- Projet de MT10 -
TP2

22 avril 2010

Table des matières

Table des figures

L'objectif de ce TP est de comparer la complexité empirique de différents algorithmes d'exponentiation. Dans chaque cas, nous prenons comme référence la commande *MuPAD* usuelle x^n.

1 Organisation

1.1 Algorithme et fonctions *MuPAD*

1.1.1 Principe de réalisation

Afin de déterminer la **complexité** d'un algorithme, nous avons exploité la *complexité empirique assimilée au temps d'exécution.*

Pour **comparer deux algorithmes**, nous avons fixé la valeur de la variable x et fait varier celle de l'exposant n sur un intervalle nous permettant d'identifier le phénomène souhaité.

Afin d'obtenir des **mesures fiables**, nous avons réalisé plusieurs fois chaque opération et avons utilisé la moyenne de celles-ci.

Quand cela fût nécessaire, nous avons également exploité une régression linéaire afin de comparer plus facilement les complexités.

Pour cela nous avons eu besoins de quelques **fonctions de manipulations** :

- La fonction d'implémentation de l'algorithme d'exponentiation ;
- Une fonctionne qui nous retourne la moyenne des opérations ;
- Et une fonction qui nous calcul la liste des mesures pour un intervalle de n donné ;

Dans la majorité des cas, les échelles logarithmiques nous ont permis d'obtenir une *représentation linéaire de la complexité* en fonction de l'exposant n.

Une régression linéaire a ensuite été réalisée afin d'identifier plus facilement le coefficient de complexité.

1.1.2 Organisation du TP et du rapport

1. Le fichier source « Algos.mu », disponible à la racine du TP, contient toutes les procédures qui implémentent les différents algorithmes, ainsi que leurs variantes itératives et récursives.
2. Pour chaque type d'algorithme, nous avons réalisé un sous répertoire contenant :
   - un fichier « X-Tools.mu » qui représente les fonctions de manipulation (moyenne, liste de mesures) ;
   - un fichier « notebook.nm » qui contient les commandes réalisées pour générer les mesures, les régressions linéaires associées, les calculs de coefficients et les graphiques.

1.2 Types de complexité

Nous avons pu voir en TD que la complexité d'un algorithme pouvait suivre deux modèles différents :

- Le modèle à coûts fixes ;
- Le modèle à coûts variables ;

Avec *MuPAD*, le **modèle à coûts variable** semble le plus adapté lors de la manipulation de grand nombres[1].

2 Méthode Naïve

2.1 Méthode de comparaison de complexité

Nous avons également pu voir en TD que la complexité, en coûts variables, de ce type de méthode, que ce soit de manière itérative ou récursive, présentait une complexité de $O(n^2 * ln(x)^2)$.

En Utilisant des échelles logarithmiques sur les axes des abscisses et des ordonnées, nous obtenons :

$ln(C) = ln(n^2 * ln(x)^2) = ln(n^2) + ln(ln(x)^2) = 2ln(n) + ln(ln(x)^2)$

x étant fixe au cours de nos mesures => $ln(ln(x)^2)$ correspond à un coefficient b tel que $ln(C) = a * n + b$. Ceci représente l'équation d'une fonction linéaire.

⇒Nous pouvons donc comparer la complexité des différents algorithmes en comparant les coefficients directeurs des régressions linéaires associées.

2.2 Commandes

Dans cette partie, nous allons énoncer le déroulement du protocole de mesure et les commandes saisies afin d'obtenir les différents résultats. Etant donné que nous suivons le même protocole dans le reste du TP, nous n'allons pas redonner les commandes puisque seul le nom des fonctions change.

En cas de changement, nous préciserons tout de même les différences réalisées.

2.2.1 Algorithmes - « Algos.mu »

Algorithmes d'exponentiation naïve

```
// 1 - NAIF
// Iteratif
iNaif:=proc(x, n)
  local result;
begin
  result:=1;
  for i from 0 to n-1 do
    result:=result*x;
  end_for;
  return (result);
end_proc;

// Naif Recursif
rNaif:=proc(x, n)
begin
  if n=0 then return (1);
  else
```

[1] plus d'une centaine de chiffres décimaux

```
18      if n=1 then return(x);
19      else return (x*rNaif(x, n-1));
20      end_if;
21    end_if;
22  end_proc;
```

Commentaires :

2.2.2 Fonctions de manipulations - « 1-Tools.mu »

Fontions de manipulations - Méthode naïve

```
1  // ITERATIF
2
3  // 1 - Moyennes Naif Iteratif pour X et N
4  moyenneIN:=proc(x, n)
5    local result;
6  begin
7    result:=0;
8    // CAUTION : loops var is "v" because "i" from "iNaif" is taken like the var "i"
9    for v from 1 to 5 do
10     result:= result + float(time(iNaif(x,n)));
11     //print(result);
12   end_for;
13
14   result:=float(result/5);
15   return (result);
16 end_proc;
17
18 // 2 - LISTE des temps d'executions pour les elements de la liste AB
19 ordonneesIN:=proc(x, ab:DOM_LIST)
20   local ordonnees;
21 begin
22   ordonnees:=ab; // empty list
23   // CAUTION : loops var is "v" because "i" from "iNaif" is taken like the var "i"
24   for w from 1 to nops(ab) do
25     ordonnees[w]:=moyenneIN(x, ab[w]);
26   end_for;
27   return (ordonnees);
28 end_proc;
```

Commentaires : Comme explicité précédemment, nous avons dans ce type de fichier source deux méthodes pour chaque algorithme :

- La méthode *moyenneXX* qui calcul la moyenne de 5 appels à la fonction d'algorithme définie dans « Algos.mu » ;
- La méthode *ordonneesXX* qui renvoie la liste des mesures pour l'intervalle de n définie dans la liste *abscisses* ;

2.2.3 Commandes - « 1-Naif.mn »

Commandes et mesures - Algorithme d'exponentiation naïve

```
Algorithmes d'exponentiation - Methode Naive

// 0 - Preparation des donnees
Pour la suite de l'exercice nous allons fixer
x:=7:

// Valeurs de n pour Iteratif et x^n
abscisses:=[i*10^4 $ i = 1 .. 10]:
abscisses_ln:=map(abscisses, ln@float):
nbElement:=nops(abscisses):


// 1 - Methode naive iterative
// 1.1 - Mesures : liste des moyennes des temps d'executions de l'algorithme pour n variant
Iordonnees:=ordonneesIN(x, abscisses):

// 1.2 - On choisit ici une echelle logarothmique
Iordonnees_ln:=map(Iordonnees, ln@float):

// 1.3 - On construiy la liste des points du graphique
p:=plot::Point2d(abscisses_ln[j], Iordonnees_ln[j]) $ j=1..nops(abscisses):

// 1.4 - Regression lineaire
[coefficient, cov]:=stats::linReg(abscisses_ln , Iordonnees_ln);
```

$[[-12.69399989, 1.849639819], 0.009182066746]$

```
// on obtient [ [b, a], cov] tel que y=a*x+b = equation de la droite
// On peut remarquer un coefficient  a = 1,84

// 1.5 - Droite : Line2d( point1, point2);
// point : x, y   avec y = a*x+b
droite:=plot::Line2d( [abscisses_ln[1], coefficient[2]*abscisses_ln[1]+coefficient[1]],
[abscisses_ln[nbElement], coefficient[2]*abscisses_ln[nbElement]+coefficient[1]]):

// 1.6 -Graph
plot(p, droite);
```

Commentaires : Nous réalisons la même opération pour chaque type d'algorithme.

2.3 Mesures et graphes

Pour cette partie, nous avons fixé $x = 7$, un nombre premier moins spécifique que 2, 3, 5 et 10^2 et n appartient à l'intervalle $[1 * 10^4..1 * 10^5]$.

[2] Des multiples à ces nombres premiers sont facilement déterminables

2.3.1 Méthode itérative

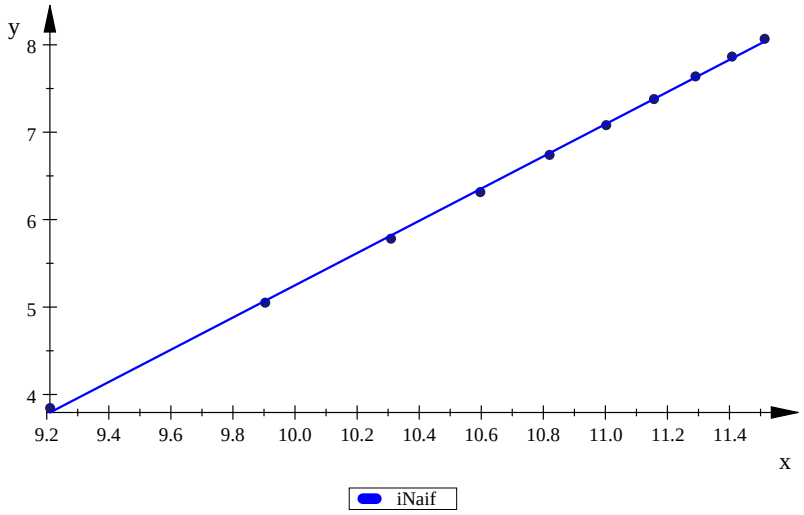

FIG. 1 – $time(iNaif(x,n)$ en fonction de n

Commentaires : Nous observons donc ici un coefficient de **1,84**, ce qui nous donne une complexité de $O(n^{1,84})$. Cette valeur est assez proche de ce que l'on devrait obtenir : $O(n^2)$.

2.3.2 Comparaison méthodes récursive - itérative

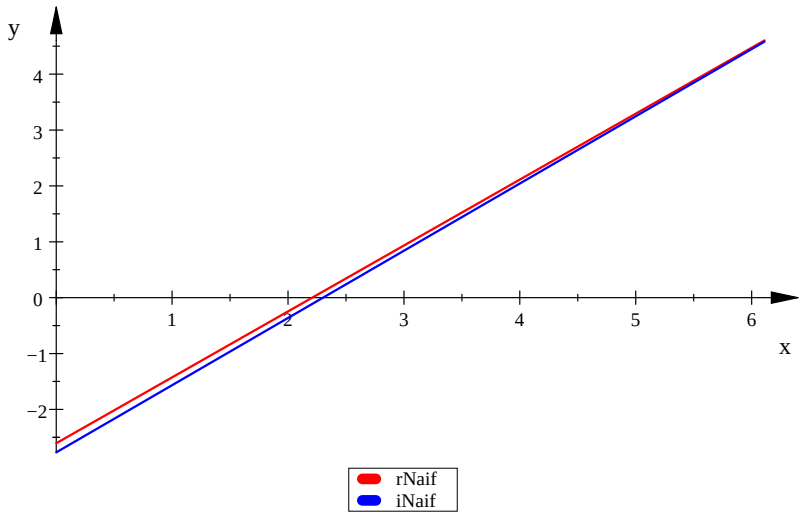

FIG. 2 – $time(iNaif(x,n)$ et $time(rNaif(x,n)$ en fonction de n

Commentaires : Pour l'algorithme récursif, nous sommes limités par *MuPAD* à `MAXDEPTH = 500` appels récursifs. Avec cette méthode naïve, le nombre d'appel est égal à la valeur de l'exposant n. De ce fait nous ne pouvons aller plus loin que $n = 500$ dans nos mesures.

Pour ce faire, nous avons fixé $x = 10^{100}$ afin d'obtenir des temps de calculs conséquents pour les seuls 500 exponentiations authorisées. Nous avons donc également redéfinie un nouvel intervalle pour n appartenant à [1..500] avec un pas de 50 pour avoir une dizaine de points.

```
// 2.3 - Regressions lineaire iterative
[coefficient2b, cov2b]:=stats::linReg(abscisses_ln2 , Iordonnees2_ln);
```

$[[-2.768268709, 1.202598087]\,, 1.799902351]$

```
[coefficient2, cov2]:=stats::linReg(abscisses_ln2 , Rordonnees_ln);
```

$[[-2.606478399, 1.179649371]\,, 1.222201223]$

Nous pouvons ainsi remarquer que les deux méthodes sont très proches pour cet intervalle de valeurs de n puisque nous avons des **coefficients de complexité** respectifs de :
- 1,20 pour la méthode itérative ;
- et 1,18 pour la méthode récursive ;

2.3.3 Comparaison avec x^n

Pour cette comparaison nous sommes revenus à n appartenant à l'intervalle $[1 * 10^4..1 * 10^5]$.

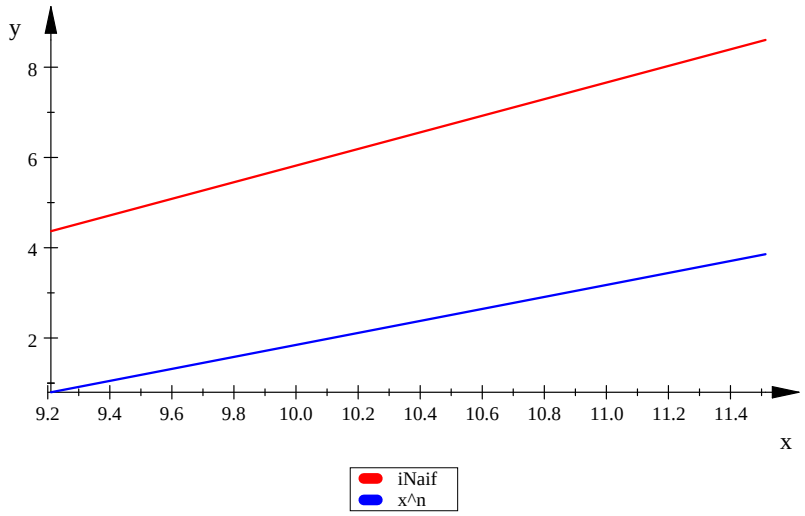

FIG. 3 – Comparaison de iNaif et de la commande usuelle x^n en fonction de n

Commentaires : La commande usuelle x^n présente un coefficient de 1,37. On remarque d'autant plus sur le graphique que l'algorithme utilisé est bien plus performant que l'algorithme naïf.

3 Méthode Dichotomique

3.1 Méthode de comparaison de complexité

Comme précédemment, nous avons vu en TD que la complexité en coûts variables de cette méthode, que ce soit de manière itérative ou récursive, était également d'une une complexité de $O(n^2 * ln(x)^2)$.

⇒ De ce fait, nous pourrons utiliser la même méthode que précédemment pour comparer les algorithmes.

3.2 Commandes

3.2.1 Algorithmes - « Algos.mu »

Algorithmes d'exponentiation dichotomique

```
// 2 - DICHOTOMIQUE
// Iteratif
iDicho:=proc(x, n)
  local result;
begin
  result:=1;
  while n>0 do
    if (n mod 2)<>0 then
      result:=result*x;
    end_if;

    x:=x*x;
    n:= n div 2;
  end_while;
  return(result);
end_proc;

// Recursif
rDicho:=proc(x, n)
begin
  if n=0 then return (1);
  end_if;

  if n=1 then return(x);
  else
    if (n mod 2)<>0 then return (x*rDicho(x*x, (n-1) div 2));
    else return (rDicho(x*x, n div 2));
    end_if;
  end_if;
end_proc;
```

Commentaires :

3.3 Mesures et graphes

Pour cette partie, nous avons remarqué que la méthode dichotomique était plus « optimisée » que la méthode naïve, nous avons ainsi pu utiliser un intervalle de valeur de n plus grand :

$n[1 * 10^5..10 * 10^5]$.

3.3.1 Comparaison itérative - récursive

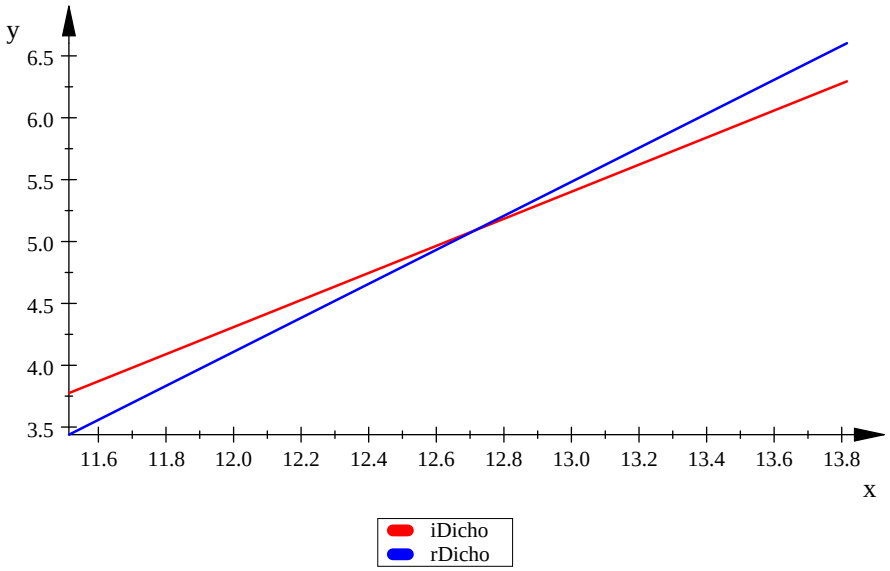

FIG. 4 – $time(iDicho(x, n))$ et $time(rDicho(x, n))$ en fonction de n

Commentaires : La méthode itérative nous donne un coefficient de **1,09**.
La méthode récursive quant à elle affiche un coefficient de 1,37.

Ainsi, nous pouvons remarquer qu'à partir d'une certaine valeur de n, la méthode itérative devient plus efficace que la méthode récursive.

Nous pouvons estimer ce n par : $ln(n) = 12,7 => n = 3,27 * 10^5$.

Nous n'avons pas eu l'occasion de tester pour des valeurs de n supérieures mais nous pouvons tout de même remarquer que la complexité est bien limitée par $O(n^2)$ puisque nous obtenons respectivement $O(n^{1,09})$ et $O(n^{1,37})$.

3.3.2 Comparaison avec la méthode usuelle x^n

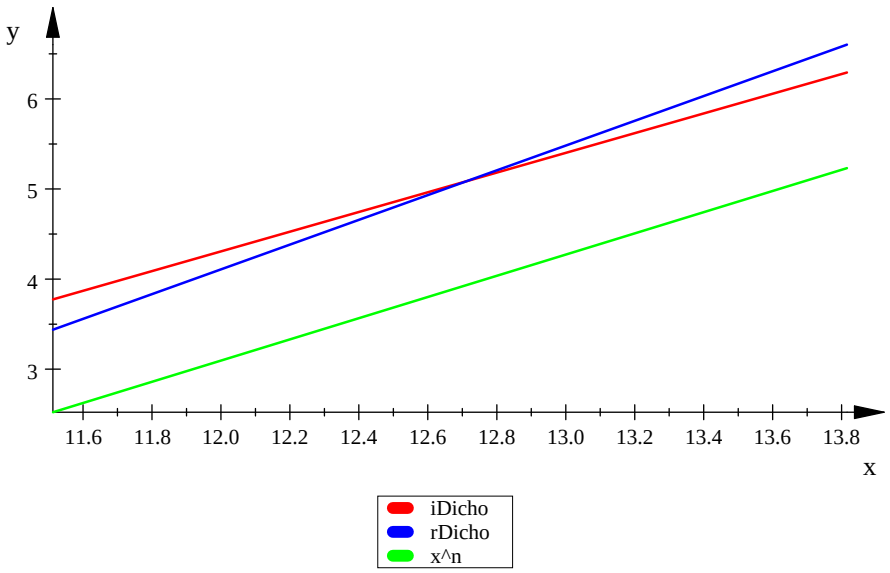

FIG. 5 – Comparaison des méthodes dichotomiques avec la commande usuelle x^n en fonction de n

Commentaires : Le coefficient de complexité de x^n est cette fois-ci de de **1,17**.

Même si l'écart entre les deux méthodes se réduit, *la commande usuelle est encore plus performante que la méthode dichotomique.*

A la vue des différentes valeurs des coefficients, il se pourrait que la méthode dichotomique itérative devienne plus efficace que x^n.

Nous pouvons conjecturer cette valeur tel que :

$(a * z + b)dich = (a' * z + b')usuelle$

$\Rightarrow$ La méthode dichotomique itérative deviendrait ainsi plus efficaces que la méthode usuelle pour des valeurs de $n > 8,8 * 10^{11}$

Nous ne pouvons malheureusement pas vérifier cette hypothèse puisque la valeur de $x^{9*10^{11}}$ est bien trop grande pour *MuPAD*.

3.3.3 Comparaison générale

Commentaires : **Le principe de la méthode dichotomique** est de diviser par 2 la valeur de n à chaque appel récursif ou à chaque itération. De ce fait, nous obtenons un nombre d'itération proche de $ln(n)$.

Mais puisque les opérations sur x sont plus « compliquées » à chaque appel, on obtient la même complexité théorique que la méthode naïve.

Avec un nombre d'appel récursif limité à 500, cette méthode nous permet de déterminer la limite de la version récursive à :

$ln(n) < 500 => n < e^{500} => n < 1,40 * 10^{217}$

Ce qui nous laisse encore pas mal de marge.

4 Méthode dichotomique modulaire

4.1 Méthode de comparaison de complexité

L'algorithme d'exponentiation modulaire est très utilisé car *il permet de manipuler de très grands nombres*. En effet, sa particularité réside sur le fait que l'opération « modulo » est appliquée à chaque itération de l'algorithme sur la valeur x à modifier. De ce fait, elle ne consomme que très peu de mémoire.

En fait, avec cette méthode, **on ne connait pas exactement la valeur de x^n mais son reste par la division par N**. Cela permet déjà de déterminer quelques propriétés intéressantes.

En effet, en choisissant un $N = 100$, on sait forcément que les deux derniers chiffres de x^n seront le résultat affiché par cette fonction. Pour l'exercice, nous avons fixé $x = 2, N = 10$ et n sur l'intervalle $[1 * 10^{(1*10^4)}..10 * 10^{(10*10^5)}]$.

Puisque les valeurs de n sont très grandes, nous avons utilisé une double échelle logarithmique pour les axes des abscisses.

D'après ce que nous avons vue en TD, la complexité de cet algorithme est $O(ln(n) * ln(N^2))$.

4.2 Commandes

4.2.1 Algorithmes - « Algos.mu »

Algorithmes d'exponentiation modulaire itératif

```
// 3 - MODULAIRE
// Iteratif
iModDicho:=proc(x, n, N)
  local result;
begin
  result:=1;
  while n>0 do
    if (n mod 2)<>0 then
      result:=result*x;
      // Application du modulo
      result:= result mod N;
    end_if;

    x:=x*x;
```

```
15      // Application du modulo
16      x:=x mod N;
17
18      n:= n div 2;
19    end_while;
20    return(result);
21  end_proc;
```

Commentaires :

4.3 Mesures et graphes

4.3.1 Fonction itérative

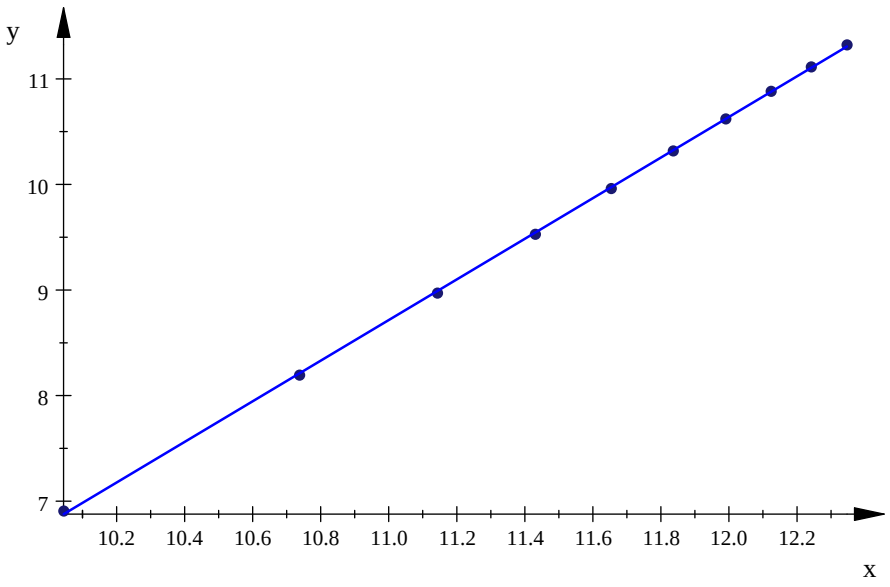

FIG. 6 – $time(iMod(x, n, N))$ en fonction de n

Commentaires : On obtient un coefficient de 1,92 pour la régression linéaire associée.
Ce qui nous donne une complexité $O(1,92.ln(ln(n))) \sim O(1.ln(ln(n)))$

4.3.2 Fonction `powermod`

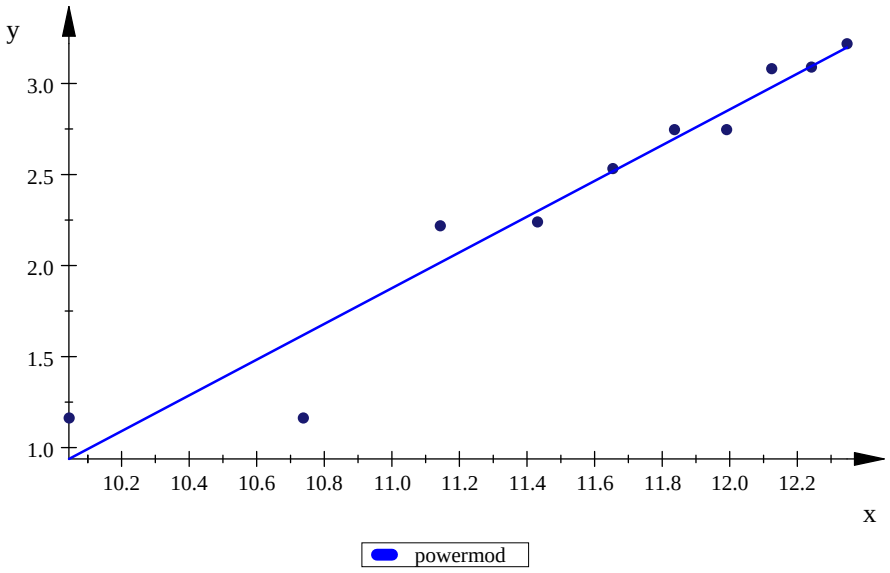

FIG. 7 – $time(powermod(x, n, N))$ en fonction de n

Commentaires : powermod(b, e, m) computes $b^e mod m$
`powermod` est donc la fonction usuelle de *MuPad* qui implémente l'exponentiation modulaire.

La fonction `powermod` est très efficace. En effet, nous avions d'abord pris n sur l'intervalle $[1 * 10^{(1*10^3)}..10 * 10^{(10*10^4)}]$. Cependant, les mesures de `powermod` sur cet intervalle ne nous donnait que des valeurs constantes. Nous avons ainsi du augmenter les valeurs de cet intervalle pour aboutir à n sur $[1 * 10^{(1*10^4)}..10 * 10^{(10*10^5)}]$.

Nous obtenons ainsi un coefficient directeur de régression linéaire égale à **0.98** qui suit de très prêt le modèle théorique de $O(1.ln(ln(n)))$.

4.3.3 Comparaison générale

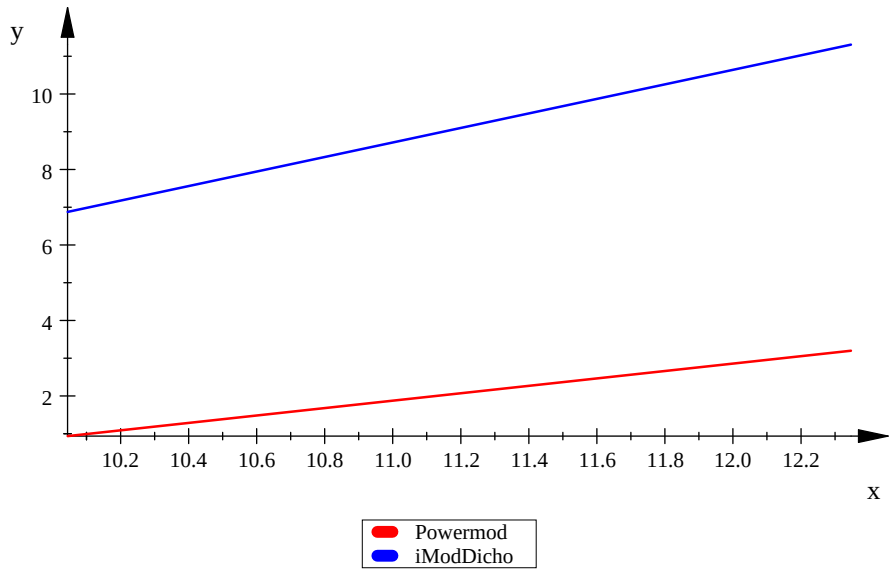

FIG. 8 – $time(powermod(x, n, N))$ et $time(iMod(x, n, N))$ en fonction de n

Commentaires : Le graphe montre sans équivoque que la fonction `powermod` est bien plus optimisée que notre implémentation pour respecter le principe et la complexité théorique de l'exponentiation modulaire.

Nous pouvons également préciser que pour cette partie, *nous avons abandonné le principe de réalisation de plusieurs mesures et de moyenne associée* pour calculer `iModDicho` pour l'intervalle de comparaison avec `powermod` tellement le temps d'execution était important[3].

Nous avons ainsi effectué :

Commande de mesures de comparaison iModDicho / Powermod

```
1   // 0 - Preparation des donnees
2  Pour la suite de l'exercice nous allons fixer
3  x:=2
4  N:=10:
5
6  // Values of n
7  abscisses2:=[i*10^(i*10^4) $ i = 1 .. 10]:
8  abscisses2_ln:=map(abscisses, ln@float):
9  abscisses2_ln_ln:=map(abscisses_ln, ln@float):
10 nbElement:=nops(abscisses):
11
12   Iordonnees2:=[time(iModDicho(2,abscisses2[i] ,10)) $ i=1..10];
```

[3] Plus de 1000 secondes pour ne réaliser qu'une passe de mesures sur l'intervalle spécifier

```
13
14  [coefficient5, cov5]:=stats::linReg(abscisses2_ln_ln , Iordonnees2_ln);
```

$[[-12.4375882, 1.923019808]\,, 0.002318142408]$

5 Calcul des nombres de Fibonacci par exponentiation dans les matrices 2*2

5.1 Méthode de calcul

Nous avons également pu voir en TD que

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}$$

De ce fait, nous cherchons à calculer la matrice A^n afin de déterminer F_n par identification de coefficients.

Pour calculer A^n, nous avons utilisé la propriété suivante :

$A = P * D * P^{-1}$;

Où D est la matrice diagonale tel que

$$(A) = P * \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix} * P^{-1}$$

Et P la matrice de passage tel que :

$P = (u_1 u_2)$;

Avec λ_1 et λ_1 les valeurs propres de A et u_1 et u_2 les vecteurs propres associés.

Ainsi, $A^n = \left(P * D * P^{-1}\right) * \left(P * D * P^{-1}\right) ... \left(P * D * P^{-1}\right)$

et nous pouvons remarquer que les $P^{-1} * P$ s'éliminent pour donner : $A^n = P * D^n * P^{-1}$.

D étant une matrice diagonale il est très facile de calculer D^n :

$D^k = diag(d_{ij})^k = diag(d_{ij}^k)$

Et donc par la même manière de calculer A^n.

Commentaires : Puisque la matrice A est constante, nous aurions pu fixer les valeurs propres dans des variables afin d'éviter leur calcul à chaque exécution de la fonction et ainsi essayer de réduire le temps de celle-ci.

5.2 Commandes

Pour cette question, nous avons réalisé des mesures pour trois fonctions :

1. Le calcul de A^n par sa matrice diagonale D^n ;
2. La manuplation par facilité de *MuPAD* concernant les matrices : A^n ;
3. Le calcul direct des nombres de *Fibonacci* par `nimlib : :fibonacci` ;

De la même manière, nous utilisons des fonctions de manipulation pour calculer une moyenne des temps d'execution puis la liste des mesures pour les représenter graphiquement[4].

5.2.1 Algorithmes - « Algos.mu »

Algorithmes de calcul des nombres de Fibonacci par exponentiation dans les matrices 2*2

```
// 4 - MATRICE
Fibo:=proc(n)
  //Matrice:=Dom::Matrix():
  local A, C, H, G, P, Da, An;
  local Id, Nul;
  local ev, V1, V2;
  begin
    A:=matrix([[1,1],[1,0]]):

    // Valeurs propres de A
    ev:=linalg::eigenvalues(A):

    // Matrices outils
    Id := matrix([[1,0],[0,1]]): // Identite
    C:= (ev[1])*Id:   //
    G:=(ev[2])*Id:
    Nul:= matrix([[0],[0]]):   // Matrice NULL
    H:=A-C:

    // Vecteurs Propres
    V1:=linalg::matlinsolve(H,Nul):
    V2:=linalg::matlinsolve(A-G,Nul):

    // matrix de Passage
    P:=matrix([ [V1[2][1][1], V2[2][1][1] ],[1,1]]):

    //Matrice diagonale
    Da:=matrix([ [V1[2][1][1], 0], [0,V2[2][1][1]] ]):

    // Calcul An
    An:=P*Da^n*P^(-1):

    return(float(An[1,2]));
end_proc;

```

[4] sources disponibles dans « 4-Tools.mu »

```
36
37
38 // 4 - MATRICE
39 Fibo2:=proc(n)
40   local A, P, Da, An;
41   local ev, Ev, Eigenvectors;
42 begin
43   Matrice:=Dom::Matrix():
44
45   A:=Matrice([[1,1],[1,0]]):
46
47   // Valeurs propres de A
48   ev:=linalg::eigenvalues(A):
49
50   // Matrice diagonale associee
51   Da:=Matrice([ [ev[1], 0],[0, ev[2]] ]);
52
53   // calcul des Vecteurs propres associes
54   Ev:=linalg::eigenvectors(A);
55   Eigenvectors:= Ev[1][3][1], Ev[2][3][1];
56
57   // Matrice de passage associee
58   P:= Eigenvectors[1].Eigenvectors[2];
59
60   // Exponentiation de A
61   An:=P*Da^n*P^(-1);
62
63   return(float(An[1,2]));
64   //return (1);
65 end_proc;
```

Commentaires : Nous avons ainsi réalisé 2 fonctions de calcul des nombres Fibonacci puisque selon les postes où nous travaillions, nous rencontrions des problèmes avec la méthode `eigenvectors` de *MuPAD*.

5.2.2 Commandes - Pas à pas du calcul de $A^n par D^n$

```
Algorithmes d'exponentiation - Calcul des nombres de Fibonacci

// 4 - Matrice
export(linalg);
Matrice:=Dom::Matrix():
Warning: the routine 'export' for exporting library functions to the global namespace
will become obsolete with future MuPAD versions.
Call the routine 'use' instead. [export::new]
Warning: 'htranspose' already has a value, not exported. [use]
Warning: 'transpose' already has a value, not exported. [use]
Warning: 'det' already has a value, not exported. [use]


// 4.1- On construit la matrice A
A:=Matrice([[1,1],[1,0]]);
```

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

```
// 4.2 - On recherche les valeurs propres de A
ev:= eigenvalues(A);
// On obtient 2 valeurs propores distincts donc A est diagonalisable
```

$$\left\{\frac{1}{2}-\frac{\sqrt{5}}{2}, \frac{\sqrt{5}}{2}+\frac{1}{2}\right\}$$

```
// On recherche désormais la matrice de passage et la matrice diagonale tel que A=P*D*P-1
// 4.3 - La matrice diagonale est donc:
Diag:=Matrice([ [ ev[1] , 0], [0,ev[2] ] ]);
```

$$\begin{pmatrix} \frac{1}{2}-\frac{\sqrt{5}}{2} & 0 \\ 0 & \frac{\sqrt{5}}{2}+\frac{1}{2} \end{pmatrix}$$

```
// 4.4 - Methode avec eigenvectors
// 4.4.1 - On cherche désormais les vecteurs propres associés pour réaliser
la matrice de passage.
Ev:=eigenvectors(A);
```

$$\left[\left[\frac{1}{2}-\frac{\sqrt{5}}{2}, 1, \left[\begin{pmatrix} \frac{1}{2}-\frac{\sqrt{5}}{2} \\ 1 \end{pmatrix}\right]\right], \left[\frac{\sqrt{5}}{2}+\frac{1}{2}, 1, \left[\begin{pmatrix} \frac{\sqrt{5}}{2}+\frac{1}{2} \\ 1 \end{pmatrix}\right]\right]\right]$$

```
Eigenvectors:= Ev[1][3][1], Ev[2][3][1];
```

$$\left(\begin{array}{c} \frac{1}{2} - \frac{\sqrt{5}}{2} \\ 1 \end{array} \right), \left(\begin{array}{c} \frac{\sqrt{5}}{2} + \frac{1}{2} \\ 1 \end{array} \right)$$

```
// 4.4.2 - on peut ainsi déterminier la matrice de passage
P:= Eigenvectors[1].Eigenvectors[2];
```

$$\left(\begin{array}{cc} \frac{1}{2} - \frac{\sqrt{5}}{2} & \frac{\sqrt{5}}{2} + \frac{1}{2} \\ 1 & 1 \end{array} \right)$$

```
// 4.4.3 - An
An2:=P*Diag^n*P^(-1);


// 4.4.4 - on peut tester simplement les coefficients pour Fn
float(An2[1,2]);
```

$$0.4472135955\,1.618033989^n - 0.4472135955\,{-0.6180339887}^n$$

```
// 4.4.4 - on peut tester simplement les coefficients pour Fn
float(An2[2,1]);
```

$$0.4472135955\,1.618033989^n - 0.4472135955\,{-0.6180339887}^n$$

Commentaires : Nous pouvons remarquer que nous avons déjà la correspondance d'égalité entre les deux coefficients de la matrice A^n ainsi calculée.

En écrivant donc nos commandes sous forme d'une fonction nous pouvons vérifier que nous obtenons les mêmes valeurs que la fonction `numlib : :fibonacci` :

```
Fibo(10);
55.0

Fibo2(10);
55.0

numlib::fibonacci(10);
55
```

5.3 Mesures et graphes

Pour cette partie, n appartient à l'interval $[1 * 10^4..1 * 10^5]$.

5.3.1 Compraison

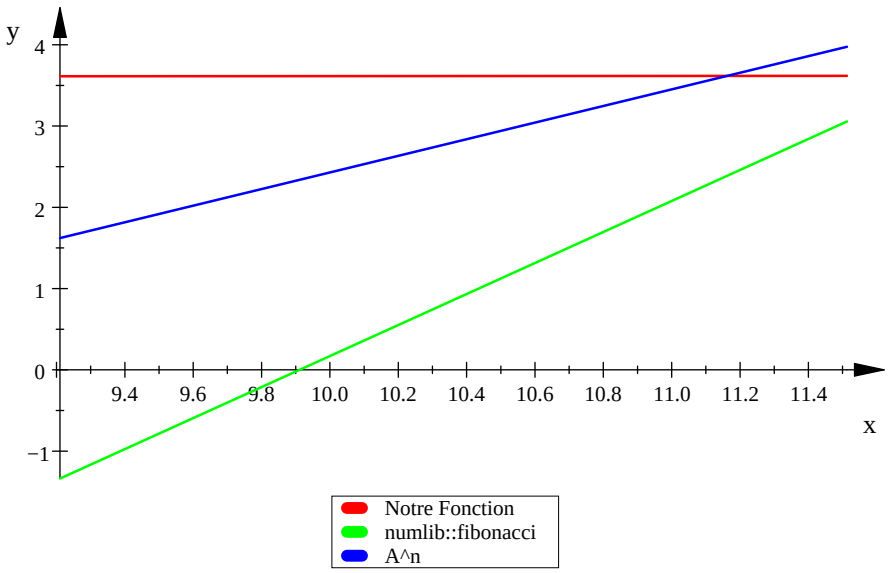

FIG. 9 – compraison A^n, `numlib : :fibonacci` et notre fonction en fonction de n

Commentaires : Lors de nos mesures, nous avons obtenu un coefficient de **1,05** pour le calcul de A^n par méthode direct de `MuPAD`.

$\Rightarrow$ Ainsi, nous pouvons dire que cette méthode implémente au moins un algorithme de type dichotomique.

En effet, nous avons trouvé une valeur de 1,09 pour notre implémentation dichotomique itérative et l'étude de `powermod` a également révélée un coefficient proche de 1.

Nous pouvons énoncer les mêmes hypothèses en ce qui concerne le fonctionnement de `numlib : :fibonacci` puisque nous obtenons la même allure de courbe pour le même intervalle de données et un coefficient de 1,90 avec une complexité annoncée est de $O(n^2)$.

En ce qui concerne notre fonction, nous avons juste simplifier le calcul de la matrice A^n par celle de sa matrice diagonale D^n. Comme expliqué précédemment, ce dernier revient à calculer des valeurs simples pour les coefficients de la matrice du type $diag(d_{ij}^k)$. Nous obtenons par contre une « courbe quasi-constante » comme ce que nous avions rencontré lorsque nous avions utilisé un intervalle de valeur trop petit lors de l'étude de la fonction `powermode`.