Exercice 1. Bézier unidimensionnelle.

On rappelle qu'une courbe de Bézier peut s'exprimer sous la forme:
$p(s)= (1-s)^3 P0 + (1-s)^2 s P1+(1-s)s^2 P2 + s^3 P3$,

avec s, un paramètre scalaire variant dans [0,1], et [P0,P1,P2,P3] le polygone de contrôle de la courbe de Bézier.

On souhaite construire une classe `bezier` qui permet de manipuler ce type de courbe.
On placer l'en-tête de cette classe dans le fichier `bezier.hpp`, et l'implémentation dans `bezier.cpp`.

- ➔ Générez une classe nommée `bezier` qui contiendra en tant que donnée privée un tableau (taille statique) de 4 floats.

- ➔ Le code qui suit correspond à la fonction `main` qui manipule la classe `bezier`. Réfléchissez aux méthodes et/ou fonctions nécessaires à ajouter pour que ce code soit valide. Implémentez ces méthodes et/ou fonctions associées à votre classe de Bézier, et vérifier que ce code donne bien le résultat attendu.

```
#include "bezier.hpp"

int main()
{
    //appel au constructeur vide
    bezier b0;
    //appel a un constructeur donnant directement [P0,P1,P2,P3]
    const bezier b1(0.0f,1.0f,1.1f,0.15f);

    float P0=b1.coeff(0); //recuperation de P0
    float P1=b1.coeff(1); //recuperation de P1

    std::cout<<P0<<std::endl; //doit afficher 0
    std::cout<<P1<<std::endl; //doit afficher 1

    //mise en place des coefficients
    b0.coeff(0)=0.0f;
    b0.coeff(1)=0.4f;
    b0.coeff(2)=0.6f;
    b0.coeff(3)=0.2f;

    //doit afficher en ligne de commande:
    // (1-s)^3*0+3s(1-s)^2*1+3s^2(1-s)*1.1+s^3*0.15
    std::cout<<b1<<std::endl;
```

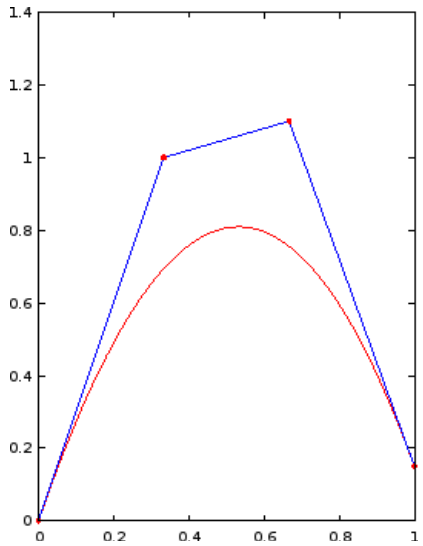
```
    //doit afficher en ligne de commande
    // (1-s)^3*0+3s(1-s)^2*0.4+3s^2(1-s)*0.6+s^3*0.2
    std::cout<<b0<<std::endl;

    //calcul d'echantillons de la courbe
    int N_sample=10;
    for(int k=0;k<N_sample;++k)
    {
        //s varie sur [0,1] avec N_sample echantillons
        float s=static_cast<float>(k)/(N_sample-1);

        //calcul de p(s)
        float valeur=b1(s);

        //verifiez en particulier que:
        // valeur=P0 pour s=0
        // valeur=P3 pour s=1
        std::cout<<k<<" : "<<valeur<<std::endl;
    }

    return 0;
}
```

➔ Ajoutez à votre répertoire les fichiers fournis dans les données: `export_matlab.hpp`, `export_matlab.cpp`, et `viewer.m`.

➔ Incluez le fichier `export_matlab.hpp` dans votre fichier principal, et appelez dans la fonction main()
`export_matlab("data.m",b1);`

➔ Observez que l'exécution de ce code créé bien un fichier `data.m` dans le répertoire courant.

➔ Lancez Matlab (ou Octave) dans ce répertoire. Et appelez `viewer` (fichier `viewer.m`). Vous devriez visualiser votre polygone de contrôle et la courbe de Bézier associée comme illustré sur la figure de droite. (les abscisses sont supposés s'incrémenter de manière régulière entre 0 et 1).



Exemple de courbe de Bézier obtenue après appel à viewer.m.

Exercice 2. Classe template.

Dans l'exercice 1, nous avions supposé que les données P0,P1,P2 et P3 était des données scalaires (float). On peut définir une courbe de bézier dans le plan 2D si les P sont définis comme des vecteurs du plan (x,y), ou bien en 3D si les P sont définis comme des positions 3D (x,y,z). De même, il est possible d'étendre la notion de courbe de Bézier à toute dimension.

Nous proposons d'étendre votre fonction de courbe de Bézier à l'application en toute dimension et à tout type de variable (float, double, long double, ...).
Pour cela, le paramètre P ne sera plus définit de manière stricte comme étant un float, mais nous allons utiliser une définition sous forme de **template**.

➔ Dans un autre répertoire, générez le fichier `bezier.hpp`. Cette fois, définissez la classe `bezier` comme étant une classe template dont le type du polygone de contrôle sera donné au moment de la création de la classe. Notez que l'on aura toujours un tableau de 4 cases, mais dont le type est définit par un template. On aura par exemple un code de ce type

```
template <typename T>
class bezier
{
private:
    T P[4];
};
```

➔ Etant donnée que la classe est définie comme un template, il n'est plus possible de définir le corps des fonctions dans un fichier .cpp à part. Nous définirons donc l'implémentation des fonctions template directement dans le fichier `bezier.hpp`.

➔ Complétez cette classe avec les même méthodes et fonctions que précédemment, en ayant cette fois un type template.

Vérifiez **au fur et à mesure** que vous puissiez utiliser votre classe dans une fonction main() en définissant votre classe comme étant
`bezier<float> nom;`

Attention: N'attendez pas d'avoir codé totalement votre classe avant d'essayer de l'utiliser dans une fonction main() !! Vous risquer d'avoir beaucoup d'erreur, et de devoir reprendre l'ensemble de vos fonctions!

Notez que l'on ne connaît pas les propriétés du type ou de la classe (T) qui sera utilisée pour le template. Mais on supposera qu'il devra vérifier les propriété suivantes pour que le code compile:

- Multiplication par un scalaire.
- Addition interne (avec le même type).
- Envoie possible dans un flux ostream& (affichage avec std::cout<< par exemple).

Notez également que le paramètre template devra de préférence être passé comme référence constante plutôt que par copie car il pourra s'agir de classes (autre que float ou double).

➔ Assurez-vous que le code de la fonction main précédente puisse être utilisé avec votre nouvelle classe implémentée sous la forme bezier<float>.

Exercice 3. Courbe de Béziers 2D.

Nous allons utiliser la classe de Bézier template afin que celle-ci puisse servir directement pour réaliser des courbes 2D. Nous allons donc désormais considérer des Bézier du type `bezier<vec2>`, avec `vec2` désignant un point du plan (x,y).

➔ Créez une struct de type `vec2` telle que les méthodes et/ou fonctions associées à celles-ci rende le code suivant valide:

```
#include "vec2.hpp"

#include <iostream>
using std::cout;
using std::endl;

int main()
{
    vec2 v0;
    vec2 v1(1.5f,2.0f);

    v0.x=2.2f;
    v0.y=4.3f;

    cout<<v0<<endl; //doit afficher [2.2;4.3]

    v1*=0.5f;
    cout<<v1<<endl; //doit afficher [0.75;1]

    const vec2 v2(1.3f,2.2f);
    vec2 v3=1.5f*v2*2.2f;
    cout<<v3<<endl; //doit afficher [4.29;7.26]

    v3=v2/0.5f;
    cout<<v3<<endl; //doit afficher [2.6;4.4]

    v3+=vec2(0.5f,0.2f);
    cout<<v3<<endl; //doit afficher [3.1;4.6]
    cout<<v2+vec2(0.5f,0.2f)<<endl; //doit afficher [1.8;2.4]

    v3-=vec2(0.4f,0.5f);
    cout<<v3<<endl; //doit afficher [2.7;4.1]
    cout<<v2-vec2(1.1f,0.4f)<<endl; //doit afficher [0.2;1.8]

    //doit afficher 1.80278
    std::cout<<norm(vec2(1.0f,1.5f))<<std::endl;

    return 0;
}
```

Notez que cette struct donne directement un accès publique aux paramètres x et y. Ce choix consiste à supposer que la classe vec2 doit être simple et légère à utiliser. Cela implique que cet accès direct au travers du code impose que x et y n'évoluerons pas au niveau du code, ils resteront toujours deux paramètres de type `float`.
(Dans le cas, ou l'on souhaiterait se laisser la liberté de faire évoluer le code, ils pourrait être

avantageux de considérer des paramètres privés ou l'accès serait réaliser par set/get. Mais on ne considérera pas cette option la dans la suite du programme.)

➔ Une fois que votre struct `vec2` est terminée, vérifiez que vous soyez capable d'instancier des classes de type `bezier<vec2>`.

➔ Utilisez les fichiers `export_matlab` des données de cet exercice, et appelez la fonction d'export sur une courbe de type `bezier<vec2>`.
On pourra par exemple utiliser le code suivant:

```
bezier<vec2> b(vec2(0,0),vec2(1,1),vec2(0,1.5),vec2(-1,0.5));
export_matlab("data.m",b);
```

Observez le résultat au visualiseur, en appelant `viewer` depuis Matlab (/Octave).

Exercice 4. Interaction avec une interface Qt

➔ Prenez les fichiers de l'exercice 4. Il s'agit cette fois d'un ensemble de fichier réalisant une interface `Qt` permettant de manipuler interactivement votre courbe de Bézier.
Pour que le programme compile, vous devez ajouter vos fichiers: `vec2.hpp`, `vec2.cpp` et `bezier.hpp` dans le répertoire `bezier/`.

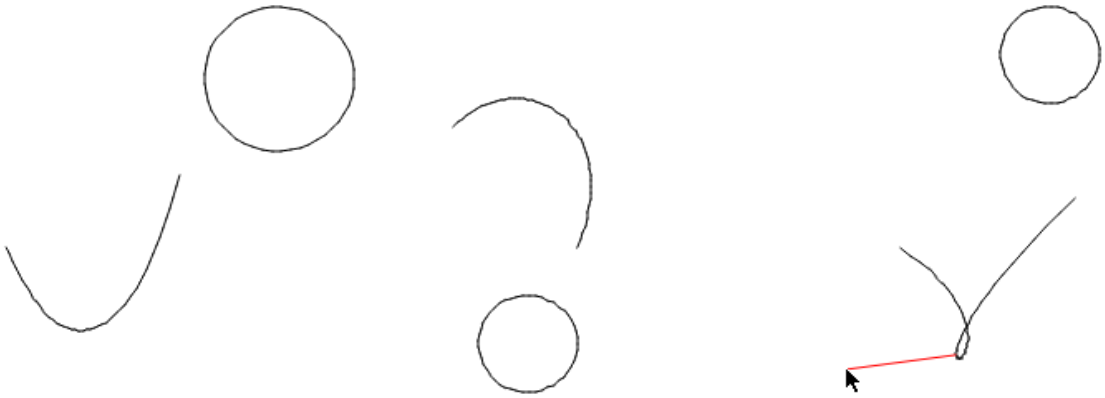
➔ Lancez ensuite la compilation (à l'aide du fichier `.pro` avec `QMake`, ou `CMakeLists.txt` avec `CMake`).
Notez que ce programme viens utiliser plusieurs méthodes de vos classes, si celles-ci ne respectaient pas le cahier des charges, elle ne permettront pas au programme de compiler.

➔ Observez que vous puissiez manipuler interactivement votre courbe de Bézier.

Exercice 5. Polymorphisme

Nous considérons désormais une scène 2D où sont placés des objets géométriques de natures différentes. Dans notre cas, on supposera qu'une scène pourra être constituée de cercles et de courbes de Bézier.

Lorsque l'utilisateur désigne un endroit de la scène, on souhaite connaître le point de l'objet le plus proche, et dessiner le segment reliant la sélection de l'utilisateur à ce point de l'objet.



Exemple de scène contenant des arcs de courbes de Bézier et des cercles. Le segment rouge indique le chemin reliant la souris au point le plus proche par rapport à tous les objets.

L'algorithme de recherche du point le plus proche parmi l'ensemble des objets est le suivant:

```
p0 : point sélectionné par l'utilisateur
dist_min=infini
Pour tous les objets i de la scène
    pi : point de l'objet i le plus proche de p
    dist_i : distance entre pi et p
    Si di<dist_min
       dist_min=di
       p_plus_proche=pi
return p_plus_proche
```

Cet algorithme nécessite que l'on puisse connaître pour un point p quelconque du plan, le point pi le plus proche de p d'une forme géométrique (cercle ou courbe de Bézier).

➔ Soit un cercle de centre c et de rayon R. Soit p un point quelconque du plan. Quelle est l'expression du point pi le plus proche de p appartenant à ce cercle?

(On définira dans la suite, une classe de cercle implémentant cette évaluation de point le plus proche.)

Dans le cas de la courbe de Bézier, on utilisera une approche discrète approchée. Pour cela, on calculera N échantillons de la courbe, et on considère que le point pi le plus proche de la courbe de Bézier est donné par l'échantillon le plus proche du point p.

Afin d'avoir une scène générique, nous souhaitons placer tous les objets géométriques dans un même conteneur, ce cette manière, il sera possible d'étendre aisément la scène à d'autres types de figures géométriques.
Par contre, l'implémentation de la fonction de calcul du point le plus proche est différente si l'on considère un cercle, ou si l'on considère une courbe de Bézier.
Pour n'avoir qu'un seul appel générique, nous allons utiliser une approche **polymorphe**. La classe `cercle` et la classe `bezier` vont donc hériter d'une même classe parente permettant l'évaluation générique du point le plus proche. On nommera cette classe parente `objet_geometrique`.

➔ Implémentez la méthode `closest_point` de la classe `bezier`. Cette méthode prendra en argument un `vec2` (en référence constante) et retournera le point le plus proche sous forme de `vec2`. Cette méthode est constante car elle ne modifie pas les attributs de la classe `bezier`.

➔ Faites en sorte que votre classe `bezier` dérive d'une classe générique `objet_geometrique`. Faite en sorte que la classe `objet_geometrique` permette de rendre la méthode `closest_point` polymorphe, ainsi que l'évaluation d'un point de la courbe de Bézier en fonction de son paramètre (surcharge de l'opérateur()).

➔ Implémentez la classe `cercle` qui dérivera également de `objet_geometrique`. Un cercle sera définit par un centre c et un rayon R. Votre cercle devra posséder au moins une méthode permettant de calculer le point le plus proche, ainsi que d'évaluer un point du cercle suivant un paramètre s variant entre 0 et 1. On pourra supposer pour cela que votre cercle est paramétré par
`c+R*(cos(2 pi s),sin(2 pi s)).`

➔ Vérifiez sur quelques exemples simples le comportement polymorphe de vos classes.

Exercice 6. Point le plus proche et interface Qt

➔ Prenez les fichiers de l'exercice 6. Il s'agit cette fois d'un ensemble de fichier réalisant une interface Qt qui présente une scène formée d'un ensemble de cercles et de courbes de Bézier. Lors d'un clic souris, le plus le plus proche est affiché.
Pour que le programme compile, vous devez ajouter vos fichiers: `vec2.hpp`, `vec2.cpp` , `bezier.hpp, cercle.hpp, cercle.cpp, et objet_geometrique.hpp` dans le répertoire `bezier/`.
Vérifiez le bon comportement de ce programme.

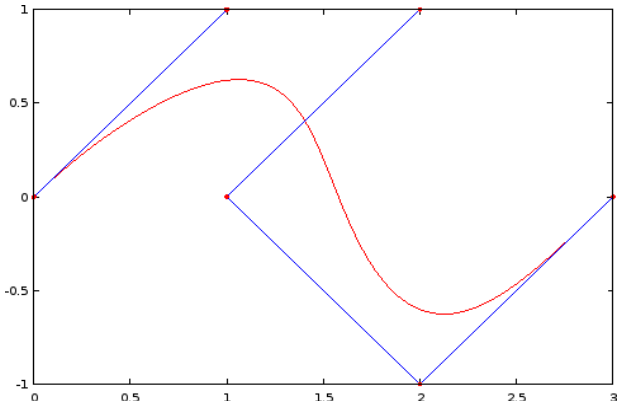
Exercice 7. Ordre générique de la courbe.

➔ Rappelez la relation entre C^n_k, C^n_{k-1} et C^{n-1}_{k-1}

➔ Créez une fonction qui ait la capacité de calculer les valeurs des C^n_k au moment de la compilation. On pourra utiliser le mot clé `constexpr`.

➔ Modifiez votre classe de Bézier afin que celle-ci ait un ordre donné (un ordre n correspond à un polygone de contrôle de n+1 points) au moment de la compilation. La classe prendra donc désormais 2 paramètres templates: un type, et un entier donnant le degré du polynôme.

➔ Adaptez la fonction `export_matlab` afin que celle-ci puisse afficher une courbe de bézier dont la taille du polygone de contrôle est caractérisé par un paramètre template.
Faites en sorte que l'évaluation des N points de la courbe de Bézier à afficher soit réalisé en parallèle (les valeurs de la courbes seront temporairement stockés dans un vecteur avant d'être écrits dans le fichier dans l'ordre). Vérifiez visuellement que votre courbe correspond bien à une Bézier du degré fixé.



Exemple de courbe de Bézier de degré 5 et son polygone de contrôle.