Laboratoire 2

La parabole de la chute

Ceci est le second de deux laboratoires traitant principalement de divers aspects de base de la programmation en C.

2.1 Objectifs

1. Apprendre à utiliser les différents types d'instructions conditionnelles en C: `if`, `else`, etc.;
2. Apprendre à définir et utiliser des fonctions en C;
3. Apprendre à utiliser les différents types d'instructions de répétition en C: `for`, `while`, etc.;
4. Apprendre à définir et utiliser des tableaux unidimensionnels en C
5. Apprendre à faire un graphique simple "y versus x" à l'aide de la librairie graphique PLPLOT;

2.2 Rapport de laboratoire

Il n'y a encore pas de rapport à remettre pour ce labo! profitez-en, c'est la dernière fois...

2.3 Les structures conditionnelles

Dans bien des situations de programmation que nous rencontrerons dans le cadre de ce cours, nous aurons besoin d'instructions permettant de contrôler l'exécution de certaines instructions ou blocs d'instructions sur la base de critères qui dépendent de la valeur de variables calculées durant l'exécution du code même, et donc qui ne peuvent être posées *a priori* au moment de l'écriture du code source. Ce sont les instructions dites **conditionnelles**. En C, le canevas général d'une instruction conditionnelle simple a la forme:

`if ( condition )` { **instructions** }

tandis qu'une instruction conditionnelle composée aurait comme canevas:

`if ( condition )` { **instructions 1** } `else` { **instructions 2** }

Les instructions conditionnelles peuvent être imbriquées, dans le sens qu'un bloc d'instructions sujet à exécution conditionnelle peut lui-même contenir d'autres instructions conditionnelles impliquant une ou plusieurs instructions `if` et/ou `else`.

Un petit exemple simple sera plus utile qu'une longue description; le code suivant renvoie en sortie à l'écran le plus grand de deux entiers fournis par l'usager au moment de l'exécution, via l'instruction `scanf`:

```
#include <stdio.h>
int main(void)
/* Ce code choisi le plus grand de deux entiers en entree */
{
/* Declarations ---------------------------------------- */
  int n, p, maxi ;
/* Executable ------------------------------------------ */
  printf("donnez deux nombres entiers : ") ;
  scanf ("%d%d", &n, &p) ;
 if (n < p )
   { maxi = p ; }
 else
   { maxi = n ; }
 printf ("le plus grand des deux est : %d\n", maxi) ;
}
```

Ici la condition `(n < p)` peut être vue comme une variable logique qui peut valoir `VRAI` ou `FAUX`; si son évaluation est `VRAI`, alors le premier bloc d'instructions est exécuté. Si elle évalue à `FAUX`, alors c'est le bloc suivant le `else` qui est exécuté. Le tableau suivant liste les divers opérateurs de comparaison disponibles dans le langage C.

Table 2.1: Opérateurs de comparaison

Opérateur	sens mathématique
`==`	égal à
`!=`	pas égal à
`>`	plus-grand-que
`<`	plus-petit-que
`>=`	plus-grand-ou-égal-à
`<=`	plus-petit-ou-égal-à

Ces opérateurs peuvent également servir à comparer des variables de type caractères (déclaration `char`); ceci est rarement utilisé en simulation numérique, sauf peut-être pour la saine gestion des nom de répertoires et fichiers crées automatiquement à l'exécution. Allez voir votre bouquin de C si vous êtes intéressé(e)s à savoir comment ca marche.

Les conditions contrôlant le comportement des instructions conditionnelles peuvent également être composées entre elles, à l'aide des opérateurs logiques `&&` (le AND logique), `||` (le OR logique), et `!` (le NON logique). Les deux "tables de vérité" suivantes illustrent le résultat de l'utilisation de ces opérateurs logiques sur deux conditions A et B qui chacune peuvent valoir `VRAI` ou `FAUX`[1]

Le code ci-dessus souffre d'une faille évidente; si vous fournissez deux entiers identiques, le code déclarera le second plus grand que le premier. Comprenez bien pourquoi, et ensuite votre premier défi est de modifier le code ci-dessus afin qu'il puisse, le cas échéant, produire une sortie du genre:

```
Les deux entiers fournis sont egaux
```

Il y a plusieurs façon d'accomplir ceci; essayez de la faire de manière "élégante", dans le sens que votre solution comporte un minimum d'instructions `if` et/ou `else`.

[1] Fait à noter pour ceux/celles ayant déjà maitrisé un langage de programmation autre que le C: À l'interne, le C encode le résultat d'une comparaison du genre `n < p` non pas comme une variable booléenne, mais comme un entier valant 0 pour `FAUX` et 1 pour `VRAI`.

Table 2.2: Opération logique A && B

	$B =$VRAI	$B =$FAUX
$A =$VRAI	VRAI	FAUX
$A =$FAUX	FAUX	FAUX

Table 2.3: Opération logique A || B

	$B =$VRAI	$B =$FAUX
$A =$VRAI	VRAI	VRAI
$A =$FAUX	VRAI	FAUX

2.4 Les fonctions en C

Le concept mathématique d'une fonction vous est déjà familier. Conceptuellement, une fonction est une "boite noire" acceptant un ou plusieurs arguments en entrée, en produisant un résultat via une ou plusieurs opération "internes". Par exemple, la fonction trigonométrique

$$\sin\theta$$

calcule la longueur de la projection sur l'axe des y d'un point situé sur le périmètre d'un cercle de rayon unitaire, à un angle θ de l'axe des x mesurée dans le sens antihoraire. Entre autres.

Le code C ci-dessous illustre la définition et l'utilisation d'une fonction en langage C. Ce code définit une fonction acceptant en argument deux entiers, en produisant pour résultat la valeur de celui de ces deux entiers qui est le plus grand. Opérationnellement, ce code fait la même chose que celui considéré précédemment comme exemple d'introduction aux instructions conditionnelles. Comparez avec attention ces deux codes!

```
#include <stdio.h>
int main(void)
{
/* Declarations ------------------------ */
   int trouveMax(int, int) ;
   int n, p, maxi ;

/* Executables ------------------------- */
   printf("donnez deux nombres entiers : ") ;
   scanf("%d%d", &n, &p) ;
   maxi = trouveMax(n,p) ;
   printf ("le plus grand des deux est : %d\n", maxi) ;
}
/* Fonction trouvant le plus grand de deux entiers */
int trouveMax (int a, int b)
{
   int ff ;
   if (a < b )
     { ff = b ; }  /* b est le plus grand */
   else
```

```
    { ff = a ; }  /* a est le plus grand */
  return  ff ;
}
```

Il y a plusieurs choses importantes à noter ici:

1. La fonction (ici appelée `trouveMax`) est une unité fonctionelle indépendante, qui dans le code source apparait à la suite du programme principal, i.e., à l'extérieur des délimiteurs {...} de ce dernier.

2. La section de déclaration du programme principal doit inclure une instruction déclarant à la fois le nom de la fonction, le type de la valeur retournée (un `int`, ou un `float`, etc), ainsi que le nombre et type de ses variables en argument; ainsi, ici l'instruction `int trouveMax(int, int)` indique que la fonction s'appelle `trouveMax`, qu'elle prend deux entiers en argument et calcule un résultat qui est aussi un entier.

3. La fonction même doit définir le même nom et le mêmes nombre et types de variables en argument et en résultat; ainsi, vu la déclaration de `trouveMax` dans le programme principal, l'instruction de définition de la fonction elle-même *doit* prendre une forme du genre `int trouveMax(int a, int b)`, où les *noms* des variables en argument (ici `a` et `b`) peuvent évidemment être autre chose.

4. Les noms de variables définis à l'intérieur de la fonction (ici `a`, `b`, et `ff`) demeurent locaux à la fonction, et n'ont pas besoin d'être identiques aux noms données aux variables passées en argument et reçues en résultat à l'appel de la fonction dans le code principal (soit ici `n`, `p`, et `maxi`). C'est l'ordre (n, p) des arguments dans l'appel à la fonction qui établit les associations $a \equiv n$, $b \equiv p$.

5. Une instruction de type `return` doit ici être incluse à la fin de la fonction, de manière à renvoyer l'exécution au programme principal, et déterminer laquelle des variables locales est renvoyé comme résultat à l'évaluation de la fonction (ici c'est `ff`).

6. Certaines fonctions, dites "fonctions-action", ne renvoient aucune valeur au programme principal `main`; c'est le cas, par exemple de la fonction `printf`, donc l'action consiste à envoyer à l'écran l'argument qui lui est fourni.

7. Une fonction peut être invoquée directement à l'intérieur d'une expression, ou de l'argument d'une autre fonction; ainsi, dans le programme principal (`main`) ci-dessus on aurait pu éviter la définition de la variable `maxi` en écrivant simplement:

   ```
   printf("le plus grand des deux est : %d\n", trouveMax(n,p) ) ;
   ```

Une fonction peut appeller d'autres fonctions, et le langage C permet même qu'une fonction s'appelle soi-même. Ceci s'appelle une **récursion**.

L'erreur la plus commune dans l'écriture d'un code C incluant des fonctions est d'introduire une ou plusieurs incompatibilités dans le nombre et type des variables en argument ou en sortie à l'évaluation de la fonction. Vous pouvez (lire: devriez!) vous amuser à un moment ou un autre à coder le programme ci-dessus, et changer un `int` pour un `float`, le nombre de paramètres en argument, etc., et voir lesquelles de ces erreurs seront détectées par le compilateur, et lesquelles se manifesteront seulement à l'exécution.

Le Tableau ci-dessous donne une liste de fonctions prédéfinies en C que vous trouverez probablement utiles. Le fait que ces fonction soient prédéfinies implique que vous n'avez pas à les déclarer explicitement dans vos code; cependant, certaines de ces fonctions sont regroupées en **librairies**, et dépendant du détail de l'installation du compilateur C il est possible que ces librairies doivent être explicitement chargées. Par exemple, autant sur mon Mac que sur le

Table 2.4: Quelques fonctions mathématiques prédéfinies du langage C

Fonction	sens mathématique	Notez bien
`sqrt(`x`)`	$\sqrt{x}$	$x \geq 0$.
`pow(`x, y`)`	x^y	
`log(`x`)`	$\ln(x)$	$x > 0$.
`log10(`x`)`	$\log_{10}(x)$	$x > 0$.
`exp(`x`)`	e^x	
`fabs(`x`)`	$\|x\|$	
`ceil(`x`)`	Partie entière de x	
`sin(`x`)`	$\sin(x)$	x en radian
`cos(`x`)`	$\cos(x)$	x en radian
`tan(`x`)`	$\tan(x)$	x en radian
`asin(`x`)`	$\arcsin(x)$	résultat en radian
`acos(`x`)`	$\arccos(x)$	résultat en radian
`atan(`x`)`	$\arctan(x)$	résultat en radian
`sinh(`x`)`	Sinus hyperbolique	
`cosh(`x`)`	Cosinus hyperbolique	
`tanh(`x`)`	Tangente hyperbolique	
`floor(`x`)`	Plus grand entier $< x$	résultat `int`, x est un réel
`ceil(`x`)`	Plus petit entier $> x$	résultat `int`, x est un réel

(vieux) système Unix avec lequel je teste tous les petits codes de ces labos, pour avoir accès aux fonctions de type mathématique je dois inclure en toute première ligne de mon code source:

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

et, au moment de la compilation, cette librairie doit être explicitement chargée:

```
gcc monCode.c -lm
```

Notons finalement que la très fameuse librairie < `stdio.h` > est chargée automatiquement par tous les compilateurs C auxquels vous n'aurez jamais à faire face, bien qu'elle doivent être explicitement incluse via une instruction `#include` en en-tête au code source.

2.5 Les structures de répétition

En programmation scientifique, la **répétition** d'une instruction ou bloc d'instructions est un des éléments de programmation le plus courant. Le langage C permet de définir deux types de boucles, en fonction de la manière dont le contrôle de la répétition est effectué.

2.5.1 Boucles conditionelles: les instructions `while` et `do...while`

Une boucle conditionnelle est structurée selon des canevas ressemblant fort aux instructions conditionnelles; les deux principaux canevas diffèrent au niveau de l'ordre dans lequel la condition est évaluée, par rapport aux instructions qu'elle contrôle. La structure de base prend la forme:

`while (` **condition** `)` { **instructions** }

Par exemple, le petit bout de code suivant écrirait en sortie les entiers de un à dix:

```
k=0 ;
while ( k <= 9 )
  {
    k=k+1 ;
    printf ("%d\n", k) ;
  }
```

On peut aussi tester la condition après le bloc d'instructions:

`do` { **instructions** } `while (` **condition** `)`

L'équivalent du petit bout de code ci-dessus, produisant la même sortie, serait:

```
k=0 ;
do {
    k=k+1 ;
    printf ("%d\n", k) ;
   }
while ( k <= 9 ) ;
```

Comparez attentivement ces deux bouts de code; notez et comprenez bien que:

1. À la sortie des boucles, la variable `k` vaudra 10 dans les deux cas;
2. Cependant, dans le second cas, le bloc d'instruction sera toujours exécuté au moins une fois, quelle que soit la valeur à laquelle `k` est initialisée;
3. Le point-virgule suivant la condition est essentiel dans le second cas, puisqu'il indique ici la fin de boucle.

Notez ici le danger qu'une boucle se mette à tourner sans jamais stopper, si la condition de contrôle ne se retrouve jamais satisfaite, suite à une erreur conceptuelle ou de codage. Nous verrons en temps et lieux comment parer à une telle éventualité.

2.5.2 Boucles inconditionelles: l'instruction `for`

Un moment de réflexion devrait vous convaincre que les deux boucles conditionnelles ci-dessus sont, fondamentalement, des boucles **inconditionnelles** puisque que l'on a décidé *a priori*, au moment de l'écriture du code source, que l'on voulait écrire en sortie les entiers de 1 à 10, plutôt que de 3 à 17 ou de 1 à 31415926536. Dans une telle situation on peut utiliser une structure de boucle dite inconditionnelle. Le canevas d'une telle boucle a la forme:

`for (` **décompte** `)` { **instructions** }

où **décompte** est une série d'instruction contrôlant le comportement de la boucle. Par exemple, l'équivalent du bout de code ci-dessus s'écrirait maintenant comme:

```
for ( k=1 ; k<=10; k++ )
  {
    printf ("%d", k) ;
  }
```

Notez la série de trois instructions de décompte, séparées comme il se doit par des point-virgules: (1) initialisation à un d'une variable compteur, ici `k`; (2) une condition de fin de boucle; (3) une instruction d'incrément, `k++` indiquant ici que la variable `k` est incrémentée de un à chaque "tour" de la boucle, raccourci en C pour `k=k+1`.

2.6 Définir et utiliser des tableaux en C

Dans bien des situations il est pratique de pouvoir définir des variables qui sont en fait des vecteurs, ou matrices, ou tenseurs, etc. Du point de vue du langage C, tous ces objets mathématique sont des **tableaux**. Un tableau est défini par un **type**, une **dimension** et une (ou plusieurs) **longueurs**. Par exemple, la position d'un point (x, y, z) en coordonnées cartésiennes peut être définie comme une seule variable réelle (`pos`, disons); dans la section déclaration, on n'a qu'à définir `pos` comme un tableau de type `float`, de dimension 1 et de longueur trois, comme ceci:

```
float pos[3]
```

dans lequel cas on pourrait avoir, dans la section exécutable, les associations

`pos[0]`$\equiv x$, `pos[1]`$\equiv y$, `pos[2]`$\equiv z$.

NOTEZ BIEN qu'en C la numérotation des éléments d'un tableau commence à zéro, pas à un! Donc, si vous avez déclaré votre tableau de longueur 3, son premier élément se voit assigner l'indice `[0]`, le second l'indice `[1]`, et le troisième l'indice `[2]`. Ceci sera une source d'erreur commune; tenter d'invoquer par inadvertance l'élément `pos[3]` causera un **débordement de tableau**, qui se traduira par une erreur à l'exécution. Ceci sera probablement votre plus commune source d'erreur en C, après l'oubli du "`;`"...

Un exemple spécifique vaut facilement une longue discussion; le petit canevas de code C ci-dessous montre comment créer et remplir un tableau de dimension 1, qui contiendra les valeurs d'une variable —disons le temps, pour prendre un exemple comme ça au hasard— où chaque valeur est plus grande que la précédente par le même incrément, comme dans le cas d'une maille équidistante:

```
#include <math.h>
#define N 11          /* longueur du tableau */
int main(void)
{
/* Declarations ------------------------------------------- */
  int   k ;
  float t[N] ;
/* Executable --------------------------------------------- */
/* calculer une maille equidistante entre t=0 et t=10 */
    for (k=0 ; k<N ; k++) {
      t[k]=10.*fabs(k)/(N-1) ;
    }
}
```

Je vous laisse vérifier, par ajout d'une instruction `printf` judicieusement formattée, qu'à la sortie de la boucle, le tableau `t` contiendra les valeurs:

$$[0., 1., 2., 3., 4., 5., 6., 7., 8., 9., 10.]$$

Premier point extrêmement important: en C, les dimensions des tableaux *doivent* être fixées au moment de la compilation, et donc spécifiées dans la section déclaration. En C, **on ne peut pas** utiliser une variable calculée à l'exécution pour spécifier la longueur d'un tableau; si cette longueur n'est pas connue *a priori*, on doit alors spécifier une taille maximale dans la section déclaration, et il serait sage d'inclure alors dans la partie exécutable un ou plusieurs tests s'assurant qu'il n'y aura pas de débordement de tableau. Remarquez ensuite ici une nouveauté, soit l'utilisation d'une instruction `#define`, placées en en-tête au programme même; elle spécifie à la compilation les valeurs numériques de la variable `N` utilisée pour spécifier la dimensions des divers tableaux déclarés. On aurait tout aussi bien pu omettre ces définitions; mais alors la ligne de déclaration aurait du avoir la forme:

```
float t[11] ;
```

L'avantage de l'utilisation des instructions `#define` est qu'un seul changement dans la valeur de `N` assignée par l'instruction affecte automatiquement tous les tableaux dont une ou plusieurs dimensions sont déterminées par `N` et, pour le code tel qu'écrit ci-dessus, ajuste automatiquement le nombre d'itérations effectuées par la boucle ainsi que la taille de l'incrément. Notez finalement que, puisque les dimensions du tableau sont fixées à la compilation du code, on utilise ici une structure de répétition inconditionnelle (`for` ...).

Notez également que, et c'est quelque peu contre-intuitif je l'admet, que dans une instruction `define` il n'y a pas de signe "=" entre une variable et sa valeur assignée. Amusez vous à remplacer l'instruction define ci-dessus par `#define N=11` et voyez ce qui se passe... L'expérience des années passées montre que ce sera une autre erreur commune qui sera répétée par plusieurs d'entre vous dans les semaines qui viennent.

L'utilisation de majuscules pour les noms de variables dont les valeurs sont spécifiée dans des instructions `#define` est facultative, mais c'est une bonne habitude de programmation qui permet, d'un coup d'oeil au code source, de détecter les "vraies" variables de celles initialisées à la compilation via des instructions `#define`.

Dans le cas d'un tableau de faible longueur, comme ci-dessus, il est également possible de définir explicitement le contenu du tableau en une seule ligne d'assignation (dans la partie "déclarations" du programme), comme suit:

```
float t[11] = { 0., 1., 2., 3., 4., 5., 6., 7., 8., 9., 10. }
```

Notez finalement qu'en C (comme en FORTRAN) les opérations arithmétiques sur les tableaux doivent se faire élément par élément; ainsi, si par exemple on veut doubler les valeurs des éléments du tableau `t` et placer ces valeurs dans un tableau `tx2` (préalablement déclaré `float` et de même dimensions que `t`), on doit écrire une boucle du genre:

```
for ( k=0 ; k<N ; k++ ) { tx2[k]=2.*t[k] ; }
```

plutôt que simplement `tx2=2.*t`, comme le permettent certains langages de programmation.

2.7 La trajectoire parabolique d'un corps en chute libre

Vous avez appris au secondaire que pour un mouvement rectiligne, la position x d'un mobile sujet à une accélération uniforme a est donnée par

$$x(t) = x_0 + v_0 t + \frac{1}{2} a t^2 \ , \tag{2.1}$$

où x_0 et v_0 sont les position et la vitesse du mobile à $t = 0$. Dans cette partie du labo nous allons écrire et exécuter un petit code C qui calcule $x(t)$ à une série de valeurs de t croissante, dans le contexte d'un mobile lancé vers le haut à une vitesse initiale v_0, et sujet uniquement à l'action de la gravité. L'idée ici est de calculer la trajectoire du mobile, i.e., x versus t, ainsi que le temps requis pour qu'il retombe à (mais pas plus bas) que son altitude de lancement.

Comme l'expression ci-dessus est parabolique en t, exiger que le mobile se retrouve à la position x_0 revient à exiger que

$$v_0 t + \frac{1}{2} a t^2 \equiv t \left(v_0 + \frac{1}{2} a t \right) = 0 \ ,$$

soit un polynôme à strictement parler quadratique, mais ici homogène, et donc acceptant comme solutions:

$$t = 0 \ , \qquad t = \frac{2 v_0}{a} \ .$$

La première de ces racines est évidemment la condition initiale, et la seconde représente l'intervalle de temps recherché.

Votre défi est maintenant le suivant: écrire un code C qui calcule $x(t)$ à partir d'une condition initiale $x_0 = 0$, $v_0 = 10\,\mathrm{m\ s^{-1}}$ à $t = 0$, pour $a = -g = -9.8\,\mathrm{m\ s^{-2}}$. Votre code doit évaluer l'éq. (2.1) ci-dessus et écrire le résultat en sortie à incrément de temps $\Delta t = 0.01$, stopper une fois $x(t)$ revenu à sa valeur initiale. Vous pourriez (par exemple) structurer ceci autour d'une boucle de type `while`. Vous devez calculer le temps total écoulé depuis le lancement, ainsi que le nombre de pas de temps ayant dus être effectués. De plus, vous devrez emmagasiner les valeurs de t et $x(t)$ dans deux tableaux de longueurs appropriées. Votre premier défi à ce niveau est que vous ne connaissez pas *priori* combien de valeurs de t seront requises pour revenir à $x = 0$, donc vous ne pouvez pas spécifier d'avance la taille requise des tableaux; vous pouvez cependant spécifier une taille maximale (`MNAX`, disons), et comptabiliser dans la boucle `while` le nombre de valeurs calculées. La structure de la boucle pourrait être du genre:

```
#define NMAX 500
...
double t[NMAX], x[NMAX] ;
...
t[0]=0. ;
x[0]=0. ;
n=0     ;
while ( x[n] >= 0 && n < NMAX ) {
  n=n+1 ;
  t[n] = ... ;
  x[n] = ... ;
}
if ( n == NMAX ) { printf( "NMAX atteint\n") ; }
...
```

Remarquez, et comprenez bien, les aspects suivants:

1. L'instruction `#define` régit maintenant la taille des tableaux unidimensionnels, via la valeur de la variable `NMAX`;

2. Une nouvelle variable `n` (type `int`) est initialisée à zéro et incrémentée de un à chaque passage dans la boucle `while`; à la sortie de la boucle, cette variable sera égale au nombre de points à tracer.

3. La condition de fin de boucle inclut une clause vérifiant que le nombre d'itérations de la boucle, mesuré par la variable `n`, demeure inférieur ou égal à la taille maximale `NMAX` des tableaux, telle que définie par l'instruction `define`, histoire d'éviter un débordement de tableau. Si cette condition est satisfaite, un avertissement est envoyé à l'écran via un appel à `printf` afin de prévenir l'usager (vous!) que le calcul est probablement incomplet, dans le sens que vous n'avez pas encore atteint atteint $x = 0$ même si la boucle est terminée.

Mesdames et Messieurs, à vos claviers... ! Et montrez le résultat à un des TP-istes avant de continuer.

2.8 Quand ça foire...

Avec des codes incluant des boucles (ou des commandes graphiques, que nous verrons sous peu), il est possible qu'une "fausse manoeuvre" vous laisse dans une situation où le code tourne "dans le vide". Dans une telle situation, pour stopper l'exécution, il suffit de peser simultanément sur les touches "control" et "c" sur le clavier. **NE PAS FERMER LA FENÊTRE COMMANDE EN CLIQUANT SUR LE "X" ROUGE AU HAUT DE LA FENÊTRE !!**

```
#include <stdio.h>
#include <plplot.h>
/* Ce programme est un cavenas de code C effectuant des sorties
   graphiques a l'aide de la librairie PLPLOT; plus precisement,
   ce code porte en graphique un segment de droite reliant deux points */
int main(void)
{
/* Declarations ------------------------------------------------------ */
  float x1, x2, y1, y2 ;
  float xmin, xmax, ymin, ymax ;
/* Executable -------------------------------------------------------- */
  plinit() ;                                /* Initialisation de PLPLOT */

  xmin = 0. ; xmax=2. ;                     /* intervalle en x du graphique */
  ymin = 1. ; ymax=5. ;                     /* intervalle en y du graphique */
  just = 0  ; style=1 ;                     /* format du graphique */
  plenv(xmin,xmax,ymin,ymax,just,style) ; /* Tracage du cadre */
  pllab("Axe x","Axe y","Mon titre") ;    /* Titre et annotation des axes */

  x1=1.  ; y1=1.5 ;                         /* Coordonnee du premier point */
  x2=1.5 ; y2=4.  ;                         /* Coordonnee du second point */
  pljoin(x1,y1,x2,y2) ;                     /* Tracage du segment */

  plend() ;                                  /* Fermeture de PLPLOT */
}
```

Figure 2.1: Canevas de code C faisant appel à la librairie graphique PLPLOT pour tracer un segment de droite sur un graphique simple.

COMPRIS ?!!. Ceci tuerait la fenêtre, mais le processus tournant à vide demeure actif sur le système ESIBAC, et donc bouffe des ressources (temps CPU, mémoire, etc) qui peuvent causer un ralentissement général du système, voire même un crash si trop de ces processus fantômes en viennent à co-exister sur le système.

2.9 Introduction au graphisme en C: PLPLOT

PLPLOT est une librairie de fonctions graphiques écrites en C, et utilisables directement dans tous les codes que vous développerez dans le cadre des labos. Pour un aperçu des capacités de cette librairie, voir la Page Web PLPLOT:

`http://plplot.sourceforge.net`

Vous y trouverez également un manuel de référence complet pour toutes les fonctions PLPLOT:

`http://plplot.sourceforge.net/docbook-manual/plplot-5.9.9.pdf`

L'idée est de bâtir graduellement vos habiletés graphiques au cours des labos, mais même à la fin du cours nous n'aurons touché qu'à une fraction des possibilités de PLPLOT. Pour aujourd'hui, nous nous en tiendrons au plus simple, soit le traçage d'une courbe sur un graphique classique de type "y versus x". Un exemple valant bien un long blabla, la Figure 2.1 montre un exemple d'un code C (minimal) qui trace un segment de droite reliant deux points (x_1, y_1) et (x_2, y_2) sur un graphique, tel qu'on le voit sur la Figure 2.2. Il y a plusieurs choses à noter ici:

1. Comme pour la librairie de fonctions mathématiques discutée précédemment, la librairie PLPLOT doit être incluse explicitement en en-tête au code source, via l'instruction `#include <plplot.h>`.

2. Toute série de commandes PLPLOT produisant des graphiques dans un code C doit commencer par un appel à la fonction d'initialisation `plinit()`, et se terminer par un appel à la fonction de clôture `plend()`; remarquez que ces deux fonctions sont des fonctions-action, qui ne renvoient aucun résultat numérique au code `main`, et de surcroit ne demandent aucun argument en entrée; leur exécution ne fait qu'effectuer diverses opérations au niveau du système d'exploitation de l'ordi.

3. La fonction `plenv` traces les axes et établit les échelles du graphique; ce dernier couvre ici les intervalles `xmin` $\leq x \leq$ `xmax`, et `ymin` $\leq y \leq$ `ymax`, les bornes étant spécifiées dans le code source même. Les deux derniers paramètres `just` et `style`, entiers valant ici 0 et 1, contrôlent le style des axes; nous en explorons d'autres options plus tard, pour l'instant utilisez ces valeurs.

4. La fonction `pllab` ajoute une annotation au bas de l'axe x (première chaine de caractères en argument), à gauche de l'axe y (seconde chaine de caractère en argument), et un titre global au dessus du graphique (troisième chaine de caractères).

5. La fonction `pljoin` trace un segment de droite entre deux points (x_1, y_1) et (x_2, y_2) spécifiés en argument.

Examinez bien les Figures 2.1 et 2.2 et assurez vous de bien comprendre les appels aux fonctions PLPLOTs, et les résultats graphiques qu'ils produisent. Ensuite, il s'agira de tracer la trajectoire parabolique que vous avez calculée précédemment. Pour ce faire, vous avez en fait deux options:

1. Utiliser la fonction `pljoin` à l'intérieur de votre boucle `while` pour connecter successivement chaque paire de points (t_{n-1}, x_{n-1}), (t_n, x_n).

2. Faire un seul appel, après la fin de la boucle, à la fonction `plline`, qui prend en argument les deux tableaux `t` et `x` ainsi que le nombre `n` de points à tracer:

   ```
   plline(n,t,x) ;
   ```

   Une subtilité agacante est que la définition de `plline` présuppose que les deux tableaux 1D en argument ont été définis `double`; vous devez faire de même dans votre code `main`, sinon il y aura incompatibilité entre l'appel à `plline` et sa définition à l'intérieur de la librairie PLPLOT.

À la compilation, vous devez explicitement charger la librairie PLPLOT. Vous devez d'abord spécifier où, sur les disques ESIBAC, le compilateur peut trouver la librairie. Nos collègues de la DGTIC nous ont facilité la tâche à ce niveau, vous n'avez qu'à taper, dans la fenêtre ESIBAC, la commande suivante *avant* la première compilation de votre session de travail:

```
module load plplot
```

Ensuite, lorsque vous compilez avec `gcc`, il faut spécifier le chargement de la librairie:

```
gcc code.c -lplplotd
```

Notez bien le "d" à la fin du nom de la librairie PLPLOT! Quand vous taperez la commande `./a.out`, une série d'instructions `printf` et `scanf` cachées à l'intérieur de la fonction `plinit` vous demanderont, à l'exécution, de spécifier le type de sortie graphique via un code numérique. Pour une sortie graphique à l'écran, vous devez spécifier la valeur 1. Si tout s'est bien passé, admirez la belle parabole... Une fois la phase d'émerveillement passée, montrez votre beau graphique à un des démonstrateurs, et voilà le second labo terminé!

Lectures supplémentaires:

Notes de cours: Chapitre 1

Delannoy — premier langage : Chapitres 4 et 5

Delannoy — programmer en C : Sections 3.1 à 3.6, chapitre 5

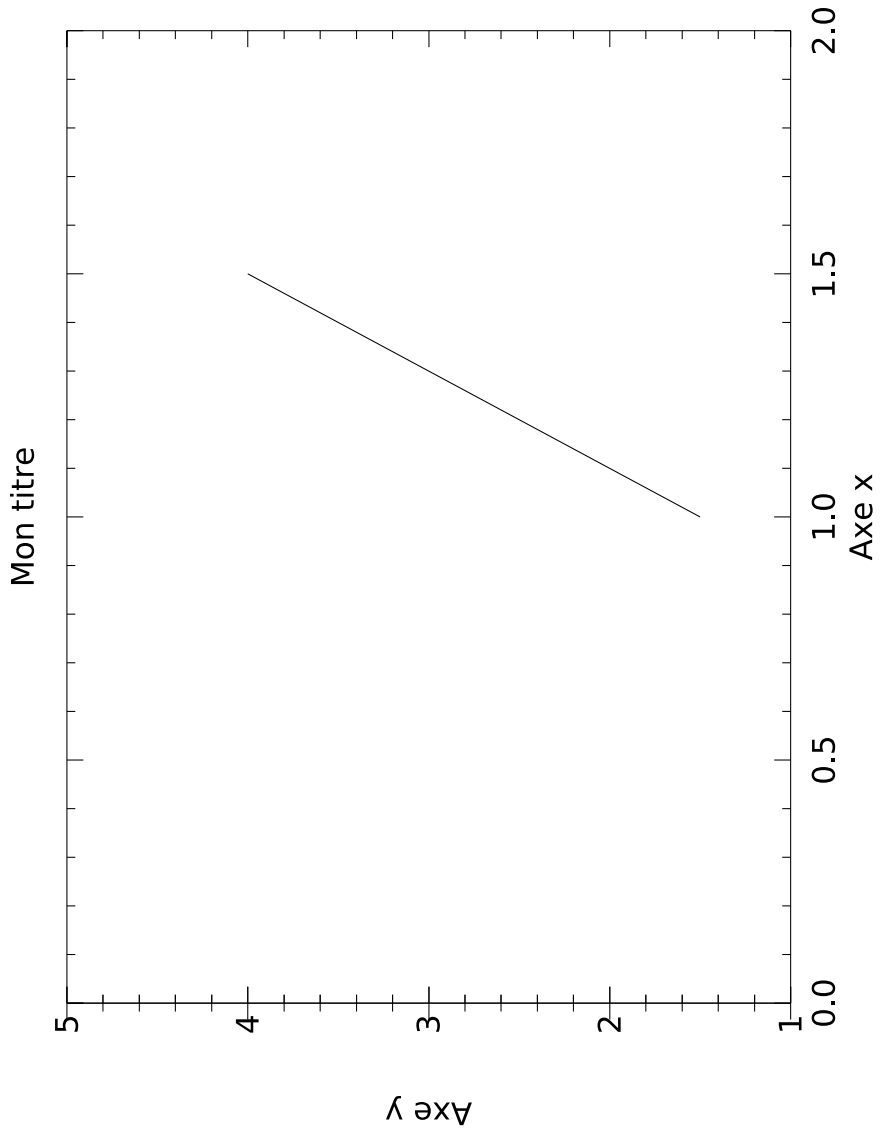

Figure 2.2: Graphique produit par le code source de la Figure 2.1. Ce graphique, tel qu'inclus dans ces notes, a été produit en spécifiant l'option graphique 4 au moment de l'initialisation de PLPLOT (appel à `plinit()`). Lorsque produit en forme fenêtre graphique à l'écran (option 1 au moment de l'initialisation), le graphique sera tracé en rouge sur fond noir... et ne sera pas pivoté de 90 degrés, comme ici.