quelques tests de rapidité entre différents logiciels matriciels

Bruno Pinçon

Institut Elie Cartan Nancy
E.S.I.A.L.
Université Henri Poincaré
Email : Bruno.Pincon@iecn.u-nancy.fr

Table des matières

1 Introduction

Ce document présente un certain nombre de tests de rapidité concernant les logiciels de calcul matriciel suivants :

- matlab ;
- octave ;
- scilab ;
- nsp.

Ces tests ne portent pas sur les bibliothèques extérieures de calcul (sauf éventuellement BLAS) mais cherchent plutôt à jauger la rapidité de ces interpréteurs[1]. Concernant les temps de calcul, les trois logiciels disponibles en code source (scilab, octave et nsp) ont été compilés avec gcc 3.4.3 avec les mêmes options (`-O2 -march=pentium-m`), la machine ayant servie aux tests étant mon portable Dell D400 (1400 MHz, 512 MO). Ils utilisent aussi tous les trois la même bibliothèque BLAS (Atlas) adaptée à cette machine. Pour les temps avec matlab, il s'agit de la version 7.1 que j'utilise (sur la même machine) sous Windows XP. Pour scilab il s'agit d'une version scilab-gtk très récente avec correction du bug 2386[2]. Les temps sont aussi donnés avec la version Windows binaire la plus récente (build 4-16748) de scilab[3] qui doit aussi intégrer la correction du bug 2386.

> **Sur une autre machine, avec un autre compilateur ou d'autres versions de gcc les rapports entre les temps obtenus peuvent certainement différer.**

Concernant les codes nsp, *ils sont tous identiques* aux codes scilab. En utilisant certaines spécificités de nsp (essentiellement des méthodes qui permettent de modifier leur argument "en place") certains codes peuvent être plus rapides. Les codes octave et matlab, sont dans la mesure du possible le plus équivalent possible aux codes scilab.

[1] une partie des tests ne correspond pas vraiment à une utilisation/programmation usuelle mais permet plutôt de juger telle ou telle caractéristique et/ou fonction.

[2] Ce bug ralentissait les mouvements mémoires dans la stack scilab dans certains cas.

[3] j'ai aussi effectué des tests avec la version Windows binaire 4.1.1 mais elle est moins rapide que la build 4-16748 plus récente.

Les tests pour scilab et nsp sous unix n'utilisent pas l'interface graphique (option `-nw` de la ligne de commande). Le test pour scilab sous Windows est a priori en mode graphique.

Finalement pour ne pas trop privilégier tel ou tel codes, j'ai fait en sorte que tous les temps de calcul soit, pour nsp, de l'ordre de 1 à 2 secondes (éventuellement en répétant l'opération).

> *La dernière partie du rapport présente un petit test avec des matrices de pointeurs (Cells). Là il est clair que la technologie "stack" de scilab fait qu'il est fortement handicapé et j'ai préféré ne pas insérer ce test dans le benchmark précédent.*
>
> *J'aurais aimé rajouter quelques explications et aussi montrer le bénéfice que peut apporter l'utilisation de méthodes avec nsp, ce sera pour une autre fois... Il serait aussi sans doute instructif de voir les résultats obtenus avec d'autres logiciels comme yorick ou python (avec les extensions numpy et scipy).*

2 Utilisation du benchmark

L'archive fournie comporte deux répertoires `Sci` et `Mtlb`. Pour utiliser ce bench il suffit pour scilab ou nsp de les lancer à partir du répertoire `Sci` et de rentrer :

```
-->exec bench.sce;
```

Les résultats apparaissent à l'écran et aussi sont sauvegardés dans un fichier. Pour changer le nom de ce fichier, il faut aller modifier la variable `filename` au début du script `bench.sce`.

Pour matlab et octave, il faut les lancer à partir du répertoire `Mtlb` et de rentrer la commande `bench` à l'invite. De même pour changer le nom du fichier de sauvegarde il faut modifier la variable `filename` au début du script `bench.m`.

3 Les résultats

3.1 Tableau récapitulatif

	octave 2.9.10	matlab 7.1	nsp	scilab-unix	scilab-win
fibonnaci	5.768	2.153	1.230	1.460	1.362
subsets	6.444	1.883	1.484	2.760	2.854
irn55_rand	7.485	1.372	1.750	1.560	1.692
tri fusion	7.881	0.571	1.407	1.430	1.282
tri rapide	8.144	1.973	1.494	1.900	1.612
harmvect	1.745	1.702	1.169	1.230	1.152
harmloop	5.486	0.030	1.876	1.790	1.692
fannkuch	7.433	0.831	1.343	0.900	0.831
mon_lu	3.135	2.023	1.544	1.850	1.953
crible	1.092	0.981	0.985	1.210	1.432
make_perm	5.431	1.021	1.146	1.740	1.472
inv_perm	6.206	0.010	1.128	0.910	0.831
fftx	18.020	3.104	1.737	7.290	7.140
pascal	3.413	0.481	1.084	1.310	2.964
hmeans	4.351	1.833	1.605	3.090	4.987
simplexe	1.883	1.072	1.248	1.630	1.773
loop_call_f	6.611	0.100	1.306	1.710	1.292
loop_call_p	3.706	0.300	1.039	0.470	0.461
form_vect1	1.963	2.664	0.029	2.110	4.266
form_vect2	4.230	2.864	0.815	1.390	3.024
loop1	4.576	0.010	1.407	1.330	1.242
loop2	14.155	0.040	2.115	1.960	1.853
loop3	5.378	0.010	1.644	1.550	1.452
test bool	1.416	3.986	1.238	2.840	3.175
test find	4.801	2.363	1.417	4.490	4.857
prime_factors	17.843	0.060	0.851	5.000	4.236
extraction	3.557	1.562	1.189	3.080	3.124
insertion	2.209	1.242	1.177	2.270	2.293
temps total	164.36	36.24	36.45	60.26	66.31

3.2 Commentaires

1. les temps calculs pour octave sont assez supérieurs aux trois autres logiciels ; une explication possible est que l'évaluateur d'octave ne serait pas suffisamment économe dans l'utilisation de la mémoire[4]. Pour nsp nous avons beaucoup progressé en faisant attention à cet aspect ce qui me suggère cette explication qui n'est sans doute qu'un facteur parmi d'autres.
2. lorsque le temps de calcul pour Matlab est bien inférieur c'est qu'il a certainement réussi à utiliser son compilateur à la volée (dénommé JIT) , c'est évident pour les tests harmloop, invperm, loop1, loop2, loop3, prime_factor.

[4]Ce qui reste à prouver en profilant le code

3. initialement, du fait de l'allocation via malloc comparée au système "stack" de scilab, on s'attendait à ce qu'nsp soit plus lent que scilab sur les codes "scalaires"[5] :
   - tri fusion
   - harmloop
   - fannkuch
   - make_perm
   - inv_perm
   - loop1, loop2 et loop3
   - prime_factor

   Cette tendance se retrouve effectivement (sauf pour make_perm et prime_factor) mais nsp n'est jamais très loin (sauf peut être sur fannkuch).
4. par contre pour les codes qui manipulent de gros tableaux (avec une complexité proportionnelle nombre d'éléments du tableau) comme :
   - hmeans
   - test bool
   - test find
   - insertion
   - extraction

   nsp prend le pas sur scilab d'un facteur 2 et parfois plus. Ceci est du en partie aux mouvements mémoires supplémentaires exigés par la gestion "stack". Ce qui est étrange ce sont les résultats d'octave sur ces mêmes tests.
5. le temps de fftx pour scilab semble trop long : peut être un problème pour certaines opérations avec les nombres complexes ?
6. nsp est assez bon sur les tests form_vect1 et form_vect2 où l'on augmente systématiquement la taille d'un vecteur. C'est sans doute du à l'utilisation de la fonction `realloc` qui doit éviter dans certains cas des copies inutiles.

4 Description des tests

4.1 Nombres de Fibonnaci

Ce test permet de juger la rapidité d'évaluation d'une fonction récursive toute simple[6].
Le test consiste à calculer :

```
f = fib(24);
```

Avec :

```
function f=fib(n)
   // calcul du n ieme terme de la suite de Fibonnaci :
   // fib(0) = 1, fib(1) = 1, fib(n+2) = fib(n+1) + fib(n)
   if n <= 1 then
      f = 1
   else
      f = fib(n-1) + fib(n-2)
   end
endfunction
```

[5] qui ne manipulent quasiment que des varaibles scalaires et pas de gros tableaux.

[6] et constitue un mauvais exemple d'utilisation de la récursivité : il ne faut jamais calculer ces nombres de cette manière !

4.2 Sous-ensembles à m éléments d'un ensemble à n éléments

Il s'agit encore d'une fonction récursive mais elle permet aussi de tester l'extraction et la concaténation.

Le test :

```
F = subsets(1:20,7);
```

La fonction :

```
function F = subsets(E,m)
   // compute all the subsets with m elements of the set E
   // (E must be given as a row vector and all its elements
   // must be differents). For sets of numbers.
   n = length(E)
   if m > n then
      F = []
   elseif m == n
      F = E
   elseif m == 1
      F = E';
   else
      F = [];
      for i = 1:n-m+1
         EE = E(i+1:n)
         FF = subsets(EE,m-1)
         mm = size(FF,1)
         F = [F ; E(i)*ones(mm,1),FF]
      end
   end
endfunction
```

4.3 Un générateur aléatoire

J'ai écrit cet ensemble de fonctions pour avoir un générateur portable entre les 4 logiciels. En effet, dans la suite, certains codes prennent en entrée des vecteurs ou matrices aléatoires et leur nombre d'opérations peut dépendre précisemment des nombres générés. J'ai trouvé intéressant de le mettre dans ces tests. Il est bien sûr plus rapide d'utiliser `grand` mais comme ce générateur présente un aspect vectoriel il n'est que 40 fois moins rapide que `rand` (sur un code non vectorisable la différence est généralement bien plus grande). Avant toute utilisation il faut penser à initialiser le générateur avec `irn55_init(ix)` ix étant un entier entre 0 et 999999999.

Les appels sont de la forme :

```
X = irn55_rand(m,n)       // matrice m x n de réalisations de v.a. suivant U[0,1)
X = irn55_rand(m,n,n1,n2) // matrice m x n de réalisations de v.a. suivant U{n1,n1+1,...,n2}
```

Le test consiste à tirer un million de nombres aléatoires :

```
X = irn55_rand(1,1e6);
```

Les codes de ce générateurs :

```
// portable random number generator between scilab, nsp, matlab, octave
// see Knuth, 2d edition, TAOCP tome 2, p 171 et 172
// coded by Bruno Pincon
//
function irn55_adv_state()
   global irn55_state irn55_ind
   J = irn55_state(1:24) - irn55_state(32:55)
   ineg = find(J < 0);
   J(ineg) = J(ineg) + 1e9
   irn55_state(1:24) = J

   J = irn55_state(25:48) - irn55_state(1:24)
   ineg = find(J < 0);
   J(ineg) = J(ineg) + 1e9
   irn55_state(25:48) = J

   J = irn55_state(49:55) - irn55_state(25:31)
   ineg = find(J < 0);
   J(ineg) = J(ineg) + 1e9
   irn55_state(49:55) = J

   irn55_ind = 1
endfunction

function X = irn55_rand(m,n)
   global irn55_state irn55_ind
   mn = m*n
   X = zeros(mn,1)
   i = 0
   while mn > 0
      nb = min(mn,56 - irn55_ind)
      if nb == 0 then
         irn55_adv_state()
         nb = min(mn,55)
      end
      X(i+1:i+nb) = 1e-9*irn55_state(irn55_ind:irn55_ind+nb-1)
      irn55_ind = irn55_ind + nb
      i = i + nb
      mn = mn - nb
   end
   X = matrix(X,m,n)
endfunction

function X = irn55_irand(m,n,n1,n2)
   // random integers between n1 and n2
   X = n1 + floor((n2-n1+1)*irn55_rand(m,n))
endfunction

function [] = irn55_init(ix)
   global irn55_state irn55_ind
   if length(ix) ~= 1 | floor(ix) ~= ix | ix < 0 | ix > 999999999 then
      error("bad value for ix")
   end
   irn55_state = zeros(55,1)
   irn55_state(55) = ix
   j = ix
```

```
   k = 1
   for i=1:54
      ii = modulo(21*i, 55)
      irn55_state(ii) = k
      k = j - k
      if k < 0 then, k = k + 1e9, end
      j = irn55_state(ii)
   end
   irn55_adv_state()
   irn55_adv_state()
   irn55_adv_state()
endfunction
```

4.4 Tri fusion

Dans cette fonction la difficulté est la partie "fusion" des deux sous tableaux triés car elle ne peut pas se faire vectoriellement.

Le test consiste à trier un tableau de 10000 nombres (obtenu avec la fonction `irn55_rand`) :

```
y = merge_sort(x);
```

Le code :

```
function [x] = merge_sort(x)
   n = length(x)
   if n > 1 then
      m = floor(n/2); p = n-m
      x1 = merge_sort(x(1:m))
      x2 = merge_sort(x(m+1:n))
      i = 1; i1 = 1; i2 = 1;
      for i = 1:n
         if x1(i1) <= x2(i2) then
            x(i) = x1(i1)
            i1 = i1+1
            if (i1 > m) then, x(i+1:n) = x2(i2:p), break, end
         else
            x(i) = x2(i2)
            i2 = i2+1
            if (i2 > p) then, x(i+1:n) = x1(i1:m), break, end
         end
      end
   end
endfunction
```

4.5 Tri rapide

Ce tri peut s'écrire de façon plus vectorielle que le précédent. Il permet de tester la rapidité de la concaténation (avant dernière ligne), des comparaisons et des extractions du type *A(vecteur booléen)*.

Le test consiste à trier un tableau de 40000 nombres (obtenu avec la fonction `irn55_rand` :

```
y = qSort(x);
```

Le code :

```
function v = qSort(v)
   // écrit par F. Delebecque
   n = length(v)
   if n > 1 then
      piv =  sum(v)/n;
      low = v(v < piv);
      eql = v(v == piv);
      gtr = v(v > piv);
      v = [qSort(low), eql, qSort(gtr)];
   end
endfunction
```

4.6 nombres harmoniques

Voici deux codes pour calculer :

$$H_n = \sum_{k=1}^{n} \frac{1}{k}$$

l'un vectoriel et qui est celui qu'on devrait utiliser si l'on veut effectivement calculer H_n avec ces logiciels et l'autre "purement" scalaire. L'appel au code vectoriel est répété 25 fois pour avoir un temps de calcul suffisamment grand.

Les tests associés :

```
for k=1:25,s = harmvect(1e6);end;  // version vectorielle
s = harmloop(1e6);                 // version scalaire
```

```
function s=harmvect(n)
   s = sum( 1 ./(1:n) )
endfunction

function s=harmloop(n)
   s = 1;
   for i=2:n
      s = s + 1/i
   end
endfunction
```

4.7 fannkuch

Il s'agit d'un code assez complet et peu vectoriel (on l'utilise pour $n = 7$). Sa description peut se trouver à l'url :

`http://shootout.alioth.debian.org/gp4/benchmark.php?test=fannkuch&lang=all`

Le test : l'appel est répété 3 fois[7] :

```
for k=1:3, mf = fannkuch(7); end;
```

Le code :

[7] l'appel à `fannkuch(8)` étant trop long.

```
function [maxflips]=fannkuch(n)
   //
   q = 1:n
   maxflips = 0
   while %t
      p = q
      f = p(1);
      flips = 0
      while f ~= 1
         p(1:f) = p(f:-1:1)
         f = p(1); flips = flips+1
      end
      maxflips = max(maxflips,flips)

      // got next permutation q
      k = max(find(q(1:n-1) < q(2:n)))
      if k==[] then, break, end
      jsearch = find(q(k+1:n) > q(k)) + k
      [m,j] =min(q(jsearch))
      j = jsearch(j)
      q([j,k]) = q([k,j])
      q(k+1:n) = q(n:-1:k+1)
   end
endfunction
```

4.8 Factorisation LU

Voici un code pour calculer une factorisation $A = LU$ sans permutation. Il est inutile car tous ces logiciels sont interfacés avec une routine Lapack effectuant ce travail. Néanmoins cette version vectorielle (`mon_lu`) permet de tester la rapidité pour des grosses extractions et insertions.

Le test consiste à effectuer la factorisation d'une matrice 400 x 400 :

```
[L,U] = mon_lu(A);
```

Le code :

```
function [L,U] = mon_lu(A)
   [m,n] = size(A)
   if  m ~= n  |  m == 0 then, error(" bad dim for A"),end
   for k = 1:n-1
      if ( A(k,k) == 0 ) then
         error(" zero pivot ...")
      end
      A(k+1:n,k) = A(k+1:n,k)/A(k,k)
      A(k+1:n,k+1:n) = A(k+1:n,k+1:n) - A(k+1:n,k)*A(k,k+1:n)
   end
   L = tril(A,-1)+ eye(A)
   U = triu(A)
endfunction
```

4.9 Crible d'Erathostène

Un grand classique, il permet de calculer tous les nombres premiers inférieurs ou égaux à n.

Le test consiste à calculer tous les nombres premiers inférieurs à 3 million :

```
p = crible(3e6);
```

Le code :

```
function [vp] = crible(n)
   // Sieve of Erathostenes; return a vector of all primes vp that are
   // less or equal than n.  1 is not considered to be a prime number
   // in crible().
   if n < 2 then
      vp = []
   else
      sieve = 1:n
      sieve(1) = 0  // 1 is not a prime
      for i=2:sqrt(n)
         // unset all multiples of i if i has not already been unset
         if sieve(i) ~= 0 then
            sieve(2*i:i:n) = 0
         end
      end
      vp = sieve(sieve ~= 0)
   end
endfunction
```

4.10 Construction puis inversion "en place" d'une permutation

Ce sont deux codes "scalaires". Les 2 tests (l'inversion est répété 4 fois) :

```
p = makeperm(20000);           // construction
for k=1:4,q = invperm(p);end; // inversion
```

Les codes :

```
function p = makeperm(n)
   // tire une permutation de (1,...,n) au hasard
   p = 1:n
   for k = 1:n-1
      i = irn55_irand(1,1,k,n);
      p([i,k]) = p([k,i])
   end
endfunction

function [p] = invperm(p)
   // inversion d'une permutation en place
   n = length(p)
   m = 1
   while m <= n
      next = p(m)
      if next < 0 then // deja traite
         p(m) = -p(m)
      else
         current  = m
         while next ~= m   // on est tj ds le meme cycle
            new_next = p(next)
            p(next) = - current
```

```
            current = next
            next = new_next
         end
         p(m) = current
      end
      m = m + 1
   end
endfunction
```

4.11 Transformée de Fourier rapide

Elle nous vient de Mr Cleve Moler himself! Je pense qu'elle permet de bien tester l'arithmétique complexe ainsi que la concaténation. Remarque : dans la vraie vie il faut utiliser la fonction `fft` !

Le test consiste à calculer la fft d'un vecteur de 2^{15} composantes :

```
x = rand(2^15,1);
timer(); y = fftx(x); t=timer()
```

Le code :

```
function y = fftx(x)
  // FFTX Fast Finite Fourier Transform. algo C. Moler
  // paru ds la lettre de Cleve Moler...
  x = x(:);
  n = length(x);
  omega = exp(-2*%pi*%i/n);
  if modulo(n,2) == 0 & n > 2 then
     // Recursive divide and conquer.
     k = (0:n/2-1)';
     w = omega.^k;
     u = fftx(x(1:2:n-1));
     v = w.*fftx(x(2:2:n));
     y = [u+v; u-v];
  else
     // The Fourier matrix.
     j = 0:n-1;
     k = j';
     F = omega.^(k*j);
     y = F*x;
  end
endfunction
```

4.12 Triangle de Pascal

Ce code permet de calculer les coefficients binomiaux C_N^k pour $k = 0, \ldots, N$. Il est vectorisé. Il ne faut pas aller au delà de $N = 1029$ puisqu'on dépasse alors le plus grand nombre flottant. Il s'ensuit des calculs avec des Infs puis des Nans.

Le test : l'appel est répété 30 fois :

```
for i=1:30,T = pascal(1029);end
```

Le code :

```
function [C] = pascal(N)
  // computes binomial coefs
  if N < 0 | floor(N)-N ~= 0 then
    error(" no binomial coefficients with N = "+string(N))
  else
    if N == 0 then
      C = 1
    else
      C = zeros(1,N+1) ; C(1:2) = 1
      for i=2:N
        C(2:i+1) = C(1:i) + C(2:i+1)
      end
    end
  end
endfunction
```

4.13 Classification par hmeans

A partir d'un nuage de n points et d'un nombre entier K cet algorithme stochastique essaie de séparer le nuage en K groupes en utilisant la distance euclidienne. C'est un problème difficile sur le plan algorithmique et cette méthode (comme la plupart des méthodes) ne cherche pas l'optimum global. Dans le cas d'une utilisation normale l'algorithme doit être utilisé plusieurs fois.

Le test consiste à classer 8000 points de $\mathbb{R}^8$ en 4 classes[8] :

```
irn55_init(11);
XX = irn55_rand(8,8000);
[classe] = init_classe(XX,4);
timer(); [classe, I, suiteI] = hmeans(XX, classe); t=timer();
```

Les codes :

```
function [classe, I, suiteI] = hmeans(X, classe)
   //
   //  algorithme hmeans :
   //     X est un nuage de n points de R^p que l'on cherche
   //     à partitionner en K classes.  X est un tableau pxn
   //     classe : donne la classe initiale pour chaque point
   //              (et la classe finale en sortie)
   [p, n] = size(X)
   K = max(classe)
   v = ones(1,n)
   tab = zeros(K,n)
   C = zeros(p,K)
   i_old = %inf
   suiteI = []
   nb_iter = 0

   while %t
      // calcul des nouveaux centres par barycentrage
      for k = 1:K
```

[8] C'est non réaliste car les points étant obtenus comme des réalisations de variables aléatoires uniformes dans l'hypercube de dimension 8 il n'y aucune raison qu'ils forment des groupes dissociés.

```
         indk = find(classe == k)
         C(:,k) = sum(X(:,indk), "c")/length(indk)
      end

      // attribution de la classe de chaque point
      for k = 1:K  // calcul du tableau des distances
         tab(k,:) = sum( (X - C(:,k)*v).^2 , "r" )
      end
      [I, classe] = min(tab,"r")

      i_new = sum(I)

      suiteI = [ suiteI ; i_new];

      if (i_new == i_old) then, break, end
      i_old = i_new

      nb_iter = nb_iter + 1
   end
endfunction


function [classe] = init_classe(X,K)
   [p, n] = size(X)
   // il faut tirer K entiers de {1,...,n] tous distincts
   ind = zeros(1,K)
   ind(1) = irn55_irand(1,1,1,n)
   for k = 2:K
      while %t
         u = irn55_irand(1,1,1,n);
         if isempty(find(ind(1:k-1) == u)) then, break, end
      end
      ind(k) = u
   end
   C = X(:,ind(1:K))
   v = ones(1,n)
   tab = zeros(K,n)

   // attribution de la classe de chaque point
   for k = 1:K  // calcul du tableau des distances
      tab(k,:) = sum( (X - C(:,k)*v).^2 , "r" )
   end
   [I, classe] = min(tab,"r")
endfunction
```

4.14 Simplexe

Un code utilisant la méthode du simplexe révisée basique[9] pour résoudre :

$$\begin{array}{c} \max \quad cx \\ x \in \mathbb{R}^n \\ Ax \leq b \\ x \geq 0 \end{array}$$

[9] cad avec mise à jour rapide de l'inverse de la base et pas d'une factorisation de la base.

où $c \in (\mathbb{R}^n)^*$ (c est un vecteur ligne), $A \in \mathbb{R}^{m \times n}$, la matrice des m contraintes, $b \in \mathbb{R}^m$ le second membre des contraintes avec aussi $b \geq 0$ (toutes les composantes de b sont positives). Dans cette configuration on connait une solution de base réalisable[10] ($x = 0$) et on a pas à recourir à la première phase du simplexe.

Le test consiste à utiliser le code sur un problème avec 200 contraintes et 300 variables :

```
[A,b,c] = gen_lp(200,300);
[xopt, copt, slack, info, J, iter] = simplex(c, A, b, 400, %f);
```

Les codes associés :

```
function [A,b,c] = gen_lp(m,n)
   A = ones(m,n) + 0.1*irn55_rand(m,n);
   c = ones(1,n)
   b = ones(m,1) + 0.1*irn55_rand(m,1);
endfunction

function [xopt, copt, slack, info, J, iter] = simplex(c, A, b, itermax, verb)
//
// solve  Max c x
//        st  Ax <= b
//             x >= 0
// where b >= 0
//
// using basic revised simplex method
//
// info = 0 : itermax reached (without cvg)
// info = 1 : normal output
// info = 2 : unbounded pb
// info = 3 : singular basis matrix

   [m,n] = size(A)
   J = n+1:n+m
   K = 1:n
   B = eye(m,m)
   A = [A,B]
   c = [c zeros(1,m)]
   xopt = zeros(n+m,1)
   iter = 0

   while %t

      xopt(J) = B*b; xopt(K) = 0;
      copt = c*xopt
      if verb then
         printf("\n iter = %d, copt = %e",iter, copt)
      end

      // compute reduced costs
      cr = c(:,K) - (c(:,J)*B)*A(:,K)
      [crmax, i] = max(cr)
      if crmax <= 0 then
         info = 1, break
      elseif iter >= itermax then
```

[10] cad vérifiant les contraintes.

```
         info = 0, break
      end
      ig = K(i) // global index of the input variable

      // compute output variable
      y = B*A(:,ig)
      ip = find(y>0)
      if isempty(ip) then, info = 2, break, end
      [xmin,jmin] = min( xopt(J(ip))./y(ip) )
      j = ip(jmin)
      jg = J(j) // global index of the output variable

      // updates
      J(j) = ig;
      K(i) = jg
      z = B*(A(:,ig)-A(:,jg))
      if z(j) == -1 then, info = 3, break, end
      B = B - z*(B(j,:)/(1+z(j)))
      iter = iter + 1
   end

   if verb then, printf("\n"), end

   slack = xopt(n+1:m+n)
   // delete slack variables
   xopt(n+1:m+n) = [];

endfunction
```

4.15 Tests particuliers

Ces tests n'ont aucun objectif algorithmique (sauf le dernier) et servent à tester des points précis comme la rapidité des boucles for, while, la rapidité d'appels à des fonctions et ou des primitives, la rapidité des opérateurs booléens et de comparaisons, la rapidité de réallocation, etc...

4.15.1 appels à des macros utilisateurs : loop_call_f

Le test :

```
x = rand(30000,1);
timer(); y=loop_call_f(x); t=timer();
```

Les fonctions :

```
function [y] = loop_call_f(x)
   y = 0
   for i = 1:length(x)
      z = x(i)
      y = f1(z) + f2(z) + f3(z) + f4(z) + f5(z) + f6(z) + f7(z)
   end
endfunction

function [y] = f1(x), y = x, endfunction
function [y] = f2(x), y = 2*x, endfunction
```

```
function [y] = f3(x), y = 3*x, endfunction
function [y] = f4(x), y = 4*x, endfunction
function [y] = f5(x), y = 5*x, endfunction
function [y] = f6(x), y = 6*x, endfunction
function [y] = f7(x), y = 7*x, endfunction
```

4.15.2 appels à des primitives : loop_call_p

Le test :

```
x = rand(50000,1);
timer(); y=loop_call_p(x); t=timer();
```

La fonction :

```
function [y] = loop_call_p(x)
   y = 0
   for i = 1:length(x)
      z = x(i)
      y = sin(z) + cos(z) + exp(z) + log(z) + gamma(z) + atan(z) ...
  + floor(z) + ceil(z)
   end
endfunction
```

4.15.3 construction dynamique d'un vecteur : form_vect1

Le test :

```
v = form_vect1(20000);
```

La fonction :

```
function v = form_vect1(n)
   v = []   // inutile normalement...
   for i=1:n
      v(i) = i
   end
endfunction
```

4.15.4 autre construction dynamique d'un vecteur : form_vect2

Le test :

```
v = form_vect2(20000);
```

La fonction :

```
function v = form_vect2(n)
   v = [];
   for i=1:n
      v = [v,i]
   end
endfunction
```

4.15.5 boucle simple : loop1

Le test :

```
v = loop1(1e6);
```

La fonction :

```
function s = loop1(n)
   s = 0
   for i=1:n
      s = s + i
   end
endfunction
```

4.15.6 autre boucle simple : loop2

Le test :

```
v = loop2(1e6);
```

La fonction :

```
function s = loop2(n)
   x = 1:n
   s= 0
   for i=1:n
      s = s + x(i)
   end
endfunction
```

4.15.7 boucle avec quelques instructions : loop3

Le test :

```
v = loop3(3e5);
```

La fonction :

```
function s = loop3(n)
   s = 0
   t = 1
   u = 2
   v = 3
   for i=1:n
      s = s + 1
      t = t + v/u
      u = s + u
      v = t + v
   end
endfunction
```

4.15.8 test sur les opérateurs booléens et les opérateurs de comparaison

Le test utilise un vecteur d'un million de composantes et est répété 5 fois :

```
for k=1:5,s = test_bool_and_comp_ops(X); end;
```

La fonction :

```
function s = test_bool_and_comp_ops(x)
   s =  ~( 0.2 < x & x < 0.3 | 0.5 < x & x <= 0.6 | 0.7 <= x & x < 0.8 | ...
   0.9 < x & x <=0.99 )
endfunction
```

4.15.9 test sur la fonction find

La fonction `find` permettant de vectoriser des tests est essentielle dans ces langages.
Le test utilise un vecteur d'un million de composantes et est répété 40 fois :

```
for k=1:40,s = test_find(X); end;
```

La fonction :

```
function s = test_find(x)
   s = find(x < 0.5)
endfunction
```

4.15.10 décomposition en facteurs premiers

Ce test, utilisé sur un nombre premier, permet essentiellement de juger de la rapidité de la boucle `while`.

Le test :

```
d = prime_factors(160001);
```

La fonction :

```
function [d] = prime_factors(n)
   // stupid prime factorisation : this is in fact
   // a test for while loop speed
   d = []
   if ( n ~= floor(n) | n < 2 ), return, end
   p = 2;
   while n ~= 1
      // on teste si n est divisible par p
      while modulo(n,p) == 0
       d = [d,p]
       n = n/p
      end
      p = p+1
   end
endfunction
```

4.15.11 extraction et insertion

Ces deux tests permettent de juger la rapidité de grosses extractions et insertions (opérations répétées 100 fois chacune) :

```
A = rand(900,900);
i = irn55_irand(1,700,1,900);
j = irn55_irand(1,700,1,900);

timer(); for k=1:100, B=A(i,j); end; t=timer();gt=gt+t;all=[all;t];
printf("\n extraction test---------- %g s",t)

timer(); for k=1:100, A(i,j) = B; end; t=timer();gt=gt+t;all=[all;t];
printf("\n insertion test----------- %g s",t)
```

5 Mini test avec des Cells

Dans ce test on préalloue la mémoire du tableau de pointeurs puis on remplit les cases avec des objets de taille variable (le temps de ces 2 opérations est appelé temps de construction). Dans un deuxième temps, on extrait successivement tous les objets pointés. Le code pour nsp est identique au code pour matlab/octave[11]. Pour lancer ce test il suffit d'exécuter le script `bench_cells.sce` ou `bench_cells.m`.

tableau des résultats[12]

	octave	nsp	scilab
temps construction	0.045	0.009	16.17
temps lecture	0.009	0.004	17.27

Le code scilab / nsp :

```
// mini bench cells
n = 600;
if exists('%nsp') then
  // code nsp
  timer();
  H=cells_create(1,n);
  for i=1:n; H{i} = rand(1,i); end
  t1 = timer();
  for i=1:n; obj = H{i};end
  t2 = timer();
else
  // code scilab
  timer();
  H=cell(1,n);
```

[11] modulo le raccourci `cells` pour `cells_create` qui n'est pas encore défini.

[12] Il manque le temps pour Matlab et scilab sous windows : pour une prochaine fois...

```
  for i=1:n, H(i).entries=rand(1,i); end
  t1 = timer();
  for i=1:n; obj = H(i).entries; end
  t2 = timer();
end

printf("\n bench cells : temps construction = %g", t1);
printf("\n bench cells : temps lecture = %g \n", t2);
```

Le code pour octave / matlab (identique au code nsp modulo la fonction cputime) :

```
% mini bench cells
n = 600;
tinit = cputime();
H=cell(1,n);
for i=1:n; H{i} = rand(1,i); end
t1 = cputime() - tinit;
for i=1:n; obj = H{i};end
t2 = cputime() - tinit - t1;

fprintf(1,'\n bench cells : temps construction = %g', t1);
fprintf(1,'\n bench cells : temps lecture = %g \n', t2);
```