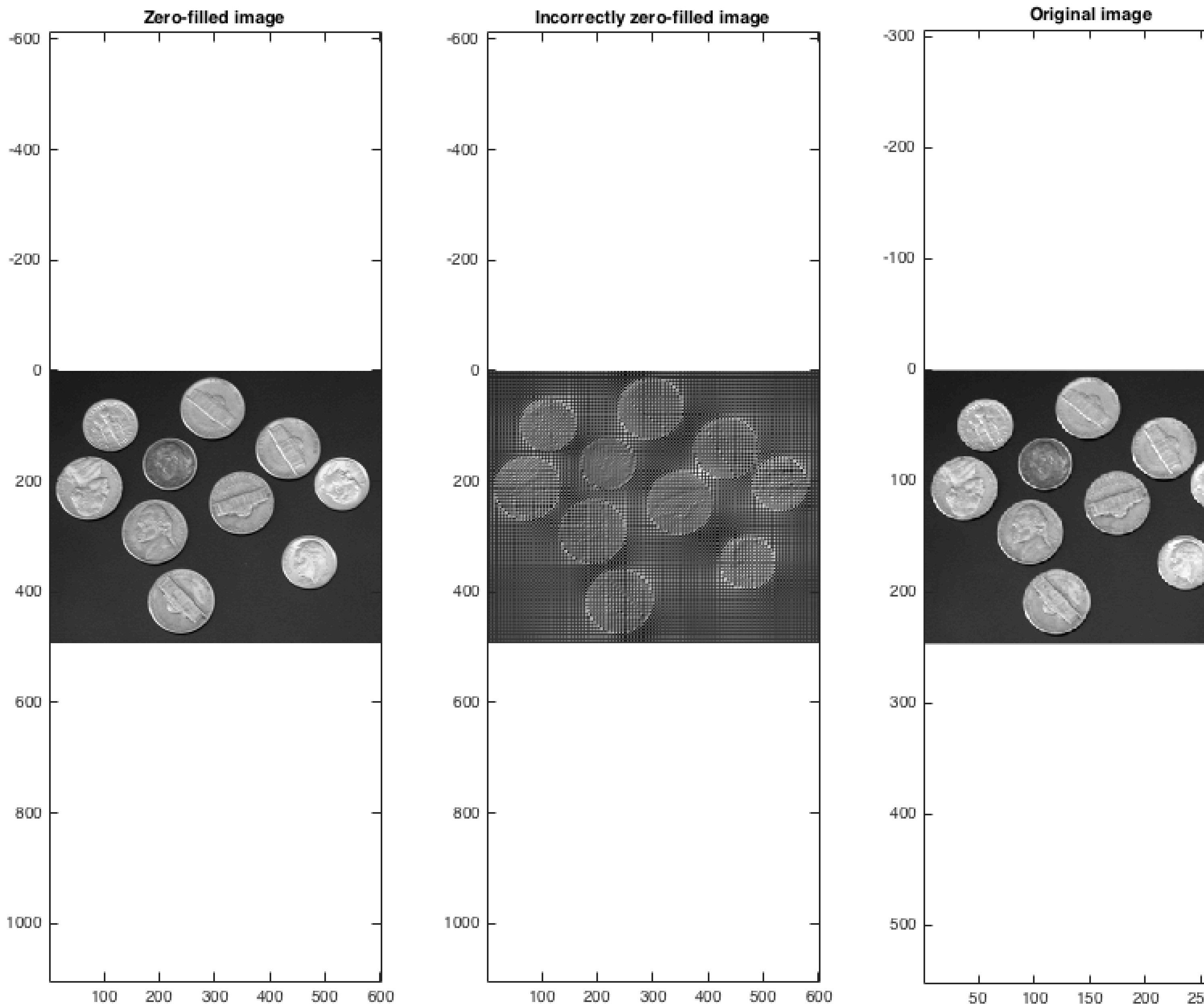
À ce stade, le résultat est plutôt banal:





Une fois que nous prenons les back-transforms, nous pouvons voir que (correctement!) Les données sans remplissage fournissent une méthode sensée d'interpolation:

```
% Take the back transform and view
Izf = fftshift(ifft2(ifftshift(kzf)));
Izfwrong = ifft2(kzfwrong);

figure; subplot(1,3,1);
imagesc(abs(Izf)); colormap gray; axis equal;
title('Zero-filled image');
subplot(1,3,2);
imagesc(abs(Izfwrong)); colormap gray; axis equal;
title('Incorrectly zero-filled image');
subplot(1,3,3);
imagesc(I); colormap gray; axis equal;
title('Original image');
set(gcf,'color','w');
```



Notez que la taille de l'image à zéro est le double de celle de l'original. On peut mettre à zéro plus d'un facteur deux dans chaque dimension, bien que cela n'augmente pas de manière arbitraire la taille d'une image.

Trucs, astuces, 3D et au-delà

L'exemple ci-dessus vaut pour les images 3D (souvent générées par les techniques d'imagerie médicale ou la microscopie confocale, par exemple), mais nécessite que `fft2` soit remplacé par `fftn(I, 3)` , par exemple. En raison de la nature quelque peu lourde de l'écriture de `fftshift(fft(fftshift(...` plusieurs fois, il est assez courant de définir des fonctions telles que `fft2c` localement pour fournir localement une syntaxe plus facile, comme:

```
function y = fft2c(x)

y = fftshift(fft2(fftshift(x)));
```

Notez que la FFT est rapide, mais les grandes transformations multidimensionnelles de Fourier prendront du temps sur un ordinateur moderne. En outre, elle est intrinsèquement complexe: l’ampleur de l’espace k était indiquée ci-dessus, mais la phase est absolument vitale; les traductions dans le domaine de l'image sont équivalentes à une rampe de phase dans le domaine de Fourier. Il existe plusieurs opérations beaucoup plus complexes que l'on peut souhaiter effectuer dans le domaine de Fourier, telles que le filtrage de fréquences spatiales hautes ou basses (en le multipliant par un filtre) ou en masquant des points discrets correspondant au bruit. Il existe en conséquence une grande quantité de code généré par la communauté pour gérer les opérations de Fourier courantes disponibles sur le principal site de référentiel communautaire de Matlab, à savoir File Exchange .

Lire Transformées de Fourier et Transformées de Fourier Inverse en ligne: https://riptutorial.com/fr/matlab/topic/2181/transformees-de-fourier-et-transformees-de-fourier-inverse

Chapitre 30: Utilisation de fonctions avec sortie logique

Examples

Tous et tout avec des tableaux vides

Des précautions particulières doivent être prises lorsqu'il est possible qu'un tableau devienne un tableau vide en ce qui concerne les opérateurs logiques. On s'attend souvent à ce que si `all(A)` est vrai, `any(A)` doit être vrai et que si `any(A)` est faux, `all(A)` doit également être faux. Ce n'est pas le cas dans MATLAB avec des tableaux vides.

```
>> any([])
ans =
     0
>> all([])
ans =
     1
```

Donc, si par exemple vous comparez tous les éléments d'un tableau avec un certain seuil, vous devez être conscient du cas où le tableau est vide:

```
>> A=1:10;
>> all(A>5)
ans =
     0
>> A=1:0;
>> all(A>5)
ans =
     1
```

Utilisez la fonction `isempty` pour vérifier les tableaux vides:

```
a = [];
isempty(a)
ans =
1
```

Lire Utilisation de fonctions avec sortie logique en ligne: https://riptutorial.com/fr/matlab/topic/5608/utilisation-de-fonctions-avec-sortie-logique

Chapitre 31: Utilisation de la fonction `accumarray ()`

Introduction

`accumarray` permet d'agréger les éléments d'un tableau de différentes manières, en appliquant éventuellement une fonction aux éléments du processus. `accumarray` peut être considéré comme un réducteur léger (voir aussi: Introduction à MapReduce).

Cette rubrique contient des scénarios courants dans lesquels `accumarray` est particulièrement utile.

Syntaxe

- accumarray (subscriptArray, valuesArray)
- accumarray (subscriptArray, valuesArray, sizeOfOutput)
- accumarray (subscriptArray, valuesArray, sizeOfOutput, funcHandle)
- accumarray (subscriptArray, valuesArray, sizeOfOutput, funcHandle, fillVal)
- accumarray (subscriptArray, valuesArray, sizeOfOutput, funcHandle, fillVal, isSparse)

Paramètres

Paramètre	Détails
`subscriptArray`	Matrice d'indices, spécifiée comme vecteur d'indices, matrice d'indices ou matrice de vecteurs d'index.
`valuesArray`	Données spécifiées en tant que vecteur ou scalaire.
`sizeOfOutput`	Taille du tableau de sortie, spécifiée comme vecteur d'entiers positifs.
`funcHandle`	Fonction à appliquer à chaque ensemble d'éléments lors de l'agrégation, spécifiée en tant que descripteur de fonction ou `[]` .
`fillVal`	Valeur de remplissage, lorsque `subs` ne référence pas chaque élément dans la sortie.
`isSparse`	La sortie doit-elle être un tableau fragmenté?

Remarques

- Introduit dans MATLAB v7.0.

Références :

1. " `accumarray` sous- `accumarray` ", *par Loren Shure* , 20 février 2008 .
2. `accumarray` dans la documentation officielle de MATLAB.

Examples

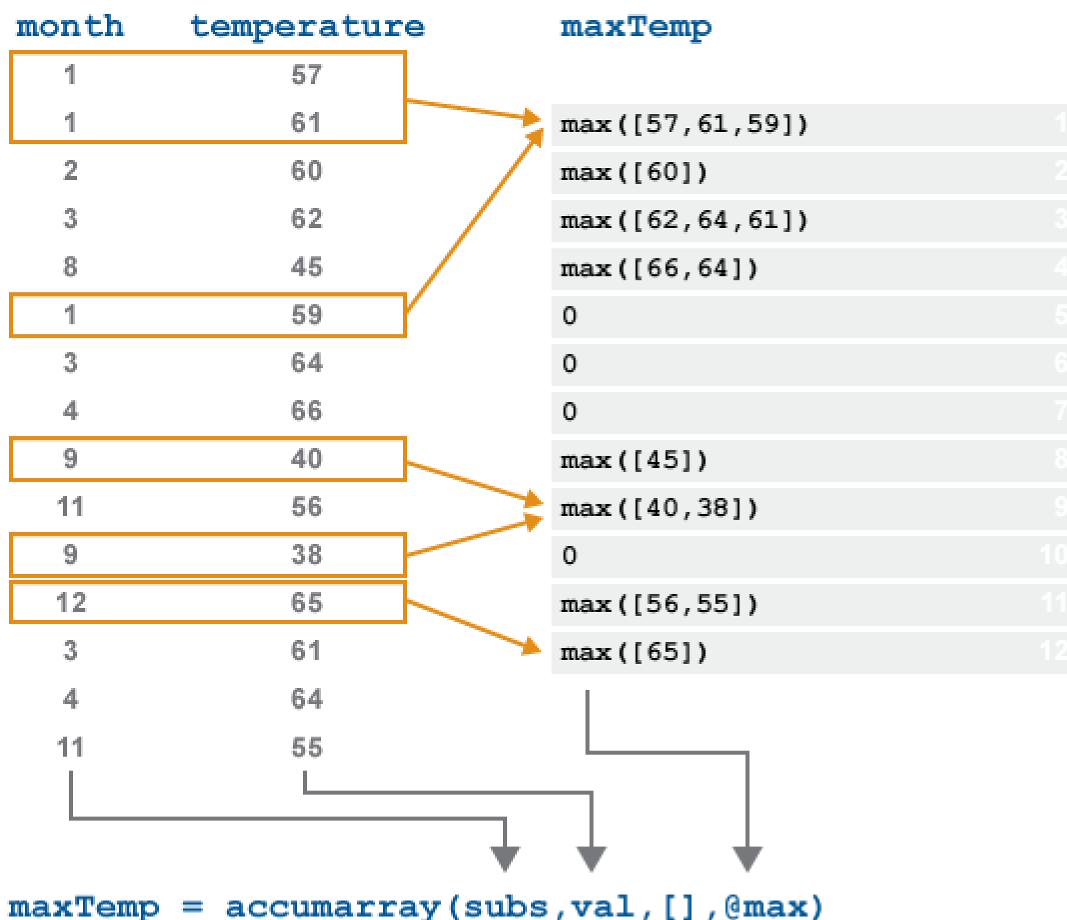
Recherche de la valeur maximale parmi les éléments regroupés par un autre vecteur

Ceci est un exemple officiel de MATLAB

Considérez le code suivant:

```
month = [1;1;2;3;8;1;3;4;9;11;9;12;3;4;11];
temperature = [57;61;60;62;45;59;64;66;40;56;38;65;61;64;55];
maxTemp = accumarray(month,temperature,[],@max);
```

L'image ci-dessous illustre le processus de calcul effectué par `accumarray` dans ce cas:



Dans cet exemple, toutes les valeurs ayant le même `month` sont d'abord collectées, puis la fonction spécifiée par la 4 ème entrée d' `accumarray` (dans ce cas, `@max`) est appliquée à chaque ensemble.

Appliquer un filtre aux patchs d'image et définir chaque pixel comme la moyenne du résultat de chaque patch

De nombreux algorithmes de traitement d'image modernes utilisant des correctifs sont leur élément de base sur lequel travailler.
Par exemple, on pourrait débruiter les patchs (Voir Algorithme BM3D).

Pourtant, lors de la construction de l'image à partir des patchs traités, nous avons de nombreux résultats pour le même pixel.
Une façon de le gérer consiste à prendre la moyenne (moyenne empirique) de toutes les valeurs d'un même pixel.

Le code suivant montre comment fractionner une image en patches et reconstruire l'image à partir de patches en utilisant la moyenne en utilisant `[accumarray()][1]` :

```
numRows = 5;
numCols = 5;

numRowsPatch = 3;
numColsPatch = 3;

% The Image
mI = rand([numRows, numCols]);

% Decomposing into Patches - Each pixel is part of many patches (Neglecting
% boundariwes, each pixel is part of (numRowsPatch * numColsPatch) patches).
mY = ImageToColumnsSliding(mI, [numRowsPatch, numColsPatch]);

% Here one would apply some operation which work on patches

% Creating image of the index of each pixel
mPxIdx = reshape(1:(numRows * numCols), [numRows, numCols]);

% Creating patches of the same indices
mSubsAccu = ImageToColumnsSliding(mPxIdx, [numRowsPatch, numColsPatch]);

% Reconstruct the image - Option A
mO = accumarray(mSubsAccu(:), mY(:)) ./ accumarray(mSubsAccu(:), 1);

% Reconstruct the image - Option B
mO = accumarray(mSubsAccu, mY(:), [(numRows * numCols), 1], @(x) mean(x));

% Rehsape the Vector into the Image
mO = reshape(mO, [numRows, numCols]);
```

Lire Utilisation de la fonction `accumarray ()` en ligne:
https://riptutorial.com/fr/matlab/topic/9321/utilisation-de-la-fonction--accumarray----

Chapitre 32: Utiliser des ports série

Introduction

Les ports série sont une interface commune pour communiquer avec des capteurs externes ou des systèmes intégrés tels que Arduinos. Les communications série modernes sont souvent implémentées sur des connexions USB utilisant des adaptateurs USB-série. MATLAB fournit des fonctions intégrées pour les communications série, y compris les protocoles RS-232 et RS-485. Ces fonctions peuvent être utilisées pour les ports série matériels ou les connexions série USB "virtuelles". Les exemples illustrent les communications série dans MATLAB.

Paramètres

Paramètre de port série	ce qu'il fait
`BaudRate`	Définit le débit en bauds. Le plus commun aujourd'hui est 57600, mais 4800, 9600 et 115200 sont également fréquemment vus
`InputBufferSize`	Le nombre d'octets conservés en mémoire. Matlab a une FIFO, ce qui signifie que les nouveaux octets seront supprimés. La valeur par défaut est 512 octets, mais elle peut facilement être définie sur 20 Mo sans problème. Il y a seulement quelques cas extrêmes où l'utilisateur voudrait que ce soit petit
`BytesAvailable`	Le nombre d'octets en attente de lecture
`ValuesSent`	Le nombre d'octets envoyés depuis l'ouverture du port
`ValuesReceived`	Le nombre d'octets lus depuis l'ouverture du port
`BytesAvailableFcn`	Spécifiez la fonction de rappel à exécuter lorsqu'un nombre spécifié d'octets est disponible dans le tampon d'entrée ou qu'un terminateur est lu
`BytesAvailableFcnCount`	Indiquez le nombre d'octets devant être disponibles dans le tampon d'entrée pour générer un événement `bytes-available`
`BytesAvailableFcnMode`	Indiquez si l'événement d' `bytes-available` est généré après qu'un nombre spécifié d'octets est disponible dans le tampon d'entrée ou après la lecture d'un terminateur

Examples

Créer un port série sur Mac / Linux / Windows

```
% Define serial port with a baud rate of 115200
rate = 115200;
if ispc
    s = serial('COM1', 'BaudRate',rate);
elseif ismac
    % Note that on OSX the serial device is uniquely enumerated. You will
    % have to look at /dev/tty.* to discover the exact signature of your
    % serial device
    s = serial('/dev/tty.usbserial-A104VFT7', 'BaudRate',rate);
elseif isunix
    s = serial('/dev/ttyusb0', 'BaudRate',rate);
end

% Set the input buffer size to 1,000,000 bytes (default: 512 bytes).
s.InputBufferSize = 1000000;

% Open serial port
fopen(s);
```

Lecture depuis le port série

En supposant que vous avez créé le port série objet `s` comme dans cet exemple, puis

```
% Read one byte
data = fread(s, 1);

% Read all the bytes, version 1
data = fread(s);

% Read all the bytes, version 2
data = fread(s, s.BytesAvailable);

% Close the serial port
fclose(s);
```

Fermer un port série même perdu, supprimé ou écrasé

En supposant que vous avez créé le port série objet `s` comme dans cet exemple, puis de le fermer

```
fclose(s)
```

Cependant, il peut arriver que vous perdiez accidentellement le port (par exemple, effacer, écraser, modifier la portée, etc.) et `fclose(s)` ne fonctionnerait plus. La solution est facile

```
fclose(instrfindall)
```

Plus d'infos sur `instrfindall()` .

Écriture sur le port série

En supposant que vous avez créé le port série objet `s` comme dans cet exemple, puis

```
% Write one byte
fwrite(s, 255);

% Write one 16-bit signed integer
fwrite(s, 32767, 'int16');

% Write an array of unsigned 8-bit integers
fwrite(s,[48 49 50],'uchar');

% Close the serial port
fclose(s);
```

Choisir votre mode de communication

Matlab prend en charge la communication *synchrone* et *asynchrone* avec un port série. Il est important de choisir le bon mode de communication. Le choix dépendra de:

- comment se comporte l'instrument avec lequel vous communiquez.
- Quelles autres fonctions votre programme principal (ou GUI) devra-t-il faire en dehors de la gestion du port série?

Je définirai trois cas différents à illustrer, du plus simple au plus exigeant. Pour les 3 exemples, l'instrument auquel je me connecte est une carte de circuit avec un inclinomètre, qui peut fonctionner dans les 3 modes que je vais décrire ci-dessous.

Mode 1: synchrone (maître / esclave)

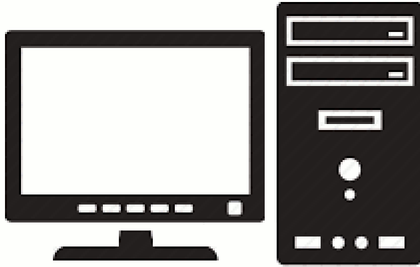
Ce mode est le plus simple. Cela correspond au cas où le PC est le *maître* et l'instrument est l' *esclave* . L'instrument n'envoie rien au port série seul, il ne **répond** qu'une réponse après avoir reçu une question / commande du maître (le PC, votre programme). Par exemple:

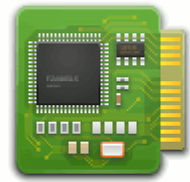
- Le PC envoie une commande: "Donnez-moi une mesure maintenant"
- L'appareil reçoit la commande, effectue la mesure puis renvoie la valeur de mesure sur la ligne série: "La valeur de l'inclinomètre est XXX".

OU

- Le PC envoie une commande: "Passer du mode X au mode Y"
- L'instrument reçoit la commande, l'exécute, puis envoie un message de confirmation à la ligne série: " *Commande exécutée* " (ou " *Commande NON exécutée* "). Ceci est communément appelé une réponse ACK / NACK (pour "Acknowledge (d)" / "NOT Acknowledged").

Résumé: dans ce mode, l'instrument (l' *esclave*) envoie uniquement des données à la ligne série **immédiatement après** avoir été invité par le PC (le *maître*)





Mode 2: Asynchrone

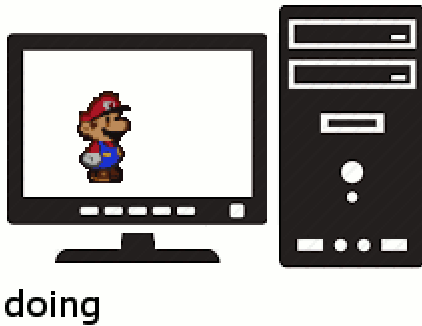
Maintenant, supposons que j'ai démarré mon instrument, mais c'est plus qu'un simple capteur muet. Il surveille constamment sa propre inclinaison et tant qu'il est vertical (dans une tolérance, disons +/- 15 degrés), il reste silencieux. Si l'appareil est incliné de plus de 15 degrés et se rapproche de l'horizontale, il envoie un message d'alarme à la ligne série, suivi immédiatement par une lecture de l'inclinaison. Tant que l'inclinaison est supérieure au seuil, il continue à envoyer une lecture d'inclinaison toutes les 5 secondes.

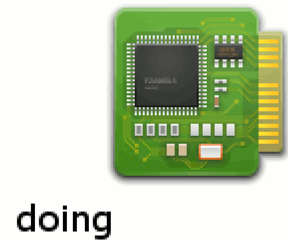
Si votre programme principal (ou interface graphique) attend constamment que le message arrive sur la ligne série, il peut le faire très bien ... mais il ne peut rien faire d'autre entre-temps. Si le programme principal est une interface graphique, il est extrêmement frustrant d'avoir une interface graphique apparemment "figée" car elle n'acceptera aucune entrée de la part de l'utilisateur. Essentiellement, il est devenu l' *esclave* et l'instrument est le *maître* . À moins d'avoir un moyen sophistiqué de contrôler votre interface graphique à partir de l'instrument, ceci est à éviter. Heureusement, le mode de communication *asynchrone* vous permettra:

- définir une fonction distincte qui indique à votre programme quoi faire lorsqu'un message est reçu
- conservez cette fonction dans un coin, elle ne sera appelée et exécutée que **lorsqu'un message arrive** sur la ligne série. Le reste du temps, l'interface graphique peut exécuter tout autre code à exécuter.

Résumé: Dans ce mode, l'instrument peut envoyer un message à la ligne série à tout moment (mais pas nécessairement *tout* le temps). Le PC *n'attend* pas de façon permanente pour un message à traiter. Il est autorisé à exécuter tout autre code. Seulement quand un message arrive, il active une fonction qui lira et traitera ce message.

ASYNCHRONOUS COMMUNICATION



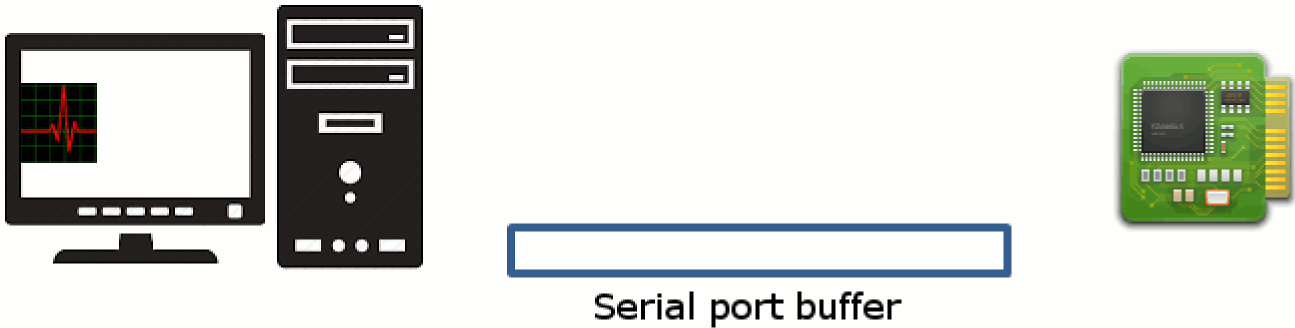


Mode 3: Streaming (*temps réel*)

Maintenant, libérons toute la puissance de mon instrument. Je le mets dans un mode où il enverra constamment des mesures à la ligne série. Mon programme souhaite recevoir ces paquets et les afficher sur une courbe ou un affichage numérique. S'il n'envoie qu'une valeur toutes les 5 secondes comme ci-dessus, pas de problème, conservez le mode ci-dessus. Mais mon instrument complet envoie un point de données à la ligne série à 1000 Hz, c'est-à-dire qu'il envoie une nouvelle valeur chaque milliseconde. Si je reste dans le *mode asynchrone* décrit ci-dessus, il y a un risque élevé (en fait une certitude garantie) que la fonction spéciale que nous avons définie pour traiter chaque nouveau paquet prendra plus de 1 ms (si vous voulez tracer ou afficher la valeur, les fonctions graphiques sont assez lentes, ne prenant même pas en compte le filtrage ou le FFT du signal). Cela signifie que la fonction commencera à s'exécuter, mais avant la fin, un nouveau paquet arrivera et déclenchera à nouveau la fonction. La deuxième fonction est placée dans une file d'attente pour exécution et ne démarre que lorsque le premier est terminé ... mais à ce moment-là, quelques nouveaux paquets sont arrivés et chacun a placé une fonction à exécuter dans la file d'attente. Vous pouvez rapidement prévoir le résultat: au moment où je trace les 5èmes points, il y a déjà des centaines de personnes qui attendent d'être tracées aussi ... la gui ralentit, finit par se figer, la pile grandit, les tampons se remplissent Finalement, il vous reste un programme complètement gelé ou simplement un programme écrasé.

Pour résoudre ce problème, nous déconnecterons encore davantage le lien de synchronisation entre le PC et l'instrument. Nous laisserons l'instrument envoyer des données à son propre rythme, sans déclencher immédiatement une fonction à chaque arrivée de paquet. Le tampon du port série accumulera simplement les paquets reçus. Le PC ne collecte les données dans le tampon qu'à un rythme qu'il peut gérer (un intervalle régulier, configuré du côté du PC), fait quelque chose (pendant que le tampon est rempli par l'instrument), puis collecte un nouveau lot de données. les données du tampon ... et ainsi de suite.

Résumé: Dans ce mode, l'instrument envoie des données en continu, qui sont collectées par le tampon du port série. À intervalles réguliers, le PC collecte les données du tampon et fait quelque chose avec lui. Il n'y a pas de lien de synchronisation matérielle entre le PC et l'instrument. Tous deux exécutent leurs tâches à leur propre rythme.



Traitement automatique des données reçues d'un port série

Certains appareils connectés via un port série envoient des données à votre programme à un débit constant (données en continu) ou envoient des données à des intervalles imprévisibles. Vous pouvez configurer le port série pour qu'il exécute une fonction automatiquement afin de gérer les données à chaque fois qu'il arrive. C'est ce qu'on appelle la "fonction de rappel" pour l'objet port série.

Deux propriétés du port série doivent être définies pour utiliser cette fonctionnalité: le nom de la fonction souhaitée pour le rappel (`BytesAvailableFcn`) et la condition qui doit déclencher l'exécution de la fonction de rappel (`BytesAvailableFcnMode`).

Il existe deux manières de déclencher une fonction de rappel:

1. Lorsqu'un certain nombre d'octets a été reçu sur le port série (généralement utilisé pour les données binaires)
2. Lorsqu'un certain caractère est reçu sur le port série (généralement utilisé pour du texte ou des données ASCII)

Les fonctions de rappel ont deux arguments d'entrée obligatoires, appelés `obj` et `event` . `obj` est le port série. Par exemple, si vous souhaitez imprimer les données reçues du port série, définissez une fonction pour imprimer les données appelées `newdata` :

```
function newdata(obj,event)
    [d,c] = fread(obj);  % get the data from the serial port
    % Note: for ASCII data, use fscanf(obj) to return characters instead of binary values
    fprintf(1,'Received %d bytes\n',c);
    disp(d)
end
```

Par exemple, pour exécuter la fonction `newdata` chaque fois que 64 octets de données sont reçus, configurez le port série comme `newdata` :

```
s = serial(port_name);
s.BytesAvailableFcnMode = 'byte';
s.BytesAvailableFcnCount = 64;
```

```
s.BytesAvailableFcn = @newdata;
```

Avec du texte ou des données ASCII, les données sont généralement divisées en lignes avec un "caractère de terminaison", comme le texte d'une page. Pour exécuter la fonction `newdata` chaque fois que le caractère de retour chariot est reçu, configurez le port série comme `newdata` :

```
s = serial(port_name);
s.BytesAvailableFcnMode = 'terminator';
s.Terminator = 'CR';  % the carriage return, ASCII code 13
s.BytesAvailableFcn = @newdata;
```

Lire Utiliser des ports série en ligne: https://riptutorial.com/fr/matlab/topic/1176/utiliser-des-ports-serie

Chapitre 33: Utilitaires de programmation

Examples

Minuterie simple dans MATLAB

Ce qui suit est une minuterie qui se déclenche à intervalle fixe. Son délai d'attente est défini par `Period` et il appelle un rappel défini par `Timerfcn` lors de l'expiration du délai.

```
t = timer;
t.TasksToExecute = Inf;
t.Period = 0.01; % timeout value in seconds
t.TimerFcn = @(myTimerObj, thisEvent)disp('hello'); % timer callback function
t.ExecutionMode = 'fixedRate';
start(t)
pause(inf);
```

Lire Utilitaires de programmation en ligne: https://riptutorial.com/fr/matlab/topic/1655/utilitaires-de-programmation

Chapitre 34: Vectorisation

Examples

Opérations élémentaires

MATLAB supporte (et encourage) les opérations vectorisées sur les vecteurs et les matrices. Par exemple, supposons que nous ayons `A` et `B` , deux matrices `n` - `m` et que nous voulons que `C` soit le produit élémentaire des éléments correspondants (c.-à-d. `C(i,j) = A(i,j)*B(i,j)`).

La manière non vectorisée d'utiliser des boucles imbriquées est la suivante:

```
C = zeros(n,m);
for ii=1:n
    for jj=1:m
        C(ii,jj) = A(ii,jj)*B(ii,jj);
    end
end
```

Cependant, la manière vectorisée de le faire est d’utiliser l’opérateur par élément `.*` :

```
C = A.*B;
```

- Pour plus d'informations sur la multiplication d'éléments dans MATLAB, consultez la documentation des `times` .
- Pour plus d'informations sur la différence entre les opérations matricielles et matricielles, voir Opérations Matrice ou Matrice dans la documentation MATLAB.

Somme, méchant, prod & co

Étant donné un vecteur aléatoire

```
v = rand(10,1);
```

si vous voulez la somme de ses éléments, n'utilisez **PAS de** boucle

```
s = 0;
for ii = 1:10
    s = s + v(ii);
end
```

mais utilisez la capacité vectorisée de la fonction `sum()`

```
s = sum(v);
```

Les fonctions comme `sum()` , `mean()` , `prod()` et autres ont la possibilité de fonctionner directement le long des lignes, des colonnes ou d'autres dimensions.

Par exemple, donner une matrice aléatoire

```
A = rand(10,10);
```

la moyenne pour chaque **colonne** est

```
m = mean(A,1);
```

la moyenne pour chaque **ligne** est

```
m = mean(A,2)
```

Toutes les fonctions ci-dessus ne fonctionnent que sur une seule dimension, mais que se passe-t-il si vous souhaitez additionner toute la matrice? Vous pouvez utiliser:

```
s = sum(sum(A))
```

Mais que faire si un tableau ND? appliquer la `sum` sur la `sum` sur la `sum` ... ne semble pas être la meilleure option, utilisez plutôt l'opérateur `:` pour vectoriser votre tableau:

```
s = sum(A(:))
```

et cela se traduira par un nombre qui est la somme de tout votre tableau, peu importe combien de dimensions il a.

Utilisation de bsxfun

Très souvent, la raison pour laquelle le code a été écrit dans une boucle `for` est de calculer des valeurs à partir de celles situées à proximité. La fonction `bsxfun` peut souvent être utilisée pour le faire de manière plus succincte.

Par exemple, supposons que vous souhaitiez effectuer une opération par colonnes sur la matrice `B` , en soustrayant la moyenne de chaque colonne:

```
B = round(randn(5)*10);                  % Generate random data
A = zeros(size(B));                      % Preallocate array
for col = 1:size(B,2);                    % Loop over columns
    A(:,col) = B(:,col) - mean(B(:,col));   % Subtract means
end
```

Cette méthode est inefficace si `B` est grand, souvent dû au fait que MATLAB doit déplacer le contenu des variables en mémoire. En utilisant `bsxfun` , on peut faire le même travail facilement et facilement en une seule ligne:

```
A = bsxfun(@minus, B, mean(B));
```

Ici, `@minus` est un handle de fonction pour l'opérateur `minus` (`-`) et sera appliqué entre les éléments des deux matrices `B` et `mean(B)` . D'autres poignées de fonction, même celles définies par l'utilisateur, sont également possibles.

Supposons ensuite que vous souhaitiez ajouter un vecteur de ligne `v` à chaque ligne de la matrice `A` :

```
v = [1,  2,  3];

A = [8,  1,  6
     3,  5,  7
     4,  9,  2];
```

L'approche naïve consiste à utiliser une boucle (*ne le faites **pas***):

```
B = zeros(3);
for row = 1:3
    B(row,:) = A(row,:) + v;
end
```

Une autre option serait de répliquer `v` avec `repmat` (*ne le faites **pas non** plus*):

```
>> v = repmat(v,3,1)
v =
     1     2     3
     1     2     3
     1     2     3

>> B = A + v;
```

Utilisez plutôt `bsxfun` pour cette tâche:

```
>> B = bsxfun(@plus, A, v);
B =
     9     3     9
     4     7    10
     5    11     5
```

Syntaxe

```
bsxfun(@fun, A, B)
```

où `@fun` est l'une des fonctions prises en charge et que les deux tableaux `A` et `B` respectent les deux conditions ci-dessous.

Le nom `bsxfun` aide à comprendre comment fonctionne la fonction et il est synonyme de **FUN B** inaire ction avec **S** Ingleton e **X** pansion. En d'autres termes, si:

1. deux tableaux partagent les mêmes dimensions sauf un
2. et la dimension discordante est un singleton (c'est-à-dire a une taille de `1`) dans l'un ou l'autre des deux tableaux

alors le tableau avec la dimension singleton sera développé pour correspondre à la dimension de l'autre tableau. Après l'extension, une fonction binaire est appliquée par élément sur les deux tableaux.

Par exemple, soit `A` un tableau `M` -by- `N` -by `K` et `B` un tableau `M` -by- `N` . Premièrement, leurs deux premières dimensions ont des tailles correspondantes. Deuxièmement, `A` a `K` couches alors que `B` a implicitement seulement `1` , donc c'est un singleton. Toutes les **conditions** sont remplies et `B` sera répliqué pour correspondre à la 3ème dimension de `A`

Dans d'autres langues, il s'agit généralement de *diffusion* et se produit automatiquement en Python (numpy) et Octave.

La fonction, `@fun` , doit être une fonction binaire, ce qui signifie qu'elle doit prendre exactement deux entrées.

Remarques

En interne, `bsxfun` ne réplique pas le tableau et exécute une boucle efficace.

Masquage logique

MATLAB prend en charge l'utilisation du masquage logique afin d'effectuer une sélection sur une matrice sans utiliser des boucles for ou des instructions if.

Un masque logique est défini comme une matrice composée de `1` et `0` .

Par exemple:

```
mask = [1 0 0; 0 1 0; 0 0 1];
```

est une matrice logique représentant la matrice d'identité.

Nous pouvons générer un masque logique en utilisant un prédicat pour interroger une matrice.

```
A = [1 2 3; 4 5 6; 7 8 9];
B = A > 4;
```

Nous créons d'abord une matrice 3x3, `A` , contenant les nombres de 1 à 9. Nous interrogeons ensuite `A` pour les valeurs supérieures à 4 et stockons le résultat dans une nouvelle matrice appelée `B`

`B` est une matrice logique de la forme:

```
B = [0 0 0
```

```
     0 1 1
     1 1 1]
```

Ou `1` lorsque le prédicat `A > 4` était vrai. Et `0` quand c'était faux.

Nous pouvons utiliser des matrices logiques pour accéder aux éléments d'une matrice. Si une matrice logique est utilisée pour sélectionner des éléments, les indices où un `1` apparaît dans la matrice logique seront sélectionnés dans la matrice que vous sélectionnez.

En utilisant le même `B` ci-dessus, nous pourrions faire ce qui suit:

```
C = [0 0 0; 0 0 0; 0 0 0];
C(B) = 5;
```

Cela sélectionnerait tous les éléments de `C` où `B` a un `1` dans cet index. Ces indices dans `C` sont alors mis à `5` .

Notre `C` ressemble maintenant à:

```
C = [0 0 0
     0 5 5
     5 5 5]
```

Nous pouvons réduire les blocs de code complexes contenant `if` et `for` en utilisant des masques logiques.

Prenez le code non vectoriel:

```
A = [1 3 5; 7 9 11; 11 9 7];
for j = 1:length(A)
  if A(j) > 5
    A(j) = A(j) - 2;
  end
end
```

Cela peut être raccourci en utilisant le masquage logique au code suivant:

```
A = [1 3 5; 7 9 11; 11 9 7];
B = A > 5;
A(B) = A(B) - 2;
```

Ou même plus court:

```
A = [1 3 5; 7 9 11; 11 9 7];
A(A > 5) = A(A > 5) - 2;
```

Extension implicite de la matrice (diffusion) [R2016b]

MATLAB R2016b comportait une généralisation de son mécanisme d'expansion scalaire [1], [2] , pour prendre en charge également certaines opérations entre éléments de *tableaux* de tailles

différentes, à condition que leur dimension soit compatible. Les opérateurs qui prennent en charge l'expansion implicite sont [1] :

- **Opérateurs arithmétiques élémentaires:** `+` , `-` `./` `.*` `./` `.^` , `./` `.\` .
- **Opérateurs relationnels:** `<` , `<=` , `>` , `>=` , `==` , `~=` .
- **Opérateurs logiques:** `&` , `|` , ou `xor` .
- **Fonctions** `bitand` **:** `bitand` , `bitor` , `bitxor` .
- **Fonctions mathématiques élémentaires:** `max` , `min` , `mod` , `rem` , `hypot` , `atan2` , `atan2d` .

Les opérations binaires susmentionnées sont autorisées entre les baies, à condition qu'elles aient des "tailles compatibles". Les tailles sont considérées comme "compatibles" lorsque chaque dimension d'un tableau est soit exactement égale à la même dimension dans l'autre tableau, soit égale à `1` . Notez que les dimensions de singleton final (c'est-à-dire de taille `1`) sont omises par MATLAB, même s'il en existe théoriquement une quantité infinie. En d'autres termes, les dimensions qui apparaissent dans un tableau et n'apparaissent pas dans l'autre sont implicitement adaptées à une expansion automatique.

Par exemple, dans les versions MATLAB **antérieures à R2016b,** cela se produirait:

```
>> magic(3) + (1:3)
Error using  +
Matrix dimensions must agree.
```

Considérant que à **partir de R2016b** l'opération précédente réussira:

```
>> magic(3) + (1:3)
ans =

     9     3     9
     4     7    10
     5    11     5
```

Exemples de tailles compatibles:

La description	1ère taille de tableau	2 ème taille de tableau	Taille du résultat
Vecteur et scalaire	`[3x1]`	`[1x1]`	`[3x1]`
Vecteurs de lignes et de colonnes	`[1x3]`	`[2x1]`	`[2x3]`
Matrice vectorielle et 2D	`[1x3]`	`[5x3]`	`[5x3]`
Tableaux ND et KD	`[1x3x3]`	`[5x3x1x4x2]`	`[5x3x3x4x2]`

Exemples de tailles incompatibles:

La description	1ère taille de tableau	2 ème taille de tableau	Solution de rechange possible
Vecteurs où une dimension est un multiple de la même dimension dans l'autre tableau.	`[1x2]`	`[1x8]`	`transpose`
Tableaux avec des dimensions qui sont des multiples les uns des autres.	`[2x2]`	`[8x8]`	`repmat` , `reshape`
Les tableaux ND qui ont la bonne quantité de singleton mais dans le mauvais ordre (# 1).	`[2x3x4]`	`[2x4x3]`	`permute`
Les tableaux ND qui ont la bonne quantité de singleton mais qui sont dans le mauvais ordre (# 2).	`[2x3x4x5]`	`[5x2]`	`permute`

IMPORTANT:
Le code reposant sur cette convention n'est **PAS** compatible avec *les* anciennes versions de MATLAB. Par conséquent, l'appel explicite de `bsxfun` [1, 2] (qui produit le même effet) doit être utilisé si le code doit être exécuté sur d'anciennes versions de MATLAB. Si un tel problème n'existe pas, les notes de publication de MATLAB R2016 encouragent les utilisateurs à passer de `bsxfun` :

> *Par rapport à l'utilisation de* `bsxfun` *, l'expansion implicite offre une vitesse d'exécution plus rapide, une meilleure utilisation de la mémoire et une meilleure lisibilité du code.*

Lecture connexe:

- Documentation MATLAB sur " *Tailles de matrices compatibles pour les opérations de base* ".
- NumPy Broadcasting [1, 2] .
- Une comparaison entre la vitesse de calcul à l'aide de `bsxfun` et l'expansion de tableau implicite .

Obtenir la valeur d'une fonction de deux arguments ou plus

Dans de nombreuses applications, il est nécessaire de calculer la fonction de deux arguments ou plus.

Traditionnellement, nous utilisons `for` boucles. Par exemple, si nous devons calculer le `f = exp(-x^2-y^2)` (ne l'utilisez pas si vous avez besoin **de simulations rapides**):

```
% code1
x = -1.2:0.2:1.4;
y = -2:0.25:3;
for nx=1:lenght(x)
   for ny=1:lenght(y)
      f(nx,ny) = exp(-x(nx)^2-y(ny)^2);
   end
end
```

Mais la version vectorisée est plus élégante et plus rapide:

```
% code2
[x,y] = ndgrid(-1.2:0.2:1.4, -2:0.25:3);
f = exp(-x.^2-y.^2);
```

que nous pouvons le visualiser:

```
surf(x,y,f)
```

Remarque 1 - Grilles: généralement, le stockage matriciel est organisé *ligne par ligne* . Mais dans le MATLAB, il s'agit du stockage *colonne par colonne* , comme dans FORTRAN. Ainsi, il existe deux fonctions simulaires `ndgrid` et `meshgrid` dans MATLAB pour implémenter les deux modèles susmentionnés. Pour visualiser la fonction dans le cas de `meshgrid` , on peut utiliser:

```
surf(y,x,f)
```

Note2 - La consommation de mémoire: Soit la taille de `x` ou `y` est 1000. Ainsi, nous avons besoin de stocker `1000*1000+2*1000 ~ 1e6` éléments pour ***code1*** non vectorisé. Mais il faut `3*(1000*1000) = 3e6` éléments dans le cas du ***code*** vectorisé2. Dans le cas 3D (laissez `z` avoir la même taille que `x` ou `y`), la consommation de mémoire augmente considérablement: `4*(1000*1000*1000)` (~ 32 Go pour les doubles) dans le cas du ***code*** vectoriel2 vs `~1000*1000*1000` (seulement ~ 8 Go) dans le cas du ***code1*** . Il faut donc choisir entre la mémoire ou la vitesse.

Lire Vectorisation en ligne: https://riptutorial.com/fr/matlab/topic/750/vectorisation

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec MATLAB Language	adjpayot, Amro, chrisb2244, Christopher Creutzig, Community, Dan, Dev-iL, DVarga, EBH, Erik, excaza, flawr, Franck Dernoncourt, fyrepenguin, GameOfThrows, H. Pauwelyn, honi, Landak, Lior, Matt, Mikhail_Sam, Mohsen Nosratinia, Sam Roberts, Shai, Tyler
2	Applications financières	Amro, Franck Dernoncourt, Mikhail_Sam, Oleg, Royi, StefanM
3	Astuces utiles	Celdor, EBH, Erik, fyrepenguin, Malick
4	Conditions	ammportal, EBH
5	Contrôle de la coloration des sous-parcelles dans Matlab	Noa Regev
6	Décompositions matricielles	StefanM
7	Définir les opérations	Shai, StefanM
8	Dessin	il_raffa, nitsua60, NKN, thewaywewalk, Trilarion, Zep
9	Erreurs communes et erreurs	Amro, Ander Biguri, Dev-iL, EBH, edwinksl, erfan, Franck Dernoncourt, Hoki, Landak, Malick, Matt, Mikhail_Sam, Mohsen Nosratinia, nahomyaja, NKN, Oleg, R. Joiny, rafa, rayryeng, Rody Oldenhuis, S. Radev, Sardar Usama, strpeter, Suever, Thierry Dalon, Tim, Umar
10	Fonctionnalités non documentées	Amro, codeaviator, Dev-iL, Erik, matlabgui, thewaywewalk
11	Fonctions de documentation	alexforrence, Ander Biguri, ceiltechbladhm, Dev-iL, Eric, excaza
12	Graphiques: Tracés de lignes 2D	Amro, Celdor, EBH, Erik, Matt, Oleg, StefanM
13	Graphiques: Transformations 2D	itzik Ben Shabat, Royi

	et 3D	
14	Initialisation de matrices ou de tableaux	Parag S. Chandakkar, rajah9, Théo P.
15	Interfaces utilisateur MATLAB	Dev-iL, Hoki, Royi, Suever, Zep
16	Interpolation avec MATLAB	Royi, StefanM
17	Introduction à l'API MEX	Amro, Ander Biguri, Kamiccolo, Matt, mike
18	L'intégration	StefanM
19	Le débogage	Dev-iL, DVarga, jenszvs, Justin
20	Lecture de gros fichiers	JCKaz
21	Les fonctions	Batsu
22	Meilleures pratiques MATLAB	il_raffa, Malick, MayeulC, McLemon, mnoronha, NKN, rayryeng, Sardar Usama, Thierry Dalon
23	Multithreading	Adriaan, daren shan, Franck Dernoncourt, Hardik_Jain, Jim, Peter Mortensen
24	Performance et Benchmarking	Celdor, daleonpz, Dev-iL, il_raffa, Oleg, pseudoDust, thewaywewalk
25	Pour les boucles	agent_C.Hdj, drhagen, EBH, Tyler
26	Programmation orientée objet	alexforrence, Celdor, daren shan, Dev-iL, jenszvs, Mohsen Nosratinia, Trogdor
27	Solveurs des équations différentielles ordinaires (ODE)	Royi, StefanM
28	Traitement d'image	A.Youssouf, Ander Biguri, Cape Code, girish_m, Royi, Shai, Trilarion
29	Transformées de Fourier et Transformées de Fourier Inverse	GameOfThrows, Landak

30	Utilisation de fonctions avec sortie logique	Landak, Mikhail_Sam, Mohsen Nosratinia, S. Radev, Trilarion
31	Utilisation de la fonction `accumarray ()`	Dev-iL, Royi
32	Utiliser des ports série	Abdul Rehman, Ander Biguri, Hoki, Kenn Sebesta, mhopeng, Oleg
33	Utilitaires de programmation	Celdor, The Vivandiere
34	Vectorisation	Alexander Korovin, Ander Biguri, Dan, Dev-iL, EBH, Landak, Matt, Oleg, Shai, Tyler