Licence STS Université Lyon 1

LIFAMI

Des exemples d'Applications des Mathématiques et de l'Informatique

Printemps 2023

Alexandre MEYER & Elodie DESSEREE

http://licence-info.univ-lyon1.fr/LIFAMI

GRAPIC
http://licence-info.univ-lyon1.fr/grapic

Table des matières

LIFAMI – Consignes

I. Les 12 commandements de LIFAPI

1. Une fonction renvoie UN résultat et UN seul (pas deux, ni trois, ni quatre).
2. Une procédure ne renvoie rien, elle modifie l'environnement ou ses paramètres passés en Donnée/Résultat.
3. Il faut bien distinguer les paramètres des fonctions/procédures, des variables locales. Une variable locale sert à l'intérieur du sous-programme. Le code appelant n'a pas à savoir que cette variable locale existe.
4. Un paramètre modifié dans la procédure est passé en Donnée/Résultat.
5. Un paramètre consulté dans une fonction ou procédure est passé en Donnée.
6. Renvoyer un résultat pour une fonction ne veut pas dire faire un affichage.
7. Les tableaux en C/C++ sont des exceptions : ils sont toujours passés en Donnée/Résultat, il ne faut pas mettre le &.
8. Un tableau est de taille constante. Le tableau ne peut pas changer de taille. On ne peut pas connaitre la taille d'un tableau, il faut avoir accès à la constante définissant la taille. Un tableau ne peut pas être affecté à un autre tableau : int a[5], b[5] ; a=b ; ou cout<<a ; ne fonctionnent pas.
9. Pour gérer un tableau de taille variable, on déclare un grand tableau de taille constante et on utilise une variable entière pour indiquer combien de cases sont réellement utilisées dans le tableau.
10. Une chaine de caractères est représentée par un tableau de caractères. Le '\0' terminal indique la fin de la chaine de caractères mais pas la fin du tableau. Il y a des caractères présents entre le '\0' et la fin du tableau qui ne seront pas affichés en cas d'affichage. Les règles sont les mêmes que pour les tableaux : pas d'affectation d'un tableau à un autre. Seules exceptions sont l'affichage et la saisie. char a[10], b[10] ; cout<<a ; cin>>a ; sont correctes. Cependant, a=b ; ne fonctionne pas.
11. 'a' est le caractère a. "a" est la chaine de caractères comportant le caractère 'a' et le '\0' terminal donc 2 cases.
12. Le passage d'une structure en paramètre en C/C++ n'est pas une exception comme les tableaux, même si la structure contient un tableau. Si elle est modifiée, il faut la passer en Donnée/Résultat avec le & en C/C++.

II. Consignes pour les intervenants de TD et TP (à lire aussi par les étudiants)

LIFAPI et LIFAMI sont des UE à but éducatif. Les étudiants vous seront reconnaissants de leur montrer des algorithmes jolis, élégants et efficaces. Les artifices de programmation sont à proscrire (du type C++-11 ou C++-14), le correcteur des CCF les comptera comme faux, car trop éloignés d'un objectif pédagogique.

- Pas d'orienté objet, pas de class, pas d'héritage, pas de STL, pas de pointeur, pas de string, pas d'allocation dynamique, pas de printf/scanf, etc.

- Les passages de paramètres se font avec les références

```
void rechercheMin(int t[MAX], int& entier_modifie, int entier_consulte)
```

Pas de pointeur

- Les entrées/sorties se font avec `cout<<"bonjour";` et `cin>>x;`
  Pas de printf / scanf

- Les boucles avec un FOR servent à itérer
  - Un for ressemble toujours à : `for(i=0;i<100;i++)`
  - Pas de *break*, ni de *continue*
- Si vous avez besoin d'une boucle plus compliquée faites un WHILE

- Une fonction devrait dans la mesure du possible ne comporter qu'un seul *return* à la fin

- Les variables ont un nom cohérent et avec du sens.
- Les noms des structures commencent par des majuscules, puis sont en minuscules.
  ```
  struct Camion
  ```
- Les constantes sont entièrement en majuscules.
  ```
  const int CHMAX = 32 ;
  ```

- Les allocations dynamiques ne sont pas au programme des UE :
  pas de malloc / ni de new

- Les chaines de caractères se traduisent par des tableaux de caractères :
  ```
  char nom[32] ;
  ```
  Pas de std ::string

- Certains concepts du langage sont à proscrire car ce sont des concepts inutiles ou trop compliqués pour débuter. Ils viendront plus tard, en LIFAP4 ou M1IF01
  - Pas de STL : ~~vector<int>~~, ~~list<float>~~, etc.

  - Pas de class

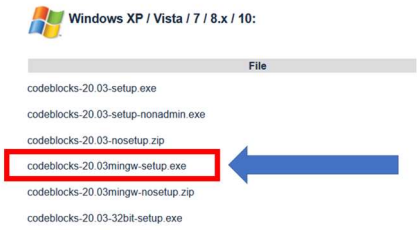
  - Pas de C++-11 pour initialiser les champs d'une structure
    ```
    struct Camion
    {
       int x ~~= 0~~ ;
    ```

LIFAMI – Pour coder dans ce cours …

I. Etape 1 : Codeblocks (Uniquement Windows et Linux)

Codeblocks est déjà installé sur les ordinateurs de l'université.

Pour installer codeblocks sur votre ordinateur personnel, sous Windows, installer le à partir du lien suivant : http://www.codeblocks.org/downloads
Puis "Download the binary release", puis **codeblocks-20.03mingw-setup.exe**



Sous Linux, il faut ajouter un repository (source de paquet) pour avoir la version 17 puis installer codeblocks. Entrer les commandes suivantes

```
sudo add-apt-repository ppa:pasgui
sudo apt update
sudo apt install codeblocks
```

I. Etape 1 : sous MacOS installer XCode et mettre à jour la version de l'OS

- Mettre à jour son mac à la dernière version
  - Cliquer sur le menu pomme en haut à gauche quand vous êtes sur le bureau
  - Cliquer sur Préférence Système
  - Cliquer sur Mise à jour du logiciel
  - Mettre à jour si nécessaire

- Installer la dernière maj de XCode
  - Ouvrez l'application App Store
  - Cliquez sur Mise à jour
  - Mettre à jour si nécessaire

II. Etape 2 : GrAPiC : GRaphics for Algo/Prog In C/C++

GRAPIC est une librairie facilitant l'affichage graphique pour des programmes en C/C++.
http://licence-info.univ-lyon1.fr/grapic

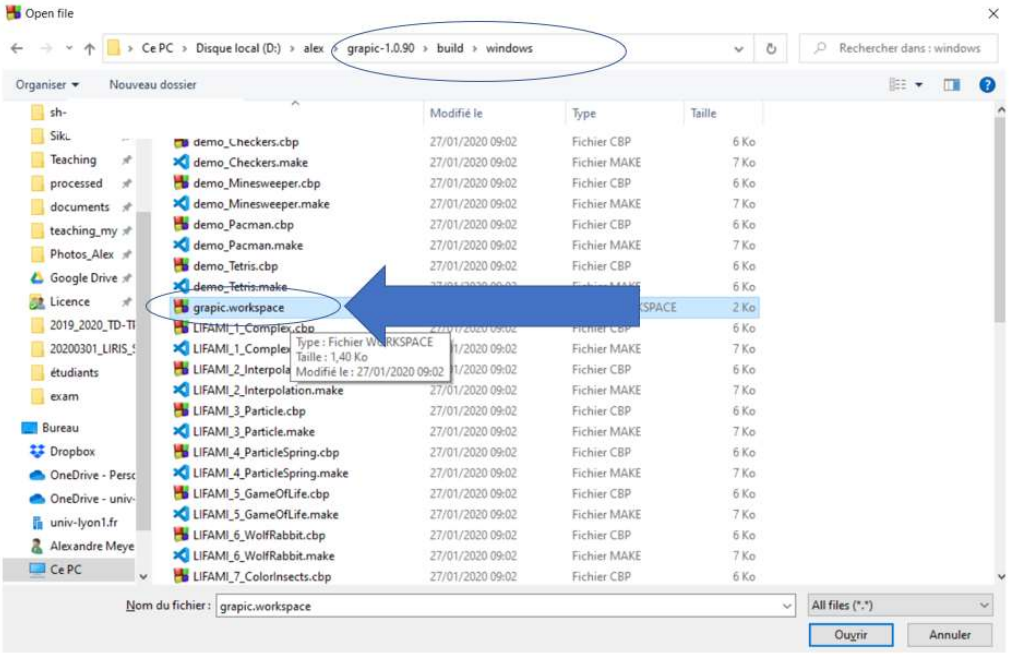
1. Récupérer le fichier .zip qui correspond à votre OS.

2. Décompresser l'archive (.zip ou tgz) dans votre répertoire de travail**. Si vous travaillez sur un ordinateur de l'université sous Windows ne mettez surtout pas vos fichiers dans les répertoires spéciaux de Windows (Mes Documents, Bureau, etc.). Mettez grapic simplement dans W:**

3. WINDOWS : faire un double clic sur le script *lifami.bat* qui créer tous les projets.

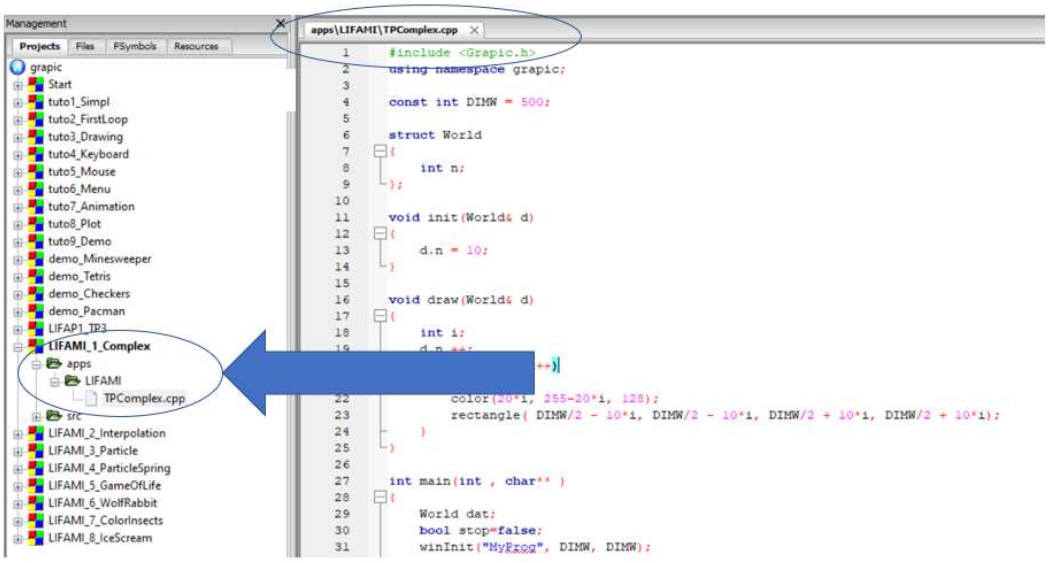
LINUX ou MacOS : make lifami (dans un terminal)

4. Windows : ouvrer avec Codeblocks le fichier *build/windows/grapic.workspace*



Linux : ouvrer avec Codeblocks le fichier buid/linux/grapic.workspace
MacOS : ouvrer les fichiers de projet du répertoire buid/macos avec XCode

5. Avec Codeblocks, le projet compilé et exécuté est celui en gras.
Un double clic sur un projet rend celui-ci actif.
Faire un double clic sur le projet ***LIFAMI_1_Complex***
Le fichier source est ***apps/LIFAMI/TPComplex.cpp***



Les problèmes courants avec Grapic sont décrits sur la page de Grapic tout à la fin dans la rubrique « Current problems »

Votre programme va souvent ressembler au programme de la page suivante. Vous renommerez la structure *Data* en fonction du problème à résoudre.

```
#include <Grapic.h>
using namespace grapic;
const int DIMW = 500;

struct Data             // Le type de vos données : A RENOMMER
{
  int n;
};

void init(Data& d)          // Initialise les données
{
  d.n = 10;
}

void update(Data& d)        // Met à jour les données
{
  d.n++;
}

void draw(Data& d)          // Affiche les données
{
  int i;
  for(i=0;i<d.n;i++)
  {
    color(20*i, 255-20*i, 128);
    rectangle( i*5, i*5, (i+1)*5, (i+1)*5 );
  }
}

int main(int , char** )
{
  Data dat;
  Menu m;
  bool stop=false;
  winInit("LIFAMI", DIMW, DIMW);
```

```
 init(dat);
 backgroundColor( 100, 50, 200 );
 menu_add( m, "Question 1");
 menu_add( m, "Question 2");
 menu_add( m, "Question 3");
 menu_add( m, "Question 4");

 while( !stop )
 {
  winClear();
  switch(menu_select(m))
  {
   case 0 : dat.n = 5; break;
   case 1 : dat.n = 15; break;
   case 2 : dat.n = 10; break;
   case 3 : dat.n = 20; break;
  }
  draw(dat);
  update(dat);
  menu_draw(m, 5,5, 100, 102);
  stop = winDisplay();
 }
 winQuit();
 return 0;
}
```

LIFAMI – TD : Nombres complexes et transformations du plan

Objectifs : Définition d'un nombre complexe
Transformations du plan : translation, homothétie, rotation

En mathématiques, les nombres complexes forment une extension de l'ensemble des nombres réels. Un nombre complexe z se présente en général sous forme algébrique comme une somme.

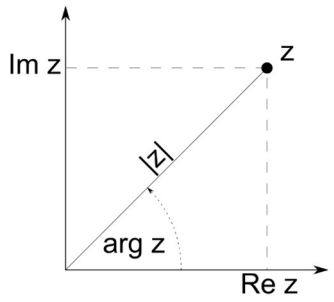
- z = x + iy, forme algébrique

où x et y sont des nombres réels quelconques et où i (l'unité imaginaire) est un nombre particulier tel que $i^2 = -1$. Le réel *x* est appelé partie réelle de *z* et se note Re(z), le réel *y* est sa partie imaginaire et se note Im(z).

Pour tout couple de réels (x,y) différent du couple (0,0), il existe un réel positif r et une famille d'angles déterminés à un multiple de 2π près tels que a = r cos(θ) et b = r sin(θ). Tout nombre complexe non nul peut donc s'écrire sous une forme trigonométrique :

- z = r (cos(θ) + i sin(θ)) avec r > 0, forme polaire
- $z = re^{i\theta}$, forme exponentielle

Le réel positif r est appelé le module du complexe z et est noté |z|. Le réel θ est appelé un argument du complexe z et est noté arg(z).



Dans un plan complexe $\mathcal{P}$ muni d'un repère orthonormé $(O; \vec{u}, \vec{v})$, l'image d'un nombre complexe $z = x + iy$ est le point *M* de coordonnées (*x*, *y*), son image vectorielle est le vecteur $\overrightarrow{OM}$. Le nombre *z* est appelé affixe du point *M* ou du vecteur $\overrightarrow{OM}$.

Un vecteur V du plan peut donc être représenté en coordonnées cartésiennes par V(x,y) ou par le nombre complexe x + iy. Un nombre complexe peut à la fois représenter un vecteur ou un point P qui n'est autre que le vecteur OP, avec O l'origine.

I. Les nombres complexes sont super cools !

1. Déclarez en C/C++ une structure *Complex* comportant deux champs réels nommés x et y.

2. Ecrivez en C/C++ les fonctions suivantes.

```
Fonction make_complex ( x, y : Reel) → Complex
Fonction make_complex_exp ( r, theta_deg : Reel) → Complex
```

3. Soient le point P et le vecteur V représentés par les complexes p et v suivants :
   - p = x + i.y
   - v = dx + i.dy

   L'addition de ces deux complexes p+v = x+dx + i.(y+dy) correspond à la translation du point P par le vecteur V.

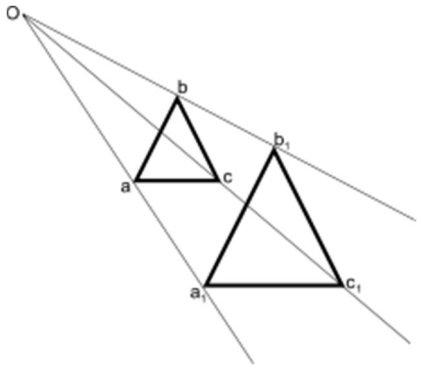
   En C++ il est possible d'écrire des fonctions particulières entre deux structures pour pouvoir utiliser les symboles +, - et *. Ces fonctions sont dites opérateurs d'addition, de soustraction et de multiplication. Ecrivez en C++ les opérateurs suivants.

```
Fonction operator+( a,b : Complex) → Complex  // Algo
Complex operator+(Complex a, Complex b) { …   // C++

Fonction operator-( a,b : Complex) → Complex   // Algo
Complex operator-(Complex a, Complex b) { …   // C++
```

   On pourra alors écrire ceci en C++ :

```
Complex a = make_complex(1, 1);
Complex b = make_complex_exp( 1, M_PI/2) ;
Complex c;
c = a+b;
```

4. Soient le point *P* et λ un réel, l'image P' du produit $\lambda.p$ est définie par la relation OP'=λ.OP. Cette multiplication d'un nombre complexe par un scalaire s'interprète géométriquement comme une homothétie de centre *O* et de rapport λ sur le plan complexe.



   Soit le triangle ayant pour points A(1,-1), B(0,1), C(-1,-1). Multipliez ces 3 points par le scalaire $\lambda=2$ (Faites le calcul). Dessinez les deux triangles. Et si $\lambda=0.5$ ou $\lambda=0$?

5. Un changement d'échelle (*Scale* en anglais) se fait par rapport à un centre C de coordonnées (Cx,Cy). Ecrivez les deux fonctions suivantes :

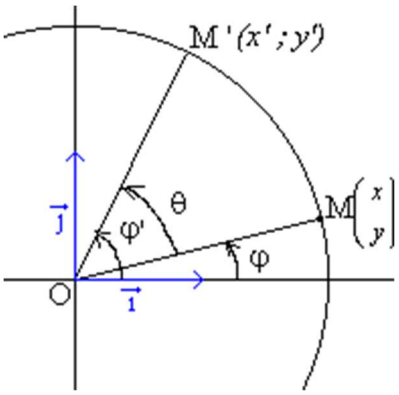
   `Complex operator*(float lambda, Complex b) // C++`
   qui multiplie un complexe par un scalaire.

   `Complex scale(Complex p, float cx, float cy, float lambda)// C++`
   qui effectue l'homothétie de centre (cx,cy) d'un facteur lambda. Pour cela il faut placer le point/complexe *p* dans le repère ayant pour origine (cx,cy), faire l'homothétie donc la multiplication, puis replacer le point dans le repère O.

Soit M un point représenté par le complexe p et *z* un complexe de norme 1 et d'argument θ (donc $z=e^{i\theta}$), l'image *M'* du produit *p.z* est définie par les relations :

- OM' = OM
- l'angle (OM,OM') = θ

La multiplication d'un nombre complexe quelconque par un autre complexe de module 1 et d'angle θ s'interprète géométriquement comme une rotation de centre O (l'origine) et d'angle θ.



6. Soit le triangle ayant pour points A(1,-1), B(0,1), C(-1,-1). Multipliez ces 3 points par le complexe $r= e^{i\theta}$ *avec* $\theta= \pi /2$. Dessinez les deux triangles.

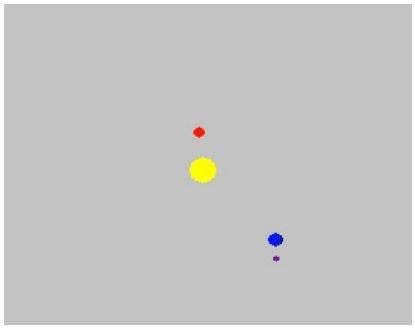
7. Ecrivez les deux fonctions suivantes.

```
Complex operator*(Complex a, Complex b)          // C++
```
qui multiplie deux complexes ensemble.

```
Complex rotate(Complex p, float cx, float cy, float theta_deg) // C++
```
qui effectue la rotation de centre (cx,cy) d'un angle theta_deg. Pour cela il faut placer le point/complexe p dans le repère ayant pour origine (cx,cy), faire la rotation donc la multiplication, puis replacer le point dans le repère O.

II. Saturne, ça tourne !

Nous allons afficher un système solaire simplifié comportant 3 planètes en plus du Soleil : Mercure, la Terre et la Lune. Chaque planète est représentée par un nombre complexe représentant sa position.



Une représentation du système solaire :
le Soleil en jaune, Mercure en rouge, la Terre en bleu et la Lune en gris.

Définissez la structure *SolarSystem* et écrivez les 3 fonctions suivantes.

```
Procédure init(ss : donnée-résultat SolarSystem)
```

qui initialise la position du soleil au centre de la fenêtre et les 3 planètes alignées horizontalement.

```
Procédure draw(ss : donnée SoloarSystem)
```

qui affiche avec 4 couleurs différentes les 4 astres.

```
Procédure update(ss : donnée-résultat SolarSystem)
```

qui met à jour la position des 3 planètes de façon à ce que mercure, la terre et la lune tournent autour du soleil et la lune tourne autour de la terre. Pour cela utilisez la procédure *rotate*.

LIFAMI – TP : Nombres complexes et transformations du plan

Ce TP va chercher à illustrer les différents aspects des nombres complexes au travers de différentes applications.

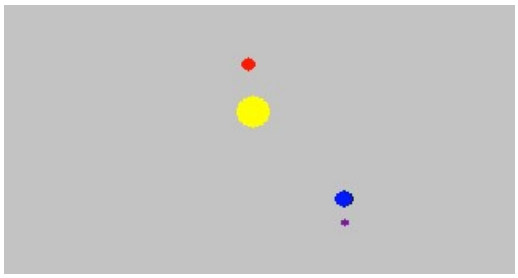
I. Saturne, ça tourne !

1. Codez les fonctions et procédures du TD questions de 1 à 7.

2. Reprenez la question du système solaire du TD et codez-la.

3. Vous ajouterez 4 images dans votre structure *SolarSystem* pour afficher une image de la planète à la place d'un cercle.

4. Vous pourrez également faire tourner le soleil autour d'un centre de galaxie placé au milieu de la fenêtre.



II. Battre des ailes pour voler !

L'objectif de cet exercice est de guider au clavier un oiseau qui bat des ailes.

1. Un oiseau est défini par une position représentée par un nombre complexe et un angle d'ouverture des ailes. Définissez la structure.

2. Ecrivez la fonction qui affiche l'oiseau.
   Pour faire une animation la fonction main va ressembler à ceci :

```
int main(int , char** )
{
    bool stop=false;
    winInit("Birds", DIMW, DIMW);
    backgroundColor( 100, 50, 200 );
    Bird bi;
    init(bi);
    while( !stop )
    {
           winClear();
           draw(bi);
           update(bi);
           stop = winDisplay();
    }
    winQuit();
    return 0;
}
```

3. A chaque itération de la boucle principale, l'oiseau bat des ailes en changeant l'angle. Ceci va se faire dans la fonction *update*. On pourrait aussi faire le changement d'état dans la procédure *draw*, mais pour bien structurer le code la procédure *draw* affiche et la procédure *update* met à jour les variables.

   Indication : l'angle va varier entre -20 et +20 degrés. Basez-vous sur une fonction oscillante et périodique que vous connaissez bien.

4. Ajoutez des comportements :
   - l'oiseau tombe ;
   - si on appuie sur la flèche haut, l'oiseau bat des ailes et monte ;
   - si on appuie sur flèche gauche/droite l'oiseau va à gauche/droite.

   Pour tester une touche :

```
if (isKeyPressed(SDLK_UP))
{
```

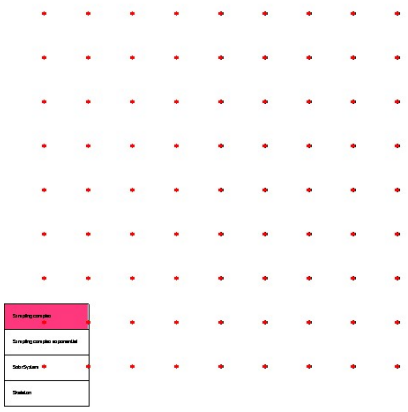
   Indication : utilisez la fonction *elapsedTime()* qui renvoie le temps passé depuis le lancement du programme.

III. Le pavé dans le plan complexe

En TD nous avons vu deux manières de construire un nombre complexe.

- Représentation algébrique : $C = x + i.y$
- Représentation exponentielle : $C = r(\cos(\theta) + i \sin(\theta)) = r.e^{i\theta}$

1. Ecrivez une procédure *draw* qui va effectuer une double boucle affichant une grille régulière sur l'espace des complexes, c'est-à-dire des points espacés de 10 unités entre 0 et 500 (si 500 est la taille de votre fenêtre) en utilisant *make_complex*.



2. Faites de même mais en explorant les paramètres *r* et θ en utilisant *make_complex_exp*.
    - La boucle sur *r* ira de …… à ……. par pas de …….
    - La boucle sur θ ira de …… à ……. par pas de …….

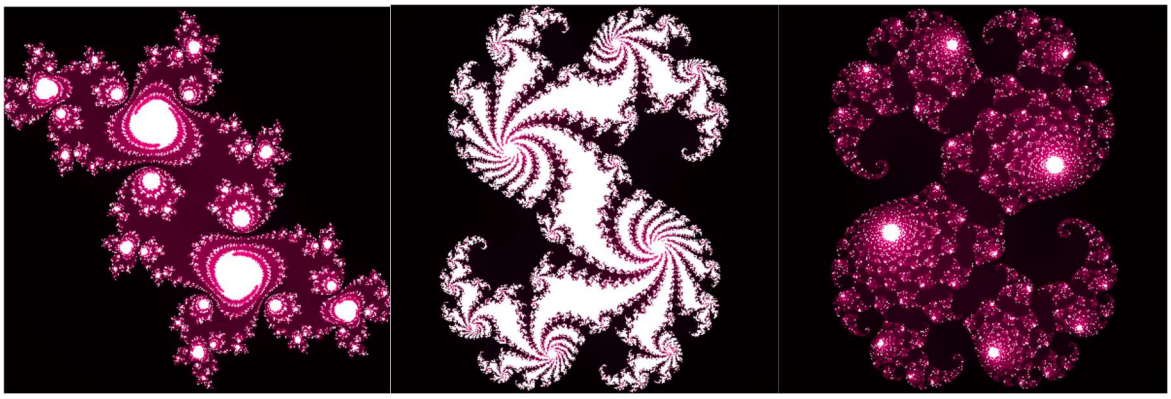
    Quelle figure allez-vous obtenir ?

IV. Julia

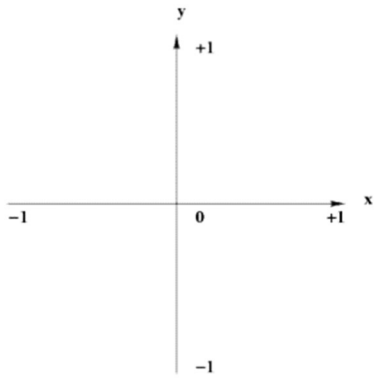
Définition de l'ensemble de Julia (cf. http ://fr.wikipedia.org/wiki/Ensemble de Julia). Etant donnés deux nombres complexes, C et Z_0, définissons la suite (Z_n) par la relation suivante :

$$Z_{n+1} = Z^2_n + C$$

Pour une valeur donnée de C, l'ensemble de Julia est la frontière de l'ensemble des valeurs initiales Z_0 pour lesquelles la suite est bornée. Dans notre cas, nous allons faire correspondre une valeur de Z_0 pour chaque pixel de l'image.



C = -0.0986 - 0.65186.i C = 0.32 + 0.043.i C = 0.284 - 0.0122.i



Chaque pixel de l'image va correspondre à un point complexe entre [-1 ;1]

1. Ecrivez une fonction qui implémente la suite (Z_n) et qui retourne le numéro du terme dont le module (la norme) est supérieur à une borne, si le module reste inférieur à cette borne, la fonction renverra le nombre maximal d'itérations. La borne d'arrêt, le nombre d'itérations maximal, le terme Z_0 et la constante C sont des paramètres de la fonction. Cette fonction renverra le nombre d'itérations avant que la série dépasse la borne.

2. Ecrivez un sous-programme qui « renvoie » une couleur en prenant en entrée le nombre d'itérations qui a permis l'arrêt du calcul de la suite de la question précédente. Pour les couleurs, choisissez celles que vous voulez ! En divisant le nombre d'itérations par le nombre d'itérations maximal vous obtenez un réel entre 0 et 1. Cette fonction pourra par exemple modifier 3 champs *r,g,b* passés en paramètres.

3. Ecrivez la procédure *draw_julia()* qui permette d'afficher un ensemble de Julia. Cette procédure utilisera les 2 procédures précédentes pour chacun des pixels de l'image. Vous utiliserez la fonction *put_pixel* de Grapic pour colorier un pixel de la fenêtre.

   Convertissez les coordonnées (i,j) de chaque pixel en Complexe de coordonnées de l'espace de la fonction entre [-1,1]. Passez ce complexe en paramètre Z_0. Essayez C = 0.32 + 0.043.i

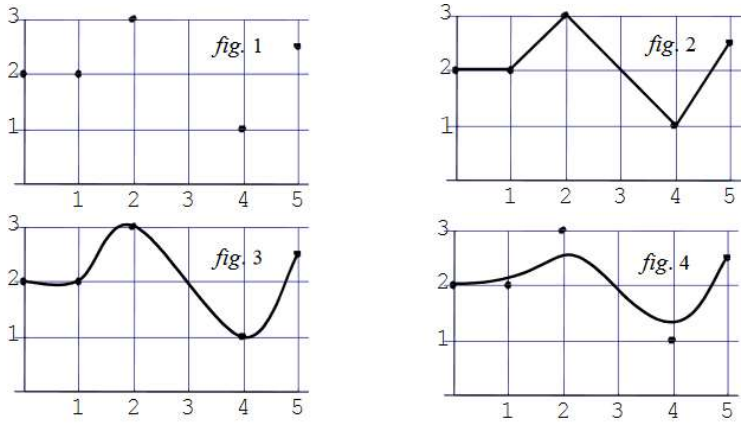
4. Testez votre programme, en changeant les dimensions de l'image, le nombre d'itérations maximal, la valeur de la constante C, les couleurs, la plage [-X ;X], ...

LIFAMI – TD : Interpolation

Objectifs : Notion d'interpolation linéaire, bilinéaire, barycentrique, etc.

Un **système discret** est un ensemble qui introduit une relation entre des variables d'entrée et des variables de sortie, dans lequel ces variables ne peuvent avoir qu'un nombre fini de valeurs, par opposition à un système continu, dans lequel entre deux valeurs de variables, quelles qu'elles soient, on peut toujours supposer une valeur intermédiaire.

En informatique, nous représentons souvent des systèmes continus dans la réalité par un ensemble de valeurs, donc par une représentation discrète. Par exemple, la courbe définissant une température mesurée par un capteur est continue. Pour n'importe quelle valeur de temps donnée il y a une température. En informatique nous n'allons stocker qu'un nombre discret de valeurs, par exemple une température toutes les heures dans le cas d'un capteur de météo. On dit que le signal est échantillonné à une fréquence d'une valeur par heure. Il est souvent important de pouvoir reconstruire une courbe continue à partir des valeurs discrètes.



I. Interpolation linéaire et fluor

L'extrait ci-dessous est tiré d'un décret français du 3 janvier 1989 concernant la teneur maximale en fluor dans l'eau destinée à la consommation humaine.

« Pour les eaux destinées à la consommation humaine, la teneur en fluor doit être inférieure à 1500 microgrammes par litre pour une température moyenne de l'aire géographique considérée comprise entre 8°C et 12°C et à 700 microgrammes par litre pour une température moyenne de l'aire géographique considérée comprise entre 25 et 30°C.
Pour les températures moyennes comprises entre 12 et 25°C, la teneur limite en fluor est calculée par interpolation linéaire. »

1. Quelle est cette teneur maximale tolérée pour une région pour laquelle la température moyenne est de 17°C ? Il est attendu ici le calcul et non du code.

2. Ecrivez la fonction qui renvoie la teneur en fluor en fonction de la température.

II. Interpolation linéaire et température

On a mesuré la température à différents moments de la journée toutes les heures, et reporté ces valeurs dans le tableau ci-dessous.

Heure	9h	10h	11h	12h	13h	14h	15h	16h	17h	18h
Température	11	14	16	17	19	16	16	13	12	11

1. Représentez ces données par un graphique (sur votre feuille).

2. Peut-on interpoler ces données pour obtenir une température à 13h30 et à 10h20 ?
   Remarque : *t* est donné en heures/minutes, il faut convertir *t* en minutes.

3. Créez une fonction *temperature* qui donne une interpolation de la température pour t∈ [9h00;18h00]. Cette fonction prendra en paramètre le tableau de 10 températures et le temps en heures et minutes.

4. (A faire à la fin du TD) Ecrivez une fonction qui affiche cette courbe à l'aide de la fonction *line* de *Grapic*.

III. Les polygones sont des petis lyonnais polis ! (ahahah …)

Un polygone à N cotés comporte N sommets qui seront représentés par un point/*Complex*.

1. Soient deux nombres complexes ou points A et B, écrivez la fonction qui interpole entre ces deux points par un paramètre t. Remarque : vous obtenez l'équation paramétrique d'une droite D : d(t) = A + t.AB
   Quand t=0, d(0) = A
   Quand t=1, d(1) = B
   Pour t entre [0 ;1] d(t) = segment de droite AB

```
Fonction Complex_Interp(A,B : donnée Complex) → Complex
```

2. Définissez la structure *Polygon* comportant un entier indiquant le nombre de sommets et un tableau de *Complex* de taille MAX.

3. Ecrivez la procédure qui ajoute un sommet au polygone *p*.

```
Procédure polygon_add(p : donnée-résultat Polygon ; px, py : Réel)
```

4. Ecrivez la procédure qui affiche le polygone en utilisant la fonction *line*.

```
Procédure polygon_draw(p : donnée Polygon)
```

5. Ecrivez la procédure qui calcule un nouveau polygone résultant de l'interpolation entre deux polygones. La procédure prendra en paramètres les 2 polygones, un réel *t* et le polygone résultat.

6. Ecrivez le sous-programme qui calcule le centre de gravité d'un polygone.

7. Ecrivez le sous-programme qui déplace un polygone de (dx,dy).

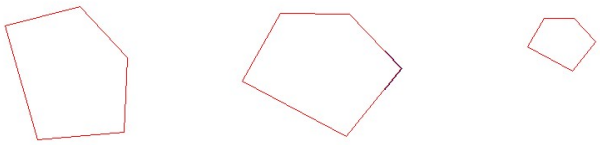
8. Ecrivez la procédure qui applique l'homothétie de centre (cx,cy) et de facteur *lambda* au polygone *p*.

```
Procédure polygon_scale(p : donnée-résultat Polygon ; cx,cy : Réel ;
lambda : Réel)
```

9. Ecrivez la procédure qui fait tourner le polygone *p* d'un angle *theta* par rapport au centre (cx,cy).

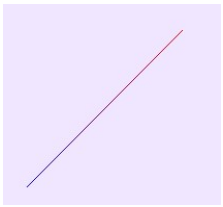
```
Procédure polygon_rotate(p : donnée-résultat Polygon ; cx,cy :
Réel ; theta : Réel)
```

10. Ecrivez les sous-programmes qui modifient le polygone *p* de manière à en faire le symétrique par rapport à X et Y. Il faut le recentrer au milieu de la fenêtre



Rotation et changement d'échelle d'un polygone avec des nombres complexes

IV. Pour aller plus loin



La droite

Soient deux points du plan 2D A et B. La température de A est T_a et celle de B est T_b. La droite passant par A et de vecteur directeur AB peut être représentée par une équation paramétrique qui est très pratique pour faire une interpolation linéaire.

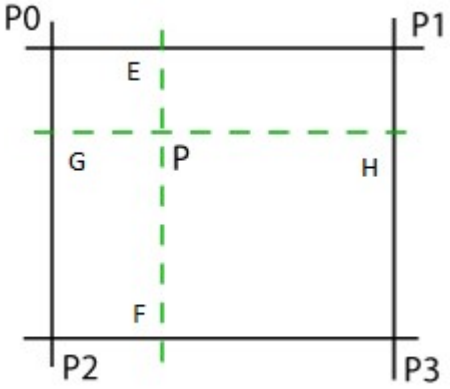
1. Donnez cette équation paramétrique.
   Utilisez cette équation pour calculer la température le long de la droite par interpolation linéaire. La température -30°C est représentée par du bleu (0, 0, 255), la température +40°C est représentée par du rouge (255,0,0).

2. Ecrivez la structure *Color* comportant 3 composantes R,G et B. Vous pouvez écrire les opérateurs +, * :

```
Color operator+(Color a, Color b);
Color operator*(float a, Color b);
```

3. Ecrivez une procédure affichant une droite en couleurs en fonction de sa température. On découpera la droite en N segments, chacun de ces segments aura une couleur unique.

V. Le carré coloré



Soit le quadrilatère P_0 P_1 P_3 P_2 dont les côtés sont alignés sur les axes. La température de P_0 est T_0 , celle de P1 est T_1, celle de P2 est T_2 et celle de P3 est T_3 . Comme pour l'exercice précédent nous souhaitons colorier ce quadrilatère en fonction de sa température. Pour cela nous allons introduire la notion d'interpolation bilinéaire qui n'est autre qu'une interpolation linéaire faite deux fois : une fois suivant l'axe des X et une fois suivant l'axe des Y. Soit P un point interne au quadrilatère. En fonction des longueur GP, PH, EP et PF nous allons trouver une manière d'interpoler la température.

1. Nous allons définir le point E projection de P sur P_0P_1 et F projection de P sur P_2P_3. Par interpolation linéaire calculez la température de E et de F. Puis par une 2^e interpolation linéaire calculez la température de P. Cette manière d'interpoler est appelée interpolation bilinéaire.

2. Ecrivez le sous-programme *draw_quad_bilineaire* qui remplit le quadrilatère avec la couleur correspondant à sa température. On utilisera la procédure de Grapic :

```
void put_pixel(int x, int y, unsigned char r, unsigned char
g,unsigned char b) ;
```

qui affiche donne la couleur (r,g,b) au pixel (x,y)

VI. Le triangle de feu

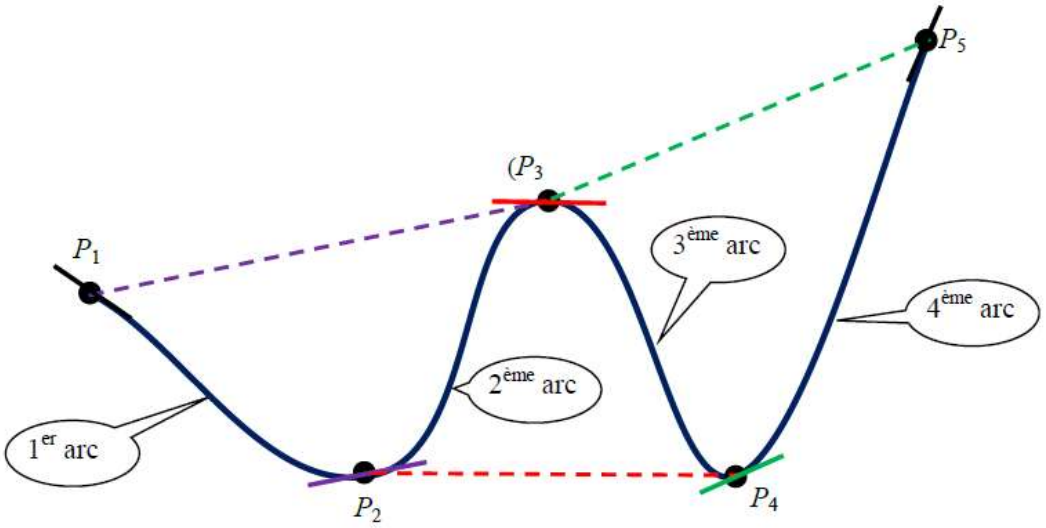
Soit le triangle ABC. La température de A est T_a, celle de B est T_b et celle de C est T_c. Comme pour l'exercice précédent nous souhaitons colorier ce triangle en fonction de sa température. Pour cela nous allons utiliser les coordonnées barycentriques.

1. Soit G(x,y) un point du triangle, trouver les coordonnées barycentrique de G en fonction des coordonnées de A,B et C.

2. Ecrivez la fonction *draw_triangle* qui remplit un triangle avec la couleur correspondant à sa température. Indication : vous pouvez faire une boucle sur les pixels du rectangle englobant.

VII. Interpolation cubique

Étant donnés n points du plan, P_1, P_2, ..., P_n, on cherche à construire une courbe formée de n–1 arcs de parabole. Le premier de ces arcs est d'extrémités P_1 et P_2, le deuxième d'extrémités P_2 et P_3, le n–1 ème d'extrémités P_{n-1} et P_n. Chacun de ces arcs est le graphe d'une fonction $f(x) = ax^3 + bx^2 + cx + d$. Ces arcs doivent être reliés sans rupture de pente, la dérivée à la fin d'un arc doit donc être égale à la dérivée à l'origine de l'arc suivant.



1. Trouvez les coefficients a, b c et d d'une fonction $f(x) = ax^3 + bx^2 + cx + d$ telle que f(0) = 0, f(x1) = y1 et f '(0) = p0 et f '(x1) = p1 (x1, y1, p0 et p1 sont donnés).
   Indice : vous devez trouver ceci

$$d = 0,\ c = p_0,\ b = \frac{3y_1 - x_1(2p_0 + p_1)}{x_1^2} \text{ et } a = \frac{(p_1 - p_0 - 2bx_1)}{3x_1^2}$$

2. La tangente en P_i sera la droite allant du point précédent P_{i-1} au point P_{i+1}. Ecrivez la fonction qui trace la courbe.

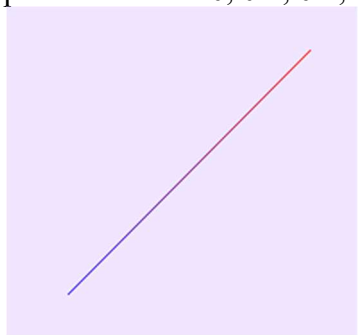
LIFAMI – TP : Interpolation

Objectifs : Notion d'interpolation linéaire, bilinéaire, etc.

I. La base

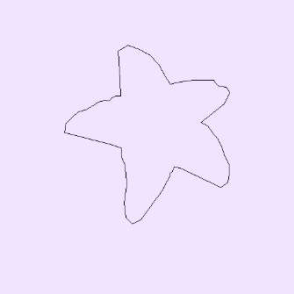
1. Ecrivez une fonction qui prend en paramètres deux valeurs et un coefficient d'interpolation entre 0 et 1 et qui calcule l'interpolation (la moyenne pondérée) des deux valeurs.

2. Ecrivez une fonction qui dessine une droite entre A et B avec un dégradé du bleu au rouge.
   Indications : la droite D paramétrique est
   $$D(t) = \vec{A} + t.\overrightarrow{AB}.$$
   Quand t=0, la fonction donne le point A.
   Quand t=1, la fonction donne le point B.
   Quand t=0.5, la fonction donne le point milieu entre A et B
   Etc.
   On découpera la droite en 10 portions avec t=0, 0.1, 0.2, 0.3, … 0.9, 1



II. Morphing ou comment changer de forme

Un polygone 2D est défini par un nombre *n* de points et un tableau *p* de points. Un point est défini par deux coordonnées réelles.



1. Ecrivez la structure *Point* et la structure *Polygon*. Pour la structure *Point* vous pouvez réutiliser la structure *Complex* du *TP-Complexe* en la renommant. Copiez/Collez

également tous les opérateurs +, -, *. Les champs de la structure *Polygon* doivent s'appeler *p* pour le tableau et *n* pour le nombre de points.

2. Sur la page de l'UE, récupérez l'archive contenant les fichiers *poly1.h*, *poly2.h*, etc. (http://liris.cnrs.fr/alexandre.meyer/teaching/LIFAMI/data/data_interpolation.zip). Rangez-les dans le même dossier que votre fichier *TP_interpolation.cpp*. Dans votre code, **après avoir déclaré les structures** ajoutez

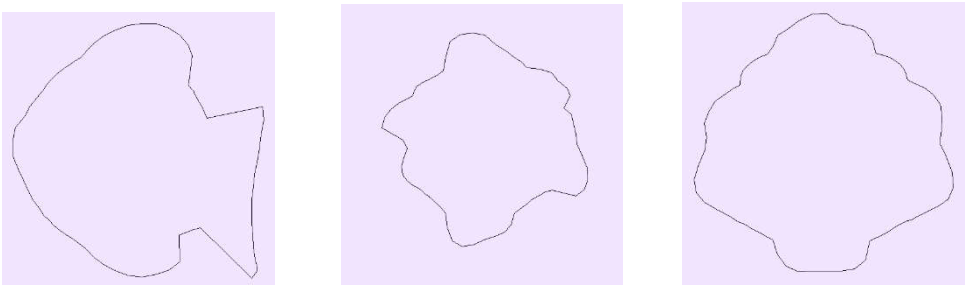
   ```
   #include "poly1.h"
   ```

   Vérifiez que les données ont bien été importées en appelant la procédure d'initialisation. Sinon renommez vos champs de la structure *Polygon*.

3. Ecrivez la procédure *draw_polygon* qui affiche un polygone avec des lignes. Vous pouvez utiliser la fonction *line* qui trace une ligne.

4. Ecrivez la procédure *interpolation_polygon* qui calcule un nouveau polygone interpolé à partir de 2 polygones en entrée et d'un poids.

5. Utilisez la procédure précédente pour afficher une animation déformant le polygone 1 vers le polygone 2. Indication : utilisez la fonction *elapsedTime()* qui renvoie le temps passé depuis le lancement du programme et trouvez une fonction périodique qui pourrait faire osciller l'animation d'un polygone à l'autre.



6. Ecrivez une procédure qui déplace une soucoupe volante (un cercle) le long d'un polygone.

III. Images : morphing par mélange

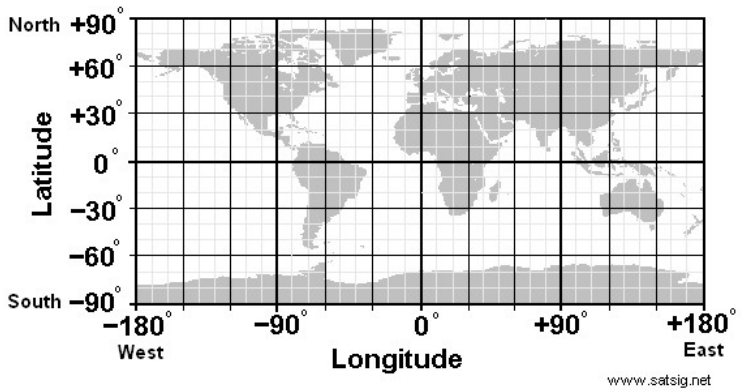
1. Ecrivez la procédure qui affiche un carré dont les côtés sont alignés avec les axes, et qui sera rempli par interpolation bilinéaire des couleurs des 4 sommets. Voir l'exercice IV- du TD.

2. Ecrivez une fonction qui interpole chaque pixel de deux images pour passer d'une image à l'autre avec un fondu. Ceci n'interpole que la couleur, pas la forme de l'objet.

IV. Froid et chaud (Pour aller plus loin)

Une station météo est représentée par une position et une mesure de température à un temps donné. La position est une coordonnée longitude et latitude en degrés. La température est mesurée en degrés Celsius.

Les données de cet exercice sont issues de la NASA et indiquent un écart par rapport à la température moyenne de la station, cette mesure est appelée l'anomalie et permet de voir si la zone se réchauffe ou se refroidit :
http://data.giss.nasa.gov/gistemp/maps/



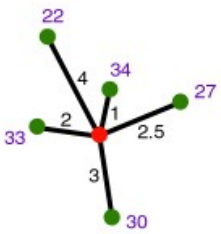
1. Ecrivez une structure *WeatherStation* qui comprend la longitude/latitude en degrés, les coordonnées en pixels et la température ; ainsi que la structure stockant toutes les stations météo dans un tableau.

   Sur la page de l'UE, vous trouverez la fonction *init* qui initialise toutes les stations, ainsi que l'image du monde pour la question suivante (archive téléchargée à la question II-2).

2. Ecrivez une fonction qui convertit une coordonnée longitude/latitude en une coordonnée pixels.

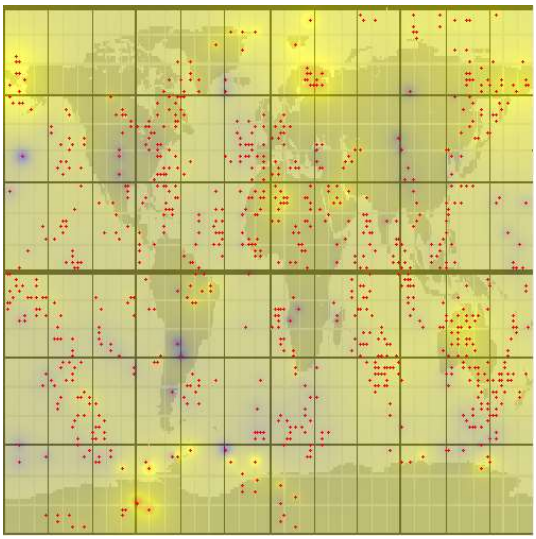
3. Ecrivez une procédure qui affiche la carte du monde, ainsi que toutes les stations météo, par exemple avec un cercle.

4. Ecrivez une fonction qui calcule par interpolation la température de n'importe quel point du monde repéré par ses coordonnées. Pour cela vous allez calculer la moyenne pondérée de toutes les stations du monde. Le poids devra dépendre de la distance, une station proche du point devra avoir un poids élevé, une station éloignée un poids faible, voire très faible. Une approche classique est de donner comme poids l'inverse de la distance : 1/d. Attention quand d=0 !! Cette méthode s'appelle *Inverse distance weighting*.



$$Z(x) = \frac{\sum w_i z_i}{\sum w_i} = \frac{\frac{34}{1^2} + \frac{33}{2^2} + \frac{27}{2.5^2} + \frac{30}{3^2} + \frac{22}{4^2}}{\frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{2.5^2} + \frac{1}{3^2} + \frac{1}{4^2}} = 32.38$$

5. Ecrivez la procédure qui affiche la température de tous les pixels en surimpression sur toute la carte du monde. Utilisez le champ A de la fonction *color(R,G,B,A)*.



Rechauffement de la planète : en bleu la température moyenne a baissé de maximum 4°C jusqu'à rouge où la température a augmenté de 10°C en passant par le jaune avec une augmentation de 3°C

6. (Pour aller plus loin) Calculez et tracez la courbe des températures moyennes en fonction de la latitude.

LIFAMI – TD : Dérivée et intégrale de fonctions discrètes

Objectifs : Notion de dérivation et d'intégration de fonctions discrètes

Les notions abordées dans ce TD sont assez générales et fondamentales. Il est important de comprendre ce qu'est une dérivée, une intégrale car ces notions sont utilisées dans de nombreux domaines des sciences. Essayez de faire le lien avec vos enseignements de mathématiques.

I. Les loups et les intégrales

Nous nous intéressons ici à la fonction P(t) qui représente le nombre de loups d'une population. Cette fonction évolue au cours du temps. Il est plus commode pour un gardien de noter le nombre de morts et de naissances de loups dans son secteur par mois. En effet, il trouve les cadavres et repère facilement les mères mettant bas. Il range ces valeurs dans deux tableaux N et M. La variation de la population par rapport au temps peut donc s'écrire

$$P'(t) = \frac{dP}{dt} = N_{naissance} - N_{mort}$$

où dP/dt est la dérivée de la fonction Population P(t) par rapport au temps.

1. Cas d'école. Supposons que le nombre de naissances moins le nombre de mort soit toujours égale à une constante, par exemple 2. Il y a toujours 2 naissances de plus que de morts. On a dans cet exemple : dP/dt = 2 ; ici dt = 1 mois.

   Quelle est la taille de la population après t mois. L'intégrale de cette fonction donne la réponse.

$$P(t) = P(t_0) + \int_{t_0}^{t} P'.dt \;\; car \;\; \int_{t_0}^{t} P'.dt = P(t) - P(t_0)$$

2. Dans la pratique, les naissances et les morts sont souvent notées/stockées de manière discrète (c'est à dire « tous les x jours »). En informatique par exemple, ces valeurs peuvent être stockées avec le tableau :

Mois	1	2	3	4	5	6	7	8	9	10	11	12
Naissance	0	0	0	1	3	2	4	2	1	0	0	0
Mort	2	3	2	1	2	3	0	0	0	1	0	1

L'intégrale se transforme alors en somme :

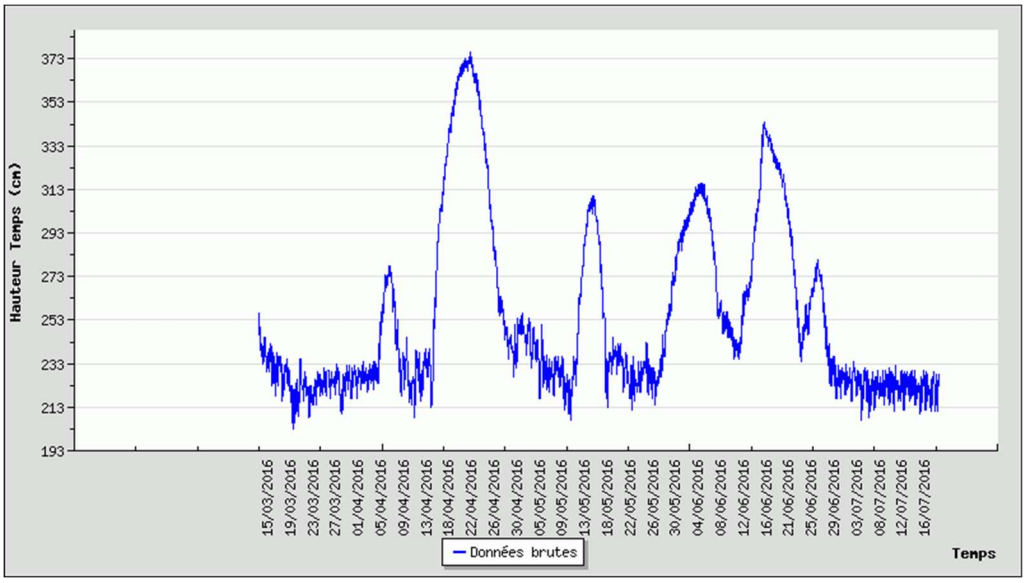
$$\int_{t_0}^{t} P'.dt = \sum_{t0}^{t} P'(t)$$

et en sachant que la population au mois 0 était de 17 individus. Ecrivez une fonction qui calcule le nombre de loups après D mois.

II. Le niveau d'eau de la Saône et les dérivées

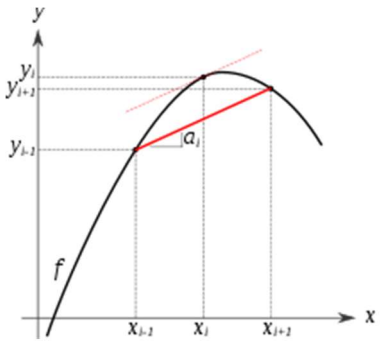
Dans le problème de l'exercice I, il était commode de produire les données en donnant une variation, puis une intégration permettant de retrouver la fonction à analyser. Dans cet exercice, nous verrons que pour certains problèmes la variation d'une valeur est plus pertinente à analyser.

Sur tous les fleuves de France, il y a des stations de mesure du niveau d'eau. Par exemple la station U4720020 mesure le niveau de la Saône à Lyon (Pont La Feuillée), voici les données entre mars et juillet 2016 :



Nous aimerions avoir une information sur les jours de pluie dans le département à partir de ces valeurs. Connaître le niveau de la Saône à un instant donné ne vous donne pas des informations très pertinentes. Il est sûrement plus intéressant de regarder comment le niveau a varié. Donc de calculer la dérivée de la fonction :

$$\frac{dH}{dt} = \lim_{dt \to 0} \frac{H(t) - H(t - dt)}{dt}$$



$$f'(x_i) \simeq \frac{y_{i+1} - y_{i-1}}{x_{i+1} - x_{i-1}}$$

1. Ecrivez la fonction qui calcule la dérivée discrète de la fonction Hauteur dont les valeurs sont stockées dans un tableau. L'intervalle de mesure des valeurs de hauteurs est constant : une mesure chaque jour, donc dt = 1 jour.

Jour	…	105	106	107	108	109	110	111	112	113	114	115
Hauteur en cm		233	230	234	240	245	247	248	241	237	243	246

2. Une estimation des jours de pluie peut se faire en comptant le nombre de jours où le niveau d'eau monte, donc où la dérivée est positive. Ecrivez la fonction qui calcule le nombre de jours de pluie.

3. Pour estimer les jours de forte pluie, nous pouvons chercher les jours où l'augmentation du niveau d'eau est plus forte que la moyenne des augmentations journalières.

    a. Ecrivez la fonction qui calcule la moyenne des augmentations sur l'ensemble des données. Indication : il ne faut prendre en compte que les jours où la variation est positive, donc où la dérivée est positive.
    b. Ecrivez la fonction qui calcule le nombre de jours de forte pluie.

4. Avec Grapic vous pouvez affichez un graphique comme ceci :

```
Plot p;
float t;
for(x=0.0; x < 2.0*M_PI; x+=0.1)
   plot_add(p, x, cos(x));
// ajoute les valeurs de y=cos(x) pour les x entre 0 et 2*PI

…
plot_draw( p, 20, 20, 180, 180);
// dessine la courbe cosinus dans le rectangle définit par les
coordonnées pixel (20,20) (180,180)
```

Ecrivez la fonction qui trace la courbe d'une fonction H dont les valeurs sont rangées dans un tableau, puis qui trace la dérivée de cette fonction.

LIFAMI – TD : Système de particules / Mécanique du point

Objectifs : Notion de mécanique du point
Seconde loi de Newton et intégration

I. Ça va vite !

1. Une voiture avance de manière rectiligne et se trouve à la borne kilométrique 1.2 km à une vitesse de 30 km/h. La voiture n'accélère pas et ne freine pas, quelle distance aura-t-elle parcourue après 10s ? Où se trouvera-t-elle ?

2. Une sprinteuse court le 100 m en 11.05s, quelle est son accélération ? On supposera qu'elle accélère de façon constante tout au long du mouvement et part à l'arrêt.

3. Si on laisse tomber une pièce d'un édifice dont la hauteur est de 365m, à quelle vitesse, en km/h, percutera-t-elle le sol ?

II. Dis Papa, c'est encore loin !

Vous êtes en voiture et vous notez toutes les minutes la vitesse approximative de la voiture dans un tableau.

Temps en minutes	0	1	2	3	4	5	6	7	8	9	10	11
Vitesse en km/h	35	45	52	72	90	85	95	87	61	54	48	44
Distance en m ?	0	583										

1. Essayez de produire le calcul qui vous mène à trouver la distance après 1 min, après 2 min, etc. On supposera que la voiture roule à la vitesse indiquée pendant toute la minute précédant la notation.

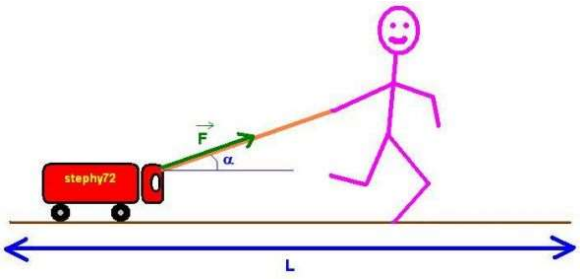
2. Ecrivez la fonction qui calcule la distance parcourue à partir du tableau de vitesses et du temps parcouru t en minutes. Chaque case i du tableau contient la vitesse de la voiture au temps t = i minutes.

III. Avec le principe fondamental de la dynamique, ça bouge !

1. Rappelez la deuxième loi de Newton.

2. Toujours en 1D, donc de manière rectiligne, avec une force constante tout au long du temps écrivez les calculs qui donnent la vitesse puis la position à partir de l'accélération. Vous devez intégrer l'accélération pour obtenir la vitesse puis intégrer la vitesse pour obtenir la position.

   En 2D, dans un repère orthonormé (X,Y) qu'est-ce qui change ?



Jusqu'à présent (Terminale) vous avez toujours supposé que les forces étaient constantes, par exemple uniquement la gravité. Mais il y a de nombreuses situations où les forces varient en fonction du temps. Imaginez par exemple une caisse poussée par 3 personnes en même temps, ces 3 personnes ne poussent pas de manière continue. Parfois l'une se repose un peu, une autre pousse dans une direction un peu différente, etc. La trajectoire de la caisse ne peut donc pas se déduire de manière analytique en faisant l'intégrale analytiquement. Il faut faire une intégration discrète, c'est-à-dire en découpant le calcul par petits intervalles de temps (on dira aussi : pas de temps).

Nous allons noter p_t la position au temps t, v_t la vitesse au temps t et a_t l'accélération au temps t. L'accélération est la variation de la vitesse. Soit dt un intervalle de temps, petit, par exemple 10^{-3}seconde (0.001 seconde). L'accélération est la variation de la vitesse :

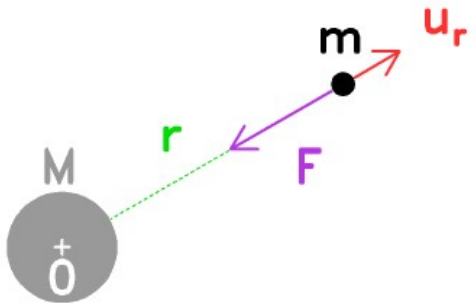
$$\frac{dv_t}{dt} = \frac{v_t - v_{t-dt}}{dt} = a_t$$

Nous pouvons faire de même pour la vitesse qui est la variation de position :

$$\frac{p_t - p_{t-dt}}{dt} = v_t$$

3. Calculez la nouvelle position en fonction de la vitesse et de la position au pas de temps précédent.
4. Calculez la mise à jour de la vitesse en fonction de la vitesse au pas de temps précédent et de l'accélération.
5. A partir de ces deux expressions et du principe fondamental de la dynamique, calculez la position d'une particule en fonction des forces subies au temps t.

IV. Et si on codait ça ?



La position d'une particule se trouvant au point P peut s'écrire comme étant le vecteur OP, un vecteur à 2 dimensions dans le cas du plan que nous allons traiter ici. La vitesse de la particule est également un vecteur.

1. Écrivez en C++ la structure *Vec2* qui stocke un vecteur à 2 dimensions. Vous pouvez reprendre la structure *Complex* et la renommer. Prenez également les opérateurs +, -, *, et /.

2. Écrivez en C++ la structure *Particle* qui stocke les informations nécessaires à une simulation par le second principe de la dynamique (2e loi de Newton). Une particule est représentée par une position, une vitesse, une force et une masse. Si la particule subit plusieurs forces, les vecteurs forces seront sommés.

3. Écrivez la fonction *partInit* qui initialise une particule avec les paramètres suivants : une position *p,* une vitesse *v*, et une masse *m.*

4. Écrivez la procédure *void partAddForce(Particles& p, Vec2 force)* qui ajoute une *force* à la particule *p*.

5. Écrivez la procédure *void partUpdatePV* qui met à jour la vitesse et la position d'une particule.

6. Écrivez le programme principal simulant et affichant une particule.

   Remarque : il faut prévoir des forces, sinon la particule restera immobile.

7. Définissez la constante *MAX_PART*, ainsi que la structure *World* qui contient un ensemble de particules. La structure World contiendra un tableau de *MAX_PART* particules, ainsi que le nombre de particules réellement simulées dans le tableau.

8. Écrivez la procédure *init* qui prend en paramètres un monde *World* et un nombre de particules à initialiser dans ce monde. Initialisez le vecteur vitesse des particules de manière à avoir une fontaine.

   Écrivez en même temps une fonction *addParticle* qui prend en paramètre un monde *World*, une particule et qui renvoie l'indice de la particule ajoutée dans le tableau. Ceci vous sera utile pour le TD suivant.

   void addParticle(World& w , Particle p)

LIFAMI – TD : Collisions et système Masses-Ressorts

Objectifs : Physique des ressorts et codage de structures masses-ressorts

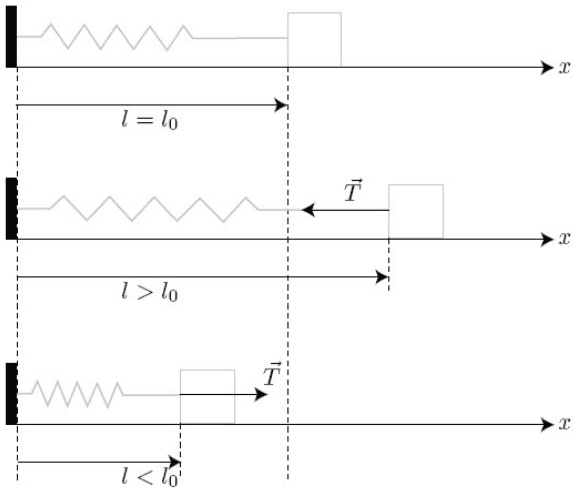
Le TD système de particules doit être entièrement terminé. S'il ne l'est pas, terminez-le.

I. Les collisions

Regardez la partie collisions du TP. Les particules doivent rebondir sur les bords de la fenêtre.

II. Masses-Ressorts

Nous allons allez un peu plus loin avec la manipulation de particules en mouvement en connectant deux particules ensemble avec un ressort.

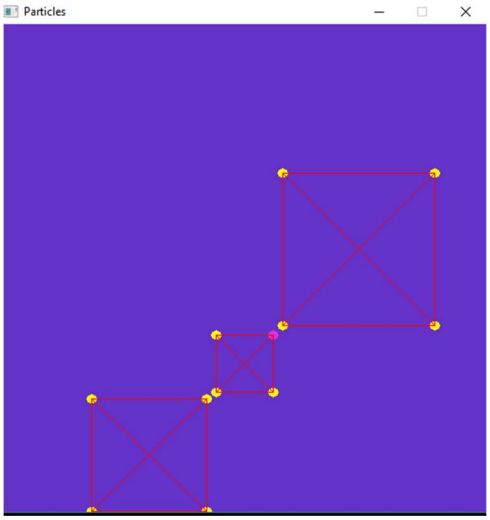


La force de rappel élastique exercée par le ressort sur la masse m est

$$T = -k \cdot \Delta l \cdot e_r$$

où:

- e_r est un vecteur unitaire dirigé suivant l'axe du ressort, orienté vers l'extérieur.
- $\Delta l = l - l_0$ est l'allongement du ressort en notant l la longueur du ressort, et l_0 sa longueur à vide.
- k est la **raideur du ressort**, intrinsèque au ressort considéré, elle caractérise la capacité du ressort à résister au mouvement de la masse m, et par conséquent à revenir à sa position d'équilibre.



1. Définissez une structure *Spring* (ressort) s'adossant à la structure *World* contenant l'ensemble des particules du monde. Vous pourrez utiliser l'indice d'une particule dans le tableau comme identifiant.

2. Ajoutez un tableau de ressorts à la structure du monde *World.*

3. Modifiez la procédure *init* qui initialise le monde pour qu'elle créé également les ressorts. Nous vous conseillons d'écrire une structure *addParticle* qui renvoie l'indice de la particule ajoutée. Vous pourrez commencer par créer 2 particules reliées par un ressort, puis faire un triangle, puis un carré, etc.

```
procedure addParticle(w : donnée/Resultat World)
```

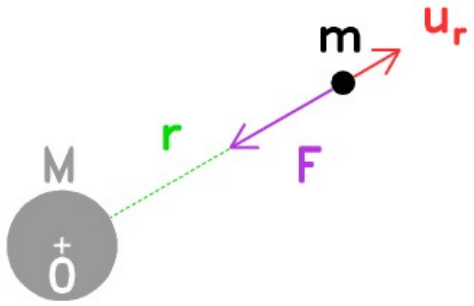
4. Modifiez la procédure *draw* pour qu'elle affiche les ressorts représentés par des lignes.

5. Ecrivez la procédure *computeParticleForceSpring* qui calcule les forces qu'exercent les ressorts sur toutes les particules du monde.

LIFAMI – TP : Système de particules / Masses-Ressorts

Objectifs : Mécanique du point et masses-ressorts

I. La base



1. Codez les questions relatives à « Et si on codait ça ? » du TD Système de particules.

Le programme principal avec *Grapic* peut ressembler à ceci :

```
int main(int , char ** )
{
    World dat;

    bool stop=false;
    winInit("Particles", DIMW, DIMW);
    backgroundColor( 100, 50, 200 );

    init(dat);

    Menu menu;
    menu_add( menu, "Init" );
    menu_add( menu, "Run");


    while( !stop )
    {
        winClear();

        draw(dat);          // dessine les particules
        update(dat);        // déplace les particules

        if (menu_select(menu)==0)
        {
           init(dat);
           menu_setSelect(menu,1);
        }
        menu_draw( menu);
        stop = winDisplay();
    }
    winQuit();
    return 0;
}
```

II. Les collisions

Sur terre, une approximation de la force que subit une particule soumise à l'attraction terrestre est le poids, avec m la masse et G est la constante de gravitation universelle G=9.81 :

$$P = m.G$$

1. Ajoutez une structure *World* qui va stocker plusieurs particules dans un tableau. En complément de la structure, programmez une fonction *init.*

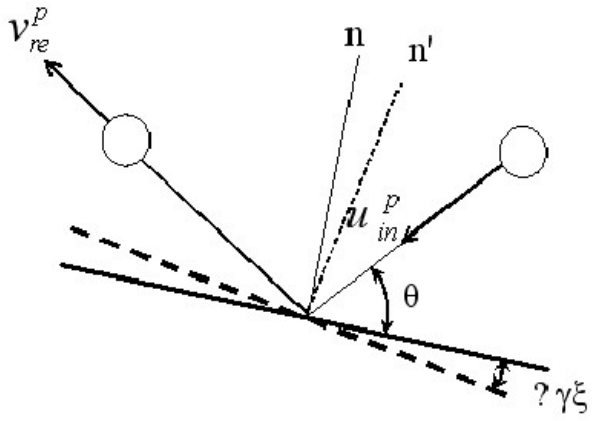
2. Ajoutez à votre programme cette force de gravité pour que vos particules « tombent ».

Pour la gestion des collisions nous allons considérer uniquement des murs horizontaux ou verticaux : x=0, y=0, $x=X_{max}$, $y=Y_{max}$

3. Modifiez le code de la fonction *update* pour intégrer ces collisions. Vous devriez obtenir des particules qui rebondissent dans la boite de l'écran. Le test de collision est assez simple : if (x<0) { … }

La réaction à une collision peut se faire en remettant artificiellement la particule dans le rectangle de la simulation : x = -x ;

Il faut aussi réorienter le vecteur vitesse de manière symétrique : $v_x = -v_x$



4. Après un choc une particule perd un peu de vitesse (énergie) absorbée par le mur. Ceci se fait en réduisant le vecteur vitesse : v = -FRICTION * v où FRICTION est une constante inférieure à 1, par exemple *0.6*.

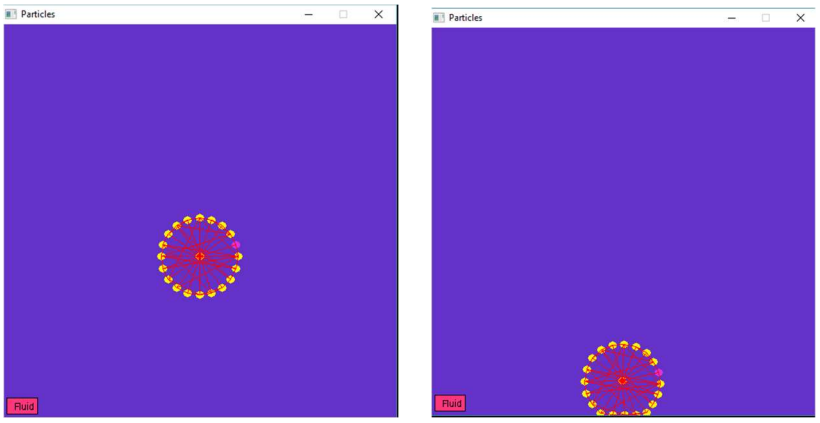
III. Rebondissons avec les masses-ressorts

La partie Loi universelle de la gravitation qui vient après peut se faire indépendamment de cette partie. Il s'agit de deux branches possible du TP.

1. Codez toutes les fonctions du TD Masses-Ressorts.

2. Vous pouvez imaginer des interactions en ajoutant des forces à certaines particules lorsqu'on appuie sur une touche.

3. Vous pouvez imaginer des formes différentes (cercles, etc.), et par exemple faire tourner une roue en appliquant les bonnes forces extérieures au système.



IV. Loi universelle de la gravitation (Pour les plus rapides)

Dupliquez votre programme des questions précédentes pour en garder une sauvegarde.

1. Deux particules interagissent par la force de gravité. Notons M et m leur deux masses respectives (en kg), r leur distance (en m), u_r étant le vecteur unitaire entre les deux particules. La force F que subit chaque particule est

$$\mathbf{F} = -\frac{\mathcal{G}Mm}{r^2}\,\mathbf{u}_\mathrm{r}$$

   Modifiez votre programme pour que les particules ne subissent plus la gravité terrestre mais plutôt pour que chaque particule se comporte comme un astre subissant la gravité de tous les autres astres et provoquant une force de gravité vers tous les autres astres.

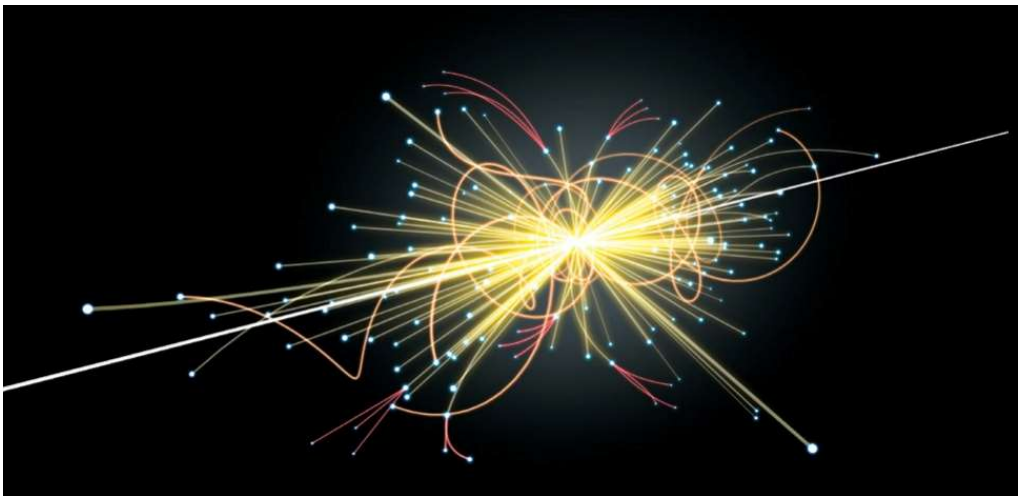
2. En premier test vous simulerez uniquement 2 particules : un soleil au centre avec une masse assez élevée et une particule positionnée à une distance de 100 pixels.
   Quelle trajectoire allez-vous observer ?

   Modifiez votre programme pour que la petite particule tourne autour de la grande.

   Remarque : la particule peut sortir de l'écran.

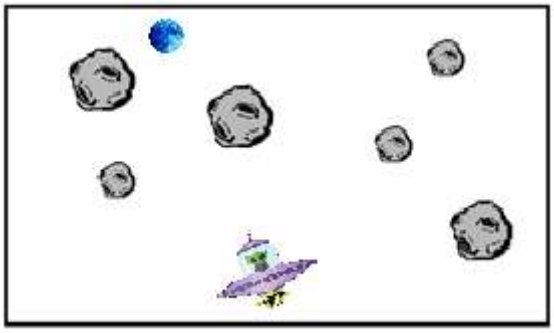
3. Testez plusieurs configurations de particules avec différentes masses. Essayez de retrouver quelque chose qui ressemble au système solaire.
   Masse_lune = 0,0123 * Masse_terre
   Masse_soleil = 330000 * Masse_terre

4. Faites des tests avec des dizaines de particules.



LIFAMI – TP : Astéroïdes

Objectifs : Mécanique du point, gravité entre astres et codage en C/C++



Nous souhaitons réaliser un mini-jeu. La fenêtre de jeu est composée en bas au centre d'un lanceur de soucoupes volantes, en haut de la planète à atteindre et au milieu d'un champ d'astéroïdes qui influence la trajectoire de la fusée. L'utilisateur peut régler le déplacement de la soucoupe en allumant des réacteurs dirigés vers la souris : une force dirigée vers la souris sera appliquée quand l'utilisateur cliquera. L'utilisateur ne dirige qu'une soucoupe à la fois et doit passer entre les astéroïdes. Une soucoupe est assimilée à un point. Nous supposons que vous disposez déjà du code de mouvement de la soucoupe avec les particules :

```
struct Vec2 { float x,y ; } ;   + opérator
struct Particle
{
        Vec2 p ;      // Position (x,y)
        Vec2 v ;   // vitesse (x,y)
        Vec2 m ;      // masse
        Vec2 f ;    // force (x,y)
} ;
void initPart(Particule& part, Vec2 position, Vec2 vitesse) ;
void updatePart(Particule& part) ;
```

I. La base

1. Un astéroïde et la lune sont représentés par une position *pos* (Vec2) et un rayon *radius* qui serviront pour calculer s'il y a collision et pour afficher une image plus ou moins grande. Définissez cette structure *Asteroid.*

2. Le monde est représenté par une unique soucoupe (une Particule), un score, un tableau d'*Asteroide*s de taille MAX_ASTEROID (une constante), un nombre d'astéroïdes réellement utilisés, la cible *moon* représentée également par une structure *Asteroid*, une image pour la soucoupe, une image pour les astéroïdes, et une image pour la lune. Écrivez la structure *Word*.

3. Écrivez le sous-programme *initWorld* qui prend en paramètre la structure à initialiser. Vous initialiserez :
   - la soucoupe en bas au milieu de la fenêtre, avec une vitesse et une force nulle ;
   - la lune en haut à une position horizontale aléatoire ;
   - le champ d'astéroïdes au milieu en tirant au hasard leur position et leur rayon ;
   - les images.

4. Écrivez le sous-programme *draw* qui prend en paramètre un Monde *World* et qui l'affiche.

5. Écrivez l'algorithme principal et testez votre programme.

II. Le lanceur

1. Écrivez un sous-programme *forceLauncher* qui prend en paramètre le monde World et qui renvoie la force (un Vec2) du lanceur en fonction de la position de la souris : le vecteur force sera le vecteur entre la position de la souris et la position de la soucoupe.

2. Dans le sous-programme *draw* appelez la fonction *forceLauncher* et affichez le vecteur sous forme d'une ligne rouge sur le lanceur.

3. Ecrivez le sous-programme *update* qui applique la force précédemment calculée à la soucoupe quand l'utilisateur clique sur le bouton gauche de la souris.

4. Modifiez le programme principal pour appeler *update* après *draw*. Relancez votre programme pour le débuguer.

III. Partie sans influence

1. Écrivez le sous-programme *ovniWin* qui prend en paramètre le monde *World* et renvoie un booléen indiquant si la soucoupe est sur la cible ou non, par exemple quand la distance entre la soucoupe et la lune est inférieure au rayon de la lune.

2. Écrivez le sous-programme qui prend en paramètre la structure *World*, et qui teste pour tous les astéroïdes si la soucoupe est dedans ou non. Si la soucoupe est dans l'un des astéroïdes, le sous-programme renvoie vrai sinon faux.

3. Modifiez votre sous-programme *update* pour détecter la fin de la partie et réinitialiser la soucoupe sur le lanceur avec une force nulle.

IV. Astéroïde influencent le mouvement de la soucoupe

1. Modifiez votre programme pour que les astéroïdes appliquent une force à la soucoupe. Cette force d'influence sera proportionnelle à l'inverse de la distance entre l'astéroïde et la soucoupe et proportionnelle à la masse de l'astéroïde (son rayon). Regardez la question IV-1 du TP Particules.

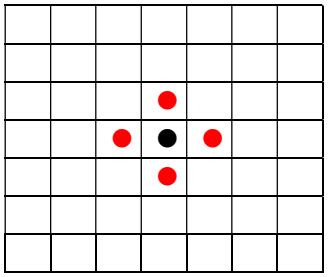
2. Vous pouvez laisser la force de gravité qui attire la soucoupe vers le bas ou non. Vous pouvez appliquer une force d'attraction par la lune. Faites au mieux pour que le jeu soit jouable : gestion des scores, astéroïdes en mouvement, etc.

LIFAMI – TD / TP : Dynamique des populations

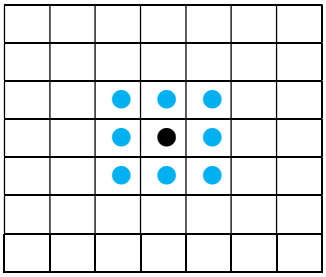
Objectifs : Applications en biologie
Manipulation des structures, des boucles, des tests, etc.

I. Automate cellulaire et jeu de la vie

Un **automate cellulaire** est une grille régulière de « cellules » contenant chacune un « état » choisi parmi un ensemble fini d'états et qui peut évoluer au cours du temps. L'état d'une cellule au temps *t+1* est fonction de l'état au temps *t* d'un nombre fini de cellules appelé son « voisinage ».



4 voisins



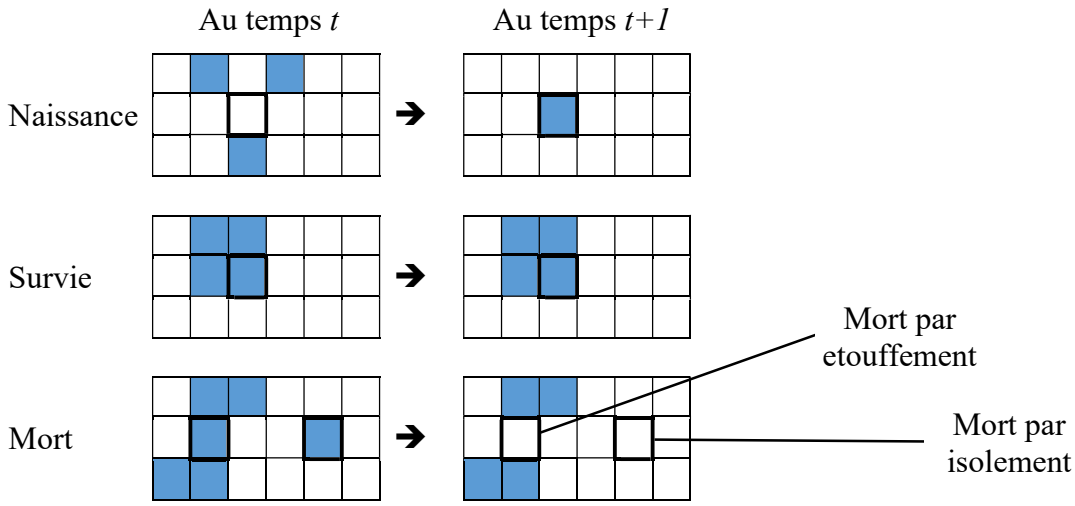
8 voisins

Le **jeu de la vie** est un automate cellulaire imaginé par le mathématicien britannique John Horton Conway en 1970 dans lequel chaque cellule peut prendre l'un des deux états « vivant » ou « mort » et est entourée de **huit** cases susceptibles d'accueillir d'autres cellules.

Les règles d'évolution entre le temps *t* et le temps *t+1* sont les suivantes.

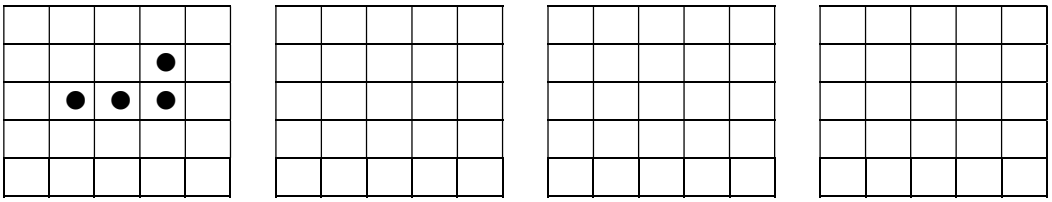
- **La survie / la stance** : chaque cellule ayant deux ou trois cellules adjacentes vivantes survit jusqu'à la génération suivante.
- **La mort** : chaque cellule ayant quatre cellules adjacentes vivantes ou plus disparaît, ou meurt, par surpopulation / étouffement. Chaque cellule n'ayant qu'une ou aucune cellule adjacente meurt d'isolement.
- **La naissance** : chaque emplacement adjacent ayant exactement trois cellules, fait naître une nouvelle cellule pour la génération suivante.

Toutes les naissances et toutes les morts ont lieu en même temps au cours d'une génération.



A faire en TD

1. En appliquant les règles énoncées précédemment, prévoyez l'évolution du système suivant.



2. Écrivez la structure `Jeu_de_la_vie` qui contiendra une grille 2D d'états (0 pour morte et 1 pour vivante), la taille de la grille utilisée `dx` et `dy`, et le nombre de cellules vivantes dans la grille `alive`.

3. Écrivez le sous-programme `init` permettant de fixer les paramètres du jeu (taille de la grille et nombre de cellules vivantes) et de remplir la grille avec des cellules mortes.

4. Écrivez le sous-programme `etat_initial` permettant de positionner aléatoirement des cellules vivantes dans la grille.

5. Écrivez enfin le sous-programme `etat_suivant` qui à partir d'une configuration calcule l'état suivant en fonction des règles énoncées en préambule.

A faire en TP en complément de ce qui précède

6. Pour permettre un affichage de la grille sous Grapic, ajoutez des champs `Img_Dead` et `Img_Alive` de type `Image` à votre structure `Jeu_de_la_vie`. Ajoutez à votre dossier `data` deux images de votre choix symbolisant la `vie` et la `mort`.

7. Implémentez en adaptant à la nouvelle structure les procédures `init`, `etat_initial`, et `etat_suivant`.

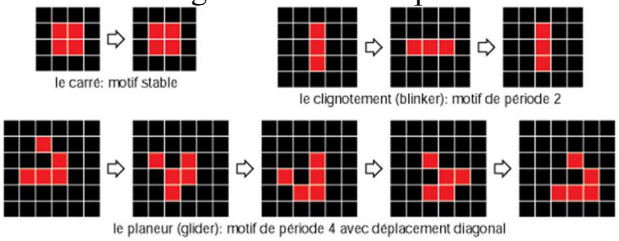
8. Écrivez la procédure d'affichage de la grille de jeu en fonction des états des cellules contenues dans la structure `Jeu_de_la_vie`.

9. Écrivez enfin la fonction principale qui appellera successivement les différents sous-programmes en laissant quelques secondes de latence entre l'affichage de deux états successifs.

Pour aller plus loin … en TP !

10. Implémentez la configuration décrite au tout début du TD et vérifiez que ce que vous aviez prédit est correct.

11. Cherchez et tester d'autres configurations "remarquables".



II. Proies et prédateurs : version analytique

L'écologie mathématique est née dans les années 1920 avec les travaux d'Alfred Lotka (1880-1949) et de Vito Volterra (1860-1940) qui ont proposé indépendamment l'un de l'autre le premier modèle décrivant une interaction de type « proie-prédateur » ou, plus généralement, de type « ressource-consommateur ».

Les équations dites de Lotka -Volterra s'écrivent fréquemment :

$$\begin{cases} \frac{\mathrm{d}x(t)}{\mathrm{d}t} &= x(t)\left(\alpha - \beta y(t)\right) \\ \frac{\mathrm{d}y(t)}{\mathrm{d}t} &= -y(t)\left(\gamma - \delta x(t)\right) \end{cases}$$

où

- t est le temps ;
- x(t) est l'effectif des proies en fonction du temps ;
- y(t) est l'effectif des prédateurs en fonction du temps ;
- les dérivées dx(t)/dt et dy(t)/dt représentent la variation des populations au cours du temps.

Les paramètres suivants caractérisent les interactions entre les deux espèces :

- α est le taux de reproduction des proies (constant, indépendant du nombre de prédateurs) ;
- β est le taux de mortalité des proies dû aux prédateurs rencontrés ;
- γ est le taux de mortalité des prédateurs (constant, indépendant du nombre de proies) ;
- δ est le taux de reproduction des prédateurs en fonction des proies rencontrées et mangées.

A faire en TD

1. Un état d'équilibre de la population est observé quand aucune des deux populations en présence n'évolue, c'est-à-dire quand les dérivées correspondantes sont nulles. Ecrivez ce nouveau système d'équation et trouvez ses solutions.

2. En appliquant les coefficients $\alpha = 0.045$, $\beta = 0.001$, $\gamma = 0.025$ et $\delta = 0.0002$ à une population initiale de 120 proies et 40 prédateurs, écrivez les équations permettant de calculer le nombre de proies et de prédateurs au temps t+1 en fonction du temps t.

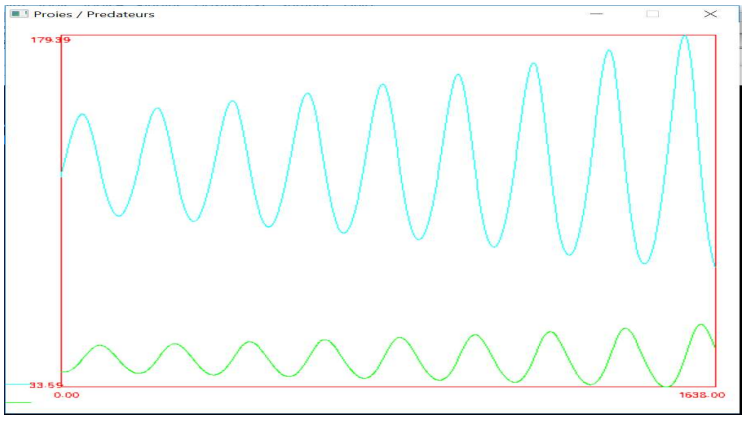
3. Nous allons maintenant décrire les structures nécessaires à l'implémentation du modèle proies / prédateurs en C/C++. Définissez une structure `Ecosysteme` qui dans sa version initiale contiendra les champs suivants : `nb_proies`, `nb_predateurs`, ainsi que les 4 coefficients réels des équations de Lotka-Volterra notés `alpha`, `beta`, `gamma` et `delta`.

4. Écrivez une fonction `evolution_ecosysteme` qui simule l'évolution de chacune des populations au cours du temps. Les résultats obtenus (nombre de proies et de prédateurs à chaque instant) seront dans un premier temps stockés dans un tableau.

A faire en TP

5. Après avoir défini les structures nécessaires à la modélisation du système proies / prédateur et initialisé les paramètres avec les valeurs précédentes, modifiez la procédure `evolution_ecosysteme` en faisant afficher à chaque pas de temps le nombre de proies et de prédateurs sur un graphique. Vous utiliserez pour cela les fonctions Grapic :
   - `plot_setSize (Plot &p, const int n)`
     qui définit le nombre de valeurs conservées (si n<0 une infinité, cas par défaut).
   - `plot_add (Plot &p, float x, float y, int curve_n)`
     qui ajoute un point (x,y=f(x)) à la courbe numéro `curve_n`.
   - `plot_draw (const Plot &p, int xmin, int ymin, int xmax, int ymax, bool clearOrNot)`
     qui dessine la courbe dans le rectangle `(xmin,ymin,xmax,ymax)`; et efface le contenu du rectangle si `clearOrNot` est à `true`.

Vous devriez obtenir les courbes suivantes :



III. Proies vs. Prédateurs : version expérimentale !

Nous allons dans cette partie simuler le système proies / prédateurs en nous basant sur des règles de mort / survie telles qu'elles se produisent (approximativement) dans la nature. L'objectif est de suivre l'évolution de chaque population au cours du temps comme dans la version analytique basées sur les équations de Lotka-Volterra et de vérifier qu'on obtient un comportement similaire. Pour chaque individu on énoncera des règles (plus ou moins complexes et réalistes) de naissance, reproduction, survie et mort (par prédation, faim, ou encore vieillesse).

A faire en TD

1. Définissez une structure `Individu` qui contiendra les champs `type_individu` (proie, prédateur ou herbe), `duree_vie` et `duree_jeune`.

2. La représentation visuelle du système se fera au moyen d'une grille 2D dans laquelle seront présent des individus (proie ou prédateur). Définissez la structure `Ecosysteme` qui contiendra une grille 2D d'individus (ainsi que les paramètres de taille de la grille `dx` et `dy`), le nombre de proies et de prédateurs et pour chaque population une image (proie, prédateur et herbe).

3. Ecrivez la procédure `evolution_ecosysteme` qui prédit le devenir de chaque individu en fonction de son environnement (voisinage) avec les deux règles suivantes.
   - Si deux proies sont dans des cases adjacentes, elles se reproduisent et donnent donc naissance à un nouvel individu de type proie placé dans une case libre.
   - Si un prédateur a dans son voisinage une proie, il la mange, s'il a un autre prédateur dans son voisinage il se reproduit uniquement s'il reste de la place dans le voisinage.

A faire en TP

4. Définissez les constantes `DUREE_VIE_PROIE`, `DUREE_VIE_PREDATEUR`, `MAX_JEUNE_PROIE`, et `MAX_JEUNE_PREDATEUR`.

5. Définissez les structures `Individu` et `Ecosysteme`. Choisissez une image pour chaque type d'individu (proie, prédateur et herbe) et enregistrez-les dans le dossier `data`. Ces images doivent stockées dans la structure `Ecosysteme`.

6. Ecrivez la procédure `init_ecosysteme` qui initialisera le système avec des valeurs par défaut (grille de taille 10 par 10, avec 40 proies et 10 prédateurs). Vous remplirez ici toute la grille avec de l'herbe puis positionnerez aléatoirement vos proies et prédateurs.

7. Ecrivez la procédure `draw_ecosysteme` qui affiche le contenu de la grille avec les images correspondantes dans une fenêtre *Grapic*.

8. Implémentez la version de base de la procédure `evolution_ecosysteme` et passez à chaque itération le nombre de proies et de prédateurs à une courbe `Plot` de *Grapic*.

9. Affichez la courbe d'évolution des deux populations et comparez-la à la courbe théorique obtenue grâce aux équations de Lotka-Volterra.

IV. Modification des règles (pour aller plus loin…)

1. Rajoutez / modifiez les règles d'évolution afin de rendre le système plus réaliste. On pourra par exemple ajouter les règles suivantes.
    a. Les prédateurs peuvent mourir de faim (au-delà de `MAX_JEUNE_PREDATEUR` itérations de jeûne, par exemple 4), ou de vieillesse (au-delà de `DUREE_VIE_PREDATEUR` itérations, par exemple 10).
    b. Les proies peuvent également mourir de faim (si plus de nourriture à proximité).

2. Donnez à chaque individu la faculté de se déplacer dans son voisinage ; les proies en sens inverse de leurs prédateurs et les prédateurs en direction des proies.

3. Comparez les nouvelles courbes aux courbes théoriques.

LIFAMI – TD / TP : Evolution de la couleur des insectes cherchant à se camoufler

Objectifs : Révision sur les structures, boucles, etc.
Savoir écrire un enchainement de plusieurs fonctions dans un but applicatif
Un exemple d'application inspirée de la biologique

Notre objectif est de programmer un mini jeu où l'ordinateur « apprend » un camouflage pour une population d'insectes se faisant dévorer par votre souris. La phase d'apprentissage est une simulation de l'évolution génétique des gènes de couleur que pourrait faire une espèce d'insectes voulant maximiser ses chances de survie.

Initialement (gauche) les insectes ont toutes les couleurs possibles (il y a un cercle de couleur pour chaque insecte sur l'image). Après plusieurs itérations (droite) les meilleurs insectes qui ne se sont pas fait manger rapidement sont gardés pour construire la population suivante. La population a des couleurs vertes/jaunes qui offrent un meilleur camouflage.

1. Un *insecte* est représenté par une position (x,y), une couleur (r,g,b), un temps de naissance et une durée de vie. Une *population* d'insectes est représentée par un tableau de NB_INSECTS insectes et une image de fond représentant le paysage dans lequel les insectes vivent.

   Déclarez ces deux structures, ainsi qu'une structure *Color* munie des opérateurs d'addition, de soustraction, de multiplication et de division par un réel.

2. Écrivez les deux procédures suivantes d'initialisation du monde des insectes.

   ```
   void initInsect(SomeInsects& si, Color good, int range)
   ```
   →Initialise les insectes. Leur position est choisie au hasard. Leur couleur sera choisie au hasard dans un rayon *range* autour de la couleur *good*. Le champ de la durée de vie est initialisé à -1 : un chiffre négatif signifie que l'insecte est toujours vivant, un positif indique combien de temps il a vécu.

   ```
   void init(SomeInsects& si)
   ```
   →Initialise l'image du paysage et appelle la procédure qui initialise les insectes.

3. Écrivez la procédure *draw* qui prend en paramètre la population d'insectes et l'affiche, ainsi que l'image de paysage. Chaque insecte est un cercle plein de rayon 3 avec la couleur stockée dans la structure.

4. Écrivez la procédure *Update* avec 2 aspects : mort d'un insecte (question 5), régénération de toute la population d'insectes (question 6).

5. Les insectes dans un rayon de 20 pixels de la souris sont mangés. La souris est le prédateur. Un insecte mort n'est plus visible et aura son champ de durée de vie qui contiendra la durée qu'il a vécu avant de se faire manger par la souris. Utilisez :
   `ElapsedTime()` qui renvoie la durée depuis le lancement du programme.

6. Une fois tous les insectes morts nous allons garder les insectes les mieux adaptés à leur environnement, et sélectionner leur couleur pour régénérer une population.
   a. Écrivez la procédure *minMaxLifeTime* qui trouve la durée de vie minimale et la durée de vie maximale dans une population d'insectes.
   b. Écrivez la fonction *averageColorOfGoodInsects* qui calcule la couleur moyenne des insectes dont la durée de vie a été supérieure à une certaine durée.
   c. Dans une procédure *update* ajoutez du code qui régénère une nouvelle population avec une couleur mieux adaptée calculée par les questions a. et b.

De nombreuses améliorations sont possibles …

L'algorithme principal pourra ressembler à ceci :

```
int main(int , char** )
{
    bool stop=false;
    winInit("Interpolation !", DIMW, DIMW);
    backgroundColor( 240, 230, 255 );
    Menu menu;
    menu_add( menu, "Init");
    menu_add( menu, "Run");
    menu_setSelect( menu, 1);

    SomeInsects li;
    init(li);

    while( !stop )
    {
        winClear();
        switch(menu_select(menu))
        {
            case 0 : init(li); menu_setSelect(menu, 2); break;
            default: break;
        }
        draw(li);
        update(li);
        menu_draw( menu );
        stop = winDisplay();
    }
    winQuit();
    return 0;
}
```

LIFAMI – TD : Dérivée, intégrale et applications à l'économie

Objectifs : Manipuler les notions d'intégrales, de dérivées
Savoir écrire un enchainement de plusieurs fonctions dans un but applicatif

I. Prix du pétrole

Des économistes modélisent le prix P du pétrole en fonction de la quantité Q de pétrole disponible dans les stocks (exprimée en millions de barils). Leur raisonnement ne porte pas sur le prix du pétrole directement mais sur sa variation. Leur modèle indique que les marchés financiers vont faire augmenter le prix du pétrole s'ils pensent que les stocks deviennent plus faibles qu'une certaine valeur S (seuil psychologique) et diminuer le prix s'ils pensent que les stocks dépassent cette valeur.

$$P'(t) = \frac{dP}{dt} = Q - S$$

$$Q'(t) = \frac{dQ}{dt} = Q_{produit} - Q_{consommé}$$

Nous rappelons que

$$\frac{df}{dt} = \lim_{dt \to 0} \frac{f(t) - f(t - dt)}{dt} \sim \frac{f(t) - f(t - dt)}{dt}$$

1. Ici nous choisissons dt = 1 mois et exprimons $Q_{produit}$ et $Q_{consommé}$ en barils/mois. Écrivez le prix du baril en fonction du prix du baril du mois précédent, de la valeur seuil S, et de $Q_{produit}$ et $Q_{consommé}$.

2. Avec par exemple, la valeur seuil S=150 millions de barils, le prix du baril au temps 0 est P_0=\$50 et la quantité de pétrole disponible au temps 0 est Q_0=150. La quantité produite est un nombre aléatoire entre 8 et 12, la quantité consommée est un nombre aléatoire entre 10 et 14. Écrivez la fonction qui calcule le prix du baril après N mois.

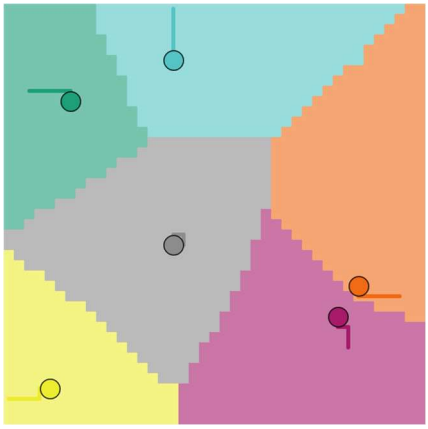
```
const float S = 150.0 ;
const float P0 = 50.0 ;
const float Q0 = 150.0;

float Qproduite()
{
  …
}

float Qconsomme()
{
  …
}
```

II. Préparez les questions du TP « Marchands de glaces »

LIFAMI – TD / TP : Simulation économique des marchands de glaces

Objectifs : Info/Math, des outils pour l'économie ?
Manipulation des structures, des boucles, etc.

La **loi de Hotelling** affirme que sur la plupart des marchés, la concurrence conduit les producteurs à réduire la différence entre leurs produits. Cette loi est aussi appelée **principe de différenciation minimale**. Elle a été formulée par le statisticien et économiste américain Harold Hotelling (1895-1973) dans un article intitulé *Stability in Competition. (Source Wikipedia : https://fr.wikipedia.org/wiki/Loi_de_Hotelling).*



Le **problème des marchands de glaces ambulants** est un exemple célèbre de la théorie des jeux, et une approche simplifiée du modèle de Hotelling.

Soit une zone géographique où viennent s'installer N marchands de glaces. Pour simplifier, tous les marchands vendent la même glace, exactement identique mais pas au même prix. Un client choisit son marchand en minimisant la fonction « distance + prix ».
A chaque tour un marchant se pose deux questions :
- bouger son stand avec 4 choix possibles : x+1 ou x-1 ou y+1 ou y-1 ;
- monter ou baisser son prix de 10%.

Chaque marchand teste les 4 positions, les compare à la position actuelle et change s'il augmente ses rentrées. Il fait de même pour les prix.

1. Un marchand est représenté par une position, un prix de vente, une couleur (ceci pour l'affichage) et le nombre de clients. Pour la position, vous pouvez récupérer la structure *Point* des TP précédents. On retrouve la rentrée d'argent d'un marchand en multipliant le nombre de clients par le prix de vente.

   LesMarchands sera une structure comportant un grand tableau de *Marchand* et la taille réellement utilisée dans ce tableau.
   En algo :
   structure LesMarchands
   mar : Tableau[MAX] de Marchand
   nm : Entier
   FinStructure

Écrivez les deux structures *Marchand* et *LesMarchands* comportant tous les marchands de la région.

2. Écrivez la procédure *init* qui initialise aléatoirement tous les marchands. Les marchands auront une position dans la fenêtre *Grapic* de taille DIMW x DIMW. Les prix seront compris entre 2 et 6 euros.

3. Écrivez la procédure *drawMarchands* qui affiche un cercle rouge de rayon 2 pixels à la position de chaque marchand.

4. Écrivez la procédure *TraiteVente* qui traite une zone géographique. Avec *Grapic*, cette zone sera toute la fenêtre. En initialisation, vous mettrez à 0 le nombre de clients de chaque marchand. Chaque pixel de la fenêtre sera un client qui prendra comme couleur celle du marchand qu'il choisit. Rappel : un client choisit le marchand qui minimise la somme distance + prix. Chaque fois qu'un client choisit un marchand, il faut ajouter le prix de la glace à la rentrée d'argent du marchand.

   Pour chaque pixel, vous appellerez la procédure *put_pixel* pour colorier le pixel avec la couleur du marchand choisi :
   ```
   Proc put_pixel(x,y : Entier, r,g,b : Entier)
   ```

   Remarque : n'appelez pas la fonction *Grapic* d'affichage *winDisplay* dans cette procédure car cette procédure pourra être utilisée pour faire des calculs intermédiaires lors de l'évolution des positions/prix des marchands à la question suivante.

5. Écrivez la procédure *MiseAJourMarchands* qui va faire évoluer chaque marchand. Pour cette procédure vous devez avoir deux structures *LesMarchands*, celle pour avant la mise à jour et celle qui sera mise à jour. En effet, tous les marchands doivent faire leur évolution de manière simultanée et non séquentiellement.

6. Ajoutez le code qui permet d'afficher sous forme de courbes l'évolution des prix de tous les marchands. Faites des observations avec différentes initialisations. Quand les prix sont-ils les plus bas ?