Programmation scientifique orientée objet

Master 1 MI
UFR Sciences et Techniques
Université du Havre

Damien Olivier

13 novembre 2009

Avertissement

Certains exercices ont été ou pourront être empruntés à d'autres enseignants, je les remercie ici et je m'excuse de ne pas les avoir cités dans ce document. Ils le seront dans la version finale. Les étudiants sont ce qu'ils sont et choisissent parfois la facilité en allant directement chercher les solutions j'ai donc choisi de ne pas leur faciliter la tache.

Chapitre 1

Tour de chauffe

Cette partie à pour objectif de revoir les bases qui ont déjà été enseignées en L3 à certains d'entre vous et qui peuvent être nouvelles pour d'autres.

1.1 UML

Exercice 1

I L'équipage d'un avion est constitué d'un pilote, d'un copilote et de plusieurs hôtesses. Chacune de ces personnes est identifiée par son nom et sa fonction. Ces équipages doivent être opérationnels sur trois types d'avions : Airbus A320, Boeing 747 et Concorde. Les vols cités dans la table ci-dessous seront identifiés par la tour de contrôle de l'aéroport d'Orly ou de Roissy, par le modèle de l'avion, leur numéro de vol et leur destination.

On présente ci-dessous un extrait du tableau de service de quelques employés de la compagnie AIR FRANCE :

VOL	DEST	DATE	MODELE	AEROPORT	NOM	FONCTION
AF347	Londres	11/10/97	A320	Orly	Pierre	Pilote
AF347	Londres	11/10/97	A320	Orly	Paul	Copilote
AF347	Londres	11/10/97	A320	Orly	Jeanne	Hôtesse
AF347	Londres	11/10/97	A320	Orly	Marie	Hôtesse
AF347	Londres	11/10/97	A320	Orly	Isabelle	Hôtesse
AF545	New-York	12/10/97	Concorde	Roissy	Jacques	Pilote
AF545	New-York	12/10/97	Concorde	Roissy	Paul	Copilote
AF545	New-York	12/10/97	Concorde	Roissy	Marie	Hôtesse
AF545	New-York	12/10/97	Concorde	Roissy	Véronique	Hôtesse

I.1 Etablir la liste des classes et de leurs champs. On identifiera aussi les instances des classes.

I.2 Modéliser les classes correspondant aux différentes notions énoncées. Représenter graphiquement les classes et leurs instances.

I.3 Représenter les liens d'héritage entre ces différentes classes.

I.4 Représenter les liens d'association (en précisant les rôles) et d'agrégation entre ces classes.

Exercice 2

I Dans une gare, on veut faire des statistiques sur les billets délivrés pendant une journée. Le coût d'un trajet est proportionnel au nombre de kilomètres parcourus. Les trajets en TGV sont majorés d'un supplément proportionnel à la longueur du parcours effectué et d'un coût de réservation dépendant de la gare de départ. De plus un utilisateur peut bénéficier d'une réduction (pourcentage), qui ne s'applique pas aux suppléments dus à un parcours TGV.

On veut construire un système permettant de délivrer les billets de train et faire des statistiques, par exemple sur le nombre de billets pour un parcours supérieur à 250km. Etablir la liste des classes de leurs champs et de leurs méthodes.

I.1 Donner le diagramme de cas d'utilisation.

I.2 Identifier les différentes classes et définir le diagramme de classe en spécifiant les associations entre les classes.

Exercice 3

I L'objectif de cet exercice est d'utiliser UML sur un cas simple et couvrant de nombreux aspects de ce langage.

Un gérant de bibliothèque désire automatiser la gestion des prêts. Il commande un logiciel permettant de répondre à la liste des besoins suivants :

1. Les utilisateurs peuvent connaître les livres présents et en réserver jusqu'à 2 en même temps.
2. L'adhérent peut connaître la liste des livres qu'il a empruntés ou réservés.
3. L'adhérent possède un mot de passe qui lui est donné à son inscription.
4. L'emprunt est toujours réalisé par les employés qui travaillent à la bibliothèque. Après avoir identifié l'emprunteur, ils savent si le prêt est possible (nombre maximum de prêts = 5), et s'il a la priorité (il est celui qui a réservé le livre, si ce dernier est réservé).
5. Ce sont les employés qui mettent en bibliothèque les livres rendus et les nouveaux livres. Il leur est possible de connaître l'ensemble des prêts réalisés dans la bibliothèque.

I.1 Identifier les acteurs du système.

I.2 Donner la liste des cas d'utilisation.

I.3 En utilisant les acteurs et les différents cas d'utilisation définir un diagramme détaillé de cas d'utilisation.

I.4 Proposer une description d'un scénario d'un cas d'utilisation réaliste avec authentification.

I.5 En fonction de ce scénario définir les conditions (pré et post) d'utilisation.

I.6 Lister de la façon la plus exhaustive possible, les variantes pouvant intervenir dans votre scénario.

I.7 Proposez une modélisation des médias disponibles dans une bibliothèque : livre, CD, vidéo, etc. sous forme de diagramme de classes.

I.8 Modéliser sous forme de diagramme de classe l'application bibliothèque.

I.9 Donner le diagramme de séquences concernant les informations.

I.10 Donner le diagramme de séquences concernant les réservations.

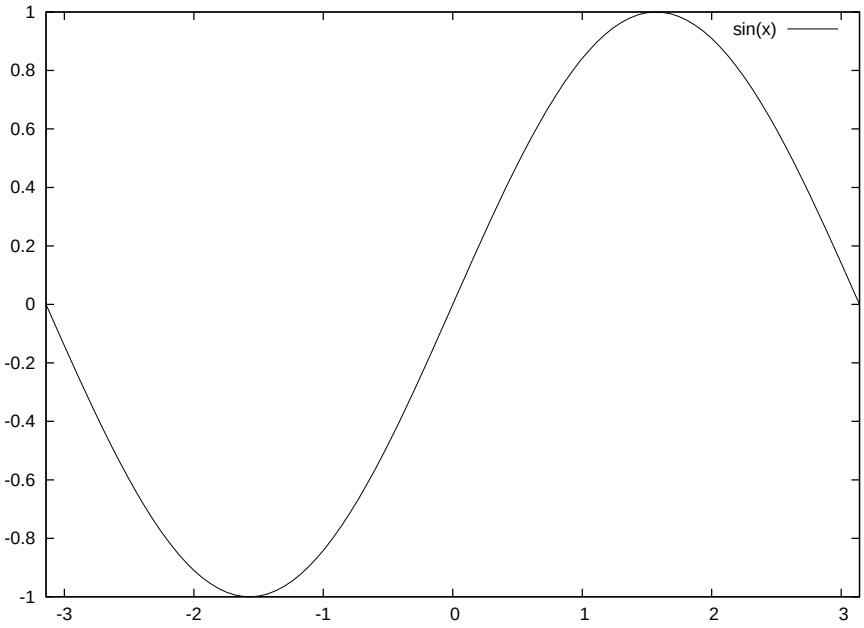

FIG. 1.1 – fonction sin(x) sur l'intervalle $[-\Pi, \Pi]$.

I.11 Donner le diagramme d'états-transitions concernant un livre.

I.12 Donner le diagramme d'états-transitions concernant un utilisateur.

Exercice 4

I Maintenant c'est l'heure de votre pause café.

I.1 Avant de la prendre, définir un diagramme d'activité correspondant.

1.2 Gnuplot

Dans cette partie nous allons présenter Gnuplot par l'exemple qui vous permet de tracer des courbes aussi bien en ligne de commande qu'à l'aide d'un fichier contenant ces mêmes commandes. Nous nous proposons dans la suite de survoler ces différentes commandes.

1.2.1 Généralité

Pour commencer on lance Gnuplot dans un terminal `gnuplot`.

```
1 gnuplot>
2 gnuplot> plot [-pi : pi] sin(x)
```

Vous tracera la fonction $sin(x)$ sur l'intervalle $[-\Pi, \Pi]$ comme le montre la figure 1.1. Pour sortir il vous restera à utiliser la commande `quit`. Vous pouvez également stocker vos commandes dans un fichier, il n'y a plus alors d'interactivité. Voici un exemple `tangente.gnu`.

```
#définition de quelques variables
        xmin=-6.
        xmax=6.
        ymin=-4.
        ymax=4.

        xdec=0.25
        ydec=0.25
        pasx=1.0
        pasy=1.0

#initialisation du terminal
reset
set term wxt
unset autoscale
set xr [xmin:xmax]
set yr [ymin:ymax]

#options
unset border
unset label
unset xtics
unset ytics

set title "fonction tangente"

#les axes
set arrow 3 from  xmin,0 to xmax,0,3 lt -1 lw 0.5
set arrow 4 from  0,ymin to 0,ymax,3 lt -1 lw 0.5

#l'origine
set label "O" at xdec/2, -ydec

set label "x" at xmax - pasx, -ydec
set label "y" at -xdec, ymax - pasy/3

set label "1" at pasx, -ydec
set label "1" at -3*xdec/2, pasy
set label "-1" at -3*xdec/2, -pasy

set arrow from  1, -ydec/2 to  1, ydec/2 nohead lt -1

set arrow from  1.57, -ydec/2 to  1.57, ydec/2 nohead lt -1
set label "\34/2" at 1.57, -ydec center

set arrow from  -1.57, -ydec/2 to  -1.57, ydec/2 nohead lt -1
set label "-\34/2" at -1.57, -ydec center

set arrow from  3.1415, -ydec/2 to  3.1415, ydec/2 nohead lt -1
set label "\34" at 3.1415, -ydec center

set arrow from  -3.1415, -ydec/2 to  -3.1415, ydec/2 nohead lt -1
set label "-\34" at -3.1415, -ydec center

```

```
set arrow from  0, 0  to pasx, pasx lt 1
set arrow from  0, 0 to  -pasx, -pasx lt 1

set arrow from  -xdec/3, 1 to  xdec/3, 1 nohead lt -1
set arrow from  -xdec/3, -1 to  xdec/3, -1 nohead lt -1

set arrow from  -1.57, ymax to  -1.57, ymin nohead lt 0
set arrow from  1.57, ymax to  1.57, ymin nohead lt 0

plot tan(x) title "tan" w l lt 3 lw 2

# On s'arrête et on attend entrée
pause -1 "maintenant va créer un fichier au format png appuyer sur entrée"

set term png
set out "exemple2.png"
rep
set out
set term x11
pause -1 "touche entrée pour sortir"
```

Vous l'aurez compris un commentaire commence par # et `pause -1` interrompt l'éxécution des commandes jusqu'au `return` de l'utilisateur. Pour exécuter ce fichier de commande :

```
gnuplot>load 'tangente.gnu'
```

Si vous êtes perdu, il vous reste :

```
gnuplot>help plot
```

par exemple. Sous gnuplot vous pouvez appeler une commande en la faisant précéder d'un !.

```
gnuplot>! ls -alt
```

1.2.2 Tracé de courbes et surface

Vous pouvez obtenir des tracés, soit à partir d'équations soit à partir de points dont vous donnerez les coordonnées.

Courbe plane d'équation $y = f(x)$

```
gnuplot>plot sin(x)
gnuplot>plot[-15 : 15] cos(x)*x**2
```

On peut bien évidemment définir une fonction :

```
gnuplot>f(x)=cos(x)*x**2
gnuplot>plot[-15 : 15] f(x)
gnuplot>plot[-15 : 5] f(x)*exp(-x)
```

Courbe plane d'équation paramétrique

Pour tracer une belle elipse :

```
gnuplot>set parametric
gluplot>plot [0  : 2*pi] 2*cos(t), sint(t)
```

Ou encore une rosace de Grandy

$$\begin{aligned} x &= a \sin 2\theta \cos\theta \\ y &= a \sin 2\theta \sin\theta \\ & \text{où } 0 \leq \theta \leq 2\pi \end{aligned}$$

Je vous laisse essayer. Et maintenant une cardioïde, ou plus particulièrement un limaçon.

$$\begin{aligned} x &= (a + 2a\cos\theta)\cos\theta \\ y &= (a + 2a\cos\theta)\sin\theta \\ & \text{où } 0 \leq \theta \leq 2\pi \\ & a \in \Re \end{aligned}$$

Pour sortir du mode paramétique `unset parametric`.

Courbe plane en polaire $\rho = f(\Theta)$

L'angle s'appelle t

```
gnuplot>set polar
gluplot>plot [-12*pi : 12*pi] 1/t
```

Pour sortir du mode polaire `unset polar`. Essayons maintenant un escargot $\rho = a\frac{sin(\Theta)}{\Theta}$. Pourquoi pas un bestiaire :

- Spirales d'archimède : $\rho = a\Theta$;
- Paraboles : $\rho = 2p\frac{cos(\Theta)}{sin^2(\Theta)}$;
- Lemniscates de Bernouilli : $\rho = a\sqrt{cos(2\Theta)}$;
- Limaçons de Pascal : $\rho = a\ cos(\Theta) + b$;
- Cardioïdes : $\rho = a(1 + cos(\Theta)$;
- Rosaces : $\rho = asin(n\ \Theta)$.

Surface d'équation de la forme $z = f(x, y)$

Pour tracer la surface d'équation $z = x^2 + y^2$

```
gnuplot> f(x,y)=x**2+y**2
gnuplot> set isosamples 30,40 #réseau de pts 30x40 (10x10 par défaut)
gnuplot> splot f(x,y)
```

En repartant de ce même exemple on peut développer un exemple plus complet avec des lignes de niveaux. Essayer donc le ficheur suivant :

```
reset
f(x,y)=x**2+y**2
set isosamples 60,60
set size 1, 1
set title "x*x+y*y"
set xlabel "x"
set ylabel "y"
set zlabel "z"
splot f(x,y)
```

```
pause -1
set isosamples 80,80
set title "Lignes de niveau de S2"
set xlabel "x"
set ylabel "y"
set view 0,0
set size .75, .75
unset surface
set contour base
unset clabel
set cntrparam levels auto 20
splot f(x,y)
pause -1
reset
r(x,y)=sqrt(f(x,y))
g(x,y)=sin(r(x,y))/r(x,y)
set hidden3d
set isosamples 30,30
splot g(x,y)
pause -1
```

Surface à partir d'équations paramètriques

```
gnuplot> set parametric
gnuplot> set isosamples 30,30
gnuplot> splot cos(u)*cos(v),cos(u)*sin(v),sin(u)
```

1.2.3 Utilisation de données numériques

Supposons pour commencer qu'un programme C++ par exemple a généré un fichier texte (lisible sous un éditeur le fichier suivant) :

```
#an Lapins renards
1970 1230 256
1975 987 320
1980 518 350
1985 780 302
1990 1024 150
1995 1250 281
2000 1580 520
```

censé représenter l'évolution de la population de lapins et de renards de 1970 à 2000 dans votre quartier. La première ligne, commençant par un # est un commentaire. Vous pouvez tracer simplement les points de différentes façons :

```
gnuplot> plot "LapinsRenards.dta"
gnuplot> plot "LapinsRenards.dta" with lines
gnuplot> plot "LapinsRenards.dta" with steps
gnuplot> plot "LapinsRenards.dta" with impulse
```

Pour avoir l'évolution des deux populations on peut procéder de la façon suivante :

```
gnuplot> plot [*:*] [0:1600] "LapinsRenards.dta" title "Lapins" with linespoints
#intervalle qcq sur x de 0 à 1660 sur y
gnuplot> replot "LapinsRenards.dta" using 1:3 title "Renards" with linespoints
#sur la même courbe colonnes 1 et 3
```

1.2.4 C++ et Gnuplot

Nous abordons ici le couplage de C++ et Gnuplot. Il se fait le plus souvent par l'intermédiaire de fichiers, c'est donc dans ce cas que nous nous plaçons. Pour commencer nous allons survoler les entrées, sorties et fichiers en C++.

Dans presque tous les cas vous trouverez en début de programme :

```
#include<iostream>
using namespace std;
```

qui demande l'inclusion du fichier d'entête `iostream` vous permettant de manipuler les différents objets, méthodes et fonctions d'entrée sorties.

Écrire dans un fichier

Pour pouvoir écrire dans un fichier, il faut ajouter l'inclusion :

```
#include<fstream>
```

et ouvrir le fichier en écriture par :

```
ofstream fichier("donnees.dta",ios::out);
```

`donnees.dta` représente le nom (relatif ici) de votre fichier au niveau de votre système d'exploitation et `fichier` sera ensuite l'objet que vous manipulerez représentant le fichier au niveau de votre programme. `ios : :out` précise que le fichier est ouvert en écriture. Une fois votre fichier ouvert, vous écrivez dedans comme vous le faites sur `cin`. Supposons que vous vouliez écrire la valeur de `x`.

```
fichier << x;
```

Ensuite une fois les écritures terminées, vous devez fermer le fichier.

```
fichier.close();
```

Un exemple plus complet :

```
#include <iostream>
#include <fstream>

using namespace std;

int main()
{
        ofstream fichier("test.txt", ios::out | ios::trunc);
        //déclaration du flux et ouverture du fichier

        if(fichier)  // si l'ouverture a réussi
        {
            // instructions
                fichier.close();  // on referme le fichier
        }
        else  // sinon
                cerr << "Erreur à l'ouverture !" << endl;

        return 0;
}
```

Les modes d'ouverture sont :

- `ios::out` (pour output) : spécifie qu'on ouvre le fichier en écriture. Obligatoire - mais par défaut - quand on utilise un objet `ofstream`.
- `ios::app` (pour append = ajouter à la suite) : lorsqu'on ouvre le fichier en écriture, on se trouve à la fin pour écrire des données à la suite du fichier (sans effacer le contenu, s'il y en a un). Avec ce mode d'ouverture, à chaque écriture, on est placé à la fin du fichier, même si on se déplace dans celui-ci avant.
- `ios::trunc` (pour truncate = tronquer) : lorsqu'on ouvre le fichier en écriture, spécifie qu'il doit être effacé s'il existe déjà, pour laisser un fichier vide.
- `ios::ate` (pour at end) : ouvre le fichier en écriture et positionne le curseur à la fin de celui-ci. La différence avec `ios::app` est que si on se repositionne dans le fichier, l'écriture ne se fera pas forcément à la fin du fichier, contrairement à `ios::app`.

Lecture dans un fichier

On ajoute comme dans le cas de l'écriture

```
#include<fstream>
```

et le fichier est ouvert en lecture par :

```
ifstream fichier("donnees.dta",ios::in);
```

Pour lire :

```
fichier >> x;
```

```
#include <iostream>
#include <fstream>

using namespace std;

int main()
{
        ifstream fichier("test.txt", ios::in);  // on ouvre le fichier en lecture

        if(fichier)  // si l'ouverture a réussi
        {
            // instructions
                fichier.close();  // on ferme le fichier
        }
        else  // sinon
                cerr << "Impossible d'ouvrir le fichier !" << endl;

        return 0;
}
```

Lecture - écriture dans un fichier

En utilisant `fstream`, on ouvre en lecture et en écriture un fichier. Le fonctionnement est le même que pour `ifstream` ou `ofstream`.

Le prototype pour utiliser cette méthode d'ouverture est :

```
fstream flux("fichier.extention", ios::in | ios::out | [ios::trunc | ios::ate]);.
// soit ios::trunc ou ios::ate
```

Avec ce mode d'ouverture, le fichier doit exister ! Le `ios::in | ios::out` est obligatoire pour bien spécifier que l'on ouvre le fichier en lecture **et** en écriture. Comme le fichier est ouvert en écriture et qu'il existe déjà, il faut rajouter soit ios : :ate pour ne pas effacer le contenu, soit `ios::trunc` pour l'effacer.

```
#include <fstream>
#include <string>
using namespace std;

int main()
{
      string mon_fichier = "test.txt";
      // je stocke dans la chaîne mon_fichier le nom du fichier à ouvrir
      ifstream fichier(mon_fichier.c_str(), ios::in);
      // On remarquera l'utilisation de c_str() qui renvoie
      // un pointeur vers un tableau de caractères comme en C
      if(fichier)  // si l'ouverture a réussi
      {
            // instructions
              fichier.close();  // je referme le fichier
      }
      else  // sinon
              cerr << "Erreur à l'ouverture !" << endl;

      return 0;
}
```

Pour lire un fichier, il y a différentes méthodes qu'il faut employer en fonction de ce que l'on veut faire :

- `getline(flux, chaineDeCaractères)` : pour lire une ligne complète ;
- `flux.get(caractère)` : pour lire un caractère ;
- `flux >> variable` : pour récupérer à partir du fichier jusqu'à un délimiteur (espace, saut à la ligne, ...).

Attention les espaces, retours à la ligne, retours chariot, ... sont considérés comme des caractères !

Pour écrire dans un fichier :

- `flux << élémentQuelconque` : écrit dans le fichier un élément quelconque (string, int, ...) ;
- `flux.put(caractère)` : écrit un seul caractère dans le fichier.

Formater les sorties

Pour cela il faut ajouter l'inclusion

```
#include<iomanip>
```

Vous pouvez manipuler une notation standard, fixe ou scientifique :

```
cout << x << endl; // affiche en notation standard
cout << fixed; // passe en notation fixe pour les cout suivants
cout << scientific; // passe en notation scientifique pour les cout suivants
cout.setf(ios_base::floatfield); // revient en notation standard
```

De la même façon, vous pouvez fixer la précision :

```
n=cout.precision(); // stocke la précision actuelle dans la variable entière n
cout << setprecision(15); // fixe la précision à 15 pour les "cout" suivants
cout << setprecision(n); // revient à la précision initiale
```

Et pour finir vous pouvez également fixer la largeur minimale d'affichage :

```
cout << setw(30) << x << endl; // impose une largeur minimum de 30 caractères
```

Programme C++ et Gnuplot

On trace la fonction :

$$y = \frac{1}{1+x^2}$$

```
#include<iostream>
#include<fstream>
#include<stdlib.h>
const int N=100;
using namespace std;
int main()
{
  int i; double x,y,dx=0.05;
  ofstream donnees("courbe.res",ios::out);
  for(i=-N/2; i<=N/2; i++) {x=i*dx; y=1./(1+x*x); donnees << x << " " << y << endl;}
  donnees.close();
  ofstream commande("courbe.gnu",ios::out);
  // constitution du fichier contenant les commandes Gnuplot
  commande << "plot 'courbe.res' with lines" << endl;
  commande << "pause -1" << endl;
  commande << "set term png" << endl;     // utile seulement si on veut conserver
                                          // la courbe dans un fichier
  commande << "set output 'courbe.png'" << endl;
  commande << "replot" << endl;
  commande.close();
  system("gnuplot courbe.gnu"); // exécution du fichier de commandes Gnuplot
  system("rm courbe.res courbe.gnu");
  // effacement des fichiers courbe.res et courbe.gnu
  return 0;
}
```

Exercice 5

I De même que pour les courbes, Gnuplot peut tracer des surfaces à partir d'un fichier de points. On suppose que la surface est définie par une fonction $z = f(x, y)$ et que les valeurs de x et y forment un réseau rectangulaire régulier dont les points sont indicés par i pour x et j pour y. Il suffit alors d'écrire un fichier dans lequel figurent uniquement les valeurs de z, rangées dans le bon ordre.

I.1 Écrire le programme C++ correspondant pour la fonction $z = sin(x)cos(3y)$.

I.2 Tracer cette même fonction directement avec Gnuplot !

Exercice 6

I Imaginez que vous vouliez connaître l'évolution d'une population de bactéries dans une boite de Petri, vous avez deux approches possibles : l'approche expérimentale ou l'approche par modélisation et simulation. On va bien sur retenir la deuxième. Pour cela on utilise le modèle de Verhulst qui considère que plus la taille de la population augmente, plus son taux de natalité diminue et son taux de mortalité augmente. D'autre part, lorsque les populations sont de petites tailles, elles ont tendance à croître. Cela se traduit par l'équation différentielle suivante traduisant l'évolution de la population :

$$\frac{dx}{dt} = x\,(n(x) - m(x))$$

Avec x taille de la population, $m(x)$ taux de mortalité, $n(x)$ taux de natalité.

Le temps est alors ici continu. On peut transformer en temps discret le modèle. Nous considérerons donc la suite logistique suivante avec les paramètres bornés ci-dessous.

$$\begin{cases} x_{n+1} = \mu x_n(1 - x_n) \\ x_0 \in [0, 1] \\ \mu \in [0, 4] \end{cases}$$

I.1 Écrivez un programme C/C++ qui permet d'étudier la suite logistique.

I.2 En utilisant votre programme, que vous devrez éventuellement modifier, et gnuplot tracer $x(t)$ pour $\mu = 2$ et $x_0 = 0.2$. Que constatez vous ? Quelle valeur remarquable appelée point fixe, obtenez vous pour $x(t)$ avec t «suffisamment grand ». Faites varier x_0, que constatez vous ?

I.3 Continuons cette exploration de l'espace des paramètres en fixant maintenant $\mu = 3.1$ et $x_0 = 0.2$. Que constatez vous ? Pouvez vous déterminer un point fixe ? En conservant la valeur de μ et en faisant varier x_0 observez vous plusieurs points remarquables (attracteurs).

I.4 Étudiez le comportement de $x(t)$ pour $\mu \in [3.4, 3.5]$ et $x_0 = 0.2$. À nouveau que constatez vous ?

I.5 Modifier votre programme afin qu'il détermine les attracteurs.

I.6 Tracer sur un même graphe gnuplot la trajectoire de la suite logistique avec comme valeur initiale $\mu = 4$ et $x_0 = 0.2$ et $x_0 = 0.2000000001$. Décrivez les trajectoires qui mettent en évidence le chaos.

I.7 En utilisant ce que vous avez vu précedemment modifier votre programme pour tracer le diagramme de Feigenbaum ou diagramme de bifurcation de façon à reporter pour les différentes valeurs de μ les valeurs des attracteurs.

Exercice 7

I On se propose de réaliser les différentes classes nécessaires pour intégrer une fonction d'une variable définie dans $\mathbb{R}$. Les méthodes d'intégration numérique sont utilisées en général lorsque trouver la primitive d'une fonction f est compliqué ou qu'elle est inconnue (la primitive). Un autre cas peut également se présenter lorsque f n'est connue que par points. Pour notre problème, on considère uniquement les intégrales du type :

$$I = \int_a^b f(x)\,dx$$

où $[a, b]$ est un intervalle fini de $\mathbb{R}$ et f est continue sur $[a, b]$.

I.1 Définir une classe `IntegrationTrapeze` qui permette de calculer l'intégrale d'une fonction réelle d'une variable réelle sur un intervalle $[a, b]$ en utilisant la méthode des trapèzes. On subdivise l'intervalle $[a, b]$ en n sous-intervalles égaux et on approche l'intégrale de la fonction sur la subdivision $[x_i, x_{i+1}]$ par le trapèze passant par $(x_i, 0), (x_{i+1}, 0), (x_{i+1}, f(x_{i+1}))$, et $(x_i, f(x_i))$.

I.2 Tester la classe `IntegrationTrapeze` pour $\int_0^{\Pi/2} sin(x)\, dx$ et pour $\int_1^9 \frac{1}{x}\, dx$.

I.3 L'erreur d'approximation commise dans la méthode précédente dépend de la largeur h de chaque subdivision. On pourrait penser que plus h est petit, plus l'approximation est bonne. Malheureusement, on augmente le nombre d'opérations de manière importante et on obtient un cumul d'erreur d'arrondis non négligeable. Le problème est donc de savoir quel est le bon choix pour h ? On se base sur le principe suivant :
- h doit être petit sur les subdivisions où la fonction à intégrer varie beaucoup ;
- h peut-être plus grand pour les subdivisions où la fonction à intégrer varie peu.

On applique le procédé suivant : pour un intervalle donné $[a, b]$, on effectue deux calculs de l'intégrale le premier avec 10 subdivisions le second avec 5. Si la différence obtenue entre ces deux calculs est faible, il n'est pas utile d'aller plus loin et on garde comme résultat le premier calcul. Si la différence entre ces deux calculs est importante, on subdivise l'intervalle $[a, b]$ en deux sous-intervalles et on réapplique le processus de calcul initial. Écrire la classe correspondante.

I.4 Écrire un programme pour tester cette méthode adaptative. Vous pourrez, par exemple, tester cette méthode pour $f(x) = \frac{sin(x)}{x}$, $\int_0^{10*\Pi} f(x)\, dx$. Vous prendrez en compte la singularité en 0, en fixant $f(0) = 1$. Comparez vos résultats avec la méthode des trapèzes.

I.5 A l'aide de gnuplot tracer la fonction ainsi que les trapèzes utilisés pour intégrer la fonction entre $[a, b]$ dans le cas des deux méthodes.

I.6 Pour ceux qui sont à l'aise avec C++ reprendre votre programme, utiliser l'héritage et définissez une une classe `IntegrationRectangle`. La classe mère devra en particulier permettre de tracer la fonction avec Gnuplot et le découpage géométrique.

Chapitre 2

C++ langage orienté objet

Le langage C++ est un langage impératif orienté objet. Il hérite du C avec des fonctionnalités supplémentaires comme :

- les déclarations reconnues comme instructions ;
- les opérateurs `new` et `delete` pour la gestion d'allocation mémoire ;
- les références ;
- les fonctions inline ;
- le mot clé `const` pour définir des constantes ;
- les paramètres par défaut dans les fonctions ;
- les espaces de noms ;
- les classes, ainsi que tout ce qui y est lié : l'héritage, les fonctions membres, les fonctions membres virtuelles, les constructeurs et le destructeur ....
- le polymorphisme ;
- l'opérateur de résolution `::` ;
- la librairie STL ;
- la surcharge des opérateurs ;
- les templates ;
- la gestion d'exceptions ;
- ....

Nous ne verrons pas tous ces aspects, mais nous en développerons un certain nombre dans le cadre de ce chapitre.

2.1 Références et surcharge d'opérateurs

Exercice 8

I On se propose de développer un exercice sur l'arithmétique d'intervalles[1]. On considère donc un intervalle $[m, n]$ avec m et n des `double` comme un type arithmétique sur lequel on peux effectuer des opérations. Soit donc deux intervalles $[a, b]$ et $[c, d]$, on définit :

- l'addition $[a, b] + [c, d] = [a + c, b + d]$;
- la soustraction $[a, b] - [c, d] = [a - d, b - c]$;
- la multiplication $[a, b] * [c, d] = [\min(ac, ad, bc, bd), \max(ac, ad, bc, bd)]$;

[1] Exercice emprunté à C. Milour

– la division

$$[a,b]/[c,d] = [a,b] * (1/[c,d]) \text{ avec } \begin{cases} [1/d, 1/c] & \text{si} \quad 0 < c \\ \text{indéfini} & \text{si} \quad c \leq 0 \leq d \\ [1/c, 1/d] & \text{si} \quad d < 0 \end{cases}$$

Un tel type numérique peut être utile pour propager des incertitudes dans des calculs. Par exemple, combien cela me coûtera-t-il d'acheter entre 3 et 5 kilos de cerises, pour un prix compris entre $1,5$ et $2,5$ euros le kilo ? En arithmétique d'intervalles, le résultat est donné immédiatement par un produit d'intervalles : $[3,5]$ pour la quantité multiplié par $[1.5, 2.5]$ pour le prix au kilo donnent $[4.5, 12.5]$ et je vais donc dépenser entre 4.5 et 12.5 euros. L'intérêt de l'arithmétique d'intervalles est évidemment plus évident lorsque les calculs effectués sont plus complexes... En pratique, pour un coût à peine double ou triple en moyenne du coût d'un calcul scalaire, elle offre un encadrement strict du résultat en fonction des marges d'erreurs sur les données ou les coefficients, et ceci quelle que soit la complexité des transformations effectuées. Ce n'est vraiment pas cher payer. Il y a bien entendu des limites : dans certains cas l'encadrement n'est pas aussi fin qu'on le désirerait. Si vous êtes curieux, allez donc voir le site web d')I(nterstices.

I.1 Ecrire une classe d'arithmétique d'intervalle `Intervalle`. On doit pouvoir effectuer les opérations traditionnelles : +, -, *, /, =, +=, -=, *=, /=, ==, !=, «, ». Vous penserez en particulier aux différents contructeurs et au - unaire. Les opérateurs de comparaisons verifiront les égalités ou les inégalités sur les intervalles, mais également l'appartenance (respectivement la non appartenance) d'un `double` pour l'opérateur == (resp. !=).

I.2 Compléter votre classe de façon à pouvoir écrire :

```
1  Intervalle sqrt(const Intervalle &x)
2  {
3    const Intervalle::Borne precision = 1. E-12;
4    Intervalle a=x ;
5    Intervalle f= a*a - x ;
6    while(fabs(a * a - x) > precision)
7      {
8       a = (a + x/a) / 2;
9      }
10 #ifdef TRACE
11   cout << "␣sqrt(x)␣=␣" << a << endl ;
12 #endif
13  return a ;
14 }
```

I.3 Ecrire un programme complet qui teste votre classe et la fonction `sqrt()`.

I.4 On peut par exemple aussi calculer la résistance équivalente de deux résistances placées en parallèles :

$$R = \frac{R_1 * R_2}{R_1 + R_2}$$

avec $R_1 = 2500\Omega \pm 2\%$ et $R_2 = 4000\Omega \pm 4\%$.

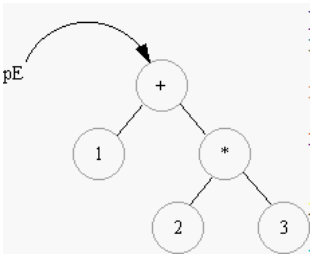

FIG. 2.1 – Arbre représentant l'expression arithmétique : $1 + 2 + 3$

2.2 Héritage et virtualité

Exercice 9

I On s'intéresse à des expressions arithmétiques simples, représentées dans un premier temps sous forme d'arbres et ensuite sous forme d'un graphe. Par exemple la figure 2.1 illustre l'expression $1 + 2 * 3$. Chaque nœud de l'arbre est lui même une expression. Vous pourrez compliquer le problème en complétant les expressions en introduisant des variables et des fonctions et chercher à éveluer ces expressions. :

I.1 Créer une classe abstraite `Expression` avec en particulier une méthode `eval()` qui retourne l'évaluation de l'expression. Pourquoi cette classe doit-être abstraite ?

I.2 Peut-on créer une instance de cette classe ? Commenter le programme suivant, vous semble t-il correct, compile t-il ? Expliquez pourquoi.

```
1  #include <iostream>
2  #include "Expression.h"
3
4  using namespace std;
5
6  // analyseur syntaxique, qui retourne l'expression.
7  extern Expression* f();
8
9  int main() {
10   Expression * e = f();
11   cout << e->eval() << endl;
12   return 0;
13 }
```

I.3 Pour manipuler l'expression $1 + 2 * 3$, vous devez pouvoir représenter chacun de ses éléments : constantes, somme et produit. Créer à cet effet trois classes dérivées de `Expression`, respectivement `Constante`, `Somme` et `Produit`.

I.4 On veut pouvoir afficher correctement nos expressions de façon à pouvoir écrire :

```
1  #include <iostream>
2  #include "Expression.h"
3
```

```
4  using namespace std;
5
6  // analyseur syntaxique, qui retourne l'expression.
7  extern Expression* f();
8
9  int main() {
10   Expression * e = f();
11   cout << *e << " = " << e->eval() << endl;
12   return 0;
13 }
```

et obtenir à l'exécution

```
$ mainExpression
(1 + (2 * 3)) = 7
$
```

Pour réaliser cela chaque classe dérivée d'`Expression` doit avoir sa propre formatée, malheureusement l'opérateur d'écriture dans un flux est une fonction statique et pas une méthode er ne peut pas être directement polymorphique. Comment résoudre ce problème ? Compléter vos classes.

I.5 Votre affichage est pour le moment infixe, ajoutez la possibilité de d'avoir un affichage préfixe et postfixe.

I.6 Nous nous sommes pas péroccupé de la gestion de mémoire, toutes les instances manipulées étaient supposées statiques. Cette approche est irréaliste. Par exemple si l'arbre d'expressions est produit par un analyseur syntaxique, il faudra d'une manière ou d'une autre allouer dynamiquement les nœuds de cet arbre. On peut mettre en place une stratégie selon laquelle chaque nœud non-feuille de l'arbre est propriétaire de ses enfants, et qu'il détruit ces derniers lors de sa propre destruction. Cela implique de n'utiliser que des instances dynamiques. Il n'est donc pas possible de les mélanger avec des instances automatiques ou statiques. Modifier vos classes en conséquence.

I.7 La gestion simple proposée à la question précédente convient lorsque les expressions sont représentées par des arbres simples. Elle est inadaptée aux graphes. Par exemple l'expression 2+2*2 peut être réalisée par un graphe acyclique orienté (voir figure 2.2), ou DAG (directed acyclic graph). Manifestement la destruction récursive de la question précédente n'est pas adaptée : la `Constante` 2 serait détruite trois fois... L'utilisation du comptage de référence permet de pallier ce problème. L'idée est de maintenir pour chaque `Expression` un compte des pointeurs qui la désignent et de ne la détruire effectivement que lorsque ce compte passe à zéro. Ce comportement peut être implanté par une classe de base. Modifier les classes en conséquence.

2.3 Généricité et exceptions

Exercice 10

I Dans le premier exercice du ğ 2.1 ajouter la généricité sur le traitement des intervalles ainsi que la gestion des exceptions.

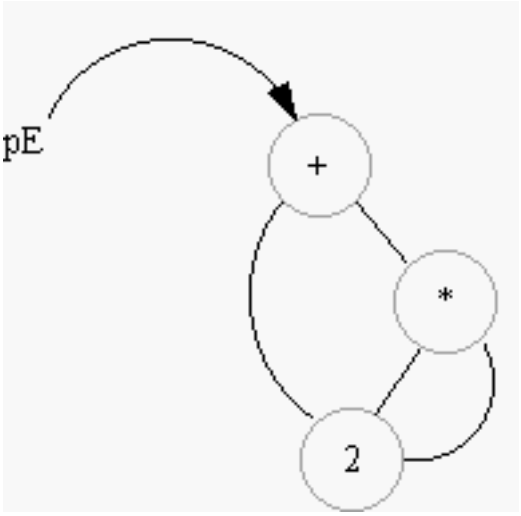

FIG. 2.2 – DAG de l'expression $2 + 2 * 2$

2.4 Design pattern et STL

On cherche à mettre en œuvre un design pattern communément appelé « singleton». Ce design pattern est utile lorsqu'il est important pour certaines classes d'avoir exactement une instance dans un programme. Par exempleă :

- Un point d'accès à une ressource extérieure unique, comme le spooler des imprimantes, le pilote de la carte son ou une base de donnée particulière.
- Un point d'accès à une ressource interne unique, comme une fenêtre graphique spéciale, ou un thread spécial.

A priori on peut créer une variable globale, mais rien n'empêche d'instancier la classe ailleurs dans le programme. Le design pattern singleton consiste à rendre la classe responsable de son unique instance, et de l'unique accès à cette instance.

Au niveau de l'implémentationăil suffit deăprévoir :

- Une méthode statique publique, qui constitue l'unique point d'accès à l'instance, en retournant un pointeur sur l'instance. Cette méthode crée l'instance la première fois qu'elle est appelée.
- Un attribut statique privé, qui est en fait le pointeur sur la future instance. Le constructeur de la classe privée, de façon à ce qu'il ne puisse être appelé hors de la classe.

Cela a pour conséquence que la classe contrôle l'unique point d'accès à son unique instance et c'est une très bonne alternative aux variables globales *polluantes*. Il est bien sur possible de modifier ce design pattern de façon à n'autoriser qu'un nombre défini d'instances (> 1).

Exercice 11

I Définir une classe liste de `long` ne permettant de créer qu'une instance unique. Permettant de réécrire le code suivant plus proprement en n'utilisant pas de variable globale. Vous devez également utiliser la librairie STL pour la manipulation de liste.

```
1 UneBelleListe maliste;
```

```
2
3  int main()
4  {
5    maliste.addElement(12);
6    if(maliste.existeValeur(12) )
7      maliste.supprimeElement(53);
8    return 0;
9  }
```

Exercice 12

I En programmation il est très courant de représenter les données dans un arbre. On distingue alors deux type de nœudsă : Les nœuds avec des enfants, appelés nœud composite. Les nœuds sans enfant, appelés nœud feuille (leaf en anglais). Cette structure d'arbre est constamment utilisée dans les logiciels graphique, ou dans tout autre logiciel de gestion. Une approche simpliste consiste à définir une classe de base pour les nœuds *composite* et une autre pour les nœuds *leaf*. Cependant le client souhaite manipuler tous les nœuds de la même façon. Par exemple si le client a l'opération `Affiche()` sur un nœud graphique, il s'attend a ce que cette opération puisse être appelée sur un nœud *composite*, auquel cas la demande d'affichage sera transférée aux enfants du nœud *composite*. La clé du design pattern consiste à définir une classe abstraite qui représente à la fois les nœuds *composite* et les nœuds *leaf*.

I.1 Définir une classe de base `CNoeud` qui présente l'interface générale à tous les nœuds. Cette classe peut implémenter les méthodes non concernées par la liste des nœuds fils, et n'avoir aucun code fonctionnel dans les méthodes concernées par la liste des nœuds fils, qui sont alors virtuelles. Cette classe peut optionnellement proposer des méthodes pour accéder au nœud parent et ainsi pouvoir remonter la hiérarchie.

I.2 Définir une classe `CLeaf` représentant les nœuds *leaf*. Cette classe ne redéfinie pas les méthodes virtuelles concernées par la liste des nœuds fils et garde ainsi le code non fonctionnel.

I.3 Définir une classe `CComposite` représentant les nœuds *composite*. Cette classe redéfinie les méthodes virtuelles concernées par la liste des nœuds fils et possède une collection stockant les références vers les nœuds fils.

II L'objectif est de décrire l'assemblage d'une carte mère d'ordinateur, puis d'avoir le prix et la description de la carte. Précisons juste que le coût de l'assemblage est de 45 euros et que chaque composant aăun prix et une description.

II.1 Tester avec le programme principal suivant en ayant défini les élements nécessaires.

```
1  int main()
2  {
3   // assemble l'ordinateur
4   CCarteMere CarteMere("Carte␣mère:ABIT␣KR7-A␣133␣␣",175);
5   CChip Chip("Chip:␣Celeron␣1,2␣GHz␣␣␣",130);
6   CRAM RAM("RAM:␣SD-RAM␣512M␣PC␣133␣␣␣",196);
7   CCarteGraphique CarteGraphique("Carte␣graphique:␣Ge␣Force␣4␣␣␣",
8                                  575);
9
10  CarteMere.Add(&Chip);
```

FIG. 2.3 – Application Qt de visualisation d'image fixe

```
11    CarteMere.Add(&RAM);
12    CarteMere.Add(&CarteGraphique);
13
14    // calcule les prix selon les tarifs
15    string Desc = CarteMere.Desc();
16    long Prix = CarteMere.Prix();
17    return 0;
18  }
```

2.5 Qt

Exercice 13

I On cherche à écrire une application Qt de visualisation d'images fixes. L'application devra ressembler à la figure 2.3.

II Définir une classe `ImageViewer`, qui correspond à la fenêtre principale. Cette classe doit hériter de `QMainWindow` et elle possède une barre de menus comportant le menu *File*, comportant lui-même deux sous-menus : *Open File* et `Quit`. Au centre vous devrez ajouter une zone d'affichage d'image sous forme d'objet `ImageCanvas` que vous définirez par la suite ainsi que deux `QLabel` permettant de connaître l'image chargée ainsi que les coordonnées de la souris lorsque celle-ci se déplace sur l'image et les coordonnées couleurs RGB du pixel correspondant. Pour organiser les différents composants graphiques, vous les rangerez dans un layout `QVBoxLayout`.

Lorsque le sous-menu OpenFile sera sélectionnée, un nom de fichier sera saisi en utilisant un objet de la classe `QFileDialog`. Pour relier l'action de sélection du sous-menu *Open File* à la méthode d'ouverture du fichier

`openImage()` que vous aurez défini, il est nécessaire de spécifier que cette méthode est un slot de la classe appropriée.

III Définir la classe `ImageCanvas`. Elle hérite de `QWidget` et possède une image `QImage` comme variable d'instance. L'image est chargée à partir d'un fichier et sera donc chargée dans cette variable. Afin que votre fenêtre se redimensionne aux dimensions de l'image chargée et affiche l'image définissez une méthode :

```
1 void ImageCanvas::paintEvent ( QPaintEvent *event) {...}
```

Le principe consiste à créer un object de type `QPainter` de père `ImageCanvas` dans lequel l'image est dessinée à l'aide de la méthode :
`QPainter : :drawImage()`

IV Compléter vos classes de façon à lire la position de la souris et les coordonnées couleurs RGB du pixel correspondant. C'est votre classe `ImageCanvas` qui va gérer les événements associés aux déplacements de la souris. En appelant la méthode `setMouseTracking()`, héritée de la classe `QWidget`, avec une valeur booléenne `TRUE`, cela active la réception des événements de la souris. Il reste alors définir la méthode :
`void ImageCanvas : :mouseMoveEvent( QMouseEvent *e)`
`...` pour traiter ces événements. La classe `QMouseEvent` comporte des méthodes d'accès aux coordonnées de la souris. On affichera les coordonnées de la souris dans un des `QLabel` de la fenêtre principale ainsi que les composantes rouge, verte et bleue de l'image si celle-ci existe et si la souris est effectivement positionnée sur l'image.