Programmation pour le calcul scientifique
Année : 2019-2020
Formation : L3 Ingénierie Mathématique

TP2 : Rappels, pointeurs

Les fichiers de ce TP sont disponibles sur Moodle dans une archive nommée `tp2.tar.gz`. Vous pouvez décompresser l'archive avec la commande

```
tar xzf tp2.tar.gz
```

1 Pointeurs

1. Expliquez l'affichage du code suivant :

```
#include <iostream>
int main() {
  int m = 23;
  int* p;
  std::cout << "p: " << p << std::endl;
  p = &m;
  std::cout << "p: " << p << std::endl;
  return 0;
}
```

   Comment accèder à `m` en n'utilisant que `p` ?

2. Que fait le code suivant ?

```
#include <iostream>

int main() {

  int size  = 50;
  double* x = new double[size];
  double* y = new double[size];

  for(int i=0;i<size;i++) {
    x[i] = i+1.;
    y[i] = 1.-i;
  }

  std::cout << x[10]   << std::endl;
  std::cout << *(x+10) << std::endl;

  delete[] x;
  delete[] y;

  return 0;
}
```

3. Nous insérons ceci juste avant les instructions `delete` pour copier un tableau dans l'autre :

```
x = y;

for(int i=0; i<size; i++)
  std::cout << x[i] << " ";
std::cout << std::endl;
```

Quel est le problème ? Comment le résoudre ?

2 Passage d'arguments aux fonctions

Pourquoi le code suivant ne donne pas le résultat attendu ? Quelle correction proposez-vous ?

normalizeCoord.cpp

```cpp
#include <iostream>
#include <cmath>

struct point {
  double x;
  double y;
}

inline double norm2(point p) {
  return sqrt(p.x*p.x + p.y*p.y)
}

inline void normalize(point p) {
  double norm = norm2(p);
  p.x /= norm;
  p.y /= norm;
}

int main() {
  point M;
  M.x = -3;
  M.y = 0;

  normalize(M);

  std::cout << "Normalized coordinates: " << M.x << " " << M.y << std::endl;

  return 0;
}
```

3 Pointeurs et grosses structures de données

1. Créez un programme qui échange 10001 fois de suite, terme à terme, le contenu de deux `vector<double>` de taille 10000. Chronomètrez l'exécution de ce programme avec la commande

   ```
   $> time ./myProgram
   ```

   (Vérifiez tout de même que le contenu est bien échangé à la fin !)
2. Dans un autre programme, échangez les vecteurs à l'aide de pointeurs. Chronomètrez.

4 Solveur d'EDO multi-pas

1. Téléchargez le squelette de programme suivant :

multistepODE.cpp

```cpp
#include <iostream>
#include <fstream>
#include <vector>
#include <cmath>
#include <algorithm>

using rvec = std::vector<double>;

// ...

/* Adams Bashforth numerical step which computes
 * y^{n+1} = y^n + 3/2*dt*f(t^n,y^n) - 1/2* dt*f(t^{n-1},y^{n-1})
 * in order to solve y' = f(t,y)
 * Vectors yNP1, yN and yNM1 are supposed to have the same size.
 */
void oneStepAdamsBashforth2(rvec& yNP1, rvec& yN, rvec& yNM1, double (*f) (double, double),
    double dt, double tN) {

  for (int i=0; i<yNP1.size(); i++)
    yNP1[i] = yN[i] + dt*1.5*f(tN,yN[i]) - 0.5*dt*f(tN-dt,yNM1[i]);

};

/* Forward Euler step
 * y^{n+1} = y^n + dt*f(t^n,y^n)
 * in order to solve y' = f(t,y)
 * Vectors yNP1 and yN are supposed to have the same size.
 */
void oneStepEuler(rvec& yNP1, rvec& yN, double (*f) (double, double), double dt, double tN) {

  for (int i=0; i<yNP1.size(); i++)
    yNP1[i] = yN[i] + dt*f(tN,yN[i]);

};

int main() {

  int N = 1000;
  // Data vector at different step
  rvec y0(N),y1(N),y2(N);
  // Points
  rvec x(N);

  // Problem times
  double dt     = 0.01;
  double t      = 0.;
  double tFinal = 1.;

  // x_i and Initial condition
  for (int i=0; i<N; i++) {
    x [i] = i/(double)(N-1);
    y0[i] = initCond(x[i]);
  }

  // First step is performed with Euler scheme
  oneStepEuler(y1,y0,&fTest,dt,t);
  t += dt;

  // These scalars will be used in the loop to compute the L2 norms
  // of the solution and the difference with approx at all times
```

```
60  double normSol(0.e0),normDiff(0.e0);
61
62  // ...
63
64  // Now we use Adams-Bashforth method for all remaining steps
65  while (t<tFinal) {
66
67    // ...
68
69    t += dt;
70
71    // Add contribution to norms for error computation
72    for (int i=0; i<N; i++) {
73      double sol = exactSol(t,initCond(x[i]));
74      normSol  += pow(sol,2);
75      normDiff += // ...
76    }
77
78    // ...
79
80  }
81
82  std::cout << "dt: \t" << dt << " \t error: " << sqrt(normDiff/normSol) << std::endl;
83
84  return 0;
85
86 }
```

Ce programme nous servira à tester la convergence de la méthode d'Adams-Bashforth lors de la résolution de l'équation différentielle

$$\begin{cases} \partial_t y(t,x) = t^2 y(t,x), \\ y(0,x) = y_0(x) = -\sin\left(\frac{\pi}{2}x\right). \end{cases}$$

pour x et t compris entre 0 et 1. La solution de cette équation est

$$y(t,x) = \frac{2y_0(x)}{2 - t^2 y_0(x)}.$$

Les méthodes d'Euler et d'Adams-Bashforth font partie des nombreuses méthodes d'intégration numérique permettant de résoudre des équations différentielles ordinaires qui vous étudierez en détail au semestre suivant. La méthode d'Euler se base sur la troncature à l'ordre 1 du développement limité de la dérivée en temps de y :

$$y(t+\delta t) \sim y(t) + \delta t f(t,y).$$

Cela donne directement le schema numérique

$$y^{n+1} = y^n + \delta t f(t^n, y^n).$$

La méthode d'Adams-Bashforth utilise en plus de la donnée au temps t^n celle au temps précédent t^{n-1}. C'est pour cela qu'on parle de méthode multi-pas (ici multi=2). La valeur au temps t^{n+1} est obtenue par la formule :

$$y^{n+1} = y^n + \frac{3}{2}\delta t f(t^n, y^n) - \frac{1}{2}\delta t f(t^{n-1}, y^{n-1}).$$

Nous mesurons l'erreur commise par le schema numérique en calculant l'erreur relative en norme l^2 entre la solution analytique y et l'approximation obtenue par le schéma $\tilde{y}$ au temps $t = 1$:

$$e = \frac{\sqrt{\sum_i \left(\tilde{y}_i - y(t=1, x_i)\right)^2}}{\sqrt{\sum_i y(t=1, x_i)^2}}.$$

2. Que signifie l'argument `double (*f) (double,double)` des fonctions `oneStepEuler` et `oneStepAdamsBashforth` ? Quel est le lien avec le problème posé ?
3. Complétez le fichier `multistepODE.cpp` aux endroits indiqués pour parvenir à mesurer cette erreur. Vous pourrez vous inspirer de l'exercice précédent pour que le code soit efficace, même pour de grandes valeurs de N et de petites valeurs de dt.
4. Ajoutez une boucle pour calculer l'erreur en fonction du pas de temps qui prendra les valeurs 0.01, 0.005, 0.002 et 0.001. Vous écrirez les résultats dans un fichier `results.dat` dont la première colonne est le pas de temps et la seconde l'erreur. Vous pourrez ainsi déterminer l'ordre de convergence de la méthode. Le script gnuplot contenu dans l'archive de l'exercice réalise une régression linéaire du log des données et génère le graphique `convergence.png`. Les instructions pour lancer ce script sont données dans son en-tête.
5. L'instruction

```
#include <algorithm>
rotate(v.begin(),v.begin()+1,v.end());
```

permet de faire une permutation circulaire d'un cran vers la gauche du contenu du vecteur `v`. Si celui contient

```
0 1 2 3 4
```

l'appel à `rotate` donnera

```
1 2 3 4 0
```

Utilisez ceci dans la méthode multipas pour faire les échanges de tableaux.

5 Complexité algorithmique : tris de tableaux

L'exercice 3 montrait comment un choix d'implémentation pouvait considérablement améliorer le temps de calcul. Ici sera mis en avant l'importance du choix d'un algorithme en termes de performances.

Estimez la complexité des deux algorithmes de tri suivants en fonction de la taille N du tableau à trier. Implémentez, testez et chronométrez les avec des tableaux d'entiers générés aléatoirement (cf fonction `rand`).

```
triSelection(tableau t, taille n) :
        pour i de 0 à n-2 :
                imin = i
                pour j de i+1 à n-1 :
                        si t[j] < t[imin], imin = j
                fin pour
                si imin != i, échanger t[imin] et t[i]
        fin pour
fin triSelection
```

```
triFusion(tableau t, taille n) :
        si n<=1 :
                renvoyer t
        sinon :
                renvoyer fusion(triFusion(t[0,...,n/2]),triFusion(t[n/2+1,...,n-1]))
fin triFusion

fusion (tableau A, taille a, tableau B, taille b) :
         si A est vide :
                 renvoyer B
         si B est vide :
                 renvoyer A
         si A[0] < B[0] :
                 renvoyer [A[0],fusion(A[1,...,a],B)]
         sinon
                 renvoyer [B[0],fusion(A,B[1,...,b])]
fin fusion
```