

Rapport projet info - POINET Léo

Sujet 6 : Traceur de courbe


Il s'agit d'écrire un programme permettant de visualiser le tracé d'une courbe dans la console. Bien que la console soit par défaut destinée à afficher des caractères, on peut avec quatre instructions supplémentaires afficher des points (appelé pixel) dans la console. Vous écrirez une classe CPolynome, permettant de stocker un polynôme de degré quelconque, vous permettrez à l'utilisateur de définir plusieurs polynômes dans la fonction principale, puis vous afficherez la ou les courbes point par point dans la console.

Le problème

Le premier problème que l'on rencontre est de savoir comment afficher un pixel à une position donnée dans la console. Avec une rapide recherche sur le net, on découvre la méthode **SetPixel()** appartenant à la librairie **windows.h**. Celle-ci n'est disponible que sur Windows. Tous les autres OS qui ne disposent pas de cette librairie ne pourront pas exécuter le programme.

Les informations à propos de cette méthode sur la doc de Microsoft :

SetPixel function (wingdi.h) - Win32 apps


The SetPixel function sets the pixel at the specified coordinates to the specified color. COLORREF SetPixel(HDC hdc, int x, int y, COLORREF color); A handle to the device context. The x-coordinate, in logical units, of the point to be set. The y-coordinate, in logical units, of the point to be set.
 <https://docs.microsoft.com/en-gb/windows/win32/api/wingdi/nf-wingdi-setpixel?redirectedfrom=MSDN>



Les paramètres nécessitent de comprendre le *hdc*, le *COLORREF* et le *macro rgb*

Ce qui conduit à :

COLORREF (Windef.h) - Win32 apps

The COLORREF value is used to specify an RGB color. typedef DWORD COLORREF; typedef DWORD* LPCOLORREF; When specifying an explicit RGB color, the COLORREF value has the following hexadecimal form: 0x00bbggrr The low-order byte contains a value for the relative intensity of red; the second byte contains a value for green; and the third byte contains a value for blue.
 <https://docs.microsoft.com/en-gb/windows/win32/gdi/colorref>

Puis :

RGB macro (wingdi.h) - Win32 apps

The RGB macro selects a red, green, blue (RGB) color based on the arguments supplied and the color capabilities of the output device. void RGB(r, g, b); The intensity of the red color. The intensity of the green color. The intensity of the blue color.

 <https://docs.microsoft.com/en-us/windows/win32/api/wingdi/nf-wingdi-rgb>



On peut alors créer un objet de type COLORREF :

```
COLORREF COULEUR = RGB(255,255,255);
```

Des exemples trouvés sur le net permettent d'arriver à cet exemple simple qui cherche à placer un pixel blanc en $x = 30$ et $y = 20$ avec x l'axe des abscisses et y l'axe des ordonnées (orienté vers le bas) :

```
#include <windows.h>
#include <iostream>
using namespace std;

int main()
{
    //choix de la couleur
    COLORREF COULEUR = RGB(255,255,255);

    SetPixel(NULL, 30, 20, RGB(255, 255, 255));
    return 0;
}
```

La compilation ne fonctionne pas et indique "undefined reference to SetPixel@16". La solution :

Faire un lien avec la librairie *libgdi32.a* du compilateur :

```
g++ .\src\main.cpp -lgdi32 -o main
```

Cependant, cet exemple n'affiche rien. Ajouter le *hdc* permet d'accéder aux paramètres de la console et ainsi d'afficher le pixel :

```
#include <windows.h>
#include <iostream>

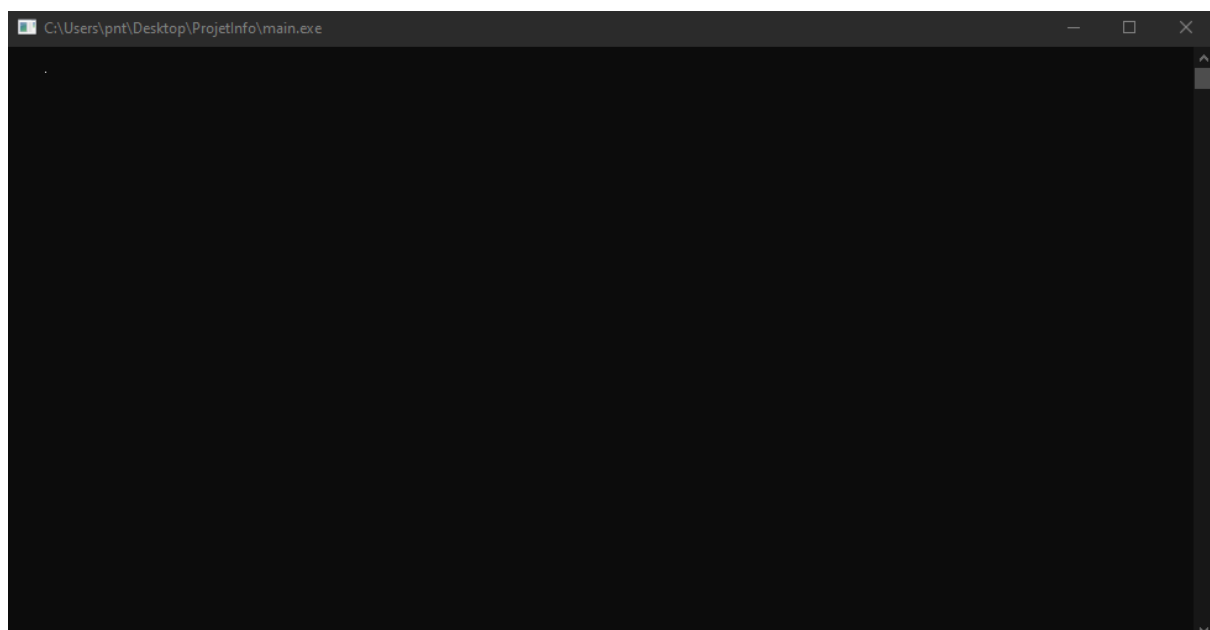
using namespace std;

int main()
{
    //initialisation de la console
    HWND fenetreConsole = GetConsoleWindow();
    HDC monDC = GetDC(fenetreConsole);

    //choix de la couleur
    COLORREF COULEUR = RGB(255,255,255);

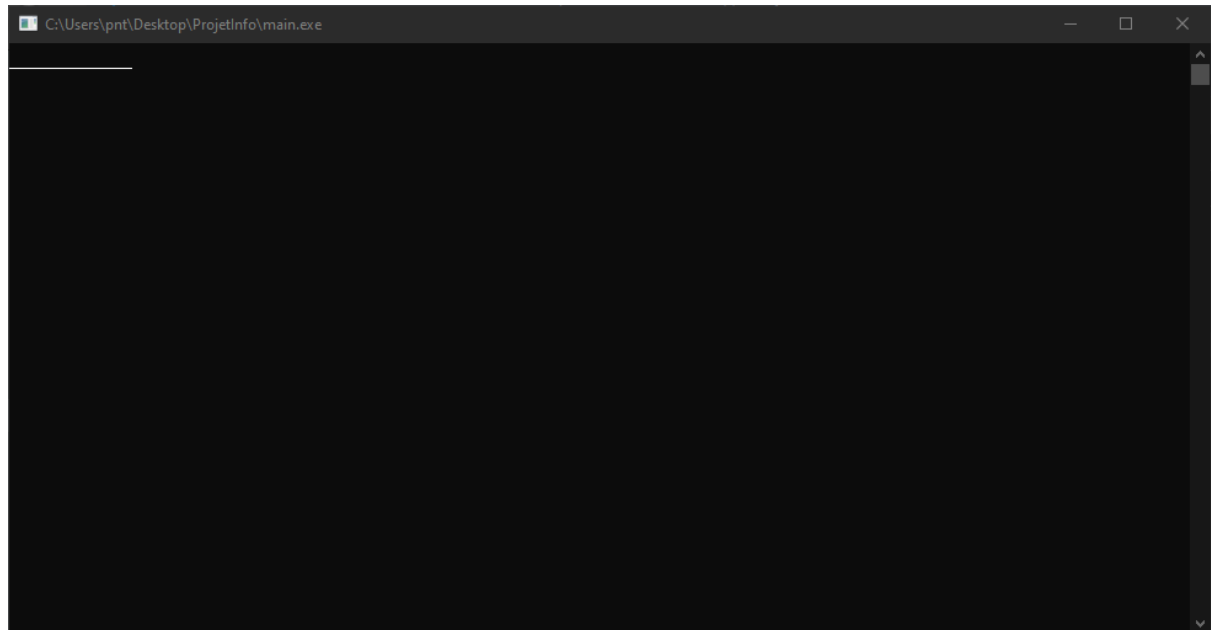
    SetPixel(monDC, 30, 20, COULEUR);

    cin.ignore();
    return 0;
}
```



On peut facilement faire un trait droit de longueur 100px avec une boucle :

```
for(int i = 0; i < 100; i++)
{
    SetPixel(monDC, i, 20, COULEUR);
}
```



Afficher une fonction simple

Maintenant que l'on sait utiliser la méthode **SetPixel()**, on veut afficher des fonctions simples.

On crée alors une fonction *f*.

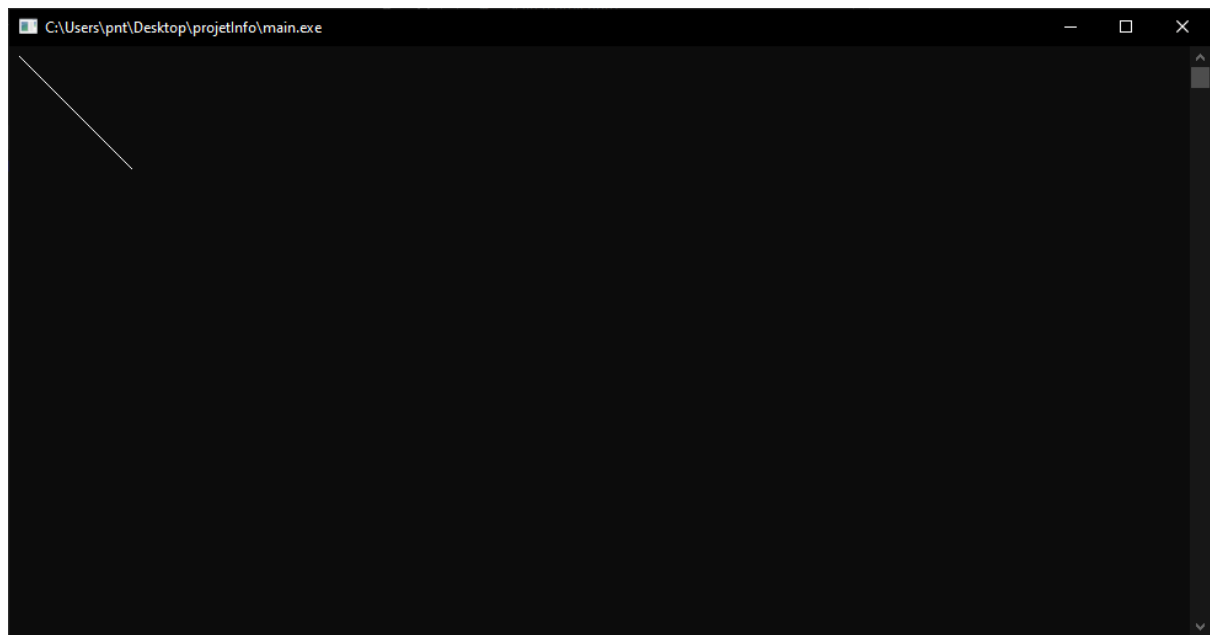
```
double f(double x) {
    return x;
}
```

Et on affiche

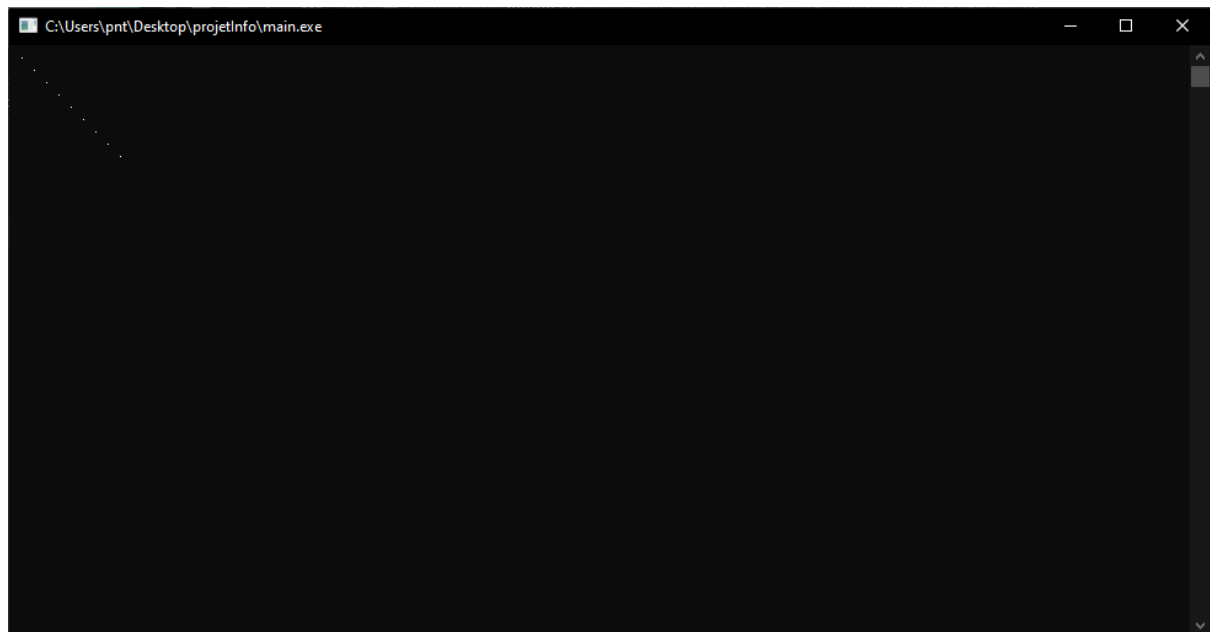
```
double precision = 1; //la precision augmente le nombre de points

for(double x = 0; x < 100; x+=precision)
{
    SetPixel(monDC, x, f(x), COULEUR);
}
```

ici on calcule 1 point/pixel :



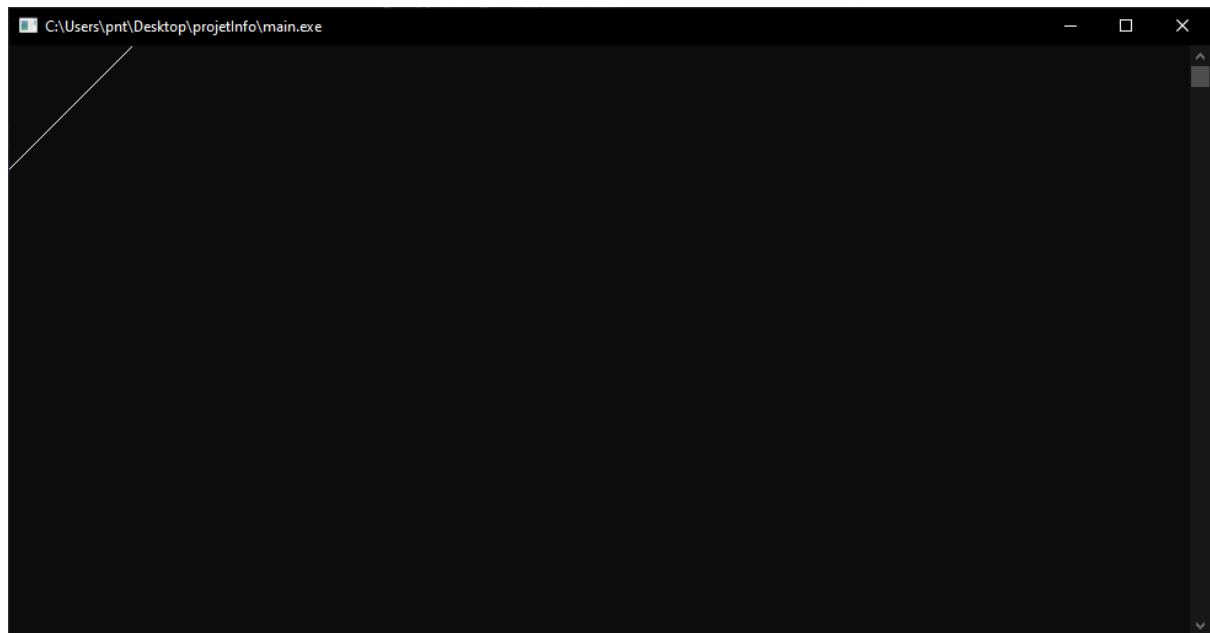
Le résultat avec une précision de 10 i.e. $1/\text{precision} = 1/10 = 0.1$ point/pixel :



On remarque de suite que la droite est dans le mauvais sens. Cela viens du fait que l'axe des ordonnées est dirigé vers le bas ! Contrairement au sens conventionnel.

On va donc inverser la courbe et introduire un décalage que l'on va appeler 'rect_y'. Sans cet offset, la courbe se dessinerait hors de la console.

```
int rect_y = 100;
for(double x = 0; x < 100; x+=precision)
{
    SetPixel(monDC, x, -f(x) + rect_y, COULEUR);
}
```



La fenêtre du graphique

Il semble intéressant d'afficher les coordonnées, non seulement pour faciliter la lecture mais aussi pour debugger les éventuels problèmes.

L'idée est la suivante : L'utilisateur choisit la taille de la fenêtre, les bornes inférieures et supérieures ainsi que le zoom et la courbe s'adapte automatiquement. On centrera l'origine au milieu de la fenêtre pour l'instant.

Création des variables (avec `offset_x` et `offset_y` les coordonnées de l'origine) :

```
int offset_x, offset_y, rect_x, rect_y, min, max;
rect_x = 600; rect_y = 400;
offset_x = int(rect_x/2); offset_y = int(rect_y/2);

min = -10; max = 10; //les bornes en x de la courbe

int zoom_y = 8; //compression en ordonnée i.e. nb de graduation sur l'axe
```

On crée alors la fonction `afficherRect()` qui permettra de construire la fenêtre dans la console :

```
void afficherRect(HDC& monDC, int& rect_x, int& rect_y, int& offset_x, int& offset_y, int& min, int& max, int& zoom_y, COLORREF
{
    for(int m = 0; m <= max-min; m++) //graduation abscisse
    {
        for(int m2 = 0; m2 <= rect_y; m2++) //grande graduation
            SetPixel(monDC, m*(rect_x/(max-min)), m2, RGB(100,100,100));

        for(int m1 = -3; m1 <= 3; m1++) //epaisseur de la petite graduation
            SetPixel(monDC, m*(rect_x/(max-min)), offset_y + m1, COULEUR);
    }

    for(int n = 0; n <= zoom_y; n++) //graduation ordonnée
    {
        for(int n2 = 0; n2 <= rect_x; n2++) //grande graduation
            SetPixel(monDC, n2, n*(rect_y/(zoom_y)), RGB(100,100,100));

        for(int n1 = -3; n1 <= 3; n1++) //epaisseur de la graduation
            SetPixel(monDC, offset_x + n1, n*(rect_y/(zoom_y)), COULEUR);
    }

    for(int i = 0; i <= rect_x; i++) //partie haute et basse
    {
        SetPixel(monDC, i, 0, COULEUR);
        SetPixel(monDC, i, rect_y, COULEUR);
    }
}
```

```

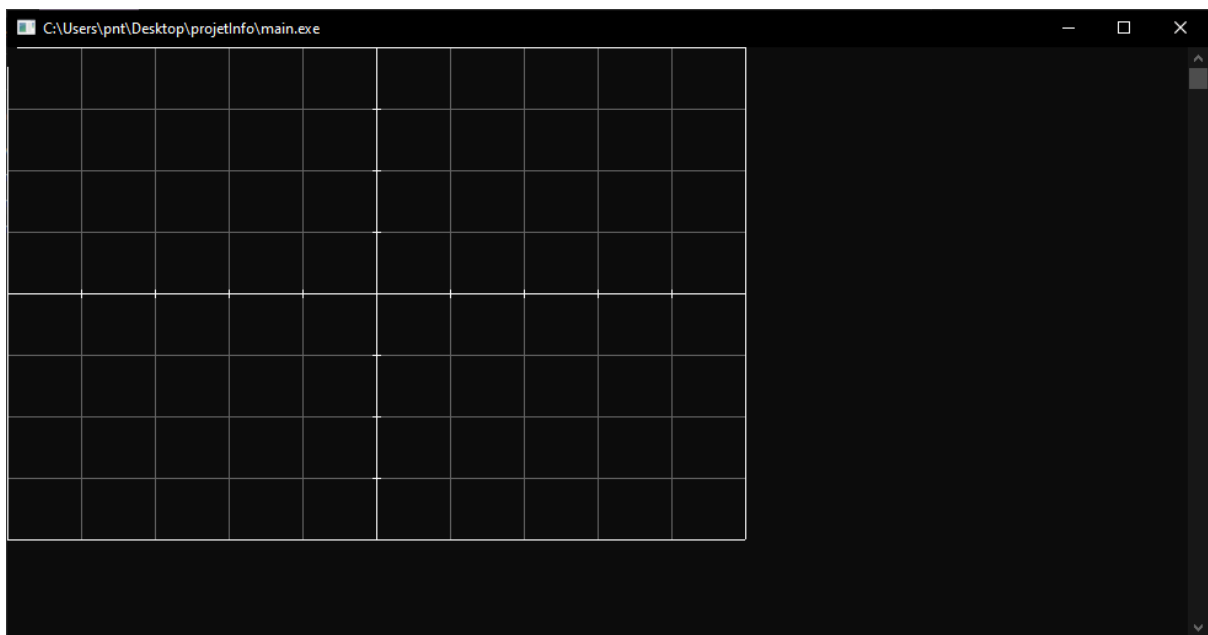
    }

    for(int j = 0; j <= rect_y; j++) //partie gauche et droite
    {
        SetPixel(monDC, 0, j, COULEUR);
        SetPixel(monDC, rect_x, j, COULEUR);
    }

    for(int k = 0; k < rect_x; k++) //axe des abscisses
    {
        SetPixel(monDC, k, offset_y, COULEUR);
    }

    for(int l = 0; l < rect_y; l++) //axe des ordonnées
    {
        SetPixel(monDC, offset_x, l, COULEUR);
    }
}

```



Le tracer de la courbe

On créer maintenant la boucle qui permet de mettre la courbe à l'échelle et de l'afficher :

```

for(double x = min; x < max; x+=precision)
{
    int yC = -(offset_y + int((rect_y/zoom_y*(f(x))))) + rect_y; //le - pour inverser la courbe
    if((offset_x + ((rect_x/(max-min)*x)) < rect_x && yC < rect_y))
        //on n'affiche pas la courbe en dehors du cadre
        SetPixel(monDC, offset_x + ((rect_x/(max-min)*x)), yC, COULEUR);
}

```

Ici, on met à l'échelle $f(x)$ en multipliant par $\text{rect_y}/\text{zoom_y}$ = nb de pixels/nb de graduations, on ajoute l'offset_y pour placer la courbe sur l'origine et enfin on met la courbe dans le bon sens.

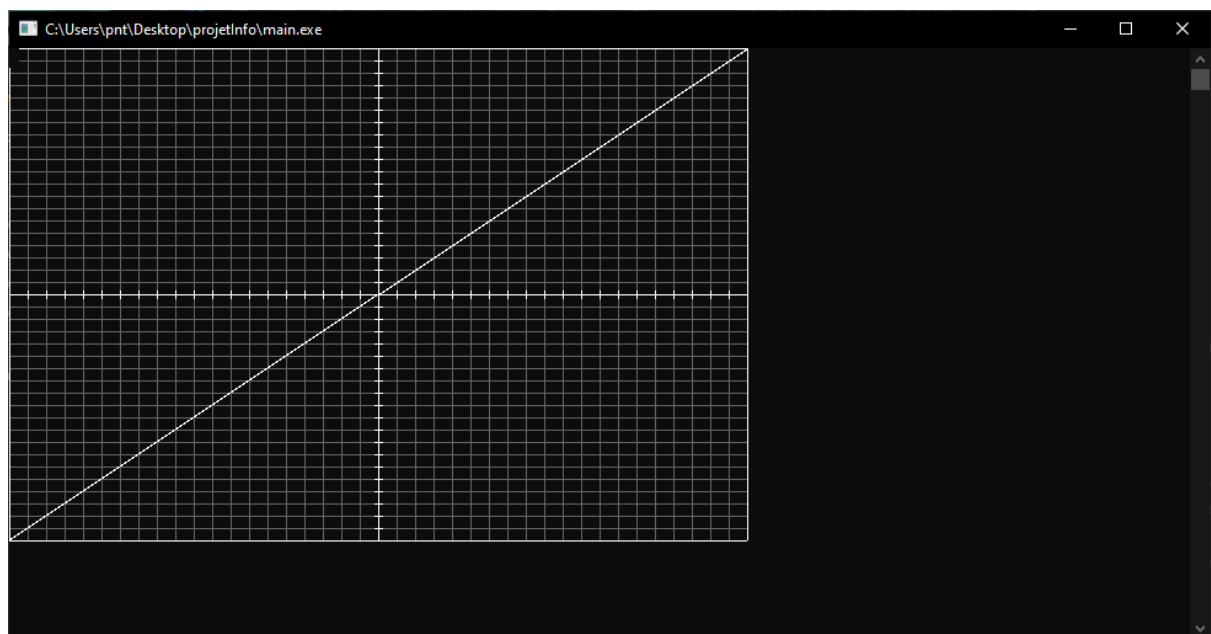
La méthode s'applique aussi pour la mise à l'échelle sur l'axe des abscisse mais cette fois-ci, le nb de graduations est l'intervall choisit par l'utilisateur.

Avec une précision de 0.04 i.e. 25 points/graduation, on a pour $f(x) = x$:

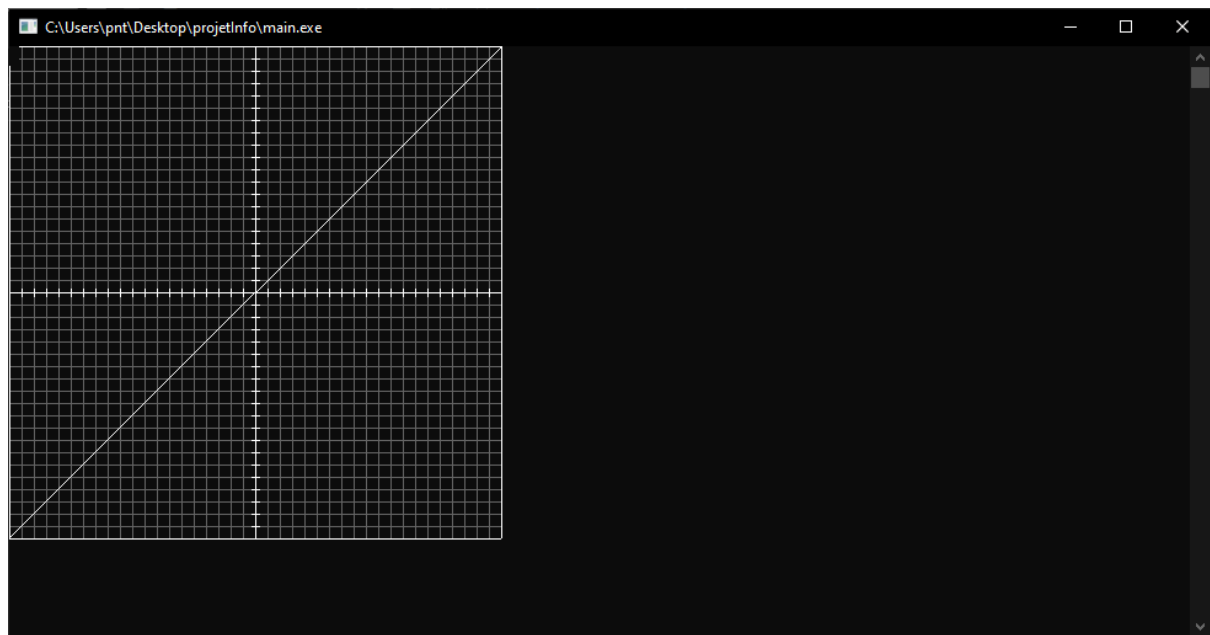


On peut faire varier simplement min, max et zoom_y pour visualiser plus de valeurs.

Ici avec min = -20, max = 20, zoom_y = 40 :



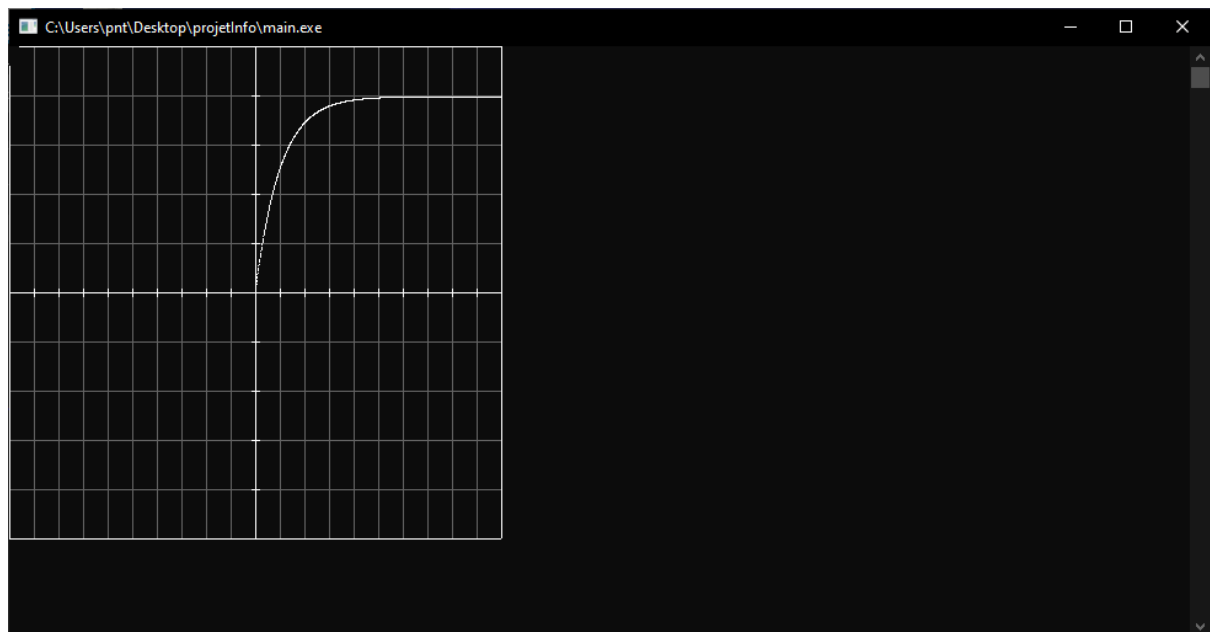
Enfin avec `rect_x = 400` et `rect_y = 400` :



On préférera des rapports 1:1, 1:2, 1:4 etc... entre `rect_x` et `rect_y`.

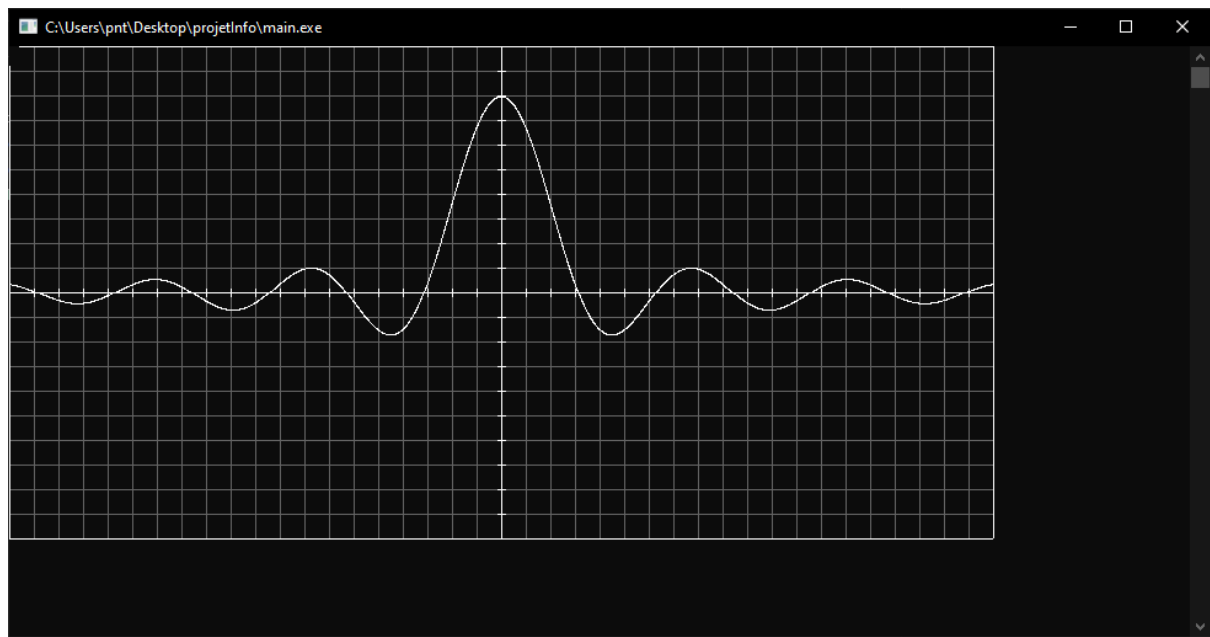
On peut dès lors s'amuser à tracer des fonctions plus complexes.

$$f(x) = -4 * e^{-x} + 4$$



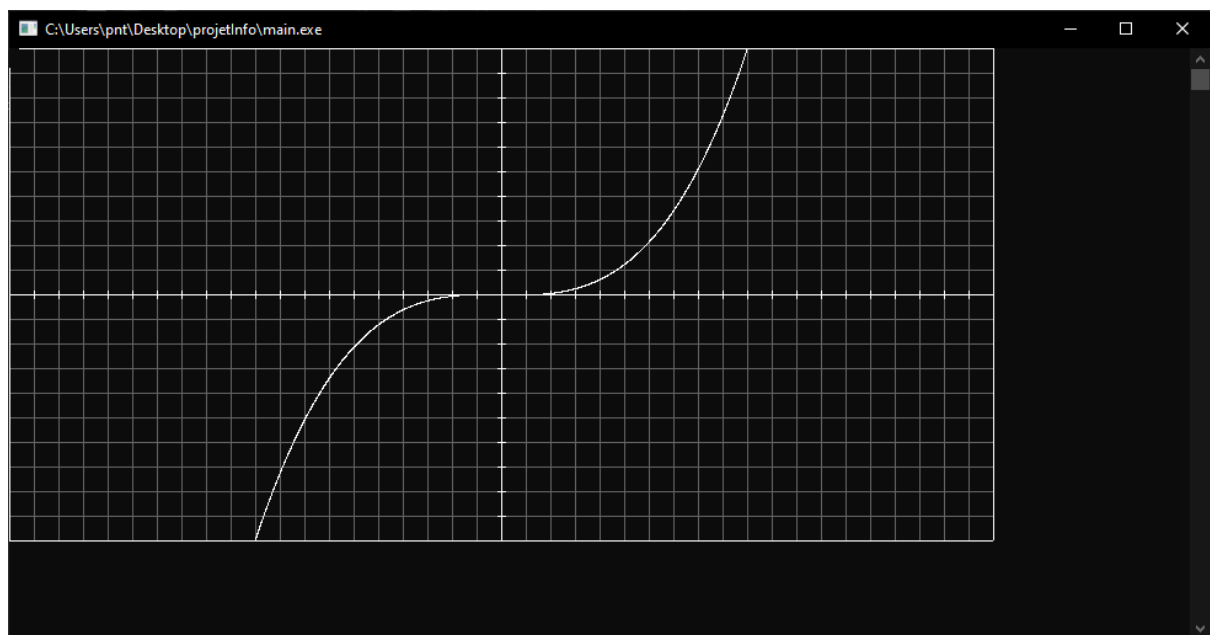
`min = 0`, `max = 20`, `zoom_y = 10`, `precision = 0.01`

$$f(x) = (8/x) * \sin x$$



$min = -20, max = 20, zoom_y = 20, precision = 0.01$

$$f(x) = 0.01.x^3$$



$min = -20, max = 20, zoom_y = 20, precision = 0.01$

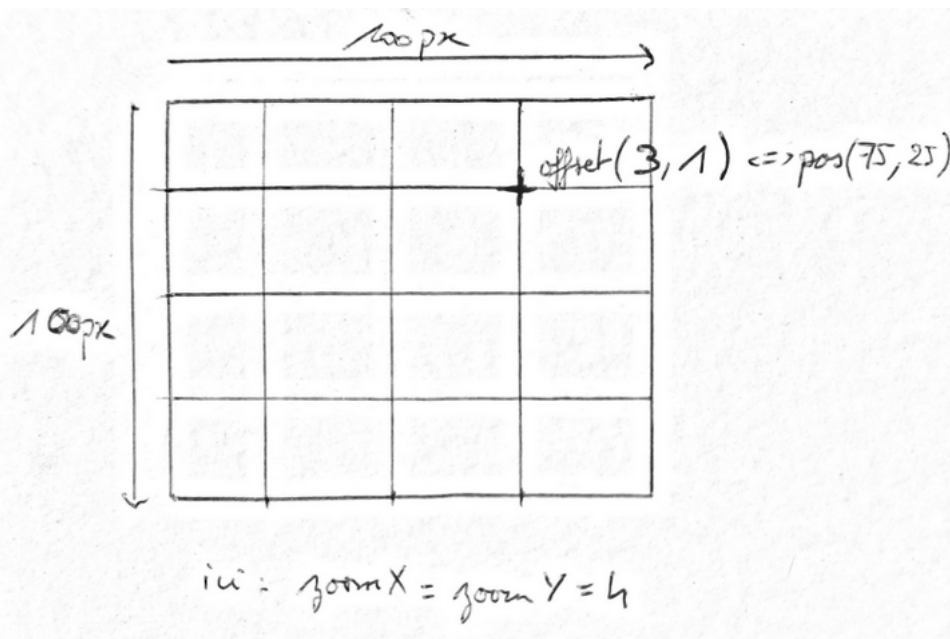
La class CFenetre

Maintenant que l'on a assimilé le principe, on peut construire une class, CFenetre, qui nous permettra d'ajouter des fonctionnalités. Cette class a besoin des attributs suivants :

```
private:
    HDC m_monDC;
    int m_rectx, m_recty; //la taille de la fenêtre
    int m_zoomx, m_zoomy; //les intervalles de calcul en x et y
    int m_offsetx, m_offsety; //les coordonnées de l'origine
    int m_rox, m_roy; //le coordonnées de la fenêtre dans la console
```

A noter que les coordonnées de l'origine seront dans la base (m_rect/m_zoom) i.e. pour une fenêtre de 100×100px si le zoom est à 10, on aura une fenêtre de 10×10 graduation. Ainsi, un offset de (1,1) aura pour effet de placer l'origine à (10,10) et un offset de (5,5) aura pour effet de placer l'origine à (50,50).

"Un bon croquis vaut mieux qu'un long discours" :



Pour ce qui est des méthodes, il ne me semble pas judicieux de détailler leur code car il s'agit surtout de calculs de coordonnées. En revanche la méthode afficherCourbe est intéressante car un de ces paramètres est une fonction.

Le paramètre se découpe ainsi :

<le type que renvoie la fonction> (pointeur vers la fonction)(<type du paramètre que prend la fonction>)

```
[class CFenetre]
public:
    //rect_x et rect_y : largeur et longueur de la fenêtre en pixel
    //zoom_x et zoom_y : amplitude en abscisse et ordonnée de la fenêtre
    //x et y : coordonnées de l'origine dans la base de zoom_x et zoom_y
    CFenetre(int rect_x, int rect_y, int zoom_x = 4, int zoom_y = 4, int x = 2, int y = 2);

    void afficherRectV2(COLORREF const& COULEUR) const;
    void afficherCourbe(double (*f)(double), double const& precision, COLORREF const& COULEUR) const;
    void afficherPolynome(CPolynome monPoly, double const& precision, COLORREF const& COULEUR) const;
    void afficherIntervalles() const; //affiche Dx et Dy

    void setRectOffset(int const& x, int const& y);
    void setOrigin(int const& x, int const& y);
    void setAmplitude(int const& zoom_x, int const& zoom_y);
```

On pourra alors appeler la méthode `afficherCourbe` de la manière suivante :

```
double g(x) {
    return 1/x;
}

int main() {
    [...]
    maFenetre.afficherCourbe(g, precision, couleur);
    [...]
}
```

La méthode `setRectOffset()` permettra, si l'utilisateur le souhaite, de positionner la fenêtre dans la console.

On prévoit aussi une méthode `afficherPolynome()` qui permettra l'affichage d'un objet de la class `CPolynome`.

La class `CPolynome`

Cette class a déjà été abordée en TP. Elle ne présente donc pas de difficultés majeurs pour le sujet.

```
class CPolynome {
public:
    CPolynome(int nbcoeff);
    ~CPolynome();

    double calcule(double const& x); //calcule P(x)

    void afficher() const;
    void saisirCoeff();

    friend std::ostream& operator<<(std::ostream& out, CPolynome const& self);
    friend std::istream& operator>>(std::istream& in, CPolynome & self);

private:
    double* m_coeff; //m_coeff
    int m_nc; //nombre de coeff
};
```

Utilisation

Il ne nous reste plus qu'à utiliser nos objets dans le `main()`. En voici un exemple :

```
#include <math.h>
#include "CFenetre.h"
#include "CPolynome.h"

#include <iostream>
using namespace std;

double f(double x) {
    return pow(x,2); //x^2
}

int main() {

    CPolynome monPoly(4); //création d'un polynome de degré 3
    cin >> monPoly; //l'utilisateur rentre les coeffs
    cout << monPoly << endl; //on affiche le polynôme

    CFenetre maFenetre(200, 200); //création d'une fenêtre de taille 200x200

    //placement de la fenêtre dans la console (optionnel)
    maFenetre.setRectOffset(300, 10);

    maFenetre.afficherRectV2(255, 255, 255); //affichage des graduations
    maFenetre.afficherPolynome(monPoly, 0.001, RGB(255,0,0));

    maFenetre.afficherIntervalles();

    ///---affichage d'une fonction---
    //on précise l'amplitude et l'origine
    CFenetre maFenetre2(400, 400, 10, 10, 3, 8);
```

```

maFenetre2.setRectOffset(510, 10);

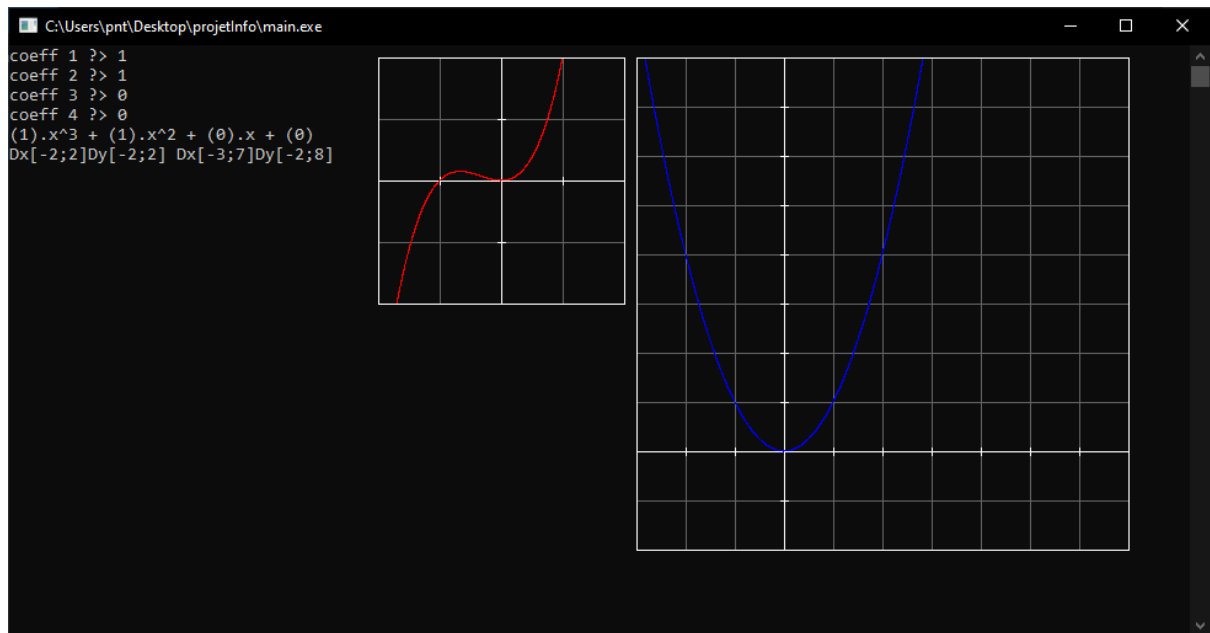
maFenetre2.afficherRectV2(255, 255, 255));
maFenetre2.afficherCourbe(f, 0.001, RGB(0,0,255));

maFenetre2.afficherIntervalles();

cin.get(); cin.get(); //pause la console
return 0;
}

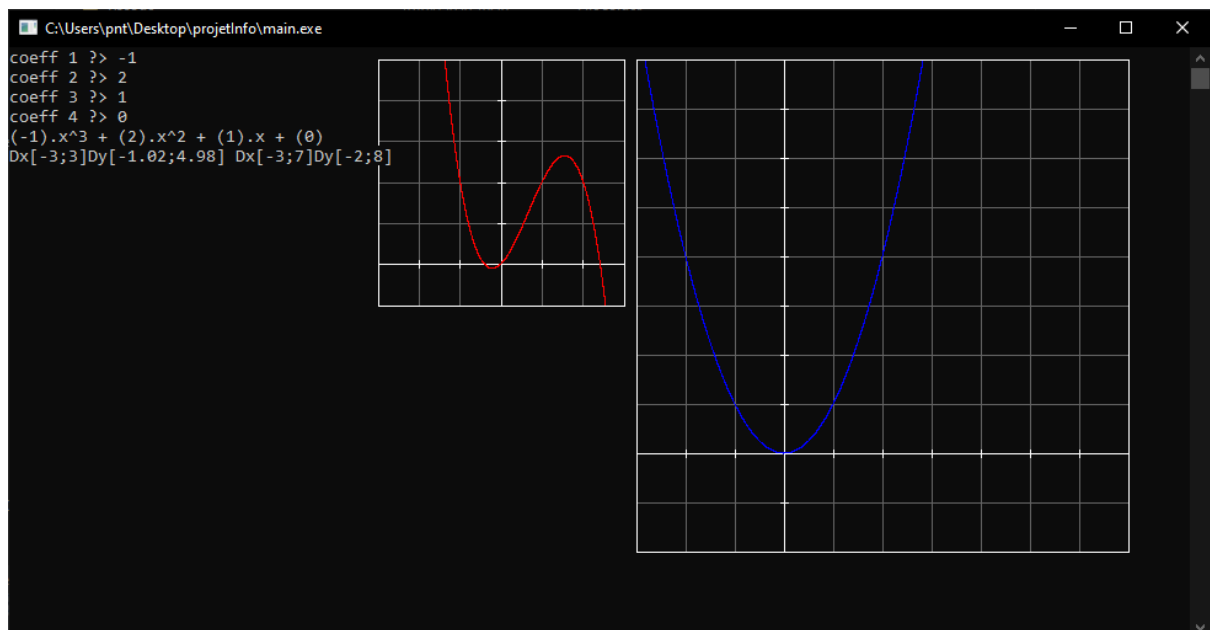
```

Ce qui donne avec une précision de 1000 point/pixel :

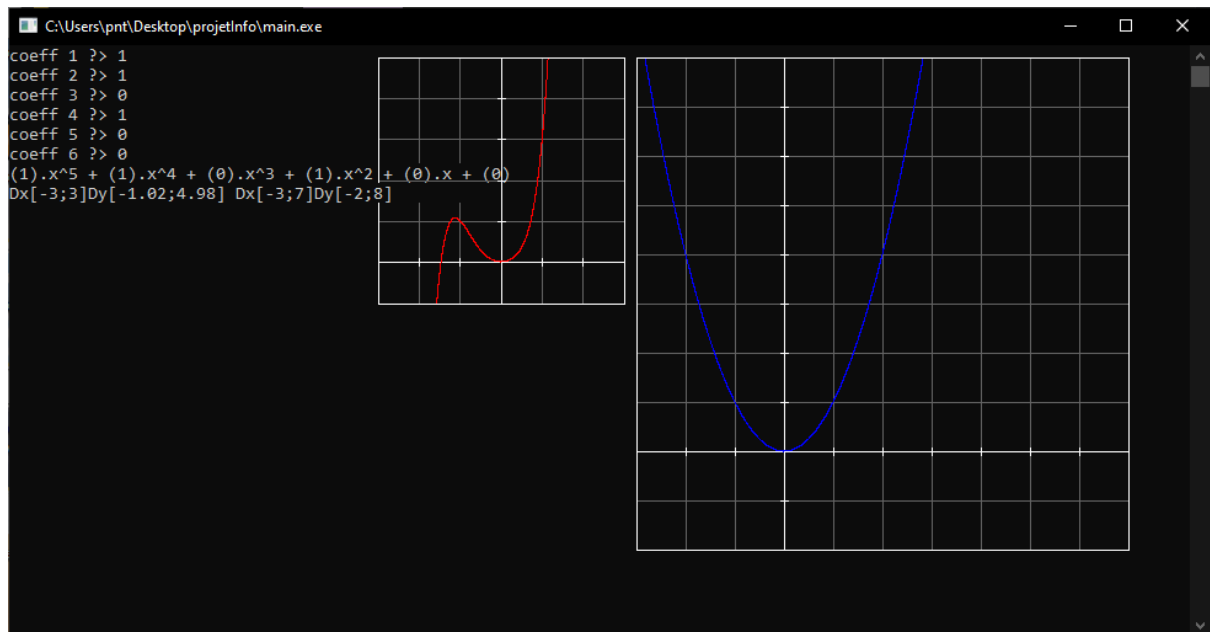


On peut changer de polynôme :

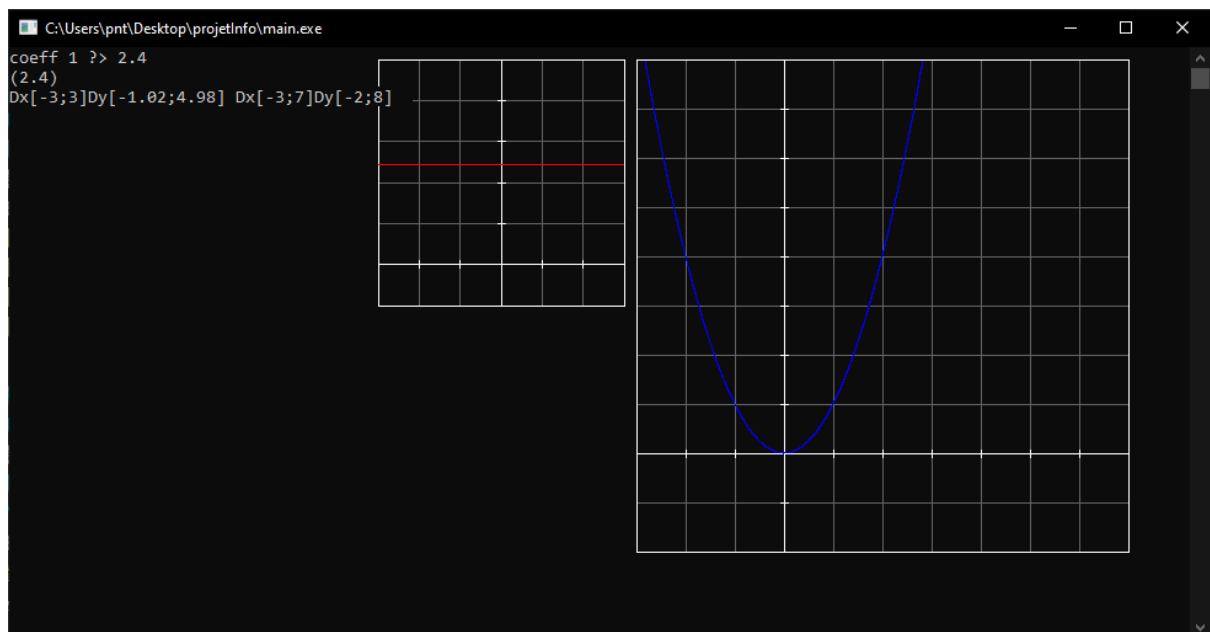
(CFenetre maFenetre(200, 200, 6, 6, 3, 5);)



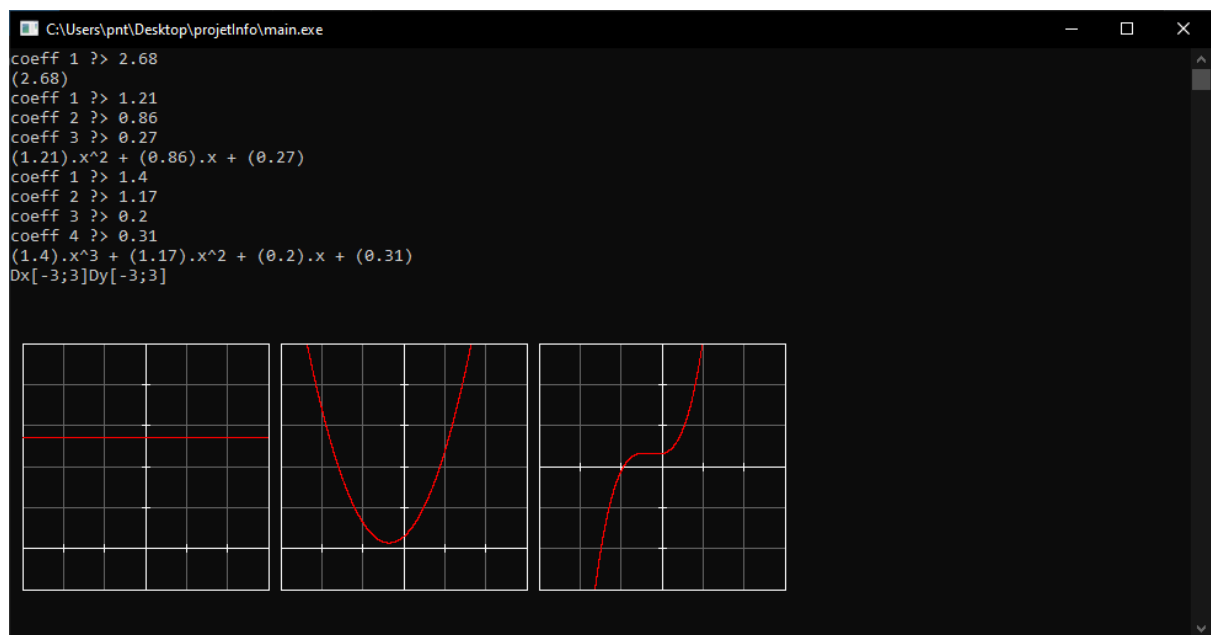
Polynome de degré 5 :



De degré 0 :



Ou encore :



Conclusion

Ce sujet permet d'en apprendre plus sur les coordonnées utilisées pour les écrans, mais aussi que l'utilisation des classes est absolument indispensable pour rendre le code modulable et ainsi résoudre des problèmes complexes.

On pourrait améliorer la classe CFenetre en laissant la possibilité de mettre l'origine dans la base (rect_x, rect_y).

Il existe aussi de nombreuses bibliothèques qui permettent d'effectuer ces tracés de données/courbes dits de "plotting" hors de la console tel que *qcustomplot*.