TD/TP 2 et 3 — corrigés partiels

1 Résolution d'une équation du second degré

Écrire un programme qui lit trois réels a, b, c et affiche les solutions de l'équation $ax^2+bx+c=0$, lorsqu'elles existent. On rappelle que les solutions sont données par l'expression

$$\frac{-b \pm \sqrt{b^2-4ac}}{2a}$$

Lorsqu'il y a des solutions x_1 et x_2, calculer $ax_1^2+bx_1+c$ et $ax_2^2+bx_2+c$. Trouve-t-on bien 0 ?

On calcule $\Delta = b^2 - 4ac$. Lorsque $\Delta < 0$, on signale qu'il n'y a pas de solution. Lorsque $\Delta \geq 0$, on affiche les deux solutions x_1 et x_2 (on a $x_1 = x_2$ lorsque $\Delta = 0$).

```
#include <math.h>
#include <stdio.h>

int main () {
    float a, b, c, delta, rdelta, x1, x2, z1, z2;
    printf("Résolution de l'équation a x^2 + b x + c = 0.\n");
    /* Il faudrait vérifier que chaque scanf réussit, et afficher un
       message d'erreur sinon. */
    printf("a = "); scanf("%f", &a);
    printf("b = "); scanf("%f", &b);
    printf("c = "); scanf("%f", &c);
    delta = b * b - 4 * a * c;
    if (delta < 0) {
        printf("Pas de solution réelle.\n");
    } else {
        rdelta = sqrt(delta);
        x1 = (- b - rdelta) / (2 * a);
        x2 = (- b + rdelta) / (2 * a);
        printf("Solutions : x1 = %f, x2 = %f\n", x1, x2);
        z1 = a * x1 * x1 + b * x1 + c;
        z2 = a * x2 * x2 + b * x2 + c;
        if (z1 == 0) printf("z1 == 0\n"); else printf("z1 == %f !\n", z1);
        if (z2 == 0) printf("z2 == 0\n"); else printf("z2 == %f !\n", z2);
    }
    return 0;
}
```

Voici un exemple de session du programme :

```
Résolution de l'équation a x^2 + b x + c = 0.
a = 1
b = 50
c = 1
Solutions : x1 = -49.979992, x2 = -0.020008
z1 == -0.000004 !
z2 == -0.000004 !
```

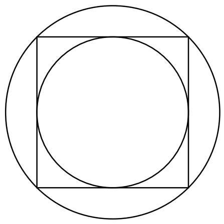
Les nombres x_1 et x_2 sont irrationnels, ils ne sont donc pas représentés exactement en machine. Avec le type `float`, le C standard ne garantit même pas deux décimales de précision dans les calculs et les résultats intermédiaires ; la norme IEEE des nombres flottants spécifie environ 7 décimales de précision. Les « solutions » calculées `x1` et `x2` sont donc des valeurs approchées des vraies solutions mathématiques ; en les réintroduisant dans l'équation, on ne peut en général pas espérer trouver 0 à mieux que $b10^{-7}$ près environ (pour des flottants IEEE).

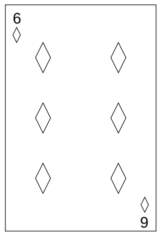
Nous ne discuterons pas plus précisément la précision des calculs en virgule flottante. La règle générale à retenir est que chaque pas de calcul fait irrémédiablement perdre un peu de précision, ce qui fait que les dernières décimales du résultat finales n'ont aucune signification.

En pratique, lorsqu'on code des calculs sur des nombres flottants (notamment pour des modélisations physiques ou biologiques), il faut en parallèle faire un calcul mathématique pour évaluer la précision du résultat. Si on trouve que 0 décimales sont fiables, c'est que la méthode de calcul est vraiment trop mauvaise ! Pour les calculs financiers, on utilise toujours des nombres entiers (nombre de millièmes de centimes par exemple, avec des règles d'arrondi bien précises).

2 Premiers dessins

Pour chacune des figures suivantes, écrire un programme qui dessine la figure dans une fenêtre graphique. Le programme devra se terminer lorsque l'utilisateur clique dans la fenêtre.





Pour dessiner la carte à jouer, on fait commence par définir une fonction auxiliaire qui dessine un motif « carreau » à une position et d'une taille passées en paramètre.

```
#include <math.h>
#include <graphics.h>

/* Dessine un carreau centré en (cx, cy) de demi-largeur d */
void carreau (int cx, int cy, int d) {
    gr_moveto(cx, cy - 2 * d);
    gr_lineto(cx - d, cy);
    gr_lineto(cx, cy + 2 * d);
    gr_lineto(cx + d, cy);
    gr_lineto(cx, cy - 2 * d);
}

int main () {
    gr_open_graph(" 300x500");
    gr_moveto( 50,  50);
    gr_lineto(250,  50);
    gr_lineto(250, 450);
    gr_lineto( 50, 450);
    gr_lineto( 50,  50);
    carreau(230,  95, 8); gr_moveto(230, 60); gr_draw_char('9');
    carreau(100, 150, 20); carreau(200, 150, 20);
    carreau(100, 250, 20); carreau(200, 250, 20);
    carreau(100, 350, 20); carreau(200, 350, 20);
    carreau( 70, 405, 8); gr_moveto(70, 425); gr_draw_char('6');
    (void) gr_wait_event(BUTTON_DOWN);
    gr_close_graph();
    return 0;
}
```

3 Déplacement d'une tortue

Dans le langage Logo, on peut afficher des dessins à l'aide d'une tortue. *À tout instant, la tortue a une* position *et un* cap. *Pour dessiner, on utilise deux fonctions :*

1. `avance(float l)` *avance d'une longueur* `l` *en laissant une trace ;*
2. `tourne_droite(float a)` *tourne sur place d'un angle* `a` *(en degrés) vers la droite.*

Écrire ces deux fonctions, ainsi qu'une fonction `tourne_gauche`*. Utiliser trois variables globales* `x`*,* `y`*,* `cap` *pour stocker la position et le cap courants de la tortue.*

Afin de pouvoir dessiner des figures qui nécessitent plus d'un trait de crayon, ajouter une variable globale entière `crayon`*. Modifier la fonction* `avance` *pour qu'elle trace un trait si* `crayon` *vaut 1, et qu'elle ne fasse que déplacer la tortue sans dessiner si* `crayon` *vaut 0.*

Voici le début d'une bibliothèque de dessin avec tortue. Le fichier `tortue.c` *définit* les variables globales et les fonctions de la bibliothèque. Le mot-clé `static` devant une définition de variable ou de fonction signifie que celle-ci n'est accessible que depuis `tortue.c`.

```
#include <math.h>
#include <graphics.h>
#include "tortue.h"

#define PI 3.14159265359

int crayon = 0; /* on ne dessine que si crayon != 0 */
static float x = 0, y = 0; /* position courante */
static float cap = 0; /* 0 = vers la droite, pi/2 = vers le haut (comme en maths) */

void avance (float longueur)
{
    gr_moveto(x, y);
    x += longueur * cos(cap);
    y += longueur * sin(cap);
    if (crayon == 0) {
        gr_moveto(x, y);
    } else {
        gr_lineto(x, y);
    }
}

void tourne_gauche (float degres)
{
    float radians = degres / 180 * PI;
    cap += radians;
}
void tourne_droite (float degres) { tourne_gauche(-degres); }
```

Le fichier `tortue.h` contient des *déclarations* des variables et fonctions de la bibliothèque. Le mot-clé `extern` devant une déclaration de variable indique que celle-ci est définie dans une autre unité de compilation : il transforme ce qui serait une définition en une simple déclaration. Si un fichier C inclut `tortue.h`, il a donc accès à la variable `crayon` de type `int`.

```
/* Bibliothèque de dessin à la tortue */
extern int crayon; /* On ne dessine que si (crayon != 0) */
void avance(float longueur); /* Avance de N pixels dans le cap (direction) courant */
void tourne_droite(float degres);
void tourne_gauche(float degres);
```

Pour utiliser la tortue dans un programme, on inclut l'en-tête `tortue.h`. Un exemple est donné plus loin (exercice 4 sur le polygone régulier).

Note sur les programmes qui ont plus d'un fichier source Voici quelques règles concernant le partage des variables et des fonctions entre les fichiers sources.

- La fabrication d'un exécutable à partir du source C passe par deux étapes : la compilation proprement dite (du source `toto.c` au fichier objet `toto.o` : commande Unix `cc [...] -c toto.c`), et le liage (*link*) (rassemblement des fichiers objets en un exécutable : commande Unix `cc [...] -o executable toto.o tutu.o`). Chaque fichier source C est appelé *unité de compilation*.[1]
- Une variable globale et une fonction dont la définition est précédée du mot-clé `static` est locale à l'unité de compilation. Sinon, elle est visible depuis toutes les unités de compilation.
- Une variable ou une fonction définie comme `static` doit être déclarée comme `static`. Une variable ou une fonction qui n'est pas définie comme `static` doit être déclarée sans `static` (éventuellement avec `extern`).
- Pour utiliser une variable définie dans une autre unité de compilation, il faut la déclarer comme `extern` (par exemple « `extern int toto;` »).
- Pour utiliser une fonction définie dans une autre unité de compilation, il faut donner son prototype (par exemple « `int fonction(int argument);` »). Le mot-clé `extern` peut être utilisé mais est en général omis.
- Une variable ou une fonction non `static` ne doit être définie qu'en un seul endroit. Elle peut être déclarée autant de fois qu'on veut.

Quand on écrit un programme un tant soit peu compliqué, il est d'usage d'écrire un ou plusieurs en-têtes (fichiers `.h`) contenant une déclaration de toutes les variables et les fonctions non `static`. Lorsqu'un fichier `.c` fait appel à l'une de ces variables ou fonction, on inclut l'en-tête avec la ligne

```
#include "toto.h"
```

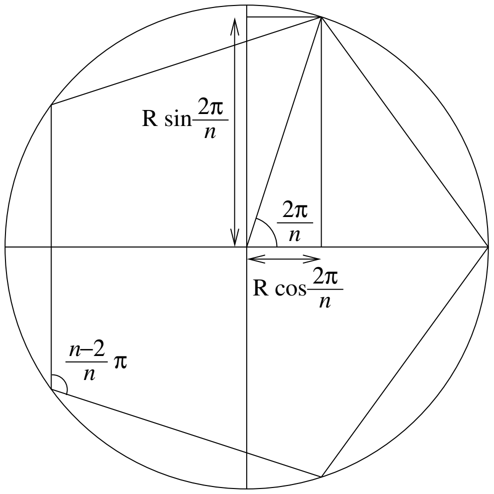
(on utilise des guillemets doubles `"toto.h"` pour les en-têtes dans le répertoire courant, et des crochets angulaires `<foobar.h>` pour les en-têtes des bibliothèques installées). On ne met pas de déclaration dans les fichiers `.c`, pour éviter de se retrouver avec des déclarations incompatibles le jour ou on change quelque chose.

Note finale : en fait, en un sens, tous les programmes regroupent plusieurs unités de compilation, puisqu'il y a celles (cachées) de la bibliothèque standard.

4 Polygones réguliers

Écrire un programme qui lit un nombre entier n et dessine un polygone régulier à n côtés. Utiliser au choix la bibliothèque graphique directement ou la tortue de l'exercice 3.

Première possibilité : une boucle `for (i= 1; i<=n; i++)`, on calcule les coordonnées des sommets avec de la trigonométrie élémentaire, et on tire une ligne de la position courante au sommet suivant. Pour le calcul des coordonnées, voir la figure ci-dessous, pour un polygone régulier à n côtés inscrit dans un cercle de rayon R.



Variante : on calcule les coordonnées des sommets et on les place dans un tableau, puis on utilise la fonction `gr_draw_poly`.

[1] C'est la situation sous Unix. Sous d'autres systèmes, le principe est en général le même, mais les noms des fichiers peuvent changer, par exemple `.obj` pour Windows.

Noter que les fonctions trigonométriques de C prennent des arguments en radians comme il se doit. La bibliothèque standard ne fournissant pas π, on peut utiliser

```
#define PI 3.14159265359 /* il y a assez de décimales pour la précision des dessins */
```

ou encore

```
double pi = 2 * asin(1);
```

Passons maintenant à une méthode avec la tortue (*cf.* l'exercice 3). On n'a pas besoin de calculer les coordonnées de chaque point, mais on n'a plus aussi facilement le choix du centre du polygone.

```
#include <stdio.h>
#include <graphics.h>
#include "tortue.h"

#define TAILLE 500
#define MARGE 10
#define GEOMETRY " 500x500"

void polygone_regulier(int n) {
    double angle = (2.0 / n) * 180.0;
    double cote = TAILLE / n * 2;
    int i;
    crayon = 0;
    avance(TAILLE - MARGE);
    tourne_gauche(90);
    avance(TAILLE / 2);
    tourne_gauche(angle / 2.0);
    crayon = 1;
    for (i = 0; i < n; i++) {
        avance(cote);
        tourne_gauche(angle);
    }
}

int main () {
    int cotes;
    printf("Dessiner un polygone a combien de cotes ? ");
    fflush(stdout);
    scanf("%d", &cotes);
    gr_open_graph(GEOMETRY);
    polygone_regulier(cotes);
    gr_wait_event(KEY_PRESSED);
    gr_close_graph();
    return 0;
}
```

Questions subsidiaires

1. D'où vient la valeur choisie pour la longueur du côté ?
2. Pourquoi la méthode consistant à calculer les coordonnées de chaque sommet est-elle meilleure que celle utilisée pour dessiner avec la tortue ? (Il y a une deuxième raison, en plus de la difficulté à lier la longueur du côté à la taille de la fenêtre. Observer ce qui se passe lorsqu'il y a beaucoup de côtés, par exemple 1000.)

5 Calcul de e

Le nombre e est défini comme la limite de la somme infinie

$$e = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \ldots + \frac{1}{n!} + \ldots$$

Utiliser une fonction `factorielle` définie sur les nombres entiers et à valeurs entières pour calculer une valeur approchée de e. Est-ce la meilleure méthode ?

Le principe de base du calcul est le suivant :

```
int n = 0;
double e = 0;
while (continuer)
    e += 1.0 / factorielle(n);
    n += 1;
```

Remarquer le `1.0` ; sinon on aurait une division entière qui donnerait 0 dès que $n! \geq 2$.

Il reste à trouver un bon test d'arrêt. Remarquons que le terme général $1/n!$ est positif, et décroît lorsque n augmente. Ceci signifie que la variable `e` augmente à chaque tour de boucle, et ce par incréments de plus en plus faible. Comme de plus $1/n!$ tend vers 0 lorsque n est grand, il existe une valeur de n à partir de laquelle $1/n!$ est négligeable devant `e`, au sens où, en valeurs flottantes : `e + 1.0 / factorielle(n) == e`. Bien voir que ceci ne signifie pas que `1.0 / factorielle(n) == 0`...

Avant d'écrire le programme complet, faisons deux remarques. Tout d'abord, nous avons défini une fonction `factorielle` qui renvoie un nombre entier (`int`) parce que la fonction mathématique factorielle renvoie un nombre entier. Cependant, ici, nous n'avons besoin que d'une valeur approchée de $n!$, donc il vaut mieux utiliser le type `double` pour calculer la factorielle. En effet, le type `double` permet en général de représenter des valeurs beaucoup plus grande que `int` (ou même `long int`), quitte à ne stocker qu'une approximation de la valeur entière exacte.

Deuxième remarque, d'efficacité. Le calcul de $n!$ utilise le calcul de $(n-1)!$, il est donc stupide de calculer $n!$ en ayant tout oublié de $(n-1)!$. C'est un cas où utiliser une fonction pré-existante est une mauvaise idée... Nous calculerons donc $n!$ au fur et à mesure, ou en fait $1/n!$.

```
int n = 1;      /* numéro de l'itération */
double a = 1;   /* 1 / n! */
double e = 1;   /* approximation de e */
while (e + a != e)
    e += a;
    n += 1;
    a = a / n;
```

Si on aime les programmes cryptiques, on peut aussi écrire la boucle sous la forme

```
while (e + a != e) e += a /= n++;
```

Question subsidiaire Qu'est-ce qui ne va pas dans le programme suivant :

```
while((e+=a/=n++)!=e);
```