TP nº 4 - Un dessin vaut mieux qu'un long discours

Tous les sujets et les corrigés sont disponibles aux adresses suivantes :
`http://www.lif.univ-mrs.fr/~vpoupet/enseignement/matlab09.php`
`http://www.lif.univ-mrs.fr/~pvanier/?q=cours`

Exercice 1. *Révisions*

Quelques rappels parce que vous avez probablement oublié toute la syntaxe *Matlab* pendant les vacances.
Supposons que l'on ait un script `mystere.m` contenant le code suivant :

```
MAX = 50;
tab = 1:MAX;
tab(1) = 0;
while 0 == 0
    i = 1;
    while tab(i) == 0
        i = i + 1;
    end
    p = tab(i)
    for k = p:p:MAX
        tab(k) = 0;
    end
end
```

1. Vérifiez que vous comprenez bien ce que fait chacune des lignes du programme (chacune prise indépendamment).

2. Que fait le programme lorsqu'on l'exécute ? (recopiez-le pour le tester)
Remarque : Le script est assez mal écrit parce qu'il s'arrête en provoquant une erreur. C'est mal mais ça simplifie un peu le code.

Réponse : Ce programme affiche les nombres premiers inférieurs à 50 en utilisant la méthode du crible d'Eratosthène :
- on crée un tableau contenant les entiers de 1 à 50 ;
- on « supprime » 1 du tableau en mettant la case 1 à 0 parce que 1 n'est pas un nombre premier ;
- la boucle `while 0 == 0` est exécutée jusqu'à ce que le programme provoque une erreur ($0 = 0$ est toujours vrai) ;
- on parcourt le tableau en cherchant la première case non nulle ;
- ce nombre est un nombre premier, on l'enregistre dans la variable p et on l'affiche (pas de point virgule en fin de ligne) ;
- puis on annule toutes les cases du tableau qui sont des multiples de p (qui ne sont donc pas des nombres premiers) ;
- on recommence jusqu'à ce que toutes les cases du tableau soient nulles.

Lorsque toutes les cases du tableau sont nulles la boucle qui cherche la première case non nulle finit par essayer de trouver la valeur de la 51-ième case du tableau qui n'existe pas. Cela provoque une erreur qui arrête le déroulement du programme.

Exercice 2. *Tracé de fonctions*

Nous allons maintenant voir comment on peut utiliser *Matlab* pour représenter graphiquement des fonctions, parce qu'au fond, faire des calculs sur des matrices ça va 5 minutes mais faire des dessins c'est plus marrant.
La commande de base permettant de dessiner des graphes est `plot`. Sous sa forme la plus simple, `plot` permet de représenter les valeurs d'un tableau :

```
>> plot([1, 2, 4, 8]);
>> plot([1; 2; 1; 2]);
```

Si l'argument de `plot` est une matrice à plusieurs lignes, *Matlab* trace chacune des colonnes séparément :

```
>> m = [1, 2; 1, 3; 1, 2]
>> plot(m)
```

On peut également modifier l'apparence du tracé en ajoutant un argument final composé de symboles cryptiques entre guillemets simples :

```
>> plot([1, 1, 1, 2, 4, 8, 4], 'r+--')
>> plot([2, 3, 4, 5, 3, 1], 'md:')
```

La commande `help plot` donne plus de détails, en particulier la liste des codes permettant de choisir la couleur du tracé, la forme des marqueurs des points (croix, cercle, losange, ...) et le type de tracé (pointillé, plein, tirets, etc.).
On peut également (et c'est ce que l'on va utiliser le plus souvent par la suite) représenter les valeurs d'un tableau `y` en fonction d'un tableau `x` :

```
>> x = 0:.1:1
>> y = x.^2
>> plot(x, y)
```

1. Dans l'exemple précédent, que fait la ligne `y = x.^2` ? Quelle différence y a-t-il entre `x^2` et `x.^2` ?

Réponse : Par défaut *Matlab* considère que tout est une matrice. Donc l'opérateur d'exponentiation `^` est l'exponentiation matricielle (qui ne fonctionne que sur des matrices carrées). Pour préciser que l'on veut utiliser l'exponentiation sur les réels (et donc l'appliquer individuellement à chaque case du tableau) il faut utiliser la notation `.^`.

Le tableau des abscisses (premier des deux arguments) ne doit pas nécessairement être régulier, ni même croissant. On peut par exemple utiliser `plot` pour tracer des courbes paramétriques...
La *cycloïde* est la courbe que dessine un point sur la circonférence d'une roue de vélo lorsque celui-ci (le vélo) avance (figure 1).

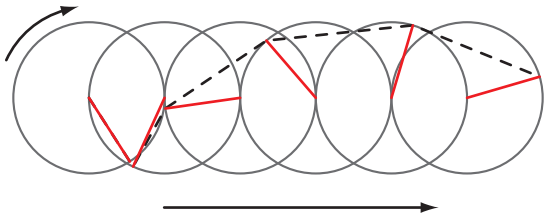

FIGURE 1 – Mouvement d'un point de la circonférence d'une roue de vélo.

Au temps t, l'ordonnée du point est $sin(-t)$ (on prend $-t$ parce que la roue tourne dans le sens inverse trigonométrique) et son abscisse est $cos(-t) + t$ (parce que le centre de la roue a avancé de t).

2. Définissez un tableau `t` contenant les valeurs entre 0 et 20 espacées de $0,1$ (ce sera l'ensemble des temps auxquels on veut dessiner un point).

Réponse :

```
>> t = 0:.1:20;
```

3. Définissez les tableaux `x` et `y` contenant les abscisses et les ordonnées des points de la cycloïde pour chacun des temps dans le tableau `t` (un peu comme quand on a écrit `y = x.^2` plus haut).

Réponse :

```
>> x = t + cos(-t);
>> y = sin(-t);
```

4. Tracez la cycloïde à l'aide de la commande

```
>> plot(x, y, 'o-')
```

Le problème que l'on peut remarquer sur la figure précédente est que les axes ne sont pas orthonormés, ce qui déforme le tracé de la fonction. Par défaut *Matlab* choisit les unités sur les axes de telle sorte que l'ensemble du tracé demandé tienne dans un rectangle prédéfini, mais il est possible de modifier manuellement les axes à l'aide de la commande `axis`.

5. Exécutez les instructions suivantes et observez l'effet qu'elles ont sur la figure précédemment tracée :

```
>> axis equal
>> axis square
>> axis normal
>> axis([2 8 0 1])
```

Vous aurez probablement remarqué qu'à chaque fois qu'on trace une nouvelle courbe, la précédente disparaît. C'est passablement gênant quand on essaie de comparer plusieurs fonctions et que l'on voudrait donc les tracer ensemble sur la même figure. Il est cependant possible de superposer des courbes de différentes manières (au moins deux à ma connaissance).
L'instruction `hold on` (qui s'annule à l'aide de la commande `hold off`) indique que les nouvelles fonctions à tracer doivent être dessinées avec les précédentes :

```
>> x = -pi:pi/20:pi;
>> y1 = cos(x);
>> y2 = sin(x);
>> y3 = -cos(x);
>> plot(x, y1, 'b')
>> hold on
>> plot(x, y2, 'g')
>> plot(x, y3, 'r')
>> hold off
```

Sinon, on peut aussi donner toutes les courbes à tracer en même temps à la fonction `plot` :

```
>> plot(x, y1, '+r-', x, y2, 'g--', x, y3, ':')
```

Voici quelques autres fonctions qui peuvent servir[1] :
– `title` : titre du graphe
– `xlabel`, `ylabel` et `zlabel` : étiquette sur les axes
– `legend` : légende (pas le film avec Tom Cruise...)
– `text` : permet de placer du texte

1. regardez l'aide de chaque fonction si vous ne comprenez pas comment elle fonctionne, ou demandez directement au chargé de TP qui est un peu là pour ça...

Exercice 3. *Prem's*

Utilisons les tracés de fonctions pour vérifier expérimentalement un résultat sur la répartition des nombres premiers :

Théorème (Felgner (1990)). *Pour tout entier n, le n-ème nombre premier (noté p_n) vérifie*

$$0,91n\ln(n) < p_n < 1,7n\ln(n)$$

1. Définissez un tableau `p` contenant les nombres premiers inférieurs ou égaux à 10000.
Indication : C'est exactement ce que fait la fonction `primes`, qui prend en argument un entier...).

Réponse :

```
>> p = primes(10000);
```

2. Déterminez le nombre N de nombres premiers inférieurs à 10000 en utilisant la fonction `size`.

Réponse :

```
>> size(p)
ans =
           1        1229
```

Donc p est un tableau à une ligne et 1229 colonnes. Il y a 1229 nombres premiers inférieurs à 10000.

3. Tracez sur une même figure les fonctions $f : n \mapsto 0,91n\ln(n)$ (en vert), $g : n \mapsto 1,7n\ln(n)$ (en bleu) et $\phi : n \mapsto p_n$ (en rouge).

Réponse :

```
>> n = 1:1229;
>> f = 0.91 * n .* log(n);
>> g = 1.7 * n .* log(n);
>> plot(n, f, 'g', n, g, 'b', n, p, 'r');
```

4. Le résultat de Felgner vous semble-t-il vrai jusqu'à la valeur 10000 ?

Exercice 4. *3D*

Il est également possible de demander à *Matlab* de tracer des courbes ou des surfaces en 3 dimensions à l'aide des commandes `plot3`, `ezmesh` et `ezsurf`.
Recopiez la fonction suivante dans un script intitulé `sombrero.m` (ou un autre nom si celui-ci est déjà-pris... comme d'habitude) :

```
function z = sombrero(x, y)
r = sqrt(x.^2 + y.^2);
z = sin(r)./r;
```

1. En regardant un peu la définition de la fonction $\mathrm{sombrero}(x, y)$, essayez de vous représenter la forme de la surface qu'elle définit.

Représentez la surface sous la forme d'un maillage à l'aide de l'instruction

```
>> ezmesh(@sombrero)
```

Si vous voulez changer les axes, la fonction `axis` fonctionne toujours...
Si vous voulez représenter la surface de manière plus lisse, utilisez les commandes

```
>> ezsurf(@sombrero)
>> shading interp
```