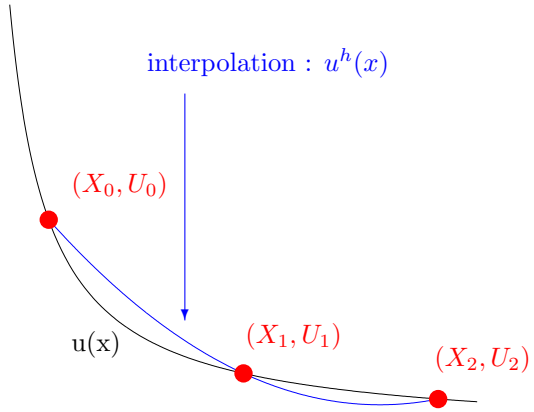
Matlab 14-15 for dummies : problème 1
Chef, il y a encore moyen de faire mieux !

Il est possible de rendre nettement plus rapide l'implémentation MatLab de l'interpolation de Lagrange en tirant profit de la pré-allocation des tableaux et de la vectorisation. Si nous comparons deux fonctions qui permettent d'obtenir les ordonnées des m abscisses `x` du polynôme de Lagrange de degré n passant par les $n+1$ points (X_i, U_i), le temps de calcul requis par la fonction `lagrange_naive(X,U,x)` est nettement plus élevé que celui de `lagrange(X,U,x)`.



La mise en oeuvre immédiate de la formule de l'interpolation de Lagrange requiert un calcul d'une complexité $\mathcal{O}(mn^2)$. Pour obtenir chaque ordonnée, il faut effectuer un nombre de n^2 opérations. Les trois boucles imbriquées de l'implémentation naïve permettent de déduire immédiatement cela. Evidemment, c'est bien ennuyeux cela :-(

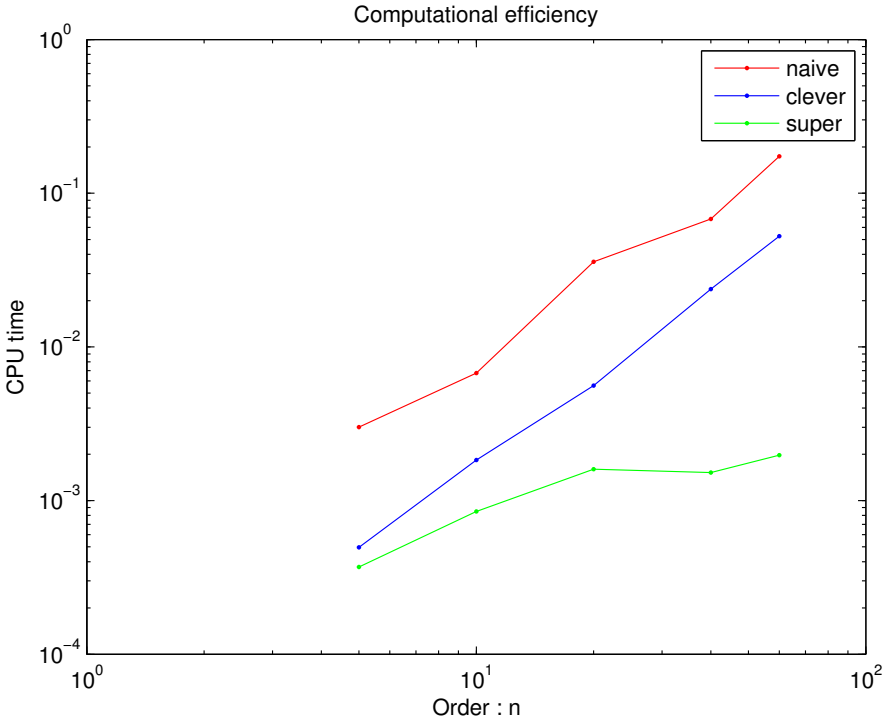
Est-il possible de modifier cette formule afin de pouvoir effectuer certaines opérations a priori pour l'ensemble des ordonnées et de pouvoir réduire le coût calcul ? En particulier, considérons un nombre $n = 40$ assez conséquent et un nombre $m = 1000$ élevé. L'idée de -Benoît dit le futé- est alors de ré-écrire l'équation de l'interpolation. Afin de simplifier notre propos, nous allons nous restreindre au cas $n = 3$, mais la généralisation de l'approche est immédiate et est laissée à votre sagacité :-)

$$u^h(x) = U_0 \frac{(x-X_1)}{(X_0-X_1)}\frac{(x-X_2)}{(X_0-X_2)} + U_1 \frac{(x-X_0)}{(X_1-X_0)}\frac{(x-X_2)}{(X_1-X_2)} + U_2 \frac{(x-X_0)}{(X_2-X_0)}\frac{(x-X_1)}{(X_2-X_1)}$$

En mettant en évidence un facteur commun pour toutes les abscisses...

$$\begin{aligned} u^h(x) = (x-X_0)(x-X_1)(x-X_2)\Bigg(& \frac{1}{(x-X_0)}\frac{U_0}{(X_0-X_1)(X_0-X_2)} \\ & + \frac{1}{(x-X_1)}\frac{U_1}{(X_1-X_0)(X_1-X_2)} \\ & + \frac{1}{(x-X_2)}\frac{U_2}{(X_2-X_0)(X_2-X_1)}\Bigg) \end{aligned}$$

On peut ainsi obtenir une complexité nettement réduite en effectuant subtilement les calculs. Tout d'abord, on va calculer tous les termes en bleu et les stocker dans un vecteur de taille $n+1$. Ensuite, on peut effectuer le calcul du facteur commun en rouge qu'on stockera dans un vecteur de taille m. Et finalement on obtiendra la valeur des m ordonnées. Les deux dernières étapes peuvent aussi être permutées sans souci ! Cette approche permet de réduire la complexité du calcul ! Lorsqu'on compare les temps d'exécution pour une implémentation naïve, pour l'implémentation présentée au cours et cette nouvelle super implémentation, on va observer qu'elle est nettement plus rapide, comme on l'observe sur la figure. C'était donc bien une bonne idée :-)



La complexité d'un algorithme : c'est quoi ?

Pour obtenir la complexité d'un algorithme qui dépend d'un paramètre n, on compte le nombre d'opérations arithmétiques élémentaires qu'il faut effectuer pour obtenir le résultat. En d'autres mots, on compte le nombre d'additions, de soustractions, de multiplications et de de divisions requises pour obtenir le résultat et puis on estime l'ordre de grandeur en prenant le terme le plus important.
Par exemple, si le nombre d'opération est exactement

$$3n^3 + 2n^2 + 4n + 8$$

on dira que l'algorithme a une complexité de $\mathcal{O}(n^3)$. C'est le même principe que ce qu'on fait pour estimer l'ordre de de précision d'une méthode numérique. La notation est exactement la même.
A titre d'exemple, la multiplication d'une matrice carrée $n \times n$ par un vecteur de taille n est un algorithme d'une complexité $\mathcal{O}(n^2)$.

On vous demande de :

1. Ecrire une fonction `[uh] = lagrange_super(x,X,U)` qui effectue le calcul des abscisses en utilisant l'approche ci-dessus. Normalement, votre fonction implémentée avec soin doit être nettement plus rapide que celles fournies avec le devoir : `lagrange` et `lagrange_naïve`.
2. Afin de tester la précision et l'efficacité de votre fonction, un petit programme test `test_matlab1.m` vous est fourni qui permet d'obtenir les temps d'exécution de votre fonction pour une série de valeurs de n avec $m = 1000$.
3. Dans les commentaires de votre fonction, donner la complexité de l'approche proposée par Benoît. Expliquer aussi quel est le problème majeur de cette approche ! Ce gros problème a requis de définir

les abscisses x avec une ligne de code un peu mystérieuse : `x = linspace(eps,5*(1-eps),m);` Pourquoi faut-il faire cela et mais ainsi, est-on sûr de n'avoir jamais aucun problème ? En d'autres mots, notre super implémentation est efficace, mais toujours précise et pas très robuste : eh oui !

4. Et que se passe-t-il lorsque $n > 60$?

5. Votre programme doit être exact, mais il doit aussi être le plus rapide possible.

6. Certains pourraient être tenté d'obtenir directement la meilleure implémentation avec `polyfit` et `polyval` ! Malheureusement, un enseignant facétieux a retiré ces deux fonctions lorsque l'on testera votre devoir : il est donc strictement interdit de les utiliser ! D'ailleurs, au passage, vous pourriez observer que l'implémentation de Mathworks est souvent encore plus efficace que celle de Benoît.

7. Votre fonction (avec les éventuelles sous-fonctions que vous auriez créées) sera incluse dans un unique fichier `lagrange_super.m`, sans y adjoindre le programme de test fourni ! Cette fonction devra être soumise via le web avant le mardi 14 octobre à 23h59 : ce travail est individuel et sera évalué. Pour faciliter la correction automatique, ne pas inclure les commandes `clc` et `close all` dans votre fonction!