Rapport TP :
Résolution numérique de la diffusion de chaleur dans une plaque par différence finis

Dupont Ronan
Croguennec Guillaume

TP Différences Finies 2019 MOCA 2A

Table des matières

1) Algorithme du Gradient conjugué

1.1) Cas-tests élémentaires

Dans cette première partie, le but est de créer un algorithme pour résoudre Ax=b.
On choisit d'utiliser la méthode itérative du gradient conjugué qui converge en maximum l équations (l étant la dimension de la matrice A).

Mais avant cela nous créons des matrices A symétriques positives afin d'essayer plus tard notre algorithme du gradient conjugué.

```
Définition de A
eps=0.2
do i=1,n
   do j=1,n
      if (i.eq.j) then
         A(i,j)=5.d0*i
      else
         A(i,j)=eps
   enddo
enddo
```

Voici comment l'on créé l'une de ces matrices A. On peut facilement la modifier en modifiant juste e ou en prenant une autre valeur que 5*i pour les A(i,i).
On créé ensuite des matrices b de manière à obtenir un vecteur x rempli de 1.

```
Définition de b
do i=1,n
   xx(i)=1
enddo
call matvop(l,xx,A,b)
```

Ainsi on devrait obtenir un vecteur x quasiment unitaire tant qu'on garde une « grande » valeur à la place du 5 de 5*i pour les A(i,i) et une petite valeur pour e.

1.2) Programmation

On peut maintenant faire l'algorithme du gradient conjugué dont on pourra vérifier qu'il marche bien grâce aux matrices A et aux matrices b.
Pour réaliser cet algorithme, il faut pouvoir faire des produits de matrices avec des vecteurs ainsi que des produits scalaires entre deux vecteurs. Sur matlab on peut les faire directement, mais pas sur fortran. Nous avons donc dû créer deux subroutines avant de programmer l'algorithme du gradient conjugué.

```
Calcul du produit d'une matrice avec un vecteur

            SUBROUTINE matv(n,x,A,v)
            implicit double precision(a-h,o-z)
            dimension A(1000,1000),v(1000),x(1000)

            do i=1,n
             v(i)=0.d0
               do j=1,n
                   v(i)=v(i)+A(i,j)*x(j)
               enddo
            enddo
            return
            end
```

Cette subroutine calcul un produit matrice-vecteur.

```
Calcul du produit scalaire de 2 vecteurs

            SUBROUTINE scal(n,x,y,z)
            implicit double precision(a-h,o-z)
            dimension x(1000),y(1000)

            z=0.d0
            do i=1,n
               z=z+x(i)*y(i)
            enddo
            return
            end
```

Cette subroutine calcul un produit scalaire

Une fois ces deux subtourines faites, nous pouvons donc coder l'algorithme du gradient conjugué. Pour ce faire, nous reprenons pas à pas l'algorithme du cours et nous obtenons ceci :

La méthode du gradient conjugué permettant de résoudre Ax=b se base sur le principe mathématique suivant :
On cherche x $x \in R^n \; tel \; que \; J(x) = \frac{1}{2} < Ax, x > - < b, x >$
Il en découle de ce postulat un tat d'équations mathématiques permettant de trouver x. Ces équations nous amènent à en déduire le pseudo-code suivant :

On initialise :
$v = A.x_0$
$r_0 = b - v = b - A.x_0$
$d_0 = r_0$

On répète k fois la boucle suivante :

$$\alpha_k = \frac{r_k^T . r_k}{d_k^T . A . d_k}$$
$$x_{k+1} = x_k + d_k . \alpha$$
$$r_{k+1} = b - A.x_1 = r_k - \alpha_k . A . d_k \qquad r \; est \; le \; résidus \; indiquant \; la \; précision$$
$$\beta_k = \frac{r_{k+1}^T . r_{k+1}}{r_k^T . r_k}$$
$$d_{k+1} = r_{k+1} + d_k . \beta$$

Fin de la boucle

Sur notre code, nous avons mis une condition pour que l'algorithme continue à boucler tant que le résidu a une valeur supérieure à 10^{-8}.
On aura utilisé différentes variables x0, x1 / r0, r1 afin de pouvoir garder le terme en k et en k+1.

Ceci nous donne donc l'algorithme suivant :

```
            call matvop(l,x0,A,v)

            do i=1,n
               r0(i)=b(i)-v(i)
            enddo

            do i=1,n
               d0(i)=r0(i)
            enddo

Début des itérations

             t=1000
             do while(sqrt(t).ge.0.00000001)
              call matvop(l,d0,A,w)
              call scal(n,w,d0,z)
              call scal(n,r0,r0,y)

alpha=(r.r)/(p.a.p)

              alpha0=y/z

              do i=1,n

c xk+1=xk+alp.p
                   x1(i)=x0(i)+alpha0*d0(i)

              enddo
              do i=1,n
                   call matvop(l,x1,A,g)

rk+1=rk-alpha.a.p

                   r1(i)=b(i)-g(i)
              enddo
```

$$\alpha_k = \frac{r_k^T . r_k}{d_k^T . A . d_k}$$

$$x_{k+1} = x_k + d_0 . \alpha$$

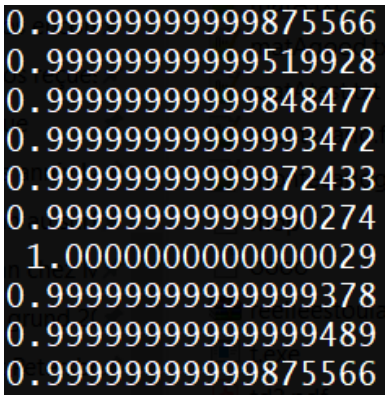
$$r_{k+1} = b - A . x_1 = r_k - \alpha_k . A . d_k$$

```
beta=(rk+1.rk+1)/(rk.rk)

              call scal(n,r1,r1,t)
              call scal(n,r0,r0,u)
              beta=t/u

              do i=1,n
                  d1(i)=r1(i)+beta*d0(i)
              enddo

              do i=1,n
                  x0(i)=x1(i)
                  d0(i)=d1(i)
                  r0(i)=r1(i)
              enddo
            enddo

Affichage des valeurs de xk

              do i=1,n
                   print*,x1(i)
              enddo
```

$$\beta_k = \frac{r_{k+1}^T . r_{k+1}}{r_k^T . r_k}$$

$$d_{k+1} = r_{k+1} + d_k . \beta$$

On essaye donc la méthode du gradient conjugué pour résoudre un système du type Ax=b avec les matrices A et b. On obtient ces vecteurs x suivants :



Ce sont quasiment des vecteurs unitaires, cela montre bien que notre méthode du gradient fonctionne. Et bien sûr plus le quotient du nombre à la place du 5 sur e est grand, plus x est proche du vecteur unitaire.
Cette méthode du gradient permet donc de résoudre tous les systèmes Ax=b, tant que A est symétrique positive. On peut donc grâce à cette méthode tenter de résoudre numériquement des problèmes physiques.

2) Différences finies pour le Laplacien

2.1) Problème de la plaque

Dans cette partie, nous nous intéressons à la résolution du problème suivant :

$$\begin{cases} -\Delta u(x,y) = 0 \\ u(x,1) = 100 \; et \; u(x,y) = 0 \end{cases} \quad sur \; D = [0,1] \text{ x } [0,1] \quad \Leftrightarrow$$

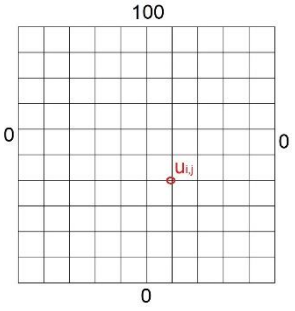


Figure 1: Grille de température

Physiquement, ces équations représentent l'évolution de la température sur D lorsqu'un bord à une température de 100 °C et que les autres ont une température nulle.

Dans cette partie, nous allons donc ramener la résolution d'un problème physique à la résolution d'une équation du type Ax=b.
Le problème physique choisi est la diffusion de l'énergie thermique dans une plaque, c'est-à-dire que l'on veut savoir la température en chaque point de la plaque dans le cas où on maintient les bords de la plaque à des températures fixes (ici de 100 °C). On aura aussi une autre condition concernant la manière dont l'énergie thermique va se diffuser dans la plaque. En effet, le mode de diffusion va être décidé par la valeur du Laplacien de u(x,y), u étant la température aux coordonnées (x,y).

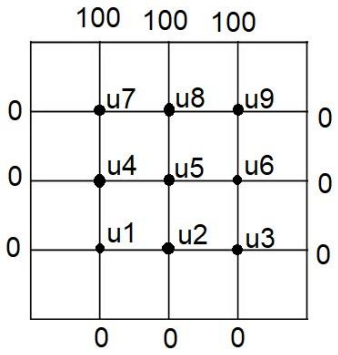
a) Création des matrices A et b

Pour se ramener à un système matriciel de type Ax=b, nous commençons dans un premier temps à étudier le modèle pour une grille 3x3.
Pour cela on va utiliser la méthode des différences finies qui consiste à discrétiser la plaque, donc à la considérer comme un nombre fini de points.

En chaque point de la grille (Figure 1), il prend la moyenne de température des 4 points autour soit :

$$u_{i,j} = \frac{1}{4}\left(u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1}\right) \Leftrightarrow \; 4u_{i,j} - u_{i-1,j} - u_{i+1,j} - u_{i,j-1} - u_{i,j+1} = 0$$

Dans notre cas n=3, nous avons la grille suivante :



On en déduit donc le système d'équation :

$$\begin{cases} 4u_1 - u_2 - u_4 = 0 \\ 4u_2 - u_1 - u_3 - u_5 = 0 \\ 4u_3 - u_2 - u_6 = 0 \\ 4u_4 - u_1 - u_5 - u_7 = 0 \\ 4u_5 - u_2 - u_4 - u_6 = 0 \\ 4u_6 - u_3 - u_5 - u_9 = 0 \\ 4u_7 - u_4 - u_8 = 100 \\ 4u_8 - u_7 - u_5 - u_9 = 100 \\ 4u_9 - u_8 - u_6 = 100 \end{cases} \Leftrightarrow \text{Ax=b}$$

Nous pouvons donc écrire ce système sous forme de système matriciel avec :

$$A = \begin{pmatrix} 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 4 & 0 & 0 & -1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 4 & -1 & 0 & -1 & 0 & 0 \\ 0 & -1 & 0 & -1 & 4 & -1 & 0 & -1 & 0 \\ 0 & 0 & -1 & 0 & -1 & 4 & 0 & 0 & -1 \\ 0 & 0 & 0 & -1 & 0 & 0 & 4 & -1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 \end{pmatrix} \quad b = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 100 \\ 100 \\ 100 \end{pmatrix}$$

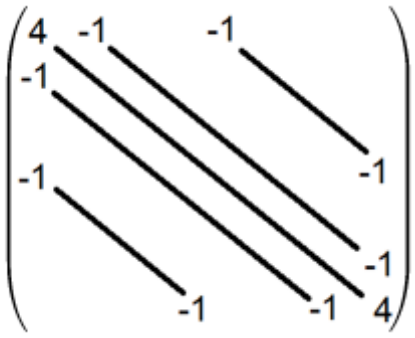
Nous avons écrit la matrice A et le vecteur b pour le cas où n=3.

Pour la matrice A, nous avons deux approches pour la construire.
La première approche consiste à considérer la matrice de taille m x m (m=n x n) avec donc m blocs de matrices de taille n x n. On peut voir sur la matrice ci-dessous qu'il est nécessaire de compléter les blocs de la diagonale ainsi que ceux qui sont autour.

Pour le vecteur b, nous avons tout simplement à affecter la valeur 100 aux n derniers termes du vecteur.

$$A = \begin{pmatrix} 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 4 & 0 & 0 & -1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 4 & -1 & 0 & -1 & 0 & 0 \\ 0 & -1 & 0 & -1 & 4 & -1 & 0 & -1 & 0 \\ 0 & 0 & -1 & 0 & -1 & 4 & 0 & 0 & -1 \\ 0 & 0 & 0 & -1 & 0 & 0 & 4 & -1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 \end{pmatrix} \quad b = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 100 \\ 100 \\ 100 \end{pmatrix}$$

La deuxième approche consiste à considérer la matrice A comme la matrice à 5 diagonales construite de la manière suivante avec des diagonales sur la diagonale 0 (centrale),1,-1, n, -n.

$A =$ 

Cependant on remarque que ce n'est pas totalement la matrice A car il y a quelques zéros sur les diagonales 1 et -1 :

$$A = \begin{pmatrix} 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 4 & 0 & 0 & -1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 4 & -1 & 0 & -1 & 0 & 0 \\ 0 & -1 & 0 & -1 & 4 & -1 & 0 & -1 & 0 \\ 0 & 0 & -1 & 0 & -1 & 4 & 0 & 0 & -1 \\ 0 & 0 & 0 & -1 & 0 & 0 & 4 & -1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 \end{pmatrix}$$

On programme donc un algorithme permettant de faire les 5 diagonales de la matrice puis de retirer les zéros.

Nous obtenons donc le programme suivant pour la création de la matrice A et du vecteur b.

Remarque : Dans notre programme, la matrice A est de taille n = l x l.

```
c On construit les diagonales de la matrice

      DO I=1,n
          DO J=1,n
              IF (i.EQ.j) THEN
              M(i,j)=4.d0
          ELSE IF (i.EQ.j-1) THEN
              M(i,j)=-1.d0
          ELSE IF (i-1.EQ.j) THEN
              M(i,j)=-1.d0
          ELSE IF (i.EQ.j-l) THEN
              M(i,j)=-1.d0
          ELSE IF (i-l.EQ.j) THEN
              M(i,j)=-1.d0
          ELSE
              M(i,j)=0
          END IF
          ENDDO
      ENDDO

c On positionne les zéros manquants

          k=n/3+1
          DO I=0,k
             M(l*i+(l+1),l*i+l)=0
             M(l*i+l,l*i+(l+1))=0
          ENDDO

  *** Définition de bl pour le problème de la plaque ***

          do i=1,n
             IF (i.GE.n-l+1) THEN
                 bl(i)=100
             END IF
          ENDDO
```

Ensuite, pour des raisons de simplicités, nous affectons la matrice M à A ainsi que le vecteur bl à b juste avant de rentrer dans le programme du grandient conjugué.

```
7.1428571428571432
9.8214285714285730
7.1428571428571432
18.750000000000000
25.000000000000004
18.750000000000000
42.857142857142854
52.678571428571431
42.857142857142854
```

Nous essayons donc pour le cas l=3 le programme et nous observons qu'il nous renvoie les valeurs suivantes :
Ces valeurs semblent cohérentes car on a bien une symétrie sur certains points. De plus, la valeur centrale a une une température de 25°C ce qui est cohérent car un côté a une température de 100 °C et les autres 0 °C.

Une fois ceci fonctionnel, on l'essai pour des l de plus en plus grands. Le programme arrive à générer une réponse jusqu'à l=31 cependant il s'avère que plus l est grand, plus le programme est lent. Nous nous intéressons donc à optimiser celui-ci en limitant les calculs.
Pour ce faire, nous allons nous intéresser au caractère creux de la matrice. En effet, la matrice est constitué de seulement 5 diagonales. Il est donc inutile d'effectuer toutes les multiplications de zéros.

b) Amélioration de la subroutine produit matrice-vecteur

Nous allons donc améliorer la subroutine de multiplication matrice-vecteur afin qu'elle prenne en compte le caractère creux de la matrice. C'est-à-dire que l'on va utiliser la fait que la matrice contienne beaucoup de 0 pour faire moins de calculs en sautant ces valeurs plutôt que de perdre notre temps à ajouter des zéros en faisant le produit scalaire.
Pour se faire, nous créons dans un premier temps une subroutine qui, comme on peut le faire en matlab, extrait une certaine diagonale (0=centrale, -1= centrale inférieur, 1=central supérieur,…) et la maltiplie avec un vecteur (un produit scalaire).

```
SUBROUTINE diagv(l,x,A,k,v)
implicit double precision(a-h,o-z)
dimension A(1000,1000),v(1000),x(1000)
n=l*l

IF (k.EQ.0) THEN
do i=1,n
   v(i)=A(i,i)*x(i)
enddo
ELSE IF (k.GT.0) THEN
do i=1,n-k
   v(i)=A(i,i+k)*x(i+k)
enddo
ELSE IF (k.LT.0) THEN
do i=-k+1,n
   v(i)=A(i,i+k)*x(i+k)
enddo
ENDIF
return
end
```

Une fois cette subroutine obtenue, il nous reste donc plus qu'à faire 5 multiplications diagonale/vecteur et le les additionner pour avoir un programme de calcul matrice/vecteur optimise. Nous le programmons et nous obtenons le programme ci-dessous :

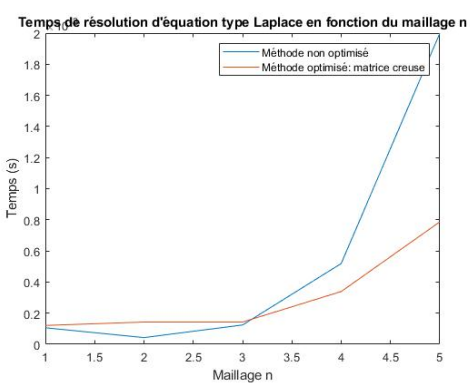
```
SUBROUTINE matvop(l,x,A,v)
implicit double precision(a-h,o-z)
dimension A(1000,1000),v(1000),x(1000)
dimension v1(1000),v2(1000),v3(1000)
dimension v4(1000),v5(1000)
n=l*l
call diagv(l,x,A,-l,v1)
call diagv(l,x,A,-1,v2)
call diagv(l,x,A,0,v3)
call diagv(l,x,A,1,v4)
call diagv(l,x,A,l,v5)
do i=1,n
   v(i)=v1(i)+v2(i)+v3(i)+v4(i)+v5(i)
enddo
```

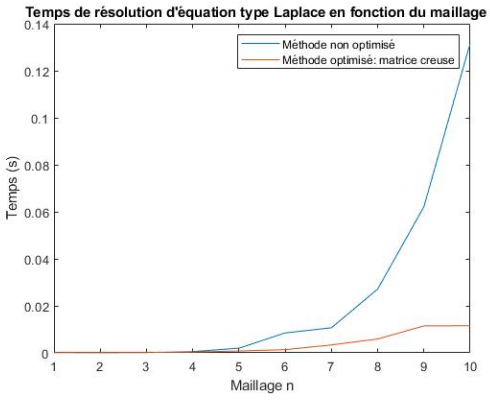
c) Temps de calcul

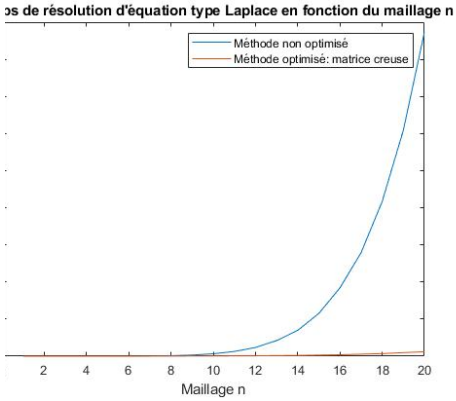
Une fois notre programme de produit matrice-vecteur obtenu, nous allons nous intéresser aux temps de calculs à la résolution de l'équation de Laplace en fonction des différents pas de discrétisation.

Pour ce faire, nous utilisons la fonction « CALL CPU_TIME » , nous plaçons une variable contenant ce temps avant le programme du gradiant conjugué, une après et nous calculons la différence en valeur absolue. Ceci nous permet donc d'obtenir le temps de calcul.

Nous stockons les différents temps de calculs pour les différents l, nous traçons les graphiques représentant les temps de calculs en fonction de la discrétisation et nous obtenons ceci :







On remarque qu'au tout début jusqu'à n=3, la subroutine matrice/vecteur semble légèrement plus performante (assez insignifiant) mais rapidement la subroutine optimisée devient beaucoup plus performante : à tel point que pour n=31, le temps de calcul avec la subroutine est de 17,5 secondes alors que pour la subroutine optimisé il est de 0.25s.

d) Erreur

Afin de calculer l'erreur générée par le programme, nous devons dans un premier temps avoir la solution exacte. Cette solution exacte en un point (x,y) étant le résultat de la série donnée par le professeur, nous programmons donc le calcul de la solution exacte pour le point (1,1) : premier élément du vecteur solution.

```
pi=4*datan(1.d0)
so=0
xt=1.d0/(l+1)
yt=1.d0/(l+1)
do k=0,100
   aux=400*dsin((2*k+1)*pi*xt)*dsinh((2*k+1)*pi*yt)/pi
   baux=(2*k+1)*dsinh((2*k+1)*pi)
   so=so+aux/baux
enddo
```

Nous calculons donc les différences entre les solutions exactes et simulées pour différents l : l=3,10,17,23,30. Nous stockons ces valeurs dans un vecteur.

Ensuite, nous traçons à l'aide de gnuplot l'erreur en fonction du pas de discrétisation en échelle logarithmique. On remarque que cette courbe est très linéaire. On cherche donc à calculer le coefficient directeur en faisant la moyenne de plusieurs coefficients directeurs calculés de la manière suivante :

```
c On calcul les erreurs:

      DO i=1,5
          xerreur(i)=dlog(abs(Uc(i)-Ur(i)))
      ENDDO

      ypas(1)=-log(4.d0)
      ypas(2)=-log(11.d0)
      ypas(3)=-log(18.d0)
      ypas(4)=-log(24.d0)
      ypas(5)=-log(31.d0)

c On calcul la moyenne des coefs directeurs:

      coefm=0
      do i=1,4
          coefm=coefm+(xerreur(i+1)-xerreur(i))/(ypas(i+1)-ypas(i))
      enddo
      coefm=coefm/4
      print*,coefm
```

On obtient donc un coefficient directeur de 1.8.

e) Représentation graphique

Afin de représenter graphiquement le gradient de température, nous devons dans un premier temps créer la matrice de température représentant la température en tout points puis ensuite créer un fichier contant les température sous la même forme qu'une matrice.
Ceci nous donne le programme suivant :

```
c Ecriture de la matrice de température

          DO I=1,l
              DO J=1,l
                  E(i,j)=x1((i-1)*l+j)
              ENDDO
          ENDDO

c Ecriture de xk dans un fichier sous forme de matrice de temperature

            OPEN(UNIT=48,FILE='heat_map_data.txt')
            DO i=1,l
              WRITE(48,*) (E(i,j),j=1,l)
            ENDDO
            CLOSE(48)
```

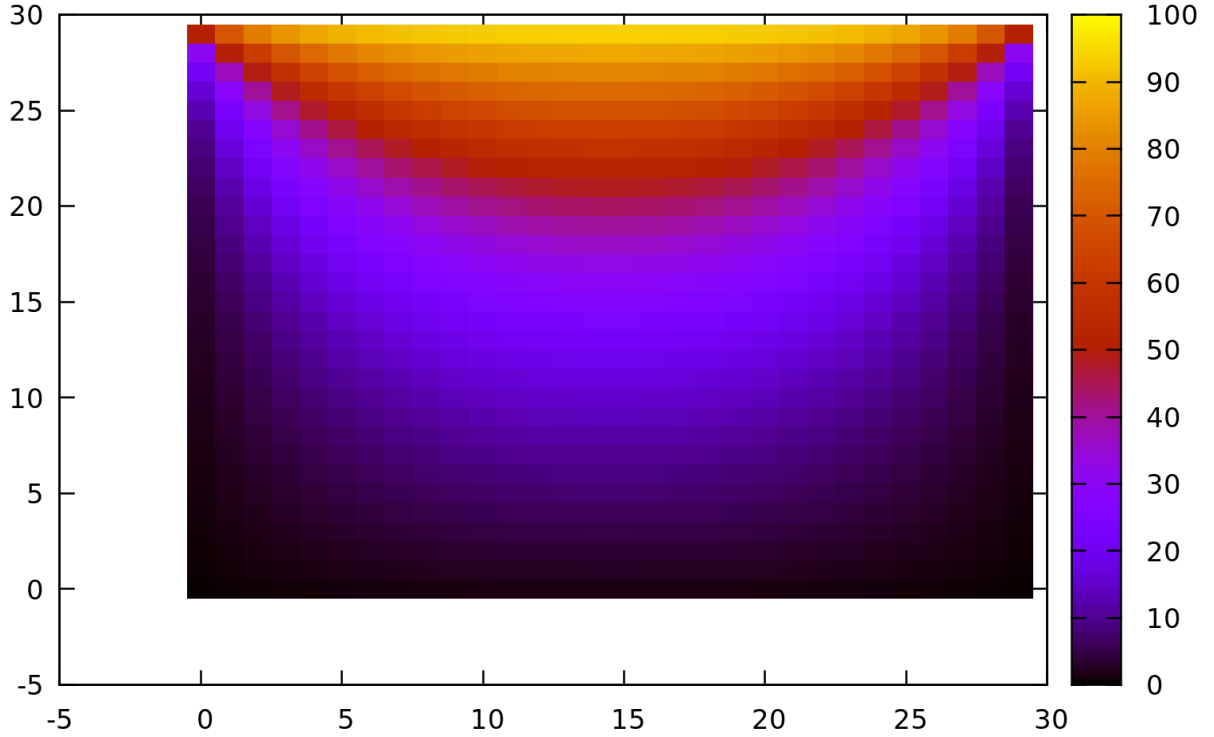
Nous obtenons un fichier texte ressemblant à ceci (ici pour l=3)

```
1    7.1428571428571432    9.8214285714285730    7.1428571428571432
2    18.750000000000000    25.000000000000004    18.750000000000000
3    42.857142857142854    52.678571428571431    42.857142857142854
```

Pour un l plus conséquent (l=31), nous pouvons tracer avec la fonction de gnuplot :

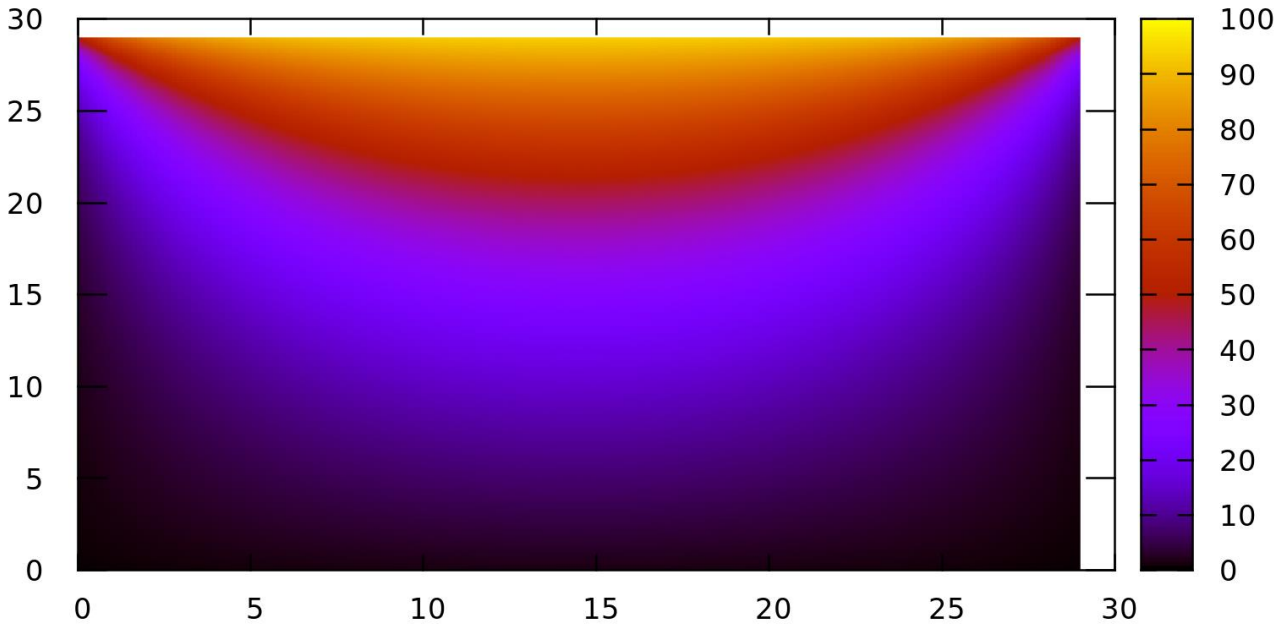
```
plot 'heat_map_data.txt' matrix with image
```

et nous obtenons la représentation suivante :



Ceci concorde bien à la solution du problème physique de diffusion de chaleur.

Avec quelques commandes, on peut lisser e graphique pour obtenir quelque chose de « plus propre » :



On voit bien que le pic de chaleur est bien au centre du bord supérieur de la plaque.

2.2) Équation de Poisson

Dans cette partie, nous nous intéressons à la résolution du problème de Poisson suivant :

$$\begin{cases} -\Delta u(x,y) = 2cos\,(x+y) \\ CL\ Dirichlet : u(x,y) = cos\,(x+y) \end{cases} \quad sur\ D = [0,1] \text{ x } [0,1]$$

Ce problème est très similaire au problème précédent (c'est toujours la diffusion de chaleur dans une plaque), donc nous utiliserons les mêmes méthodes pour sa résolution. Dans cette partie nous n'aborderons plus le temps de calcul étant donné que les deux problèmes sont très similaires.
Ce qui change dans ce problème est les températures des bords ainsi que la manière dont l'énergie thermique se propage.

a) Création des matrices A et b

La matrice A elle ne change pas, seulement le vecteur b change.

Pour ce nouveau système, la matrice ne change pas car l'expression du Laplacien est la même. En revanche, la matrice b change car le Laplacien de u n'a plus la même valeur. En faisant les calculs on arrive donc aux équations :
$4u_{i,j} - u_{i-1,j} - u_{i+1,j} - u_{i,j-1} - u_{i,j+1} = 2.\cos\,(x+y)$
Avec $x = \frac{i}{(l+1)}$ et $y = \frac{j}{(l+1)}$.
Pour construire la matrice b on doit donc d'abord affecter à chaque « équation (i,j) » le terme cos(x+y), en prenant en compte le fait que c'est *b((i-1)l+j)* qui correspond à l'équation (i,j). Ensuite, comme les u ayant i ou/et j qui vaut 0 ou l+1 ne sont pas présent dans le vecteur x, on les fait passer de l'autre côté de l'équation pour les avoir dans le vecteur b. Il faut donc ajouter un terme aux valeurs de b qui correspondent aux points près d'un bord, et deux termes pour les points près d'un coin. Ces termes sont bien sûr la/les valeurs du/des bord(s) à cet endroit-là.

```
    do i=1,l
     bp(i)=dcos(i/(l+1.d0))
     bp(i+(l-1)*l)=dcos(i/(l+1.d0)+1.d0)
     bp((i-1)*l+1)=bp((i-1)*l+1)+dcos(i/(l+1.d0))
     bp((i-1)*l+l)=bp((i-1)*l+l)+dcos(i/(l+1.d0)+1.d0)
     do j=1,l
         bp((j-1)*l+i)=bp((j-1)*l+i)+2*dcos((i+j)/(l+1.d0))/((l+1)*(l+1))
     enddo
enddo
```

b) Erreur

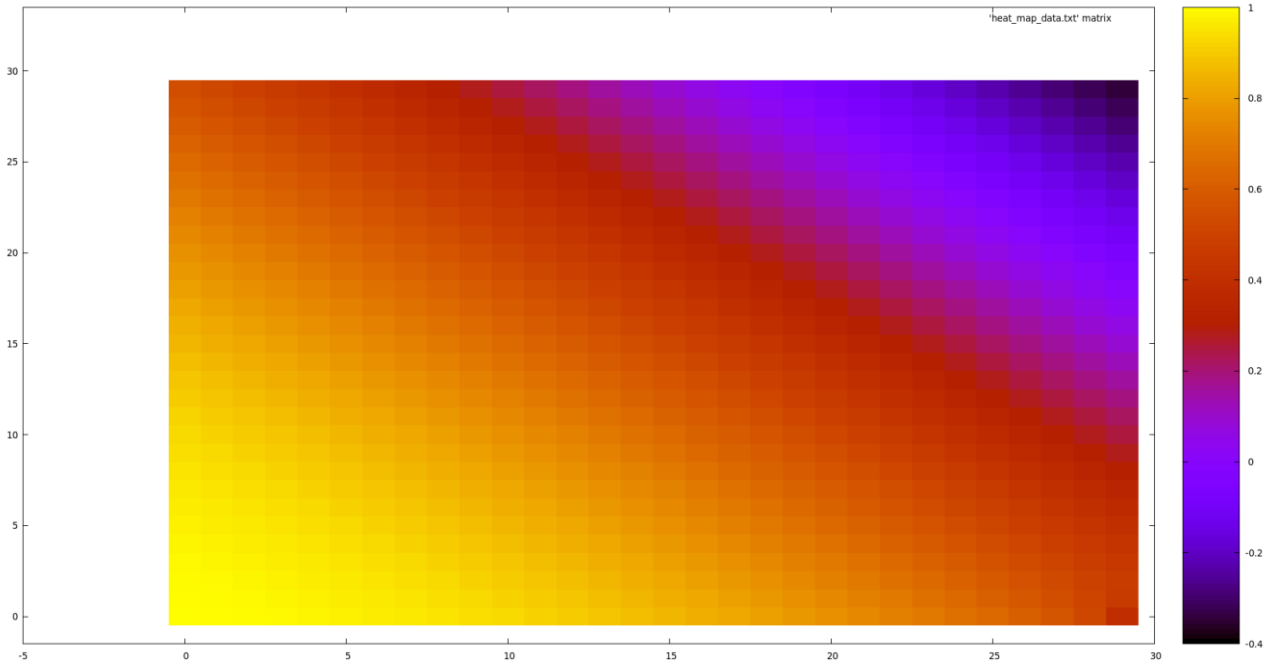
De la même manière que précédemment, nous calculons l'erreur en un point entre la solution exacte (ici beaucoup plus simple : elle est égale à cos(x+y)) et la solution simulée pour différentes discréditation (l=3,10,17,23,30).

```
so=cos(2.d0/(l+1))
print*,so
c On calcul les erreurs:

      DO i=1,5
          xerreur(i)=dlog(abs(Uc(i)-Ur(i)))
      ENDDO

      ypas(1)=-log(4.d0)
      ypas(2)=-log(11.d0)
      ypas(3)=-log(18.d0)
      ypas(4)=-log(24.d0)
      ypas(5)=-log(31.d0)

c On calcul la moyenne des coefs directeurs:

      coefm=0
      do i=1,4
          coefm=coefm+(xerreur(i+1)-xerreur(i))/(ypas(i+1)-ypas(i))
      enddo
      coefm=coefm/4
      print*,coefm
```

Nous obtenons un coefficient directeur moyen de 1,95.

c) Représentation graphique

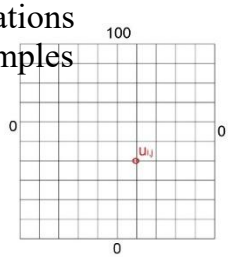
De la même manière que précédemment, nous traçons à l'aide de gnuplot la solution simulée de cette équation et nous obtenons ceci :



Ceci correspond bien à la solution : cos(x+y) sur D=[0,1] x [0,1].
En effet dans le coin en bas à gauche nous avons bien une température qui vaut cos(0), et dans le coin en haut à droite on a bien une température qui vaut cos(2) (en radian).

3) Chaînes de Markov

Dans cette partie, nous allons utiliser une approche probabiliste afin de déterminer les solutions des équations pour le Laplacien 2D.
Pour ce faire, nous utiliserons les chaines de Markov en faisant beaucoup de simulations de Monte Carlo. Nous détaillerons cette méthode au cas par cas dans les exemples suivants.



3.1) Solution ponctuelle du problème de la plaque

La méthode ponctuelle de Monte Carlo consiste effectuer N (très grand) simulations en tous points de la grille. A chaque point de la grille, nous effectuerons N simulations de « marches aléatoires ». Ceci consiste à lancer N fois un « marcheur » en un point de la grille puis de compter le nombre de fois où il sort du côté de la grille valant 100 °C (ce qui ajouter 100 au score). Avec cette méthode on peut donc calculer les valeurs de chaque point de la grille simplement en divisant le score obtenu (nombre de fois sorti de la grille par le haut x 100) par le nombre N de simulations.

Pour générer une marche aléatoire, nous devons utiliser une fonction random nous renvoyant un nombre entre 0 et 1. Pour se faire, nous utilisons la méthode que le professeur nous a donné :
On initialise les variables suivantes :

```
ix=51477
na=16807
nmax=2147483647 ! nombre premier tres grand
```

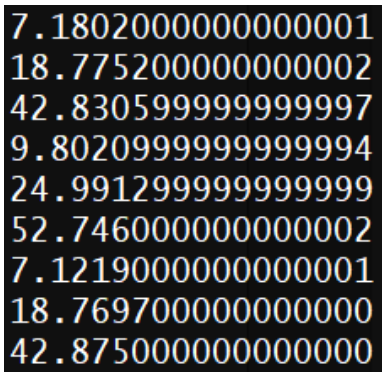
Puis nous tapons les lignes suivantes nous renvoyant un aléatoire a.

```
ix=abs(ix*na)
ix=mod(ix,nmax)
a=(dble(ix)/dble(nmax))
```

Nous pouvons donc créer le programme simulant N marches aléatoires en tous points de la grille. Nous obtenons le code suivant :

```
do i=1,l
  do j=1,l
  cpt=0
  do k=1,iter
     ai=i
     aj=j
     do while((0.LT.ai).and.(ai.LE.l).and.(0.LT.aj).and.(aj.LE.l))
        ix=abs(ix*na)
        ix=mod(ix,nmax)
        a=(dble(ix)/dble(nmax))
        IF ((0.LE.a).and.(a.LE.0.25)) THEN
         ai=ai+1.d0
        ELSE IF ((0.25.LE.a).and.(a.LE.0.5)) THEN
         ai=ai-1.d0
        ELSE IF ((0.5.LE.a).and.(a.LE.0.75)) THEN
         aj=aj-1.d0
        ELSE
         aj=aj+1.d0
        END IF
     enddo
     IF (aj.EQ.(l+1)) THEN
           cpt=cpt+100
     end if
     enddo
     cpt=cpt/iter
     print*,cpt
     E(j,i)=cpt
  enddo
```

Nous effectuons une simulation pour n=3 et 1 000 000 d'itérations pour vérifier que nous obtenons les bonnes valeurs. Nous obtenons les valeurs suivantes :



Ces valeurs sont bien celles que nous sommes censés obtenir. Nous évaluerons plus tard la précision ainsi que le temps de calcul.

3.2) Solution globale du problème de la plaque

La méthode globale de Monte Carlo consiste à ne pas effectuer N simulations en tous points de la grille mais à effectuer seulement N simulations en un point précis de la grille. Ceci est possible car nous allons considérer que quand le « marcheur » passe par un point, la suite du chemin jusqu'au bord est une simulation pour ce point.

Pour sauvegarder les passages, nous avons 3 vecteurs x, x2 et ymt de la taille l*l.
Dans chaque simulation, lorsque le « marcheur » passe sur un point on ajoute 1 aux termes dans les vecteurs x et x2 qui correspondent à ce point. Puis à la fin de la simulation, si le « marcheur » est sorti par le haut on ajoute à chaque valeur dans ymt 100 fois le nombre de passage par le point correspondant indiqué par x. Puis à la fin de la simulation, le vecteur x est remis à zéro. C'est-à-dire qu'on met toutes ses valeurs à 0.
Mais le vecteur x2 n'est jamais réinitialisé, donc à la fin de toutes les simulations, on sait combien de fois en tout on est passé sur chaque point. C'est-à-dire qu'on sait combien de « simulations » on a fait pour chaque point. Il suffit donc de diviser les valeurs dans ymt par les nombres de passage associés pour se retrouver avec les températures en ces points.

Ceci nous donne donc le code suivant : **(1)**

```
do k=1,10000000
    i=int(l/2)
    j=int(l/2)
    do ii=1,l*l
        x(ii)=0
    enddo
    do while((i.gt.0).and.(i.lt.l+1).and.(j.gt.0).and.(j.lt.l+1))

        x((i-1)*l+j)=x((i-1)*l+j)+1
        x2((i-1)*l+j)=x2((i-1)*l+j)+1
        ix=abs(ix*na)
        ix=mod(ix,nmax)
        d=(dble(ix)/dble(nmax))
        if ((0.le.d).and.(d.le.0.25)) then
            i=i-1
        else if ((0.25.le.d).and.(d.le.0.5)) then
            i=i+1
        else if ((0.5.le.d).and.(d.le.0.75)) then
            j=j-1
        else
            j=j+1
        endif
    enddo
    if (i.eq.l+1) then
        do ii=1,l*l
            ymt(ii)=ymt(ii)+100*x(ii)
        enddo
    endif

enddo
```

Nous effectuons une simulation pour n=3 et 1 000 000 d'itérations pour vérifier que nous obtenons les bonnes valeurs. Nous obtenons les valeurs suivantes :

```
7.1911706108046740
18.772138152219142
42.851923336589863
9.8575657852940797
24.865704046039767
52.538196615737846
7.2025836570898685
18.727697912105317
42.633339527968779
```

Ces valeurs sont bien celles que nous sommes censés obtenir. On remarque que le temps de calcul est beaucoup plus rapide mais la précision semble moins bien (ceci est expliqué par le fait que nous avions effectués n*n*N itérations dans la première méthode contre N dans la seconde).

3.3) Solution ponctuelle du problème de Poisson

La méthode ponctuelle de Monte Carlo pour le problème de Poisson est très similaire à celle pour le problème de la plaque.
Nous devons faire quelques changements par rapport à l'algorithme d'avant :

- Changer le score affecté lorsqu'on sort de la grille (car les conditions limites ne sont plus les mêmes) **(1)**

Les conditions limites sont définies par $u(x,y) = \cos(x+y)$, on ajoutera donc la valeur $cos\left(\frac{i}{(l+1)} + \frac{j}{(l+1)}\right) = cos\big((i+j).h\big)$ au score une fois sorti de la grille.

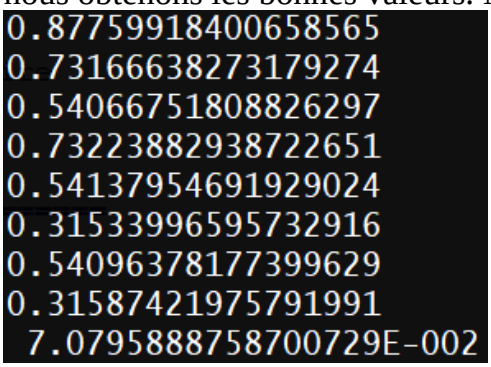
- Ajouter au score un terme à chaque déplacement du marcheur.

En effet, les équations de Poisson nous ont fait apparaitre un terme supplémentaire **(2)**:
$4u_{i,j} - u_{i-1,j} - u_{i+1,j} - u_{i,j-1} - u_{i,j+1} = 2.\cos(x+y)$
Ceci se traduit par l'ajout de $\frac{1}{2.(l+1)^2} cos\left(\frac{i}{(l+1} + \frac{j}{(l+1)}\right) = \frac{h^2}{2} cos\big((i+j).h\big)$ à chaque changement de cellules sur la grille.

Ceci se traduit par le code suivant :

```
do i=1,l
 do j=1,l
      cpt=0
      do k=1,iter
         ai=i
         aj=j
     do while((0.LT.ai).and.(ai.LE.1).and.(0.LT.aj).and.(aj.LE.l))
            ix=abs(ix*na)
            ix=mod(ix,nmax)
            a=(dble(ix)/dble(nmax))
            cpt=cpt+h*h/2*dcos((ai+aj)*h)   (2)
            IF ((0.LE.a).and.(a.LE.0.25)) THEN
               ai=ai+1.d0
            ELSE IF ((0.25.LE.a).and.(a.LE.0.5)) THEN
               ai=ai-1.d0
            ELSE IF ((0.5.LE.a).and.(a.LE.0.75)) THEN
               aj=aj-1.d0
            ELSE
               aj=aj+1.d0
            END IF
     enddo
     cpt=cpt+dcos((ai+aj)*h)   (1)
 enddo
 cpt=cpt/iter
 print*,cpt
 E(j,i)=cpt
 enddo
enddo
```

Nous effectuons une simulation pour n=3 et 1 000 000 d'itérations pour vérifier que nous obtenons les bonnes valeurs. Nous obtenons les valeurs suivantes :



Ces valeurs sont bien celles que nous sommes censés obtenir. Nous évaluerons plus tard la précision ainsi que le temps de calcul.

3.4) Solution globale du problème de Poisson

Pour avoir la résolution du problème de poisson avec la méthode globale, il faut lors de la simulation, comme dans la partie (3.3), ajouter $\frac{1}{2.(l+1)^2} cos\left(\frac{i}{(l+1} + \frac{j}{(l+1)}\right) = \frac{h^2}{2} cos\big((i+j).h\big)$ pour chaque point de la grille que le marcheur parcourt, mais également ajouter $cos\left(\frac{i}{(l+1)} + \frac{j}{(l+1)}\right) = cos\big((i+j).h\big)$ lorsque le « marcheur » sort de la plaque. Mais il faut aussi comme dans la partie (3.2) commencer toutes les simulations au même point, mais considérer qu'à chaque fois que le « marcheur » passe par un point la suite de la simulation est une simulation pour ce point.

Pour cela, on a encore utilisé les vecteurs x, x2 et ymt, qui ont les mêmes fonctions que dans la partie (3.2). Et de même, on ajoute à la fin de chaque simulation $cos\left(\frac{i}{(l+1)} + \frac{j}{(l+1)}\right) = cos\big((i+j).h\big)$ fois le nombre de fois où on est passé en ce point depuis le début de cette simulation à chaque valeur du vecteur ymt.
Sauf que cette fois il faut ajouter quelque chose à chaque nouveau « pas » du « marcheur ». Pour cela, après chaque pas, on ajoute $\frac{1}{2.(l+1)^2} cos\left(\frac{i}{(l+1} + \frac{j}{(l+1)}\right) = \frac{h^2}{2} cos\big((i+j).h\big)$ fois le nombre de fois où on est passé en ce point depuis le début de cette simulation à chaque valeur du vecteur ymt.

Et à la fin il faut donc comme dans la partie (3.2) diviser chaque valeur de ymt par le nombre de fois où x2 indique que la « marcheur » est passé par le point correspondant en prenant en compte toutes les simulations.

Nous traduisons ceci par le code suivant :

```
do k=1,1000000
    i=2
    j=2
    do ii=1,l*l
        x(ii)=0
    enddo
    do while((i.gt.0).and.(i.lt.l+1).and.(j.gt.0).and.(j.lt.l+1))

        x((i-1)*l+j)=x((i-1)*l+j)+1
        x2((i-1)*l+j)=x2((i-1)*l+j)+1
        do ii=1,l*l
            ymt(ii)=ymt(ii)+x(ii)*dcos((i+j)/(l+1.d0))/(2*(l+1)*(l+1))
        enddo
        ix=abs(ix*na)
        ix=mod(ix,nmax)
        d=(dble(ix)/dble(nmax))
        if ((0.le.d).and.(d.le.0.25)) then
            i=i-1
        else if ((0.25.le.d).and.(d.le.0.5)) then
            i=i+1
        else if ((0.5.le.d).and.(d.le.0.75)) then
            j=j-1
        else
            j=j+1
        endif
    enddo
    do ii=1,l*l
        ymt(ii)=ymt(ii)+dcos((i+j)/(l+1.d0))*x(ii)
    enddo

enddo
```

Nous effectuons une simulation pour n=3 et 1 000 000 d'itérations pour vérifier que nous obtenons les bonnes valeurs. Nous obtenons les valeurs suivantes :

```
0.87878143519931329
0.73204212904530397
0.54098399329076852
0.73221545483283590
0.54019231269985968
0.31449828654752493
0.54166834678859577
0.31600810225136788
 7.0179171750367461E-002
```

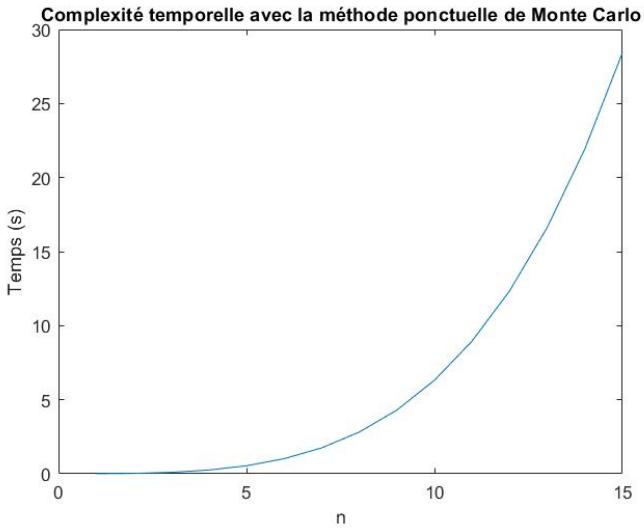
Ceci correspond bien aux valeurs que nous devons obtenir. On remarque que la simulation est bien plus rapide qu'avec la méthode ponctuelle.

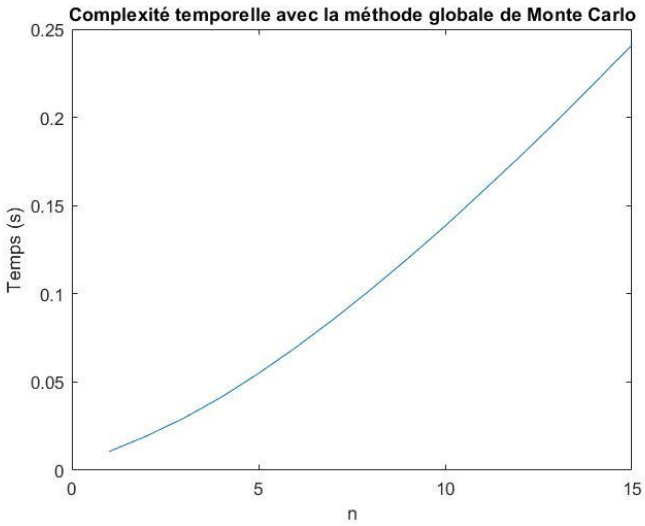
3.1-4) Comparons les deux méthodes.

Pour comparer ces deux méthodes, nous comparerons uniquement la résolution de l'équation de Laplace car les résultats en termes de complexité temporelle et d'erreur sont très similaires.

a) Complexité temporelle

En plaçant des CPU_TIME, on a pu extraire les différents temps de calculs. On a ensuite tracé les courbes sur Matlab pour des simulations à N=100 000 itérations.

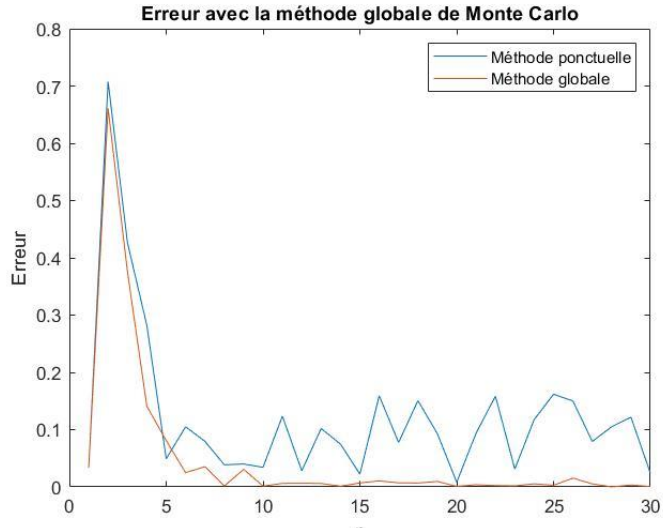




Sans grandes surprises, on remarque que la méthode globale est bien plus rapide que la méthode ponctuelle. On peut expliquer ceci par le fait que la méthode ponctuelle effectue ($n^2.N$) marches aléatoires (ce qui explique la croissance de manière exponentielle) tandis que la méthode globale n'en effectue que (N) (ce qui explique la forme linéaire).

b) Erreur

Pour comparer les deux méthodes de manière « égale », nous avons choisis de faire en sorte que les deux méthodes effectuent exactement le même nombre de marches aléatoires soit N=1 000 000. Comme la méthode ponctuelle effectue n^2N marches, on a donc initialisé le $N=N/n^2$. On a de même tracé le graphique sur matlab et nous avons obtenu ceci :



On remarque qu'à partir de n=5, la méthode globale devient nettement plus précise. L'ordre de grandeur pour la méthode ponctuelle est de $\mathbf{10^{-1}}$ tandis que pour la méthode globale, il est de l'ordre de $\mathbf{10^{-3}}$ soit environ 100 fois plus précis.

Cette parie étant facultative, nous expliquons que brièvement la méthode.

3.5) Séquentiel Monte Carlo pour la plaque

La méthode séquentielle consiste à calculer y par la méthode Monté Carlo avec $Ay = Ax - A\tilde{x} = b - A\tilde{x} = r$. C'est donc un mixte entre la méthode de Monté Carlo et la méthode matriciel. Ceci permet d'obtenir rapidement un résidus très petit ainsi d'obtenir rapidement une bonne précision.

En exécutant le programme donné par le professeur, on remarque qu'on utilise une grille où les bords en 0 sont atteignables. Ceci permet donc de créer la grille ainsi que ses conditions limites.

En compilant, on peut voir qu'on obtient bien les bonnes solutions, ceci d'une manière plus optimisée que les deux méthodes précédentes.

3.6) Séquentiel Monte Carlo pour le problème de Poisson

Pour la méthode Poisson, nous avons quelques changements à effectuer afin d'obtenir les bons résultats :

- Nous changeons les conditions aux limites de la grille :

```
do i=0,n2
     r(i,n2)=dcos((i+n2)*h)
     r(n2,i)=dcos((i+n2)*h)
     r(i,0)=dcos((i)*h)
     r(0,i)=dcos((i)*h)
enddo
```

- Nous changeons l'équation qui régit les valeurs obtenues en tous points de la grille.

```
                do j=1,n1
              f(i,j)=(-r(i,j-1)+4*r(i,j)-r(i,j+1)-r(i-1,j)-r(i+1,j)
&-2*h*h*dcos((i+j)*h))
```

En compilant, on peut voir qu'on obtient bien les bonnes solutions.

Conclusion générale :

Le but de ce TP était de résoudre le problème physique de la diffusion de la chaleur dans une plaque selon deux modèles (Laplace et poisson) avec deux méthodes différentes (les différences finies et répétition de simulations de Monte Carlo).
Notre première conclusion est que le problème physique est bien plus complexe lorsque l'on choisit le modèle de poisson, car le laplacien étant non nul cela rend les calculs plus complexes.
Notre deuxième conclusion est que la méthode des différences finies requiert un code plus long, surtout car elle nécessite un algorithme qui résout une équation matricielle (ici la méthode du gradient conjugué), alors que la méthode probabiliste est un code relativement rapide à faire.
Notre troisième conclusion est que la méthode des différence finies semble plus rapide que les méthodes de Monté Carlo ponctuelle / globale. En outre, nous avons eu la chance de pouvoir essayer la méthode « Sequential Monté Carlo » qui est un mixte entre les deux méthodes. Elle est aussi rapide et efficace que la méthode des différences finis. L'avantage de celle-ci est qu'une fois paramétré, il n'y a qu'à changer l'équation régissant le système ainsi que les conditions de bords pour obtenir les solutions.