© *J.-B. A. K. <jean-baptiste.apoung@math.u-psud.fr>*

Solution du devoir surveillé : 26 mars 2015

Thème - 1 *Problème*

On considère l'équation aux dérivées partielles suivante :

$$\left\{\begin{array}{rll} \dfrac{\partial u(t,x)}{\partial t} - \dfrac{\partial^2 u(t,x)}{\partial x^2} & = f(t,x) & \forall (t,x) \in]0,T[\times]0,1[\\ u(t,0) & = 0 & \forall t \in [0,T] \\ u(t,1) & = 0 & \forall t \in [0,T] \\ u(0,x) & = u_0(x) & \text{sur } [0,1] \end{array}\right. \tag{1}$$

où, f est une fonction continue de ses arguments et u_0 une fonction de classe C^2 avec $u_0(0) = 0$ et $u_0(1) = 0$.

Nous effectuons une discrétisation de ce problème par une méthode des différences finies en espace et un θ-schéma en temps. La procédure est la suivante (voir cours).

Pour la semi-discrétisation spatiale, on se donne un entier $N \in \mathbb{N}^*$ et on définit une subdivision uniforme de $[0,1]$ de pas $h = \frac{1}{N+1}$ appelée **grille** ou **maillage**. Ceci génère les points $0 = x_0 < \ldots < x_j < \ldots < x_{N+1} = 1$ définis par $x_j = j \times h, j = 0 \ldots, N+1$.

Pour tout $t \in [0,T]$, on désigne par $U(t)$ le vecteur de $\mathbb{R}^n$ dont la j-ème composante est une approximation de $u(t,x_j), \quad \forall j = 0 \ldots, N$. On obtient le système d'équations différentielles ordinaires suivant :

$$\left\{\begin{array}{rll} \dfrac{dU(t)}{dt} + A_h U(t) & = F(t) & \text{sur }]0,T[\\ U(0) & = U_0 & \end{array}\right. \tag{2}$$

où $A_h = \text{tridiag}(-\frac{1}{h^2}, \frac{2}{h^2}, -\frac{1}{h^2})$ est la matrice du laplacien en dimension 1, de taille N sur un maillage uniforme de pas $h = \frac{1}{N+1}$. On a posé $F(t) = [f(t,x_1), \ldots, f(t,x_N)]^T \ \forall t \in]0,T[$ et $U_0 = [u_0(x_1), \ldots, u_0(x_N)]^T$.

Pour la semi-discrétisation en temps, on se donne un entier $K \in \mathbb{N}^*$ et on considère une subdivision uniforme de $[0,T]$ de pas $\Delta t = \frac{T}{K}$, qui génère les instants $t_k = k\Delta t, k = 0, \ldots, K$. On désigne pour tout $k = 0 \ldots, K$, par U^k une valeur approchée de la solution $U(t_k)$ de (2) fournie par le θ-schéma suivant ($\theta \in [0,1]$) :

$$\left\{\begin{array}{rll} \dfrac{U^{k+1} - U^k}{\Delta t} + (1-\theta) A_h U^k + \theta A_h U^{k+1} & = (1-\theta) F(t_k) + \theta F(t_{k+1}), \quad k = 0, \ldots, K-1 \\ U^0 & = U_0 \end{array}\right. \tag{3}$$

qui se met encore sous la forme

$$\left\{\begin{array}{rll} B_{\Delta t,h,\theta}\, U^{k+1} & = B_{\Delta t,h,\theta-1}\, U^k + R^k_{\theta,h}, \quad k = 0, \ldots, K-1 \\ U^0 & = U_0 \end{array}\right. \tag{4}$$

où on a posé $R^k_{\theta,h} = (1-\theta)F(t_k) + \theta F(t_{k+1}) \quad \forall k = 0, \ldots, K-1$. $B_{\Delta t,h,\beta} = \left(\dfrac{1}{\Delta t} I_N + \beta A_h\right) \ \beta \in \mathbb{R}$

Q-1 : Construction des matrices et vecteurs.

Q-1-1 : Fonction générant les matrices

Listing 1 – Construction de la matrice $B_{\Delta t,h,\beta}$

```
gsl_matrix *
gen_matrice_B (double dt, int N, double beta)
{
  gsl_matrix *B = matrice_laplacien1d (N, 0.0, 1.0);
  gsl_matrix *IN = gsl_matrix_alloc (N, N);
  gsl_matrix_set_identity (IN);
  gsl_matrix_scale (IN, 1. / dt);
  gsl_matrix_scale (B, beta);
  gsl_matrix_add (B, IN);
  gsl_matrix_free (IN);
  return B;
}
```

Q-1-2 : Fonction qui calcule le terme source

Listing 2 – Construction de $R^k_{\theta,h}$

```
//expression de la fonciton f
double f(double t, double x)
{
  return 1.;
}
//foncton demandee:
//Cette fonction est appelee a chaque iteration en temps. Pour des raisons economiques,
//il est preferable qu'elle retourne son resultat en modifiant un de ses arguments, R ici.
// Ainsi, on pourra creer ce vecteur avant d'entrer dans les iterations en temps.
void
gen_vecteur_R (gsl_vector* R,  double dt, int N,
               double theta, int k, const gsl_vector* x)
{
  int i;
  double d;
  double tk = k * dt;
  for (i = 0; i < N; i++)
    {
      d =
        (1 - theta) * f (tk, gsl_vector_get (x, i + 1))
        +
        theta * f (tk + dt, gsl_vector_get (x, i + 1));
      gsl_vector_set (R, i, d);
    }
}
```

Q-2 : Constructibilité du schéma (4).

Q-2-1 : Pour que le schéma (4) soit constructible, il est nécessaire que $B_{\Delta t,h,\theta}$ soit inversible. C'est le cas par exemple si cette matrice est symétrique définie positive.

Q-2-2 : Fonction qui vérifie si une matrice est symétrique définie positive.

Listing 3 – Fonction qui teste si une matrice est symétrique définie positive (S.D.P)

```
 double est_sym_def_pos (const gsl_matrix* A)
{
  gsl_matrix *C = gsl_matrix_alloc (A->size2, A->size1);
  gsl_matrix_transpose_memcpy (C,A);  // C  =  A^T
  gsl_matrix_sub (C, A);              // C  =  C -  A
  int is_null = gsl_matrix_isnull (C);
  if (is_null != 1)
    return 0;   // la matrice n'est pas symetrique
  double lambda[2];
 // On ne veut pas modifier A , on continue avec C
  gsl_matrix_memcpy (C,A);
  valeurs_propres_extremes (C, lambda);
  gsl_matrix_free (C);
  return lambda[1] > 0;
}
```

Q-2-3 : Vérifications : On pose $N = 100, \Delta t = 10^{-1}$.

Listing 4 – Fonction validant presque sûrement le caractère S.D.P. de $B_{\Delta t,h,\theta}$ $\forall \theta \in [0,1]$

```
void
test_est_sym_def_pos ()
{
  int N = 100, k;
  double dt = 1e-1;
  double theta;
  int nb_tirages = 10000;
  gsl_vector *var = gsl_vector_alloc (nb_tirages);

  gsl_rng *r;
  gsl_rng_env_setup ();
  r = gsl_rng_alloc (gsl_rng_mt19937);

  for (k = 0; k < nb_tirages; ++k)
    {
      theta = gsl_rng_uniform (r);
      //Optimisation possible ici car la taille de B ne change pas
      gsl_matrix *B = gen_matrice_B (dt, N, theta);
      gsl_vector_set (var, k, (double) est_sym_def_pos (B));
      gsl_matrix_free (B);
    }
  // calcule de la moyenne
  fprintf (stdout,
           "Nombre de Tirages: %d. Pour theta dans [0,1], la probalilite pour B d'etre S.D.P est %f \n",
           nb_tirages, (var) / (double) nb_tirages);
  gsl_rng_free (r);
}
```

Le schéma peut alors s'écrire sous la forme :

$$\left\{ \begin{array}{rl} U^{k+1} & = B^{-1}_{\Delta t,h,\theta} B_{\Delta t,h,\theta-1}\, U^k + B^{-1}_{\Delta t,h,\theta} R^k_{\theta,h}, \quad k = 0, \ldots, K-1 \\ U^0 & = U_0 \end{array} \right. \tag{5}$$

Q-3 : Stabilité du schéma

Q-3-1 : Le schéma est stable dans la norme $\| \ \|_h^2$ dès que $\|B^{-1}_{\Delta t,h,\theta} B_{\Delta t,h,\theta-1}\|_2 < 1$

Q-3-2 : Fonction qui teste si une matrice est normale : A est normale si $A^T\, A = A\, A^T$

Elle fait appel à une fonction **`matrice_norme_inf (const gsl_matrix* )`** qui est fournie dans le Listing12. Cette fonction nous permet de prendre en compte l'arithmétique non exacte dans la comparaison de $A^T\, A$ et $A\, A^T$

Listing 5 – Fonction qui teste si une matrice est normale

```
1  double
2  est_normale (const gsl_matrix * A)
3  {
4    double precision_machine = 1.0e-16;   // machine 32 bits
5    double c_min, c_max;
6    double normeinfA = matrice_norme_inf (A);
7    double epsil = normeinfA * normeinfA * A->size1 * precision_machine/(1. - A->size1 *  precision_machine);
8    gsl_matrix *C1 = gsl_matrix_alloc (A->size1, A->size1);
9    gsl_matrix *C2 = gsl_matrix_alloc (A->size2, A->size2);
10   gsl_blas_dgemm (CblasNoTrans, CblasTrans, 1., A, A, 0, C1);   // C1 = A*A'
11   gsl_blas_dgemm (CblasTrans, CblasNoTrans, -1., A, A, 0, C2);  //C2 = -A'*A
12   gsl_matrix_add (C1, C2);      //C1 = C1 - C2;
13   // Cette portion de code peut etre supprimee
14   int is_null = gsl_matrix_isnull (C1); // est problematique
15   if (is_null == 1)
16     {
17       gsl_matrix_free (C1);
18       gsl_matrix_free (C2);
19       return 1.;
20     }
21   //On ameliore la verification.
22   //car gsl_matrix_isnull suppose une arithmetique exacte
23   gsl_matrix_minmax (C1, &c_min, &c_max);
24   gsl_matrix_free (C1);
25   gsl_matrix_free (C2);
26   return fmax (fabs (c_min), fabs (c_max)) < epsil;
27 }
```

Q-3-3 : Vérifications : On pose $N = 100, \Delta t = 10^{-2}$. On procède comme à la question précédente.

Listing 6 – Fonction qui confirme presque sûrement que $\forall \theta \in [0,1],\ B^{-1}_{\Delta t,h,\theta} \times B_{\Delta t,h,\theta-1}$ est normale

```
void
test_est_normale ()
{
  int N = 100, k;
  double dt = 1e-1;
  double theta;
  int nb_tirages = 1000;
  gsl_vector *var = gsl_vector_alloc (nb_tirages);
  gsl_rng *r;
  gsl_rng_env_setup ();
  r = gsl_rng_alloc (gsl_rng_mt19937);

  for (k = 0; k < nb_tirages; ++k)
    {
      theta = gsl_rng_uniform (r);
      //Optimisation possible ici car la taille des matrices ne change pas
      gsl_matrix *B1 = gen_matrice_B (dt, N, theta);
      gsl_matrix *B2 = gen_matrice_B (dt, N, -1 + theta);
      gsl_matrix *B1inv = gen_matrix_inverse (B1);      //B1inv = B1^-1
      gsl_matrix *C = gsl_matrix_alloc (B1->size1, B1->size2);
      gsl_blas_dgemm (CblasNoTrans, CblasNoTrans, 1., B1inv, B2, 0, C); // C = B1^-1 * B2

      gsl_vector_set (var, k, (double) est_normale (C));
      gsl_matrix_free (B1);
      gsl_matrix_free (B2);
      gsl_matrix_free (B1inv);
      gsl_matrix_free (C);
    }

  // calcule de la moyenne
  fprintf (stdout,
           "Nombre de Tirages: %d. Pour theta dans [0,1], la probalibilite pour B d'etre normale est %f \n",
           nb_tirages, gsl_blas_dasum (var) / (double) nb_tirages);
  gsl_rng_free (r);
}
```

Q-3-4 : Fonction qui retourne 1 si le schéma est stable pour la norme $\| \ \|_h$ et 0 sinon.

On va générer la matrice $B^{-1}_{\Delta t,h,\theta}\ B_{\Delta t,h,\theta-1}$. On sait qu'elle est normale d'après la question précédente. Par conséquent, $\|B^{-1}_{\Delta t,h,\theta} \times B_{\Delta t,h,\theta-1}\|_2 = \rho(B^{-1}_{\Delta t,h,\theta} \times B_{\Delta t,h,\theta-1})$. Il suffit donc de déterminer ses valeurs propres (extrêmes) : $\lambda_{\min}, \lambda_{\max}$ car dans ce cas $\rho(B^{-1}_{\Delta t,h,\theta} \times B_{\Delta t,h,\theta-1}) = \max(|\lambda_{\min}|, |\lambda_{\max}|)$.

Listing 7 – Fonction qui teste si le θ-schema, pour N (donc h), Δt, θ donnés est stable

```
double
schema_est_stable (double dt, int N, double theta)
{
  gsl_matrix *B1 = gen_matrice_B (dt, N, theta);
  gsl_matrix *B2 = gen_matrice_B (dt, N, -1 + theta);
  gsl_matrix *B1inv = gen_matrix_inverse (B1);        //B1inv = B1^-1

  gsl_matrix *C = gsl_matrix_alloc (B1->size1, B1->size2);
  gsl_blas_dgemm (CblasNoTrans, CblasNoTrans, 1., B1inv, B2, 0, C); // C = B1^-1 * B2

  double lambda[2];
  valeurs_propres_extremes (C, lambda);
  printf ("lambda min = %f    lambda max = %f  \n", lambda[0], lambda[1]);
  gsl_matrix_free (B1);
  gsl_matrix_free (B2);
  gsl_matrix_free (B1inv);
  gsl_matrix_free (C);
  return fmax(fabs(lambda[0]), fabs(lambda[1])) < 1;
}
```

Q-3-5 : Testons la fonction pour les valeurs suivantes :

— $N = 100, \Delta t = 10^{-3}, \theta = 0.75$

— $N = 100, \Delta t = 10^{-3}, \theta = 0.4$

— $N = 100, \Delta t = 2\ 10^{-}4, \theta = 0.4$

On fournit pour cela le Listing8

Listing 8 – Fonction qui teste si le θ-schema, pour N (donc h), Δt, θ donnés est stable

```
void
test_stabilite ()
{
  int N = 100;
  double dt, theta;
  dt = 1e-3;
  theta = 0.7;
  printf ("\n\n");
  printf ("schema_est_stable(dt = %e, N = %d,  theta = %1.1f) = %1.0f \n", dt,  N, theta, schema_est_stable ↩
       (dt, N, theta));
  printf ("Raison:  theta >= 0.5\n");

  dt = 1e-3;
  theta = 0.4;
  printf ("\n\n");
  printf ("schema_est_stable(dt = %e, N = %d,  theta = %1.1f) = %1.0f \n", dt,  N, theta, schema_est_stable ↩
       (dt, N, theta));
  printf ("Raison:  Comme theta < 0.5 Il faut dt/h^2 < 1/(2 (1-2theta))  Or , dt/h^2 = %f  et 1/(2 (1-2theta↩
       ) = %f\n", dt * (N + 1) * (N + 1), 1. / (2 - 4 * theta));

  dt = 2e-4;
  theta = 0.4;
  printf ("\n\n");
  printf ("schema_est_stable(dt = %e, N = %d,  theta = %1.1f) = %1.0f \n", dt,  N, theta, schema_est_stable ↩
       (dt, N, theta));
  printf ("Raison:  Comme theta < 0.5 Il faut dt/h^2 < 1/(2 (1-2theta))  Or , dt/h^2 = %f  et 1/(2 (1-2theta↩
       ) = %f\n", dt * (N + 1) * (N + 1), 1. / (2 - 4 * theta));
}
```

Q-4 : Déterminons la solution effective à l'instant final et comparons la solution pour deux valeurs de θ.

On peut se simplifier la tâche en choisissant convenablement la solution exacte. En effet, pour $u(t,x) = \sin(\pi x)e^{-\pi^2 t}$ on a $u_0(x) = \sin(\pi x)$ et $f(t,x) = 0$.

Q-4-1 : Il n'est pas raisonnable de calculer explicitement $B^{-1}_{\Delta t,N,\theta}$. Il suffit simplement de stocker la factorisation LU de $B_{\Delta t,N,\theta}$. En effet l'évaluation du produit $B^{-1}_{\Delta t,N,\theta}\ v$ revient à faire une descente et une remontée avec la factorisation LU de $B_{\Delta t,N,\theta}$. Ce qui peut être économique dans la plupart des cas.
Le Listing 9 ci-dessous détermine la solution à l'instant final, pour $N, \Delta t, \theta$ et T donnés.

Listing 9 – Fonction qui détermine U^K par le θ-schema pour N (donc h), Δt, θ et T donnés

```
void
solution_finale_schema (double dt, int N, double theta, double T, const char *fichier)
{
  double xj, h;
  double t0;
  int j, k;
  double pi = 4 * atan (1.);

  gsl_matrix *B1 = gen_matrice_B (dt, N, theta);
  gsl_matrix *B2 = gen_matrice_B (dt, N, -1 + theta);
  gsl_vector *v = gsl_vector_alloc (N);
  gsl_vector *u = gsl_vector_alloc (N);
  //Maillage  ou grille
  gsl_vector *x = gsl_vector_alloc (N + 2);
  h = 1. / (N + 1);
  for (j = 0; j < N + 2; ++j)
    gsl_vector_set (x, j, j * h);
  //initialisation  construction de u0
  for (j = 0; j < N; ++j)
    {
      xj = gsl_vector_get (x, j + 1);
      gsl_vector_set (u, j, sin (pi * xj));
    }
  // Decomposition LU   decomLU(B1);
  int s;
  gsl_permutation *p = gsl_permutation_alloc (N);
  gsl_linalg_LU_decomp (B1, p, &s);
  // Boucle sur le temps
  int K = (int) (T / dt);
  double t = 0.;
  for (k = 0; k < K; ++k)
    {
      gsl_blas_dgemv (CblasNoTrans, 1., B2, u, 0., v);  // v = B2 * u
      // Si on n'avait pas f=0, il aurait fallu ajouter R a v
      // ie v = v + R  avec R = (1 - theta) * F(tk,u) + theta * F(tk+dt))
      // obtenu en appelant gen_vecteur_R
      gsl_linalg_LU_solve (B1, p, v, u);        // u = B1^-1 * v
```

```
      t = t + dt;
    }
  //printf(" t - T = %7.9e\n",t - T);
  gsl_permutation_free (p);
  // Ecriture de la solution dans un fichier
  FILE *os = fopen (fichier, "w");
  fprintf (os, "%e   %e  %e\n", gsl_vector_get (x, 0), 0.0, 0.0);
  for (j = 0; j < N; j++)
    {
      fprintf (os, "%e   %e  %e\n", gsl_vector_get (x, j + 1), gsl_vector_get (u, j),
               sin (pi * gsl_vector_get (x, j + 1)) * exp (-pi * pi * t));
    }
  fprintf (os, "%e   %e  %e\n", gsl_vector_get (x, N + 1), 0.0, 0.0);
  fclose (os);
  gsl_vector_free (u);
  gsl_vector_free (v);
  gsl_matrix_free (B1);
  gsl_matrix_free (B2);
}
```

Q-4-2 : Ecriture dans un fichier de la solution pour $N = 100$, $\Delta t = 10^{-1}$, $\theta = 1.$, $T = 1$. Voir Listing10
Q-4-3 : Ecriture dans un fichier de la solution pour $N = 100$, $\Delta t = 10^{-1}$, $\theta = 0.5$, $T = 1$. Voir Listing10
Q-4-4 : Meilleure solution attendue : celle pour $\theta = 0.5$ voir Figure 1

Listing 10 – Test de la génération de la solution finale

```
void
test_solution_finale ()
{
  int N = 100;
  double dt = 1e-2, theta;
  double T = 0.1;
  theta = 1.;
  solution_finale_schema (dt, N, theta, T, "sol1.txt");
  theta = 0.5;
  solution_finale_schema (dt, N, theta, T, "sol2.txt");
}
```

Listing 11 – La fonction principale

```
int
main (int argc, char **argv)
{
  // test_est_sym_def_pos ();
  // test_est_normale ();
  // test_stabilite ();
  test_solution_finale ();
  return 0;
}
```

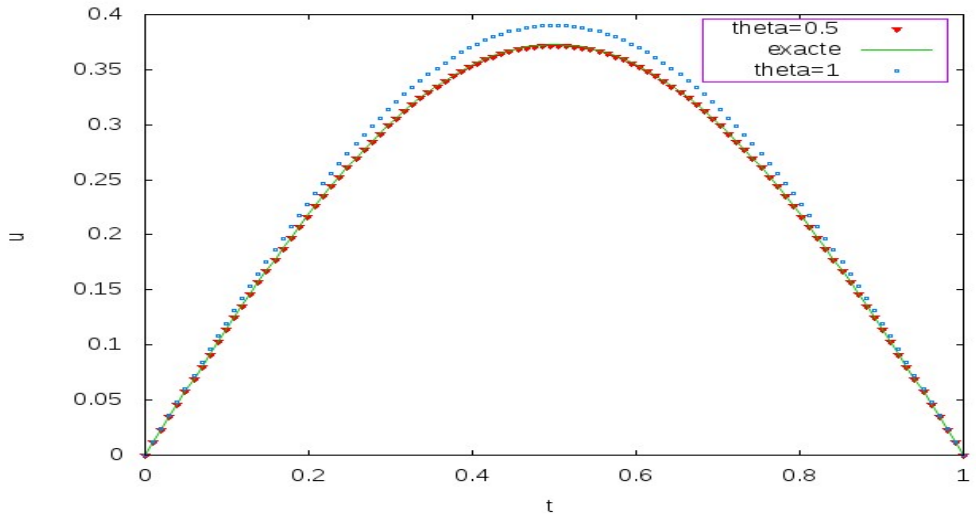

FIGURE 1 – Solutions exacte et approchées à $T = 0.1$, pour $N = 100$, $\Delta t = 10^{-2}$, $\theta = 1$ et $\theta = 0.5$

Thème - 2 *Utilitaires*

On fournit ici les fonctions utilisées dans la définition des fonctions précédentes.

Listing 12 – Quelques fonctions fournies

```
/* Il faut ajouter :
#include <gsl/gsl_eigen.h>
#include <gsl/gsl_blas.h>
*/
void valeurs_propres_extremes(gsl_matrix * A, double lambda[2])
{
  int n = A->size1;  int m = A->size2;
  assert(n == m);
  gsl_vector *eval = gsl_vector_alloc (n);
  gsl_eigen_symm_workspace * w = gsl_eigen_symm_alloc (n);
  gsl_eigen_symm (A, eval, w);
  gsl_vector_minmax (eval, &lambda[0], &lambda[1]);
  gsl_eigen_symm_free(w);
  gsl_vector_free(eval);
}
//Calcul l'inverse d'une matrice inversible et le retourne
gsl_matrix* gen_matrix_inverse(gsl_matrix*  A)
 {
  int n = A->size1;   int m = A->size2;  assert(n == m);
  int s;
  gsl_permutation * p = gsl_permutation_alloc (A->size1);
  gsl_matrix* inverse = gsl_matrix_alloc (A->size1, A->size2);
  gsl_matrix* lu = gsl_matrix_alloc (A->size1, A->size2);
  gsl_matrix_memcpy (lu,A);
  gsl_linalg_LU_decomp (lu, p, &s);
  gsl_linalg_LU_invert (lu, p, inverse);
  gsl_permutation_free (p);
  gsl_matrix_free (lu);
  return inverse;
  }
// Generation de la matrice du laplacien 1D
gsl_matrix *
matrice_laplacien1d (int n, double a, double b)
{
  gsl_matrix *A = gsl_matrix_alloc (n, n);

  double h = (b - a) / (n + 1);
  double ih = 1. / (h * h);

  int i;
  for (i = 1; i < n - 1; ++i)
    {
      gsl_matrix_set (A, i, i, 2. * ih);
      gsl_matrix_set (A, i, i + 1, -ih);
      gsl_matrix_set (A, i, i - 1, -ih);
    }
  gsl_matrix_set (A, 0, 0, 2 * ih);
  gsl_matrix_set (A, 0, 1, -ih);
  gsl_matrix_set (A, n - 1, n - 1, 2. * ih);
  gsl_matrix_set (A, n - 1, n - 2, -ih);
  return A;
}

double
matrice_norme_inf (const gsl_matrix * A)
{
  int i, j, n, m;
  double s, res = 0.;
  n = A->size1;
  m = A->size2;

  for (j = 0; j < m; ++j)
    {
      s = 0.;
      for (i = 0; i < n; i++)
        {
          s += fabs (gsl_matrix_get (A, i, j));
        }
      res = fmax (res, s);
    }
  return res;
}
```