MT44 – TP1
Polynômes d'interpolation

Laurent Couvidou – rendu le 25 avril 2006

1. Évaluation d'une fonction polynôme en t

a) Évaluation d'un polynôme avec schéma itératif

On débute par un algorithme simple d'évluation de polynôme, en calculant directement depuis l'écriture de Newton, puis on l'adapte à Matlab (le premier indice des vecteurs est matrice est 1, et non 0).

Le code résultant est le suivant (cf fichier `evaluation1.m`) :

```
function val=evaluation1(n,a,c,t)
val=0;
for i=0:n % pour chaque degré
    pdt=1;
    for j=1:i % on fait le produit des (x-cj)
        pdt=pdt*(t-c(j));
    end
    val=val+a(i+1)*pdt;
end
end
```

Cet algorithme est O(nlog(n)) – deux boucles for donc une « s'étends » - il n'est donc pas particulièrement rapide.

Il peut être nécessaire d'ajouter des vérifications sur les paramètres avant le lancement de l'algorithme en lui-même.

C'est l'objet du code `evaluation1_verif.m`. Cependant, on mesurera par la suite les temps d'exécution de l'algorithme, on conservera donc cette version, sans « s'encombrer » de code supplémentaire, qui risque de fausser les résultats. On utilisera le même principe par la suite.

b) Évaluation d'un polynôme avec schéma de Hörner

Cette fois, on utilise un schéma de Hörner, tel que vu en cours. Il utilise une propriété importante de l'écriture de Newton des polynômes : la possibilité de changer la famille de centres. La valeur pour

laquelle on évalue le polynôme est introduite comme nouveau premier centre (on enlève le dernier), et la première composante de la nouvelle expression donne le résultat.

Code résultant (cf. `evaluation2.m`) :

```
function [val,nouv_a,nouv_c]=evaluation2(n,a,c,t)
nouv_a=a;
for i=n-1:-1:0 % calcul des nouvelles composantes
    nouv_a(i+1)=a(i+1)+(t-c(i+1))*nouv_a(i+2);
end
val=nouv_a(1);
nouv_c=[t,c(1:n-1)];
end
```

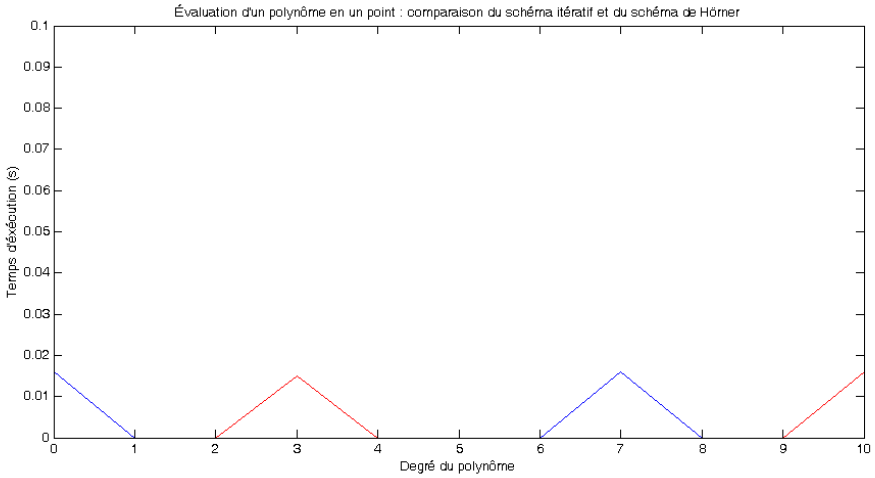
Cette fois-ci, l'algorithme est O(n), ce qui est nettement plus intéressant.

c) Comparaison des temps d'éxécution

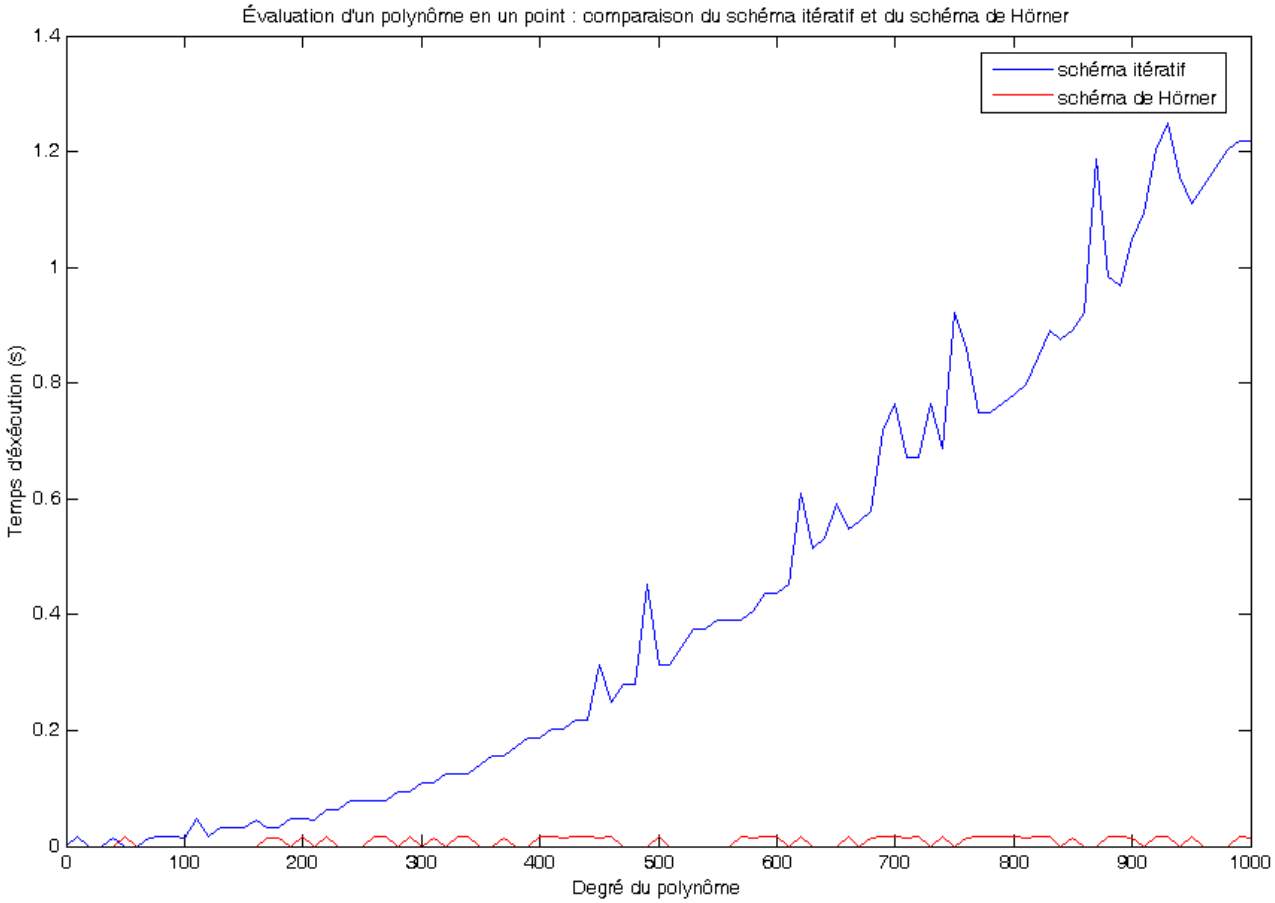
On écrit les fonctions Matlab suivants, qui permettent de comparer les deux algorithmes vus ci-dessus (commande `help` pour plus de détails) :

- `exo1c_compare_rand.m` : Compare les 2 fonctions pour un polynôme de degré n et pour un nombre d'itérations donné (permet d'obtenir des temps significatids), et renvoie les 2 temps obtenus. Les composantes, les centres et la valeur en laquelle évaluer sont determinés aléatoirement ;
- `exo1c_compare.m` : Même chose, mais en fournissant les données à l'algorithme.
- `exo1c_plot.m` : Donne une représentation graphique en faisant varier le degré du polynôme. On procède de la manière suivante : on détermine aléaloirement les composantes, centres et valeur d'évaluation. Puis on calcule les temps pour le polynôme de degré 0 (première composante), de degré 1 (deuxième composante, premier centre), etc.

Pour les premiers degrés, les résultats ne sont pas très significatifs (ici, jusqu'aù degré 10) :



Mais la différence d'ordre des algorithmes apparaît clairement pour des polynômes de degré très grand (ici, les degrés de 0 à 1000 et de 10 en 10) :



Courbe obtenue par : `exo1c_plot(0,1000,10,100).`

On voit nettement l'avantage à utiliser le schéma de Hörner : son temps d'éxécution reste stable, tandis que celui du schéma itératif « explose » avec le degré du polynôme.

d) Version matricielle de l'algorithme d'évaluation

On souhaite adapter l'algorithme d'évaluation aux matrices, pour correspondre au fonctionnement de Matlab et donc faciliter les calculs (cf. 3.)

On réutilise evaluation2 (qui, on l'a vu, est plus efficace), qu'on encapsule afin d'obtenir une version matricielle. La méthode employée est simple : on parcoure le vecteur de réels et on éxécute la fonction pour chacune.

Code résultant : `evaluation.m`.

2. Table des différences divisées et polynôme d'interpolation

a) Table des différences divisées

Comme pour le schéma de Hörner, on réutilise l'algorithme vu en cours, que l'on adapte aux contraintes de Matlab.

Le code obtenu est le suivant (cf. table_diff_div.m) :

```
function t=table_diff_div(x,y)
n=size(x,1)-1;
for i=1:n+1
  t(i,1)=y(i);
end
for j=2:n+1 % avance dans les colonnes
  for i=1:n-j+2 % avance dans les lignes
    t(i,j)=(t(i+1,j-1)-t(i,j-1))/(x(i+j-1)-x(i));
  end
end
end
```

Exemple d'éxécution :

```
>> x=[1;3;6;5];
>> y=[4;6;8;10];
>> t=table_diff_div(x,y)

t =

    4.0000    1.0000   -0.0667   -0.3167
    6.0000    0.6667   -1.3333         0
    8.0000   -2.0000         0         0
   10.0000         0         0         0
```

On obtient le résultat attendu.

b) *Construction du vecteur des composantes*

Pour construire le polynôme d'interpolation, on utilise l'extension Symbolic Math Toolbox de Matlab, qui permet de travailler avec des expressions littérales.

Il nous suffit alors de calculer le polynôme en utilisant la forme de Newton (somme des produit des composantes multipliées par les (x-cj)). On obtient le code suivant (cf interpol.m) :

```
function p=interpol(n,x,y)
syms X;
t=table_diff_div(x,y);
% On prend la première ligne de la table des dd
for i=1:n
    dd(n-i+1)=t(1,i);
end
p=0;
% Expression du polynôme grâce aux syms
for i=1:n
    a=1;
    for j=1:n-i
        a=a*(X-x(j));
    end
    p=p+dd(i)*a;
end
end
```

Pour l'exemple, en utilisant les x et y définis précédemment, on obtient :

```
>> p=interpol(4,x,y)
p =
-19/60*(X-1)*(X-3)*(X-6)-1/15*(X-1)*(X-3)+X+3
```

c) *Applications*

Pour la fonction exponentielle, sur un support linéaire, on obtient $p_{7,1}$:

```
>> p71=interpol(7,(-1:0.25:1)',exp(-1:0.25:1)')
p71 =
5066825350349587/4611686018427387904*(X+1)*(X+3/4)*(X+1/2)*
(X+1/4)*X*(X-1/4)+6689751672698739/1152921504606846976*(X+1
)*(X+3/4)*(X+1/2)*(X+1/4)*X+1840105915606357/72057594037927
936*(X+1)*(X+3/4)*(X+1/2)*(X+1/4)+6478666371007659/72057594
037927936*(X+1)*(X+3/4)*(X+1/2)+33413326358533/140737488355
328*(X+1)*(X+3/4)+941136233459491/2251799813685248*X+884763
545273989/1125899906842624
```

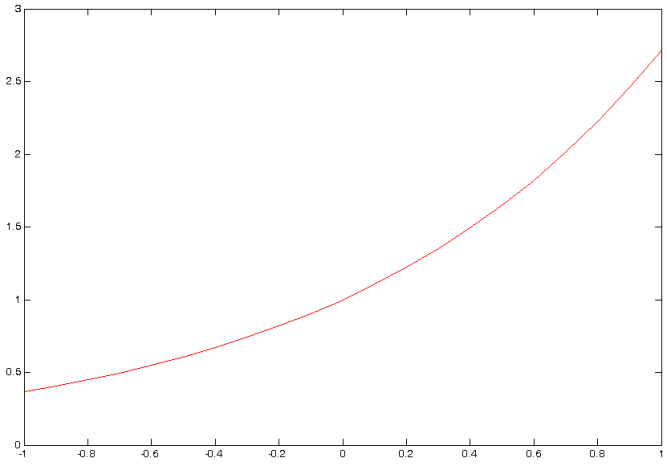
Et sur le support de Tchebyschev, on obtient $p_{7,2}$:

```
>> j=(0:7)';

>> x=cos(pi*((2*j)+1)/8);

>> p72=interpol(7,x,exp(x))

p72 =
6189791032238361/4503599627370496*(X-4160783518353059/45035
99627370496)*(X-1723452963400281/4503599627370496)*(X+34469
05926800561/9007199254740992)*(X+4160783518353059/450359962
7370496)*(X+8321567036706119/9007199254740992)*(X+689381185
3601133/18014398509481984)-6894686004298899/144115188075855
872*(X-4160783518353059/4503599627370496)*(X-17234529634002
81/4503599627370496)*(X+3446905926800561/9007199254740992)*
(X+4160783518353059/4503599627370496)*(X+8321567036706119/9
007199254740992)-6138369435212329/144115188075855872*(X-416
0783518353059/4503599627370496)*(X-1723452963400281/4503599
627370496)*(X+3446905926800561/9007199254740992)*(X+4160783
518353059/4503599627370496)+6311369523484171/36028797018963
968*(X-4160783518353059/4503599627370496)*(X-17234529634002
81/4503599627370496)*(X+3446905926800561/9007199254740992)+
6347751271711505/9007199254740992*(X-4160783518353059/45035
99627370496)*(X-1723452963400281/4503599627370496)+87611985
07643083/4503599627370496*X+1463883534254702464642417338284
7/2028240960365167042394725128601 6
```

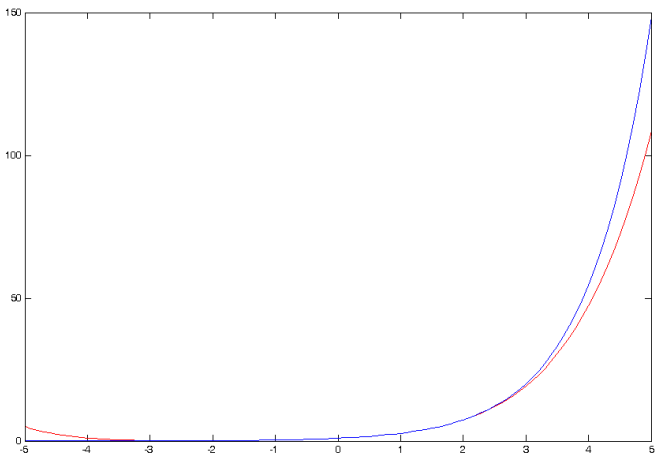
3. Visualisation de l'erreur d'interpolation

Pour $p_{7,1}$ l'erreur d'interpolation est faible sur [-1,1], et lex courbes semblent confondues. Par contre, elle s'aggrave si on étends l'intervalle (cf. `exo3_plot71.m`). En bleu, la courbe réelle, et en rouge la courbe de l'interpolation :

Sur [-1,1] :



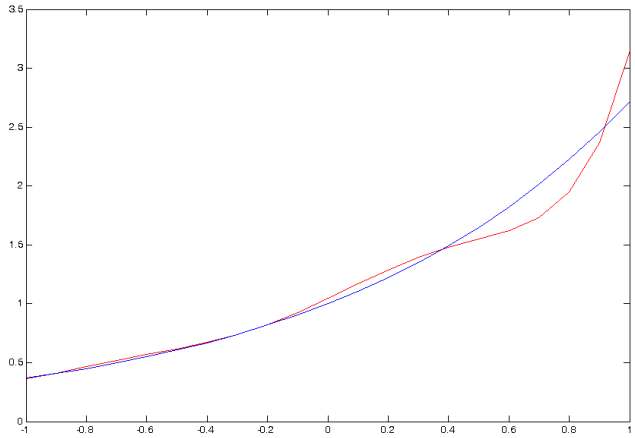
Sur [-5,5]



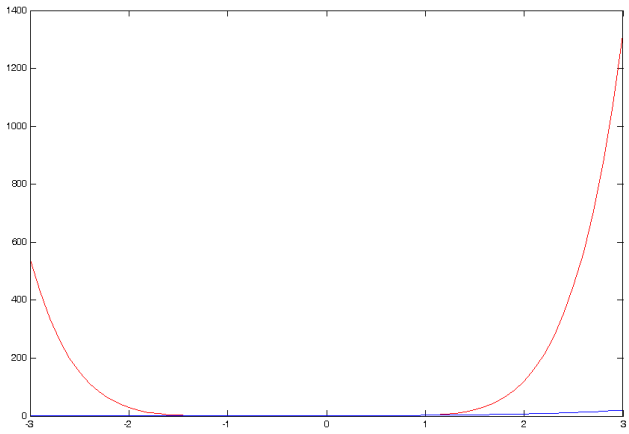
Pour $p_{7,2}$, les résulats sont plus « étranges ». Le polynôme interpole assez bien la fonction sur [-1,1], mais les résultats sont très mauvais sur un intervalle plus grand.

Voir le fichier de test : `exo3_plot71.m`.

Sur [-1,1]



Sur [-3,3] :



4. Conclusion

On a obtenu un bon aperçu des polynômes d'interpolation, des résultats qu'ils fournissent, ainsi que des problématiques informatiques liées à leur calcul (via les mesures de temps d'éxécution).