Université Paul Sabatier

Faculté des Sciences et Ingénierie
Licence E.E.A. parcours Fondamental

Travaux Pratiques

DE TECHNIQUES SCIENTIFIQUES

rév 9.0

MISES EN GARDE IMPORTANTES

Vous allez utiliser une salle équipée d'ordinateurs, l'Université met à votre disposition du matériel qui est coûteux. Cela implique que vous respectiez le matériel et ses utilisateurs.

Il est **formellement interdit** de modifier les réglages écran, de naviguer sur le web durant les heures de TP, d'installer un logiciel et/ou de modifier de quelque façon que ce soit la configuration des machines. *Tout étudiant pris sur le fait sera exclu de la séance de TP.*

De plus, pour des raisons de courtoisie, il vous est demandé d'éteindre vos téléphones portables en entrant en salle de TP.

Vous ne pouvez vous contenter de travailler que durant les TP. Vous devez travailler entre les séances. Le langage C est une langue que vous devez apprendre. On n'apprend pas une langue en ne travaillant que 2h par semaine !

Cet enseignement est compliqué : il vous faut apprendre le langage, mais aussi un nouveau mode de raisonnement pour traiter les problèmes posés.

L'équipe met à votre disposition un site web : http://eeacastelan.ups.free.fr/. Vous trouverez sur ce site des fiche Langage. Si vous désirez qu'elles évoluent, signalez-le en envoyant un message à M. Castelan.

Si vous voulez tirer profit de ces TD/TP, il est impératif que vous les prépariez avant de venir en séance.

Vous êtes venus à l'Université pour apprendre, ne laissez pas passer votre chance. Si vous rencontrez des difficultés, n'hésitez pas à questionner votre enseignant.

Nous vous rappelons également que les **TPs sont obligatoires** et que toute absence doit être justifiée, auprès du responsable Mr Castelan, et rattrapée. Les étudiants qui auraient plus d'une absence injustifiée au cours du semestre ne seront pas convoqués à l'oral de projet en fin de semestre.

Les rattrapages peuvent se faire avec l'accord de l'enseignant en séance sous réserve de places disponibles, bien entendu, rattraper une séance de projet sans le reste de l'équipe n'a que peu de sens.

Introduction

Convention d'écriture

Dans tout ce qui suit, le texte représentant du code est dans la police de caractère suivante :

```
Mon_code_génial.
```

La partie du texte décrivant le travail que vous devez effectuer est dans la police suivante :

➡ Travail à effectuer.

Organisation

Les 3 premières manipulations servent à se familiariser avec l'environnement de travail. Les thèmes abordés seront les suivants :

- Découverte du compilateur et de l'environnement de travail.
- Lecture/Écriture de fichiers.
- Tracé de courbes avec Octave.
- Travailler avec des pointeurs

Les manipulations suivantes sont en fait des mini-projets sur deux niveaux ou vous êtes en semi-autonomie avec obligation de rédiger un rapport sur le thème de niveau 2 et recherche bibliographique sur le thème proposé et les méthodes à mettre en oeuvre pour aboutir.

Le projets de niveau 1 se font seul. Les projets de niveau 2 se font en équipe.

Modalité du contrôle des connaissances :

Vous êtes évalués de trois façons :

- Académique lors d'un examen écrit de langage C ou votre connaissance du Langage est évaluée par un examen QCM en distanciel via Moodle. Les objectifs sont la connaissance des instructions du langage C et des erreurs fréquemment rencontrées. La connaissance de l'accès aux fichiers de type texte, et l'utilisation de pointeurs et de variables dynamiques servent de base a ces questions mais seulement.
- Lors du projet de niveau 2, votre niveau est évalué. Vous êtes noté par niveau allant de A à C. Cette mesure sert à la constitution des groupes de projet de la troisième phase. Vous travaillez en autonomie.
- Un examen oral, lors duquel vous présentez individuellement un projet dit de niveau 2 (vous devrez fournir un dossier présentant le projet : ce que fait le programme, les méthodes utilisées, les références bibliographiques, les résultats obtenus etc.)

Conseils de programmation

Prenez la peine d'annoter de vos réflexions chacun des sujets abordés. Commentez, en particulier, le code que vous serez amenés à écrire. N'ayez pas peur de paraître simpliste, vous comprenez peut-être maintenant les tenants et les aboutissants de la méthode, mais dans 3 mois ?

S'il n'est pas utile d'avoir tous les listages des programmes, il est par contre indispensable d'avoir les algorithmes !

Pour la lisibilité des programmes et, par voie de conséquence, la facilité de leur mise au point, respectez les quelques conseils qui suivent :

- Donnez aux variables et fonctions des **noms significatifs**. Ces noms doivent être explicites mais courts.
- **Décomposez** votre programme en fonctions (qui doivent tenir idéalement sur moins de deux pages écrans). Séparez-les clairement sur votre listage avec des lignes de **commentaires utiles** (description d'une particularité de programmation, de la méthode, rappel de la signification d'une variable etc.…). Les lignes d'étoiles font joli mais n'expliquent rien !
- Utilisez le moins possible de variables globales et passez les données en paramètres de vos fonctions (mais ne passez que les données réellement nécessaires!).
- Évitez de surcharger votre listing !
- **Indentez votre code** et ne mettez qu'une seule instruction par ligne. La présentation du code, sa lisibilité, son efficacité seront pris en compte lors de la notation !
- Ne vous contentez pas d'une lecture succincte du cahier. Réfléchissez sur les thèmes proposés, imaginez les problèmes que vous allez rencontrer. Lisez le cahier et préparez chaque TP (algorithme, liste des fonctions à écrire, structure générale du programme, variables à utiliser…).
- **Posez des questions ! Soyez curieux**.

Rédigez des comptes rendus en tenant compte des quelques remarques suivantes, idéalement la structure d'un compte rendu est la suivante :

- Dites ce que vous allez faire. Résumez le sujet du TP avec **votre** vocabulaire, il ne sert à rien de recopier le texte du cahier. Cette phase est généralement **longue**, il faut s'y prendre à l'avance.
- Décomposez le problème à résoudre en actions simples. Décrivez en "bon français" les actions que vous allez accomplir. C'est à dire, pas un langage "pseudo algorithmique" ou "pseudo C". Ce qui se conçoit bien s'énonce bien…
- Donnez un algorithme des actions décrites au point 2. Commentez celui-ci pour que vous puissiez le comprendre quand vous le reprendrez plus tard.
- Pendant le TP, complétez votre compte rendu **en précisant ce qui n'a pas fonctionné** comme prévu et **pourquoi**. Ceci vous aidera à comprendre le type d'erreur que vous commettez et comment corriger votre raisonnement. Prévoyez bien une quatrième partie à votre compte rendu, ne surchargez pas les parties 2 et 3 autrement qu'en précisant en marge que le point en question est faux ou incomplet.

Rappels de langage C

Nous vous invitons à réviser les points suivants :

- Commandes d'entrée / sortie : printf - scanf
- Instructions de branchement (if / switch)
- Instructions de répétition (while, do…while, for)
- Les types de variables
- Les variables dimensionnées.
- Les pointeurs
- Les fonctions

Certains points sont présentés ici, pour les autres référez-vous à des ouvrages disponibles à la bibliothèque universitaire par exemple "Langage C, par Claude Delannoy, ed. Eyrolles cote 681.306C", ou "Le langage C, de P. Aitken et BL Jones, cote 6813.06C AIT", voire "S'initier à la programmation avec des exemples en C, C++ C#, Java et PHP de C. Delannoy, cote 681.3.06(076)DEL".

Manipulation de fichiers texte

Le lien entre votre programme et le fichier présent sur le disque dur est créé par la fonction **`fopen()`**. Cette fonction crée le lien et vous retourne une adresse, celle du tampon. L'adresse de ces tampons est définie à l'aide de pointeurs d'un type particuliers pré définis dans la bibliothèque **`stdio.h`**.

La syntaxe est la suivante :

```
FILE *mavariable;
mavariable = fopen(chemin d'accès, mode d'ouverture du fichier);
```

Nota Bene :

Le chemin d'accès est un chemin absolu ou relatif d'accès au fichier ***en utilisant la syntaxe UNIX exclusivement****. Ainsi, même dans un environnement de type windows et en supposant que le fichier toto.txt soit stocké dans le dossier travail (ce qui donne C:\TRAVAIL\TOTO.TXT) le chemin à passer à la fonction* **`fopen()`** *est : "c:/travail/toto.txt".*

Il existe plusieurs modes d'ouvertures de fichier résumés dans le tableau ci dessous :

mode	ouverture en :
"r"	lecture seule
"w"	écriture seule
"a"	ajout
"r+"	lecture/écriture
"w+"	écriture/lecture

Ainsi si on ne doit que lire le contenu du fichier *toto.txt* se trouvant dans le dossier travail on procédera comme suit :

```
FILE *mavariable;
mavariable = fopen("toto.txt", "r" );
```

L'ouverture en mode écriture "**w**" provoque la création d'un fichier portant le nom "toto.txt" si celui n'existe pas. **Si ce fichier existe, il est d'abord effacé**.

Une fois le fichier ouvert on peut accéder aux données contenues dans le fichier ou y écrire les données que l'on à y stocker.

Les fichiers avec lesquels nous travaillerons sont des fichiers de type texte. Les données sont stockées sous forme de caractères et sont séparées entre elles par un séparateur de champ.

2-a/Lecture :

La lecture proprement dite des données s'effectue avec la fonction **fscanf()**. La syntaxe est la suivante :

```
fscanf(variable_FILE,"format", liste d'adresse de variables);
```

La partie format et la liste d'adresse ont la même syntaxe que pour la fonction **scanf()**. La seule différence réside dans le fait qu'il ne faut pas oublier de séparer les champs par le séparateur dans la chaîne de format.

Le premier paramètre de la fonction **fscanf** est un pointeur de type ***FILE***. Ce pointeur doit avoir été initialisé auparavant à l'aide de la fonction **fopen()**.

La fonction **fscanf()** lit le fichier ligne par ligne et les données séparées par le séparateur sont attribuées aux différentes variables dont les adresses ont fournies. Le champ1 à la première variable etc.

Afin de savoir si toutes les lignes du fichier ont été lues, le C propose une fonction permettant de tester si la fin du fichier a été atteinte : **feof()**. La syntaxe est la suivante :

```
feof(variable_FILE);
```

Si la fin du fichier est atteinte, la fonction retourne **TRUE**, **FALSE** sinon.

Le code suivant réalise la lecture d'un fichier comportant deux colonnes de **float** séparées par un espace.

```
FILE *f1;
f1 = fopen("toto.txt","r");
if (f1==NULL)
{ printf("Erreur de fichier\n");
  exit(1); // arrête le programme, nécessite <stdlib.h>
}
while ( !feof(f1)) // tant qu'on  n'est pas à la fin !
{ fscanf(f1,"%f %f\n",&a,&b); // a et b sont des float
  printf("%f\t%f\n",a,b);
}
```

Notez l'utilisation de **feof()** et la chaîne de format de **fscanf()**. Enfin, l'utilisation de **exit()** (**stdlib.h**) permet d'éviter une lecture si le fichier n'existe pas ce qui provoquerait un "plantage" du programme.

2-b/Ecriture :

Lorsqu'on écrit une fichier la question du nombre de lignes ne se pose pas. Une boucle **for** est donc suffisante.

La fonction que vous utiliserez pour écrire les données est **fprintf()**. Sa syntaxe est la suivante :

```
fprintf(variable_FILE,"chaîne de format", liste de variables );
```

La seule différence entre **`fprintf()`** et **`printf()`** est la **`variable_FILE`**, qui est bien entendu obtenue via **`fopen()`**. Enfin il ne faut pas oublier le séparateur de champs dans la chaîne de format. Le code suivant crée le fichier donné en exemple plus haut.

```
float a[2][2] = {{3.1415, 2.717},{1.414, 1.73}};
int i;
FILE *f2;
f2=fopen("toto.txt","w");
for ( i=0 ; i<2 ; i++)
  fprintf(f2,"%f;%f\n",a[i][0],a[i][1]);
```

3/Fermeture des fichiers

Tant qu'un fichier n'est pas fermé, il est ouvert ! Si vous ne fermez pas un fichier vous avez de fortes chances pour qu'il soit incomplet.

La fonction permettant de fermer un fichier est **`fclose()`**, sa syntaxe est :

```
fclose(variable_FILE);
```

Les pointeurs

Un pointeur est une variable qui contient des adresses. Ce peut être l'adresse d'une variable, d'un port de communication ou n'importe quoi d'autre.

Lorsque l'on connaît l'adresse mémoire des données auxquelles on veut accéder, il est possible d'y accéder. Cette Lapalissade va pourtant être mise à profit tout au long de l'année.

Définition d'un pointeur.

```
type *nom_du_pointeur;
```

Un pointeur est typé. C'est à dire qu'il y a des pointeurs de type **`float`**, de type **`int`** etc. Si la variable à laquelle vous devez accéder est type **`float`**, il vous faudra un pointeur de type **`float`** (*respectivement* **`int`** *etc*.)

Un pointeur venant d'être créé ne "**pointe**" vers rien.

Lors de sa création un pointeur contient la constante **`NULL`**. Tel quel il n'est pas utilisable !

Il est impératif de mettre dans le pointeur l'adresse d'une variable qui existe. Sans cela on ne peut rien faire du pointeur.

Cette année vous utiliserez les pointeurs à deux titres :

Pour passer des paramètres aux fonctions.

Pour gérer de la mémoire dynamique (pour ceux d'entre vous qui sont les plus avancés)

Utilisation de pointeurs pour échanger des données avec les fonctions :

Une fonction peut avoir des paramètres en entrée, ils sont indiqués entre parenthèses. De même une fonction peut avoir un type, il est indiqué avant le nom de la fonction :

```
type nom ( liste de paramètres )
```

Les paramètres d'une fonction sont typés. Il peuvent être de type **`float`**, **`int`** ou bien être de type pointeur.

Lorsqu'on appelle une fonction, il faut :

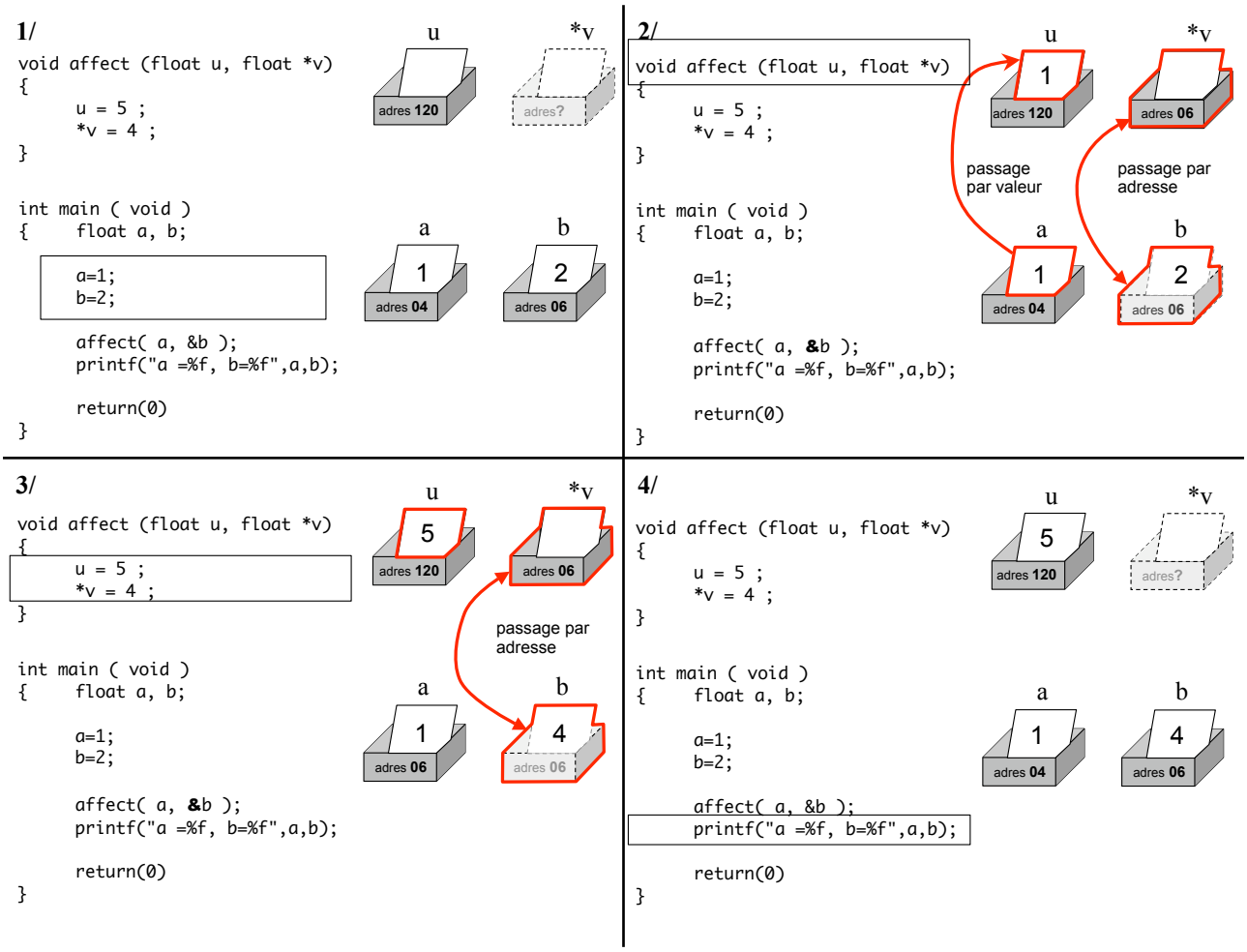
- fournir autant de paramètres qu'il y en a dans la liste.
- respecter les type et leur ordre

- si un des paramètres est un pointeur, transmettre l'adresse d'une variable qui soit du type du pointeur.

Lorsqu'on transmet à une fonction une variable, on transmet en fait une copie de son contenu, la variable d'origine n'est pas modifiée.

Lorsqu'on transmet à une fonction une adresse, on donne la possibilité à la fonction d'accéder à l'adresse en question, donc d'accéder à la variable elle même.

L'exemple suivant illustre bien ceci.



On a ici affaire à une fonction *affect* qui a deux paramètres, l'un est une variable de type **`float`**, l'autre est un *pointeur de type* **`float`**.

Le cadre montre le déroulement du programme et le contenu des différentes variables, *a* et *b* de la fonction *main()*, et *u* et **v* de la fonction *affect*.

Au point 1/

> le programme se déroule dans la fonction *main()* (cadre gris) et à ce moment on affecte à la variable a la valeur 1, et à la variable b la valeur 2.

Au point 2/

> on appelle la fonction *affect*. Lors de l'appel de la fonction *affect()* on transmet au paramètre *u* la valeur de la variable *a* et au paramètre **v* l'adresse de la variable *b* : "*&b*". Le contenu de *a* est copié dans la *variable locale* à la fonction *affect() u*. L'adresse de la

variable *b* est copiée dans le *pointeur local à la fonction affect v* ; en conséquence le lien entre *b* et **v* persiste alors qu'il est rompu entre *a* et *u*.

Au point 3/

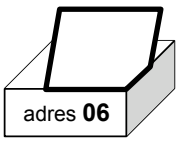
on modifie le contenu de *u* et le contenu de la variable dont l'adresse est contenue dans le pointeur **v*. *(lisez plusieurs fois pour bien comprendre !)* Ici il s'agit de la variable *b* locale à la fonction *main()*.

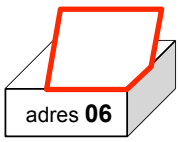
Au point 4/,

lorsque le programme se déroule à nouveau dans la fonction *main()* la variable *a* n'a pas été modifiée alors que la variable *b* si !

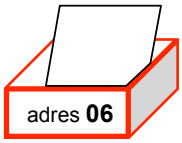
L'utilisation des variables et des pointeurs peut se résumer ainsi :

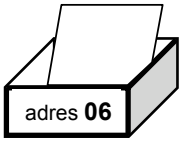
<table>
<tr><th>Déclaration d'une variable</th><th>Utilisation dans le programme</th></tr>
<tr><td>Déclaration par valeur

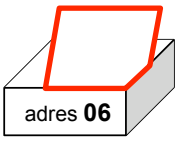
a  </td><td>Comment accéder à la valeur de la variable ?

a 

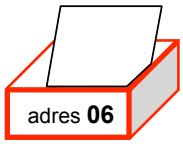
Comment accéder à l'adresse de la variable ?

&a  </td></tr>
<tr><td>Déclaration par adresse

*a  </td><td>Comment accéder à la valeur de la variable ?

*a 

Comment accéder à l'adresse de la variable ?

a  </td></tr>
</table>

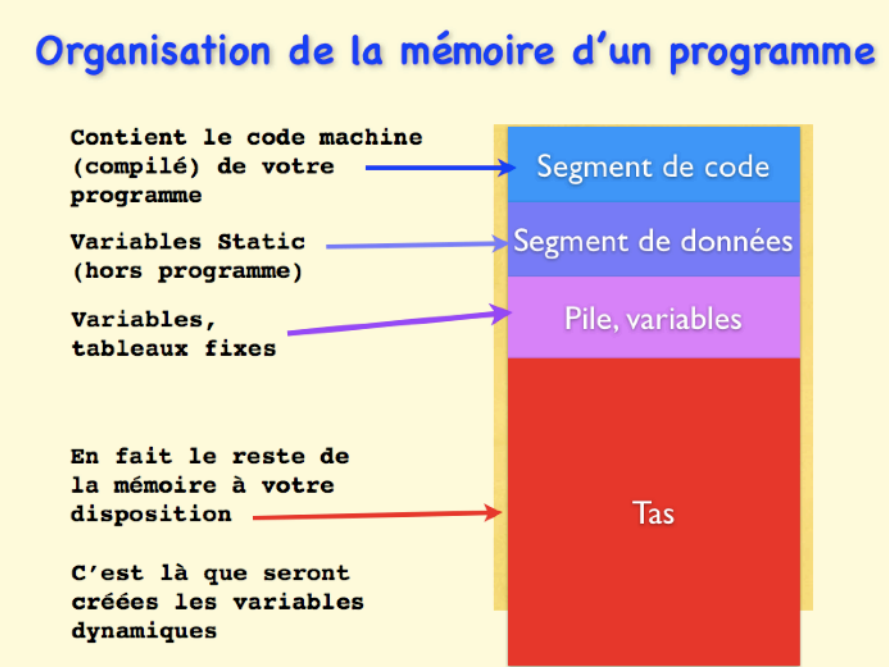
Gestion dynamique de la mémoire :

Comment est organisée la mémoire d'un programme ?

Cette question dépend du mode de compilation de votre programme. Il existe en effet plusieurs modèles en fonction de la taille de l'application que l'on va écrire.

Nous ignorerons ces directives, nous n'aurons pas cette année des programmes de taille telle qu'il faille choisir un modèle différent du modèle par défaut du compilateur gcc.

La figure ci-dessous résume l'organisation de la mémoire d'un programme C.



- Le segment de code contient le programme proprement. Il s'agit de la suite des instructions pour le microprocesseur.
- Le segment de données n'est en fait quasiment jamais utilisé pour nos programmes.
- La zone de pile, contient les environnements, les variables etc. Il faut comprendre que lorsqu'on appelle une fonction, on place les informations nécessaires au bon déroulement du programme lorsqu'il revient dans la fonction appelante.
- Le tas c'est le reste de la mémoire, c'est là que nous créerons les variables dynamiques.

En fait tous les programmes accèdent au tas, et c'est pour cela que c'est le système d'exploitation qui donne accès à la réservation de la mémoire.

L'accès à la mémoire dynamique se fait via des fonctions que le C met à la disposition, ces fonctions sont dans la bibliothèque **`stdlib.h`**.

Le processus pour réserver de la mémoire est le même que pour les fichiers : on **réserve** de la mémoire, on **l'utilise**, puis on la **libère**.

La fonction pour réserver de la mémoire est la fonction **`malloc()`**. la syntaxe est :

```
pointeur = malloc(nombre d'octets demandés);
```

`malloc` retourne un pointeur de type **`void`**, si on veut de la mémoire pour stocker des **`int`**, il faut "**caster**" la fonction. Par exemple si l'on veut une zone mémoire pour stocker 200 **`int`** :

```
int *p;
p = (int *)malloc(200*sizeof(int));
```

S'il n'y a plus de mémoire libre la fonction **`malloc`** retourne la valeur **`NULL`**.

Il est dès lors possible d'utiliser la mémoire comme une variable dimensionnée classique.

Enfin, une fois la mémoire utilisée il faut la libérer. La libération de la mémoire se fait avec la fonction **`free`**. La syntaxe est la suivante :

```
free(adresse du bloc à libérer).
```

Le programme suivant permet de créer un tableau de n entiers et l'utilise.

```
1 #include<stdio.h>
2 #include<stdlib.h>
```

```
 3 int main(void)
 4 {     int *p,n,k;
 5
 6     printf("Entrez la taille :");
 7     scanf("%d",&n);
 8     p = (int *)malloc(n*sizeof(int));
 9     if (p==NULL)
10     {    printf("Erreur");
11          exit(1);
12     }
13     for (k=0 ; k<n ;k++)
14          p[k] = k*k;
15     for (k=0 ; k<n ; k++)
16          printf("%d\n",p[k]);
17     free(p);
18     return 0;
19 }
```

TP 1 – L'environnement de développement.

Le système avec lequel vous travaillerez est un système UNIX dérivé d'une version de LINUX (Darwin pour être complet). Vous utiliserez un compilateur C et un éditeur de texte.

Ce sont ces éléments que nous allons vous présenter lors de ce premier TP.

Vous serez par la suite amenés à utiliser le logiciel Octave présent sur ces postes.

1/ L'éditeur de texte

Il s'agit d'un programme en licence libre pour les universités et particuliers nommé BBEDIT.

Un simple clic sur l'icône dans le Dock (la barre à droite de l'écran) permet le lancement de cette application.

La première action va consister à écrire un programme. Saisissez le code suivant dans la fenêtre :

```
#include<stdio.h>
int main(void)
{ printf("Coucou \n");
  return 0;
}
```

Pour obtenir les caractères "spéciaux" les combinaisons de touches sont les suivantes :

Caractère	Touche	Caractère	Touche
{	alt⌥ + (	]	alt⌥ +⇧+)
}	alt⌥ +)	\	alt⌥ +⇧+ /
[	alt⌥ +⇧+ (	| (pipe)	alt⌥ +⇧+ L

Attention le '/' est le '/' du pavé central, pas du pavé numérique.

Une fois le code saisi, enregistrez-le soit en déroulant le menu "Fichier" et en choisissant l'item "Enregistrer", soit en tapant au clavier le raccourci "⌘" + S.

Une fenêtre se déroule au dessus du document, naviguez jusqu'au dossier "Bureau".

Enregistrez sous le nom qui vous convient en n'oubliant pas de faire suivre le nom du suffixe ".c" ('c' **minuscule** et non 'C' majuscule).

Veillez à ne pas utilisez pas de caractère espace dans le nom du fichier, ni de caractère accentués ou de signe de ponctuation, parenthèse, etc. Vous n'avez le droit qu'aux 26 lettres de l'alphabet, au caractère souligné et aux chiffres.

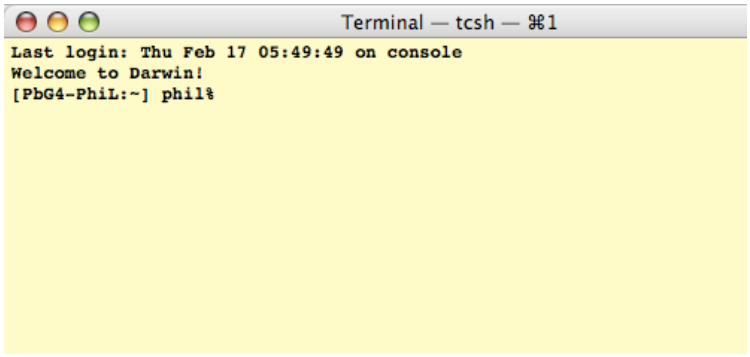
2/ Le Terminal

Une fois le programme sauvegardé, il vous faudra le compiler pour pouvoir exécuter le programme que vous venez d'écrire. La compilation se fera dans une autre fenêtre appelée "Terminal".

Pour lancer le terminal, cliquez sur son icône dans le Dock (un moniteur noir avec un prompt blanc ">")

Vous devez obtenir une fenêtre du type :

Manipulation 1



Cette fenêtre est un accès à la ligne de commande UNIX, un peu comme la fenêtre de commande dans un environnement Windows.

La première chose à faire une fois le terminal lancé, est d'aller dans le dossier de travail qui est le Bureau. Pour ce faire tapez :

```
cd Desktop
```

Puis vérifiez que vous vous trouvez bien dans le dossier travail : tapez

```
pwd
```

cette commande (Print Working Directory) affiche le chemin d'accès au dossier où vous vous trouvez. Cela doit ressembler peu ou prou à :

```
/Etudiant_1/etudiant/Desktop
```

Notez que contrairement à Windows, le séparateur sous UNIX est "/" et non "\".

Les commandes suivantes peuvent vous être utiles pour travailler sous UNIX :

Commande	Action
pwd	affiche le dossier courant
cd	change de dossier, taper " .." pour revenir au dossier parent
ls	affiche le contenu du dossier courant
clear	efface la fenêtre du terminal (il est possible avec l'ascenseur sur le coté de voir les résultats précédents, par défaut 10 000 lignes sont gardées)
↑	fait défiler les commandes saisies précédemment (remonte dans la liste)
↓	fait défiler les commandes saisies (descend dans la liste)

3/ Le compilateur

Pour compiler un programme il suffit de taper la commande **cc** suivie du nom du fichier.

La version du Shell de l'UNIX que vous utilisez comporte une fonction d'auto-complétion. Cela veut dire que le système est capable s'il n'y a pas ambiguïté de compléter tout seul la commande que vous tapez.

Par exemple, imaginons que vous ayez sauvegardé le programme saisi au point 1/ sous le nom toto.c. Tapez alors

```
cc t
```

puis appuyez sur la touche tabulation ("→|")

Le nom *toto.c* apparaît alors, vous n'avez plus qu'à valider en appuyant sur la touche "↩".

Le programme est alors compilé et un fichier nommé **a.out** est créé. Ce fichier est le programme. Vérifiez que ce fichier existe bien à l'aide de la commande **ls** (vous devez aussi le voir sur le bureau)

Pour vérifier que le programme fonctionne bien, il suffit de le lancer à l'aide de la commande :

```
./a.out
```

Vous devriez obtenir ceci :

```
Coucou
```

Bien entendu, il est possible que vous ayez commis des erreurs en saisissant le programme. Par exemple, omettez le point virgule après la commande printf dans le programme toto.c. Le code doit être celui-ci.

```
#include<stdio.h>
int main(void)
{ printf("Coucou \n")
  return 0;
}
```

➡ Cliquez dans la fenêtre de BBEDIT, et modifiez le code. Puis enregistrez-le.

Notez que si le document n'est pas enregistré, il apparaît un point noir dans le bouton rouge en haut à gauche de la fenêtre de l'éditeur.

➡ Activez le terminal (cliquez sur sa fenêtre ou sur son icône dans le Dock), puis compilez à nouveau le programme.

```
cc toto.c
```

Vous devriez obtenir quelque-chose ressemblant à ceci (suivant la version du compilateur installée) :

```
toto.c: In function 'main':
toto.c:5: error: parse error before 'return'
```

Note Bene:

*Il est possible d'arrêter un programme à tout instant en tapant au clavier "**ctrl + C**". Il est aussi possible de fermer la fenêtre du terminal. Il suffit d'en ouvrir une nouvelle et de ne pas oublier d'aller dans le dossier travail.*

4/ Travail à effectuer

➡ Ecrivez un programme qui calcule la valeur moyenne d'une série de nombre. Ce programme devra comporter les fonctions suivantes :

➡ *Saisie* : Saisie de la dimension et des éléments d'un vecteur.

➡ *moyenne* : la fonction calcule la moyenne des éléments d'un tableau.

4-c pour ceux qui ont du temps,

➡ *moyenne_pond* : cette fonction calcule la moyenne pondérée des éléments d'un vecteur. Deux vecteurs sont utilisés : un pour les données, l'autre pour les coefficients pondérateurs. Vous prévoirez une saisie des valeurs des valeurs et des coefficients pondérateurs dans une fonction.

TP 2 - Pointeurs

Objectifs

- Ce TP n'est en fait qu'une compilation des TP précédents avec comme seule consigne : Faite une fonction par action !
- Toutes les données seront transmises par adresse et on utilisera au maximum les pointeurs.

TP : calcul de surface / sauvegarde

Le fichier "*datastp3.txt*" comporte 3 colonnes séparées par des espaces. La première colonne contient les abscisses, la seconde les valeurs prises par une fonction *f1(x)*, la dernière colonne, les valeurs prises par une fonction *f2(x)*.

Il vous est demandé d'écrire un programme qui :

➡ Lise ce fichier

➡ Calcule la valeur de l'intégrale sur l'ensemble des points et l'affiche.

➡ Calcule les points de la fonction primitive F1(x) de la fonction f1(x)

➡ Calcule les points de la fonction primitive F2(x) de la fonction f2(x)

➡ Sauvegarde ces points dans un fichier contenant les abscisses, les points pris par la fonction F1(x) et ceux pris par la fonction F2(x).

➡ Trace ces points avec Octave.

Pour ce faire vous écrirez les fonctions avec les interfaces suivantes :

```
typedef float vect[128];
void lecture( vect x, vect y, vect z, int *n);
void ecrire( vect x, vect y, vect z, int n);
void trapeze(vect Y, float h, int n, float *I);
void primitive(vect Y, float h, int n, vect prim);
```

Les noms des fonction sont explicites et permettent de savoir ce que chaque fonction fait.

Votre programme devra impérativement, et en commentaire, préciser le rôles de chaque variable et si elle est passée par adresse ou par valeur.

➡ Modifiez votre programme pour qu'il utilise des variables dynamiques.

TP 3 – Fichiers

Objectifs :

- Savoir lire / créer / écrire dans un fichier texte en langage “C”.
- Savoir utiliser Octave pour tracer les courbes obtenues par le calcul en langage “C”.
- Savoir calculer la valeur d'une intégrale et déterminer les valeurs d’une fonction primitive

TP : Lecture de fichiers / Tracé sous Octave

➡ Écrivez un programme permettant d’écrire un fichier de type texte contenant les valeurs de x et sin(x) où x varie de 0 à 2π. Utilisez l’espace comme caractère de séparation. Le fichier devra s'appeler obligatoirement "sortie.txt" et être sauvegardé sur le Bureau (Desktop) de votre machine. Vous calculerez le cosinus sur 128 points.

➡ Lors de l'écriture des données dans le fichier un écho sera affiché à l’écran dans le terminal.

Votre programme devra utiliser une fonction d’écriture ayant le prototype suivant :

```
void ecrire(vect x, vect y, int n);
```

Les paramètres sont deux vecteurs et le numéro du dernier élément. Le type **vect** sera défini comme suit :

```
typedef float vect[128];
```

Nota Bene :

Pour des vecteurs il n’est pas syntaxiquement obligatoire de préciser la dimension des tableaux. Afin de faciliter votre travail, nous vous demandons de spécifier systématiquement la taille des vecteurs. Mieux, définissez des types appropriés !

➡ Lancez Octave en cliquant sur son icône dans le Dock.

vous devez obtenir

```
octave-3.0.5:1>
```

➡ Créez un nouveau fichier dans BBEDIT. Puis saisissez le code suivant :

```
load -ascii sortie.txt;
[l,c] = size(sortie);
clf
grid on
hold on
for k=2:c
  plot(sortie(:,1),sortie(:,k),';;')
end
hold off
```

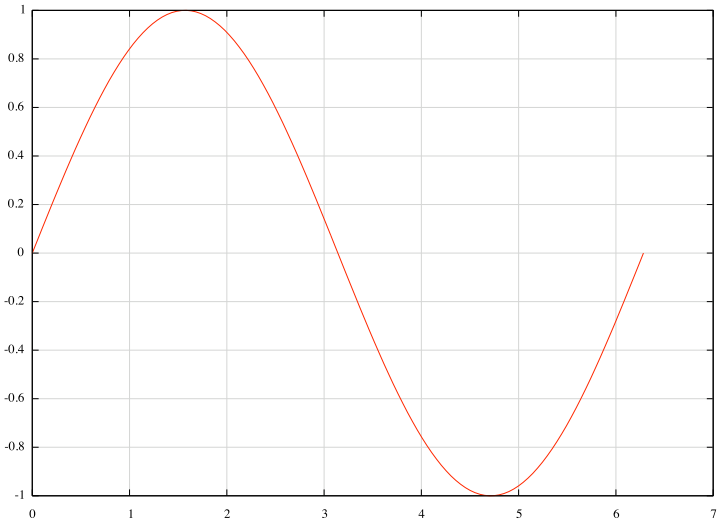
➡ Sauvegardez le nouveau fichier sous le nom **tracer.m**.

Vous venez de créer votre premier script Octave/Matlab. Ces instructions seront détaillées/ expliquées dans le BE matlab (début des TP fin octobre).

Si vous tapez :

```
tracer
```

dans le terminal ou Octave est lancé, vous devez obtenir la figure suivante :



TP : Intégration à pas constant – Méthode des trapèzes.

Considérons un ensemble de points $f(x_i)$ décrivant un signal ou une fonction $f(t)$. Nous supposons connaître l'abscisse de début x_0 (-> $f(x_0)$ et de fin x_n (-> $f(x_n)$ du domaine d'intégration, h, l'écart entre deux points consécutifs (le pas) est supposé constant.

L'intégrale :

$$I = \int_{X_0}^{X_n} f(t)dt$$

peut être approchée par :

$$I \approx \frac{h}{2}\left[f(x_0) + f(x_n) + 2\sum_{i=1}^{n-1} f(x_i)\right]$$

Il est possible de déterminer la fonction primitive (par ses points) via la méthode suivante :

Notons $F(x_r)$ la valeur de la primitive en $x=x_r$, par définition :

$$F(X_r) = \int_{0}^{X_r} f(t)dt$$

Il suffit donc de calculer l'intégrale I entre :

X0 et X1 pour F(X1),

X0 et X2 pour F(X2),

…

X0 et Xn pour F(Xn).

Travail à effectuer

➡ Écrivez un programme qui calcule les valeurs prises par la fonction qui à x associe 5*x+3, dans l'intervalle [0, 10] sur 101 points régulièrement espacés. Calculez à partir de ces points une valeur approchée de l'intégrale :

$$I = \int_0^{10} (5x+3)dx$$

Vous écrirez une fonction “intégrale” qui aura le prototype suivant :

```
float integrale(vect y, float h , int n);
/* y contient les ordonnées, h le pas, n numéro du dernier
point<128 !! */
```

➡ Complétez votre programme pour qu’il calcule les points de la fonction primitive. Tracez ces points sous Octave à l’aide du script écrit précédemment.

➡ Comparez la valeur théorique de la fonction primitive en x=5 avec la valeur calculée.

Circuit Magnétique.

Rédacteur : P. Castelan

Objectifs :

Dans ce projet de niveau 1, nous vous proposons de mettre en oeuvre les techniques d'intégration, comprendre les circuits magnétiques.

Position du problème :

Vous avez mesuré aux bornes d'un circuit magnétique (une bobine à noyau de fer comportant n=440 spires de section S=1,56 10^{-3} m^2 et de longueur l=480 mm) le champ magnétique B et l'excitation magnétique H.

Ces données sont contenues dans le fichier "CM.txt". La première colonne de ce fichier contient H, la seconde B, le tout en unités mKsA.

On veut estimer la puissance active consommée par cette bobine. Pour ce faire deux solutions sont envisageables :

1/ par le calcul de la surface du cycle d'Hystérésis en tenant compte du volume du circuit magnétique. On a : $E = Sl \oint HdB$. Il faut alors calculer la puissance moyenne qui est égale à l'énergie consommée en une seconde. C'est donc F*E où F est la fréquence à laquelle la bobine est alimentée.

2/ Par le calcul de la puissance instantanée dans le circuit : P(t) = V(t)I(t), ce qui permet de calculer la puissance moyenne : $P = \frac{1}{T}\int_{t_0}^{t_0+T} P(t)dt$.

Rappels de Physique :

- $V = n\frac{d\Phi}{dt} = n\frac{B.S}{dt} = nS\frac{dB}{dt}$
- $Hl = nI$
- $E_{par\ periode} = V_{circuit\ magnetique} \oint BdH$

Indications

Il vous faudra passe de B(H) à V(t) en sachant que le signal a une durée de 20ms.

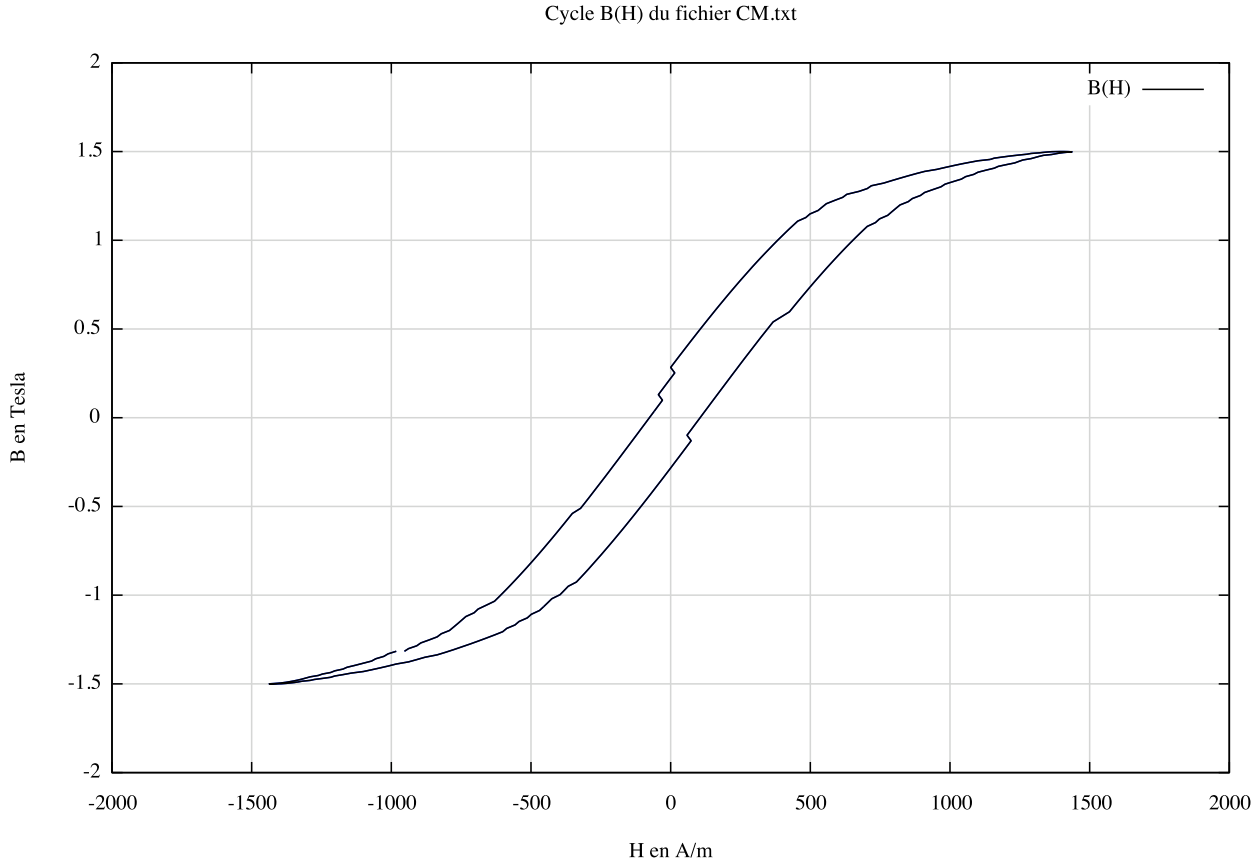
Vous serez amenés à intégrer B(H). La surface du cycle. Il est possible de calculer cette surface en décomposant le cycle en deux courbes, la supérieure et l'inférieure. Ensuite il suffit de calculer la surface en partant du point où H est nul pour les deux courbes. La surface sur le demi cycle est alors la différence :

$$S = \int_{H=0}^{H_{max}} B_{courbe\ sup.}dH - \int_{H=0}^{H_{max}} B_{courbe\ inf.}dH$$

Il vous faudra donc déterminer H_{max}.

L'intégration numérique se fera par la méthode des trapèzes vue précédemment.

Pour la dérivation numérique, utilisez des méthodes à fenêtre centrée (ordre 3 ou 5). Vous trouverez des informations sur ces méthodes dans le livre " méthodes de calcul numérique" de M. Nougier qui est à la Bibliothèque Universitaire.


Cycle B(H) du fichier CM.txt
B(H)
B en Tesla
H en A/m
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-2000
-1500
-1000
-500
0
500
1000
1500
2000

Point de fonctionnement d'un circuit

Rédacteur : Thierry Callegari

Nous vous proposons dans ce projet de niveau 1 de mettre en œuvre deux méthodes numériques (Dichotomie et Newton) afin de déterminer le point de fonctionnement d'un circuit. De façon plus générale, ces deux méthodes permettent de résoudre une équation du type f(x) = 0.

1. Positionnement du problème

Le circuit considéré est représenté sur la figure 1. Il se compose d'une source de tension de force électromotrice E = 1 V en série avec une résistance R = 100 Ω et d'une diode. Le courant traversant la diode est donné par l'expression :

$$I = I_0\left(e^{\eta U/U_0} - 1\right)$$ où $I_0 = 10^{-15}$ A, $\eta = 0.68$ et $U_0 = 0.025$ V

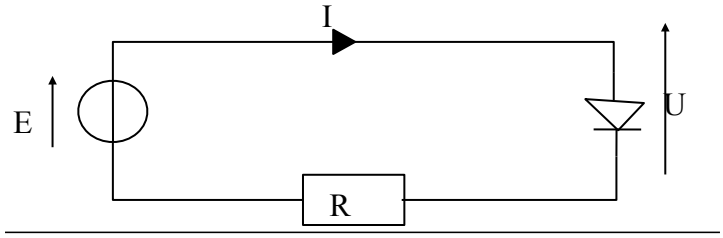


Figure 1

Le but est donc de déterminer le couple unique (U, I) qui correspond au fonctionnement du circuit sachant que la valeur de U est comprise entre 0 et 1 V.

2. Mise en œuvre

A. Mise en forme de l'équation

Les méthodes de Newton et de dichotomie résolvent une équation du type f(x) = 0. Vous écrirez donc l'équation qui régit l'évolution de la tension U sous la forme f(U) = 0.

.

B. Ecriture des fonctions suivantes

f0 : Calcule la valeur de f(U)
f1 : Calcule la valeur de f '(U)
dichotomie : Détermine la solution de l'équation f(U) = 0 par la méthode de dichotomie
newton : Détermine la solution de l'équation f(U) = 0 par la méthode de Newton
echantillon : Echantillonne l'intervalle [0,1] sur 100 points

C. Validation du résultat

Vous déterminerez le résultat à l'aide des deux méthodes en confrontant leur rapidité et leur précision. Vous contrôlerez le résultat en traçant sur Octave les deux caractéristiques I(U) de la diode et du générateur. **Le projet sera validé par la remise du script commenté et des courbes.**

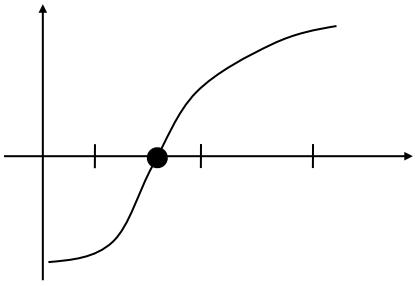
3. Documentation sur les méthodes numériques

Vous trouverez une documentation détaillée des méthodes numériques à la bibliothèque de l'UPS mais aussi sur de nombreux sites web.
Nous vous proposons ici un bref descriptif.

Dichotomie :

*Soit une fonction f(x) donnée qui s'annule une fois dans un intervalle [a,b] connu. On en déduit que a et b vérifient f(a)*f(b) < 0. Soit* α *la racine de la fonction dans cet intervalle. Le calcul de* α *se fait en diminuant à chaque itération la largeur de l'intervalle dans lequel se trouve la racine. Le calcul se poursuit tant que la largeur de l'intervalle est supérieure à une précision 2**ε *demandée.*

Ainsi,
si :
f(a)*f((a+b)/2) < 0 ➔ $\alpha \in$ à [a, (a+b)/2]
b = (a+b)/2,
sinon
a = (a+b)/2
f(x)



Newton :

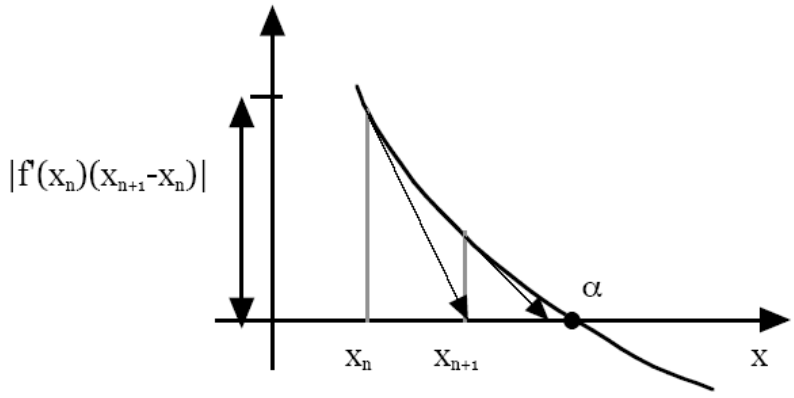
Soit une fonction f(x) donnée qui s'annule une fois à l'abscisse α *d'un intervalle [a,b] connu.*
L'itération de Newton s'écrit à partir d'une valeur x_0 *initiale:* $x_{n+1} = x_n - f(x_n)/f'(x_n)$
La valeur α *est la limite de la suite récurrente* $\{x_n\}$*. Cette méthode suppose donc que la dérivée f '(x) ne s'annule pas dans l'intervalle [a,b].*

Conditions de convergence de la suite {xn}

On est assuré de la convergence si f est strictement monotone, de concavité constante dans [a,b] et si on choisit comme point initial $x_0 \in [a,b]$ *une valeur telle que f(x0) * f ''(x0) >0.*

Interprétation graphique de la méthode

La formule de Newton peut également s'écrire : $f'(x_n)(x_{n+1} - x_n) + f(x_n) = 0$
On voit ainsi que, x_{n+1} *est l'abscisse du point d'intersection entre la tangente de f(x) en* x_n *et l'axe des x.*

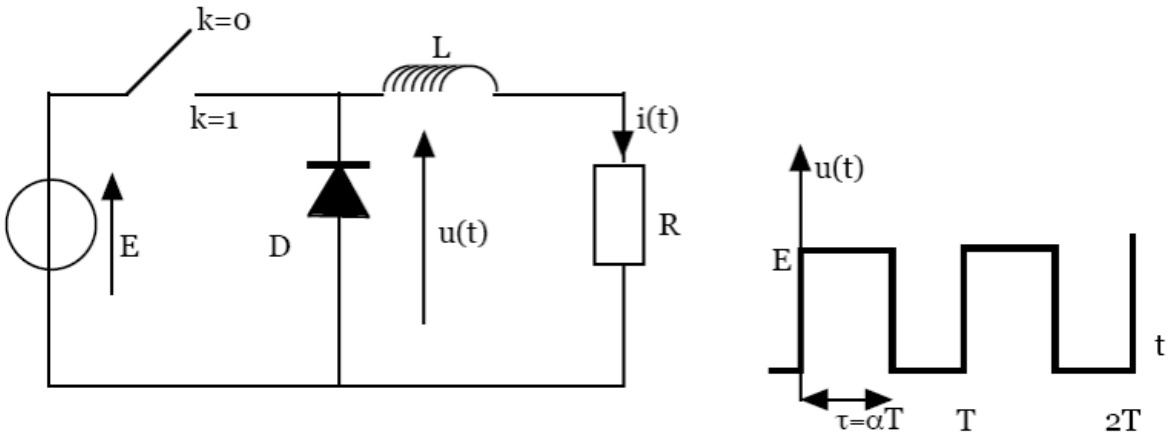


Hacheur Abaisseur de Tension

Rédacteur : H. Caquineau

Objectif :
Dans ce projet de niveau 1, nous vous proposons de mettre en œuvre une des techniques les plus utilisées d'intégration d'équations différentielles ordinaires : la méthode de Runge-Kutta.

Position du problème :
Le hacheur est un outil qui permet de modifier la valeur d'une source de tension continue via l'utilisation d'un interrupteur commandé. Suivant la configuration du circuit électrique, on peut soit augmenter la tension moyenne fournie, soit l'abaisser.
Le circuit présenter ci-dessous permet de diminuer la tension continue. L'évolution temporelle de la tension u(t) est représentée sur la figure à droite du circuit. Il s'agit d'une modulation de la tension appliquée constante E qui dépend de l'ouverture ou de la fermeture d'un interrupteur commandé. k est une constante d'état de l'interrupteur qui vaut 1 quand il est fermé et 0 sinon. La diode D est considérée parfaite. La durée du cycle totale d'ouverture/fermeture de l'interrupteur est T. A cette durée, on pourra faire correspondre la fréquence f. α est le rapport cyclique de fonctionnement du hacheur : pendant αT l'interrupteur est fermé et il est ouvert le reste de la période.



Le but de ce mini-projet est de tracer en fonction du temps la tension aux bornes de la charge R.

A cette fin, il vous faudra écrire en utilisant la loi des mailles (sur un papier) les expressions de $\frac{di}{dt}$ pour k = 1 et pour k = 0 en introduisant la constante de temps τ du système qui vaut L/R. Vous chercherez alors une expression unique de $\frac{di}{dt}$ permettant de décrire ces 2 expressions (vous pourrez pour cela utiliser une variable nommée "**etat**" dont la valeur dépend du temps et en particulier du positionnement dans la période). Vous utiliserez la méthode de Runge-Kutta (ordre 2 ou 4 au choix) pour connaître l'évolution temporelle de la tension aux bornes de la charge sur un certain nombre "**ncycle**" de cycles. En utilisant matlab, vous tracerez cette évolution.

Indications :

Dans l'ouvrage intitulé " méthodes de calcul numérique" de M. Nougier que vous trouverez à la Bibliothèque Universitaire, vous trouverez les principes de base des méthodes de Runge-Kutta. On trouve cela également très facilement sur la toile.

Ces méthodes permettent de calculer à partir d'une équation différentielle de type $\frac{dy}{dt} = f(y,t)$ la valeur de y pour un temps t + dt à partir de la connaissance de y au temps t. On vous conseille d'écrire une fonction "**RK**" qui effectue seulement ce passage du temps t au temps t + dt. Pour utiliser cette fonction, il vous faudra bien entendu discrétiser l'intervalle temporel de résolution en un certain nombre "**npas**" de pas de taille "**dt**".

Exceptionnellement, vous pourrez déclarer "**etat**" en tant que variable globale.

Données numériques :

L = 500 μH R = 10 Ω E = 30 V f = 10 kHz α = 0,5 ncycle = 5
Le courant dans l'inductance est nul avant la mise en service du hacheur.

Exercice 1

Objectif : Ecrire un programme simple qui mette en œuvre les fonctions **printf** et **scanf**. utilisation de l'instruction if pour des tests simples, et de **switch**. Étude des sorties formatées avec **printf**.

- Ecrivez un programme qui demande à l'utilisateur de saisir son âge, puis affiche le message, "vous avez [la valeur saisie] ans".
- Ecrivez un programme qui demande à l'utilisateur de saisir deux valeurs entières relatives puis affiche leur somme.
- Ecrivez un programme qui demande à l'utilisateur de saisir deux valeurs entières relatives A et B puis affiche leur différence.
  Complétez ce programme pour qu'il affiche l'un des trois messages suivant le cas :

```
"La différence A - B est plus grande que A"
"La différence A - B est plus petite que A"
"La différence A - B est nulle".
```

Dans les messages ci-dessus, A et B seront remplacées par les valeurs saisies.

- Complétez le programme précédent pour qu'il saisisse des valeurs réelles. L'affichage des valeurs dans les trois messages se fera alors sur 8 caractères et 3 décimales.
- Ecrivez un programme qui saisisse deux valeurs puis demande l'opération à effectuer : les réponses possibles seront +, -, * ou /. Vous utiliserez d'abord une série de test puis un switch pour choisir le type de calcul à effectuer.
- A ce point vous devez être capable d'utiliser les instructions printf et scanf sans réfléchir à leur syntaxe. Vous devez connaître la syntaxe de l'instruction switch

Exercice 2

Objectif : Ecrire un programme qui permette le calcul d'une somme d'entiers saisis au clavier. découverte des test logiques et des boucles **for** et **while/do while**. Règles d'usage de ces différentes boucles.

On se propose d'écrire un programme qui calcule la somme de N entiers entre 0 et 20 La saisie des nombres et le calcul de la somme se feront dans une même boucle.

- Dans une première version du programme, N est demandé à l'utilisateur avant la saisie des valeurs
- Dans une seconde version du programme, la saisie des valeurs dure tant que l'on n'entre pas un code d'arrêt, ce code sera le nombre 99
- Prévoyez un test pour que x, la valeur saisie, reste dans les limites imposées $0 \leq x \leq 20$, d'abord avec le programme du point 1, puis avec le programme du point 2.

Exercice 3

Objectifs : Ecrire un programme qui calcule une moyenne, comprendre l'adaptation de l'opérande à l'opérateur, apprendre à réutiliser un programme précédent.

- Editez le programme du point 2 de l'exercice 2. Modifiez-le pour qu'il affiche en fin de programme le nombre de valeurs saisies et la somme.
- Calculer la valeur moyenne et affichez-là. Que constatez-vous ?
- Quelle solution proposez-vous ?
- Modifier le programme pour calculer uniquement la moyenne des éléments pairs. On rappelle que l'opérateur % calcule le reste de la division entière: par exemple 7%2 retourne 1, puisque 7 = 3*2 + 1.

Exercice 4

Objectifs : Ecrire un programme qui calcule la factorielle d'un **entier** quelconque. Utilisation d'une boucle for décroissante ou d'un test.

On rappelle que la factorielle est une fonction qui à un entier positif N associe :
N! = N*(N-1)*(N-2)*…*2*1, par convention 0! = 1

- Ecrivez un programme qui calcule N! d'un entier positif quelconque (0 compris).
- Calculez avec votre programme 8! (ou 17! suivant les réglages du compilateur). Que constatez-vous ? Quelle solution proposez-vous pour corriger cette erreur ?

Exercice 5

Objectifs : Programme qui calcule la moyenne de N **entiers**, utilisation de **variables dimensionnées.**

- Saisir le nombre de valeurs
- Saisir les valeurs dans une boucle.
- Calculer la somme des éléments du vecteur dans une autre boucle.
- Affichez la valeur moyenne

Exercice 6 : Moyenne

Objectif : Utiliser une fonction typée simple.

Reprenez le programme qui calcule la moyenne avec un tableau.

- Ecrivez la fonction moyenne avec le prototype suivant :

```
float moy(float somme , int N);
```

Cette fonction calcule la valeur moyenne des éléments d'un vecteur.

- Modifiez la fonction main( ) pour qu'elle utilise moy().
- Conclusion ?

Exercice 7 : Fonction et variable dimensionnée

O**bjectif** : Passer un tableau dans une fonction

- Ecrire une fonction saisie avec le prototype suivant:

```
void saisie(float A[], int N);
```

➡ Dans le main, affichez les valeurs saisies et calculer la somme

➡ En utilisant la fonction écrite dans l'exercice précédent, calculer la moyenne

Exercice 8 : synthèse

O**bjectif** : Passer un tableau dans une fonction

➡ Ecrire une fonction somme avec le prototype suivant:

```
float som(float A[], int N);
```

Cette fonction réalisera la somme des N réels stockés dans le tableau A

➡ Dans la fonction main, remplissez le tableau A grâce à une fonction appelée saisie, calculez la somme des éléments de A grâce à une fonction appelée som, et calculez la moyenne des éléments grâce à une fonction appelée moy.

Exercice 9 : Factorielle (facultatif)

Reprenez le programme qui calcule la factorielle.

➡ Ecrivez la fonction fact( ) avec le prototype suivant :

```
float fact( int N );
```

Cette fonction calcule la valeur de factorielle N.

➡ Modifiez la fonction main( ) pour qu'elle utilise fact( ).

TP 1 : Retour sur les fonctions

1. Rappels.

1.1. Fonctions

Une fonction est un bloc d'instruction que l'on peut appeler en lui transmettant des paramètres. Les paramètres sont des variables locales à la fonction, c'est à dire qui n'existent que dans le bloc d'instructions de la fonction, elles sont initialisées avec les valeurs transmises à l'appel.
Les fonctions ont un nom et un type. Le type peut être un des types simples, **`void`** ou un type de structure.
Dans le cas ou le type n'est pas **`void`**, la fonction comporte une instruction **`return`**.
L'instruction **`return`** est suivie d'une expression du même type que celui de la fonction.

```
type nom_de_fonction ( liste de variables paramètres )
{  ...
   return expression ; // du type de la fonction
}
```

L'appel d'une fonction se fait avec son nom et en précisant la valeur de chacun de ses paramètres.
Toute fonction (sauf main) ***doit*** avoir un prototype. Les prototypes sont ***obligatoirement*** groupés en début de programme après les directives de compilation et les déclarations de type. Ils sont constitués de la ligne d'en tête de chacune des fonctions du programme suivit d'un point virgule.

1.2. Déclaration de types

Il est possible de déclarer des types à l'aide de l'instruction **`typedef`**. C'est une instruction, il y a donc un point virgule en fin de ligne. Pour un type de variable dimensionné la syntaxe est :

typedef type_de_variable_simple Nom_du_nouveau_type[nombre d_éléments] ;

En général, on déclare des types globaux en début de programme ***après*** les directives de compilation et ***avant*** les prototypes.

2. Exercice 1

★ Ecrivez une fonction qui calcule la valeur de la fonction $F(t) = 230\sqrt{2}sin(\omega t)$ en ***un*** point t.
- ✦ Déterminez les paramètres d'entrée de la fonction.
- ✦ Donnez le type de variable retournée.
- ✦ Choisissez un nom de fonction, et écrivez-là ainsi que son prototype.
- ✦ À l'aide de la fonction écrite, calculez les valeurs prises par un signal de fréquence 50 Hz durant ***une*** période sur 100 points. (Calculs dans la fonction main())

★ Modifiez la fonction pour qu'elle retourne tous les points dans une variable dimensionnée. Déterminez les paramètres de cette fonction ainsi que son type.

★ Créez un type de variable dimensionnée que vous appellerez ***datas***. Modifiez la fonction précédente pour qu'elle utilise en paramètre une variable de type ***datas***.

3. Exercice 2

- ✦ Créez un type de variable appelé ***vect*** pouvant contenir ***dim*** éléments réels. ***dim*** est une constante définie en directive de compilation.
- ✦ Ecrivez une fonction appelée ***saisie***, qui saisisse les coordonnées d'un vecteur de dimension ***dim***. ***dim*** est-il un paramètre de la fonction ?
- ✦ Ecrivez une fonction appelée ***produit*** qui calcule le produit scalaire de deux vecteurs. Déterminez quelles sont les paramètres nécessaires à cette fonction.

★ À l'aide de ces deux fonctions, saisissez deux vecteurs X et Y, puis calculez le produit scalaire de ces deux vecteurs et affichez le résultat.
- ✦ Créez un type de variable appelé ***mat***, permettant de stocker les éléments d'une matrice composée d'une collection de ***dim vect***.

- ✦ Ecrivez une fonction appelée ***extrait***, qui extrait d'une matrice ***M*** la colonne ***k*** et la stocke dans un vecteur ***V***. Déterminez les paramètres de la fonction.
- ✦ Ecrivez une fonction affiche qui affiche sous forme matricielle une matrice à l'écran. Déterminez les paramètres nécessaires à cette fonction.

★ À l'aide des fonctions ***produit*** et ***extrait***, calculez le produit matriciel de deux matrices A et B dans la fonction main. La saisie des deux matrices se fera à l'aide de la fonction ***saisie*** dans la fonction ***main***.

★ Testez votre programme sur des matrices de rang 3 et des vecteurs de dimension 3.

4. Rappels produit scalaire / Produit matriciel

$^t\overrightarrow{V_1} = 3\overrightarrow{e_x} + 2\overrightarrow{e_y} - \overrightarrow{e_z} : V_1 = \{3 \quad 2 \quad -1\}$ et $\overrightarrow{V_2} = 3\overrightarrow{e_x} + 2\overrightarrow{e_y} - \overrightarrow{e_z} : V_1 = \begin{cases} 2 \\ 1 \\ -1 \end{cases}$

$^tV_1 \,.\, V_2 = \sum V_1^k V_2 k$ où V^k est la coordonnée k du vecteur V

$^tV_1$ est un vecteur à gauche ou vecteur ligne, V_2 est un vecteur à droite ou vecteur colonne. On note les vecteurs ligne ^tV

Une matrice est un collection de vecteurs : $M = [[V_1] \ldots [V_n]]$

Ainsi la matrice $M = \begin{bmatrix} 1 & 2 & 3 \\ 5 & 6 & 7 \\ 9 & -4 & 3 \end{bmatrix}$ est en fait le regroupement des trois vecteurs : $V_1 = \begin{bmatrix} 1 \\ 5 \\ 9 \end{bmatrix}$,

$V_2 = \begin{bmatrix} 2 \\ 6 \\ -4 \end{bmatrix}, V_3 = \begin{bmatrix} 3 \\ 7 \\ 3 \end{bmatrix}$,soit $M = \begin{bmatrix} \begin{bmatrix} 1 \\ 5 \\ 9 \end{bmatrix} \begin{bmatrix} 2 \\ 6 \\ -4 \end{bmatrix} \begin{bmatrix} 3 \\ 7 \\ 3 \end{bmatrix} \end{bmatrix}$

Mais la matrice M peut aussi être vue comme composée de vecteurs Ligne :

$$M = \begin{bmatrix} \begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \\ \begin{bmatrix} 5 & 6 & 7 \end{bmatrix} \\ \begin{bmatrix} 9 & -4 & 3 \end{bmatrix} \end{bmatrix} = \begin{bmatrix} {}^tU_1 \\ {}^tU_2 \\ {}^tU_3 \end{bmatrix}$$

Le produit matriciel est donc le produit des vecteurs ligne de la matrice à gauche par les vecteurs colonnes de la matrice à droite :

$$M.M = \begin{bmatrix} {}^tU_1 \\ {}^tU_2 \\ {}^tU_3 \end{bmatrix} . [[V_1] \, [V_2] \, [V_3]]$$

Il s'agit donc d'un ensemble de produits scalaires et l'élément ligne i, colonne j de la matrice résultat est en fait le produit scalaire du vecteur $^tU_i.V_j$

Exemple de résultat :

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 5 & 6 & 7 \\ 9 & -4 & 3 \end{pmatrix}; B = \begin{pmatrix} 2 & 6 & 7 \\ -5 & 2 & 7 \\ 1 & 0 & 1 \end{pmatrix}; A * B = \begin{pmatrix} -5 & 10 & 24 \\ -13 & 42 & 84 \\ 41 & 46 & 38 \end{pmatrix}$$

TP 2 : les Pointeurs

1. Rappels.

1.1. Pointeurs simples

Un pointeur est une variable qui contient des adresses. Au même titre qu'une variable de type **`int`** contient un entier, le contenu d'un pointeur est une adresse.

On déclare un pointeur en faisant précéder son nom d'un astérisque.

```
type *nom;
```

Les variables n'ayant pas toutes la même longueur les pointeurs sont typés. Ainsi, il y a des pointeurs sur des **`int`**, des pointeurs sur des **`float`** etc.

Avec un pointeur il est possible d'afficher, de manipuler des adresses.

```
printf("%X", adresse) ; // affiche une adresse sous forme hexadécimale (base 16)
```

Il est possible d'ajouter des adresses. Si X est une adresse correspondant à une variable **`int`**, alors X+1 est l'adresse du prochain **`int`** possible.

1.2. Variable pointée.

Si P est un pointeur, on appelle variable pointée, la variable dont l'adresse est contenue dans P.

Il est possible d'obtenir l'adresse de la variable pointée en utilisant le pointeur P : c'est son contenu

Il est possible d'obtenir le contenu de la variable pointée en utilisant le pointeur P, il suffit d'utiliser l'opérateur d'indirection *. Le tableau ci-dessous résume l'utilisation d'un pointeur P déclaré comme suit :

```
int a, *P;
P = &a ; // P contient désormais l'adresse de la variable a.
```

Code	*Signification*	*Exemple*
P	*Utilise le contenu de P, c'est à dire l'adresse de a*	*scanf("%d",P) ;*
**P*	*Utilise le contenu de la variable pointée par P, c'est à dire, ici, a.*	*printf("a vaut : %d\n",*P);*

1.3. Variables dimensionnées

Une variable dimensionnée est une collection de variables du même type, regroupées sous un même nom, auxquelles on accède à l'aide d'un numéro appelé indice. On place cet indice entre crochets droits.

```
type nom[ nombre_d_éléments ] ; // déclare une variable appelée nom du type spécifié.
```

Le nom d'une variable dimensionnée est l'adresse du premier élément de cette variable. Il s'agit donc d'un pointeur, mais il est ***impossible*** de modifier le lien ***nom = &nom[0];*** contrairement aux pointeurs simples.

À l'aide de l'addition des adresses, il est possible de calculer l'adresse de l'élément k d'une variable dimensionnée :

```
&nom[ k] <=> nom + k ;
```

Il est donc possible d'accéder au contenu de la variable d'indice k d'une variable dimensionnée à l'aide de l'opérateur d'indirection.

```
nom[k] <=> *(nom + k)
```

La parenthèse est obligatoire, l'indirection est prioritaire sur l'addition.

2. Exercice 1

Tout ce qui suit doit être écrit dans la fonction main.

★ Créez deux variables de type **`float`** a, b, ainsi que deux pointeurs p1 et p2. Faites pointer p1 vers a, p2 vers b.

A partir de maintenant vous n'accéderez plus aux variables a et b qu'à travers les pointeurs.

★ Entrez, à l'aide d'instructions **`scanf`** un contenu dans a et b.

★ Calculez b = b*a

★ Affichez le contenu de a et de b

★ Comparaison : si a est plus grand que b affichez "A est plus grand" ; "B est plus grand ou égal" sinon.

L'utilisation de pointeurs tel que vu dans cet exercice est possible. Toutefois, convenez-en, cela ne sert à rien, autant utiliser a et b ! Cet exercice n'a d'autre but que de vous faire manipuler les syntaxes pointeur.

3. Exercice 2

★ Ecrivez une fonction qui saisisse deux **`float`** à l'aide de pointeurs, le prototype sera
void saisie(float *a, float *b);

★ Ecrivez une fonction calcule le produit de deux **`float`**, et retourne le résultat à l'aide d'un pointeur. Le prototype sera d'abord :
void calcul1 (float *a, float *b); // résultat dans b

puis

void calcul2 (float a, float b, float *c); // résultat dans c

★ A l'aide des trois fonctions écrites ci dessus, depuis la fonction main, saisissez deux nombres réels, puis calculez et affichez leur produit. Vous n'utiliserez que deux variables dans main. D'abord avec la fonction calcul1, puis calcul 2.

4. Exercice 3

★ Dans la fonction main, déclarez une variable dimensionnée appelée X pouvant contenir 50 **`float`**.

★ Ecrivez une fonction appelée *saisie*. Elle demandera le nombre de points que l'utilisateur veut saisir, puis procédera à la saisie des points. L'interface de la fonction sera

```
void saisie( float x[50], int *n);
```

✦ Saisissez les valeurs suivantes : { 5, 9, 4, -2 } à l'aide de *saisie*, puis affichez-les depuis la fonction main.

★ Même question avec cette fois l'interface suivante pour la fonction saisie :

```
void saisie( float *x, int *n);
```

★ Créez un type de variable pouvant contenir 50 **`float`** appelé datas.

✦ Remplacez dans la fonction main, la déclaration de X par **`datas X;`** Lancez le programme *sans modifier la fonction saisie*. Conclusion ?

✦ Modifiez le programme précédent pour que l'on saisisse 4 valeurs à partir de l'élément 9 (donc dans **`X[9], X[10], X[11], X[12]`**). Affichez les 15 premiers éléments de x. Conclusion ?

★ Complétez le programme précédent en écrivant une fonction qui affiche à l'écran, les n dernières valeurs en ordre inverse d'une variable dimensionnée ainsi que leur position, la fonction aura l'interface suivante :

```
void affiche( float *P, int n);
```

Indication : Si l'on appelle la fonction ainsi : **`affiche( &X[10], 3),`** cela provoquera l'affichage suivant :

```
 0   X[10] // comprendre par X[10] la valeur de cette variable.
-1   X[9]  // idem etc.
-2   X[8]
-3   X[7]
```

★ A l'aide de ces fonctions, saisissez les 5 valeurs suivantes à partir de la position 7 de X : { 8, -2, 4, 0, 3 }. Affichez alors ces valeurs dans l'ordre inverse de leur saisie à l'aide de la fonction affiche.

TP 3 : les Structures

1. Structures

Une structure est le regroupement de plusieurs variables pouvant être de types différents sous un même nom, ces variables sont appelées champs.

Pour utiliser une variable structurée, il est préférable de créer un type. La syntaxe est :

```
typedef struct { type champ_1 ; type champ_2; … ; type champ_n ; } nom_du_type ;
```

Pour utiliser le type défini on crée des variables ou des pointeurs.

```
nom_du_type var1, var2, *var3;
```

L'accès aux différents champs des variables structurées se fait en indiquant précisant le champ à l'aide d'un point.

```
var1.champ_1 ou var1.champ_2 etc.
```

S'il s'agit d'un pointeur vers une structure, on accède aux champs à l'aide de l'opérateur -> ou en utilisant des parenthèses :

```
var3->champ_1 ou (*var3).champ_2
```

Il est possible d'affecter des structures entre elles, pour une fonction de retourner une structure.

```
var1 = var2; var2 = *var3; // affectation de structures
var1.champ_1=var2.champ_1; var2.champ_2 = var3->champ_2; // affectation de champs
nom_du_type MaFonction( ... ) ; // prototype d'une fonction retournant une structure
```

2. Exercice

L'objet est de définir les fonctions permettant de réaliser des calculs en complexe : addition, soustraction, multiplication, conjugué, module, division, angle, logarithme.

★ Saisie / affichage de nombres complexes. Dans chaque cas vous déterminerez le type des fonctions écrites ainsi que leurs paramètres d'entrée.
  - ✦ Créez un type de variable structurée comportant deux champs re pour la partie réelle, im pour la partie imaginaire.
  - ✦ Ecrivez une fonction qui affiche un nombre complexe transmis en paramètre à l'écran sous la forme a + ib. Les nombres seront affichés avec 4 décimales au minimum.
  - ✦ Ecrivez une fonction qui réalise la saisie d'un nombre complexe et le retourne. La fonction devra demander la partie réelle, puis la partie imaginaire successivement. Cette fonction a-t-elle besoin d'un paramètre en entrée ?
  - ✦ Testez votre programme en saisissant puis affichant un nombre complexe.

★ Ecrivez dans cet ordre les fonctions réalisant les opérations suivantes. Vous pouvez utiliser les fonctions précédemment écrites. Respectez le nom de chacune des fonctions qui vous est proposé. Déterminez pour chaque fonction le nombre de paramètres et le type de valeur retournée.
  - ✦ Addition : C_add
  - ✦ Soustraction : C_Sous
  - ✦ Multiplication : C_mul
  - ✦ Conjugué : C_conj
  - ✦ Module : C_mod
  - ✦ Division : C_div
  - ✦ Angle : C_angle (regardez l'aide en ligne -fonction man- sur la fonction atan2)
  - ✦ Logarithme Népérien : C_log

Testez vos fonctions avec des exemple connus (cf ci-après)

★ Modifiez la fonction saisie pour qu'elle soit de type **void**. (**A TRAITER IMPERATIVEMENT**)

3. Exemples de résultats

```
Entrez la partie reelle : 2
Entrez la partie imaginaire : 3
Entrez la partie reelle : -1
Entrez la partie imaginaire : 1
```

```
Addition :
   1.0000 + i   4.0000
Soustraction :
   3.0000 + i   2.0000
Multiplication :
  -5.0000 + i  -1.0000
Conjugue :
   2.0000 + i  -3.0000
Module :
  3.6056
Division :
   0.5000 + i  -2.5000
Angle :
  0.9828
Logarithme :
   1.2825 + i   0.9828
```

TP 4 : Lecture de fichiers

1. Rappels de langage

Le programmeur n'accède pas directement aux fichiers, il demande au Système d'Exploitation de s'occuper de toute la gestion technique : pilotage des disques durs etc.

Pour ce faire, il suffit de créer un pointeur de type **`FILE *`** (en majuscules !), puis de demander au SE de créer un lien via ce pointeur. C'est le rôle de la fonction **`fopen`**. La lecture s'effectue à l'aide des fonctions **`fscanf`** et **`fprintf`**.

Lorsqu'on écrit un fichier, le plus simple est de mettre l'écriture dans une boucle **`for`**, en effet on connaît forcément le nombre d'éléments à écrire.

Par contre lors de la lecture, ce n'est pas le cas. Il faut alors être capable de savoir si l'on a atteint la fin du fichier. La fonction **`feof`** donne cette information. Elle s'utilise le plus souvent avec une boucle de type "tant que" : tant que (la fin du fichier n'est pas atteinte)

Enfin, une fois la lecture ou l'écriture finie, il faut fermer le fichier à l'aide de l'instruction **`fclose`**.

La syntaxe exacte des fonctions sus-citées est :

```
fopen( "chemin d'accès au fichier","mode"); // retourne une adresse de type FILE *
```

Le type d'accès est soit **`"r"`** pour la lecture, soit **`"w"`** pour l'écriture. Le chemin d'accès utilise des **`"/"`** même sous l'environnement Windows.

```
fprintf( variable_de_type_FILE, "chaine de format", liste de variables ) ;
fscanf( variable_de_type_FILE, "chaine de format", liste d'adresses de variables ) ;
feof( variable_de_type_FILE ) ;
fclose( variable_de_type_FILE) ;
```

2. Exercice

Dans ce qui suit, toutes les fonctions à écrire seront de type **void**.

Vous trouverez sur le bureau de votre poste de travail un fichier intitulé **`TP4.txt`**, si le fichier n'est pas présent, demandez-le à l'enseignant.

Ce fichier est constitué de deux colonnes **`X`** et **`Y`** (dans cet ordre) de **`n`** réels séparées par un espace.

★ Lisez le contenu du fichier **`TP4.txt`** dans votre programme en faisant un écho (affichage) à l'écran.

- ✦ Ne connaissant pas la longueur du fichier, vous créerez un type de variable pouvant contenir 1000 **`float`** appelé datas.
- ✦ Ecrivez une fonction qui lise le fichier **`TP4.txt`**, et détermine le nombre de points lus. Déterminez les paramètres d'entrée et de sortie de la fonction qui s'appellera **`lecture`**.
- ✦ A l'aide de cette fonction, lisez le contenu du fichier, puis dans la fonction **`main`**, affichez toutes les valeurs lues.

Le signal lu est bruité, afin de diminuer ce bruit, on procédera de la façon suivante : chaque fois qu'un des **`Y`** est supérieur à trois fois la moyenne du vecteur **`Y`**, on fixe la valeur de cet élément à zéro.

★ Filtrez le signal lu.

- ✦ Ecrivez une fonction appelée **`ecriture`** qui crée un fichier de type texte appelé **`filtre.txt`** et y écrive deux colonnes de valeurs, séparées par une tabulation, transmises en paramètres à la fonction. Déterminez les paramètres d'entrée et de sortie de la fonction.
- ✦ Créez une fonction qui calcule la valeur moyenne d'un vecteur. Cette fonction sera appelée **`moyenne`** vous déterminerez les paramètres d'entrée et de sortie.
- ✦ Calculez, dans la fonction principale la valeur moyenne des ordonnées lues (vecteur **`Y`**), puis appliquez l'algorithme décrit ci-dessus.
- ✦ Sauvegardez dans un fichier portant le nom filtre.txt les données ainsi traitées. Le fichier contiendra les vecteurs **`X`** et **`Y`** séparés par un espace.

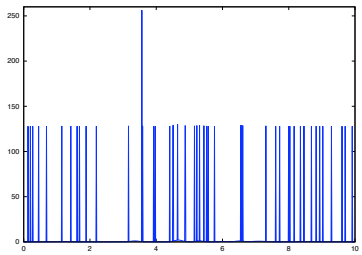
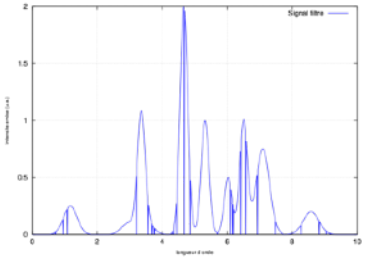
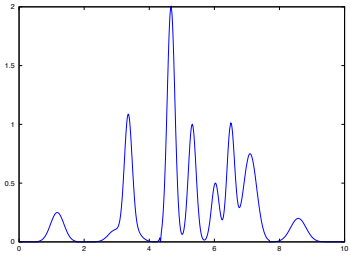
★ Lancez le logiciel Octave, puis dans la fenêtre qui s'ouvre tapez les commandes suivantes :

```
cd Desktop
M = load -ascii filtre.txt ;
```

```
x = M( : , 1 ) ;
y = M( : , 2 ) ;
plot(x,y)
```

Une fenêtre graphique devrait s'ouvrir contenant la courbe des points que vous avez filtré.

★ On peut améliorer le filtrage en modifiant la procédure utilisée en utilisant un filtrage en deux étapes. On procède comme précédemment, si **`Yi`** est supérieur à trois fois la valeur moyenne de **`Y`**, alors on impose **`Yi`** à zéro, en même temps, dans un vecteur auxiliaire **`Z`**, préalablement initialisé à zéro, on impose **`Zi`** à un. Dans une deuxième phase, si **`Zi`** vaut 1 alors on remplace **`Yi`** par la moyenne des deux points environnants.

Signal non filtré	Signal filtré (mise à zéro)	Signal Filtré (deux passes)
		

TP 5 : Dérivées de polynôme

1.Méthode de Horner :

Référez-vous au cours pour plus de détails sur la méthode d'Hörner.
Il est possible de calculer la valeur d'un polynôme et de sa dérivée à l'aide de la méthode suivante dite méthode de Horner.

Coefs de $P_n(x)$	Calcul de $P_n(\alpha)$	Calcul de $P'_n(\alpha)$
a_0	$b_0 = a_0$	$c_0 = b_0$
a_1	$b_1 = b_0\alpha + a_1$	$c_1 = c_0 * \alpha + b_1$
$\vdots$	$\vdots$	$\vdots$
a_i	$b_i = b_{i-1}\alpha + a_i$	$c_i = c_{i-1} * \alpha + b_i$
$\vdots$	$\vdots$	$\vdots$
a_{n-1}	$b_{n-1} = b_{n-2}\alpha + a_{n-1}$	$c_{n-1} = c_{n-2} * \alpha + b_{n-1} = \frac{P'_n(\alpha)}{1!}$
a_n	$b_n = b_{n-1}\alpha + a_n = P_n(\alpha)$	

Pour calculer la dérivée en un point α donné, il suffit d'enchaîner deux schémas de Horner successifs.

2.Travail à effectuer :

Ecrire un programme qui :

- ➡ Effectue la saisie des coefficients d'un polynôme de degré inférieur ou égal à 9 et de son degré. La saisie sera recommencée tant que l'utilisateur n'aura pas saisi un degré compris dans l'intervalle[1 ; 9].
- ➡ Saisisse la valeur d'un point de départ **x_0** ainsi qu'un pas **h** pour calculer la valeur du polynôme et de sa dérivée sur 26 points régulièrement espacés depuis le point de départ **x_0**.
- ➡ Cherche dans les valeurs calculées une racine du polynôme et affiche la valeur de la dérivée en ce point. Dans le cas où il n'y aurait pas de racine le programme affichera le message : "pas de racine trouvée".

Pour ce faire, vous écrirez les fonctions et définitions de type suivantes. Les commentaires C expliquent le rôle de chaque variable.

```
typedef float vect[10];
typedef float pts[26];

void Horner( float *pa , int pn , float pal, float *pP, float *pb);
/* pa pointe vers le vecteur des coefs de P_n(x)
 * pn est le degre de P_n(x)
 * pal le point de calcul
 * pP pointe vers une variable qui contiendra P_n(pal)
 * pb pointe vers un vecteur qui contiendra les b_i
 */

void saisie(float *pa , int *pn);
/* Saisie des coefs d'un polynome P_n(x)
 * pa pointe vers le vecteur qui contiendra les coefs de P_n(x)
 * pn pointe vers une variable qui contiendra le degre P_n(x)
 * La saisie sera refusée tant que n est >9
 */
```

3.Plan de travail

★ Calcul des valeurs prises par le polynôme :

- ✦ Ecrivez la fonction **saisie**. Testez-la en affichant dans la fonction **main** les coefficients d'un polynôme saisi. Vérifiez que vous ne pouvez entrer un degré > 9, ni < 1.
- ✦ Ecrivez la fonction **Horner**. Testez-la avec le polynôme suivant en α = 2 : $P_n(x) = x^2 + x - 3$

Vous devez trouver :

```
Coefs de P₂(x) calcul de P₂(2)
     1              b₀ = 1
     1              b₁ = 1*2+1 = 3
    -3              b₂ = 3*2 -3 = 3 = P₂(2)
```

- ✦ Ecrivez le code nécessaire pour réaliser la saisie de **x_0** le point de départ et de **h** le pas. Ces saisies se feront dans la fonction **main**.
- ✦ À partir de **x_0** et en utilisant le pas **h**, calculez 26 points équidistants. À l'aide de la fonction **Horner**, calculez les valeurs prises par le polynôme saisi avec la fonction **saisie** et calculez la valeur de la dérivée en ces points. Vous afficherez le résultat des calculs sous la forme de colonnes contenant les valeurs correspondant aux intitulés suivants :

```
n° du point : xi    Pn(xi)     P'n(xi)
```

Exemple avec $P_3(x) = x^3 + x^2 - 10x + 8$, point de départ $x_0 = -1$ pas : h = 0,1

```
0 :   -1.000     18.000          -9.000
1 :   -0.900     17.081          -9.370
2 :   -0.800     16.128          -9.680
3 :   -0.700     15.147          -9.930
…
```

★ Dans la fonction **main** écrivez le code nécessaire pour trouver les points dont la valeur absolue[1] de l'ordonnée est inférieure à 10^{-5}. Affichez alors le message suivant :

```
"Racine au point <indice du point> soit x= <abscisse correspondante>
 et P'(x) = <valeur de la dérivée>"
```

Exemple d'exécution du programme avec le même polynôme que ci-dessus :

```
Saisie des coefs du polynome :

Entrez le degre de P_n(x) : 3
Entrez le coef de x^3 : 1
Entrez le coef de x^2 : 1
Entrez le coef de x^1 : -10
Entrez le coef de x^0 : 8

Entrez le point de depart :-1
Entrez le pas :0.1

 0 :   -1.000     18.000          -9.000
 1 :   -0.900     17.081          -9.370
 2 :   -0.800     16.128          -9.680
 3 :   -0.700     15.147          -9.930
 4 :   -0.600     14.144         -10.120
 5 :   -0.500     13.125         -10.250
 6 :   -0.400     12.096         -10.320
 7 :   -0.300     11.063         -10.330
 8 :   -0.200     10.032         -10.280
 9 :   -0.100      9.009         -10.170
10 :    0.000      8.000         -10.000
11 :    0.100      7.011          -9.770
12 :    0.200      6.048          -9.480
13 :    0.300      5.117          -9.130
14 :    0.400      4.224          -8.720
15 :    0.500      3.375          -8.250
16 :    0.600      2.576          -7.720
17 :    0.700      1.833          -7.130
18 :    0.800      1.152          -6.480
19 :    0.900      0.539          -5.770
20 :    1.000      0.000          -5.000
21 :    1.100     -0.459          -4.170
22 :    1.200     -0.832          -3.280
23 :    1.300     -1.113          -2.330
24 :    1.400     -1.296          -1.320
25 :    1.500     -1.375          -0.250
Racine au point 20 soit x=1.000000
et P'(x) = -5.000000
```

[1] La fonction pour calculer la valeur absolue d'un **float** est **fabs**. Son prototype est dans **math.h**