Université Moulay Ismail
Faculté des Sciences
Département de Physique
Meknès

Cours d'Initiation à Maple et à la Simulation Numérique en Physique

Filière SMP
Semestre 6

Année Universitaire
2020-2021

I. Essaoudi / M. Sabbane,
Département de Physique

Initiation à Maple et à la Simulation Numérique en Physique

Chapitre Introductif : Rappel sur l'algorithmique

I. Présentation générale

Un algorithme est une succession d'étapes qui interviennent dans la résolution d'un problème quelconque en utilisant un programme informatique.

La mise en pratique de l'algorithme passe par l'écriture d'un programme, un fichier texte, en utilisant un langage de programmation. Le programme est compilé pour donner une commande exécutable par le système d'exploitation de l'ordinateur.

En calcul scientifique, ces notions sont très répandues et l'exploitation de l'ordinateur exige une bonne maîtrise de ces techniques. Généralement, les algorithmes sont conçus par des spécialistes. Pour les exploiter, il suffit de les faire implémenter en des programmes exécutables par la machine. Dans la vie courante, un algorithme peut par exemple prendre des formes très variées comme :

- d'une recette,
- d'un mode d'emploi,
- d'une fiche technique de montage,
- d'une composition musicale,
- d'un texte de loi,
- d'un itinéraire routier.

N.B : un algorithme est le plan d'un programme informatique !

II. Structure d'un algorithme

L'algorithme est une succession d'instruction élémentaire organisée et ordonnée. Il s'écrit sous forme d'un texte contenant des lettres et des symboles. Le texte final forme un plan préliminaire pour écrire un programme informatique.

```
Algorithme NomAlgorithme
Début
  ... action_1
  ... action_2

  ... action_n
Fin
```

La première ligne donne le nom de l'algorithme. Le corps de l'algorithme est délimité par les mots clés « Début » et « Fin ». Les lignes désignés par « … action_i » sont des instructions, elles désignent des lignes de traitement. Les données d'entrée de l'action « … action_i » sont toutes les données qui la précèdent. Alors que son résultat sera considéré comme entrée aux actions suivantes.

En général, une ligne d'action peut s'écrire sur plusieurs lignes de texte. Ainsi, pour délimiter les instructions, il est très utile d'utiliser un symbole convenable à la fin de chaque instruction. Pour la majorité des langages de programmation, ce symbole est « ; ». Donc, il serait très commode de l'utiliser comme fin d'instruction en algorithmique.

Le vocabulaire utilisé pour élaborer un algorithme n'est pas nécessairement standard. Toutefois, l'enchaînement des instructions est indispensable. Par exemple, on peut utiliser les mots suivant selon leur usage :

- Lire : saisir des données
- Ecrire : afficher des données
- Renvoyer : renvoyer un résultat
- Si / Si Non : les sélections
- Pour : un traitement itératif
- Tant que : un traitement répété
- Répète : un traitement répété

Éventuellement, les noms des fonctions mathématiques usuelles sont utilisés alors que les noms des variables sont laissés au libre choix de l'auteur de l'algorithme. Néanmoins, il est très utile de se limiter à des noms de variables claires et faciles à retenir pour ne pas se perdre au moment de l'implémentation de l'algorithme.

Les instructions sont une combinaison de mots clés et d'opérateurs. Les opérateurs sont les opérateurs arithmétiques (+, -, *, /), logiques (AND, OR, NOT) (ET, OU, NON), les opérateurs relationnels (<, <=, >, >=, =, <>) plus l'opérateur d'affection. L'opérateur d'affectation est souvent différent de celui de la comparaison, il est noté en algorithmique par le symbole « ← »

Exemple : Elaborer un algorithme pour calculer la somme de deux nombres réels.

```
Algorithme SommeRelle
Début
     Somme, X, Y sont des réels ;
     Lire X ;
     Lire Y ;
     Somme ← X + Y ;
     Ecrire(Somme) ;
Fin
```

III. Les Variables

Une variable est constituée d'un nom et d'un type de contenu. Les variables peuvent être simples ou composées. Les différents types de variables simples, utilisées en informatique, sont : booléens, caractères, chaînes de caractères, nombres entiers, nombres réels à précision simple, nombres réels à haute précision, etc. Les variables composées sont de deux sortes :

- Les structures qui regroupent des données simples de types différents ;
- Les tableaux qui regroupent des données simples de même type.

Les langages actuels de programmation acceptent la déclaration des variables au fur et mesure du développement du programme. Pour souci de clarté, il est très intéressant de déclarer les variables utilisées au début de l'algorithme. Quand l'algorithme sera traduit en programme cette déclaration aura d'autres avantages : réservation de l'espace mémoire correspondant au type, possibilité de vérifier le programme du point de vue de la cohérence des types, etc.

Les variables utilisées par un algorithme sont de deux types : Les variables intrinsèques, internes à l'algorithme, ou les variables extrinsèques, externes à l'algorithme. Les variables internes sont déclarées dans le corps de l'algorithme et ne sont utilisées qu'en local. Les variables externes sont des données d'entrée pour l'algorithme. Elles sont passées comme paramètres à l'algorithme si non elles sont d'ordre général et sont communes à tous les algorithmes. Pour illustrer cette description, voyons l'exemple suivant :

```
Algorithme Factoriel( n est entier)
I, F sont des entiers,
Début
     F ← 1 ;
     Pour I ← 1 à n
          F ← F * I ;
     Fin Pour
     Renvoyer (F) ;
Fin
```

L'algorithme ci-dessus calcul le factoriel d'un nombre entier « n ». Nous distinguons la variable externe « n » passée comme paramètre d'entrée. Les variables internes, I et F, sont déclarées et utilisées dans le corps de l'algorithme. Le résultat, la valeur de la variable F, est renvoyée comme sortie de l'algorithme.

IV. Traitement sélectif et Instructions conditionnelles

L'instruction de la sélection est une structure très importante dans le traitement algorithmique. Elle permet d'orienter le développement de l'algorithme selon les conditions prédéfinies ou obtenues comme résultats. Les suites d'instructions, « Si »… « Alors » … « Sinon » … « Fin Si », permettent de conditionner l'exécution de certaines instructions. L'implémentation de ces instructions se fait selon le besoin

IV.1. Sélection simple

```
Si condition alors
      instruction;       // Exécuter dans le seul cas où la condition est
vrai
Fin Si
```

Exemple :

```
Algorithme Inverse
x, y sont des réels ;
Début
      Lire x ;
      Si x<>0 alors
            y ← 1/x ;
      Fin si ;
      Ecrire(y) ;
Fin
```

IV.2. Sélection double

```
Si condition alors
      instruction 1;  // l'instruction à exécuter si la condition est
Sinon
      instruction 2; vrai  // l'instruction à exécuter si la condition est

FinSi                 fausse.
```

Exemple :

```
Algorithme Inverse
x, y sont des réels ;
Début
      Lire x ;
      Si x<>0 alors
            y ← 1/x ;
            Ecrire(y) ;
      Sinon
            Ecrire(" Nombre nom inversible ") ;
      Finsi ;
Fin
```

IV.3. Sélection multiple

```
Si condition_1 alors
       instruction 1; // l'instruction à exécuter si la condition_1 est vrai
 Sinon Si condition_2
       instruction 2; // l'instruction à exécuter si la condition_2 est vrai.
 Sinon Si condition_3
      instruction 3;  // l'instruction à exécuter la condition_3 est vrai.
.
.
.
Sinon condition_n
      instruction n;  // l'instruction à exécuter lors du dernier cas.
Fin Si
```

Exemple :

```
Algorithme EtatPhysiqueEau
Temperature est un réel
Début
      Si Temperature <0
            Ecrire("Etat solide") ;
      Sinon Si Temperature = 0
           Ecrire("Transition de phase solide - liquide") ;
      Sinon Si Temperature > 0 ET Temperature <100
            Ecrire("Etat liquide ") ;
      Sinon Si Temperature = 100
           Ecrire("Transition de phase liquide - gaz") ; Si Non
           Ecrire("Etat gazeux") ;
      FIN SI

Fin
```

V. Traitement répétitif et les boucles

Le rôle des boucles est de répéter l'exécution d'un bloc d'instructions plusieurs fois. Le nombre de répétition peut être défini comme un nombre entier ou laissé indéfini lorsque l'arrêt de la répétition est conditionné par la réalisation d'un événement. Reprenons l'exemple de la section (III) pour calculer le factoriel d'un nombre entier. La boucle « Pour … Fin Pour » définie comme suit :

```
Pour I ← 1 à n Faire
          F ← F * I ;
Fin Pour
```

exécute l'instruction « $F \leftarrow F^*I$ » pour toutes les valeurs de I = 1, 2, …, n.

En général, il existe trois types de boucles

V.1. La boucle « Pour … Fin Pour »

Cette boucle itérative utilise un compteur qui parcourt une intervalle de nombre [Borne_1, Borne_2] selon le pas d'avancement choisi. Normalement il y a pas d'ordre précis entre les deux bornes seulement il faut les harmoniser avec le pas d'avancement pour que la boucle s'arrête si non elle continuera à l'infini.

Exemple

```
Pour i←1 à 10 par 1 Faire
           Ecrire(i) ;
Fin Pour
```

```
Pour i←100 à 20 par -2 Faire
           Ecrire(i) ;
Fin Pour
```

V.2. La boucle « Tant que … Fin »

La boucle « Pour » utilise un compteur pour passer d'une étape à l'autre. Or, ça se peut que le nombre d'itérations soit terminé sans que le résultat soit atteint. Par exemple, si nous voulons calculer la valeur d'une série qui atteint sa valeur maximale en un certain rang inconnu, l'utilisation d'un compteur ne serait pas la solution convenable. Prenons le cas de la série suivante :

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots, \qquad -\infty < x < \infty$$

Pour une valeur précise de x, la valeur de e^x arrive à sa grandeur convenable lorsqu'elle ne varie que très lentement où lorsqu'elle dépasse la précision de la machine. Cependant, la condition d'arrêt de la boucle sera conditionnée par la variation de la somme d'une itération à une l'autre.

```
Tant que condition Faire
     Instructions ;
Fin Tant que
```

N.B : ***la boucle « Tant que … Fin » est très utile en programmation quand le déroulement du programme n'est pas connu à l'avance.***

V.3. La boucle « Répéter … Jusqu'à »

Cette boucle ne diffère pas beaucoup de la précédente. Toutefois, elle présente une particularité : c'est que le contenu de la boucle s'exécute au moins une fois, c-à-d, même si la condition d'arrêt est vérifiée.

```
Répéter
     Instructions ;
Jusqu'à condition   //la réalisation d'un événement
```

La réalisation de l'événement d'arrêt est attendue au moment de l'exécution du programme. Comme par exemple, une boucle infinie qui attend la pression d'un bouton au clavier ou clic par la sourie.

V.4. Comparaison entre les boucles

Lorsque on sait exactement combien de fois on doit itérer un traitement, c'est l'utilisation de la boucle « Pour .. fin » qui doit être privilégiée. Par exemple,

```
Algorithme AfficherLesChiffres
Variable i est un  entier
Début
  Pour i ← 0 à 9 faire
            Ecrire(i)
```

```
  Fin pour
Fin
```

La comparaison des boucles pour un problème simple n'est pas très convaincante. Toutefois, le problème s'accentue lorsque le nombre d'itération ou la condition d'arrêt n'est pas simple. Comme il a été évoqué, le calcul de la valeur d'une série peut se faire en quelques itérations si les termes de la série tendent rapidement vers 0. Dans d'autres situations, des séries ne convergent que lentement. Cependant, le nombre d'itérations restera variable en fonction du besoin. Prenons les deux exemples suivants :

```
Algorithme BouclePour
Variable i est un entier ;
Début
   Pour i ← 1 à 100 faire
    Ecrire(i) ;
  Fin Pour
Fin
```

```
Algorithme BoucleTantQue
Variable i est un entier
Début
  i ← 1 ;
  Tant que (i ≤ 100) faire
    Ecrire(i) ;
i ← i+1 ;
  Fin tant que
Fin
```

```
Algorithme BoucleRepeter
Variable i est un entier
Début
  i ← 1 ;
  Répéter
    Ecrire(i) ;
i ← i+1 ;
  Jusqu'à (i > 100)
Fin
```

```
Algorithme Recursif (n : entier)
Début
  Si (n ≤ 100)
  Alors
    Ecrire(n) ;
Recursif(n+1) ;
  Fin si
Fin
```

VI. Traitement des tableaux

Les tableaux sont des variables structurées. Ils constituent une liste d'éléments du même type. La déclaration se fait par :

`Nom_Tableau est un tableau dimension de type de variable.`

Soit un algorithme qui permet de saisir des réels dans un tableau à une dimension et de les afficher.

```
Algorithme TraitementTableau
T est tableau de 30 réels ;
Variable i  est un entier ;
Début
  Pour i ← 1 à 30 faire
    Lire(T[i]) ;
  Fin Pour
  Pour i ← 1 à 30 faire
      Ecrire(T[i]) ;
  Fin Pour
```

```
Fin
```

Soit un algorithme qui permet de saisir des réels dans un tableau à deux dimensions (le cas d'une matrice) et de les afficher.

```
Algorithme TraitementTableau
T est tableau de 30x40 réels ;
Variable i,j  sont des entiers ;
Début
  Pour i ← 1 à 30 faire
    Pour j ← 1 à 40 faire
            Lire(T[i,j]) ; Fin
Pour
  Fin Pour
  Pour i ← 1 à 30 faire
    Pour j ← 1 à 40 faire
            Ecrire(T[i,j]) ; Fin
Pour
  Fin Pour

Fin
```

Au fait, les tableaux sont des structures de données très importantes en calcul scientifique. Ils peuvent représenter des vecteurs, des matrices des listes etc. Il existe plusieurs algorithmes conçus spécialement pour traiter les tableaux. Les plus utilisés seront les algorithmes de trie, de classification et de recherche.

Partie I : Initiation à Maple

Chapitre I : Initiation à Maple V

I.1 Introduction

Maple est un logiciel propriétaire de calcul formel développé depuis les **années 1980** et aujourd'hui édité par la société canadienne **Maplesoft**. La dernière version est la **version 2019**. Les objets de base du calcul sont les expressions mathématiques.

Maple fournit un langage de programmation spécifique interprété et interactif. L'utilisateur peut l'exploiter en mode lignes de commandes ou mode exécution de programmes écrit en plusieurs lignes de code.

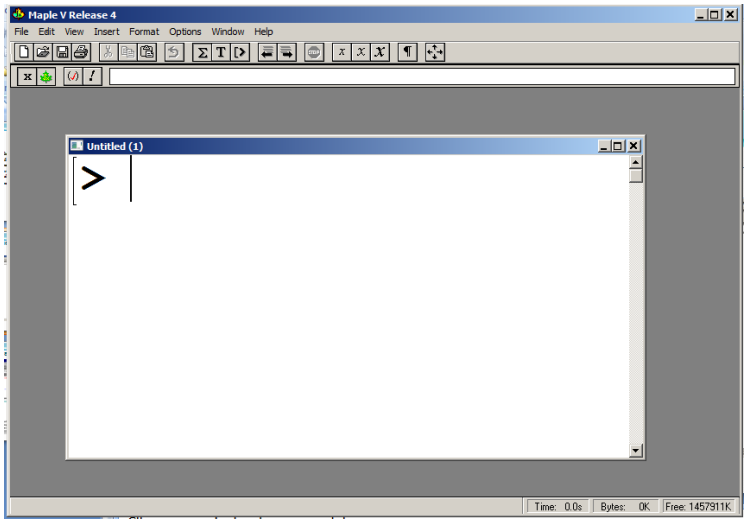
Maple fourni un environnement de calcul complet et polyvalent. Son langage intègre à la fois du calcul numérique comme du calcul formel ou symbolique. Il est également doté d'importantes bibliothèques et couvre toutes les fonctions mathématiques usuelles. Enfin, sa codification des nombres assure une très bonne précision numérique des résultats.

Maple est utilisé dans un nombre remarquable d'applications des sciences et des mathématiques allant de la démonstration du dernier théorème de Fermat en théorie des nombres, à des problèmes de la relativité générale et la mécanique quantique [1].

I.2 Fonctionnalités de Maple

I.2.1 Environnement

L'environnement de travail de Maple est similaire à la figure suivante:
Elle est formée essentiellement d'une fenêtre principale ornée par un menu et d'une barre d'outils.



[1] https://fr.wikipedia.org/wiki/Maple

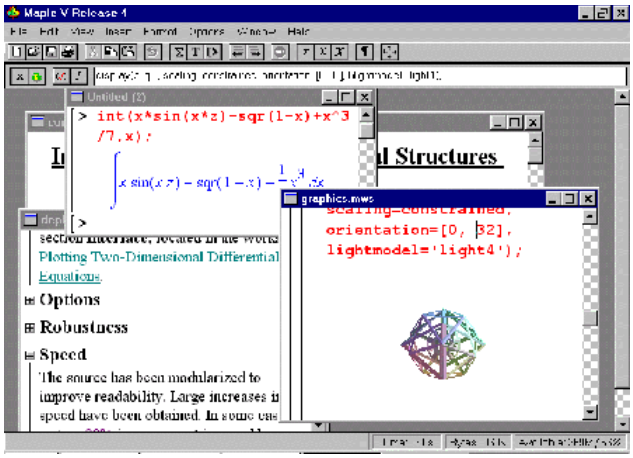


Figure 1 : Fenêtre type de Maple V version 4

L'utilisateur exploite l'environnement Maple via des sessions de travail. Chaque session est représentée par l'ouverture d'une fenêtre. Cependant, l'utilisateur peut avoir plusieurs sessions ouvertes à la fois. Une session peut être sauvegardée dans un fichier indépendant.

I.2.2 Menu standard de l'espace de travail

Ces éléments de menu sont disponibles lorsque le curseur se trouve dans une région standard d'une feuille de travail. Ces menus regroupent des commandes des tâches similaires. Les éléments de menu grisés correspondent à des commandes qui ne sont pas disponibles pour la situation actuelle. Comme tout autre logiciel, la barre d'outils représente les raccourcis aux commandes les plus utilisées.

Un bref aperçu du menu de la fenêtre principale nous permet de distinguer les options suivantes :

File : Il contient l'ensemble de commandes disponibles dites d'entrées / sorties pour gérer les fichiers ainsi que la commande de sortie Exit.

Edit : Il contient les commandes de l'édition : copier / coller / rechercher …

View : Il contient les commandes relatives à l'affichage.

Insert : Menu des commandes d'insertion, textes, formules etc.

Format : Ce menu contient des commandes pour formater le texte à un type et apparence souhaitée.

Options : des choix pour définir les actions effectuées lorsque Maple exécute une commande.

Window : le menu d'arrangement des fenêtres de travail (sessions)

Help : Contient les différents textes d'aide de l'ensemble de l'environnement Maple. Il est possible d'accéder à l'aide contextuel. Il suffit de placer le curseur sur le mot en question puis taper F1.

I.2.3 Feuille de travail.

Une feuille de travail typique se présente sous la forme suivante (figure 2). C'est une fenêtre ordinaire avec une barre de titre, les boutons systèmes et un espace de travail dedans.

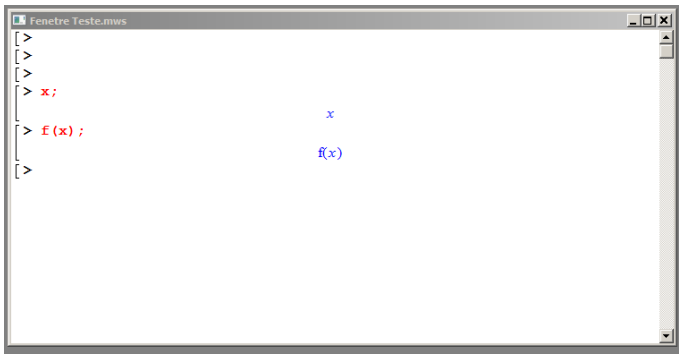


Figure 2 : Feuille de travail Maple

La zone accessible de cette feuille de travail commence là où le symbole suivant (figure 3) apparait.

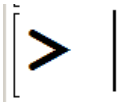


Figure 3 : Début d'une ligne de commande

A ce niveau, l'utilisateur peut taper ses commandes, ses expressions mathématiques valables, les terminer par les symboles « ; » ou « : » puis Entrée. Maple exécute la commande, si l'expression est valable, il l'exécute et retourne le résultat convenable si non, il émet un message d'erreur. ***Remarque : si la ligne de commande est terminée par :***

- « ; » alors le résultat sera affiché juste après la ligne ;
- « : » alors le résultat sera calculé mais il ne sera pas affiché.

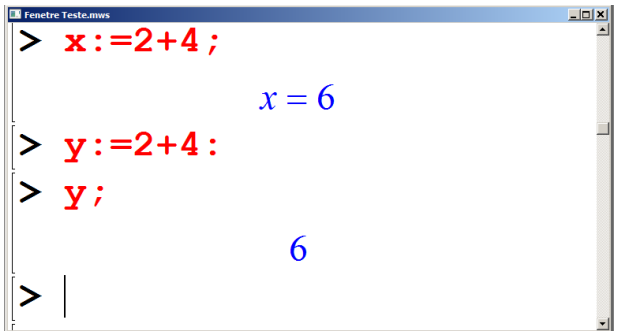


Figure 4 : Exemple de ligne de commande Maple

L'environnement Maple fourni un langage de programmation interprété qui reconnait la majorité des fonctions et opérateurs mathématiques habituels.

III. Le langage Maple

III.1 Les opérateurs

Les opérateurs sous Maple se divisent en deux catégories. Ceux qui sont standard et semblable à d'autres langages, tels que les opérateurs arithmétiques. Outre les opérateurs usuels +, -, *, / , Maple est enrichi par des opérateurs très utiles tels que :

- **!** la factorielle (exp 6!)
- ** ou ^ la puissance (x^y)
- **iquo(n,m)** le quotient de la division entière entre deux nombres
- **irem(n,m)** le reste de la division entière entre n et m

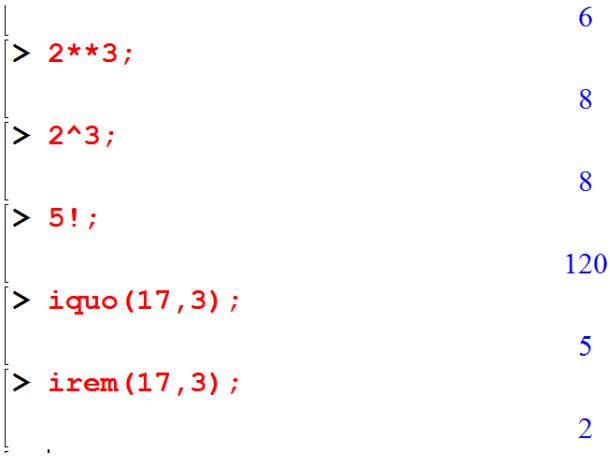


Figure 4 : exemple d'opérateurs spécifiques

III.2 Les fonctions élémentaires

Maple reconnait aussi les fonctions élémentaires classiques :

```
exp(x), log(x) ou ln(x), log10(x), log[b](x)

round(x), trunc(x), frac(x), sqrt(x), abs(x)

sin(x), cos(x), tan(x), cot(x)

sinh(x), cosh(x), tanh(x), cotanh(x)

arcsin(x), arccos(x), arctan(x), arccotan(x)

arcsinh(x), arccosh(x), arctanh(x), arccotanh(x)
```

Des constantes sont prédéfinies sous Maple comme le réel Pi, l'Imaginaire I, l'infini, la constante d'Euler. Dans la notation Maple, les lettres minuscules et majuscules sont différentes.

Chapitre II : Utiliser Maple V

L'apprentissage de Maple se fait naturellement en l'utilisant pas à pas. Son environnement offre tout ce qu'il faut pour pouvoir l'utiliser même sans avoir une expérience.

I. Les variables sous MAPLE

Une variable dans un environnement de programmation réfère à un emplacement dans la mémoire de l'ordinateur qui peut recevoir les données de l'utilisateur. Cependant, l'utilisateur doit déclarer ses variables pour réserver la mémoire avant leur utilisation. La majorité des langages de programmation préfèrent la spécification de la nature des données à stocker avant de leur allouer l'emplacement mémoire. Pour Maple, la déclaration d'une variable ne nécessite pas la spécification de sa nature. Toutefois, il respecte les traditions comme par exemple :

- Le nom d'une variable est une chaine alphanumérique qui :
  - Commence nécessairement par une lettre alphabétique,
  - Ne comprend pas de caractères spéciaux ni des opérateurs,
- Les caractères en Majuscule diffèrent de ceux en minuscules dans les noms des variables,
- Les variables peuvent être déclarées n'importe où dans la session de travail

Exemple : a, A, Ab, X1, x1, _a sont des noms de variables correctes

```
> a, A, Ab, X1, x1, _a ;
```

$$a, A, Ab, X1, x1, \_a$$

```
> _b:=3;
```

$$\_b := 3$$

```
> T[1];
```

$$T_1$$

```
> 3+sin(a/4);
```

$$3 + \sin\left(\frac{1}{4}a\right)$$

Les variables peuvent être utilisées dans des expressions sans qu'elles ne soient initialisées. La variable est dite assignée ou non assignée lorsqu'on affecte ou non une valeur à la variable. La variable assignée garde sa valeur tant qu'elle n'est pas réassignée.

La fonction **restart** permet de réinitialiser toutes les variables de la feuille de calcul en cours. Cette fonction réinitialise toutes les variables. Même celles qui sont déclarées ou assignées avant le déclanchement de restart sont remis à l'état initial.

Exemple :

```
> alpha:=4;
```

$$\alpha := 4$$

La valeur 4 est assignée à la variable alpha.

```
> beta, delta, T[1];
```

$$\beta, \delta, T_1$$

Les variables beta, delta et T[1] sont déclarées sans qu'elles soient assignées.

N.B : L'affectation comme « a=2 » par exemple n'est pas correcte sous Maple.

```
> a=2;
```

$$a = 2$$

```
> a;
```

$$a$$

```
> assign(a,2);
> a;
```

$$2$$

Une autre manière d'affecter une valeur à une variable est d'utiliser la fonction assign.
Pour désaffecter une variable assignée, il suffit de lui affecter son nom écrit entre deux apostrophes.

```
> a:='a';
```

$$a := a$$

```
> a;
```

$$a$$

II. Les constantes prédéfinies

Comme tout autre langage de programmation, Maple dispose d'un certain nombre de constantes prédéfinies :

- **true , false** : valeurs booléennes vrai , faux ,
- **I** : nombre complexe de carré -1 ,
- **Pi** : nombre Pi ,
- **infinity** : symbole plus l'infini.

Pour la base du logarithme népérien, **e**, on utilise plutôt la fonction **exp(1).**

III. Les expressions de Maple

Les expressions sont des chaines de caractères, respectant la syntaxe Maple, pour identifier une situation. Par exemple, définir une fonction, une expression mathématique, une équation, une série etc…
L'expression ainsi composée est soit insérée directement dans d'autres expressions, assignée à une variable, évaluée ou imprimée seulement.
Exemple d'une expression composée non assignée et utilisée dans l'expression qui la suit.

```
> 1+2/a+3/(1+cos(x));
```

$$1 + \frac{2}{a} + \frac{3}{1+\cos(x)}$$

```
> simplify(");
```

$$\frac{4a + a\cos(x) + 2 + 2\cos(x)}{a(1+\cos(x))}$$

*N.B : la fonction **simplify()** permet de simplifier l'expression en argument.*
Exemple d'expression composée et assignée en même temps à une variable.

```
> A:=1+2/a+3/(1+cos(x));
```

$$A := 1 + \frac{2}{a} + \frac{3}{1+\cos(x)}$$

```
> simplify(A);
```

$$\frac{4\,a + a\cos(x) + 2 + 2\cos(x)}{a\,(1+\cos(x))}$$

Exemple d'expression d'une équation définie puis assignée à une variable.

```
> a*x^2+b*x+c=0;
```

$$a\,x^2 + b\,x + c = 0$$

```
> equ:=";
```

$$equ := a\,x^2 + b\,x + c = 0$$

N.B : Généralement, la composition d'une expression et son assignement ou son évaluation par une fonction se font en même temps, c-à-d, sur la même ligne.

```
> eq:=a*x^2+b*x+c=0;
```

$$eq := a\,x^2 + b\,x + c = 0$$

IV. Les Mathématiques de base sous Maple

Maple est un environnement de travail assez vaste. Au niveau d'initiation, il est plus commode de l'explorer pas à pas. Son exploitation par les débutants vaut mieux qu'elle débute par se familier avec des instructions bien connues en terme de fonctionnalités.
Par exemple, la somme, le produit, la dérivée, l'intégration, la résolution des équations, le traçage des graphes etc.
Les fonctions de base pour exploiter Maple, sont réparties en classe :

Fonctions mathématiques de base : Ce sont des fonctions mathématiques que Maple reconnaît. Elles sont très nombreuses. Toutefois, l'utilisateur peut les chercher sur l'aide. Généralement, les plus utilisée sont celles présentées dans la section suivante.

Calcul : Cette classe regroupe les fonctions destinées au calcul analytique : dériver, intégrer, évaluer une limite, développer en série, résoudre des équations différentielles, etc.

Algèbre linéaire. Maple fourni les différents outils mathématiques pour l'algèbre linéaire : addition, multiplication, inversion de matrices, produits scalaire et vectoriel, déterminant, transposée, résolution de systèmes, valeurs propres, base, espace des colonnes, orthogonalisation, …

IV.1 Les Fonctions mathématiques de base

Les fonctions mathématiques de base couvrent les fonctions habituelles : exponentielles, logarithmes et trigonométriques.
sqrt(x): la fonction racine carrée
sum(x[i],i=a..b) : la somme d'une série de valeurs d'indice i
product(y[j],j=s..r) : le produit d'une séries de valeurs d'indice j.

floor(x) : la partie entière inférieure de x,
ceil(x) : la partie entière superieure de x ,
abs(x) : la valeur absolue de x.
Exemple :

```
> sum(x[i],i=a..b);
```

$$\sum_{i=a}^{b} x_i$$

```
> product(y[j],j=s..r);
```

$$\prod_{j=s}^{r} y_j$$

Les fonctions trigonométriques :

La fonction	Son inverse	Sa signification
cos()	arccos()	cosinus
sin()	arcsin()	Sinus
tan()	arctan()	tangente
cot()	arccot()	cotangente
sec()	arcsec()	sécante (=1/cos())
csc()	arccsc()	cosécante (=1/sin())

Les fonctions hyperboliques

La fonction	Son inverse	Sa signification
cosh()	arccosh()	cosinus
sinh()	arcsinh()	Sinus
tanh()	arctanh()	tangente
coth()	arccoth()	cotangente
csch()	arccsch()	cosécante (=1/sinh())
sech()	arcsech()	Sécante (=1/cosh())

Les fonctions Logarithmes et exponentielles

Le logarithme est décrit avec 3 fonctions sous Maple :

- ln(x) : logarithme naturel,
- log(x) : logarithme qui peut décrire des logarithmes à des bases variables,
- log10(x) : logarithme décimale (active à la suite du chargement de sa bibliothèque (readlib(log10)

La fonction exponentielle est donnée par : exp(x)
Exemples :

```
> ln(2.5);
```

$$.9162907319$$

```
> log[2](2.5);
```

$$1.321928095$$

```
> log[10](10^4);
```

$$\frac{\ln(10000)}{\ln(10)}$$

```
> simplify(");
```

$$4$$

N.B : Maple fourni une fonction de conversion : ***convert(expression, forme)*** pour convertir des expressions ou des nombres en des formats spécifiques :

```
> convert(Pi/2,degrees);
```

$$90\ degrees$$

```
> convert(cos(x),exp);
```

$$\frac{1}{2}e^{(I\,x)}+\frac{1}{2}\frac{1}{e^{(I\,x)}}$$

```
> convert(90*degrees,radians);
```

$$\frac{1}{2}\pi$$

IV.2 Les Fonctions du calcul analytique

IV.2.1- Définition d'une fonction

Une fonction mathématique à une ou plusieurs variables est définie sous Maple comme suit :

nom_de_la_fonction := (variable1, variable2, …) -> contenu (expression mathématique);

Le symbole « -> » imite la flèche utilisée généralement en mathématiques.

Exemple :

```
> f:=x->x*exp(1-x^2);
```

$$f := x \to x\, e^{(1-x^2)}$$

```
> g:=(x,y)->sin(x)*cos(y)*exp(1/(x^2+y^2));
```

$$g := (x, y) \to \sin(x)\cos(y)\, e^{\left(\frac{1}{x^2+y^2}\right)}$$

```
> evalf(g(2,1));
```

$$.6000696743$$

Dans l'exemple ci-dessus les fonctions **f(x)** et **g(x,y)** sont définies pour des variables réelles quelconques x et y.

IV.2.2- Etude de la continuité des fonctions réelles

- **discont(f(x))** : trouve les valeurs de x pour lesquelles f(x) n'est pas continue,

- **iscont(f(x),x=a..b)** : Examine la continuité de f(x) pour a<=x <=b et renvoie « **true** » si la fonction est continue ou « ***false*** » si f(x) est discontinue.

```
> readlib(iscont):
> f(x):=1/(x^2-1);
```

$$f(x) := \frac{1}{x^2 - 1}$$

```
> discont(f(x),x);
```

$$\{-1, 1\}$$

```
> iscont(f(x),x=0...100);
```

$$false$$

IV.2.3 Calcul des limites des fonctions réelles

- **limit(f(x),x=b)** : calcule la valeur de la limite de f(x) quand x->b.(la borne b peut être infinity pour représenter la borne infini)
- **limit(f(x),x=b,right)** : calcule la limite du coté droit de la borne b ,
- **limit(f(x),x=b,left)** : Calcule la limite du coté gauche de la borne b,
- **Limit(f(x),x=b)** : permet de donner l'expression de la limite calculée.

Exemples : la fonction f(x) est la même que celle prise dans l'exemple précédent.

```
> limit(f(x), x=1);
```

$$undefined$$

```
> limit(f(x),x=infinity);
```

$$0$$

```
> limit(f(x), x=1, right);
```

$$\infty$$

```
> limit(f(x), x=1, left);
```

$$-\infty$$

IV.2.4 Calcul des dérivées des fonctions réelles

La dérivée première et seconde d'une fonction à une seule variable f, dérivable n fois, est donnée par l'opérateur suivant :

- **D(f)** calcule la dérivée première de f,
- **(D@@*n*)(f)** calcule la dérivée $n^{ième}$ de f ;

```
> f:=x->sin(x);
```

$$f := \sin$$

```
> D(f);
```

$$\cos$$

```
> (D@@1)(f);
```

$$\cos$$

```
> (D@@2)(f);
```

$$-\sin$$

Le même opérateur peut calculer les dérivées partielles d'une fonction $g(x_1, x_2, \ldots\ldots x_n)$ selon l'expression suivante :

- **D[*i*](*g*) :** calcule la dérivée partielle suivant la composante *i* de *g*--

- **D[*i*,*j*](*g*)** (équivalent à **D[*i*](D[*j*](*g*))**) calcule les dérivées partielles par rapport à la variable x_i et x_j successivement.

```
> g:=(x,y,z)->sin(x)*cos(y)+x*y*exp(2*z);
```

$$g := (x, y, z) \to \sin(x)\cos(y) + x\,y\,\mathbf{e}^{(2z)}$$

```
> D[1](g);
```

$$(x, y, z) \to \cos(x)\cos(y) + y\,\mathbf{e}^{(2z)}$$

```
> D[2](g);
```

$$(x, y, z) \to -\sin(x)\sin(y) + x\,\mathbf{e}^{(2z)}$$

```
> D[3](g);
```

$$(x, y, z) \to 2\,x\,y\,\mathbf{e}^{(2z)}$$

Maple peut également calculer la dérivée (ou les dérivées partielles) d'une fonction, avec l'instruction **diff().** La syntaxe de cette instruction s'écrit comme suit :

diff(nom_de_la_fonction, variable1, variable2 ...);

Exemple :

```
> diff(g(x,y,z),x,y);
```

$$-\cos(x)\sin(y) + \mathbf{e}^{(2z)}$$

IV.2.5 Intégration des fonctions réelles

La fonction **int()** calcule l'intégrale définie ou indéfinie d'une expression algébrique f par rapport à une ou plusieurs variables. La syntaxe de l'instruction **int()** s'écrit comme suit :

int(f(x),x) : une intégrale indéfinie, le résultat est la primitive de la fonction f(x)

int(f(x) ,x=a..b) : une intégrale définie, le résultat est une valeur réelle si elle existe.

Exemple :

```
> h:=x->1+2*x+1/x;
```

$$h := x \to 1 + 2\,x + \frac{1}{x}$$

```
> int(h(x),x);
```

$$x + x^2 + \ln(x)$$

```
> int(h(x),x=2..3);
```

$$6 + \ln(3) - \ln(2)$$

```
> evalf(");
```

$$6.405465108$$

IV.2.6 Développement limité

Le développement limité d'une fonction se fait sous différentes formes. Maple reconnait toutes les sortes de développement en séries : asymptotique, Chebyshev, Poisson, Taylor.

Développement de Taylor

Le développement en série de Taylor est un cas particulier des développements limités. Le développement de Taylor d'une fonction f(x) s'écrit sous la forme :

taylor(expression, variable, n)

où :

- **expression** : une fonction f(x) ou une expression algébrique
- **variable** : une variable x, ou une équation (x = a) c'est-à-dire, le développement autour de 0 ou a respectivement.
- **n** : l'ordre du développement.

```
> f:=x->sin(x);
```

$$f := \sin$$

```
> taylor(f(x),x,7);
```

$$x - \frac{1}{6}x^3 + \frac{1}{120}x^5 + O(x^7)$$

```
> taylor(f(x),x=1,3);
```

$$\sin(1) + \cos(1)(x-1) - \frac{1}{2}\sin(1)(x-1)^2 + O((x-1)^3)$$

Développement en séries (cas général)

L'instruction **series()** calcul le développement en séries d'une expression algébrique par rapport à une variable x autour d'un point a. Maple dispose en général d'une instruction de développement en séries :

series(expression, variable)
series(expression, variable, n)

Les paramètres:

- **expression** : une expression algébrique
- **variable** : la variable de la fonction, x ou x=a
- **n** : l'ordre du développement (optionnel)

```
> f(x);
```

$$\sin(x)$$

```
> series(f(x),x,7);
```

$$x - \frac{1}{6}x^3 + \frac{1}{120}x^5 + O(x^7)$$

```
> series(f(x),x=1,3);
```

$$\sin(1) + \cos(1)(x-1) - \frac{1}{2}\sin(1)(x-1)^2 + O((x-1)^3)$$

Il est possible de convertir le développement en séries d'une fonction en un polynôme de degré n en utilisant l'instruction **convert()**.

```
> s:=series(f(x),x,7);
```

$$s := x - \frac{1}{6}x^3 + \frac{1}{120}x^5 + O(x^7)$$

```
> P:=convert(s,polynom);
```

$$P := x - \frac{1}{6}x^3 + \frac{1}{120}x^5$$

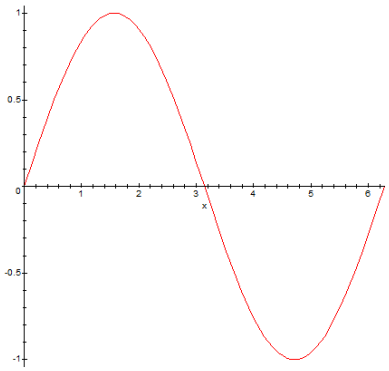
IV.2.7 Représentation graphique d'une fonction

L'instruction **plot()** constitue la façon la plus simple pour représenter graphiquement une fonction. L'intervalle de représentation par défaut est [-10,10].

```
> f(x);
```

$$\sin(x)$$

```
> plot(f(x),x=0..2*Pi);
```



Chapitre III : calcul différentiel

I. Résolution analytique

Maple dispose d'outils de résolution de certaines équations différentielles ; par exemple, les équations linéaires, du premier ordre à coefficients continus ou du second ordre calculables. Pour résoudre une équation différentielle, l'instruction dsolve() s'écrit comme suit :

dsolve(equation_diff, variable_recherche)

dsolve(equation_diff, variable_recherche, equation_opt)

Les paramètres passés en argument à ces instructions sont donnés par :

- **equation_diff** : est l'équation différentielle à résoudre, munie ou non de ses conditions initialles.
- **variable_recherchee** : la variable présumée solution de l'équation différentielle,
- **equation_opt** - une équation optionnelle de la résolution.

L'option de résolution est choisie en fonction du choix de l'utilisateur. Par exemple, si cette équation d'option est de la forme :

- **type=exact** : l'instruction **dsolve()** essai de trouver la solution exacte de l'équation différentielle.
- type=series :
- type=numeric :
- **explicit=true** : la solution est donnée explicitement en terme de variables indépendantes si c'est possible.

II. La résolution explicite des équations différentielles

L'instruction **dsolve()** est l'outil essentiel de résolution des équations différentielles. Elle prend comme premier argument :

- Une équation différentielle dans laquelle la dérivée
  - o $\frac{\partial y(x)}{\partial x}$: de la fonction *y(x)* par rapport à la variable *x* est notée *diff(y(x),x)*,
  - o $\frac{\partial^2 y(x)}{\partial x^2}$: la dérivées seconde étant notée *diff(y(x),x$2)*, et ainsi de suite.
- Un ensemble constitué d'une ou plusieurs équations différentielles
- Un ensemble d'une ou plusieurs équations différentielles et de conditions initiales ou aux limites.

Les conditions initiales ou aux limites sont dénotées par les quantités *y(a)* la valeur de *y* au point a, **D(y)(a)** celle de *y'(a)* et plus généralement **(D@@n)(y)(a)** la dérivée $n^{\text{-ième}}$ de y au point a.

Le deuxième argument de **dsolve()** est soit :

- la fonction inconnue *y(x)*,
- soit l'ensemble des fonctions inconnues, *y(x)*, *h(x)* .. dans le cas d'un système d'équations différentielles.

Dans la résolution d'équations différentielles, au cas où les conditions initiales ne sont pas spécifiées, Maple, introduit éventuellement des constantes arbitraires qu'il dénote par _C1,_C2,_C3 … Ces

constantes sont déterminées automatiquement lorsque les conditions initiales sont correctement spécifiées.

Un exemple de résolution générale d'équation différentielle au premier ordre :

```
> eqd  :=  diff(y(x),x)-2*y(x)=2*x;
```

$$eqd := \left(\frac{\partial}{\partial x} y(x)\right) - 2\,y(x) = 2\,x$$

```
> dsolve(eqd,y(x));
```

$$y(x) = -x - \frac{1}{2} + \mathbf{e}^{(2x)}\,\_C1$$

Un exemple de résolution générale d'équation différentielle au second ordre :

```
> eqd2  :=  diff(y(x),x$2)-2*diff(y(x),x)+4*y(x)=2*x;
```

$$eqd2 := \left(\frac{\partial^2}{\partial x^2} y(x)\right) - 2\left(\frac{\partial}{\partial x} y(x)\right) + 4\,y(x) = 2\,x$$

```
> dsolve(eqd2,y(x));
```

$$y(x) = \frac{1}{4} + \frac{1}{2}x + \_C1\,\mathbf{e}^{x}\sin(\sqrt{3}\,x) + \_C2\,\mathbf{e}^{x}\cos(\sqrt{3}\,x)$$

Un exemple de résolution générale d'équation différentielle au troisième ordre :

```
> eqd3:=diff(y(x),x$3)-y(x)=exp(x);
```

$$eqd3 := \left(\frac{\partial^3}{\partial x^3} y(x)\right) - y(x) = \mathbf{e}^{x}$$

```
>
> dsolve(eqd3,y(x));
```

$$y(x) = \frac{1}{3}x\,\mathbf{e}^{x} - \frac{1}{3}\mathbf{e}^{x} + \_C1\,\mathbf{e}^{x} + \_C2\,\mathbf{e}^{\left(-\frac{1}{2}x\right)}\sin\left(\frac{1}{2}\sqrt{3}\,x\right) + \_C3\,\mathbf{e}^{\left(-\frac{1}{2}x\right)}\cos\left(\frac{1}{2}\sqrt{3}\,x\right)$$

En principe, toute équation différentielle solvable peut être résolue par Maple. Les constantes _C1, _C2, _C3 des expressions ci-dessus sont déterminées automatiquement en spécifiant les conditions initiales ou aux limites. Par exemple :

```
> dsolve({eqd3,y(0)=0,D(y)(0)=1,(D@@2)(y)(0)=0},y(x));
```

$$y(x) = \frac{1}{3}x\,\mathbf{e}^{x} + \frac{4}{9}\sqrt{3}\,\mathbf{e}^{\left(-\frac{1}{2}x\right)}\sin\left(\frac{1}{2}\sqrt{3}\,x\right)$$

Dans l'expression ci-dessus, l'équation différentielle ainsi que ses conditions initiales sont mis entre accolade.

```
> eqd2:=diff(y(x),x$2)-4*y(x)=0;
```

$$eqd2 := \left(\frac{\partial^2}{\partial x^2} y(x)\right) - 4\,y(x) = 0$$

```
> dsolve({eqd2,y(0)=0,y(Pi)=1},y(x));
```

$$y(x) = -\frac{e^{(-2x)}}{e^{(-2\pi)}\left(-1+e^{(4\pi)}\right)} + \frac{e^{(2x)}}{e^{(-2\pi)}\left(-1+e^{(4\pi)}\right)}$$

```
> simplify(");
```

$$y(x) = \frac{-e^{(2\pi-2x)} + e^{(2\pi+2x)}}{-1+e^{(4\pi)}}$$

La méthode de résolution d'un système d'équations différentielles est similaire à celle utilisée dans le cas d'une seule équation.

```
> Systeme:={diff(x(t),t)=x(t)+3*y(t),diff(y(t),t)=x(t)-y(t),x(0)=0,y(0)=1};
```

$$Systeme := \{x(0)=0,\ y(0)=1,\ \frac{\partial}{\partial t}y(t) = x(t) - y(t),\ \frac{\partial}{\partial t}x(t) = x(t) + 3\,y(t)\}$$

```
> dsolve(Systeme,{x(t),y(t)});
```

$$\{x(t) = \frac{3}{4}e^{(2t)} - \frac{3}{4}e^{(-2t)},\ y(t) = \frac{3}{4}e^{(-2t)} + \frac{1}{4}e^{(2t)}\}$$

III. Récupération des résultats de la fonction dsolve()

Les résultats de la fonction **dsolve()** donnés dans les exemples précédents sont exprimés sous forme d'une équation ou d'un système d'équations égalisant les fonctions recherchées et leur expression. En général, dans le cas où l'équation différentielle est résolue, il est possible de récupérer, pour un usage ultérieur, l'expression de la solution trouvée. Cela peut se faire à la main, par sélection de l'expression affichée, la copier en mémoire puis la coller sur une ligne de commande. L'expression récupérée peut être modifiée pour s'adapter à son futur usage.

Dans un autre contexte, l'extraction des solutions peut s'effectuer en se servant des deux instructions suivantes :

- **op()** : pour extraire une opérande à partir d'une expression,
- **nops()** : pour déterminer le nombre d'opérandes dans une expression.

En général, les séquences d'appel de ces deux instructions s'écrivent sous la forme :

- **op(i,e) ; op(i..j,e) ; op(e) ; op(l,e) ;**
- **nops(e) ;**

où les paramètres sont spécifiés comme suit :

- *i, j* : sont des entiers qui marquent les positions des opérandes,
- *e* : l'expression contenant les opérandes,
- ***l*** : une liste d'entiers marquant les positions des opérandes en ordre croissant d'une expression.

Exemple :

```
> eqd:=diff(y(x),x)-2*y(x)=0;
```

$$eqd := \left(\frac{\partial}{\partial x} y(x)\right) - 2\,y(x) = 0$$

```
> dsolve({eqd,y(0)=1},y(x));
```

$$y(x) = e^{(2x)}$$

```
> Y:=op(2,");
```

$$Y := e^{(2x)}$$

La méthode susmentionnée présente des limites. En effet, elle présente l'inconvénient lorsque la solution n'est pas unique, c-à-d un ensemble de solutions est retourné. Surtout lorsque l'ordre des solutions n'est pas connu ou non respecter. Dans cette situation, la bonne méthode serait l'usage de la fonction **subs()**. Cette dernière s'écrit sous la forme suivante :

1. **subs(exp1 =exp2, expression)**
2. **subs(eq1,eq2,..,eqn, expr_recherchee)**

Dans l'instruction **1**, **exp1** désigne l'expression à substituer par l'expression **exp2** dans l'expression mère **expression**.

```
> f:=x->sin(x)*exp(a/(a-x)^2);
```

$$f := x \to \sin(x)\, e^{\left(\frac{a}{(a-x)^2}\right)}$$

```
> subs(x=t,f(x));
```

$$\sin(t)\, e^{\left(\frac{a}{(a-t)^2}\right)}$$

```
> subs(a=2,f(x));
```

$$\sin(x)\, e^{\left(\frac{2}{(2-x)^2}\right)}$$

Dans l'instruction **2**, le rôle de **subs()** consiste à chercher séquentiellement **expr_recherchee** dans chaque équation **eqi, i=1..n,** quand **expr_recherche** figure en membre de gauche de **eqi,** alors le membre de droite de **eqi** est renvoyer comme résultat.

Exemple :

```
> systeme:={x(t)=a*t,y(t)=b*t^2+2,z(t)=1/t};
```

$$systeme := \{y(t) = b\,t^2 + 2,\ z(t) = \frac{1}{t},\ x(t) = a\,t\}$$

```
> subs(systeme,x(t));
```

$$a\,t$$

```
> subs(systeme,z(t));
```

$$\frac{1}{t}$$

Le résultat de l'instruction **dsolve()** est formé d'une ou plusieurs solutions. Quelque soit leur classement, la recherche de la fonction cible peut être extraite par la méthode présenté dans la figure ci-dessus.

```
> eq1:=diff(x(t),t$2)=x(t)-diff(y(t),t)+y(t):
> eq2:=diff(y(t),t$2)=x(t)+diff(x(t),t)-y(t):
>
> solution:=dsolve({eq1,eq2,x(0)=0,y(0)=1,D(x)(0)=0,
  D(y)(0)=0},{x(t),y(t)});
```

$$solution := \{y(t) = \frac{1}{3}e^{t} + \frac{2}{3}\cos(\sqrt{2}\,t) - \frac{1}{6}\sqrt{2}\,\sin(\sqrt{2}\,t),$$

$$x(t) = -\frac{1}{3}\cos(\sqrt{2}\,t) + \frac{1}{3}e^{t} - \frac{1}{6}\sqrt{2}\,\sin(\sqrt{2}\,t)\}$$

```
>
> X:=subs(solution,x(t));
```

$$X := -\frac{1}{3}\cos(\sqrt{2}\,t) + \frac{1}{3}e^{t} - \frac{1}{6}\sqrt{2}\,\sin(\sqrt{2}\,t)$$

```
> Y:=subs(solution,y(t));
```

$$Y := \frac{1}{3}e^{t} + \frac{2}{3}\cos(\sqrt{2}\,t) - \frac{1}{6}\sqrt{2}\,\sin(\sqrt{2}\,t)$$

IV. Résolution numérique des équations différentielles

La robustesse de Maple à la résolution de plusieurs types d'équations différentielles n'est pas toujours garantie. Même dans le cas des problèmes à conditions initiales, Maple n'est pas capable de trouver une solution explicite. Par ailleurs, plusieurs autres types d'équations différentielles n'admettent pas de solutions analytiques. Cependant, la recherche de formes numériques capables d'approcher les solutions réelles tout en respectant les conditions initiales et/ou aux limites du problème sera toujours utile. En revanche, Maple améliore sa fonction **dsolve()** en lui ajoutant l'option **numeric** pour servir d'outil de résolution numérique d'équations différentielles.

Equation différentielle du premier ordre

Soit l'exemple illustré dans les instructions d'au-dessous :

```
> eqd1:=diff(y(t),t)=y(t)^2+sin(t);
```

$$eqd1 := \frac{\partial}{\partial t}y(t) = y(t)^{2} + \sin(t)$$

```
> dsolve({eqd1, y(0)=1},y(t));
>
> x_n:=dsolve({eqd1,y(0)=0},y(t),numeric);
```

$$x\_n := \mathbf{proc}(rkf45\_x)\ \ldots\ \mathbf{end}$$

```
>
> x_n(1.1);
```

$$[t = 1.1,\ y(t) = .6292363197953386]$$

Il est claire que l'instruction **dsolve()** dans la deuxième ligne n'a fourni aucun résultat. Alors, que lorsque l'option numérique est ajoutée, et que le résultat soit affecté à une variable **x_n**, un résultat apparait immédiatement sous forme d'une procédure d'argument « rkf45_x » qui signifie que la méthode numérique utilisée est celle de Runge-Kutta. Enfin, les résultats du calcul sont récupérés par la fonction **x_n(t)**.

En principe, la résolution numérique des équations différentielles ne pose pas de problème du moment qu'on peut construire la procédure de résolution en suivant l'une des méthodes bien connu dans ce domaine.

CHAPITRE IV : CALCUL ALGEBRIQUE

Le calcul algébrique ici concerne la manipulation des vecteurs et des matrices. En physique, ces deux grandeurs ont d'innombrables utilisatioes Toutefois, l'utilisateur peut se servir d'un nombre limité de fonctions et d'instructions de Maple pour tirer profil de toute la richesse de cet outil mathématique. Sous Maple, les vecteurs et les matrices sont traités selon deux méthodes. Soit ils sont considérés comme des tableaux à une et deux dimensions respectivement. Dans ce cas, ils sont manipulés par les opérations qui s'appliquent aux tableaux. Soit ils sont manipulés par les fonctions de Maple conçues spécialement pour eux et regroupées dans une bibliothèque dite package nommée « **linalg** ». Cependant, l'utilisateur doit indiquer l'usage de cette bibliothèque avant d'utiliser les fonctions concernées. Sinon, ces dernières ne seront pas reconnues.

I. Manipulation des vecteurs

Comme il a été évoqué, un vecteur sous Maple est manipulé selon deux façons :

- un tableau monodimensionnel qui obéit aux opérateurs usuels manipulant des tableaux,
- une variable de type vecteur « **vector** » après le chargement de la bibliothèque spéciale « **linalg** »

à titre d'exemple, un vecteur vu sous forme d'un tableau est déclaré comme suit :

```
> V:=array(1..3,[1,2,3]);
                    V := [1, 2, 3]
> V[1];
                          1
> V[3];
                          3
```

Les composants d'un tel vecteur sont indicés de 1,2 …

Avec le package « **linalg** », un vecteur est déclaré selon quatre formes :

1. **vector([x1, …, xn])**
2. **vector(n, [x1, …, xn])**
3. **vector(n)**
4. **vector(n, f)**

où les paramètres sont définis comme suit :

- **x1, …, xn** sont les éléments du vecteur,
- **n** la dimension d'un vecteur,
- **f** facultatif : pour initialiser les éléments du vecteur.

Exemple :

```
> V:=vector(4,[1,2,3,4]);
                    V := [1, 2, 3, 4]
> W:=vector(4);
                  W := array(1 .. 4, [ ])
> S:=vector(4,2);
                    S := [2, 2, 2, 2]
> T:=vector(4,rand(20));
                    T := [13, 1, 14, 6]
```

En dernière ligne, le paramètre **f** a pris le résultat de la fonction **rand(20)** qui signifie un nombre aléatoire allant de 0 à 20. C'est une façon de déclarer un vecteur aléatoire, sinon il est possible de déclarer dès le début un vecteur aléatoire.

Les fonctions agissantes sur les vecteurs:

Les instructions de manipulation des vecteurs sont nombreuses. Il suffit de charger le package « linalg ». A titre d'exemple, les instructions suivantes sont souvent utilisées.

- **angle(V,W)** : retourne l'angle en radians entre les vecteurs **V** et **W**,
- **dotprod(V,W)** : calcule le produit scalaire de **V** par **W**,
- **crossprod(u,v)** : calcule le produit vectoriel de **V** par **W,**
- **norm(v)** : calcule la norme du vecteur **V**,
- **normalize(V)** : normalise le vecteur **V**,
- **randvector(n)** : génère un vecteur de dimension **n** à éléments aléatoires.
- **vectdim(V)** : renvoie la dimension du vecteur **V**.

Les opérateurs agissants sur les vecteurs :

- **grad(expr,w,coords=type)** : calcule le gradient de l'expression **expr** par rapport aux composantes du vecteur **w** dans le système de coordonnées spécifié par type.

```
> with(linalg):
Warning, new definition for norm
Warning, new definition for trace
>
> X:=vector(3,[x,y,z]);
```

$$X := [x, y, z]$$

```
> f:=(x,y,z)->x^2+2*y+sin(z);
```

$$f := (x, y, z) \to x^2 + 2\,y + \sin(z)$$

```
> grad(f(x,y,z),X);
```

$$[2\,x, 2, \cos(z)]$$

- **laplacian(expr,v,coords=type)** : calcule le laplacien de l'expression **expr** par rapport aux composantes du vecteur **v** dans le système de coordonnées spécifié par type.
- **vectpotent(V,w,'U')** : retourne **TRUE** si le vecteur **V** est le rotationnel d'un potentiel vecteur **U** pour les coordonnées contenues dans le vecteur **w**.
- **potential(V,w,'F')** : retourne **TRUE** si le vecteur **V** est le gradient d'un potentiel scalaire **F** pour les coordonnées contenues dans le vecteur **w**.

II. Manipulation des matrices

En principe, les matrices sont des tableaux à 2 dimensions par les opérations valables des tableaux. Dans ce cas, les éléments des matrices sont considérés comme des expressions quelconques de Maple, même des chaines de caractères, et sont accessibles par le biais d'indices ***i,j*** parcourant les lignes et les colonnes. Par ailleurs, le package **« linalg »** offre des fonctions opérationnelles des matrices. Ainsi, il serait préférable de travailler dans son contexte.

Le package « **linalg** » contient les diverses fonctions nécessaires aux manipulations et aux calculs concernant les matrices. Une matrice diffère d'un tableau par ses indices qui sont forcément entiers et strictement positifs et le contenu de ses éléments qui devrait être nécessairement une expression mathématique valable.

II.1 Déclaration générale d'une matrice

Les matrices sont déclarées par l'instruction **matrix()** selon la syntaxe suivante :

- **matrix(Lv)**
- **matrix(m, n)**
- **matrix(m, n, Lv)**
- **matrix(m, n, f)**

Les paramètres des instructions ci-dessus ne sont pas tous optionnels et les plus importants d'entre eux sont **Lv** et **m,n**.

Lv : c'est une liste de vecteurs qui forment les colonnes de la matrice.

m,n : sont deux nombres qui définissent la dimension de la matrice. Dans cet exemple, les éléments de la matrice ne sont pas définis.

m,n,Lv : la spécification de ces trois paramètres déclare une matrice en initialisant ses éléments.

m,n,f : l'ajout du paramètre f permet d'initialiser les éléments de la matrice à des valeurs données, ici par exemple, un nombre aléatoire entre 0 et 30.

```
> A:=[[1,2],[3,4]];
```

$$A := [[1, 2], [3, 4]]$$

```
> A:=matrix([[1,2],[3,4]]);
```

$$A := \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

La création de la matrice par l'instruction **matrix(m,n)** ne permet pas son initialisation. Aucun élément de la matrice ne contient de valeur. Une matrice vide est utile lorsque son remplissage se fera par programmation ou par lecture des données provenant d'autres matrices, des fichiers etc. Cependant, il est très utile de créer une matrice initialisée dès le début par des 0 par exemple, des nombres aléatoire etc… sinon en choisissant une liste de vecteurs comme suit :

```
> B:=matrix(2,2);
```

$$B := \text{array}(1 .. 2, 1 .. 2, [\,])$$

ne

```
> evalm(B);
```

$$\begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix}$$

```
> C:=matrix(2,2,[[1,2],[3,4]]);
```

$$C := \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

```
> E:=matrix(2,2,rand(30));
```

$$E := \begin{bmatrix} 21 & 20 \\ 27 & 23 \end{bmatrix}$$

- **M :=matrix(n,m,[[v11,…v1m],[v21,…,v2m],…..[vn1,….vnm]]) ;**

Après la déclaration d'une matrice, Maple n'affiche pas cette dernière sous sa forme matricielle habituelle. Il faut appeler la fonction d'évaluation des matrice, **evalm()** pour afficher la forme désirée :

```
> A:=matrix(3,4);
```

$$A := \text{array}(1 .. 3, 1 .. 4, [\,])$$

```
> evalm(A);
```

$$\begin{bmatrix} A_{1,1} & A_{1,2} & A_{1,3} & A_{1,4} \\ A_{2,1} & A_{2,2} & A_{2,3} & A_{2,4} \\ A_{3,1} & A_{3,2} & A_{3,3} & A_{3,4} \end{bmatrix}$$

II.2. Opérations courantes sur les matrices

Contrairement aux variables scalaires, les matrices ne peuvent être utilisées qu'après leur déclaration. Les opérateurs standards agissant sur les matrices sont donnés comme suit :

- **+** : la somme
- **-** : la soustraction,
- **&*** : le produit,
- **^ ou **** : la puissance.

Exemple :

Soit les deux matrices A et B suivantes :

```
> A :=matrix([[2,3,4],[6,1,0],[8,9,7]]) ;
```

$$A := \begin{bmatrix} 2 & 3 & 4 \\ 6 & 1 & 0 \\ 8 & 9 & 7 \end{bmatrix}$$

```
> B :=matrix([[5,2,0],[4,9,3],[8,7,1]]) ;
```

$$B := \begin{bmatrix} 5 & 2 & 0 \\ 4 & 9 & 3 \\ 8 & 7 & 1 \end{bmatrix}$$

Comme il a été signalé, il faut évaluer les matrices ou les résultats sur les matrices pour obtenir un affichage selon les formes habituelles.

```
> A;
```

$$A$$

```
> A+B;
```

$$A + B$$

```
> evalm(A+B) ;
```

$$\begin{bmatrix} 7 & 5 & 4 \\ 10 & 10 & 3 \\ 16 & 16 & 8 \end{bmatrix}$$

Le produit de deux matrices est simple à obtenir, seulement il faut utiliser l'opérateur « **&*** » à la place de l'opérateur *.

```
> E:=A&*B;
```

$$E := A \,\&* B$$

```
> evalm(E) ;
```

$$\begin{bmatrix} 54 & 59 & 13 \\ 34 & 21 & 3 \\ 132 & 146 & 34 \end{bmatrix}$$

L'élévation en puissance d'une fonction ou l'application d'une fonction sur une matrice s'applique éventuellement à ses éléments un par un. L'exemple ci-dessous illustre le calcul d'une puissance et l'application de la fonction exponentielle sur une matrice A définie au début de l'exemple.

```
> evalm(A^3) ;
>
```

$$\begin{bmatrix} 666 & 531 & 468 \\ 342 & 289 & 240 \\ 1476 & 1203 & 1071 \end{bmatrix}$$

```
> evalm(exp(A)) ;
```

$$\begin{bmatrix} e^2 & e^3 & e^4 \\ e^6 & e & 1 \\ e^8 & e^9 & e^7 \end{bmatrix}$$

Les opérateurs élémentaires sont accessibles directement sous Maple. Toutefois, la manipulation des matrices ne se limite pas à ce niveau. En revanche, Maple fournit sa bibliothèque spéciale à ce propos, en l'occurrence, le package **linalg**.

II.3 Les fonctions du package linalg

Le package **linalg** est composé d'un grand nombre de fonctions. Ces fonctions sont destinées à manipuler et exploiter les vecteurs et les matrices. Pour pouvoir utiliser ces fonctions il suffit de charger le package **linalg** avec l'instruction suivante :

```
> with(linalg);
Warning, new definition for norm
Warning, new definition for trace
[BlockDiagonal, GramSchmidt, JordanBlock, LUdecomp, QRdecomp, Wronskian, addcol,
   addrow, adj, adjoint, angle, augment, backsub, band, basis, bezout, blockmatrix, charmat,
   charpoly, cholesky, col, coldim, colspace, colspan, companion, concat, cond, copyinto,
   crossprod, curl, definite, delcols, delrows, det, diag, diverge, dotprod, eigenvals, eigenvalues,
   eigenvectors, eigenvects, entermatrix, equal, exponential, extend, ffgausselim, fibonacci,
   forwardsub, frobenius, gausselim, gaussjord, geneqns, genmatrix, grad, hadamard, hermite,
   hessian, hilbert, htranspose, ihermite, indexfunc, innerprod, intbasis, inverse, ismith, issimilar,
   iszero, jacobian, jordan, kernel, laplacian, leastsqrs, linsolve, matadd, matrix, minor,
   minpoly, mulcol, mulrow, multiply, norm, normalize, nullspace, orthog, permanent, pivot,
   potential, randmatrix, randvector, rank, ratform, row, rowdim, rowspace, rowspan, rref,
   scalarmul, singularvals, smith, stack, submatrix, subvector, sumbasis, swapcol, swaprow,
   sylvester, toeplitz, trace, transpose, vandermonde, vecpotent, vectdim, vector, wronskian]
```

Au chargement de la bibliothèque, le résultat sera la liste des différentes fonctions prises en charge avec deux avertissements au début. Ces avertissements concerne la redéfinition des fonctions **norm** et **trace**. Les fonctions du package « **linalg** » sont nombreuses. Dans une phase d'initiation, il est préférable de se limiter à celles dont le concept est simple.

II.3.1 Matrices particulières:

Les matrices particulières sont des raccourcis vers des structures matricielles très utiles et souvent rencontrées. Ces raccourcis permettent d'éviter des lignes de commandes parfois marginales au problème. Les instructions suivantes décrivent ces situations.

- **diag(B1,B2,…Bn)** construit une matrice en plaçant les matrices B1,B2…Bn sur la diagonale. La matrice générée est diagonale par blocs. Dans l'exemple à droite, la matrice A et B sont utilisée pour former la matrice diagonale C.

- **genmatrix(eqns,vars) :** génère une matrice à partir des coefficients des variables **vars** dans le système d'équations linéaires **eqns**.
- **jacobian(V,u)** : calcule la matrice jacobéenne du vecteur **V** par rapport aux coordonnées stockées dans le vecteur **u**.
- **randmatrix(m,n)** : génère une matrice aléatoire de dimension **mxn**.

```
> A:=matrix([[1,2],[3,4]]);
```

$$A := \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

```
> B:=matrix(2,2,rand(30));
```

$$B := \begin{bmatrix} 21 & 20 \\ 27 & 23 \end{bmatrix}$$

```
> C:=diag(A,B);
```

$$C := \begin{bmatrix} 1 & 2 & 0 & 0 \\ 3 & 4 & 0 & 0 \\ 0 & 0 & 21 & 20 \\ 0 & 0 & 27 & 23 \end{bmatrix}$$

```
> A:=matrix([[1,2],[3,4]]);
```

$$A := \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

```
> B:=matrix(2,2,rand(30));
```

$$B := \begin{bmatrix} 21 & 20 \\ 27 & 23 \end{bmatrix}$$

```
> C:=augment(A,B);
```

$$C := \begin{bmatrix} 1 & 2 & 21 & 20 \\ 3 & 4 & 27 & 23 \end{bmatrix}$$

La liste des fonctions d'au dessus n'est pas exhaustive. Il existe d'autres matrices particulières sous Maple. Il suffit de revenir sur l'ensemble des fonctions de la bibliothèque « **linalg** ».

II.3.2 Les fonctions de manipulation des matrices

- **augment(A,B,…)** crée une matrice en joignant horizontalement les matrices A,B,… C'est l'équivalent d'une concaténation des matrices.
- **row(A,i)** : extrait la ligne i de la matrice A.
- **row(A,i..k)** : extrait les lignes de i à k de la matrice A.
- **col(A,i)** extrait la colonne i de la matrice A.
- **col(A,i..k)** extrait les colonnes de i à k de la matrice A.
- **delrows (A,r..s)** supprime les lignes de r à s de la matrice A.
- **delcols(A,r..s)** supprime les colonnes r à s de la matrice A.
- **copyinto(A,B,m,n)** copie la matrice A dans la matrice B en plaçant l'élément A11 en Bmn et ainsi de suite.
- **extend(A,m,n,x)** ajoute m lignes et n colonnes initialisées à x à la matrice A
- **minor(A,r,c)** supprime la ligne r et la colonne c de la matrice A.
- **submatrix(A,i..j,k..t)** extrait la sous matrice de A constituée par les lignes i à j et les colonnes k à t.
- **swapcol(A,c1,c2)** permute les colonnes c1 et c2.
- **swaprow(A,r1,r2)** permute les lignes r1 et r2.

III.3.3 Effectuer des tests sur les matrices:

- **coldim(A)** : retourne le nombre de colonnes de A.
- **rowdim(A)** : retourne le nombre de lignes de A.
- **equal(A,B)** : retourne TRUE si Aij=Bij pour tous i et j.
- **iszero(A)** : retourne TRUE si Aij=0 pour tous i et j.
- **colspace(A,'dim')** retourne une base de l'espace vectoriel engendré par les vecteurs colonnes de la matrice. Le paramètre 'dim' est optionnel et définit la dimension de cet espace vectoriel.

- **rowspace(A,'dim')** retourne une base de l'espace vectoriel engendré par les vecteurs lignes de la matrice. 'dim' est la dimension de cet espace vectoriel.
- **orthog(A)** retourne TRUE si A est une matrice orthogonale, c-à-d, A est constituée de vecteurs colonnes ortho-normaux.
- **rank(A)** retourne le rang de la matrice A.
- **trace(A)** retourne la trace de la matrice A.

II.3.4 Opérations sur les matrices:

Les opérateurs définis dans la section précédente diffèrent de ceux qui sont fournis dans le contexte du package « **linalg** ». Les opérateurs de « **linalg** » sont plus généraux et offrent plus de fonctionnalités.

- **add(A,B,c1,c2)** : calcule la somme pondérée des matrices **A** et **B** à coefficients **c1** et **c2**.
  N.B : cette fonction n'est pas disponible sur les anciennes versions de Maple.
- **multiply(A,B,…)** multiplie les matrices A,B,… entre-elles.
- **exponential(A)** : retourne la matrice formée par l'exponentiel des éléments.
- **addrow(A,li,lj**) : crée une matrice **A'** où la ligne ***lj*** est remplacée par la somme de la ligne ***li*** et celle de la ligne *lj*.
- **addcol(A,c1,c2)** : crée une matrice **A'** où la colonne **c2** est remplacée par la somme de c1 et c2.
- **mulrow(A,r,expr)** : multiplie la ligne r de la matrice A par expr.
- **mulcol(A,c,expr)** : multiplie la colonne c de la matrice A par expr.
- **adjoint(A)** : retourne la matrice adjointe de **A**. Elle est telle que **A**. **A†=det(A)*I** où **I** est la matrice identité. C'est équivalent à **adj(A)**.
- **htranspose(A) :** retourne le transposé Hermitien de **A.**
- **transpose(A**) : retourne le transposé de la matrice **A**.
- **det(A)** : calcule le déterminant, s'il existe, de la matrice **A**.
- **inverse(A)** : calcule l'inverse, s'il existe, de la matrice carrée **A**.
- **norm(A,opt)** : calcule la norme de la matrice A suivant l'option spécifiée.
- **charmat(A,lambda)** : retourne la matrice caractéristique M de A.
- **charpoly(A,lambda)** : retourne le polynôme caractéristique P de A.
- **eigenvals(A)** : donne les valeurs propres de la matrice A. Ce sont aussi les solutions du polynôme caractéristique précédent.
- **eigenvects(A)** : retourne les vecteurs propres de la matrice A sous la forme :
  [[val. prop. i, multiplicité, {vect[1,i],…vect[ni,i]}]…]
  où **vect[j,i]** est un vecteur propre associé à la valeur propre ***i*** et ***ni*** est la dimension du sous espace propre de cette valeur propre.
- **jordan(A)** retourne la forme de Jordan de la matrice **A** à savoir une matrice diagonale dont la diagonale est constituée par les valeurs propres de **A**.
- **linsolve (A,B)** si **B** est un vecteur, Maple retourne le vecteur **x** solution du système d'équations **A.x=B**. Si **B** est une matrice, Maple retourne la matrice **X** solution de l'équation matricielle **A.X=B**.

CHAPITRE V : LE GRAPHISME SOUS MAPLE

La représentation graphique des données est la phase la plus importante lors de l'étude d'un phénomène. Les données sous leurs formes numériques nécessitent plus d'effort pour les interpréter ou les évaluer. Tandis que les données représentées par un graphe permettent une lecture et une interprétation facile et de qualité. Sous Maple, la représentation graphique est très simple à utiliser. Son environnement fourni les instructions nécessaires pour représenter différentes sortes de données. Ces instructions sont données selon la dimension choisie : deux dimensions (y=f(x) 2D) ou trois dimensions (z=f(x,y) ; 3D).
Vue l'importance de la représentation graphique, ce chapitre est dédier à cette partie de Maple quoiqu'elle soit son utilisation, il pourra servir d'aide-mémoire.

I. Représentation graphique en deux dimensions (2D)

Les instructions ne sont pas nombreuses. Seulement, la façon d'utiliser une fonction diffère d'une situation à une autre. Cependant, il serait plus intéressant de parler d'abord des options de représentation graphique. Ces options font en chaque fois un cas de figure.

I.1 Les options

La majorité des fonctions graphiques admettent une série d'options dont les effets sont diversifiés. Il est plus intéressent de les essayer au moins pour la majorité d'entre elles. Comme nous allons le voir, la syntaxe est simple : les options s'écrivent à la suite des paramètres fondamentaux de chaque fonction en les séparant par des virgules. En général, la syntaxe est donnée par l'instruction suivante :

fonction(paramètres, option1=valeurs1,option2=valeur2…);

En réalité, aucune option n'est obligatoire car Maple affecte une valeur par défaut à chacune d'elles. Les options les plus importantes sont les suivantes:

- **scaling** : Elle contrôle l'échelle du graphe. Elle comporte deux valeurs :
  - **CONSTRAINED** : tout changement d'échelle affecte également x et y,
  - **UNCONSTRAINED** : La valeur par défaut, le changement d'échelle n'affecte pas x et y à la fois.
- **axes** : C'est le choix du type des axes.
  - **FRAME** : axes sur le bord;
  - **BOXED** : axes entourant le graphe;
  - **NORMAL** : axes se croisant en zéro;
  - **NONE** : pas d'axes.
- **coords=polar** : choisir l'usage des coordonnées polaires. Les coordonnées sont utilisées par ordre : rayon, angle. le nom n'est pas imposé (r,theta), (h, phi) etc.
- **numpoints=n** : nombre de points minimum générés lors de l'appel à la fonction (par défaut n=49). Maple choisit automatiquement d'en générer plus quand la fonction n'est pas régulière.
- **resolution=n**: choix de la résolution horizontale de l'écran en pixels (par défaut n= 200).

- **color=n:** choix de la couleur. ***n*** peut être un nom parmi les noms suivants:
  - aquamarine,black,blue,navy,coral,cyan,brown,gold,green,gray,grey,khaki,magenta, maroon, orange, pink, plum, red, sienna, tan, turquoise, violet, wheat, white, yellow.
  - ou peut aussi être soit COLOR(RGB,a,b,c) soit COLOR(HUE,d) où a,b,c,d sont compris entre 0 et 1.
- **xtickmarks=n** : nombre minimum de graduations sur l'axe des x.
- **ytickmarks=n** : nombre minimum de graduations sur l'axe des y.
- **style=s :** s=LINE implique que la courbe serait une ligne (par défaut LINE). S=POINT, la courbe serait une collection de points.
- **discont=s:** si s a la valeur TRUE, Maple analyse la fonction pour trouver ses discontinuités et faire ensuite un graphe propre. Par défaut, s est FALSE (n'est pas valable pour les anciennes versions).
- **title=t**: définit le titre du dessin. t est une chaîne de caractères entre ''.
- **thickness=n**: définit l'épaisseur des lignes du tracé, **n** vaut 0, 1, 2, ou 3.0 et l'épaisseur par défaut est 1.
- **linestyle=n:** définit le style des lignes. Si n=0 tracé plein.
- **symbol=s**: choix de la forme des points lors d'un tracé par points parmi BOX, CROSS, CIRCLE, POINT, et DIAMOND. La valeur par défaut est POINT.
- **view=[xmin..xmax,ymin..ymax]:** choix de la partie de la courbe à afficher. Par défaut la courbe entière est affichée.

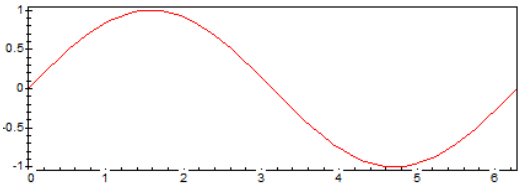
II.2 Les fonctions

Le tracé d'une courbe est obtenu par l'instruction suivante :

plot(expression, dh, dv, options) ;

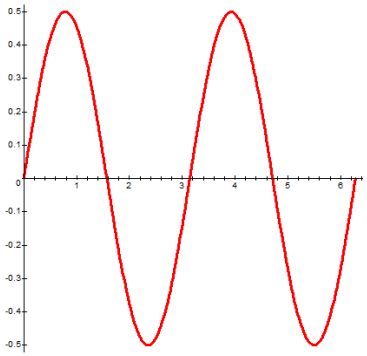
- **expression** : expression mathématique représentant la fonction à tracer,
- **dh** : est le domaine horizontal de la forme var=valeur1..valeur2, par exemple (x=0..4) ou simplement valeur1.. valeur2 (0..4) si la variable est évidente. La spécification de ce paramètre est obligatoire,
- **dv** : est la domaine vertical d'affichage. il est optionnel. Par défaut, toute la courbe est affichée.
- options: ce sont les options définies auparavant, Elles ne sont pas obligatoires. L'utilisateur peut choisir ce qu'il veut comme options sans se soucier de leur ordre.

Exemple 1 :




```
> plot(sin(x),x=0..2*Pi,axes=BOX,scaling=CONSTRAINED);
```

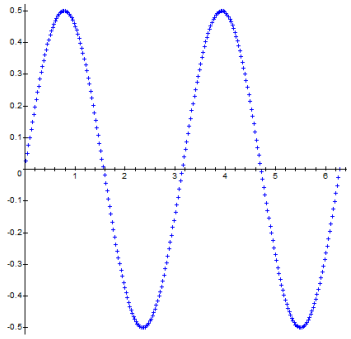
Dans cet exemple, l'option axes=BOX, ce qui fait, la courbe est tracé dans une boite ou rectangle. De même, scaling=CONSTRAINED ce qui fait l'échelle sur les axes est respectée contrairement aux cas des figures des exemples 2,3,4



Exemple 2 :

```
> plot(cos(x)*sin(x),x=0..2*Pi,numpoints=250,
  color=red,ytickmarks=8,style= LINE,
  thickness=3);
```

Dans cet exemple, la couleur est choisie : color=red, le nombre de graduation s sur l'axe des y est 8, le style de la courbe : ligne, l'épaisseur du trait est 3. l'échelle sur l'axe x n'est pas la même que celle sur l'axe des y.



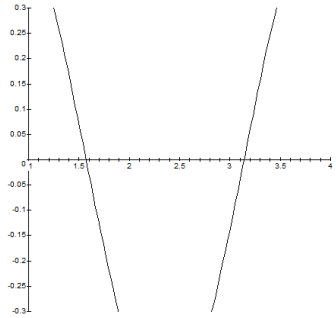
la

Exemple 3 :

```
> plot(cos(x)*sin(x),x=0..2*Pi,numpoints=25
  0,color=blue,ytickmarks=8,style= POINT);
```

Dans cet exemple, le tracé est le même que celui de l'exemple 2, sauf cette fois-ci la couleur : color=blue et le style : style = POINT, qui change la courbe du trait en pointillés.

ce

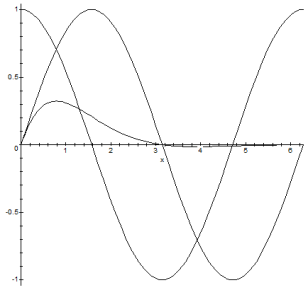


Exemple 4 :

```
> plot(cos(x)*sin(x),x=0..2*Pi,numpoints=250,col
  or=black,ytickmarks=8,style=
  LINE,view=[1..4,-0.3..0.3]);
```

Le graphe de cet exemple est un extrait de celui tracé dans l'exemple 2; il correspond à une partie du graphe 2 limité par le cadre : 1<=x<=4, et -0.3<=y<=0.3

La fonction **plot()** permet également de tracé plusieurs fonctions en même temps. Il suffit de mettre les fonctions à tracer entre crochets et substituer l'ensemble à la place de l'expression utilisée dans le cas du tracé d'une seule fonction.



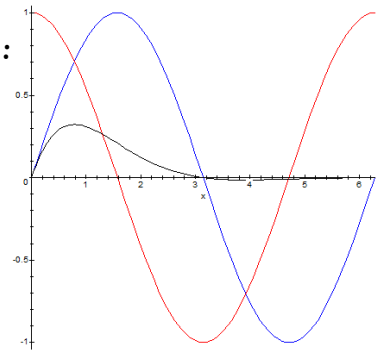
Exemple1 :

```
> plot([sin(x),cos(x),sin(x)*exp(-x)],x=0..2*Pi,
  color=black);
```

Dans cet exemple, les trois fonctions sont représentées simultanément. Le seul inconvénient est que les options ne peuvent être spécifiées pour chacune d'entre elles. Surtout pour différentier chaque courbe en spécifiant des couleurs distinctes pour chaque fonction. Pour remédier à ce problème, Maple fournit une bibliothèque ou package « **plots** ». Il est cependant possible de déclarer chaque tracé indépendant, puis les rassembler en un seul graphe.

```
> with(plots):
>
> a:=plot(sin(x),x=0..2*Pi,color=blue):
  b:=plot(cos(x),x=0..2*Pi,color=red):c:=plot(si
  n(x)*exp(-x), x=0..2*Pi,color=black):
>
> display({a,b,c});
```

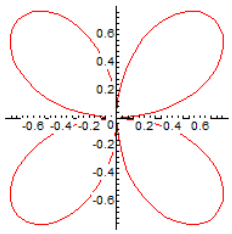
Exemple 2 :



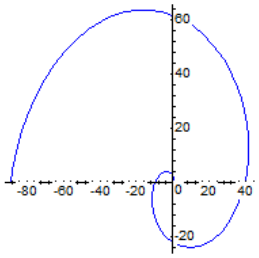
L'appel de la fonction **plot()** est fait indépendamment pour chacune des trois fonctions, chaque résultat est affecté à une variable différente, l'instruction **dipslay()** permet d'afficher le contenu des 3 variables, notamment le tracé de chaque fonction.

Le changement de type de coordonnées est possible grâce à l'option « **coords=type_cord** ». Le traçage d'une fonction en coordonnées polaires est un cas particulier des fonctions paramétriques.

```
> plot([sin(2*t),t,t=0..2*Pi],coords=polar);
```



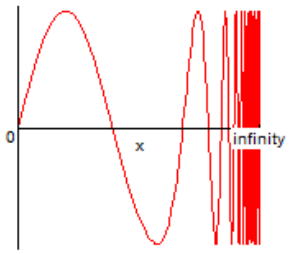
```
> plot([x^2,x,x=0..3*Pi],-8..8,-10..10,
  coords=polar, color=blue);
```



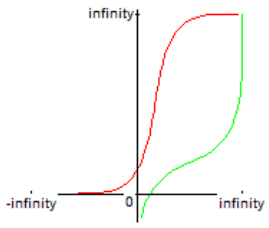
Comme il a été mentionné, les coordonnées sont prises dans l'ordre ; le rayon (x) et l'angle (y).

La représentation des fonctions dont les domaines sont infinis est possible. Maple permet de simuler le comportement à l'infini des fonctions grâce à une variation d'échelle en fonction du besoin. Prenons les figures ci-dessus comme exemple.

```
> plot(sin(x),x=0..infinity);
```




```
> plot( {exp,ln}, -infinity..infinity
  );
```

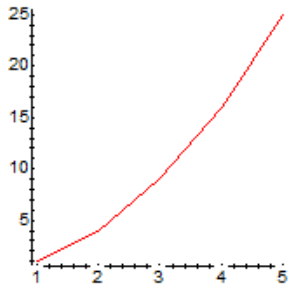


La représentation de la fonction sin(x) de x=0 à l'infini (à gauche) et les fonctions (exp(x),log(x)) à droite nous informe que Maple n'utilise pas la même échelle sur toutes les régions des graphes.

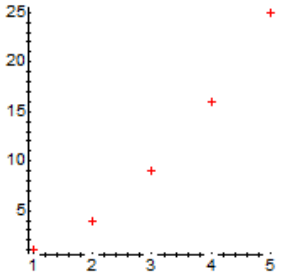
Dans certaines situations, les données à représenter graphiquement ne sont pas données par une fonction. Elles sont plutôt données sous forme d'une liste de couple (x,y) où chaque couple désigne un point du graphe. Il suffit de définir la liste des points et passer la liste en argument à la fonction plot().

L : = [[x1,y1],[x2,y2],...].
plot(l, x=0..2, style=point,symbol=diamond);

```
> l:=[[1,1],[2,4],[3,9],[4,16],[5,25]];
      l := [[1, 1], [2, 4], [3, 9], [4, 16], [5, 25]]
> plot(l);
```




```
> plot(l,style=POINT);
```



Dans le premier graphe ci-dessus, aucune option n'a été choisie même le domaine de définition. Ce n'était pas obligatoire puisque les données sont déjà spécifiées sous forme de (x,y).

Les versions récentes de Maple peuvent tracer les fonctions à variables complexes. Celles-ci sont tracées par la fonction **conformal().** Le pas de la grille est fixé par défaut (11 lignes) et modifiable par **grid=[n1,n2]**. Le nombre de points évalués sur chaque ligne de la grille est fixé par **numxy=[n1,n2]** ou par défaut (15 points).

De même, les champs de vecteurs comme **[f(x,y),g(x,y)]** sont tracés par l'instruction **fieldplot()** un vecteur évalué en x,y de direction et de norme fixée par f et g.

Les champs de gradients : Maple trace par la fonction **gradplot()** un champ de vecteurs dérivant d'un potentiel donné.

Les tracés particuliers:

- ✓ Le tracé semi-logarithmique: seule l'échelle sur l'axe y est logarithmique.
  - **logplot(x->10^x,1..10);**
  - **logplot({ x->2^(sin(x)), x->2^(cos(x))}, 1..10);**
- ✓ Le tracé logarithmique: les deux échelles sont logarithmiques.
  - **loglogplot(10^x,x=1..10);**
  - **loglogplot([1,2,3,4,5,6,7,8],style=POINT);**
  - **loglogplot([[1,2],[3,4],[5,6],[7,8]]);**
- ✓ Le tracé de densité: il permet par la fonction **densityplot()** le tracé d'une fonction 3D en 2D. Plus la valeur de la fonction au point x,y est élevée, plus le tracé est foncé.
  - **densityplot(sin(x*y),x=-Pi..Pi,y=-Pi..Pi,axes=boxed);**
- ✓ Le tracé retardé: tout tracé est une structure à laquelle on peut affecter un nom. Si on écrit variable=plot(…), le tracé est calculé mais pas affiché. Il est stocké dans variable. Pour l'afficher ensuite, il y a deux possibilités: soit variable; soit **display(variable,options)**. L'avantage de display est qu'il permet un affichage multiple.
  - **F:=plot(cos(x),x=-Pi..Pi,y=-Pi..Pi,style=line):**
  - **G:=plot(tan(x),x=-Pi..Pi,y=-Pi..Pi,style=point):**
  - **display({F,G},axes=boxed,scaling=constrained,title=`Cosinus & Tangente`);**
- ✓ l'affichage de texte: la syntaxe est **textplot([x,y,`texte`],options);.** Maple affiche le texte au point x,y. Il y a de nombreuses options sur le texte. On retiendra seulement celles d'alignement: **align=ABOVE,RIGHT,LEFT,BELLOW.**
  - **p := plot(sin(x),x=-Pi..Pi):**
  - **delta := 0.05:**
  - **t1 := textplot([Pi/2,1+delta,`Local Maxima (Pi/2, 1)`],align=ABOVE):**
  - **t2 := textplot([-Pi/2,-1,`Local Minima (-Pi/2, -1)`],align=BELOW):**
  - **display({p,t1,t2});**
- ✓ Le tracé de polygones: **polygonplot()** trace un polygone à partir de la séquence des coordonnées de ses sommets.
  - **ngon := n -> [seq([cos(2*Pi*i/n), sin(2*Pi*i/n)], i = 1..n)]:**
  - **display([polygonplot(ngon(8)), textplot([0,0,`Octagon`])], color=BLUE);**
- ✓ le tracé de matrice: en 2D, le tracé se limite à un tracé de contenu. Maple affiche par la fonction sparsematrix() un point en x (indice de ligne) et y (indice de colonne) si l'élément de matrice est non nul.
  - **with(linalg):**
  - **A := randmatrix(15,15,sparse):**
  - **B := gausselim(A):**
  - **PA := sparsematrixplot(A, color=green):**
  - **PB := sparsematrixplot(B, color=red):**

- **display({PA, PB}); changer la forme des points**

Les animations graphiques :

Il est possible de créer des animations de courbes par la fonction animate().

L'option frames=n fixe le nombre d'images de l'animation (par défaut 16); l'option numpoints=m fixe le nombre de points de chaque image.

- **animate(sin(x*t),x=-10..10,t=1..2,frames=50);**
- **animate({x-x^3/u,sin(u*x)}, x=0..Pi/2,u=1..16 ,color= red);**
- **animate([u*t,t,t=1..8*Pi], u=1..4,coords=polar,frames=60,numpoints=100);**

On peut stocker une animation dans une variable pour affichage ultérieur.

- **P := animate(sin(x+t),x=-Pi..Pi,t=-Pi..Pi,frames=8):**
- **Q := animate(cos(x+t),x=-Pi..Pi,t=-Pi..Pi,frames=8):**
- **display([P,Q]);**

II graphisme en 3d

Comme dans le cas de la représentation 2D, le graphisme en 3D dispose de plusieurs options.

II.1 Les options 3D

Les options du graphisme en 2D sont toujours valables ici mais il s'y ajoute un lot d'autres options spécifiques au graphisme en 3D. Nous ne citerons que les principales.

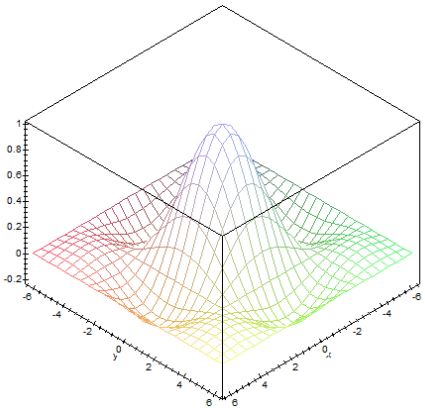
- **grid=[m,n]** : permet de choisir le maillage sur lequel sont évalués les points.
- **style=s** : permet de choisir le mode de dessin de la surface. Le paramètre **s** prend les valeurs : POINT, HIDDEN (parties cachées non représentées), PATCH, WIREFRAME (maillage uniquement représenté), CONTOUR lignes de niveau), PATCHNOGRID (combinaison de styles), PATCHCONTOUR (combinaison de styles), ou LINE. La valeur par défaut est HIDDEN.
- **contours = n** : permet de définir le nombre de lignes de niveau. **n** est un entier positif (10 par défaut).
- **coords=c** : pour choisir le système de coordonnées : cartesian, spherical et cylindrical. La valeur par défaut est « cartesian ».
- **projection=r** : pour choisir de la perspective, **r** doit être compris entre 0 et 1. r=0 correspond à une vue grand angle, r=1 correspond à une projection orthogonale, r=0.5 correspond à la projection normale (valeur par défaut:1).
- **orientation=[theta,phi]**: pour choisir le point de vue sous forme des deux angles sphériques theta et phi. Par défaut les deux angles valent 45°.
- **shading=s** : pour choisir le mode de colorisation de la surface. Le paramètre **s** prend l'une des valeurs : XYZ, XY, Z, Z_GREYSCALE, Z_HUE, NONE.
- **ambientlight=[r,g,b]**: pour définir la lumière d'ambiance dans laquelle baigne le dessin. r,g,b (pour red,green,blue) sont compris entre 0 et 1.

- **light=[phi,theta,r,g,b]:** pour choisir l'éclairage du dessin. la source éclaire sous les angles theta et phi avec une lumière dont la couleur est fixée par les valeurs de r,g et b.

II.2 Les fonctions 3D

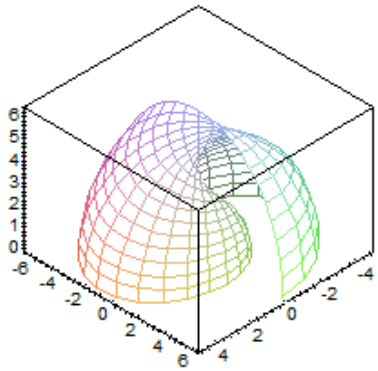
La représentation d'une courbe en mode 3D se fait par l'instruction **plot3d()** don la syntaxe s'écrit comme suit :

plot3d(expression,dx,dy,options);



Les paramètres en argument sont :

- **expression** : une expression mathématique valable,
- **dx** est le domaine de variation de x de la forme var=valeur1..valeur2 ou simplement valeur1..valeur2 si la variable est évidente. dx est obligatoire
- **dy** est le domaine de variation de y de la forme var=valeur1..valeur2 ou simplement valeur1..valeur2 si la variable est évidente. dy est obligatoire
- **options** : ce sont les options définies à la section précédente plus celles du graphisme en mode 2D. Aucune n'est obligatoire.



En graphisme 3D, il y a souvent plusieurs possibilités pour obtenir un même tracé:

1. soit par **plot3d**() avec le choix convenable des options,
2. soit par la fonction dédiée. Dans la suite, chaque fois que la fonction dédiée existe, elle est présentée avec son homologue **plot3d()**.

Exemple 1 :

```
> plot3d(sin(x)/x*sin(y)/y,x=-2*Pi..2*Pi,
  y=-2*Pi..2*Pi, axes=BOX);
```

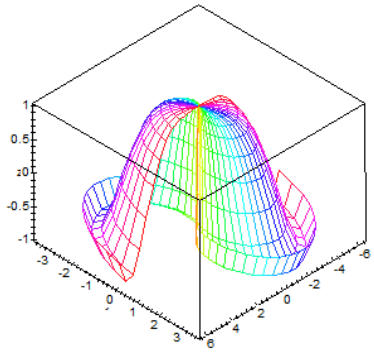
Dans cet exemple, la seule option utilisée est celle du choix des axes, « axes = BOX ». Ainsi, le graphe apparait logé dans un cube. L'axe des x est celui de droite, alors que l'axe des y se trouve à gauche. L'axe des z est vertical.

Exemple 2 :

```
> plot3d([x*sin(x)*cos(y),
  x*cos(x)*cos(y),x*sin(y)
  ],x=0..2*Pi,y=0..Pi,
  axes=BOX);
```

Dans cet exemple, deux fonctions sont représentées simultanément. Pour les différentier de vue, il est nécessaire de les tracées en différente couleur. Cependant, la meilleurs méthode serait d'utiliser des tracés différents et de les présenter par **display().**

Les tracés graphiques en mode 3D sont très variés sous Maple. Selon le besoin, des fonctions autres que **plot3d()** sont disponibles comme par exemple, tracer une courbe en coordonnées paramétriques. Pour ce faire, l'instruction de traçage est **spacecurve([fonction],domaine)**;.



Exemple

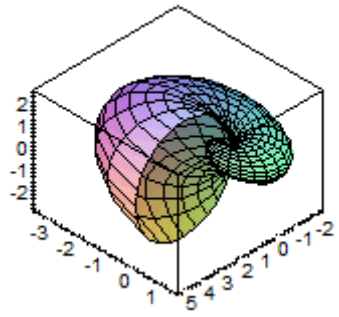
```
> spacecurve([cos(t),sin(t),t],
  t=0..4*Pi, axes=BOX);
```

Ici, les cordonnées (x,y,z) des points du tracé sont paramétrées par t telles que :
x=cos(t), y=sin(t), z=t.

Tracer une fonction en coordonnées cylindriques est facile à faire. Il suffit de spécifier l'option « **coords=cylindrical** ».

Exemple :

```
> plot3d(theta*z/5,theta=0..6*P
  i,z=-5..5,coords=cylindrical,
  axes=BOX);
```



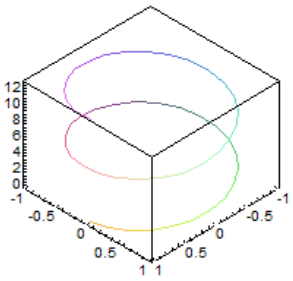
La représentation ci-dessus peut être aussi réaliser par la fonction

cylinderplot(expression, dtheta, dz, options);

où theta est le domaine de variation de theta alors que dz celui de z.

Exemple :

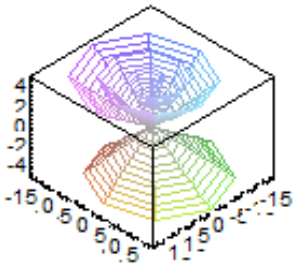
```
> cylinderplot([z*theta,theta,cos(z^2)]
  ,theta=0..Pi,z=-2..2, color = theta,
  axes=BOX);
```

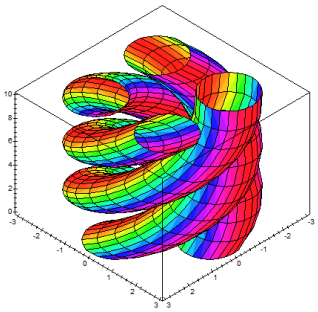


La représentation d'une expression en coordonnées sphériques est obtenue par la spécification de l'option « **coords=spherical** ».

Exemple :

```
> plot3d((1.3)^x *
  sin(y),x=-1..2*Pi,y=0..Pi,coords=spherical
  ,style=patch, axes=BOX);
```





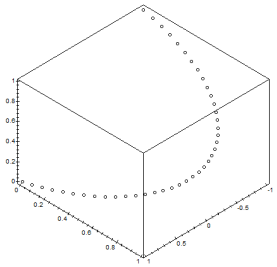
Semblable au cas des coordonnées polaires, une fonction spéciale pour la représentation en coordonnées sphériques est disponible. Il s'agit de la fonction suivante :

sphereplot(expression, dphi,dtheta, options);

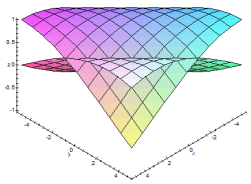
où **dphi** est le domaine de variation de **phi** et **dtheta** celui de **theta**.

Exemple :

```
> sphereplot((1.3)^z *
  sin(theta),z=-1..2*Pi,theta=0..Pi,
  style=patch,color=z, axes = BOX);
```

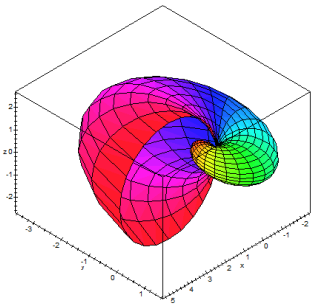


Comme dans le cas du mode 2D, il est facile de représenter un ensemble de fonctions. Il suffit de les mettre entre accolades et de les passer en arguments comme expression.



Exemple :

```
> c1:= [cos(x)-2*cos(0.4*y),sin(x)-2*sin(0.4*y),y]:
  c2:= [cos(x)+2*cos(0.4*y),sin(x)+2*sin(0.4*y),y]:
  c3:= [cos(x)+2*sin(0.4*y),sin(x)-2*cos(0.4*y),y]:
  c4:= [cos(x)-2*sin(0.4*y),sin(x)+2*cos(0.4*y),y]:
  plot3d({c1,c2,c3,c4},x=0..2*Pi,y=0..10,grid=[25,15]
  ,style=patch,color=sin(x), axes=BOX);
```



Dans plusieurs cas, l'expression à tracer n'est pas données sous forme de formule mathématique. Elle peut être un ensemble de points distincts notés selon les deux formes suivantes :

- **[x1,y1,x2,y2,...]**
- **[[x1,y1],[x2,y2],...].**

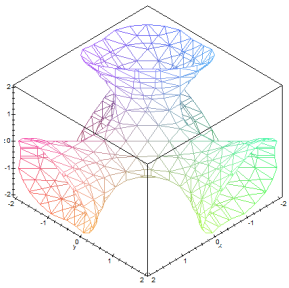
Ces ensembles sont tracés par la fonction **plotpoint3d(liste, options);**

Exemple :

```
> with(plots):
> points:={seq([cos(Pi*T/40),sin(Pi*T/40),T/40],T=0..40)}:
  pointplot3d(points,axes=BOX, symbol=CIRCLE, color=black);
```

On peut aussi tracer une surface à partir de points par la fonction **surfdata(liste,options);**

```
cosdata := [seq([ seq([i,j,evalf(cos((i+j)/5))], i=-5..5)], j=-5..5)]:
sindata := [seq([ seq([i,j,evalf(sin((i+j)/5))], i=-5..5)], j=-5..5)]:
surfdata( {sindata,cosdata}, axes=frame, labels=[x,y,z] ,style=patch);
```



Autres représentations qui peuvent être très utiles dans le cadre de simulation seront sans doute la représentation de champs de vecteurs ou de gradients.

- Le champ de vecteurs : il est noté [f(x,y,z),g(x,y,z),h(x,y,z)]. Le tracé est obtenu grâce à **fiedplot3d(fonction,domainex,domainey,domainez,options);** Le graphe est un ensemble de vecteurs évalués en des points (x,y,z) dont la direction et la norme sont données par f, g et h.

Exemple de champ de vecteurs :

```
> fieldplot3d([2*x,2*y,1],x=-1..1,y=-1..1,z=-1..1,
  grid=[5,5,5], axes=BOX);
```

- le champ de gradient: Maple trace grâce à la fonction **gradplot3d(fonction,domainex, domainey, domainez,options);** un champ de vecteurs dérivant d'un potentiel donné.

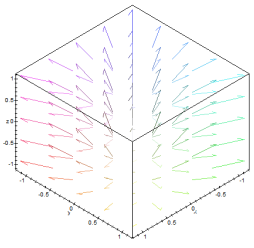
  Exemple de champ de gradients :

```
> gradplot3d(
  (x^2+y^2+z^2+1)^(1/2),x=-2..2,y=-2..2,z=-2..2
  ,axes=BOX);
```

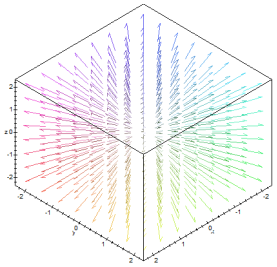
- La fonction implicite: on utilise la fonction dédiée **implicitplot3d(fonction, domainex,domainey,domainez,options);** pour tracer une fonction définie de manière implicite (équation, procédure…). Par défaut, la fonction est évaluée en 1000 points (10 sur x,10 sur y et 10 sur z dans les domaines définis.

Exemples de tracé de fonction implicite :

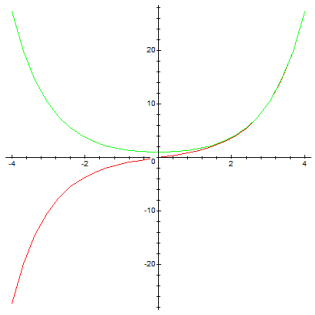
```
> implicitplot3d( x^3 + y^3 + z^3 + 1 = (x + y
  + z + 1)^3,x=-2..2,y=-2..2,z=-2..2,
  grid=[13,13,13], axes=BOX);
```



II.3 Tracés des solutions des équations différentielles



L'une des sources des données à tracer serait sans doute les solutions des équations différentielles. En effet, la modélisation mathématique des phénomènes physiques est souvent ramenée à un problème d'équations différentielles à résoudre. La représentation graphique des solutions devient très importante au point de permettre à l'analyste une meilleure vue de ses résultats.



II.3.1 tracé simple

Sous Maple, l'utilisateur peut tracer directement la solution d'une équation différentielle par l'instruction « **odeplot()** ». Cette fonction fait partie de la bibliothèque « **plots** ». La procédure est simple :

1. résoudre numériquement l'équation différentielle,
2. affecter les résultats à une variable,
3. appeler la fonction **odeplot()** avec des options convenables

Exemple :

Soit un système d'équation différentielle « sys » définie comme ci-après :

```
> sys := diff(y(x),x)=z(x),diff(z(x),x)=y(x);
```

$$sys := \frac{\partial}{\partial x} y(x) = z(x), \frac{\partial}{\partial x} z(x) = y(x)$$

L'appel à l'instruction **dsolve()** et l'affectation des résultats à la liste p :

```
> p:= dsolve({sys,y(0)=0,z(0)=1},{y(x),z(x)},type=numeric);
```

$$p := \mathbf{proc}(rkf45\_x) \ \ldots \ \mathbf{end}$$

Les résultats sont maintenant chargés dans la liste p, il suffit d'appeler l'instruction **odeplot()** argumentée selon le besoin.

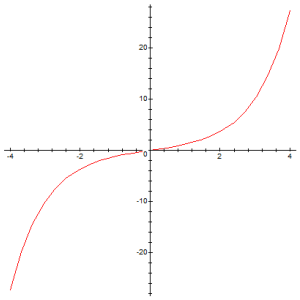
Le tracé d'une seule solution y(x) quand -4<=x <=4.

```
> with(plots):
>
> odeplot(p, [x,y(x)], -4..4,numpoints=25);
```

Le tracé de deux fonctions y(x) et z(x), solution du système d'équations différentielles, pour -4<=x<=4

```
> odeplot(p, [[x,y(x)],[x,z(x)]],-4..4,numpoints=25);
```

II.3.2 la bibliothèque DEtools



Maple dispose d'une bibliothèque dédiée au tracé de solutions d'équations différentielles ou de système d'équations différentielles. Cette bibliothèque est appelé par « **with(DEtools):** » Parmi ses fonctions nous avons :

a) la fonction DEplot:

Son rôle est de tracer soit des solutions d'équations différentielles d'ordre n du type diff(y(x),x$n)=f(x,y) où f(x,y) peut contenir des dérivées de y, soit de

système de deux équations différentielles d'ordre 1 du type :x'=f(t,x,y) y'=g(t,x,y). La syntaxe est la suivante:

DEplot(diffeq,vars,dv,cond.init.,options);

Les paramètres sont décrits comme suit :

- **diffeq:** équations à résoudre. Si le système est linéaire, on peut se contenter d'entrer la matrice des coefficients (V'=M.V où V=[x,y])
- **vars** : spécification des variables dans l'ordre, variable indépendante puis variables dépendantes [t,x,y] ou [x(t),y(t)] par exemple. Si le système admet une solution explicite y(x), il vaut mieux d'entrer les variables sous la forme [x,y] (accélération de la résolution).
- **dv**: domaine de variation de la variable indépendante,
- **cond.init**.: ce sont les conditions initiales sous la forme [t0,x0,y0] ou [x(t0)=x0, y(t0)=y0].Si un jeu de conditions est donné, Maple trace une solution par condition. Par défaut, les conditions initiales sont prises nulles.
- **Options** : il y en a de nombreuses. Les plus importantes sont : **stepsize=h** qui fixe le pas d'intégration (domaine/20 par défaut), **method=nom** (euler, backeuler, impeuler ou rk4 / par défaut rk4 pour Range Kutta d'ordre 4) qui fixe la méthode d'intégration.

b) la fonction DEplot1:

Son rôle est de tracer les solutions d'équations différentielles d'ordre 1 du type y'=f(t,y). Sa syntaxe est :

DEplot1(équadiff,vars,dv,cond.init.,dy,options);

Elle se présente comme la fonction précédente. Les différences figurent dans les cas suivants :

- **dy** : on peut en option limiter le domaine de variation de la variable dépendante.
- **options** : il y a ici une option supplémentaire: **limitrange=true** ou false qui impose l'arrêt de l'intégration lorsqu'on sort du domaine fixé pour y.

c) la fonction dfieldplot:

Cette fonction recouvre partiellement ce que fait DEplot2. En effet, elle fournit la cartographie sous forme de vecteurs de **dy/dx** où y et x sont obtenus par résolution d'un système différentiel d'ordre 1 ou par une équation différentielle d'ordre 1. La syntaxe est :

dfieldplot(équations,vars,dt,options);

d) la fonction phaseportrait:

Cette fonction recouvre aussi partiellement l'action de DEplot2. En effet, étant donné un système de deux équations différentielles d'ordre 1, elle fournit le tracé de y(x) ou à défaut le tracé en 3D de x(t) et y(t). La syntaxe est :

phaseportrait(équations,vars,dt,cond.init.,options);

Chapitre VI : Eléments du langage de programmation

Maple est un système de calcul formel et numérique. Il offre à l'utilisateur un environnement de travail simple à utiliser mais aussi très avancé. L'utilisateur n'a qu'à choisir son mode d'utilisation :

- En mode ligne de commande,
- En mode programmation.

Pour le deuxième mode, Maple offre un langage semblable à de nombreux langages de programmation structurés. Les éléments de base du langage sont déjà présentés dans les chapitres précédents. C'est que la définition des variables, l'appel des fonctions prédéfinies, la représentation des nombres et des expressions mathématiques, les conventions de syntaxes restent les mêmes. L'objectif du présent chapitre est de présenter un complément à tout ce qui précède pour pouvoir utiliser la programmation sous Maple.

L'écriture d'un bloc de programme se fait directement sur une feuille de calcul. Autrement le programmeur peut écrire son code à l'aide d'un éditeur de code, puis par copier/coller peut l'importer dans son environnement Maple.

Nous avons vu que les lignes de commandes sont exécutées une fois la touche RETURN est tapée. Lors de l'édition du code, il faut taper les deux touches combinées « Shift + RETURN » pour passer à la ligne suivante du code sans que celui-ci soit exécuté.

I Branchement conditionnel : if … else … fi

Branchements conditionnels : bloc if

Les blocs conditionnels **if** ont une structure semblable à tous ceux des autres langages de programmation. Le mot clé elif est une contraction de "else if" qui signifie "sinon si". Comme if, elif est suivi d'une condition logique puis de then. Un bloc if se ferme avec end if (ou fi) et tout bloc ouvert doit être fermé.
Deux blocs if peuvent être imbriqués (inclus) l'un dans l'autre mais ne doivent pas se "chevaucher". Seuls les éléments if…then…end if; sont indispensables. Les autres éléments de structure else et elif ne seront utilisés qu'en fonction des besoins. Comparer en analysant les résultats suivants :

```
> x:=-2:
  if x > 1 then
        y:=x-1;
  elif x < -1 then
        y:=x+1;
  else
        y:=0;
  fi;
```

$$y := -1$$

Dans l'exemple ci-dessus, le bloc d'instruction est fermé par **« fi ; »** et non pas par **« end if ».** Les anciennes versions n'admettent pas la dernière instruction.

Remarque :

L'évaluation d'une condition, « x>1 » par exemple se fait pour des valeurs de type numériques ou caractères et non pas pour des constantes.

```
> x:=sqrt(3):
  if x > 1 then
      y:=0;
  fi;
Error, cannot evaluate boolean
```

```
> x:=sqrt(3):
  if is(x > 1) then
      y:=0;
  fi;
```

$y := 0$

```
> x:=sqrt(3):
  if evalf(x) > 1 then
      y:=0;
  fi;
```

$y := 0$

Les expressions ci-dessus montrent que celle à gauche n'est pas évaluée, et un message d'erreur est émis. C'est x=sqrt(3) ne peut pas être comparé à 1. Pour remédier à ce problème, Maple agit selon deux cas : utiliser la fonction **is()** qui force la comparaison et renvoie le résultat ou utiliser la fonction evalf(x) pour évaluer d'abord la variable x ensuite la comparer à 1 avant d'envoyer le résultat.

On notera également que « if » peut être vu comme une fonction, ce qui conduit à une écriture simplifiée pour une condition « if-else ».

Exemple :

```
> x:=3:
  z:=`if`(x>2,x+Pi,x-Pi);
  x:=1:
  z:=`if`(x>2,x+Pi,x-Pi);
```

$z := 3 + \pi$

$z := 1 - \pi$

Les branchements conditionnels peuvent être imbriqués, c'est à dire inclus l'un dans l'autre comme le montre l'exemple suivant :

```
> a:=2: b:=3:
  if (a<b) then
      if (b<5) then
          test := 'vrai';
      fi:

  fi;
> test;
```

vrai

```
> a:=2: b:=3:
  if (a<b) and (b<5) then
     test := 'vrai':
  fi:
> test;
```

vrai

Pour combiner plusieurs tests logiques et formuler un seul résultat il suffit d'utiliser les opérateurs logiques « and, or, not »

II Traitement répétitif : les boucles

Maple fournit deux types de boucles :

- for …. do … od :
- while….do …. od :

II.1 Structure de contrôle while .. do:

La structure globale de cette boucle est donnée par :

while condition do

…

od;

où condition : une évaluation d'une expression logique dont le résultat une valeur logique : vrai ou faux. L'expression logique peut contenir un test logique unique ou une combinaison de plusieurs tests unis par les opérateurs logiques « and, or, not »

Entre les deux opérateurs « do … od ; » le code de la boucle est écrit. Il peut contenir une ou plusieurs lignes selon le besoin.

Exemple : calcul de la somme des 1000 premiers entiers naturels

```
> somme :=0: k:=1:
  while is(k<1001) do somme :=somme + k : k:=k+1: od:
  somme;
                                                    500500
```

La boucle et son corps sont écrits sur la même ligne. C'est un cas particulier. En général, les boucles de calcul sont écrites sur plusieurs lignes.

II.2 Structure de contrôle for … from …to … by … do … od:

La boucle for est utilisée quand un compteur est utilisé pour compter les itérations variant entre une borne initiale et une finale. Par exemple, parcourir un tableau, une matrice etc …

for variable **from** initiale **to** finale **by** pas **do**
…
do;

où

- variable désigne la variable qui servira de compteur,
- Initiale, finale sont les bornes initiale et finale de la boucle,
- Pas : le pas d'avancement de la variable compteur.

Exemple : calcul de la somme des 100000 premiers entiers naturels.

```
> somme:=0:
  for i from 1 to 100000 by 1 do
>     somme:= somme+i:
> od:
  somme;
                                                5000050000
```

Le changement du paramètre « pas » permet de sauter certain cas. Par exemple, pour calculer la somme des nombres pairs seulement de l'exemple ci-dessus, il suffit de changer le paramètre « pas » dans le code :

```
> somme:=0:
  for i from 2 to 100000 by 2 do
>     somme:= somme+i:
> od:
  somme;
                                                2500050000
```

La boucle for peut s'écrire sous une autre forme. Au lieu d'incrémenter un compteur, elle peut être choisie parmi une liste de valeurs. La syntaxe est la suivante :

for variable **in** liste **do**

..
od;

Exemple : Générer un vecteur aléatoire puis afficher ses composants un à un.

```
> with(linalg):
Warning, new definition for norm
Warning, new definition for trace
> v:=randvector(6);
```

$$v := [79, 56, 49, 63, 57, -59]$$

```
> for x in v do
>     print(x);
> od:
```

$$[79, 56, 49, 63, 57, -59]$$

II.3 Contrôle du déroulement d'une boucle

Si l'on souhaite arrêter une boucle sur une condition on peut utiliser le mot clé **break** (casser). Ainsi la somme suivante est arrêtée dès qu'un élément de la liste est négatif ou nul. On introduit dans cette boucle la notion de bloc conditionnel if...end if qui sera détaillée plus loin dans ce chapitre.

On peut aussi sauter des instructions d'une boucle sur une condition : on utilisera le mot clé **next** (suivant). La boucle reprend alors avec la première instruction de la boucle avec la valeur suivante du compteur. La somme ne porte ici que sur les logarithmes des valeurs positives de la liste.

Une boucle peut se construire avec seulement les mots clés do et end do mais l'utilisateur doit alors gérer lui-même le comptage de la boucle et l'utilisation du mot clé break est impérative sous peine de voir la boucle s'exécuter indéfiniment...

III Les procédures et fonctions

Un programme MAPLE peut être organisé en sous-programmes appelées procédures. Une procédure, de type « procédure » , est définie par le mot-clé **proc** et peut être assignée à un nom de variable. Pour définir une procédure intitulée « nom_proc » , les lignes de code sont utilisées à syntaxe suivante:

- Nom_proc := proc (paramètres_formels)
- global variables_globales; (*la ligne de déclaration de variables globales : optionnelle)*
- local variables_locales; (*la ligne de déclaration des variables locales : optionnelle*)
- description chaîne_de_description; (*la ligne description est optionnelle*)
- option nom_option; (la ligne option est optionnelle)
- . . . instructions . . . (corps de la procédure)
- end proc;

Les variables globales d'une procédure sont des variables qui gardent leur définitions même en dehors du code de la procédure. Celles qui sont locales n'ont un sens que dans son code propre.
La description ci-dessus est la forme la plus générale de la définition d'une procédure selon les versions récentes de Maple. Toutefois, il est tout à fait correct d'établir du code simplifié pour représenter une procédure comme le montre l'exemple suivant.

```
> maximal:=proc(u,v)
        if is(u<v) then v else u fi;
  end:

> maximal(sqrt(3),4);
                                                    4
```

La procédure ne rend pas explicitement un résultat. Ce sont les fonctions qui font ce travail. Sous Maple, une fonction c'est une procédure qui renvoie son résultat grâce à l'instruction **RETURN().** Dans l'exemple suivant, la fonction **nommée « ma_fonction »,** calcule la quantité n x (n-2) x (n – 4) … 1 (ou 2).

```
> ma_fonction:=proc(N)
> local r, i:
> r:=1;
> for i from N by -2 while i>0 do r:=r*i: od:
> RETURN(r):
> end:
>
> ma_fonction(11);
                          10395
```

Partie II : Simulation

« Simuler pour comprendre et anticiper. »

« Dans la logique de faire plus vite, mieux… et moins cher, la simulation numérique présente tous les atouts. »

"Loin de supplanter l'expérimentation, la simulation donne une nouvelle prise sur le réel."

[1] *Institut de recherche sur les lois fondamentales de l'Univers*
http://irfu.cea.fr/Projets/coast_documents/communication/Simulation.pdf

Chapitre VII. Simulation numérique et au calcul scientifique

I Introduction

La simulation par ordinateur progresse avec le développement du matériel informatique. Depuis l'apparition des premiers ordinateurs, de nombreuses méthodes se sont inventées pour résoudre des problèmes restaient jusqu'à alors sans solutions.

Aujourd'hui, la simulation numérique est utilisée dans de nombreux domaines de recherche et développement : mécanique, science des matériaux, astrophysique, physique nucléaire, aéronautique, climatologie, météorologie, physique théorique, mécanique quantique, biologie, chimie… ainsi qu'en sciences humaines : démographie, sociologie… Elle intervient aussi dans des secteurs comme celui de la banque et de la finance [i] .

La simulation numérique désigne le procédé selon lequel on exécute des programmes sur des ordinateurs en vue de représenter un phénomène physique. Les simulations numériques scientifiques reposent sur la mise en œuvre de modèles théoriques. Elles sont une adaptation aux moyens numériques des modèles mathématiques. Elles servent à étudier le fonctionnement et les propriétés d'un système et à en prédire son évolution. La simulation comprend deux phases : la modélisation et l'implémentation.

II. La modélisation

Un modèle est la traduction d'un phénomène naturel quelconque en une description par des équations mathématiques. A partir du 17ème siècle, les modèles se sont souvent inspirés des expériences menées en laboratoire. Ils sont parfois déduits des théories elles-mêmes conçues sans expérience, grâce à une démarche purement intellectuelle, l'« expérience de pensée » (la Théorie de la relativité conçue par Albert Einstein en1915 n'a pu être expérimentée qu'en1919).

Aujourd'hui, plusieurs théories physiques, comme la théorie des cordes, fournissent des modèles qu'ils n'ont pas encore de procédés expérimentaux qui pourront les tester. L'« expérience numérique» sert à éclaircir l'analogie entre la pratique et la conduite d'une expérience de physique. La modélisation du phénomène étudié consiste à prendre en compte les principes fondamentaux, comme par exemple la conservation de la masse, de l'énergie, et à déterminer les paramètres essentiels à sa description à la fois simple et réaliste. En chaque point de l'objet considéré, plusieurs grandeurs physiques (position, vitesse, température…) décrivent son état et son évolution et permettent de caractériser entièrement son mouvement. Ces grandeurs ne sont pas indépendantes mais reliées entre elles par des équations, qui sont la traduction mathématique des lois de la physique régissant le comportement de l'objet.

III. L'implémentation

Après la modélisation, qui produit les équations mathématiques décrivant le phénomène naturel, il reste la traduction de ces équations en des programmes informatiques pour que la machine les exécute. Or, la machine ne reconnait que du numérique et par équivalence du discret !

Par ailleurs, simuler l'état de l'objet revient à déterminer les valeurs numériques de ses paramètres en tous points. Comme il y a une infinité de points, il serait impossible de les traiter ensemble. Pour des raisons de faisabilité, il n'est possible de considérer qu'un nombre fini de points. Le nombre effectif de points traités dépendra de la puissance de ce dernier.

La discrétisation du domaine physique consiste précisément dans cette réduction de l'infini au fini. La modélisation et la simulation vont toujours ensemble. Elles s'appuient sur des méthodes mathématiques et informatiques en même temps.

IV. Les méthodes de calcul

Il existe deux approches principales pour résoudre numériquement les équations mathématiques d'un modèle :

- La méthode de calcul déterministe, qui résout les équations après avoir discrétisé les variables,
- La méthode de calcul statistique (ou probabiliste).

Dans la première, l'objet est considéré comme un ensemble de petits volumes élémentaires contigus dénommé « maillage », par analogie avec la trame d'un tissu. Les paramètres de l'état de l'objet définis dans chaque maille du maillage sont reliés par des équations algébriques à ceux des mailles voisines. Ce sera à l'ordinateur de résoudre le système de relations obtenu. Il existe de nombreuses méthodes déterministes:

- des volumes finis,
- des éléments finis,
- level set…

dont l'utilisation dépend pour partie des équations considérées.

La deuxième méthode, dite de «Monte-Carlo », est particulièrement adaptée aux phénomènes caractérisés par une succession d'étapes lors desquelles chaque élément de l'objet peut subir différents événements possibles apriori. D'étape en étape, l'évolution de l'échantillon sera déterminée grâce à des tirages au hasard (d'où le nom de la méthode).

Les méthodes déterministes et de Monte-Carlo font l'objet de nombreuses études mathématiques pour préciser leur convergence en espace, c'est-à-dire la variation de la précision de l'approximation en fonction du nombre de mailles ou d'éléments de l'objet, ou leur convergence en temps, soit la variation de la précision en fonction du « pas de temps » de calcul.

Les outils de la simulation numérique sont donc des programmes informatiques exécutés sur des ordinateurs: ces logiciels de calcul ou « codes » sont la traduction, à travers des algorithmes numériques, des formulations mathématiques des modèles physiques étudiés.

En amont et en aval du calcul, les logiciels effectuent la gestion de nombreuses opérations complexes de préparation puis de dépouillement des résultats. Les données initiales comportent la délimitation du domaine de calcul à partir d'une représentation approchée produite par le dessin et la CAO (conception assistée par ordinateur) des formes géométriques, suivie de la discrétisation sur un maillage, ainsi que des valeurs des paramètres physiques de ce maillage.

Les résultats des calculs proprement dits sont sauvegardés au fur et à mesure afin de constituer une base de données numérique. L'analyse des résultats repose sur l'exploitation de cette base: extraction sélective et transfert vers des interfaces graphiques qui permettent la visualisation par images de synthèse.

La simulation numérique se concrétise lorsque l'on visualise le phénomène initial sur l'écran de l'ordinateur. L'analyse critique des résultats, la vérification de la validité des modèles théoriques utilisés, la confrontation avec l'expérience doivent notamment faire partie intégrante de la démarche. Cette analyse comparative débouche sur des améliorations des modèles physiques, de leurs paramètres et des programmes informatiques de simulation.

V. L'expérience et la simulation

L'expérience alimente la simulation. Inversement, l'exploration des nombreuses solutions rendue possible par la simulation permet d'observer ou de prévoir des comportements inattendus, ce qui parfois suggère des expériences et fait donc progresser la connaissance de la physique. Ainsi, loin de supplanter l'expérimentation, la simulation donne une nouvelle prise sur le réel. La simulation numérique est la troisième forme d'étude des phénomènes, après la théorie et l'expérience.

On la qualifie d'ailleurs souvent d'« étude in silico », le silicium étant le matériau de base des ordinateurs. Le modèle prédictif et la simulation qui l'accompagne permettent d'anticiper le futur d'un système ou le comportement qui serait celui de ce système dans une configuration dans laquelle il ne s'est jamais trouvé. Prédire des situations inédites est l'un des intérêts essentiels de la simulation numérique.

Les limites de la simulation numérique sont de trois ordres :

1. Certains phénomènes sont encore mal compris. Ils sont donc difficilement traduisibles en équations, qui sont le seul moyen de « dialoguer » avec l'ordinateur.

2. Certains modèles très compliqués nécessitent des puissances de calcul indisponibles actuellement.

3. Le nombre d'opérations nécessaires à la résolution d'un modèle croît exponentiellement en fonction du degré de précision que l'on demande, certains modèles mathématiques ne peuvent pas être résolus par un ordinateur en un temps raisonnable.

VI. Chronologie historique

- 1939-1945 : La simulation numérique est apparue en même temps que l'informatique pour les besoins du projet Manhattan pendant la Seconde Guerre mondiale, afin de modéliser le processus de détonation nucléaire. Depuis, elle a évolué parallèlement à l'informatique.

- 1944 Colossus est une autre machine destinée au calcul intensif. Elle « cassait » les codes secrets nazis. Mais il s'agit d'une calculatrice spécialisée difficilement programmable.

- 1948 Naissance de Baby, le premier véritable ordinateur puisqu'il respecte le principe de programme enregistré en mémoire centrale.

- 1950 L'ordinateur devient un produit : 45 exemplaires de l'Univac1 de Konrad Zuse sont livrés. En 1955, l'IBM 704, premier ordinateur commercial scientifique avec virgule flottante, apparaît aux États-Unis.

- 1969 L'architecte de CDC, Seymour Cray, lance le CDC 7600, dix fois plus puissant. L'air pulsé ne suffisant plus, ses modules sont refroidis par circulation d'un liquide.

- 1972 Création de Cray Research, qui se donne pour objectif de construire l'ordinateur le plus puissant du monde. Le Cray 1 devient le supercalculateur vectoriel dès 1975.

- 1982 Arrivée de son successeur multiprocesseur le Cray XMP, puis du Cray 2 en 1985. Celui-ci est si puissant qu'il est plongé directement dans un liquide réfrigérant : c'est le premier ordinateur aquarium.

- 1989 : CDC reprend la première place avec l'ETA 10. Des offensives viennent du Japon, la notion de « parallélisme » apparaît, donnant lieu à de multiples prototypes d'Intel, nCube, BBN, Meiko… Après une phase de prolifération, la sélection naturelle fait son oeuvre.

- Après les années 90 : Le parallélisme se banalise, le processeur vectoriel recule devant le microprocesseur. Mais on voit aussi se multiplier des solutions mixtes réunissant, via un réseau d'interconnexions par messages, des nœuds constitués de plusieurs processeurs couplés par mémoire partagée.

Chapitre VIII. La simulation par la méthode de Monte-Carlo

I. Introduction

La méthode de simulation de Monte-Carlo est conçue pour calculer une valeur numérique en se basant sur des procédés aléatoires. Le nom de ces méthodes fait allusion aux jeux de hasard pratiqués à Monte-Carlo. La première version de son algorithme est décrite dans un article de 1953 par Nicholas Metropolis[ii].

C'est une méthode d'approximation. Il n'y a pas de définition précise pour la technique. La description la plus proche se résume comme suit: La méthode consiste à étudier le comportement d'un nombre limité *n* d'éléments d'un ensemble de *N* éléments, tirés aléatoirement, pour évaluer les caractéristiques de l'ensemble en entier. L'erreur, la différence entre la valeur estimée et celle calculée, est une suite qui converge normalement vers zéro quand le nombre $n \to N$.

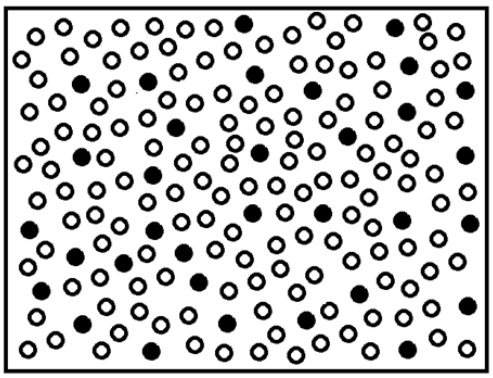
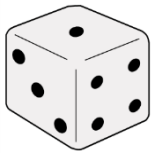


Figure 1

La figure (Figure 1) illustre un exemple concret du principe de la méthode de Monte-Carlo. L'étude réelle du système consiste à étudier le comportement de chacune de ses particules ainsi que les interactions mutuelles entre les particules et entre les particules et les parois. Dans des situations physiques réelles, le nombre des particules est très grand voir inimaginable. Par conséquent, il est impossible d'étudier le comportement de chaque particule. Dans ce cas, pourquoi ne pas choisir un nombre limité de particules (en noir) pour les étudier ?

La réponse est affirmative. Sauf, le problème devient comment choisir les particules. Les lois des statistiques présentes plusieurs méthodes d'échantillonnage. A présent, il suffit de choisir aléatoirement ces particules. Ainsi, le système qui contient initialement *N* particules est réduit à une représentation de *n* particules seulement où *n*<<*N* .

II. Premier exemple d'application de la méthode de Monte-Carlo



Considérons le jeu de tirage de Dé. Si les faces sont équiprobables alors la probabilité de chaque face est de 1/6. D'où, si nous réalisons un nombre N de tirage, le nombre de réalisation *n* d'une face précise (par exemple la face qui porte 3 points) vérifie la condition suivante :

$$\frac{n}{N} \to \frac{1}{6}$$

Cette tendance sera plus précise quand le nombre N est relativement grand. La figure 2 présente le code d'une fonction qui permet de réaliser l'exemple du dé (Figure 2.a) suivit des résultats de calcul pour différentes valeurs de N (Figure 2.b). La figure (Figure 2.c) illustre l'évolution du résultat calculé

en fonction du nombre d'itérations N. Il est clair que le résultat calculé a tendance à se confondre avec la valeur théorique (1/6) pour les grandes valeurs de N.

```
> MonteCarlo:=proc(N, n)
> local i, x, S;
>       randomize();
>       S:=0:
>      for i from 1 to N do
>         x:=rand(1..6);
>         if (x()=n) then
>             S:=S+1:
>         fi:
>     od:
>     RETURN (S/N):
> end:
```

Figure 2.a

```
> MonteCarlo(100,3): evalf(");          .20000000000
> MonteCarlo(500,3): evalf(");          .1700000000
> MonteCarlo(1000,3): evalf(");         .1570000000
> MonteCarlo(10000,3): evalf(");        .1714000000
> MonteCarlo(50000,3): evalf(");        .1662200000
> MonteCarlo(20000,3): evalf(");        .1721500000
> MonteCarlo(25000,3): evalf(");        .1702400000
> MonteCarlo(30000,3): evalf(");        .1669000000
> MonteCarlo(100000,3): evalf(");       .1644200000
> MonteCarlo(500000,3): evalf(");       .1660620000
> MonteCarlo(1000000,3): evalf(");      .1672310000
> MonteCarlo(1500000,3): evalf(");      .1664600000
```

Figure 2.b

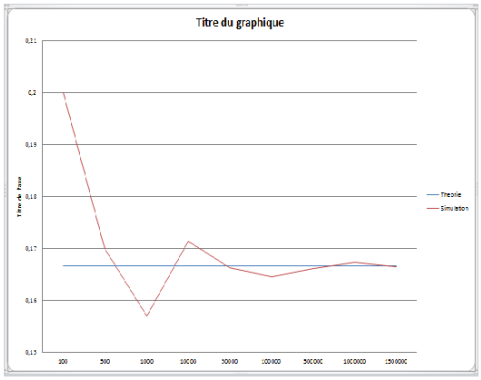
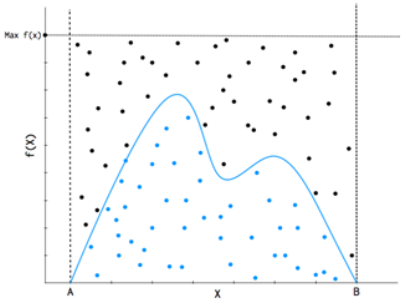


Figure 2.c

III. Deuxième exemple d'application : Calculer une intégrale

Parmi les nombreux exemples d'application de la méthode Monte-Carlo, le calcul d'intégrale se présente comme le plus simple à comprendre. En réalité, la valeur recherchée est plutôt estimée. Soit : $I = \int_A^B f(x)dx$, supposons que la fonction f(x) est continue et intégrable sur l'intervalle [A,B]. La quantité I représente la surface délimitée par les segments [A,B],[0, f(A)], [0,f(B)] et la courbe f(x) pour $x \in [A,B]$. Il est évident qu'une surface peut être subdivisée en une collection de surfaces élémentaires que nous pouvons considérer comme des points. Considérons la figure à droite, la surface du rectangle de largeur [A,B] et de hauteur [0, Max(f(x))] encadre la surface I recherchée.



Estimation de l'intégrale $\int_A^B f(x)dx$

Considérons une surface élémentaire s_0 que nous désignons comme un point. La surface totale du rectangle qui contient N points est $S_r = Ns_0$. La surface I qui contient n points est $S_I = ns_0$. Le rapport de ces deux surfaces est :

$$\frac{S_I}{S_r} = \frac{ns_0}{Ns_0} = \frac{n}{N}$$

D'où

$$S_I = \frac{n}{N} S_r$$

Comme il n'est pas possible d'examiner tous les points qui formes les surfaces S_r et S_I, il est très commode de choisir un nombre N limité de ces points, tirés aléatoirement, et de compter ceux qui vont appartenir à la surface S_I pour former le nombre n. Cependant, l'estimation de l'intégrale $\int_A^B f(x)dx$ est donnée comme suit :

$$I = S_I \approx \frac{n}{N} S_r$$

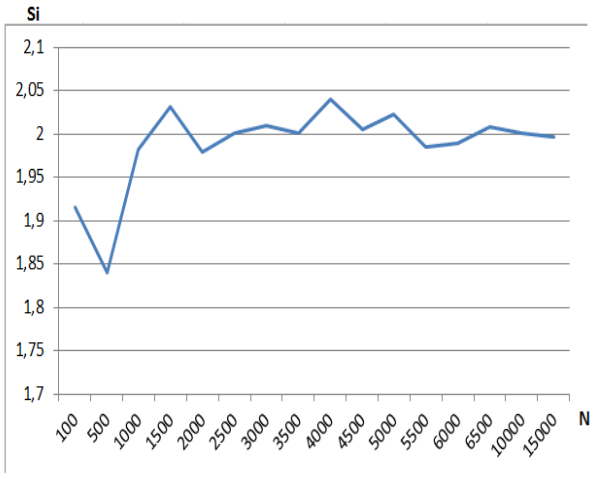
Avec $S_r = Max(f(x)) \times (B - A)$.

Algorithme :

1. Fixer un nombre N,
2. Initialiser $n \leftarrow 0$,
3. Pour i allant de 1 à N,
4. Choisir un point aléatoirement, c-à-d, x et y deux valeurs aléatoires, telles que :
   - o $A \leq x \leq B$,
   - o $0 \leq y \leq Max(f(x))$
5. Si $y \leq f(x)$ alors $n \leftarrow n + 1$
6. Fin pour
7. $I = S_I \approx \frac{n}{N} S_r$

Exemple d'application : calculer $I = \int_0^{\pi} sin(x)dx$

```
> Integral := proc(N)
> local i, Si, Sr, n,x ,y, X, Y, R:
> randomize():
> Sr:=1*Pi:
> n:=0:
> R:=1000000:
> for i from 1 to N do
>
>     x:=rand(0..R): y:=rand(0..R):
>     X:= evalf(x()*Pi/R): Y:=evalf(y()/R):
>     if is(Y<=sin(X)) then
>        n:=n+1:
>     fi:
> od:
> Si:=evalf((n/N)*Sr):
> RETURN (Si)
> end:
```



Dans les figures ci-dessus, à gauche le programme Maple, et à droite la valeur calculée de l'intégrale en fonction du nombre de points choisis.

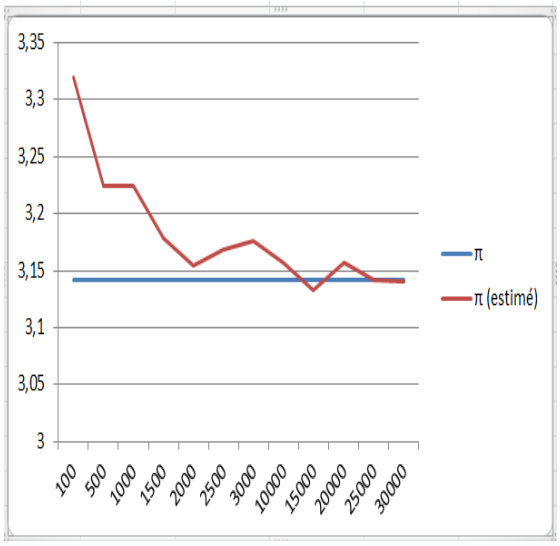
IV. Troisième exemple d'application : Estimer la valeur de π

Un autre exemple de l'application du calcul d'intégrale via la méthode de Monte-Carlo se voit l'estimation de la valeur de π. Comme dans le cas du deuxième exemple, la méthode de calcul consiste à choisir des points aléatoires dans le carré, ceux qui appartiennent au quart du cercle seront comptés.

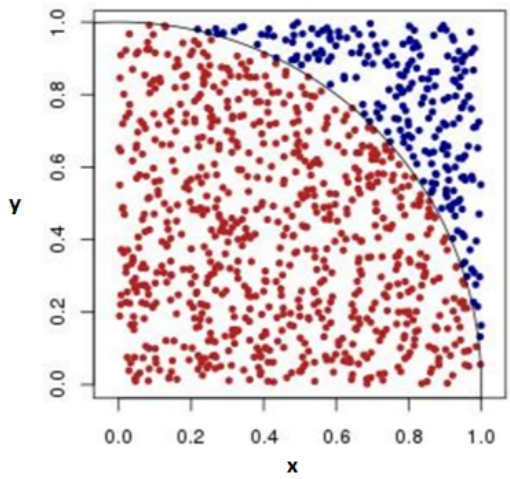
La surface S_c du quart du cercle est $S_c = \frac{\pi}{4}$. En appliquant la méthode de Monte-Carlo selon l'algorithme présenté pour le deuxième exemple, nous pouvons déduire :

$$\pi \approx 4\frac{n}{N}$$

```
> calculer_Pi := proc(N)
> local i, n, Sr, Si, r, R, x,y, X,Y :
> randomize():
> Sr:=1:
> n:=0:
> R:=100000000:
> for i from 1 to N do
>
>     x:=rand(0..R): y:=rand(0..R):
>     X:= evalf(x()/R): Y:=evalf(y()/R):
>     r:=evalf(sqrt(X^2+Y^2));
>     if r<=1 then
>        n:=n+1:
>     fi:
> od:
> Si:=evalf(4*(n/N)*Sr):
> RETURN (Si)
> end:
```



Il est clair, que la valeur estimée converge asymptotiquement à la valeur réelle quand le nombre de points considérés est élevé.



Chapitre IX. Méthodes de Monte-Carlo en Physique Statistique

I. Introduction

Ce document constitue une introduction aux méthodes de Monte-Carlo appliquées à la physique statistique. Les méthodes sont exposées à partir d'un exemple, le modèle d'Ising du ferromagnétisme.

II. Modèle d'Ising

II.1 Définition

Les matériaux ferromagnétiques (comme le fer ou le nickel) tiennent leurs propriétés magnétiques du spin des électrons libres (responsables aussi de la conduction électrique). Le spin est un moment cinétique intrinsèque de l'électron, auquel est associé un moment magnétique. Le spin est quantifié. La projection du moment cinétique de spin sur un axe Z n'a que deux valeurs possibles :

$$S_z = \pm\frac{\hbar}{2} \tag{1}$$

où $\hbar$ est la constante de Planck divisée par 2π. Le moment magnétique associé au spin est proportionnel au moment cinétique :

$$m_z = -g\mu_B S_z \tag{2}$$

où g est le facteur de Landé (égal environ à 2) et μ_B le magnéton de Bohr. En présence d'un champ magnétique **B**, l'énergie de l'électron est :

$$E = -m_z B_z \tag{3}$$

Les moments magnétiques ont donc tendance à s'orienter dans le sens du champ.

Dans un matériau ferromagnétique, les spins ont la faculté de s'orienter dans le même sens, même en l'absence de champ magnétique extérieur. C'est pourquoi un morceau de fer peut présenter une aimantation permanente. Pour expliquer ce phénomène, il faut une interaction entre les spins.

Deux électrons peuvent interagir par l'interaction dipolaire : le champ magnétique créé par le moment magnétique d'un électron exerce une influence sur le moment magnétique de l'autre électron. Cependant, l'interaction dipolaire est beaucoup trop faible pour expliquer le ferromagnétisme.

C'est un phénomène quantique qui explique l'interaction entre deux spins responsable du ferromagnétisme. Le principe d'exclusion de Pauli et la répulsion électrique entre les deux électrons, conduisent à une interaction appelée interaction d'échange. Pour deux électrons voisins *i* et *j*, cette interaction peut être représentée par une énergie de la forme :

$$E = -JS_z^i S_z^j \tag{4}$$

où J est une constante positive. Deux électrons voisins ont une énergie plus basse lorsque leurs spins sont de même sens. Ils ont donc tendance à s'orienter dans le même sens.

Le modèle d'Ising est une représentation simplifiée du matériau ferromagnétique, dans lequel la direction des spins (axe Z) est la même pour tous les électrons. Les spins sont positionnés sur un réseau. On se limite ici au modèle d'Ising à une dimension. La figure suivante montre un exemple de configuration pour un système à N spins. Les deux valeurs possibles pour chaque spin sont -1 et 1.

1	1	1	-1	1	-1	-1	……..	1	1	-1	1	1
N-1	0	1	2	3	4	5		N-5	N-4	N-2	N-1	0

Le spin d'indice i interagit avec ses deux voisins i-1 et i+1. On introduit une condition aux limites périodique, consistant à faire interagir les spins 0 et N - 1. Avec ces hypothèses, l'énergie du système s'écrit :

$$E = -\sum_{i=0}^{N-1} BS_i + S_i S_{i+1} \quad (5)$$

où B est une constante représentant le champ magnétique extérieur. Par convention, le moment magnétique est dans le même sens que le spin (physiquement, ils sont de sens opposé car l'électron a une charge négative).

II.2. Distribution de Boltzmann

En raison de l'agitation thermique, les électrons itinérants échangent en permanence de l'énergie avec les atomes du métal et les autres électrons. L'agitation thermique a tendance à maintenir un état désordonné des spins. Un résultat important de la théorie statistique est *la distribution de Boltzmann*, valable si le système est à l'équilibre thermique à la température T. Soit μ une configuration particulière du système de spins et E_μ l'énergie de cette configuration. La probabilité de cette configuration est proportionnelle à un facteur exponentiel (facteur de Boltzmann) :

$$p_\mu = \alpha e^{\left(-\frac{E_\mu}{kT}\right)} \quad (6)$$

où k est la constante de Boltzmann. Lorsque la température est faible, les configurations de faible énergie sont beaucoup plus probables que les configurations de plus haute énergie. Lorsque la température augmente, l'énergie a de moins en moins d'influence sur la probabilité. À température infinie, toutes les configurations sont équiprobables. Pour simplifier les notations, on pose :

$$\beta = \frac{1}{kT} \quad (7)$$

Pour déterminer la constante de normalisation α, il faut écrire la condition de normalisation: la somme des probabilités de toutes les configurations doit être égale à 1. On obtient ainsi :

$$p_\mu = \frac{e^{(-\beta E_\mu)}}{\sum_\mu e^{(-\beta E_\mu)}} \quad (8)$$

Cette distribution permet en principe de calculer la moyenne (au sens d'espérance), de différentes grandeurs physiques. Soit X une grandeur physique et X_μ sa valeur pour la configuration μ. La valeur moyenne de X, c'est-à-dire l'espérance de la variable aléatoire X, est :

$$\bar{X} = E(X) = \frac{\sum_\mu X_\mu e^{(-\beta E_\mu)}}{\sum_\mu e^{(-\beta E_\mu)}} \quad (9)$$

Dans le cas du modèle d'Ising, une grandeur intéressante est le moment magnétique total, défini par:

$$m_\mu = \sum_i S_i \quad (10)$$

Le moment magnétique est une grandeur macroscopique mesurable. La valeur mesurée est précisément la moyenne définie ci-dessus.

III. Méthodes de Monte-Carlo

III.1. **Principe**

Il s'agit d'évaluer l'espérance (9). Même lorsque le nombre de configurations est fini, cette somme ne peut être calculée directement car ce nombre est beaucoup trop grand. Par exemple pour le modèle d'Ising, le nombre de configurations est 2^N. Même pour un petit système de N = 100 spins, le nombre de configurations à explorer dépasse de très loin les capacités d'un ordinateur.

Il faut donc s'orienter vers une méthode de Monte-Carlo, qui consiste à opérer sur un échantillon de configurations choisies aléatoirement.

III.1. **Échantillonnage direct**

Une première idée serait de choisir des configurations aléatoirement, toutes ayant la même probabilité. Dans le cas du modèle d'Ising, ce tirage ne pose pas de difficultés. Il suffit de choisir aléatoirement chaque spin (-1 ou 1) pour obtenir une configuration.

Si M est le nombre de configurations tirées, l'espérance (9) est évaluée par la moyenne empirique :

$$moy(X, M) = \frac{1}{M}\sum_{k=1}^{M} p_k X_k \quad (11)$$

où p_k est la probabilité de la configuration k.

Cette méthode est cependant impossible à réaliser, car la somme figurant dans l'expression de p_μ n'est pas connue a priori. C'est justement ce type de somme que l'on cherche à calculer.

La solution consiste à tirer aléatoirement les échantillons avec la probabilité p_μ et non pas avec une probabilité uniforme. De cette manière, l'évaluation de l'espérance devient :

$$moy(X, M) = \frac{1}{M}\sum_{k=1}^{M} X_k \quad (12)$$

La difficulté est d'avoir une méthode pour échantillonner les configurations avec la probabilité p_μ. Les différentes méthodes d'échantillonnage de nombres aléatoires sont exposées dans

la méthode du rejet ne nécessite pas la connaissance de la constante de normalisation. Elle consiste à tirer aléatoirement une configuration μ avec un tirage uniforme, puis à calculer sa probabilité non normalisée :

$$q_\mu = e^{(-\beta E_\mu)} \quad (13)$$

La seconde étape consiste à tirer un nombre réel aléatoire x avec une densité uniforme sur l'intervalle $[0; q_{max}]$. Dans le cas présent, la valeur maximale de la probabilité non normalisée est $q_{max} = 1$. Ce nombre x est comparé à q_μ, ce qui permet de conserver ou de rejeter la configuration. Si $x < q_\mu$, la configuration est retenue, sinon elle est rejetée.

Pour le modèle d'Ising, la méthode du rejet est donc constituée des étapes suivantes :

- Choisir aléatoirement les N spins, les valeurs -1 et +1 étant équiprobables.
- Calculer l'énergie E_μ, de la configuration et la probabilité q_μ non normalisée correspondante.
- Tirer aléatoirement un réel x dans l'intervalle [0; 1] avec une densité uniforme.
- Si $x < q_\mu$, la configuration est retenue, sinon elle est rejetée.

L'évaluation de la moyenne se fait avec les configurations retenues. On évalue aussi la variance comme expliqué dans méthode de Monte-Carlo. En divisant cette variance par le nombre M de tirages, on obtient la variance de la moyenne.

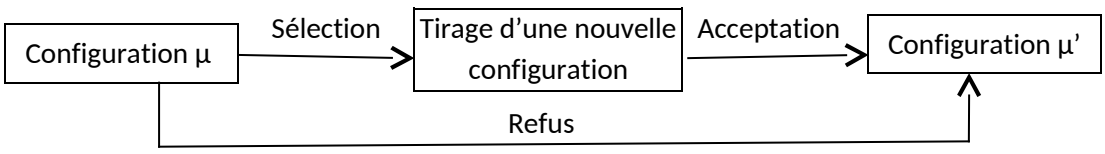
Dans le cas de la distribution exponentielle de Boltzmann, la méthode du rejet soufre d'un grave inconvénient : la très grande majorité des configurations a une probabilité non normalisée très faible, ce qui conduit à un taux de rejet très important, surtout lorsque la température est faible.

III.3. **Méthode de Metropolis**

La méthode de Metropolis a servi de point de départ pour le développement de méthodes d'échantillonnage par chaines de Markov, qui sont incomparablement plus efficaces que les méthodes d'échantillonnage direct. L'article original exposait son application au modèle des sphères dures, mais la méthode de Metropolis se généralise à tout problème de calcul statistique.

On se contente ici d'un exposé descriptif de la méthode appliquée au modèle d'Ising. Au lieu de tirer directement chaque configuration, la méthode de Metropolis tire une nouvelle configuration en modifiant légèrement la configuration précédente. Il s'agit de générer une suite de configurations avec des règles de transition, de telle sorte que les configurations obtenues obéissent à la loi de probabilité voulue, en l'occurrence la loi de Boltzmann.

L'obtention d'une nouvelle configuration à partir de la précédente se fait en deux étapes : la sélection et l'acceptation. En cas de refus de la nouvelle configuration, la configuration est conservée.



La sélection est une modification de la configuration simple à réaliser. Toute règle de sélection est possible, à condition qu'elle permette l'exploration de toutes les configurations. Dans le modèle d'Ising, on se contente de choisir un spin aléatoirement et de changer son signe. Notons μ la configuration et μ' la nouvelle configuration. Le rapport de leur probabilité est :

$$\frac{p_\mu}{p_{\mu'}} = e^{-\beta(E_{\mu'}-E_\mu)} = e^{-\beta\Delta E} \quad (14)$$

Lorsqu'on modifie un seul spin, il est très facile (et peu coûteux) de calculer la variation d'énergie :

$$\Delta E = E_{\mu'} - E_\mu \quad (15)$$

Lorsque cette différence est négative, la nouvelle configuration est plus probable que la précédente. Il est donc logique de l'accepter. Si la différence d'énergie est positive, il ne faut pas refuser systématiquement la nouvelle configuration, car cela pourrait empêcher d'explorer certaines configurations importantes. Il faut donc l'accepter ou la refuser en utilisant une méthode similaire à la méthode du rejet exposée plus haut. On tire un nombre réel aléatoire x avec une densité de probabilité uniforme sur l'intervalle [0; 1]. La nouvelle configuration est acceptée seulement si :

$$x < e^{-\beta\Delta E} \quad (16)$$

Dans le cas contraire, la nouvelle configuration est refusée et on garde la précédente.

Voici pour résumer les opérations à effectuer pour obtenir une nouvelle configuration, dans le cas du modèle d'Ising :

- Choix d'un spin aléatoirement.
- Calcul de la variation d'énergie ΔE résultant de l'inversion de ce spin.
- Si ΔE <= 0, le spin est inversé.
- Sinon, tirage d'un réel x aléatoirement avec une densité uniforme sur l'intervalle [0; 1]. Si $x < e^{-\beta\Delta E}$, le spin est inversé, sinon il est inchangé.

La suite de configurations ainsi générée converge vers la distribution de Boltzmann. La configuration initiale est quelconque ; on peut la choisir aléatoirement. Il faut calculer un certain nombre Q de configurations (qui peut être très grand) avant que la suite de configurations obéisse effectivement à la loi de Boltzmann. À partir de cet instant, le système est à l'équilibre et on peut commencer à calculer des grandeurs moyennes. Si M est le nombre total de configurations d'équilibre générées, l'évaluation de l'espérance de X est donnée par la somme suivante :

$$moy(X,M) = \frac{1}{M}\sum_{k=1}^{M} X_k \quad (17)$$

On calcule aussi la variance expérimentale.

Q est le nombre de configurations qu'il faut générer pour atteindre l'équilibre. Les calculs de moyenne ne peuvent commencer qu'à partir de ce rang. La principale difficulté de la méthode de Metropolis est l'évaluation de Q, qui n'est pas connu a priori. Comparée à la méthode d'échantillonnage directe (méthode du rejet), la méthode de Metropolis est bien meilleure pour deux raisons :

- Le calcul de ΔE est beaucoup moins coûteux que le calcul de l'énergie E d'une configuration.
- La proportion de configurations rejetées est beaucoup plus faible que dans la méthode du rejet.

Contrairement à la méthode directe, la méthode de Metropolis nécessite un certain nombre de tirages de mise à l'équilibre (Q), avant que l'on puisse commencer à faire les calculs de moyenne.

Pour améliorer une méthode de Metropolis, il faut trouver des règles de sélection qui donne des taux de rejet faibles dans l'étape d'acceptation. Une méthode de Metropolis idéale aurait un taux de rejet nul.

IV. Travaux pratiques : implémentation de la méthode de Metropolis

L'objectif est d'implémenter la méthode de Metropolis pour le modèle d'Ising à une dimension. Les deux variables aléatoires dont on cherche à calculer la moyenne sont le moment magnétique, c'est-à-dire la somme des spins :

$$m = \sum_{i=0}^{N-1} S_i \tag{18}$$

et l'énergie :

$$E = \sum_{i=0}^{N-1} BS_i + S_i S_{i+1} \tag{19}$$

Compte tenu du très grand nombre de configurations qu'il faudra générer, il faut prendre soin de minimiser les calculs. En particulier, le moment magnétique et l'énergie devront être stockés dans une variable, qui sera actualisée à chaque basculement de spin. Lorsque l'inversion d'un spin est testée, il faut calculer la variation d'énergie ΔE résultant de l'inversion de ce spin. Notons S l'état du spin avant l'inversion. Le terme d'interaction avec le champ extérieur est changé de $+2BS$. Pour le terme d'interaction entre spins voisins, il y a trois cas suivant l'état des spins avant le changement :

- Les deux spins voisins S_{i-1} et S_{i+1} sont de même signe que S_i. La variation d'énergie est +4.
- Les deux spins voisins sont de même signe mais de signe opposé à S_i. La variation d'énergie est -4.
- Les deux spins voisins sont de signes opposés. La variation d'énergie est nulle.

Finalement, il y a 6 variations d'énergie totale possibles :

$$2B + 4;\quad 2B - 4;\quad 2B;\quad -2B + 4;\quad -2B - 4;\quad -2B \tag{20}$$

Le facteur $e^{-\beta \Delta E}$ de ces 6 termes devra être calculé au préalable et stocké dans une liste.

Le programme sous Maple :

```
> restart:
> with(linalg):
Warning, new definition for norm
Warning, new definition for trace
> N, E, M, A, spins:
> E:=0:
> M:=0:
>
> ##Initialisation
> ##Constantes
> N:=100: n:=10:
> T:=1:
> B:=0.1:
> beta :=1.0/T:
> dE:=[2*B+4, 2*B-4, 2*B,-2*B+4, -2*B-4,-2*B]:
> A:=array(1..6):
> for i from 1 to 6 do
>    A[i]:=exp(-beta*dE[i]):
> od:
>
>
> ## création du tableau de spin initialisé tous à 1
> spins:=array(0..N-1,[1]):
>
> ##Initialisation des spins aléatoire
> for i from 0 to N-1 do
>     s:=rand(0..1):
>     spins[i]:=2*s()-1:
> od:
>
> ##le moment magnetique : la somme des spins
> M:=0:
> for i from 0 to N-1 do
>     M:=M+spins[i]:
> od:
> ##l'Energie  : la somme des energies des spins
> E:=0:
> for i from 0 to N-2 do
>     E:=E+B*spins[i]+spins[i]*spins[i+1]:
> od:
> E:=E+B*spins[N-1]+spins[N-1]*spins[0]:
>
> voisins:=proc(m)
> local s1, s3:
>    if m=0 then
>         s1:=spins[N-1]:
>    else
>         s1:=spins[m-1]:
>    fi:
>    if m=N-1 then
>         s3:=spins[0]:
>    else
>         s3:=spins[m+1]:
>    fi:
>    RETURN(s1, s3):
> end:
```

```
> deltaE:=proc(m)
> local s2, s1, s3, rs,p:
>      s2:=spins[m]:
>      rs:=voisins(m):
>      s1:=rs[1]: s3:=rs[2]:
>      if s2=1 then
>         p:=0:
>      else
>         p:=3:
>      fi:
>      if (s1,s2,s3)=(1,1,1) or (s1,s2,s3)=(-1,-1,-1) then
>             next:
>      elif  (s1,s2,s3)=(1,-1,1) or (s1,s2,s3)=(-1,1,-1) then
>            p:=p+1:
>      else
>            p:=p+2:
>      fi:
> ##     print(m,p,dE[p],A[p]):
>      RETURN(dE[p],A[p]):
> end:
```

```
> metropolis:=proc(m,x)
> #global spins, E, M:
> local xdE, xA, xR:
>      xR:= deltaE(m):
>      xdE:=xR[1]: xA:=xR[2]:
>
>      if xdE<=0 then
>         spins[m]:=-spins[m]:
>         E := E+xdE:
>         M := 2*spins[m]:
>      else
>             if is(x<xA) then
>                spins[m] := -spin[m]:
>                E := E+xdE:
>                M := 2*spins[m]:
>             fi:
>       fi:
> end:
```

```
> boucle :=proc(n)
> global e, m:
> local  tab_i, tab_x, i, j, k, rx :
>    m:=array(0..n-1):
>    e:=array(0..n-1):
>    for i from 0 to n-1 do
>        e[i]:=0: m[i]:=0;
>    od:
>    tab_i:=array(0..n*N-1):
>    tab_x:=array(0..n*N-1):
>    for i from 0 to n*N-1 do
>        rx:=rand(0..N-1):
>        tab_i[i]:=rx():
>        rx:=rand(0..1000000):
>        tab_x[i]:=evalf(rx()/1000000):
>    od:
>    j:=0:
>    for k from 0 to n-1 do
>         m[k]:=M:
>         e[k]:=E:
>         for i from 0 to N-1 do
>             metropolis(tab_i[j],tab_x[j]):
>             j:=j+1:
>         od:
>    od:
> end:
```

Chapitre X : Dynamique moléculaire

I. Introduction

La dynamique moléculaire est comme son nom l'indique une méthode permettant de simuler l'évolution temporelle d'un système moléculaire. Elle repose généralement sur l'utilisation de la relation fondamentale de la dynamique. La description par la mécanique classique à l'échelle atomique reste valable pour certaine valeurs de la longueur d'onde thermique de de Broglie pour le système considéré, définie par :

$$\lambda = \sqrt{\frac{h^2}{Mk_BT}}$$

où M est la masse atomique et T la température, k_B la constante de Boltzmann et h la constante de Planck.

L'approximation classique est faisable si cette longueur d'onde est plus petite que la distance qui sépare l'atome de son plus proche voisin. Pour un liquide dans des conditions thermodynamiques proches de son point triple (l'eau à 300 K par exemple) la longueur d'onde est plus de 10 fois inférieure à la distance interatomique. De manière assez générale pour une écrasante majorité des systèmes chimiques, cette approximation est parfaitement valide et l'approche quantique peut donc être négligée[iii].

II. Les équations du mouvement

Le mouvement atomique, entraînant le mouvement moléculaire est associée à la loi de Newton :

$$F_i = m_i a_i$$

où F_i représente la somme des forces exercées sur l'atome i de masse m_i et soumis à l'accélération a_i. Tenant en compte de ce concept, il est nécessaire de déterminer l'ensemble des positions et des vitesses de tous les atomes formant le système. Soit $\Gamma\{r_i v_i\}$ l'ensemble de toutes les positions r_i et des vitesses v_i pour $i = 1 \ldots N$.

La qualité d'une simulation de dynamique moléculaire dépend du nombre de positions et de vitesses qui devront être calculés.

Normalement, l'accélération est reliée à la vitesse, elle-même reliée à la position de la particule considérée.

La force sur un atome est obtenue à partir du calcul de l'énergie du système, généralement par une approche de type **« champ de force »**.

Connaissant donc l'énergie du système, appelée E, la force agissant sur un atome i est donnée par la relation suivante :

$$F_i = -\frac{dE}{dr_i}$$

Le problème consiste à résoudre l'équation du mouvement pour chaque atome, soit à résoudre le système de N équations suivant :

$$\vec{F_i} = m_i \vec{a_i} = m_i \frac{d^2 \vec{r_i}}{dt^2} = \frac{d\vec{p_i}}{dt}$$

Pou $i = 1 \ldots N$. Rappelons que pour les systèmes conservatifs, $\vec{F} = -\nabla V$. Si le potentiel d'interraction est connu, alors $\vec{F}$ est determinée explicitement.

III. Algorithmes de résolution des équations du mouvement

Le résolution du système est numérique. Dans ce sens, les équations sont descritisées en fonction de la variable temps pour utiliser des algorithmes spécifiques de résolution comme l'algorithme de Verlet publié en 1964. Ce dernier se présente en deux versions : Position Verlet et Vitesse Verlet.

II.1 Algorithme Position Verlet

Le développement de Taylor du vecteur position $\vec{r_i}$ de chaque atome i du système avec $+dt$ et $-dt$ est donné par :

$$\vec{r_i}(t + dt) = \vec{r_i}(t) + \vec{v_i}(t)dt + \frac{\vec{F_i}(t)}{2m_i} dt^2 + \frac{d^3\vec{r_i}}{dt^3} dt^3 + O(dt^4)$$

$$\vec{r_i}(t - dt) = \vec{r_i}(t) - \vec{v_i}(t)dt + \frac{\vec{F_i}(t)}{2m_i} dt^2 - \frac{d^3\vec{r_i}}{dt^3} dt^3 + O(dt^4)$$

La somme des deux expressions ci-dessus donne :

$$\vec{r_i}(t + dt) + \vec{r_i}(t - dt) = 2\vec{r_i}(t) + \frac{\vec{F_i}(t)}{2m_i} dt^2 + O(dt^4)$$

D'où :

$$\vec{r_i}(t + dt) = 2\vec{r_i}(t) + \vec{r_i}(t - dt) + \frac{\vec{F_i}(t)}{2m_i} dt^2 + O(dt^4)$$

La nouvelle position de l'atome *i* est déterminée à partir de sa position actuelle (*t*), celle d'avant (c-à-d à *t-dt*) et la force appliquée sur l'atome à l'instant (*t*). L'erreur comise sur la prédiction est de l'ordre de $O(dt^4)$. Pour des valeurs très petites de *dt*, l'erreur devient negligeable.

Pour le valeur de la vitesse de l'atome *i,* elle est calculée de la même façon. Le developement de Taylor de la vitesse s'écrit :

$$\vec{v}_i(t) = \frac{\vec{r}_i(t+dt) + \vec{r}_i(t-dt)}{2dt} + O(dt^2)$$

Les données des premiers ordres sont données par les conditions intiales. C'est que l'algorithme est initialisé par la donnée de la position $\vec{r}_i(t)$ pour $t = 0$ et $t = -dt$ de chaque atome *i* aux instants de la simulation. La force $\vec{F}_i(t)$ est calculée à l'instant $t = 0$. L'algorithme permet de calculer les valeurs des positions et des vitesses à un temps ultérieur.

II.2 Algorithme Vitesse Verlet

Cet algorithme est semblable au précédent et déduit de la même façon. Pour ce faire, reprenons le développement de Taylor du vecteur position en +*dt* à l'ordre 2. L'expression s'écrit sous la forme :

$$\vec{r}_i(t+dt) = \vec{r}_i(t) + \vec{v}_i(t)dt + \frac{\vec{F}_i(t)}{2m_i}dt^2 + O(dt^3)$$

Maintenant, si nous reprenons le développement suivant à l'étape *t*=*t*+*dt*,

$$\vec{r}_i(t-dt) = \vec{r}_i(t) - \vec{v}_i(t)dt + \frac{\vec{F}_i(t)}{2m_i}dt^2 + O(dt^3)$$

Nous obtenons :

$$\vec{r}_i(t) = \vec{r}_i(t+dt) - \vec{v}_i(t+dt)dt + \frac{\vec{F}_i(t+dt)}{2m_i}dt^2 + O(dt^3)$$

En additionnant la dernière et l'avant dernière équations, nous obtenons :

$$\vec{r}_i(t+dt) + \vec{r}_i(t) = \vec{r}_i(t) + \vec{r}_i(t+dt) + \vec{v}_i(t)dt - \vec{v}_i(t+dt)dt + \frac{\vec{F}_i(t)}{2m_i}dt^2 + \frac{\vec{F}_i(t+dt)}{2m_i}dt^2 + O(dt^3)$$

D'où

$$\vec{v}_i(t+dt) = \vec{v}_i(t) + \frac{\vec{F}_i(t) + \vec{F}_i(t+dt)}{2m_i}dt + O(dt^2)$$

Les équations donnant $\bar{r}_i(t+dt)$ et $\bar{v}_i(t+dt)$ forment l'algorithme Vitesse de Verlet dans lequel la position $\bar{r}_i$ et la vitesse $\bar{v}_i$ sont connus exactement en même instant de la dynamique $t+dt$. Les deux algorithmes de Verlet produisent les mêmes résultats. Seulement, dans l'algorithme Vitesse, il faut calculer $\bar{F}_i(t+dt)$, ce qui demande plus de calcul.

L'initialisation de l'algorithme à l'instant t=0 se fait en donnant des valeurs à la position $\bar{r}_i(0)$, à la vitesse $\bar{v}_i(0)$ et à la force $\bar{F}_i(0)$ pour chaque atime *i*.

II.3 Algorithme de Leap-Frog « saut de grenouille »

Dans cet algorithme, les vitesses et les positions des atomes sont calculées à des instants alternés. Nous avons :

$$\vec{v}_i\left(t+\frac{dt}{2}\right)=\frac{\bar{r}_i(t+dt)-\bar{r}_i(t)}{dt}+O(dt^2)$$

Ce qui donne :

$$\vec{r}_i(t+dt)=\vec{r}_i(t)+\vec{v}_i\left(t+\frac{dt}{2}\right)dt+O(dt^2)$$

De la même façon, il est facile à calculer :

$$\vec{r}_i(t-dt)=\vec{r}_i(t)-\vec{v}_i\left(t-\frac{dt}{2}\right)dt+O(dt^2)$$

Et

$$\vec{r}_i(t+dt)=2\vec{r}_i(t)-\vec{r}_i(t-dt)+\frac{\vec{F}_i(t)}{m_i}dt^2+O(dt^4)$$

Dans laquelle on remplace les expressions de , d'où :

$$\vec{v}_i\left(t+\frac{dt}{2}\right)=\vec{v}_i\left(t-\frac{dt}{2}\right)+\frac{\vec{F}_i(t)}{m_i}dt+O(dt^3)$$

La nouvelle position de l'atome *i* est obentenue par :

$$\vec{r}_i(t+dt)=\vec{r}_i(t)+\vec{v}_i\left(t+\frac{dt}{2}\right)dt$$

Comme tout algorithme, l'initialisation se fait en donant des valeurs à *t=0*, $\vec{v}_i\left(-\frac{dt}{2}\right)$, $\bar{r}_i(0)$ et $\bar{F}_i(0)$.

Il existe d'autres algorithmes plus précis mais il seront plus compliqués à implémenter et très couteux en temps de calcul.

III. Application en physique statistique

Il est évident que si les positions et les vitesses des particules sont connues alors toutes les grandeurs physiques du système seront obtenus comme valeurs moyennes des valeurs microscopiques. Par exemple :

La température instantanée du système est données par :

$$T(t) = \sum_{i=1}^{N} \frac{m_i \vec{v_i}^2(t)}{3Nk_B}$$

La température moyenne du système est comme valeur moyenne :

$$T = \sum_{i=1}^{n} \frac{T(t_i)}{n}$$

La valeur moyenne de l'énergie cinétique des particules est :

$$E = \sum_{i=1}^{N} \frac{1}{2} m_i \vec{v_i}^2(t) = \frac{3}{2} k_B T$$

N.B :

En physique statistique, toutes les grandeurs macroscopiques d'un système sont obtenu à partir de la description des positions et des vitesses de ses particules.

IV. Simulation des milieux continus

Les résultats prévus par les algorithmes présentés dans les sections précédentes sont plutôt applicables à des systèmes discrets constitués d'un nombre fini de particules. En réalité, ces systèmes ne reflètent pas la natures physiques des systèmes. Néanmoins, il existe plusieurs méthodes pour simplifier les systèmes réels en des systèmes traitables par les moyens de calcul.

Pour modéliser la matière condensée continue (liquides et solides) avec des tailles finies et raisonnables en temps de calcul et en occupation de mémoire, on introduit des conceptes pour limiter le système tels que les conditions aux bords périodiques.

Pour simplifier la procédure, considérons un système plat de N particule. La subdivision du système en des petites zones limités forme une périodicité. C'est que le contenu d'une zone sera le même que celui des zones voisines dans toutes les directions de l'espace.

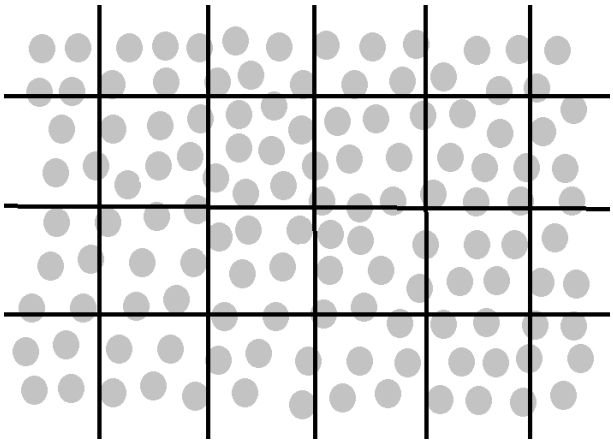
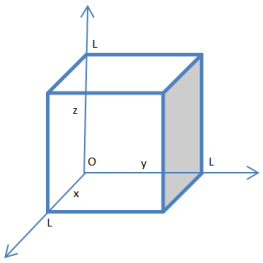


Schéma de répartition des particules d'un système continu en deux dimensions

La description du système, schématisé ci-dessus, semble plus facile en considérant une zone principale au centre alors que toutes les autres ne seront que sa réplication. Le mouvement des particules est considéré comme suit : Les particules sortant de la zone principale seront récupérées par celles qui y entrent. Ainsi, la zone étudiée gardera toujours le même nombre N de particules.

La zone centrale et ses voisines formeront 9 zones au total quand il s'agit d'un espace à 2 dimensions (2D) et 27 boites répliquées en 3D.



La simulation consiste à savoir à l'instant *t* le nombre de particules dans la boite principale. Pour simplifier la description, prenons le cube 3D d'arrêt L représentant de la zone principale de la simulation. On fixe l'origine à un coin du cube. Par conséquent, la position $\bar{r}_i(t)$ d'un atome *i* est repérée par les grandeurs : $x_i(t)$, $y_i(t)$ et $z_i(t)$. Pour que la particule soit comptée dans la boite, il faut que :

$$0 \leq x_i(t) \leq L \quad ; \quad 0 \leq y_i(t) \leq L \quad ; \quad 0 \leq z_i(t) \leq L$$

Si l'une de ces composantes ne vérifie par ces conditions, elle serat translaté par l'opération $x_i(t) \pm L$ selon les cas : $x_i(t) - L$ si $x_i(t) > L$ ou $x_i(t) + L$ si $x_i(t) < 0$.

[i] Institut de recherche sur les lois fondamentales de l'Univers
http://irfu.cea.fr/Projets/coast_documents/communication/Simulation.pdf
[ii] Extrait de cours de Marie-Pierre Gaigeot, Universitée d'Evry val d'Essonne, France.
[iii] Université en ligne, http://uel.unisciel.fr/chimie/modelisation/modelisation_ch05/co/1-3_le_controle_de_la_pression.html

Faculté des Sciences
Meknès

Département de Physique
Filière SMP : Semestre 6
Année Universitaire : 2020 / 2021

Série 1 : TD de Physique Numérique
Rappel sur l'algorithmique et la programmation

Exercice 1 : Rappel sur les notations fondamentales de l'algorithme

- Déclarer des variables simples de différentes natures,
- Déclarer des variables de type liste,
- Déclarer des variables de type matrice,
- Déclarer des variables de type Bloc de données.
- Rappeler les opérateurs
  - Arithmétiques
  - Logique
- Rappeler les structures logiques
- Rappeler les structures de contrôle :
  - Sélectives
  - Répétitives

Exercice 2 : traitement itératif

Elaborer l'algorithme pour implémenter les expressions suivantes:

- $n! = n \times (n-1) \times (n-2) \dots 2 \times 1$
- $n! = n \times (n-1)!$
- $e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots , \ -\infty < x < \infty$
- $(1+x)^n = 1 + \frac{nx}{1!} + \frac{n(n-1)x^2}{2!} + \cdots$
- $Un = \begin{cases} U_{n-1} + U_{n-3} & Si\ n\ est\ impair \\ \frac{U_{n-2}}{2} & si\ n\ est\ pair \end{cases}$

Exercice 3 : Applications

Elaborer l'algorithme nécessaire aux équations suivantes :

- $ax^2 + bx \quad + c = 0$
- $\boldsymbol{A} = \boldsymbol{B} \times \boldsymbol{C}$ où $A, B\ et\ C\ sont\ des\ matrices\ carrées\ n \times n$
- $\boldsymbol{M} = \boldsymbol{A} - \boldsymbol{\lambda I_n}$ où $\boldsymbol{I_n}\ est\ la\ matrice\ identité\ d'ordre\ \ n$

Exercice 4 : Introduction des algorithmes spécifiques

- $S_n = \int_0^{\pi} \cos(x)\, dx$
- $L_x = \int_1^x \frac{dt}{t}$
- $\frac{dy(x)}{dx} - \propto y(x) = 0$

Exercice 5 : Implémentation des algorithmes

Ecrire les programmes d'implémentation des algorithmes des exercices 2, 3 et 4 en utilisant le langage formel de Maple.

Université Moulay Ismail
Faculté des Sciences
Département de Physique
Meknès

Filière SMP
Semestre VI
Année : 2020-2021

TD N°2 de Physique Numérique

N.B : les réponses demandées sont des instructions selon la syntaxe Maple.

I. Etudes des fonctions à variables réelles

1. Définir une fonction quelconque $f(x) : \mathbb{R} \to \mathbb{R} \; / \; x \to f(x)$,
2. Tester la continuité de $f(x)$ sur l'ensemble $\mathbb{R}$,
3. Trouver les points de discontinuités de $f(x)$,
4. Chercher les limites de $f(x)$ à gauche et à droite des points de discontinuités puis vers $\pm\infty$,
5. Calculer les dérivées 1ères, 2ème et plus de $f(x)$ en utilisant :
   a. L'operateur D
   b. La fonction diff().

II. Intégration des fonctions réelles

1. Définir une fonction quelconque $f(x) : \mathbb{R} \to \mathbb{R} \; / \; x \to f(x)$,
2. Intégrer $f(x)$ sans spécifier le domaine, interpréter les résultats,
3. Intégrer $f(x)$ sur un intervalle [a, b],

III. Développement limité

1. Définir une fonction quelconque $f(x) : \mathbb{R} \to \mathbb{R} \; / \; x \to f(x)$,
2. Donner l'instruction pour effectuer le développement de Taylor de $f(x)$ à ordre n,
3. Convertir le résultat en un polynôme P,
4. Extraire le coefficient d'un exposant s de P.
5. Refaire le développement en appliquant la fonction **series().,**
6. Convertir le résultat en Polynôme

IV. Equations différentielles

1. Définir une équation différentielle quelconque selon la syntaxe Maple,
2. Donner l'instruction générale de résolution de l'équation définie,
3. Ajouter les conditions initiales à l'instruction 2,
4. Définir un système d'équations différentielles selon la syntaxe Maple,
5. Donner l'instruction de résolution du système 4,
6. Ajouter les conditions initiales à l'instruction 5,
7. Donner les instructions d'extraction des résultats selon :
   a. **nops / op**
   b. **subs()**
8. Refaire 2 et 3 pour une résolution numérique de l'équation définie en 1,
9. Evaluer la solution pour des valeurs réelle de x.