TP5 : Les fonctions sous MATLAB et l'interpolation

Cette séance de TP5 poursuit la familiarisation avec MATLAB. Le chapitre 3 du cours, en particulier ce qui concerne l'interpolation de Lagrange (section 3.2 du cours) ou l'interpolation par les polynômes trigonométriques (et ses conséquences : algorithmes de transformation de Fourier rapide, multiplication rapide des polynômes[1], *etc.*, *cf.* la section 3.1.4 du cours), fournira la trame théorique de ce TP5. Ouvrez MATLAB pour commencer.

EXERCICE 1 (Différences divisées et polynôme de Lagrange).

(1) Depuis le site web

`http://www.math.u-bordeaux1/~yger/initiationMATLAB`

téléchargez le fichier `DiffDiv.m` correspondant à la routine fournissant (dans cet ordre), la liste des `N+1` différences divisées :

`y[x(1),...,x(N+1)], y[x(1),...,x(N)],..., y[x(1)]`

attachée à une liste de nombres complexes `Y=[y(1),...,y(N+1)]` « divisée » par une liste de nombres complexes distincts

`X=[x(1),...,x(N+1)]`.

Après avoir enregistré ce fichier `DiffDiv.m` dans votre répertoire `TPMATLAB`, ouvrez le et vérifiez que la syntaxe de la fonction proposée correspond bien aux formules inductives sous tendant le tableau proposé page 51 du polycopié de cours. Montrez que le nombre de multiplications impliquées dans ce calcul est en $O(N^2)$.

(2) Ouvrez un nouveau fichier `.m`, vierge cette fois. En vous inspirant du schéma de Hörner (*cf.* la syntaxe page 40 du polycopié) et de la formule de Newton

$$\texttt{Lagrange[X;Y](z)} = \\ = \texttt{y[x(1)]} + \sum_{k=2}^{N+1} \texttt{y[x(1),...,x(k)]} * (z - \texttt{x(1)}) * \cdots * (z - \texttt{x(k-1)})$$

(formule (3.8) du polycopié, page 50), rédigez dans ce fichier une fonction :

`function LXX = LagrangeNewton (X,Y,XX)`

qui, étant donnée une liste de nombres complexes distincts `X` (de longueur `N+1`) et une liste de nombres complexes `Y` de même longueur, évalue le polynôme de Lagrange `Z` $\mapsto$ `Lagrange[X;Y](Z)` aux points de la liste

1. Ceci sera repris dans un TP ultérieur sous `Maple12`, cette fois dans le cadre du calcul arithmétique sans pertes, sous l'angle du calcul symbolique et de la cryptograhie.

de nombres complexes `XX` (de longueur arbitraire) et renvoie en sortie la liste `LXX` (de même longueur que `XX`) correspondant aux valeurs prises par ce polynôme de Lagrange aux entrées de `XX` (en respectant l'ordre de ces entrées). La fonction `DiffDiv` sera utilisée comme fonction auxiliaire. Sauvez ce fichier `.m` dans votre répertoire de travail (*Workspace*, ici `TPMATLAB`) comme `LagrangeNewton.m`. Testez ce fichier en déclarant la fonction

```
f = inline('exp(-x.^2/2).*sin(pi*x)','x');
```

puis en déclarant :

```
>> X= -3:.1:3 ;
>> XX=-3:.05:3 ;
>> LXX=LagrangeNewton(X,f(X),XX);
>> plot(XX,f(XX),'r');
>> hold
>> plot(XX,LXX,'k');
```

Qu'observez vous ? Recommencez avec cette fois les instructions

```
>> X= -6:.2:6 ;
>> XX=-6:.05:6 ;
>> LXX=LagrangeNewton(X,f(X),XX);
>> figure
>> plot(XX,f(XX),'r');
>> hold
>> plot(XX,LXX,'k');
```

Qu'observez vous cette fois ? Hors de quel segment de $[-6, 6]$ l'approximation de `f` par son polynôme de Lagrange se met-elle à dérailler ? Recommencez en prenant cette fois plus de points d'interpolation :

```
>> X=-6:.15:6;
>> XX=-6:.05:6 ;
>> LXX=LagrangeNewton(X,f(X),XX);
>> figure
>> plot(XX,f(XX),'r');
>> hold
>> plot(XX,LXX,'k');
```

Les choses s'arrangent-elles ou empirent-elles par rapport à la figure 2 ? Continuez avec encore plus de points :

```
>> X=-6:.1:6 ;
>> XX=-6:.05:6 ;
>> LXX=LagrangeNewton(X,f(X),XX);
>> figure
>> plot(XX,f(XX),'r');
>> hold
>> plot(XX,LXX,'k');
```

Vous observez ici le *phénomène de Runge*. Voir pour plus de détails le site sur `wikipedia` :

`http://fr.wikipedia.org/wiki/Ph%C3%A9nom%C3%A8ne_de_Runge`

(3) Ouvrez un nouveau fichier `.m` vierge et rédigez dans ce fichier un code

`function LXXT= LagrangeTchebychev(a,b,N,f,XX)`

qui, étant donné un segment fermé borné [a,b] de $\mathbb{R}$ (donné par deux bornes distinctes `a` et `b` déclarées en flottants), un entier strictement positif `N`, une fonction `f` de [a,b] dans $\mathbb{R}$ ou $\mathbb{C}$ déclarée `inline('f(x)','x')` au préalable (voir l'exercice 3 du TP 3), et une liste `XX` de points du segment [a,b], renvoie la liste des valeurs aux points de `XX` prises par le polynôme d'interpolation de Lagrange interpolant les `N+1` valeurs de la fonction `f` aux `N+1` points du segment [a,b] donnés par

$$\texttt{X(k)} = \frac{\texttt{a}+\texttt{b}}{2} + \Big(\frac{\texttt{b}-\texttt{a}}{2}\Big)\cos\Big(\frac{\texttt{(2*k-1)*pi}}{\texttt{2*(N+1)}}\Big), \quad \texttt{k=1,...,N+1}.$$

Ces points sont déduits par homothétie et translation des `N+1` zéros dans [-1,1] du polynôme de Tchebychev T_{N+1} donné par

$$T_{N+1}(\cos(\theta) = \cos((N+1)\theta) \qquad \forall\, \theta \in \mathbb{R}.$$

Vous serez amenés à utiliser la fonction `LagrangeNewton` construite à la question 2 comme fonction auxiliaire appelée lors de l'exécution du code correspondant à la fonction `LagrangeTchebychev`. Sauvez le fichier dans votre répertoire de travail (*Workspace*, ici `TPMATLAB`) comme le fichier `LagrangeTchebychev.m`. Testez le en prenant la fonction `f` de la question 2 :

```
>> f = inline('exp(-x.^2/2).*sin(pi*x)','x');
>> XX=-6:.05:6 ;
>> LXXT=LagrangeTchebychev(-6,6,30,f,XX);
>> figure
>> plot(XX,f(XX),'r');
>> hold
>> plot(XX,LXXT,'k');
```

Qu'observez vous maintenant en comparaison avec le phénomène de Runge observé à la question 2 ? Prenez des valeurs de `N` plus grandes que `N=30`, par exemple `N=35,40,45,50` et recommencez. La situation empire-t'elle encore cette fois ? Que se passe-t'il malgré tout si l'on persiste à augmenter `N` (prenez `N=60` pour voir). Pourquoi cette liste de points déduite des zéros du polynôme de Tchebychev T_{N+1} se comporte-t-elle mieux du point de vu de la qualité de l'interpolation au bord (si on la compare à l'interpolation de Lagrange avec des points équidistribués sur [a,b] (comme à la question 2) ? Expliquez pourquoi les choses se gâtent malgré tout lorsque `N` est pris trop grand (exemple `N=60`).

Exercice 2 (la méthode récursive de Neville-Aitken).

(1) Téléchargez depuis le site

`http://www.math.u-bordeaux1/~yger/initiationMATLAB`

le fichier `lagrange.m` (il a été réactualisé, aussi devez vous le re-télécharger même si vous l'avez déjà). Ouvrez ce fichier et vérifiez que la syntaxe du code récursif écrit ici est bien celle conduisant à la construction du polynôme d'interpolation de Lagrange suivant la démarche récursive de Neville-Aitken, telle qu'elle est présentée pages 52 et 53 du polycopié.

(2) Ouvrez un fichier `.m` vierge et, en vous inspirant du code `lagrange`, rédigez un code

```
function  LXXN=NevilleAitken(a,b,N,f)
```

qui, étant donné un segment fermé borné [a,b] de $\mathbb{R}$ (donné par deux bornes distinctes `a` et `b` déclarées en flottants), un entier strictement positif `N`, une fonction `f` de [a,b] dans $\mathbb{R}$ ou $\mathbb{C}$ déclarée `inline('f(x)','x')` au préalable (voir l'exercice 3 du TP 3), renvoie (sous la forme de la liste de ses coefficients) le polynôme d'interpolation de Lagrange interpolant les `N+1` valeurs de la fonction `f` aux `N+1` points du segment [a,b] équirépartis entre `X(1)=a` et `X(N+1)=b`. Testez ensuite ce code pour représenter (avec des valeurs de `N` égales à `N=5,7,10,12`) les graphes des interpolées sur $[-1,1]$ avec pas régulier des fonctions

```
f1= inline('x.^7+3*x.^5-2*x+1','x');
f2= inline ('x.*cos(x) -log(x+2)+1','x');
f3= inline('1./(1+x.^2)','x');
f4= inline('(1+x.^2-x.^3).*exp(x/5)','x');
```

Pensez pour cela à évaluer au préalable votre polynôme `LXXN` (dont `LXXN` figure juste la liste des coefficients) sur le vecteur `XX` des `N+1` points de $[-1,1]$ régulièrement espacés de `1/N` :

```
>> XX=-1:1/N:1;
>> LXXN=NevilleAitken(-1,1,N,f);
>> LXXNevalue=polyval(LXXN,XX);
```

Vous pouvez aussi utiliser, à la place de `polyval`, la routine `Horner` disponible ici :

`http://www.math.u-bordeaux1/~yger/initiationMATLAB`

Justifiez la qualité des résultats obtenus en vous référant aux formules de Taylor explicites (Taylor-Lagrange ou Taylor avec reste intégral, *cf.* votre cours d'Analyse 1 de S2 [2]).

EXERCICE 3 (multiplication rapide des polynômes et `fft`).

(1) Sous `MATLAB`, la routine effectuant la multiplication à gauche d'un vecteur colonne de nombres complexes, de longueur $\mathtt{N} = 2^p$ ($p \in \mathbb{N}^*$), par la matrice

$$\left[W_{\mathtt{N}}^{kj}\right]_{0\leq j,k\leq \mathtt{N}-1} \qquad (\text{où } W_{\mathtt{N}} = \exp(-2i\pi/\mathtt{N}))$$

suivant l'algorithme de Cooley-Tukey (impliquant seulement $(p-1)2^{p-1}$ « vraies » multiplications au lieu des 2^{2p} attendues, *cf.* le cours pages 44-45) est la commande

```
>> Y=fft(X,N);
```

En utilisant la relation

$$\left(\left[W_{\mathtt{N}}^{kj}\right]_{0\leq j,k\leq \mathtt{N}-1}\right)^{-1} = \frac{1}{\mathtt{N}}\left[\overline{W_{\mathtt{N}}{}^{kj}}\right]_{0\leq j,k\leq \mathtt{N}-1},$$

écrire sur un fichier `.m` vierge une routine `InverseFFT` :

2. En ligne : `http://www.math.u-bordeaux1.fr/~yger/analyse1.pdf`, sections 2.5.3 et 2.5.5.

`function X=InverseFFT(Y,N)`

qui, étant donné un vecteur colonne `Y` de nombres complexes, de longueur `N`, calcule la multiplication de `Y` à gauche par la matrice

$$\left(\left[W_N^{kj}\right]_{0\leq j,k\leq N-1}\right)^{-1}$$

(utilisez la routine `conj` qui conjugue les entrées des vecteurs ou des matrices). Validez ce code sur un exemple, par exemple :

```
>> X=rand(16,1) + i*rand(16,1);
>> Y=fft(X,16);
>> XX=InverseFFT(Y,16);
>> max(abs(X-XX))
```

Que constatez vous ? Pourquoi ne trouvez vous pas `0` comme vous vous y attendriez ? Ne perdez pas de vue que vous êtes ici en train de faire du calcul scientifique, et non du calcul symbolique. On reviendra dans un TP ultérieur sous `Maple12` cette fois sur la `fft` discrète, dans le cadre (arithmétique) du calcul symbolique cette fois. Sauvez votre fichier (appelé `InverseFFT`) comme un fichier `.m` dans votre répertoire `TPMATLAB` pour le conserver dans vos archives, tout en sachant que la routine

```
>> X=ifft(X,N);
```

(déjà implémentée dans le noyau du logiciel) correspond de fait à la même fonction.

(2) Ouvrez un fichier `.m` vierge. Rédigez dans ce fichier une procédure

`function  P1P2 = ProduitPolynomes (P1,P2)`

qui, étant donnés deux polynômes `P1` et `P2` déclarés sous forme de listes comme

```
>> P1 = [P1(1) ... P1(N1)];
>> P2 = [P2(1) ... P2(N2)];
```

lorsque

$$\begin{aligned} \texttt{P1}(X) &= \texttt{P1(1)}\,X^{\texttt{N1}-1} + \cdots + \texttt{P1(N1-1)}\,X + \texttt{P1(N1)} \\ \texttt{P2}(X) &= \texttt{P2(1)}\,X^{\texttt{N2}-1} + \cdots + \texttt{P2(N1-1)}\,X + \texttt{P2(N2)}, \end{aligned}$$

retourne (sous forme d'une liste de longueur `N1+N2-1`) les coefficients du polynôme `P1*P2`, les monômes de ce polynôme étant rangés dans l'ordre décroissant. La première étape de la procédure est de déterminer le premier entier `p` tel que $2^p >$ `N1+N2-1` (l'entier le plus proche d'un nombre réel flottant `x` lorsque l'on va à droite de ce nombre est `ceil(x)` ; c'est `floor(x)` si l'on va à gauche). Utilisez ensuite les routines `fft(.,N)` et `ifft(.,N)` comme indiqué dans le cours, avec précisément `N` $= 2^p$, ce après avoir si nécessaire complété les listes `P1` et `P2` par des zéros. Sauvez le fichier `ProduitPolynomes.m` dans votre répertoire `TPMATLAB` après l'avoir testé (et validé) sur des polynômes de bas degré (inférieur ou égal à `10`).

(3) La commande `MATLAB` permettant de déterminer l'écriture binaire d'un entier `M` (bien sûr inférieur à 2^{52}) est

```
>> P = dec2bin(M);
```

mais, attention, la sortie `P` est ici au format `char`, pas au format numérique double précision `double`. Ce format `double` est pourtant nécessaire ici car les calculs de `fft` impliquent la matrice de nombres fottants

$$[W_{\mathtt{N}}^{kj}]_{0\leq j,k\leq \mathtt{N}-1}.$$

Il faut donc convertir `P` au format `double` pour travailler numériquement ; en faisant un petit test, vous verrez que le caractère `0` correspond dans cette conversion à un certain entier positif `P=double('0')`, tandis que `1` correspond (heureusement) à `P+1=double('1')`, ce qu'il convient donc de prendre en compte pour les conversions. À vous de trouver ces deux nombres `double('0')` et `double('1')`, car vous voulez, vous, `0` et `1` comme nombres. Sur un nouveau fichier vierge `.m`, rédigez une procédure

```
function M1M2=ProduitEntiers(M1,M2)
```

qui, étant donnés deux entiers naturels tous deux inférieurs à 2^{26} (cela vaut mieux, dites pourquoi !), utilise les routines `dec2bin`, `fft(.,64)`, `ifft(.,64)` pour calculer leur produit `M1*M2`. Pensez à associer à tout entier naturel `M` (inférieur à 2^{52}) le polynôme

```
>> PolM = double(dec2bin(M))-double('0') ;
```

(pour avoir des coefficients numériquement égaux à `0` et `1` et non plus cette fois à `double('0')` et `double('1')` comme le donne la conversion numérique du format `char` au format `double`), puis ensuite à évaluer le polynôme `PolM1*PolM2` (calculé comme à la question 2) en `x=2`.

Exercice 4 (L'algorithme de Cooley-Tukey). La routine sous `MATLAB`

```
>> Y=fft(X,N);
```

(lorsque `N` est une puissance de 2) correspond à l'implémentation de l'algorithme de Cooley-Tukey (*cf.* le polycopié de cours pages 44-45). Vous pouvez aussi consulter pour plus de détails les sites `wikipedia`

`http://en.wikipedia.org/wiki/Cooley_FFT_algorithm`
`http://fr.wikipedia.org/wiki/Transform%C3%A9e_de_Fourier_rapide`

(le site en anglais est plus riche). Le but de cet exercice (prétexte, comme dans le TP4, à votre familiarisation avec le principe de récursivité) est de réaliser une routine réalisant la même opération, ce afin de comprendre le schéma récursif sur lequel se fonde la procédure. Ouvrez un fichier `.m` vierge sur lequel vous allez rédiger une procédure récursive :

```
function Y=CooleyTukey(X,p)
```

transformant, si `p` désigne un entier supérieur ou égal à `1` et `X` un vecteur colonne de longueur $2^{\mathtt{p}}$, le vecteur `X` en son image par la multiplication à gauche par la matrice

$$\left[W_{2^{\mathtt{p}}}^{jk}\right]_{0\leq j,k\leq 2^{\mathtt{p}-1}}, \qquad \text{où } W_{2^{\mathtt{p}}} := \exp(-2i\pi/2^{\mathtt{p}}). \tag{†}$$

(1) Puisqu'il s'agit de construire un algorithme récursif, vous commencerez par mettre sur pied une boucle

```
if p == 1
Y= ... ;
```

```
else
... ;
... ;
Y= ...;
end
```

en rédigeant d'abord les instructions (très simples) correspondant à l'alternative `p==1`. Ce cas correspond à l'initialisation de la procédure récursive.

(2) Il vous faut maintenant remplir les instructions sous l'alternative `else`. Regardez pour cela l'architecture du programme telle qu'elle est présentée sur le diagramme page 45 du polycopié.
   - Commencez par sérier en deux vecteurs colonne `Xpair` et `Ximpair` (tous deux de longueur $2^{\mathtt{p}-1}$) le vecteur colonne d'entrée `X`.
   - Faites agir sur ces deux vecteurs colonne la procédure au cran $\mathtt{p}-1$ comme suggéré dans le diagramme.
   - Opérez les multiplications terme-à-terme nécessaires concernant le traitement de `Ximpair` après passage à travers la procédure au cran $\mathtt{p}-1$ (regardez bien pour cela le diagramme).
   - Complétez enfin la liste des intructions sous l'alternative `else` par une boucle `do` rendant compte des calculs impliquant l'≪ action papillon ≫ de la matrice correspondant à $\mathtt{p}=1$ (suivez soigneusement la syntaxe du synopsis présenté sur le diagramme).

(3) Validez enfin votre code en comparant les résultats `Y` et `Yref` de

```
X=rand(2^p,1);
Y=CooleyTukey(X,p);
Yref=fft(X,2^p);
```

pour de petites valeurs de `p` (`p=3,4,5`). Sauvez votre fichier `CooleyTukey.m` dans votre répertoire `TPMATLAB`.

(4) Justifiez sur le papier par un raisonnement mathématique pourquoi ce code récursif correspond bien à la multiplication à gauche de `X` par la matrice (†).

EXERCICE 5 (interpolation par des polynômes trigonométriques). Soit `f` une fonction d'une variable réelle (définie sur $[0,1]$, à valeurs réelles) que vous déclarerez ultérieurement (avec une expression explicite pour `f(x)`) *via*

```
>> f= inline('f(x)','x');
```

sous MATLAB (en veillant à ce que la syntaxe permette de calculer `f(x)` lorsque `x` est un vecteur ligne ou colonne de nombres de $[0,1]$, et non seulement un nombre de $[0,1]$). Il est souvent plus judicieux (en particulier si `f` est une fonction oscillante comme on en rencontre dans le monde des télécommunications) de chercher à interpoler `f` sur $[0,1]$ non par des fonctions polynomiales (commme dans l'interpolation de Lagrange), mais par des fonctions polynomiales trigonométriques, du type

$$\theta \in [0,1] \mapsto \sum_{l=-M}^{M-1} u_k \, e^{2i\pi l\theta}, \quad \text{où } M \in \mathbb{N}^*, \tag{$*$}$$

qui sont, elles aussi, des fonctions oscillantes, donc de même nature que la fonction à interpoler `f`. Vous verrez plus tard que c'est le principe de ce que l'on appelle faire l'*analyse de Fourier* de l'information fournie par `f`. Soit $\mathtt{N}=2^{\mathtt{p}}$.

(1) Vérifiez, si `p` est un entier naturel non nul donné, que le système de 2^p équations à 2^p inconnues u_k, $k = -2^{p-1}, ..., 2^{p-1} - 1$,

$$\Big[\sum_{k=-2^{p-1}}^{2^{p-1}-1} u_k\, e^{2i\pi k\theta}\Big]_{\theta=j/2^p} = f(j/2^p), \quad j = 0, ..., 2^p - 1 \qquad (**)$$

(qui traduit le fait que la fonction polynomiale trigonométrique $(*)$, avec $M = 2^{p-1}$, interpole la fonction `f` aux 2^p points $j/2^p$, $j = 0, ..., 2^p - 1$, régulièrement espacés dans $[0, 1]$) se lit aussi

$$\sum_{k=0}^{2^p-1} v_k\, \overline{W_{2^p}{}^{kj}} = (-1)^j f(j/2^p), \quad j = 0, ..., 2^p - 1 \qquad (***)$$

si l'on pose $v_k = u_{k-2^{p-1}}$ pour k entre 0 et $2^p - 1$ et $W_{2^p} := \exp(-2i\pi/2^p)$. Ouvrez un fichier `.m` sur lequel vous rédigerez (en utilisant soit la procédure `fft`$(., 2^p)$, soit la procédure `CooleyTukey(.,p)` construite à l'exercice 4) une fonction

```
function U=InterpolTrigo1(f,p)
```

qui renvoie en sortie, sous forme d'une colonne, la liste des coefficients v_k, $k = 0, ..., 2^p - 1$, satisfaisant au système $(\dagger)$ (correspondant à la liste, dans le même ordre, des coefficients u_k solutions du système $(*)$). Pourquoi au fait ces systèmes $(**)$ ou $(***)$ sont ils des systèmes de Cramer ? Sauvez votre fichier `InterpolTrigo1.m` dans votre répertoire `TPMATLAB`.

(2) Sur un nouveau fichier `.m`, rédigez une procédure

```
function PolTrigo = InterpolTrigo2(f,p,XX)
```

qui calcule, étant donné une fonction `f`, un entier naturel non nul `p` et un vecteur ligne `XX` de nombres flottants de $[0, 1]$, la liste (en ligne) des valeurs prises aux entrées de `XX` par la fonction polynomiale trigonométrique $(*)$ (avec $M = 2^{p-1}$) dont les coefficients u_k (dans cet ordre) sont ceux fournis en colonne par la commande

```
>> U= InterpolTrigo1(f,p);
```

Testez votre routine sur un polynôme trigonométrique `f1` (mais non 1-périodique) déclaré en ligne

```
>> f5= inline('4*sin(pi/2*x) + cos(5*x) - 2*sin(3*x)','x');
```

suivant les instructions :

```
>> XX=0:1/2000:1;
>> PXXT=InterpolTrigo2(f5,p,XX);
>> plot(XX,f5(XX),'r')
>> hold
>> plot(XX,PXXT)
```

Prenez pour cela des valeurs de `p` entre `6` ($2^p = 64$) et `10` ($2^p = 1024$). Qu'observez vous lorsque `p` augmente concernant la qualité de l'approximation de `f` par son polynôme trigonométrique interpolant ? Recommencez avec cette fois la fonction :

```
>> f6= inline('4*sin(pi/2*x) + cos(5*x) - 2*sin(3*x)','x');
```

Après avoir évalué `f(1)-f(0)`, essayez de d'expliquer ce qui crée le phénomène au bords de $[0,1]$ dans le cas du premier exemple, et ne semble plus le créer (en tout cas de manière aussi nette) ici. Ce phénomène s'appelle, lorsqu'il se produit *phénomène de Gibbs*; c'est le pendant du *phénomène de Runge* observé dans le cadre de l'interpolation polynomiale par le polynôme de Lagrange (Exercice 1). Voir par exemple le site (succint) sur `wikipedia` :

`http://fr.wikipedia.org/wiki/Ph%C3%A9nom%C3%A8ne_de_Runge`

Sauvez votre fichier `InterpolTrigo2.m` dans votre répertoire `TPMATLAB`. Testez aussi votre programme avec une fonction polynomiale :

```
>> f7 = inline('x.^7+3*x.^5-2*x+1','x');
```

(c'est la fonction `f1` de l'exercice 2). Est-il préférable dans ce dernier cas d'utiliser l'interpolation par des polynômes trigonométriques plutôt que l'interpolation de Lagrange par des polynômes ?

(3) Pour mieux vous convaincre du phénomène de Gibbs, déclarez la fonction `f8` définie par

```
>> f8=inline('(1/2)*(sign ((x-1/3).*(2/3-x))+1)','x');
```

Affichez le graphe de `f8` :

```
>> XX=0:1/2000:1;
>> plot(XX,f8(XX));
```

Sur le même graphe, affichez (en utilisant la commande `hold`) avec diverses couleurs les graphes successifs des divers `PXXT` :

```
>> PXXT = InterpolTrigo2(f8,p,XX);
>> hold
>> plot(XX,PXXT,'color');
```

avec `p` variant de 6 à 10 et `color= b,g,m,r,k`. Quel constat faites vous ? Pourquoi est-il nécessaire pour observer le phénomène de Gibbs d'avoir pris un pas $1/2000 < 2^{-10}$ pour le vecteur ligne `XX` où sont évaluées la fonction et son polynôme d'interpolation trigonométrique ? Le phénomène de Gibbs apparait dans la pratique dès que l'on tente de couper les hautes fréquences d'une fonction oscillante, mais présentant tout de même des discontinuités (des « *cracks* »). On rehausse (ou affaiblit) artificiellement la fonction avant ou après le passage par une discontinuité. Le repiquage audio de vieux enregistrements (très bruités, donc contenant des composantes haute-fréquence) se heurte constamment à ce problème, que l'on tente de corriger en électronique, et que l'on appelle l'*aliasing*.