Examen de langage `C++` et calcul scientifique

2ème anné de maîtrise, Grenoble

Pierre Saramito

2 mars 2005

Examen commun aux options :
- ingénierie de la modélisation et simulation numérique
- mathématiques appliquées

Le sujet d'examen comporte trois 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.

Partie 1 : compléments sur les matrices creuses

Écrire une fonction `frobenius_norm`, qui prend en entrée une matrice creuse, de type `matrix`, telle que définie dans le cours, et renvoie :

$$\|A\|_F = \left(\sum_{i=0}^{n-1} \sum_{j=0}^{m-1} |a_{i,j}|^2 \right)^{1/2}$$

Afin d'accéder aux champs protégés, on supposera que cette fonction est *amie* de la classe `matrix`.

Partie 2 : quaternions

Les quaternions sont des nombres *hypercomplexes* qui forment un groupe non commutatif. Ils peuvent être représentés à l'aide des matrices complexes 2×2 :

$$h = \begin{pmatrix} z & w \\ -\bar{w} & \bar{z} \end{pmatrix} = \begin{pmatrix} a+ib & c+id \\ -c+id & a-ib \end{pmatrix} = a\mathcal{U} + b\mathcal{I} + c\mathcal{J} + d\mathcal{K}$$

avec

$$\mathcal{U} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad \mathcal{I} = \begin{pmatrix} i & 0 \\ 0 & -i \end{pmatrix}, \quad \mathcal{J} = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}, \quad \mathcal{K} = \begin{pmatrix} 0 & i \\ i & 0 \end{pmatrix}$$

et $\mathcal{I}^2 = \mathcal{J}^2 = \mathcal{K}^2 = -\mathcal{U}$ généralisent les nombres imaginaires purs. La norme de h est définie par $|h| = \sqrt{|z|^2 + |w|^2}$ et le conjugué de h est $\bar{h} = a\mathcal{U} - b\mathcal{I} - c\mathcal{J} - d\mathcal{K}$.

L'objectif de cette partie est de construire une classe `quaternion` ayant le même type d'interface, et compatible avec la classe `complex` de la librairie standard `C++` :

```
template <class T>
class complex {
    public:
      complex(const T& a=0, const T& b=0);
      complex(const complex<T>& z);
      complex<T>& operator= (const complex<T>& z);
      friend complex<T> operator+ (const complex<T>& z, const complex<T>& w);
      friend complex<T> operator- (const complex<T>& z); // moins unaire
      friend complex<T> operator- (const complex<T>& z, const complex<T>& w);
      friend complex<T> operator* (const complex<T>& z, const complex<T>& w);
      friend complex<T> operator/ (const complex<T>& z, const complex<T>& w);
      friend complex<T> conj (const complex<T>& z); // conjugué
      friend T abs  (const complex<T>& z); // norme
    // ...
};
```

Question 1 : Définir la classe `quaternion` en écrivant deux constructeurs, l'un à base de type `T` scalaire, et l'autre à base de type `complex<T>`.

Question 2 : Écrire le constructeur de copie et l'opérateur d'affectation.

Question 3 : Écrire les opérations d'addition, de soustraction et de multiplication.

Question 4 : Montrer que $h^{-1} = \bar{h}/|h|^2$. **Écrire** les fonction `conj` (conjugué), `abs` (norme de h) puis la division de deux quaternions.

Remarques : les quaternions ont été introduits en 1853 par HAMILTON. Ils plus tard été utilisés en mécanique quantique, et, plus récemment, en animation 3D, pour calculer des rotations d'axes.

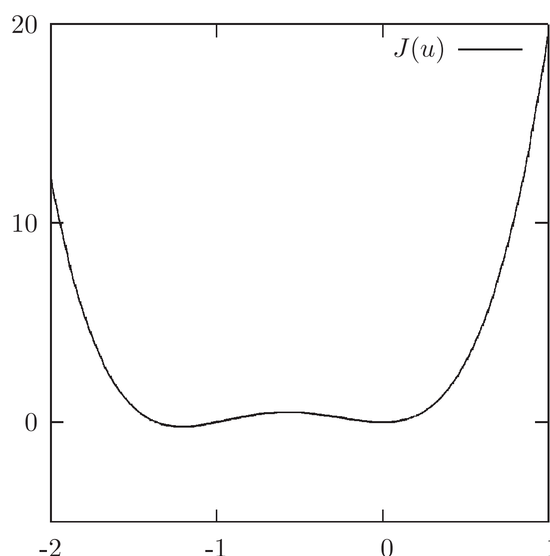
Partie 3 : différentiation automatique

On considère le problème de minimisation :
(P) : *trouver* $u \in \mathbb{R}$ solution de

$$\min_{v \in \mathbb{R}} J(v)$$

avec

$$\begin{aligned} F(v) &= 2v(v+1) \\ J(v) &= F(v) \times (F(v) + \sin(v)) \end{aligned}$$



La fonction $J(v)$ est souvent assez difficile à minimiser. En particulier, elle présente ici des minimums locaux en $u = 0$ et $u \approx -1.20662$, le second étant un minimum global.
La méthode de NEWTON recherche un zéro de $J'(v)$ en construisant une suite $(u_n)_{n \geqslant 0}$ selon :
$n = 0$: $u_0 \in \mathbb{R}$ donné
$n \geqslant 0$: $u_n \in \mathbb{R}$ connu, calculer

$$u_{n+1} := u_n - (J''(u_n))^{-1} \times J'(u_n)$$

Cette méthode converge vers un des minimums locaux de J, selon le choix initial de u_0.

Une façon de calculer $J'(u)$ est de l'approcher par la différence $(J(u+\epsilon) - J(u))/\epsilon$. Cependant, cette quantité est sensible au choix de ϵ, et l'algorithme résultant ne sera pas très robuste, d'autant plus qu'il faudra également évaluer $J''(u)$ de façon approché. Les erreurs peuvent s'accumuler.

Dans la pratique, l'algorithme est beaucoup plus robuste lorsque nous évaluons exactement $J'(u)$ et $J''(u)$.

Question 1 : Calculer J' et J''. **Écrire** le code d'une fonction résolvant le problème de minimisation de J par la méthode de Newton en n_{max} itérations au plus, et le test d'arrêt $|J'(u_n)| < \varepsilon$, où u_0, n_{max} et ε sont donnés.

Question 2 : L'expression de J peut être arbitrairement complexe, des milliers de lignes de code dans la pratique, et issu d'un calcul numérique, différences finies, éléments finis,... Nous allons utiliser une technique récente de dérivation automatique de code qui s'applique sur chaque instruction : si une fonction est décrite par son implémentation numérique, alors ses dérivées seront calculables exactement. Pour cela, considérons la classe :

```
class da_double {
    public:
      da_double (double x0 = 0) : x(x0), dx(0) {}
      da_double (const da_double& f);
      da_double& operator= (const da_double& f);

      void is_variable ()  { dx = 1; }
      const double& value() const { return x; }
      const double& diff()  const { return dx; }

      friend da_double operator+ (const da_double& f, const da_double& g);
      friend da_double operator* (const da_double& f, const da_double& g);
      friend da_double sin (const da_double& f);
    protected:
      double x, dx;
};
```

La *surcharge* de l'opérateur de multiplication fera que, chaque fois que $f \times g$ apparaît, nous calculerons simultanément $f'g + fg'$ = `f.dx*g.x + f.x*g.dx`. Un objet de type `da_double` est par défaut une constante. L'instruction `x.is_variable()` affecte 1 au champs `x.dx` : la dérivée de x par rapport à x vaut 1. Par exemple, le fragment de code :

```
da_double x = 3;
x.is_variable();          // dx = 1 : la dérivée de x par x vaut 1
da_double f = 2*x*(x+1);
cout << f.diff() << endl;
```

affichera la dérivée de F en $x0 = 3$.

S'appuyant sur les formules de dérivation de $(f+g)'$, $(f \times g)'$ et $(\sin(f))'$, **compléter** le code des fonctions de la classe `da_double`.

Question 3 : Donner un petit programme de test complet qui affiche sur trois colonnes les valeurs de $(u, J(u), J'(u))$ entre -2 et 1 par pas de 0.01.

Afin d'utiliser, lors des appels des fonctions F et J, un argument u de type `da_double`, nous définirons une fois pour toute ces fonctions à l'aide d'un type générique `T`. D'autre part, nous n'aurons plus à utiliser explicitement les expressions des dérivées de ces fonctions.

Question 4 : Nous modifions la classe `da_double` pour en faire une classe générique `da` :

```
template <class T>
class da {
    public:
      da (double x0 = 0) : x(x0), dx(0) {}
      template <class U> da (const da<U>& x0) : x(x0), dx(0) {}
      da<T>& operator= (const da<T>& f);

      void is_variable ()  { dx = 1; }
      const T& value() const { return x; }
      const T& diff()  const { return dx; }

      friend da<T> operator- (const da<T>&  f); // moins unaire
      friend da<T> operator+ (const da<T>&  f, const da<T>&  g);
      friend da<T> operator+ (const da<T>&  f, const double& g);
      friend da<T> operator* (const da<T>&  f, const da<T>&  g);
      friend da<T> operator* (const double& f, const da<T>&  g);
      friend da<T> sin (da_double f);
      friend da<T> cos (da_double f);
    protected:
      T x, dx;
};
```

Ceci va nous permettre de calculer les dérivées secondes et d'ordre supérieur de façon automatique : nous seront à même de construire *récursivement* les types `da<da<double> >`, `da<da<da<double> > >` ...

Compléter le code des fonctions membres de la classe `ad`.

Question 5 : La fonction générique :

```
template <class T>
T dJ_dx (const T& v) {
    da<T> w = v;
    w.is_variable();
    return J(w).diff();
}
```

permet de calculer J'.

Dans cet esprit, **écrire** la fonction `d2J_dx2` calculant J''.

Question 6 : la fonction `d2J_dx2` appelle `dJ_dx`, qui elle-même appelle `J` : quels sont successivement les types des paramètres `v` lors des appels de ces fonctions génériques ?

Question 7 : à quoi sert le constructeur utilisant un argument générique `U` au lieu de `T` ? Est-ce un constructeur de copie ?

Question 8 : En déduire un progamme calculant un minimum de J en utilisant la classe `da` et la méthode de Newton, en n_{max} itérations au plus, et le test d'arrêt $|J'(u_n)| < \varepsilon$, où u_0, n_{max} et ε sont donnés.

Quel est l'avantage de cette version par rapport à celle de la question 1 ?

Remarques : pour en savoir plus sur ces techniques très en vogue actuellement, on pourra consulter les librairie `fadpad` ou `adol-c` par exemple. Cette technique s'applique encore quand J est issue d'un calcul approché : traitement d'image, discrétisation d'équations aux dérivées partielles, etc.