Examen de langage C++ et calcul scientifique

DEA de mathématiques appliquées, Grenoble

Pierre Saramito

5 mars 2003

Le sujet d'examen comporte deux parties indépendantes. De plus, la plupart des questions sont indépendantes les unes des autres, bien que chaque partie ait une unité de thème.

Les documents sont autorisés : les notes de cours peuvent être utiles dans la dernière question de la première partie ainsi que dans la deuxième partie.

Partie 1 : classes-fonction

Dans cette partie, nous nous proposons d'étudier les *classes-fonction*, qui sont largement utilisées dans la librairie standard du C++. Une classe-fonction est simplement une classe qui définit `operator()`, de sorte qu'une variable de cette classe peut être utilisé de façon similaire à une fonction classique. Par exemple, pour $f(x) = e^{-x^2}$:

```
#include <functional>
#include <iostream>
#include <cmath>
using namespace std;
template <class T>
class ma_fonction : public unary_function<T,T> {
  public:
    T operator() (T x) const { return exp(-x*x); }
};
int main() {
  ma_fonction<double> f;
  cout << f(3.5) << endl;
}
```

Le fichier d'en-tête standard '`functional`' défini la classe :

```
template <class Argument, class Result>
class unary_function {
  public:
    typedef Argument argument_type;
    typedef Result   result_type;
};
```

Question 1

Écrire la fonction

```
template <class Function, class T>
void
max_abs(Function f, T a, T b, size_t n, T& value);
```

qui approche le maximum de la valeur absolue de f dans l'intervalle $[a, b]$ en effectuant n évaluations de f en des points équidistants.

Question 2

Donner un petit exemple complet d'utilisation de `max_abs`, tel qu'il a été introduit à la question précédente.

Question 3

Il serait plus confortable d'avoir une fonction `max_abs` qui donne le résultat en valeur de retour, plutôt que d'avoir à passer un argument `val`. Pour cela, on utilise les types définis dans la classe-fonction et hérités de la classe `unary_function` :

```
template <class Function, class T>
typename Function::result_type
max_abs(Function f, typename Function::argument_type a,
    typename Function::argument_type b, size_t n);
```

Écrire complètement la nouvelle fonction `max_abs` et **donner** un petit exemple complet d'utilisation.

Question 4

Le concept de classe-fonction permet aussi de représenter des fonctions dépendant d'un paramètre. Pour cela, nous déclarons dans la classe une section `protected` où sont mémorisés les valeurs des paramètres. Le constructeur prend en argument les valeurs de ces paramètres. **Écrire** une classe-fonction représentant $f_\alpha(x) = \exp(-\alpha x^2)$ et **donner** un petit exemple complet d'utilisation faisant intervenir la fonction `max_abs`.

Question 5

Est-il plus efficace, en terme de rapidité d'exécution, d'utiliser une fonction classique ou bien une classe-fonction ? On justifiera la réponse.

Question 6

Pour étendre cette notion à une fonction de $\mathbb{R}^2$ dans $\mathbb{R}$, nous réutilisons la classe `point` définie dans le cours. Si `x` est une variable de type `point`, alors `x[0]` est l'abscisse et `x[1]` l'ordonnée. **Écrire** la classe-fonction associée à $f(x, y) = xy(1-x)(1-y)$.

Question 7

Adapter la fonction `interpolate` définie dans le cours et opérant sur un maillage, de façon à ce qu'elle accepte des classes-fonctions de $\mathbb{R}^2$ dans $\mathbb{R}$.

Partie 2 : Préconditionneurs

On voudrait accélérer les algorithmes itératifs de résolution des systèmes linéaires, tels que le gradient conjugué. La *vitesse de convergence* d'un algorithme est le taux de réduction du résidu en fonction du nombre d'itérations.

La vitesse de convergence de ces algorithmes dépend directement des valeurs propres de la matrice du système linéaire à résoudre. Soit A une matrice symétrique définie positive. Une approche consiste à remplacer le système

$$Ax = b$$

à résoudre par un syst'eme de même solution x mais dont les valeurs propres sont mieux réparties, et donc plus facile à résoudre par un algorithme itératif. On dira qu'il est mieux *conditionné*.

Un *préconditionneur* est une matrice qui effectue ce type de transformation :

$$M^{-1}Ax = M^{-1}b$$

Pour $M = I$, nous obtenons le préconditionnement identité, c'est-à-dire pas de préconditionnement. Pour $M = A$, l'algorithme itératif convergera en une itération, mais il faudra résoudre un système du type

$$Mz = r$$

dont la complexité est égale au problème initial : nous n'avons pas progressé !

Il s'agit donc de faire un compromis entre ces deux extrêmes : M doit être aussi proche que possible de A tout en assurant que les systèmes du type $Mz = r$ se résolvent facilement.

Question 1

En utilisant $M = \mathrm{diag}(A)$, la matrice diagonale composée exclusivement des coefficients diagonaux de A, nous nous rapprochons un peu de A tout en s'assurant que M est très facile à résoudre.
Définir la classe `diag` telle que
- le constructeur prend en argument une matrice creuse telle que définie dans le cours ;
- la classe mémorise les valeurs des coefficients diagonaux dans un tableau de type `valarray` ;
- la fonction membre `solve` prend en argument un vecteur r, de type `valarray` et retourne $z = M^{-1}r$, de type `valarray` également.

Donner le code du constructeur, de la fonction membre `solve`, ainsi que du constructeurs de copie et de l'opérateur d'affectation.

Question 2

Donner un petit exemple complet d'utilisation de la classe précédente faisant intervenir un appel de l'algorithme du gradient conjugué.

Question 3

Le préconditionnement diagonal précédent est certes simple à programmer, mais ne permet pas d'accélérer fortement l'algorithme. Une méthode très populaire, car efficace, consiste à utiliser comme préconditionneur une itération d'une méthode de type Gauss-Seidel, ou de son extension, la méthode de relaxations successives symétrisées (SSOR : *successive symmetric over-relaxations*).

Pour cela, on décompose

$$A = D + L + L^T$$

où D est la diagonale de A et L est sa partie triangulaire inférieure. La matrice de préconditionnement M est définie par :

$$M = \frac{1}{2-\omega}\left(\frac{1}{\omega}D + L\right)\left(\frac{1}{\omega}D\right)^{-1}\left(\frac{1}{\omega}D + L\right)^{T}$$

où $\omega \in]0,2[$ est le paramètre de relaxation.

Décomposer la résolution de $Mz = r$ en trois étapes ne faisant intervenir que des résolutions de systèmes diagonaux (D) ou triangulaires (L ou L^T).

Question 3

Nous supposons que la partie triangulaire inférieure L est représentée par une matrice creuse.

Écrire en pseudo-code, et en déroulant les indices des boucles, la résolution précédente.

Indications : Le pseudo-code sera un peu analogue à celui utilisé dans le cours pour spécifier le produit matrice-vecteur. Nous utiliserons trois boucles.

Question 4

Définir la classe `ssor` telle que
- le constructeur prend en argument la matrice creuse A et le paramètre d'affectation ω ;
- la classe mémorise les valeurs des coefficients diagonaux dans un tableau de type `valarray`, et ceux de la partie triangulaire inférieure L dans une classe de type `matrix` ;
- la fonction membre `solve` prend en argument un vecteur r, de type `valarray` et retourne $z = M^{-1}r$, de type `valarray` également.

Donner le code du constructeur, de la fonction membre `solve`, ainsi que du constructeurs de copie et de l'opérateur d'affectation.

Indication : on remarquera que le nombre de coefficients non-nuls de la partie triangulaire L est donné par

$$nnz(L) = \frac{nnz(A) - nrow(A)}{2}$$

où $nrow(A)$ et $nnz(A)$ sont respectivement les nombres de lignes et de coefficients non-nuls de A.

Question 5

Donner un petit exemple complet d'utilisation de la classe `ssor`, et faisant intervenir un appel de l'algorithme du gradient conjugué.