

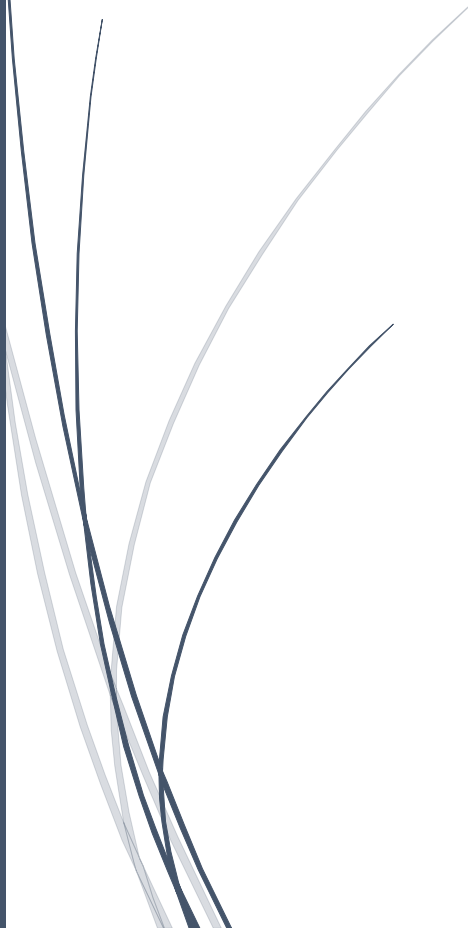
GAUTIER Arthur
L1 Promotion 2018
Groupe A

20/11/2013

Rapport de Projet

Programmation en C

Informatique et Calculs



Sommaire

<u>Introduction</u>	(2)
I. <u>Calculs simples</u>	(2)
A. <u>Nombres premiers</u>	(2)
B. <u>Nombres de Pythagore</u>	(6)
C. <u>Nombres de Smith</u>	(9)
II. <u>Courbes et Fonctions</u>	(11)
A. <u>Polynôme de degré 2</u>	(11)
B. <u>Polynôme de degré 3</u>	(14)
C. <u>Exponentielle</u>	(15)
D. <u>Cosinus et cosinus carré</u>	(16)
E. <u>Racine</u>	(19)
<u>Conclusion</u>	(20)

Introduction

Dans ce premier projet, réalisé dans le cadre de ma formation à l'algorithmique et à la programmation en langage C (et C++), il m'a été demandé, à l'aide de d'algorithmes et de calculs simples programmés dans l'un ou l'autre des deux langages suscités, de montrer qu'un ordinateur n'est fondamentalement pas capable de réaliser des opérations symboliques mais uniquement des opérations numériques, me permettant également de découvrir ce qui se cache derrière certaines fonctions mathématiques.

Ce rapport se présente sous formes de deux grandes parties. La première est en majeure partie consacrée à la recherche des nombres particuliers que sont les nombres premiers, les nombres de Pythagore et les nombres de Smith. A l'aide de méthodes simples, je commencerais par rechercher les plus petits nombres premiers. Je chercherais ensuite lesquels sont des nombres de Pythagore. Enfin, je chercherais les nombres dit de Smith ou nombres rigolos.

Dans la seconde partie, je m'attarderais sur la programmation de fonctions mathématiques telles que les polynômes, la fonction exponentielle ou encore la fonction cosinus. J'accompagnerais chaque fonction de sa courbe, obtenue grâce au tableur Excel.

I. Calculs simples

Dans cette partie, nous allons rechercher certains nombres particuliers, dotés de propriétés parfois surprenantes. Cette partie sera séparée en trois sous-parties, chacune dédiée à un type de nombres précis. La première sous-partie sera consacrée aux nombres premiers, la seconde aux nombres de Pythagore et la troisième aux nombres de Smith ou nombres rigolos.

A. Nombres premiers

Ici, je me consacrerai à la recherche des dix plus petits nombres premiers selon deux méthodes.

La première est la méthode dite « classique », c'est-à-dire diviser le nombre dont on cherche à déterminer la primauté par toutes les valeurs entières comprises entre deux et la moitié du nombre que l'on étudie. Si jamais on parvient à trouver un diviseur, cela prouve que ce nombre n'est pas premier. Toutefois, si on ne trouve pas de diviseurs, cela montre que le nombre est premier.

Ce programme de la méthode classique se divise en deux parties. La première est la recherche des dix plus petits nombres premiers, la seconde est l'affichage des dix nombres trouvés.

```

do /// Calcul les 10 plus petits nombres premiers
{
    premier = 1;
    for (divise = 2 ; divise <= nombre/2 ; divise = divise + 1)
    {
        reste = nombre % (divise);
        if (reste == 0)
        {
            premier = 0;
        }
    }
    if (premier == 1) /// Sauvegarde le nombre si il est premier
    {
        recens [compteur] = nombre;
        compteur = compteur + 1;
    }
    nombre = nombre + 1;
}
while (compteur < 10);

```

Dans la première partie du programme, j'utilise en premier lieu une boucle « faire ... tant que » associée à un compteur et à une variable qui me permettra de déterminer la primauté ou non d'un nombre. A chaque tour de boucle, cette variable est initialisée à un, je suppose donc à la base que le nombre est premier. La boucle « faire ... tant que » permettra de rechercher uniquement dix nombres. Lorsque le compteur atteindra 10, le programme sortira de la boucle. Le choix d'une telle boucle est conditionné par le fait que je connais le point de sortie sans connaître le nombre d'itérations nécessaire pour l'atteindre.

Pour déterminer si le nombre est premier, l'emploi d'une boucle « pour » me paraissait nécessaire. En effet, je connais le point de départ, celui d'arrivée ainsi que l'écart séparant deux valeurs. Dans cette boucle, je demande au programme de stocker le reste de la division euclidienne du nombre que l'on étudie par tous les nombres compris entre deux et la moitié du nombre étudié dans une variable. J'associe à cette boucle « pour » une boucle « si » qui permet de vérifier si le nombre admet un diviseur autre que un et lui-même. En effet, si la variable associée au reste de la division euclidienne contient zéro comme valeur, le programme rentre dans la boucle « si » et donne à la variable du critère de primauté la valeur zéro, montrant que le nombre n'est pas premier.

Si, à la fin de la boucle « pour », la variable du critère de primauté contient toujours la valeur un, le programme rentre dans une seconde boucle « si » qui stockera le nombre que l'on étudie dans un tableau. Après cela, je montre au programme que l'on a trouvé un nombre sur les dix recherchés en augmentant de un la valeur du compteur.

```

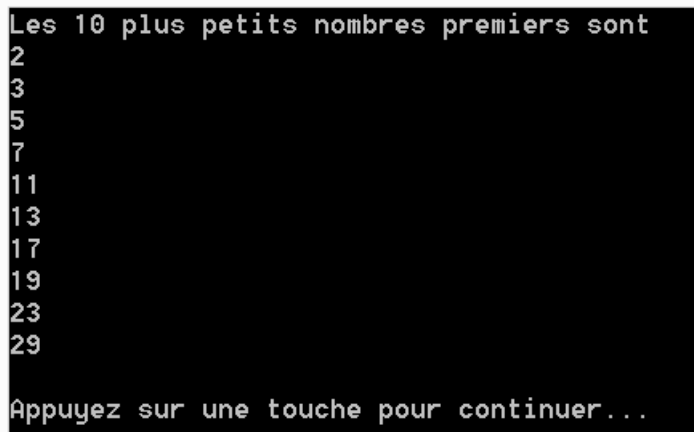
printf("Les 10 plus petits nombres premiers sont");
printf("\n");
for (num = 0 ; num < 10 ; num = num +1) /// Affiche les résultats trouvés
{
    printf("%d", recens [num]);
    printf("\n");
}
printf("\n");
system("pause");
return 0;

```

Cette seconde partie du programme sert à l'affichage des valeurs trouvées. Pour cela, j'utilise une boucle « pour » allant de zéro à la taille maximale du tableau (num) moins un. Cette boucle affiche successivement les dix nombres trouvés qui correspondent aux dix plus petits nombres premiers.

Cette première méthode a comme intérêt d'être fiable et de permettre le calcul de très grand nombres premiers puisque n'utilisant pas de calcul de factorielle. De plus, sa programmation est basique, ne demandant que peu de connaissances en langage C ou C++.

Voilà les résultats que nous donne ce programme :



```

Les 10 plus petits nombres premiers sont
2
3
5
7
11
13
17
19
23
29
Appuyez sur une touche pour continuer...

```

Passons maintenant à la seconde méthode, c'est-à-dire, avec l'aide du théorème de Wilson. Ce théorème nous dit qu'un nombre est premier si et seulement si la factorielle du nombre étudié moins un est congrue à moins un modulo le nombre que l'on étudie. De cela, il a été déduit que la fonction $f(n) = 2 + \left(2 \left((n-1)!\right) \bmod (n)\right)$ valait n si n est un nombre premier et valait 2 sinon.

Ce programme est encore une fois séparé en deux parties, l'une concernant le calcul en lui-même, la seconde servant à l'affichage. Cette seconde partie étant similaire à une valeur près à celle du programme précédent, je ne la retraiterais pas ici.

```

do /// Recherche selon la méthode de Wilson si le nombre est premier ou pas
{
    facto = n - 1;
    for (cpt = n - 2 ; cpt > 0 ; cpt = cpt - 1) /// Calcul de la factorielle
    {
        facto = facto * cpt;
    }
    quotient = floor((2*facto)/n); /// Application de la formule
    reste = 2.0*facto - (quotient * n) ;
    premier = 2 + reste;

    if (premier == n) /// Stockage du nombre si il est premier
    {
        tabprem [compteur] = n;
        compteur = compteur + 1;
    }
    n = n + 1;
}
while (compteur < 7);

```

Ici, je choisis encore une boucle « faire ... tant que » comme corps de l'algorithme. En effet, je sais à quel moment la boucle doit se terminer, et je sais également que la boucle doit faire au moins un passage. Toutefois, je ne sais pas quelle sera le nombre précis de passage. La boucle « faire ... tant que » est donc la plus justifiée. Ensuite, pour calculer la factorielle, j'utilise une boucle « pour ». Je dois calculer la factorielle du nombre étudié (je vais l'appeler n pour la formule) moins un c'est-à-dire $(n-1) * (n-2) * \dots * (2) * (1)$. Je définis donc la variable qui stockera la factorielle à nombre étudié moins un. Ensuite, j'initialise le compteur à nombre étudié moins deux puisque je recherche la factorielle de nombre étudié moins un.

Ensuite, le programme passe à l'application de la formule. La formule est appliquée en trois parties. Tout d'abord, comme j'ai défini la variable « facto » comme étant de type double, je ne peux utiliser l'opérateur « % », qui me donnerait directement ce que je recherche. J'effectue donc cette recherche en deux parties.

Tout d'abord, je demande au programme de calculer la partie entière de deux fois la factorielle du nombre étudié moins un divisé par le nombre étudié. La valeur que le programme obtient est en fait le quotient de la division euclidienne des deux termes. Toutefois, ce dont j'ai besoin dans le calcul de ma fonction est en fait le reste de cette division euclidienne. Pour l'obtenir, je demande au programme de retrancher à deux fois la factorielle du nombre moins un, le quotient obtenu multiplié par le nombre étudié. Grâce à cela, nous obtenons le reste. Pour finir le calcul de la fonction, il nous suffit d'ajouter deux à ce reste. A ce moment, le programme compare la valeur de la somme à celle du nombre étudié à l'aide d'une boucle « si ». Si elles sont égales, cela signifie que le nombre est premier, il est donc stocké dans un tableau, on signale au programme que l'un des nombres a été trouvé et le programme continue.

Cette méthode présente toutefois un désavantage majeur : elle ne me permet que d'afficher les sept plus petits nombres premiers. Cela est dû à une raison très simple. Lors du calcul de la factorielle, le programme arrive très rapidement à des valeurs dépassant les capacités de calculs de la machine, causant un overflow (ou débordement en français).

Voilà ce que nous affiche le programme :

```
Les 7 plus petits nombres premiers sont
2
3
5
7
11
13
17
Appuyez sur une touche pour continuer...
```

Ainsi, pour comparer les deux méthodes de recherche des nombres premiers, à mes yeux, la première méthode est certainement la plus efficace. Elle est certes plus longue puisqu'il est nécessaire de faire de nombreux calculs successifs, mais, elle a l'avantage de ne pas causer d'overflow et de pouvoir afficher une plus grande quantité de nombres premiers. De plus, sa programmation demande moins de technicité. En effet, il n'est employé que des fonctions et des types simples. La seconde méthode quant à elle demande une programmation légèrement plus complexe car elle nécessite l'utilisation de la fonction « floor » (partie entière) mais également de types plus compliqués et plus gourmands en mémoire comme les doubles. De plus, son utilisation sur des nombres premiers plus grands aura de grandes chances de provoquer un overflow. Elle est donc principalement intéressante quand on s'intéresse à la primauté de petits nombres. Or, quand on s'intéresse à de petits nombres, les deux méthodes vont à une vitesse quasi-similaire. C'est pourquoi je trouve la première méthode plus pratique à utiliser et plus simples à programmer.

B. Nombres de Pythagore

Cette sous-partie est dédiée à la recherche des nombres premiers de Pythagore. Un nombre premier dit « de Pythagore » est un nombre premier qui peut s'écrire sous la forme $(4n + 1)$ où n est un entier naturel (un entier strictement positif, 1 n'étant pas un nombre premier). D'après le théorème des deux carrés, ces nombres premiers sont également les seuls nombres premiers impairs que l'on peut écrire sous la forme de la somme de deux carrés.

Ce programme est divisé en trois parties. La première est le premier programme que j'ai présenté, la recherche des dix plus petits nombres premiers selon la méthode dite classique. La seconde consiste à déterminer si certains des dix nombres premiers trouvés sont des nombres premiers de Pythagore. Cette seconde partie se divise elle-même en deux sous parties : l'une consacrée à la recherche des nombres, l'autre à l'affichage. La sous partie de l'affichage étant très similaire aux affichages précédents, je ne la détaillerais pas dans la suite. La troisième partie est quant à elle consacrée à la vérification du théorème des deux carrés sur le nombre que l'on aura trouvé.

```
for (num = 0 ; num < 10 ; num = num + 1) /// Parmi les nombres premiers trouvés
{
    pythagore = 0;
    for (n = 1 ; n < 8 ; n = n + 1) /// On vérifie si ils sont divisibles par 4n + 1
    {
        inter = 4*n + 1;
        condition = recens [num] % inter;
        if (condition == 0) /// Si ils le sont, on le montre au programme
        {
            pythagore = 1;
        }
    }
    if (pythagore == 1) /// Si le programme voit que le nombre est de Pythagore, il l
    {
        pytab [indice] = recens [num];
        indice = indice +1;
        utile = utile + 1;
    }
}
```

La boucle qui donne forme à la partie de recherche des nombres premiers de Pythagore est une boucle « pour ». Cette boucle me permet de prendre dans le tableau qui contient les 10 nombres trouvés dans la première partie du programme (se référer à la première partie de la partie A) chacun des nombres premiers et de vérifier pour chacun d'eux, à l'aide d'une seconde boucle « pour » si ils sont divisibles par un nombre de la forme $4n + 1$. Pour les valeurs de n définie dans la boucle « pour », je sais que n doit commencer à un puisque si n commence à zéro, le nombre sera forcément dit de Pythagore puisque un divise tous les nombres (et donc les nombres premiers).

Ensuite, je fais en sorte que le programme empêche n de dépasser la valeur huit puisque le plus haut nombre premier que j'ai déterminé est vingt-neuf. Or, vingt-neuf est strictement inférieur à trente-deux qui est la valeur de quatre fois huit. Si jamais le programme trouve qu'un des nombres premiers est divisible par un nombre de la forme $4n + 1$, c'est-à-dire que le reste de la division euclidienne du nombre premier par $4n + 1$ est nul, il rentre dans une boucle « si » à l'intérieur de laquelle il donne à une variable la valeur un qui permettra de stocker le nombre puisqu'il est considéré comme étant de Pythagore. A l'aide d'une seconde boucle « si », si le nombre est de Pythagore, le programme stocke ce nombre dans un tableau dont la taille utile augmente à chaque fois qu'un nombre est rentré dans le tableau. Cette taille utile me permettra de manipuler le tableau et notamment, d'afficher les valeurs que le programme a trouvées.

Voici ce qu'affiche le programme lors de son exécution :

```
Les nombres premiers de Pythagore parmi les 10 plus petits nombres premiers sont
5
13
17
29
Appuyez sur une touche pour continuer...
```

Maintenant que le programme a déterminé les nombres premiers de Pythagore parmi les dix plus petits nombres premiers, il doit vérifier si ces nombres vérifient bien le théorème des deux carrés.

```
for (indice = 0 ; indice < utile ; indice = indice +1) /// Parmi les nombres
{
    carre = 0;
    for (a = 1 ; a <= 5 ; a = a + 1) /// On cherche si ils peuvent s'écrire s
    {
        for (b = 1 ; b < 3 ; b = b + 1)
        {
            somme = a*a + b*b;
            if (somme == pytab [indice]) /// Si c'est le cas, on le montre au
            {
                carre = 1;
            }
        }
    }
    if (carre == 1) /// Si le programme voit que le nombre peut s'écrire sous
    {
        printf("%d", pytab [indice]);
        printf(" suit le theoreme des deux carres ");
        system("pause");
        printf("\n");
    }
    else /// Sinon, il affiche que le nombre ne suit pas le théorème
    {
        printf("%d", pytab [indice]);
        printf(" ne suit pas le theoreme des deux carres ");
        system("pause");
        printf("\n");
    }
}
```

Ici, c'est encore une boucle « pour » qui donnera forme au programme. Cette boucle travaille avec le tableau contenant les nombres premiers de Pythagore trouvés à la partie précédente. Cette boucle servira à ce que toutes les valeurs du tableau passent dans le programme et qu'il vérifie si oui ou non, elles vérifient le théorème. J'utilise ensuite deux boucles « pour » imbriquées de façon à calculer les sommes de deux carrés. Pour la première boucle, je demande de faire varier la valeur de a de un à cinq puisque le plus grand nombre de Pythagore que l'on a est vingt-neuf. Or, il est strictement plus que trente-six, il ne peut donc pas s'écrire sous la forme de la somme de trente-six et d'un autre nombre positif. La seconde boucle varie de un à deux puisque vingt-neuf doit pouvoir s'écrire sous la forme de vingt-cinq plus un second carré. Or, ce carré est quatre (2^2). Le programme va alors calculer successivement les sommes de carrés et si jamais il trouve une valeur correspondant au nombre de Pythagore sur lequel il travaille, il entrera dans une boucle « si » qui définit la valeur d'une variable à un, variable qui montre que le nombre vérifie le théorème des deux carrés. Par la suite, une association de boucle « si ... sinon » nous permet d'afficher si le nombre vérifie ou non le théorème des deux carrés.

Voici le retour du programme :

```
5 suit le theoreme des deux carres Appuyez sur une touche pour continuer...
13 suit le theoreme des deux carres Appuyez sur une touche pour continuer...
17 suit le theoreme des deux carres Appuyez sur une touche pour continuer...
29 suit le theoreme des deux carres Appuyez sur une touche pour continuer...
```

C. Nombres de Smith

Abordons maintenant la question des nombres de Smith inférieurs à mille. Un nombre de Smith est un nombre entier dont la somme des chiffres qui le compose est égale la somme des chiffres de ses facteurs premiers. Les nombres premiers ne sont toutefois pas des nombres de Smith. On dit également que deux nombres de Smith sont jumeaux si ils sont consécutifs et tous les deux des nombres de Smith. Je vais donc vous présenter le programme que j'ai réalisé pour déterminer le nombre de nombres de Smith plus petit que mille.

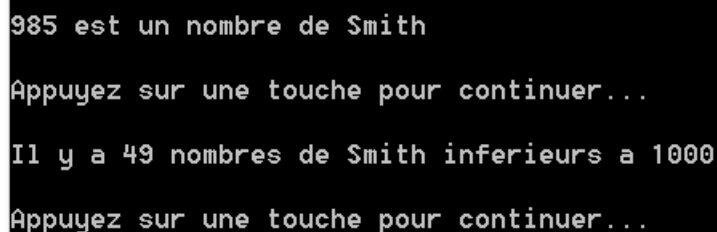
Ce programme débute par la recherche de tous les nombres premiers inférieurs à 1000. Pour cela, on emploie la méthode classique utilisée dans la partie A de cette partie.

Après cela, le programme se divise trois grandes parties, le tout dans une boucle « pour » qui se chargera de parcourir tous les nombres entiers strictement inférieur à mille à partir de trois. La première des trois grandes parties est la recherche des diviseurs premiers du nombre étudié. J'utilise pour cela une boucle « faire ... tant que » dont la sortie est définie par le fait que le nombre étudié devienne premier à force de divisions successives par ses diviseurs premiers ou lorsque que l'on a essayé de diviser le nombre par tous les nombres premiers précédemment déterminés. Dans cette grande boucle « faire ... tant que », je me sers de deux types de boucles : des boucles « si ... sinon si ... sinon » et une boucle « faire ... tant que ». A l'aide des boucles « si », le programme commence par chercher un diviseur premier. Si il en trouve un, le programme divise le nombre que l'on étudie par ce nombre. Il détermine ensuite la somme de ses chiffres. Il passe ensuite à une boucle « faire ... tant que » qui vérifie que le nombre n'est pas premier et que le programme n'a pas encore essayer de le diviser par les nombres premiers déterminés au début. Si l'une de ces conditions est réalisée, la boucle s'arrête. Le programme vérifie ensuite à l'aide d'une boucle « si » la primauté du nombre. S'il est premier, il détermine de la même façon que précédemment la somme de ses chiffres. S'il ne l'est pas, il continue à chercher un diviseur premier.

Une fois cette partie réalisée, le programme passe à la suite. Il cherche, à l'aide d'une boucle « faire ... tant que » si le nombre que l'on étudie fait partie des nombres premiers que le programme a déterminé précédemment ou pas. S'il en fait partie, on définit une variable à 1 qui nous permettra de montrer que le nombre est premier.

Enfin, si le programme trouve que le nombre n'est pas premier, et que la somme de ses chiffres est égale à la somme des chiffres de ses diviseurs premiers, il affiche que le nombre fait partie des nombres de Smith.

A ce moment, le programme sort de la boucle « pour ». Il va alors afficher le nombre de nombres Smith qu'il a trouvé à l'aide d'un compteur qui a été incrémenté à chaque affichage d'un nombre de Smith :



```
985 est un nombre de Smith
Appuyez sur une touche pour continuer...
Il y a 49 nombres de Smith inferieurs a 1000
Appuyez sur une touche pour continuer...
```

Enfin, le programme doit afficher comme jumeaux deux nombres de Smith consécutifs. Pour cela, j'utilise une boucle « pour » qui va parcourir le tableau des nombres de Smith associé à une boucle « si ». Et donc, s'il trouve que le nombre qu'il étudie plus un est égal à un des nombres du tableau, cela signifie que le nombre étudié et le nombre qui le suit sont des nombres de Smith jumeaux. Ici, le programme détermine un seul couple de nombres de Smith jumeaux :

```
728 et 729 sont des nombres de Smith jumeaux
```

II. Courbes et Fonctions

Dans cette partie, nous allons nous pencher sur la programmation de fonctions simples à l'aide des opérations de base (l'addition, la soustraction, la multiplication et la division) et des boucles. Le but de cette partie est de générer un ensemble de points que l'on importera après sous Excel afin de tracer la courbe correspondante. Les programmes se présenteront tous sous le même aspect. Une première partie concernant la saisie des valeurs, la seconde partie qui sera consacrée au calcul des points et une troisième partie qui affichera les valeurs obtenues et les exportera dans un fichier .csv.

A. Polynôme de degré 2

Ce premier programme sera consacré au calcul d'une fonction polynomiale de degré deux, j'entends par cela une fonction de la forme $f(x) = a.x^2 + b.x + c$. Ici, j'attribuerai de façon arbitraire à a la valeur trois, à b la valeur deux et à c la valeur un.

Dans ce programme, je commence par demander la saisie de trois valeurs : celle la borne inférieure, celle de la borne supérieure et celle du nombre de points que l'on désire. Ces saisies se présentent toutes sous la même forme, seul le texte et les conditions change :

```
do /// Saisie de la borne supérieure
{
    printf("Quelle est la borne superieure de l'intervalle de valeurs ? ");
    resul_saisM = scanf("%f", &maxi);
    scanf ("%*[^\\n]");
    getchar ();
    printf("\\n");
}
while (resul_saisM != 1 || maxi <= mini);
```

Cette partie du programme est pour la saisie de la borne supérieure de l'intervalle de valeur. J'utilise une boucle « faire ... tant que » pour réaliser une saisie sécurisée. Ensuite, le programme stocke le résultat de la saisie dans une variable. La boucle « faire ... tant que » rentre alors en jeu. Tant que l'utilisateur n'a pas rentré une valeur numérique ou que cette valeur est inférieure ou égale à celle de la borne minimale (demandée avant) le programme continuera à demander la valeur. Pour résoudre le problème de la boucle infinie en cas de saisie d'un caractère j'utilise deux instructions : le « scanf (« %[n] ») ; » et le « getchar () ; ». La première instruction indique que scanf va lire tout le tampon (l'endroit où est écrite la saisie et où scanf () va la lire) à part le caractère \n. Il ne reste donc plus dans le tampon que ce caractère \n. On utilise alors la seconde instruction qui permet d'éliminer le \n du tampon, le nettoyant complètement pour les saisies suivantes.

Passons au calcul de la fonction.

```
for (x = mini ; x <= maxi ; x = x + pas) /// Boucle de calcul
{
    tabx [n] = x;
    y = 3*x*x + 2*x + 1;
    taby [n] = y;
    n = n + 1;
}
```

Dans ce calcul, j'utilise une simple boucle « pour ». Le programme va calculer les images de x par la fonction $f(x) = 3x^2 + 2x + 1$. Cette boucle va faire varier x de la borne inférieure rentrée par l'utilisateur à la borne supérieure selon le pas. Pour toutes les fonctions, j'ai défini le pas comme valant (borne sup – borne inf) / (nombre de points – 1). En entrant dans la boucle, le programme rentre dans le tableau recensant les valeurs de x, la valeur de x. Il associe ensuite à y l'image de x par f. Enfin, y est stocké dans le tableau recensant les valeurs de y et le programme passe à la valeur suivante en augmentant l'indice des cases des tableaux de un pour ne pas écraser une valeur déjà rentrée.

La troisième partie du programme est l'affichage et l'écriture des valeurs dans un fichier .csv.

```
for (n = 0 ; n < npoint ; n = n + 1) /// Affichage de x et de y
{
    printf("x = %f", tabx [n]);
    printf(" f(x) = %f", taby [n]);
    printf("\n");
}

f=fopen("Polynome deg(2).csv", "w"); /// Ouverture du fichier cs

if(f==NULL)
{
    printf("Erreur lors de l'ouverture du fichier");
}

for (n = 0 ; n < npoint ; n = n + 1) /// Remplissage du fichier
{
    fprintf(f, "%lf", tabx [n]);
    fprintf(f, ",");
    fprintf(f, "%lf\n", taby [n]);
}
```

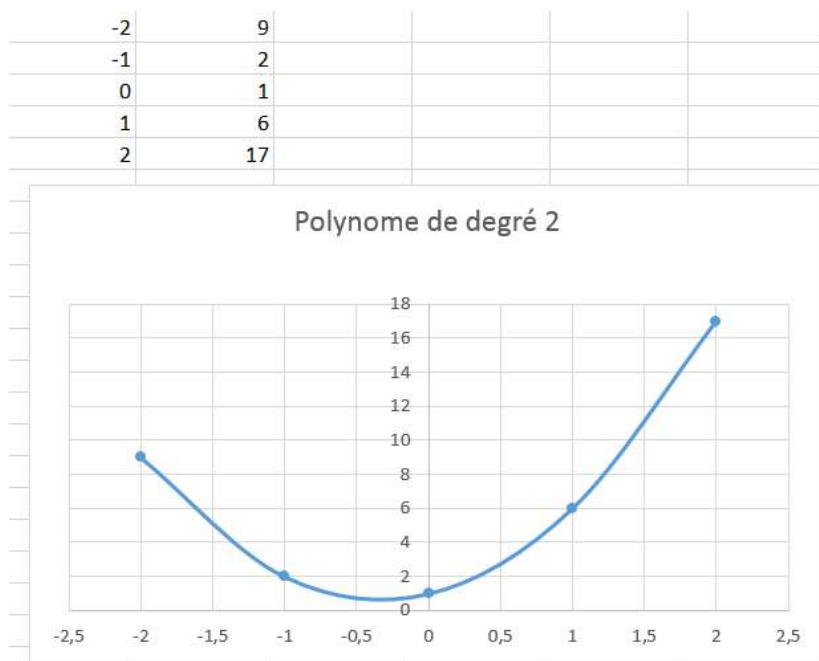
Pour l'affichage des valeurs, j'utilise une boucle « pour » qui va parcourir l'ensemble des deux tableaux de façon simultanée en affichant la valeur de x et son image par la fonction f . Cette boucle doit donc aller de 0 (la première case des tableaux) au nombre de cases moins un donc le nombre de points moins un.

Après cela, je demande au programme d'ouvrir un fichier .csv en mode écriture c'est-à-dire que les données vont être inscrites dans le fichier de façon automatisée. Le sigle csv signifie comma-separated values. Ces fichiers sont des types de fichiers excels particuliers permettant de stocker des valeurs automatiquement en les séparant par un point-virgule ou par une virgule. J'utilise encore une fois une boucle « pour » qui va parcourir les deux tableaux pour écrire les valeurs dans le fichier en les séparant dans par un point-virgule à chaque fois. Avant de pouvoir utiliser ces valeurs, il faudra toutefois les convertir car Excel ne reconnaît pas le point utilisé par le langage C pour définir un nombre rationnel.

Voici les résultats obtenus sur l'intervalle $[-2 ; 2]$ avec une demande de cinq points :

```
x = -2.000000 f(x) = 9.000000
x = -1.000000 f(x) = 2.000000
x = 0.000000 f(x) = 1.000000
x = 1.000000 f(x) = 6.000000
x = 2.000000 f(x) = 17.000000
```

Pour convertir le format des points sous Excel, il est nécessaire d'ouvrir le fichier .csv obtenu dans le répertoire du programme. Ensuite, sélectionner une colonne. Dans l'onglet Données, utiliser la fonction « Convertir ». Puis, dans la troisième étape de l'assistant, cliquer sur « Avancé » et remplacer le séparateur de décimale qui sera normalement une virgule par un point. Une fois cela réalisé, je demande à Excel de construire la courbe associée à ces points :



B. Polynôme de degré 3

Dans ce programme, je vais programmer une fonction polynomiale de degré 3 soit une fonction de la forme $a.x^3$. Dans ce programme, a prendra la valeur trois de façon arbitraire.

Ce programme ressemble fortement au programme précédent. En effet, c'est deux programmes ont le même ensemble de définition donc les conditions des saisies sécurisées ne sont pas modifiées. La seule différence avec le programme présenté en partie A est en fait la boucle de calcul :

```
for (x = mini ; x <= maxi ; x = x + pas) /// Boucle de calcul
{
    tabx [n] = x;
    y = 3*x*x*x;
    taby [n] = y;
    n = n + 1;
}
```

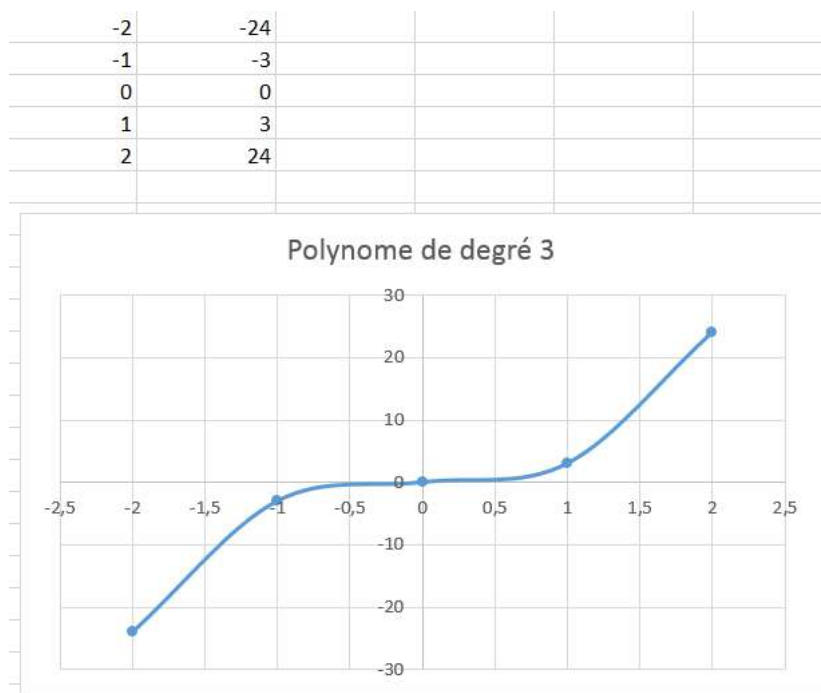
En effet, ici, y reçoit la valeur de $3.x^3$ au lieu de $3.x^2 + 2.x + 1$.

L'affichage et l'écriture sont également les mêmes, le nom du programme d'écriture changeant toutefois.

Voici les résultats obtenus en exécutant le programme sur $[-2 ; 2]$ en demandant cinq valeurs :

```
x = -2.000000 f(x) = -24.000000
x = -1.000000 f(x) = -3.000000
x = 0.000000 f(x) = 0.000000
x = 1.000000 f(x) = 3.000000
x = 2.000000 f(x) = 24.000000
```

Et voici la courbe et les valeurs après import sous Excel :



C. Exponentielle

Ici, je me consacre au calcul de la fonction exponentielle. Pour cette fonction, une formule m'a

été fournie par le sujet $e^x = \sum_{i=0}^{i=+\infty} \frac{x^i}{i!}$. Encore une fois, je ne m'attarderais pas sur les saisies ou sur l'affichage étant donné que ces deux parties sont similaires à celles des programmes traités précédemment.

Pour le calcul :

```
for (i = 1; i <= 20 ; i = i + 1) /// Pour avoir une précision suffisante
{
    j = i - 1;
    facto = 1;
    for (cpt = 1 ; cpt <= j ; cpt = cpt + 1) /// Calcul de (i - 1)!
    {
        facto = facto * cpt;
    }
    while (cpt2 <= j) /// Calcul de x^i - 1
    {
        x1 = x1 * x;
        cpt2 = cpt2 + 1;
    }
    somme = x1 / facto;
    total = ((x / i)*(somme));
    y += total; /// Somme de tous les "total"
}
if (x == 0) /// Calcul de exponentielle 0
{
    y = 1;
}
taby [n] = y;
n = n + 1;
```

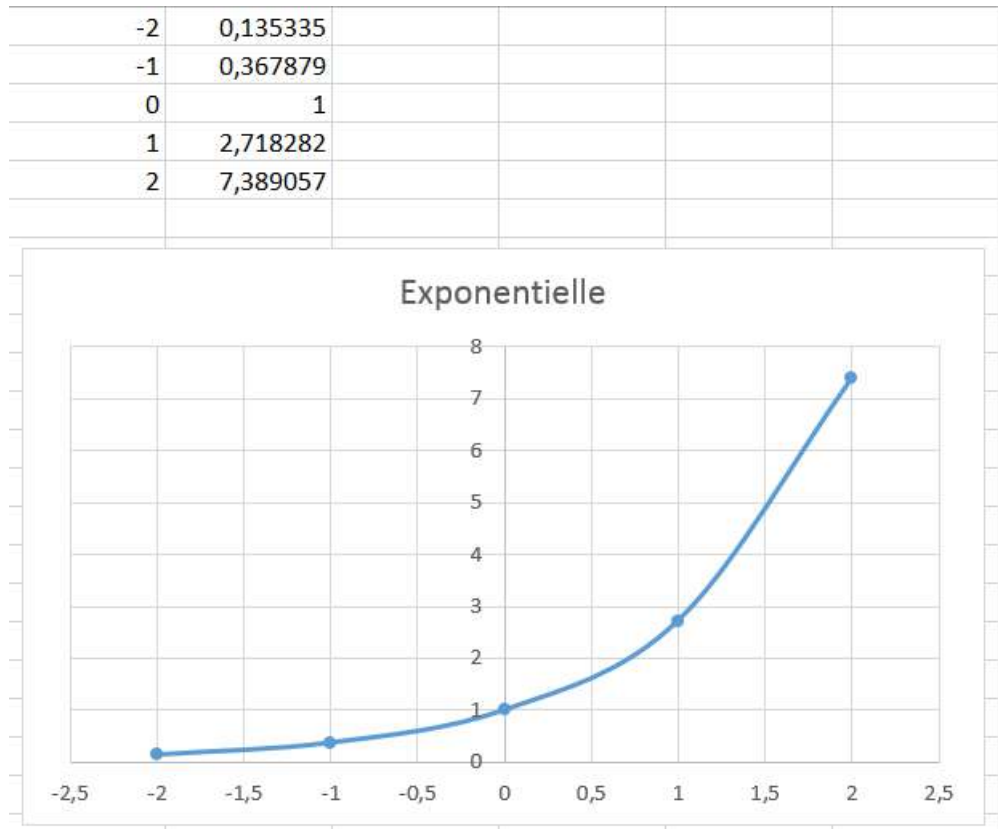
Le corps de ce calcul est encore la même boucle « pour » que pour les deux programmes précédents. On utilise ici pour ce calcul une boucle « pour » qui permet de transcrire l'idée de la somme mathématique $\sum$ qui va de $i = 1$ à $i = 20$. Je ne demande pas au programme de commencer à zéro pour simplifier le programme. En effet, si la boucle commençait à $i = 0$, la variable recevrait une erreur de calcul puisque je demande au programme de diviser par i . Donc, dans cette boucle « pour », j'utilise en parallèle une boucle « pour » afin de calculer la factorielle de $i - 1$ et une boucle « tant que » pour calculer x^{i-1} . Une fois ces calculs effectués, le programme stocke le résultat de la division de x^{i-1} par la factorielle de $i - 1$ dans une variable. Il utilise ensuite cette variable pour calculer un terme de la somme mathématique. Pour finir, il stocke la somme de tous les termes de la somme mathématique dans la variable y à l'aide de la fonction « += » qui permet d'ajouter les termes consécutifs.

Enfin, si jamais le programme passe par une valeur de x nulle, cela signifie qu'il calcule exponentielle 0, j'utilise donc une boucle « si » pour associer la valeur 1 à y dans ce cas.

Voici les résultats que donne le programme sur un intervalle $[-2 ; 2]$ en demandant cinq valeurs :

```
x = -2.000000f(x) = 0.135335
x = -1.000000f(x) = 0.367879
x = 0.000000f(x) = 1.000000
x = 1.000000f(x) = 2.718282
x = 2.000000f(x) = 7.389057
```

Et voici la courbe et les valeurs après import sous Excel :



D. Cosinus et cosinus carré

Je m'attaque cette fois-ci à l'étude d'une fonction trigonométrique. Je réunis ici ces deux programmes car ils sont très proches, seule la valeur de x va être modifiée de l'un à l'autre, le second attribuant à x la valeur de x^2 . Comme je calcule ici le cosinus d'un angle en radian, je n'ai pas besoin de modifier x . De plus, les saisies et les affichages restent très proches des programmes précédents, seule une précision sera ajoutée, dû à un fonctionnement limité du programme. Regardons maintenant la partie calcul de ces programmes :

```

for (x = mini ; x <= maxi ; x = x + pas) /// Boucle de calcul
{
    somme = 1;
    cosinus = 1;
    rang = 2;
    tabx [n] = x;
    for (rang = 2 ; rang < 201 ; rang = rang + 2) /// Calcul de la somme
    {
        cosinus = (cosinus * -(x*x)/(rang * (rang - 1)));
        somme = somme + cosinus;
    }
    y = somme;
    taby [n] = y;
    n = n + 1;
}

```

Encore une fois, la même boucle « pour » sert au corps au calcul de la fonction. Dans cette boucle « pour », on trouve une seconde boucle « pour » afin de calculer la valeur du cosinus. Cette boucle va de deux à deux cent selon deux de façon à n'avoir que les termes pairs des factorielles. Le choix de deux cent est arbitraire et permet d'afficher $\cos(x)$ pour x appartenant à $[-41 ; 41]$ ou x appartenant à $[-6 ; 6]$ pour $\cos(x^2)$. Pourquoi se limiter à ces valeurs ? C'est un choix purement arbitraire. En effet, j'aurais pu demander à ce que la boucle aille plus loin mais j'avais peur que cela ne prenne un temps de calcul trop important. Le fait que les $\cos(x)$ ne donnent pas la bonne valeur la bonne valeur si x n'appartient pas à $[-41 ; 41]$ est dû au fait qu'il manque des termes dans la somme de terme pour atteindre ce rang.

Les résultats affichés par le programme pour x appartenant à $[-3 ; 3]$ avec une demande de cinq valeurs :

```

x = -3.000000 f(x) = -0.989992
x = -2.000000 f(x) = -0.416147
x = -1.000000 f(x) = 0.540302
x = 0.000000 f(x) = 1.000000
x = 1.000000 f(x) = 0.540302
x = 2.000000 f(x) = -0.416147
x = 3.000000 f(x) = -0.989992

```

Pour $\cos(x)$

```

x = -3.000000 f(x) = -0.911130
x = -2.000000 f(x) = -0.653644
x = -1.000000 f(x) = 0.540302
x = 0.000000 f(x) = 1.000000
x = 1.000000 f(x) = 0.540302
x = 2.000000 f(x) = -0.653644
x = 3.000000 f(x) = -0.911130

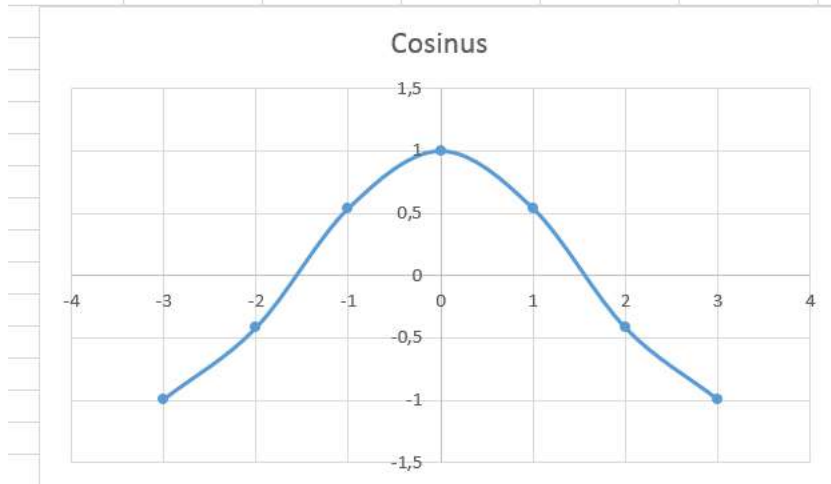
```

Pour $\cos(x^2)$

Après import des valeurs sous Excel, voilà la courbe que j'obtiens :

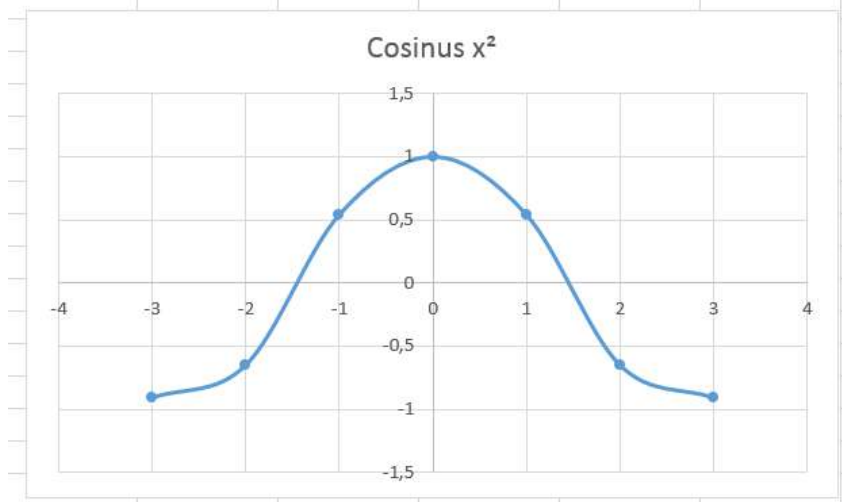
Pour $\cos(x)$:

-3	-0,989992				
-2	-0,416147				
-1	0,540302				
0	1				
1	0,540302				
2	-0,416147				
3	-0,989992				



Pour $\cos(x^2)$:

-3	-0,91113				
-2	-0,653644				
-1	0,540302				
0	1				
1	0,540302				
2	-0,653644				
3	-0,91113				



E. Racine

La dernière fonction que je vais traiter sera la fonction racine carrée. Les conditions de saisies de cette fonction diffèrent de celles des autres car il est nécessaire d'entrer des bornes nulles ou positives. En effet, il n'est pas possible de calculer la racine carrée d'un nombre négatif. L'affichage est quant à lui similaire à celui des autres fonctions. Pour le calcul, une formule m'a été fournie : la suite

$$U_0 = A$$

définie par : $U_{n+1} = \frac{U_n^2 + A}{2.U_n}$ converge vers la racine carrée de A en une quinzaine d'itérations.

Voici la boucle de calcul :

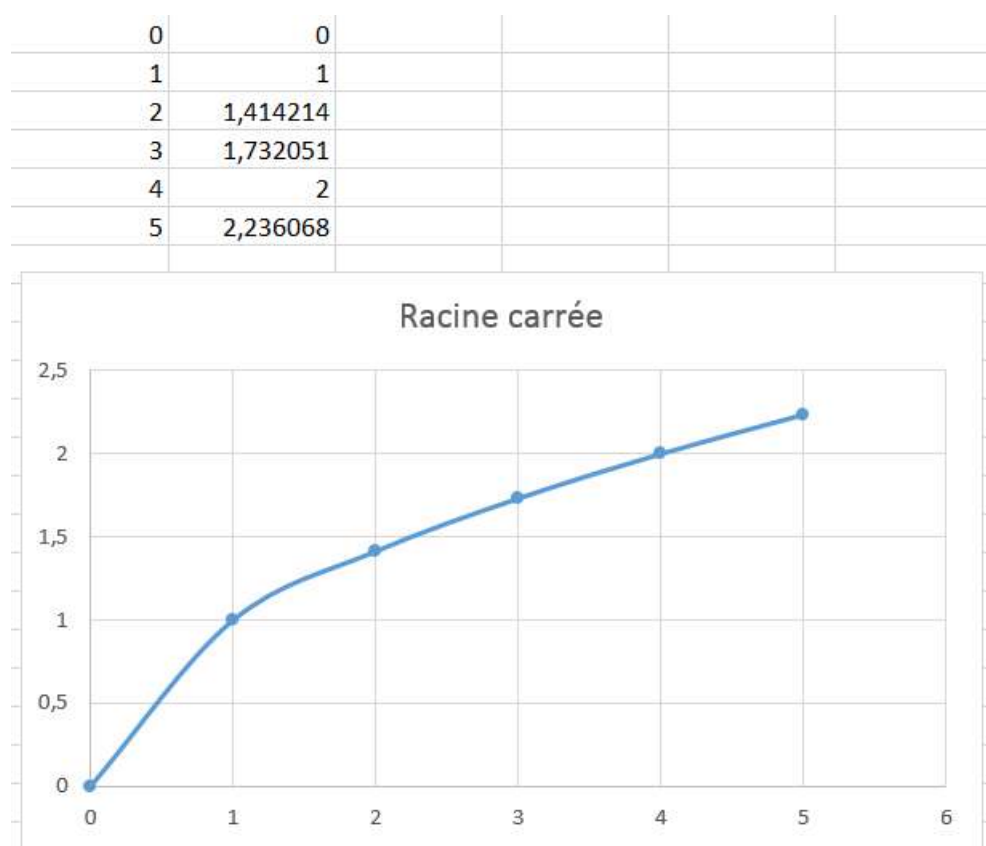
```
for (x = mini ; x <= maxi ; x = x + pas) /// Boucle de calcul
{
    tabx [n] = x;
    un1 = x;
    for (ite = 1 ; ite <= 15 ; ite = ite + 1) /// Calcul de la suite
    {
        un = ((un1 * un1) + x) / (2 * un1);
        un1 = un;
    }
    y = un;
    if (x == 0) /// Racine de 0
    {
        y = 0;
    }
    taby [n] = y;
    n = n + 1;
}
```

Encore une fois, j'utilise la même boucle « pour » afin d'avoir le corps du calcul. Dans cette boucle « pour », j'applique mot pour mot la définition de la suite que l'on m'a fournie. Ensuite, une boucle « si » me permet de traiter le cas particulier de la saisie du 0. Si le programme passe par la valeur $x = 0$, y recevra la valeur 0.

Voici les résultats que me donne le programme pour x appartenant à [0 ; 5] en demandant six points :

```
x = 0.000000 f(x) = 0.000000
x = 1.000000 f(x) = 1.000000
x = 2.000000 f(x) = 1.414214
x = 3.000000 f(x) = 1.732051
x = 4.000000 f(x) = 2.000000
x = 5.000000 f(x) = 2.236068
```

Après import de ces valeurs sous Excel, voilà la courbe tracée :



Conclusion

Dans ce projet, il nous a été demandé de programmer à l'aide d'opérations et de boucles simples des programmes de calcul. La plus grosse difficulté de ce projet s'est posée au moment de la transcription des différentes formules fournies par le sujet en programmes fonctionnels. De plus, ce programme m'a demandé de faire des recherches sur des notions que je ne maîtrisais pas encore telles que l'utilisation d'un fichier .csv ou l'utilisation de tableaux dynamiques.

Même si mes programmes ne sont pas les plus optimisés possibles, ils sont fonctionnels et je les trouve satisfaisant. Une possibilité d'optimisation aurait été de pouvoir réaliser les programmes par parties, comme, par exemple, ne pas écrire dans le programme de recherche des nombres de Pythagore le programme de recherche des nombres premiers mais pouvoir inclure ce programme uniquement par l'emploi d'une simple ligne incluant automatiquement le programme. Toutefois, je n'ai rien trouvé dans mon cours ou sur Internet me permettant de faire cela.

Pour conclure, ce projet m'a permis de mieux comprendre ce qui se cachait derrière l'utilisation d'une calculatrice ou lors de l'emploi de certaines fonctions informatiques telles que la fonction `sqrt ()` ou la fonction `pow ()`.