PFA: spécifications

Table des matières

Remarques préliminaires

1. Pour comprendre ces spécifications il faut avoir lu, au moins dans les grandes lignes, le document mathématique de PFA (disponible sur moodle).

2. Sur la nomenclature des fonctions C : les fonctions de la partie «assurance» ont toutes un nom de la forme `client<PDF ou CDF>_<nom de la variable aléatoire>` :
   - PDF ou CDF : l'acronyme PDF (Probability Density Function) désigne une densité ; l'acronyme CDF (Cumulative Distribution function) désigne une fonction de répartition.
   - Le nom de la variable aléatoire est celui qu'on retrouve dans le document mathématique.

   Par exemple, dans le document mathématique, on définit pour un client donné, en cas de sinistre, la variable aléatoire

   $$X = \text{«Remboursement de l'assurance au client lié ce sinistre»}$$

   Alors les fonctions C calculant la densité et la fonction de répartition de cette variable X seront nommée `clientPDF_X` et `clientCDF_X`.

But du projet

Le but final du projet est d'écrire les deux fonctions C suivantes :

```
double optionPrice(Option* option);
double clientCDF_S(InsuredClient* client, double x);
```

L'argument `option` de la première fonction est un pointeur sur une structure, cette structure définit le type `Option` (avec un `typedef`). La structure regroupe toutes les caractéristiques de l'option et est décrite plus loin dans ce document. Elle est définie dans le fichier `pfa.h`, il est déconseillé de la modifier.

De même, l'argument `client` de la seconde fonction est un pointeur sur une structure qui définit le type `InsuredClient`. Elle regroupe toutes les caractéristiques d'un client d'une compagnie d'assurance (c'est à dire d'un assuré) et est décrite plus loin. Cette structure est elle aussi est définie dans le fichier `pfa.h`, il vaut mieux ne pas la modifier.

La première fonction, `optionPrice`, doit retourner le prix de l'option en euros. Et la seconde, `clientCDF_S`, doit retourner la valeur en x de la fonction de répartition de la variable aléatoire :

$S =$ «Somme des remboursements que la compagnie d'assurance devra verser au client cette année»

On rappelle que la fonction de répartition de la variable aléatoire S est la fonction F_S définie pour tout $x \in \mathbb{R}$ par : $F_S(x) = P(S \leqslant x)$. Elle définit la loi de S.

Mais pour coder ces deux fonctions, il faut en écrire d'autres au préalable. Plusieurs des fonction à écrire font appel à l'intégration numérique. Vous passerez donc une bonne partie du projet à programmer et évaluer les différentes méthodes d'intégration. Ensuite seulement, vous pourrez passer aux fonctions probabilistes et finalement aux deux fonctions finales `optionPrice` et `clientCDF_S`.

1 Intégration numérique

1.1 Fonctions et structures obligatoires (fichier `integration.c`)

Les fonctions que vous devez coder sont déclarées et écrites dans les fichiers `integration.h` et `integration.c`. Ce dernier fichier ne doit pas contenir de fonction `main`. Pour tester vos fonctions d'intégration, utilisez un (ou plusieurs) autre fichier C. Vous pouvez par exemple coder vos tests dans le fichier `test_integration.c` qui vous est fourni et qui contient une fonction `main`.

Le fichier `integration.h` doit définir le type `QuadFormula` qui permet de spécifier une formule de quadrature :

```
typedef struct{
...
} QuadFormula;
```

Mettez dans cette structure tous les champs dont vous aurez besoin pour faire les calculs d'intégration.

Le fichier `integration.c` doit définir les fonctions listées ci-dessous. Votre code doit respecter exactement ces signatures, car il sera évalué en partie par une moulinette.

- `bool setQuadFormula(QuadFormula* qf, char* name)` : affecte les champs de `qf` aux valeurs correspondant à la formule de quadrature spécifiée par `name`.

  Arguments :
  `qf` : un pointeur sur l'objet de type `QuadFormula` à mettre à jour.
  `name` : une chaine de caractères donnant le nom de la formule de quadrature. Peut valoir `"left"`, `"right"`, `"middle"`, `"trapezes"`, `"simpson"`, `"gauss2"` ou `"gauss3"`.

  Valeur retournée : la fonction retourne `true` si tout s'est bien passée. Sinon, elle retourne `false` (par exemple, si le nom de la formule de quadrature contenu dans `name` n'est pas reconnu).

- `double integrate(double (*f)(double), double a, double b, int N, QuadFormula* qf)` : calcule une approximation numérique de $\int_a^b f(t)\,\mathrm{d}t$.

  Arguments :
  `f` : un pointeur sur la fonction à intégrer.
  `a` et `b` : les bornes de l'intervalle d'intégration.
  `N` : le nombres de subdivisions. L'intervalle $[a, b]$ est découpé en N subdivisions $[a_i, b_i]$, puis on utilisé une formule de quadrature pour approximer l'intégrale de f sur chaque subdivision.
  `qf` : la formule de quadrature à utiliser.

  Valeur retournée : l'approximation numérique de l'intégrale.

- `double integrate_dx(double (*f)(double), double a, double b, double dx, QuadFormula* qf)` : calcule une approximation numérique de $\int_a^b f(t)\,\mathrm{d}t$. Le nombre de subdivisions est calculé à partir de la valeur `dx`.

  Arguments :
  `f` : un pointeur sur la fonction à intégrer.
  `a` et `b` : les bornes de l'intervalle d'intégration.
  `dx` : la longueur des subdivisions. L'intervalle $[a, b]$ est découpé en N subdivisions $[a_i, b_i]$. On calcule N tel que la longueur de chaque subdivision, $(b_i - a_i) = \dfrac{b-a}{N}$, soit le plus proche possible de `dx` (compte tenu que N doit être un entier).
  `qf` : la formule de quadrature à utiliser.

  Valeur retournée : l'approximation numérique de l'intégrale.

  Indication : on écrira `N = (int) round( abs(b-a)/dx );` et on appellera la fonction `integrate`. Attention à bien prendre la ***valeur absolue*** de $b-a$ (si $b < a$, il ne faut pas que N soit négatif). Et attention aussi au cas où cette formule conduit à $N = 0$ (cas d'un très petit intervalle $[a, b]$) : il faut alors donner à N la valeur 1 pour qu'il y ait au moins une subdivision.

Exemple d'utilisation : pour calculer $I = \int_{-1}^{4} \sin\left(t^2\right)\,\mathrm{d}t$, on pourra écrire :

```
#include <math.h>
#include "integration.h"

double f(double t)
{
  return sin(t*t);
}
```

```
int main()
{
   QuadFormula qf;
   double I1, I2, I3;

   setQuadFormula(&qf, "trapezes");
   I1 = integrate(f, -1, 4, 10, &qf);

   setQuadFormula(&qf, "middle");
   I2 = integrate(f, -1, 4, 10, &qf);

   setQuadFormula(&qf, "simpson");
   I3 = integrate_dx(f, -1, 4, 0.1, &qf);
}
```

Après l'exécution de ces lignes de codes :

- `I1` est une approximation de I obtenue en découpant $[-1, 4]$ en 10 subdivisions et en appliquant la formule de quadrature des «trapèzes» sur chaque subdivision.
- `I2` est une approximation de I obtenue en découpant $[-1, 4]$ en 10 subdivisions et en appliquant la formule de quadrature «rectangle milieu» sur chaque subdivision.
- `I3` est une approximation de I obtenue en découpant $[-1, 4]$ en N subdivisions, N étant calculé de façon à ce que chaque subdivision soit de longueur 0.1 (donc dans ce cas, $N = 50$). On a appliqué la formule de quadrature de Simpson sur chaque subdivision.

Un conseil important : définissez bien votre structure `QuadFormula`. Quand vous exécutez la fonction `integrate`, il ne faut pas recalculer à chaque subdivision les points x_k et les coefficients w_k qui interviennent dans la formule de quadrature. Cela aboutirait à des calculs inutiles et parfois longs (formules de Gauss à 2 et 3 nœuds par exemple). Ces points x_k et coefficients w_k doivent donc être calculés une fois pour toutes, par exemple à l'initialisation du programme ou lors de l'exécution de `setQuadFormula`. Ils pourront être stockés dans la structure `QuadFormula`. **La moulinette de test vérifiera qu'il n'y a pas trop de calculs inutiles**.

1.2 Compiler et tester les différentes méthodes d'intégration

Vous allez devoir tester et comparer les différentes formules de quadrature. Il faut donc écrire un programme de test qui vous permet d'évaluer dans quelles conditions on obtient les meilleurs approximations d'intégrales. Le fichier `test_integration.c` qui vous est fourni au départ est fait pour ça. À vous d'y mettre les tests que vous pensez pertinents.

On compile la partie sur l'intégration par la commande suivante : `make integration`. Elle crée les fichiers objets `integration.o` et `test_integration.o` et l'édition de lien construit ensuite le fichier exécutable `test_integration`.

Évaluation de la partie sur l'intégration

- La moulinette n'évalue que les fonctions `integrate` et `integrate_dx`. Elle appellera pour cela votre fonction `setQuadFormula`, qui doit donc elle aussi être opérationnelle.
- Votre «client», représenté par l'enseignant de maths qui vous accompagne, doit voir vos tests. Vis-à-vis de lui, vous devez conclure cette partie en lui annonçant les paramètres d'intégration (formule de quadrature, nombre de subdivisions ou longueurs des subdivisions) que vous choisissez pour la suite de projet.
  Vous devez donc lui remettre **un court document (une ou deux diapos) qui annonce ce choix et le justifie** (résultats de vos tests). Le moodle contient une interface pour la remise du document.

2 Finance et assurance

Le fichier `pfa.c` contient les fonctions obligatoires. Elles seront évaluées en partie par moulinette et doivent donc respecter les signatures spécifiées ici. Le fichier `test_pfa.c` contient une fonction `main`. Il vous permet de coder vos tests des fonctions définies dans `pfa.c`. En fin de projet, `test_pfa.c` pourra par exemple vous permettre de coder un démonstrateur, pour valider vos résultats numériques avec votre «client».

2.1 Fonctions et structures obligatoires (fichier `pfa.c`)

Les variables globales suivantes sont accessibles dans toutes les fonctions de `pfa.c` :

```
QuadFormula pfaQF;
double pfa_dt;
```

Elles indiquent le choix que vous avez fait des paramètres d'intégration : choix de la formule de quadrature et de la longueur des subdivisions. Tout appel à `integrate_dx` dans `pfa.c` se fera avec ces deux variables en argument. La fonction suivante permet de fixer leurs valeurs :

```
bool init_integration(char* quadrature, double dt)
```

Arguments :
`quadrature` : une chaine de caractères donnant le nom de la formule de quadrature à utiliser. Valeurs possibles : `"left"`, `"right"`, `"middle"`, `"trapezes"`, `"simpson"`, `"gauss2"` ou `"gauss3"`.
`dt` : un nombre strictement positif donnant la longueur des subdivisions.

Valeur retournée : la fonction retourne true si tout s'est bien passé. Elle retourne false en cas de problème (valeur de `quadrature` non reconnue, ou valeur négative de `dt` par exemple).

Exemple d'utilisation dans `test_pfa.c` :

```
#include <math.h>
#include "integration.h"

int main()
{
   init_integration("trapezes", 0.1);
   ...
}
```

Dans la suite de l'exécution du programme, toutes les fonctions de `pfa.c` faisant appel à un calcul intégral utiliseront la formule de quadrature des trapèzes sur des subdivisions de longueurs 0.1. Vous les coderez en appelant `integrate_dx` avec les arguments `&pfaQF` et `pfa_dt`.

Vous devez ensuite coder les deux fonctions définissant la loi normale :

- double phi(double x) : densité de la loi normale. La fonction retourne $\varphi(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}$.
- double PHI(double x) : fonction de répartition de la loi normale. La fonction retourne

$$\Phi(x) = \frac{1}{2} + \int_0^x \varphi(t)\,\mathrm{d}t$$

Vous devez ensuite coder la fonction `priceOption` qui prend en argument une variable de type `Option` (prédéfini dans le fichier `pfa.h`) :

```
typedef enum CALL=0, PUT OptionType;

typedef struct{
   OptionType type; /* CALL or PUT option */
   double S0;
   double K;
```

```
    double T;
    double mu;
    double sig;
} Option;
```

Un objet de type `Option` a des champs qui correspondent aux notations du document mathématique : S_0 (prix initial de l'actif sous-jacent), K (prix d'exercice de l'option), T (échéance de l'option), μ et σ (paramètres de la loi de la variable aléatoire S_T).

La fonction `priceOption` a la signature suivante :

```
double priceOption(Option* option);
```

Elle retourne le prix de l'option.

Les fonctions liées à l'assurance prennent toutes en argument une variable de type `InsuredClient*`. Ce type est prédéfini dans `pfa.h` :

```
typedef struct{
    double m;
    double s;
    double* p;
} InsuredClient;
```

Les champs `m` et `s` sont les paramètres μ et σ du document mathématique : à chaque sinistre, la variable aléatoire

$$X = \text{«Remboursement que la compagnie doit verser au client suite à ce sinistre»}$$

suit une loi log-normale de paramètres $(\mu, \sigma^2) \in \mathbb{R} \times \mathbb{R}_+^*$.

Le champs `p` pointe sur un tableau de 3 nombres : `p[0]`, `p[1]` et `p[2]` sont les probabilités que le client déclare 0, 1 ou 2 sinistres sur l'année.

Les fonctions que vous devez coder décrivent les lois de ces trois variables aléatoires :

$$\begin{array}{rcl} X & = & \text{«Remboursement que la compagnie doit verser au client suite à un sinistre déclaré»} \\ X_1{+}X_2 & = & \text{«Remboursement que la compagnie doit verser au client suite à deux sinistres déclarés»} \\ S & = & \text{«Total des remboursements que la compagnie doit verser au client sur l'année»} \end{array}$$

Ces fonctions sont :

- `double clientPDF_X(InsuredClient* client, double x)` : retourne la valeur en x de la densité de la variable aléatoire X.
- `double clientCDF_X(InsuredClient* client, double x)` : retourne la valeur en x de la fonction de répartition de la variable aléatoire X.
- `double clientPDF_X1X2(InsuredClient* client, double x)` : retourne la valeur en x de la densité de la variable aléatoire $X_1 + X_2$.
- `double clientCDF_X1X2(InsuredClient* client, double x)` : retourne la valeur en x de la fonction de répartition de la variable aléatoire $X_1 + X_2$.
- `double clientCDF_S(InsuredClient* client, double x)` : retourne la valeur en x de la fonction de répartition de la variable aléatoire S.

2.2 Compiler et tester les fonctions finance et assurance

Pour tester ces fonctions, codez vos tests dans `test_pfa.c`, ce fichier contient une fonction `main`. La commande `make pfa` génère les fichiers objets `pfa.o` et `test_pfa.o`, puis l'exécutable `test_pfa`.

Ces tests doivent vous permettre de démontrer à votre «client» les résultats numériques de vos fonctions. Une partie de l'évaluation est faite par votre «client».

Une autre partie de l'évaluation est faite par moulinette. Seule les fonctions de `pfa.c` sont testées. Vous faites donc ce que vous voulez dans `test_pfa.c`.