Maple : résumé de cours et exercices en sciences physiques

François Bertault

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

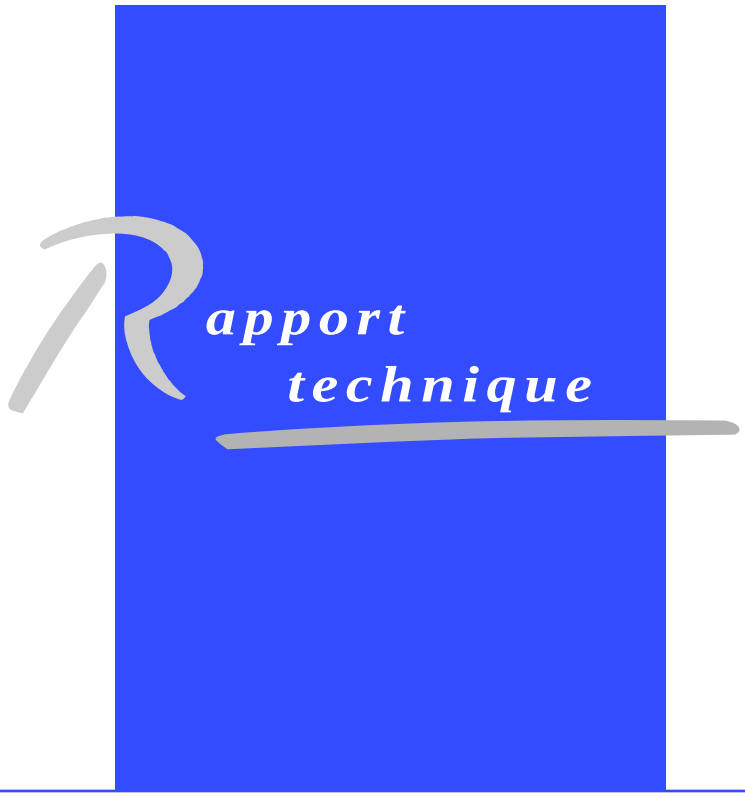
Maple : *résumé de cours et exercices en sciences physiques*

François Bertault

N■ 2936

Juillet 1996

THÈME 2



ISSN 0249-0803

INRIA
LORRAINE

Maple : résumé de cours et exercices en sciences physiques

François Bertault

Thème 2 — Génie logiciel
et calcul symbolique
Projet Euréca

Rapport technique n° 2936 — Juillet 1996 — 45 pages

Résumé : Ce rapport reprend un cours de Maple destiné à des élèves de première année de classe préparatoire scientifique. Les principales fonctions Maple ainsi que les concepts fondamentaux des systèmes de calcul formel sont présentés. Des exercices inspirés de problèmes en sciences physiques sont proposés. Une correction complète est donnée pour chaque exercice.

Mots-clé : Maple, calcul formel, cours, exercices, physique

(Abstract: pto)

Unité de recherche INRIA Lorraine
Technopôle de Nancy-Brabois, Campus scientifique,
615 rue de Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY (France)
Téléphone : (33) 83 59 30 30 – Télécopie : (33) 83 27 83 19
Antenne de Metz, technopôle de Metz 2000, 4 rue Marconi, 55070 METZ
Téléphone : (33) 87 20 35 00 – Télécopie : (33) 87 76 39 77

Maple : Course and Exercices in Physics

Abstract: This paper is an introductory short course, for undergraduate students, about the Maple computer algebra system. It contains a short description of the most useful fonctions, and exercises in physics with their solutions.

Key-words: Maple, computer algebra, course, exercises, physics

Table des matières

Introduction

Ce fascicule reprend un cours de **Maple** destiné à des élèves de première année de classe préparatoire scientifique. Il est structuré en trois parties. Dans la section 1, nous présentons brièvement les principales fonctions utilisées en **Maple**([1],[2]), ainsi que les concepts fondamentaux des systèmes de calcul formel ([3],[4]). La section 2 est un recueil d'exercices pratiques, appliqués à la Physique. Une correction complète est donnée pour chaque exercice. Enfin, la section 3 tente de répondre aux problèmes que se posent souvent les utilisateurs débutants.

1 Présentation de Maple

1.1 Maple: un système de calcul formel

Un système de calcul formel est un logiciel qui permet d'effectuer des calculs mathématiques exacts (par opposition à calcul numérique approché). **Maple** permet également d'effectuer des calculs numériques en précision arbitraire (du calcul approché donc) ainsi que des tracés graphiques. Enfin **Maple** possède un langage de programmation évolué.

1.2 Fonctionnalités principales

Décrire avec précision ne serait-ce que les principales fonctions **Maple** serait extrêmement long. Nous faisons donc ici un tour d'horizon, sur des exemples, des principales fonctions de **Maple**. Pour une description exhaustive des fonctions, se référer à l'aide en ligne.

1.2.1 Aide en ligne

Pour obtenir de l'aide sur un sujet ou une fonction, il faut faire précéder l'intitulé du sujet ou le nom de la fonction par `?`. Pour chaque fonction, l'aide présente la syntaxe à utiliser, décrit ce que fait la fonction, présente des exemples, puis donne une liste de fonctions pouvant être utiles à la place ou en complément de la fonction. Par exemple, pour avoir de l'aide sur la résolution d'équations :

```
> ?solve

FUNCTION: solve - Solve Equations

CALLING SEQUENCE:
   solve(eqn, var)
   solve(eqns, vars)

PARAMETERS:
   eqn  - an equation or inequality
   eqns - a set of equations or inequalities
   var  - (optional) a name (unknown to solve for)
   vars - (optional) a set of names (unknowns to solve for)
```

```
SYNOPSIS:
- The solution to a single equation eqn solved for a single unknown var is
  returned as an expression.

[...]

EXAMPLES:
> solve( f=m*a, a );
                                      f/m

> solve( {f=m*a}, {a} );
                                   {a = f/m}
[...]
SEE ALSO: dsolve, fsolve, isolve, msolve, rsolve, assign, invfunc, isolate,
          match, linalg[linsolve], simplex, grobner, solve[<subtopic>] where
          <subtopic> is one of: floats, functions, identity, ineqs, linear,
          radical, scalar, series, system
```

1.2.2 Affectation

Il est possible d'affecter un nom à une expression a l'aide de `:=`.

```
> a:=sin(5)+3;
                                a := sin(5) + 3
> a;
                                   sin(5) + 3
```

1.2.3 Instructions

Une instruction est une expression `Maple` se terminant par `;` ou `:`. Dans le cas où l'instruction se termine par `:`, le résultat n'est pas affiché. Dans une expression, `"` représente le dernier résultat obtenu.

1.2.4 Résolution d'équations

La fonction `solve` permet de résoudre des équations ou des systèmes d'équations d'une ou plusieurs variables :

```
> solve({cos(a)+sin(b)=5,b=3},{a,b});
                     {a = arccos(- sin(3) + 5), b = 3}
```

1.2.5 Différentiation

La fonction `diff` permet de calculer la dérivée d'une expression en une ou plusieurs variables :

```
> diff(sin(exp(x+y))*x^2,x);
                                                  2
             cos(exp(x + y)) exp(x + y) x  + 2 sin(exp(x + y)) x
```

1.2.6 Intégration

La fonction **int** permet de calculer une primitive ou une intégrale définie :

```
> int(sin(x+y)^5,x);
                  4                                2
  - 1/5 sin(x + y)  cos(x + y) - 4/15 sin(x + y)  cos(x + y) - 8/15 cos(x + y)

> int(sin(x)^2,x=0..1);
                           - 1/2 cos(1) sin(1) + 1/2
```

1.2.7 Sommes symboliques

La fonction **sum** permet de calculer une somme définie ou indéfinie :

```
> sum(k^3,k);
                                    4        3        2
                                 1/4 k  - 1/2 k  + 1/4 k

> sum(1/k^2,k=1..5);
                                        5269
                                        ----
                                        3600
```

1.2.8 Manipulation de polynômes

Création. Il est possible de définir des polynômes en une ou plusieurs variables.

```
> p:=x^3+5*x^2+8*x+4;
                                      3      2
                                p := x  + 5 x  + 8 x + 4
> p:=x^3*y+5*x^2+8*x+4+y^3;
                                  3        2              3
                           p := x  y + 5 x  + 8 x + 4 + y
```

Manipulation. Les fonctions qui manipulent les polynômes sont nombreuses. Les principales sont résumées dans le tableau suivant :

Opérations syntaxiques	**coeff**	Sélection d'un coefficient
	coeffs	Liste des coefficients
	degree	Degré du polynôme
Réécriture	**expand**	Développement
	collect	Développement par rapport à une variable
	sort	Tri des termes du polynôme
Calcul	**quo**	Quotient de la division euclidienne
	rem	Reste de la division euclidienne
	gcd	Pgcd
	factor	Factorisation

1.2.9 Développements limités

Les développements en séries se font à l'aide de la fonction **series**. Le développement à l'ordre 3 de $x + \frac{1}{x}$ en $x = 1$ s'écrit :

```
> series(x+1/x, x=1, 3 );
                                      2            3
                          2 + (x - 1)  + O((x - 1) )
```

1.2.10 Calcul de limites

Les limites se calculent à l'aide de la fonction **limit**. Si **Maple** ne sait pas calculer une limite, il retourne la forme de l'appel.

```
> limit(sin(x)/x, x=0);
                                        1

> limit(x^n, x=infinity);
                               limit(x^n, x=infinity);

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

1.2.11 Fonctions

Les fonctions simples peuvent être définies à l'aide de l'opérateur fonctionnel **->** ou de la notation crochet **<resultat | variable >**

```
> f:=x -> x^2;
                                          2
                                  f := x -> x
> f(2);
                                       4

> g:=< (x*y,2*x) | x,y>;
                              g := <x y, 2 x|x, y>
> g(2,1);
                                       2, 4
```

1.2.12 Ensembles

Maple permet de manipuler des ensembles d'objets non ordonnés et sans répétitions. Les ensembles peuvent contenir d'autres ensembles. L'ensemble vide est noté **{}**. Les opérations possibles sur les ensembles sont l'union (**union**), l'intersection (**intersect**) et la différence (**minus**) de deux ensembles. Le test d'appartenance d'un élément à un ensemble peut se faire à l'aide de la fonction **member**.

```
> {a,b,c,a};
                                  {a, b, c}
```

```
> {a,{a,b},c};
                                {a, c, {a, b}}
> member(x,{a,b} union {{x},z});
                                    false
```

1.2.13 Listes

Les listes permettent de définir des séquences ordonnées d'éléments, avec répétitions possibles. La liste vide est notée `[]`. Le `i`ème élément d'une liste `L` est donné par l'opérateur de sélection : `L[i]`

```
> [a,b,c,a];
                                 [a, b, c, a]
> [a,[a,b],c];
                                [a, [a, b], c]
> "[2];
                                    [a, b]
```

1.2.14 Suites d'expressions

Les suites d'expressions sont définies à l'aide de l'opérateur `,`. Elles apparaissent lors des appels de fonctions, de la création de listes ou d'ensembles par exemple. La suite d'expressions vide est notée `NULL`.

```
> s:=a,b,c;
                                 s := a, b, c
> [s];
                                  [a, b, c]
> a,s,d;
                                a, a, b, c, d
```

1.2.15 Tables

Les tables permettent de définir des variables indexées par des expressions. L'exemple suivant montre les différentes propriétés des tables.

```
> T[sin(x)]:=5;
                                T[sin(x)] := 5
> T[1/2]:=7;
                                 T[1/2] := 7
> T;
                                       T
> print(T);
                                table([
                                    sin(x) = 5
                                    1/2 = 7
                                ])
> T[1];
                                     T[1]
> T[sin(x)];
                                      5
```

1.2.16 Tableaux

Dans le cas où l'on manipule des variables indexées par des entiers, et que le nombre d'index est fixe, il convient d'utiliser des tableaux. Contrairement aux tables, les tableaux doivent être déclarés à l'aide de `array` avant d'être utilisés.

```
> T:=array(1..2,-2..-1):
> T[1,-1]:=1:
> T[1,-2]:=x^2:
> T[1,-1];
                                  T[1, -1] := 1
> T[2,-2]:=2;
                                  T[2, -2] := 2
> T;
                                       T
> print(T);
                            array(1 .. 2,-2 .. -1,, [
                                             2
                                (1, -2) = x
                                (1, -1) = 1
                                (2, -2) = 2
                                (2, -1) = T[2, -1]
                            ])
```

1.2.17 Calcul numérique en précision arbitraire

La fonction `evalf` permet de calculer numériquement une expression. Le nombre de décimales peut être spécifié soit comme deuxième paramètre de la fonction `evalf`, soit en modifiant la valeur de `Digits`.

```
> evalf(Pi);
                                   3.141592654
> evalf(Pi,50);

            3.1415926535897932384626433832795028841971693993751
> Digits:=20;
                                   Digits := 20
> evalf(Pi);
                              3.1415926535897932385
```

1.2.18 Types

Il est possibles de vérifier si une expression est d'un certain type, à l'aide de la fonction `type`. Les principaux types en `Maple` sont :

Nom de type	Signification
`numeric`	valeur numérique
`integer`	nombre entier
`complex`	nombre complexe
`float`	nombre décimal
`polynom`	polynôme
`table`	table
`array`	tableau

1.2.19 Courbes et surfaces

`Maple` permet de tracer des courbes en dimension 2 et 3 ainsi que des surfaces, à l'aide des fonctions `plot` et `plot3d`. Il est possible de tracer des courbes en coordonnées cartésiennes, polaires ou cylindriques. On peut également faire figurer les lignes d'isovaleur d'une surface, ou encore représenter des courbes implicites. Plusieurs tracés peuvent être représentés sur le même graphique, à l'aide de la fonction `plots[display]`. Plusieurs exemples de telles courbes sont donnés dans les exercices de la section 2.

1.3 Expressions

Les expressions sont les objets principaux des systèmes de calcul formel. Lorsque l'utilisateur décrit l'expression mathématique, par exemple $\sin(1+x)$, il connaît un certain nombre de propriétés mathématiques sur cette fonction. En revanche `Maple` ne considère cette expression que d'un point de vue syntaxique : la fonction `sin` est appliquée à la somme de l'entier `1` et de la variable `x`. Lorsque l'on souhaite appliquer des opérations mathématiques à cette expression, comme par exemple la dériver ou l'intégrer, `Maple` ne dispose que d'une information syntaxique sur l'expression.

1.3.1 Arbre syntaxique d'une expression

Les expressions `Maple` peuvent être représentées par un arbre. Par exemple, l'expression $1/3+\sin(1+x^2)+\pi$ sera représentée par l'arbre de la figure 1. Les opérations `Maple` consistent à analyser un tel arbre pour en construire un nouveau.

1.3.2 Manipulation syntaxique d'une expression

Les opérations syntaxiques suivantes sont possibles sur une expression :

- Obtenir tous les opérandes d'une expression (les sous-arbres de la racine) sous forme d'une suite d'expressions : `op`
- Obtenir le ième opérande d'une expression : `op(i,..)`
- Obtenir le nombre d'opérandes d'une expression : `nops`
- Substituer le ième opérande d'une expression par une autre expression : `subsop(i=..,..)`
- Supprimer le ième opérande d'une expression : `subsop(i=NULL,..)`
- Substituer tous les sous-arbres d'une expression égaux à celui correspondant à une certaine expression, par une autre expression : `subs`
- Supprimer tous les sous-arbres de l'expression égaux à celui correspondant à une certaine expression : `subs(...=NULL,..)`
- Substituer la racine de l'arbre de l'expression : `convert`

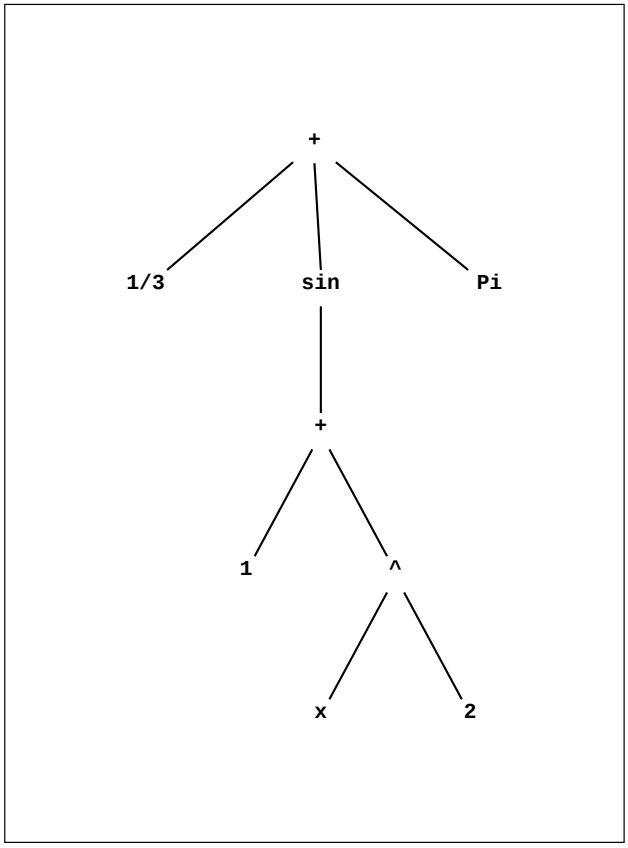


FIG. 1 – *Arbre syntaxique de l'expression* $1/3 + \sin(1 + x^2) + \pi$

1.4 Le problème de la simplification

1.4.1 Forme normale, classes d'expressions

La simplification d'une expression est un problème difficile. Un polynôme est-il plus simple sous forme développée ou sous forme factorisée ? Cette question dépend en fait du polynôme. $(x+1)^{100}$ est certainement plus simple sous forme factorisée, tandis que $x^{100}+1$ est certainement plus simple sous forme développée. De plus certains calculs sont plus rapides quand les expressions sont sous une certaine forme (par exemple sous forme factorisée dans le cas du calcul du pgcd de deux polynômes). Pour ces raisons, la réécriture d'une expression (la mettre sous forme factorisée ou développée par exemple) est laissée dans la plupart des cas à la charge de l'utilisateur.

Un autre problème directement lié à la simplification consiste à savoir si deux expressions sont égales ou sont différentes d'un point de vue mathématique. Pour que **Maple** puisse décider si une expression est égale à une autre, il faut que ces deux expressions aient la même écriture syntaxique. Mettre une expression sous forme normale consiste à lui donner une nouvelle forme syntaxique sans en changer le sens mathématique, telle que mise sous cette forme, elle est égale mathématiquement à une autre expression, elle même sous forme normale, si et seulement si elles sont égales syntaxiquement. Il existe des classes d'expressions pour lesquelles il n'y a pas de forme normale. Il existe des classes d'expressions pour lesquelles on ne peut pas tester en temps fini si elles sont nulles ou pas. Ceci peut donc conduire à des erreurs lors de simplifications. En pratique, on vérifiera les résultats obtenus chaque fois que l'on manipulera des expressions qui ne sont pas des fractions rationnelles.

1.4.2 Transformation d'expressions

Nous donnons ici les principales fonctions de réécriture d'une expression :

- **normal** : met une expression sous sa forme normale (si elle existe).
- **combine** : regroupe les termes d'une expression.
- **expand** : développe une expression.
- **simplify** : essaye de simplifier une expression. Est surtout utile avec un second paramètre qui permet de spécifier le type de simplification que l'on souhaite (*cf.* **?simplify**).

1.5 Évaluation d'une expression

Lorsque l'on affecte une expression à une variable, puis que l'on demande ce que vaut la variable, **Maple** retourne l'expression. Si aucune expression n'a été affectée à une variable, la valeur retournée est le nom de la variable.

```
> a:=3+b;
                                  a := 3 + b
> a;
                                    3 + b
```

Si l'expression contient des variables, elles sont également remplacées par l'expression qui leur est associée.

```
> b:=2*c;
                                    b := 2 c
> a;
                                    3 + 2 c
```

1.5.1 Niveaux d'évaluation

L'affectation d'une expression à une variable peut être visualisée par une flèche. Par exemple `a:=b` correspond à `a` $\rightarrow$ `b`. L'évaluation d'une variable consiste alors à suivre toutes les flèches. `a:=b: b:=c: c:=d:` correspond à `a` $\rightarrow$ `b` $\rightarrow$ `c` $\rightarrow$ `d`. En tapant `a;` on obtient donc `d`, en suivant toutes les flèches. Il est possible de limiter le nombre de flèches que l'on souhaite suivre : `eval(a,2)` produit `c`, ce qui correspond à suivre deux flèches.

1.5.2 Protection de l'évaluation

Il est possible d'empêcher l'évaluation d'une variable ou une expression,à l'aide de `'`. On dira que la variable ou l'expression est protégée. Chaque fois que l'on évalue une variable ou une expression protégées, on perd un niveau de protection.

```
> b:=3: a:='b': d:=b: b:=5:
> a,d;
                                    5, 3
> ''a'';
                                    'a'
```

Pour désaffecter une variable, on lui associe son nom : `a:='a';`

1.6 Les éléments du langage de programmation

1.6.1 Structures conditionnelles

Expressions booléennes. Une variable booléenne est une variable qui peut prendre deux valeurs : **`false`** ou **`true`**. Une expression booléenne est une expression qui manipule des booléens et les trois opérateurs suivants :

- **`not a`** : retourne **`false`** si `a` vaut **`true`**, et retourne **`true`** si `a` vaut **`false`**,
- **`a and b`** : retourne **`true`** si `a` et `b` valent **`true`**, et retourne **`false`** sinon,
- **`a or b`** : retourne **`false`** si `a` et `b` valent **`false`**, et retourne **`true`** sinon.

Par ailleurs, la fonction **`evalb`** retourne **`true`** si une expression mathématique est vraie et **`false`** dans le cas contraire. Dans certains cas, **`Maple`** ne peut pas décider : l'expression est alors retournée.

Tests. Les tests peuvent être effectués à l'aide de l'instruction `if`, dont la syntaxe est :

```
if condition1 then
  calcul1
elif condition2 then
  calcul2
else
  calculk
fi
```

En d'autres termes, si `condition1` vaut `true`, alors `calcul1` est exécuté, sinon si `condition2` vaut `true`, alors `calcul2` est exécuté, sinon `calculk` est exécuté. Les champs `elif` et `else` sont facultatifs. Par exemple pour calculer la valeur absolue d'un nombre `a`, on peut écrire :

```
if a<0 then
  -a;
else
  a;
fi;
```

1.6.2 Structures itératives

L'itération est une opération qui permet d'effectuer plusieurs fois une même suite d'opérations, en faisant varier un ou plusieurs index. Le nombre de fois où l'instruction est exécutée peut dépendre également d'une condition.

Boucle `do`. La forme la plus générale de l'itération en `Maple` est la suivante :

```
for indice from depart to fin by pas while condition do calcul od
```

En d'autres termes, `indice` vaut `depart` à la première itération. `pas` est ajouté à `indice` à chaque itération, c'est-à-dire après chaque exécution de `calcul`. L'itération s'arrete dès que `indice` dépasse `fin` ou que `condition` vaut `false`. Seuls `do` et `od` sont obligatoires. Par défaut, `depart` et `pas` valent `1`. On dira des instructions comprises entre `do` et `od` qu'elles font partie d'un même bloc d'instructions. Par exemple pour calculer $\sum_{i=1}^{4} \frac{1}{i!}$, on peut écrire :

```
> s:=0: for i to 4 do s:=s+1/i!: od: s;
                                      41
                                     ----
                                     24
```

Pour calculer le plus petit k tel que $\prod_{i=1}^{k} \frac{1}{i} \leq \frac{1}{5}$:

```
> p:=1: for k while p>1/5 do p:=p*1/k: od: k-1;
                                    3
```

Une variante de la forme précédente existe. L'indice de l'itération prend les opérandes d'une expression comme valeurs successives.

```
for indice in expression while condition do calcul od
```

```
> s:=0: for i in [1,4,3] do s:=s+i^2: od: s;
                                        26
```

`seq` et `map`. La fonction `seq` permet de construire des séquences d'expressions :

```
> seq(f(i), i = 1..5);
                        f(1), f(2), f(3), f(4), f(5)
```

La fonction `map` permet d'appliquer une fonction à chacun des opérandes d'une expression :

```
> map(f,a+b+c);
                              f(a) + f(b) + f(c)
```

`$`. `x$i` permet de produire une suite d'expressions composée de `i` fois `x`.

```
> a$3;
                                    a, a, a
```

1.6.3 Procédures

Les procédures permettent d'appeler plusieurs fois de suite une même suite d'instructions.

Définition. La syntaxe des procédures est la suivante :

```
nom:=proc(parametres)
  local varLocales;
  global varGlobales;
  instructions
end;
```

`nom` est le nom de la procédure. `parametres`, `varLocales` et `varGlobales` sont des suites d'expressions contenant des noms de variables. `parametres` peut éventuellement être nulle. Les champs `local` et `global` sont facultatifs. `instructions` est le corps de la procédure, c'est-à-dire une suite d'instructions séparées par des `;` ou des `:`. Dans la suite `parametres`, on peut demander à ce que certaines variables soient d'un certain type : `var1:type1,..,vark`. Lors de l'appel, si le paramètre `var1` n'est pas du type `type1`, un message d'erreur apparaît.

Appel. La syntaxe d'appel des procédures est la suivante :

```
nom(arguments);
```

`nom` est le nom de la procédure à exécuter. `arguments` est une suite d'expressions ayant au moins autant d'opérandes que la suite `parametres` dans la définition de la procédure `nom`.

Résultat d'une procédure. La fonction `RETURN` permet de retourner un résultat et de quitter une procédure. Si aucune instruction `RETURN` n'est rencontrée lors de l'exécution de la procédure, la valeur retournée est celle de la dernière instruction exécutée.

Passage par valeur des paramètres. Le passage des paramètres se fait par valeur, ce qui veut dire que l'on envoie une copie des expressions et non pas les expressions elles-mêmes. On ne peut pas affecter une expression à un paramètre à l'intérieur de la procédure (sauf si l'on passe le nom de la variable).

Variables locales et globales. Les variables présentes dans le champ `local` ont la propriété suivante : leur valeur une fois la procédure exécutée est celle qu'elles avaient avant d'appeler la procédure (ceci reste vrai lors des appels récursifs). Plus précisément cela revient à renommer les variables par des noms de variables qui ne seraient utilisés que lors de cet appel de la procédure. Par contre les variables placées dans le champ `global` sont effectivement modifiées.

Remarques.

- L'évaluation dans le corps d'une procédure est partielle. Elle correspond à `eval(..,1)`.
- Les variables qui ne sont déclarées ni dans le champ `local` ni dans `global` sont considérées comme locales si rien ne leur a été affecté avant la définition de la procédure, et comme globales sinon.

1.7 Conseils

- Décomposer les problèmes en sous problèmes. Il est en effet plus facile d'écrire et de tester plusieurs petites procédures qu'une seule grosse.
- Tester les procédures que vous écrivez. Pour cela, essayer la procédure en connaissant a priori le résultat qui doit être retourné. Essayer de faire en sorte que chaque instruction des procédures soit appelée lors des tests.
- Indenter less procédures. Les instructions d'un même bloc doivent être au même niveau, celles d'un sous bloc doivent être décalées vers la droite.

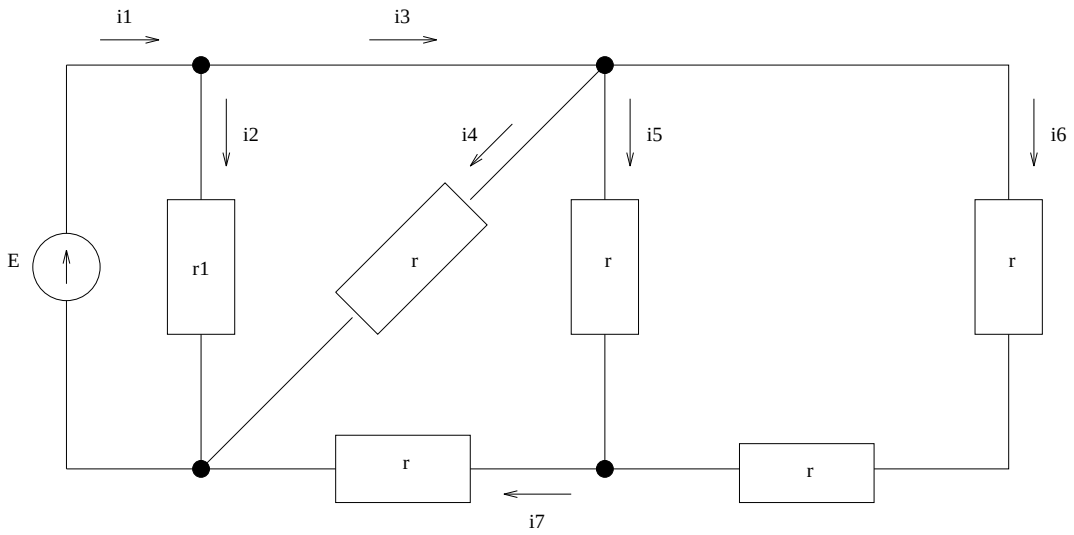


FIG. 2 –

- Commenter chaque procédure. Expliquer ce que fait la procédure, le rôle joué par chacun des paramètres. Spécifier les conditions que doivent vérifier les paramètres pour que la procédure s'exécute correctement.
- Il est également conseillé d'employer des noms de variables ou de procédures significatifs, ce qui évite ensuite l'ajout de commentaires. Appelez `distance` une procédure qui calcule la distance euclidienne entre deux points plutôt que `x5`. À l'intérieur d'une procédure, il peut être intéressant d'expliquer à quoi correspondent certaines variables. Évitez les commentaires inutiles comme "on incrémente i" ou "on affiche i".

2 Exercices pratiques

2.1 Énoncés

2.1.1 Équilibre d'une réaction chimique

Déterminez a, b, c, d, e, f afin d'équilibrer la réaction suivante :

$$aFe^{2+} + bMnO_4^- + cH^+ \rightleftharpoons dFe^{3+} + eMn^{2+} + fH_2O \tag{1}$$

2.1.2 Loi de Kirchov

On considère le circuit de la figure 2.

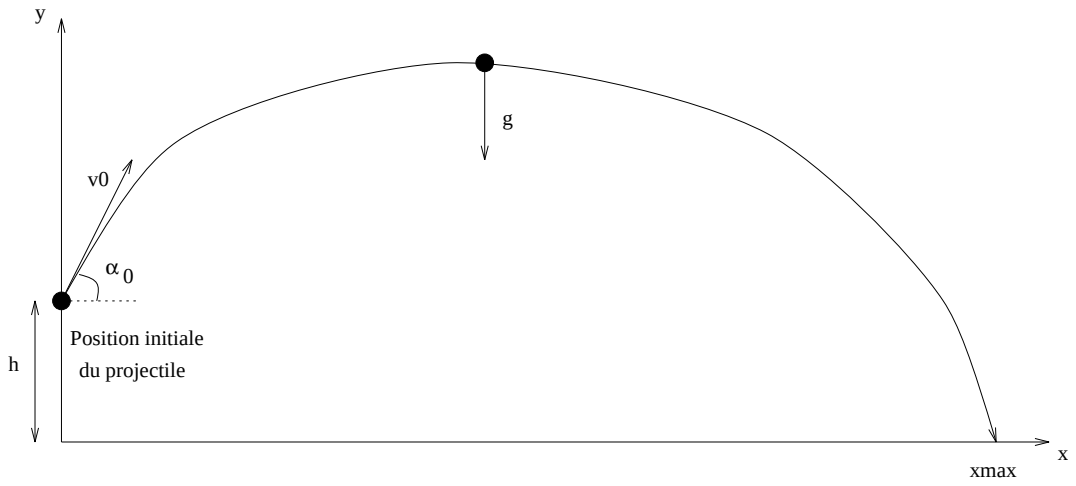


FIG. 3 –

Question 1. Déterminer les intensités $i1, \ldots, i7$ en fonction de r, $r1$ et E.

Question 2. Que valent $i1, \ldots, i7$ lorsque $r = r1$?

Question 3. Quelle valeur choisir pour $r1$ si l'on veut que $i1 = 10 \times i4$?

2.1.3 Projectile

On se propose dans ce problème d'étudier la trajectoire, dans un référentiel galiléen, d'un projectile lancé avec une vitesse initiale. La trajectoire de ce projectile est caractérisée par les équations différentielles suivantes (voir Fig. 3) :

$$\begin{cases} \ddot{x} &= 0 \\ \ddot{y} &= -g \\ \dot{x}(0) &= v_0 \cos(\alpha_0) \\ \dot{y}(0) &= v_0 \sin(\alpha_0) \\ x(0) &= 0 \\ y(0) &= h \end{cases}$$

On suppose que $\alpha_0 \in [0, \frac{\pi}{2}]$, $g > 0$ et $h \geq 0$.

Question 1. Résoudre le système différentiel afin d'obtenir la forme explicite de la trajectoire (`dsolve`). Appeler `pos_x` et `pos_y` les expressions des trajectoires en x et y respectivement.

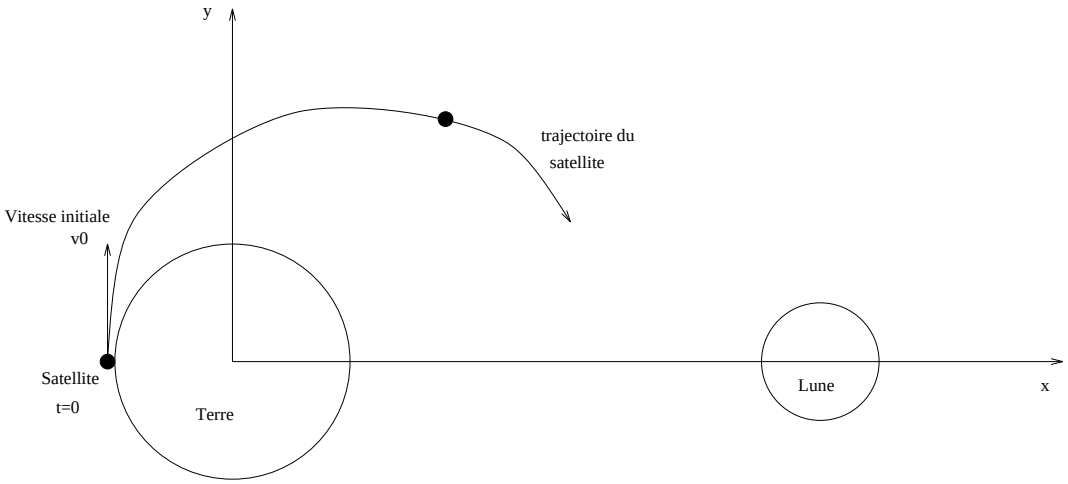


FIG. 4 –

Question 2. Déterminer à quel moment t_{max} le projectile touche le sol, en fonction de α_0, v_0, h et g. Quelle est alors la position du projectile?

Question 3. Quelle est la hauteur maximale atteinte par le projectile?

Question 4. Quelle est la hauteur moyenne du projectile lors du vol?

Question 5. On a les valeurs numériques suivantes, dans les unités appropriées:

- $g = 10$
- $v_0 = 10$
- $\alpha_0 = \frac{\pi}{4}$
- $h = 2$

Tracer la courbe de la trajectoire.

2.1.4 Apollo 8

On se propose d'étudier la trajectoire (dans un référentiel galiléen) d'un satellite en tenant compte des interactions gravitationnelles entre le satellite, la terre et la lune. Pour cela on se place dans le repère indiqué sur la figure 4. On suppose que le satellite a une

masse très inférieure à celles de la terre et de la lune. On peut donc supposer que la terre et la lune sont immobiles dans ce repère, et que seul le satellite subit les forces d'attractions de la terre et de la lune. On rappelle qu'une particule de masse m placée en $\vec{r}$ subit la force $\vec{F}$, du fait de l'interaction avec N particules de position et de masses respectives $\vec{r_i}$ et m_i, de la forme :

$$\vec{F}(\vec{r}) = Km_0 \sum_{i=1}^{N} m_i \frac{\vec{r_i} - \vec{r}}{|\vec{r_i} - \vec{r}|^3} \tag{2}$$

On supposera dans la suite du problème que les unités sont choisies de sorte que $K = 1$

Question 1. Écrire à l'aide de l'opérateur `->` une fonction qui calcule la distance euclidienne entre deux points $(x1, y1)$ et $(x2, y2)$ du plan.

Question 2. On suppose que le satellite est placé au temps $t = 0$ comme indiqué sur la figure 4, et que sa vitesse initiale est v_0. En utilisant la relation (ou postulat) fondamentale de la dynamique newtonienne, décrire (en **Maple**) les équations différentielles qui régissent le mouvement du satellite.

Question 3. On suppose que l'on a les données numériques suivantes, dans les unités adéquates :

- masse de la terre : 4
- rayon de la terre : 1
- position de la terre dans le repère choisi : $(0, 0)$
- masse de la lune : 1
- position de la lune : $(10, 0)$
- vitesse initiale du satellite : $2,61$

Résoudre numériquement le système d'équations différentielles, à l'aide de la fonction `dsolve(...,type=numeric)`

Question 4. Tracer la trajectoire du satellite à l'aide de la fonction `odeplot`.

2.1.5 Moindres carrés

Beaucoup de phénomènes physiques sont représentés par des lois linéaires. Au cours d'une expérience, on mesure une grandeur physique en fonction d'une autre (par exemple le potentiel électrique pour différentes valeurs d'intensité). On cherche alors à déterminer le coefficient directeur et l'ordonnée à l'origine caractéristique de la loi observée (par exemple pour déterminer la résistance).

Question 1. On représente les objets point et droite en dimension 2 par des tableaux. Un point sera défini à l'aide de la procédure `point` suivante :

```
# definit un point d'abscisse x et d'ordonnee y
point:=proc(x,y)
  array(1..2,[x,y]));
end;
```

On manipulera ces points à l'aide des deux primitives suivantes :

```
# retourne l'abscisse d'un point definit par point
abscisse:=proc(p)
   p[1];
end;

# retourne l'ordonnee d'un point definit par point
ordonnee:=proc(p)
   p[2];
end;
```

De même, pour manipuler les droites, on définit les procédures suivantes :

```
#definit une droite de pente a et d'ordonnee a l'origine b
droite:=proc(a,b)
   array(1..2,[a,b]);
end;

#retourne la pente d'une droite d
pente:=proc(d)
   d[1];
end;

#retourne l'ordonnee a l'origine d'une droite d
origine:=proc(d)
   d[2];
end;
```

Écrire, en utilisant les primitives ci-dessus, une procédure `carredistance` qui calcule le carré de la distance euclidienne entre un point et une droite.

Question 2. On définit l'erreur entre un point et une droite comme étant le carré de la distance du point à la droite. L'erreur entre une droite et un ensemble de points est définie comme étant égale à la somme des erreurs entre chacun des points et la droite.
Soit `P[i]:=point(x[i],y[i])` et `L:=droite(a,b)`. Trouvez `b` en fonction de `a` de sorte que l'erreur entre `L` et les `P[i]` soit minimale. Ceci revient à minimiser :
`sum(carredistance(P[i],L),i=1..N)`.
On supposera que ce minimum existe.

Question 3. Calculer la droite de moindres carrés de pente a et d'ordonnée à l'origine b pour un ensemble de N points de coordonnées (x_i, y_i) revient à calculer :

$$a = \frac{N \sum_{i=1}^{N} x_i y_i - \sum_{i=1}^{N} x_i \sum_{i=1}^{N} y_i}{N \sum_{i=1}^{N} {x_i}^2 - (\sum_{i=1}^{N} x_i)^2} \tag{3}$$

$$b = \frac{\sum_{i=1}^{N} y_i - a \sum_{i=1}^{N} x_i}{N} \tag{4}$$

Pour un ensemble de points donnés (sous forme de liste), écrire une procédure `moindresCarres` qui retourne la droite de moindres carrés.

Question 4. Quel inconvénient l'exemple suivant met-il en avant ?

```
moindresCarres([point(2,2),point(4,4),point(8,8),point(12,12),
                point(2.7,5.9),point(3,6),point(3.2,6.2)]);
```

Que proposez-vous pour y remédier ?

2.1.6 Équation de Van der Pool

On se propose d'étudier l'équation différentielle non linéaire suivante [1] :

$$\frac{d^2y(t)}{dt^2} + \mu(y(t)^2 - 1)\frac{dy(t)}{dt} + y(t) = 0 \tag{5}$$

Cette équation décrit un oscillateur entretenu à un degré de liberté caractérisé par un coefficient de frottement μ et dont les pertes d'énergie sont compensées par un apport extérieur. On choisit les conditions initiales suivantes :

$$\frac{d^2y(0)}{dt^2} = 0, \frac{dy(0)}{dt} = 0, y(0) = 1 \tag{6}$$

Question 1. Donnez un développement limité de la solution au voisinage de la date d'origine sous la forme d'une série (`dsolve(...,series)`) dans le cas où $\mu = 3$. On tracera la courbe correspondante pour $t \in [0..2]$.

Question 2. Donnez une solution numérique de l'équation au voisinage de la date d'origine (`dsolve(...,numeric)`) dans le cas où $\mu = 3$. On tracera la courbe correspondante pour $t \in [0..2]$.

Question 3. Comparez les résultats des questions 1 et 2. Pour cela, on affichera les courbes obtenues précédemment sur le même graphe (on utilisera `plots[display]`). Quelle méthode vous semble la mieux adaptée pour des calculs sur de longues périodes ?

1. Problème proposé initialement par F. Keller

Question 4. Tracez la solution de l'équation pour $t \in [0..40]$ dans les cas suivants :

- Frottements importants ($\mu = 10$)
- Frottements moyens ($\mu = 1$)
- Frottements faibles ($\mu = 0.1$)

Dans quel cas l'oscillateur prend-il le plus rapidement son régime permanent?

2.1.7 Interaction électrostatique

L'électrostatique est l'étude des interactions entre des particules chargées immobiles dans le repère d'un observateur. L'interaction électrostatique est proche de l'interaction gravitationnelle et possède les propriétés des champs newtoniens en $\frac{1}{r^2}$.

Question 1. On définit le potentiel électrostatique en un point $\vec{r}$, pour un ensemble de I particules de charges q_i et placées en $\vec{r_i}, i = 1 \ldots I$:

$$V(\vec{r}) = \frac{1}{4\pi\epsilon_0} \sum_{i=1}^{I} \frac{q_i}{|\vec{r} - \vec{r_i}|} \tag{7}$$

Écrire une procédure `potentiel_electrostatique` qui calcule le potentiel électrostatique V en un point (x, y) du plan, pour un ensemble quelconque de I particules données par q_i, $r_i = (\vec{x_i}, y_i), i = 1 \ldots I$. On supposera que les unités ont été choisies de sorte que $\frac{1}{4\pi\epsilon_0} = 1$.

Question 2. Tracer les courbes de potentiel, d'équipotentielle pour les ensembles de particules suivants :

- cas 1 : une particule de charge -1 placée en (-1,0) et une particule de charge 1 placée en (1,0),
- cas 2 : une particule de charge 1 placée en (-1,0) et une particule de charge 1 placée en (1,0),
- cas 3 : une particule de charge -1 placée en (-1,0), une particule de charge -1 placée en (1,0) et une particule de charge 2 placée en (0,0),
- cas 4 : une particule de charge 1 placée en (-1,0), une particule de charge 1 placée en (1,0), une particule de charge 1 placée en (0,-1), une particule de charge 1 placée en (0,1).

Étant donné que la valeur du potentiel tend vers $\pm\infty$ au voisinage des particules, on bornera la valeur du potentiel en chaque point par deux valeurs arbitraires v_{max} et v_{min} (à l'aide des fonctions `min` et `max`).

Question 3. On définit le champ électrostatique en un point $\vec{r}$, pour un ensemble de I particules de charges q_i et placées en $\vec{r_i}, i = 1 \ldots I$:

$$\vec{E}(\vec{r}) = \frac{1}{4\pi\epsilon_0} \sum_{i=1}^{I} q_i \frac{\vec{r} - \vec{r_i}}{|\vec{r} - \vec{r_i}|^3}. \tag{8}$$

Écrire une procédure `champ_electrostatique` qui calcule numériquement le champ électrostatique E en un point (x, y) du plan, pour un ensemble quelconque de particules.

Question 4. On désire obtenir une représentation des lignes de champ du champ électrostatique. Pour cela, on représente les directions des vecteurs champ sans tenir compte de leur intensité.

Écrire une fonction `normalise`, qui au vecteur champ résultat retourné par la procédure `champ_electrostatique` associe un vecteur de même direction, mais de norme 1. On utilisera ensuite la fonction `fieldplot` du package `plots` et la procédure `normalise` pour représenter les lignes de champ pour l'ensemble de charges suivant :

- Une particule de charge -1 placée en (-1,0) et une particule de charge 1 placée en (1,0)

Question 5. Comparez les résultats des questions 2 et 4. Que remarquez vous?

2.2 Corrections

2.2.1 Équilibre d'une réaction chimique

On écrit les équations traduisant les conservations de matière et de charge, que l'on résout à l'aide de la fonction `solve` .

```
> solve({a=d,     # qtite de Fe
         b=e,     # Mn
         4*b=f,   # O
         c=2*f,   # H
         2*a-b+c=d*3+e*2 # + et -
        },
        {a,b,c,d,e,f}
       );

              {e = b, f = 4 b, b = b, c = 8 b, d = 5 b, a = 5 b}
```

On obtient une infinité de solutions. En fixant $b = 1$, on obtient la solution la plus simple :

```
> subs(b=1,");

                  {1 = 1, d = 5, f = 4, e = 1, c = 8, a = 5}
```

ce qui représente la réaction :

$$5Fe^{2+} + MnO_4^- + 8H^+ \rightleftharpoons 5Fe^{3+} + Mn^{2+} + 4H_2O \tag{9}$$

2.2.2 Loi de Kirchov

Question 1. On choisit un arbre de recouvrement du circuit et on applique les lois des nœuds et des mailles. On obtient 7 équations pour 7 inconnues, que l'on résout à l'aide de `solve` :

```
> res:=solve({i2*r1=E,
              i4*r=E,
              i5*r+i7*r=E,
              2*i6*r+i7*r=E,
              i1=i2+i3,
              i3=i4+i5+i6,
              i7=i5+i6,
              i1=i2+i4+i7},
              {i1,i2,i3,i4,i5,i6,i7});

                                                 E (8 r1 + 5 r)
    res := {i4 = E/r, i6 = 1/5 E/r, i1 = 1/5 --------------, i3 = 8/5 E/r,
                                                      r1 r

                                              E
        i5 = 2/5 E/r, i7 = 3/5 E/r, i2 = ----}
                                             r1
```

Question 2. Dans la question 1, nous avons obtenu une solution générale au problème. Le cas particulier où `r=r1` s'écrit donc simplement :

```
> subs({r=r1},res);

                E          E          E              E              E
     {i6 = 1/5 ----, i4 = ----, i2 = ----, i1 = 13/5 ----, i3 = 8/5 ----,
               r1         r1         r1             r1             r1

                  E              E
         i5 = 2/5 ----, i7 = 3/5 ----}
                  r1             r1
```

Question 3. On détermine la valeur de `r1` pour avoir `i1=10*i4`, à l'aide de la fonction `solve` :

```
> solve(subs(res,i1)=10*subs(res,i4),r1);
                                             5/42 r
```

On peut également écrire :

```
> solve(res union {i1=10*i4},r1);
```

2.2.3 Projectile

Question 1. Les deux équations en x et y sont indépendantes :

```
> dsolve({diff(diff(x(t),t),t)=0,x(0)=0,D(x)(0)=v0*cos(alpha)},x(t));

                              x(t) = v0 cos(alpha) t
> pos_x:=subs(",x(t));
                              pos_x := v0 cos(alpha) t
```

On procède de la même façon pour la trajectoire en y :

```
> dsolve({diff(y(t),t$2)=-g,y(0)=h,D(y)(0)=v0*sin(alpha)},y(t));

                                        2
                      y(t) = - 1/2 g t  + h + v0 sin(alpha) t
> pos_y:=subs(",y(t));
                                      2
                    pos_y := - 1/2 g t  + h + v0 sin(alpha) t
```

La trajectoire obtenue est donc parabolique.

Question 2. On cherche à quel moment le projectile touche le sol :

```
> solve({pos_y=0},t);
                                      2           2         1/2
                  - v0 sin(alpha) + (v0  sin(alpha)  + 2 g h)
           {t = - ---------------------------------------------},
                                       g

                                        2           2         1/2
                    - v0 sin(alpha) - (v0  sin(alpha)  + 2 g h)
               {t = - ---------------------------------------------}
                                       g
```

On obtient deux solutions. Celle qui nous intéresse est celle pour laquelle $t > 0$ (on ne remonte pas dans le temps).

```
>  tmax:=subs(op(2,["]),t);
                                        2           2         1/2
                    - v0 sin(alpha) - (v0  sin(alpha)  + 2 g h)
            tmax := - ---------------------------------------------
                                       g
```

La position du projectile est en abscisse, dans le repère choisi :

```
> subs({t=tmax},pos_x);
                                          2           2         1/2
          v0 cos(alpha) (- v0 sin(alpha) - (v0  sin(alpha)  + 2 g h)   )
        - ------------------------------------------------------------
                                       g
```

Question 3. Une condition nécessaire pour q'un point soit extremum d'une fonction est que la dérivée de la fonction soit nulle en ce point :

```
> t_extremum:=solve(diff(pos_y,t)=0,t);
                                      v0 sin(alpha)
                        t_extremum := -------------
                                            g
```

Il n'y a qu'un seul extremum. On a de plus supposé que $\alpha_0 \in [0, \frac{\pi}{2}]$ et $g > 0$, on en déduit (la trajectoire est parabolique et $y''(t_{extremum}) < 0$) que cet extremum est un maximum. La hauteur maximale atteinte est alors :

```
> subs(t=t_extremum,pos_y);
                                    2           2
                                  v0  sin(alpha)
                             1/2 --------------- + h
                                        g
```

Question 4. On intègre la hauteur du projectile sur la durée du vol ($t \in [0, t_{max}]$) :

```
> int(pos_y,t=0..tmax);
                                2           2          1/2
   1/6 (v0 sin(alpha) + (v0  sin(alpha)  + 2 g h)   )

          2           2                     2           2         1/2
      (v0  sin(alpha)  + v0 sin(alpha) (v0  sin(alpha)  + 2 g h)    + 4 g h)

         /  2
        /  g
       /
```

Question 5.

On affiche la trajectoire jusqu'au moment où le projectile touche le sol (Fig. 5) :

```
> val_num:={g=10,v0=10,alpha=Pi/4,h=2}:
  plot(subs(val_num,pos_y),t=0..subs(val_num,tmax));
```

La hauteur maximale est atteinte au temps t suivant :

```
> evalf(subs(val_num,t_extremum));
                                  .7071067810
```

2.2.4 Apollo 8

Question 1.

```
distance:=(x1,y1,x2,y2)->sqrt((x2-x1)^2+(y2-y1)^2);
```

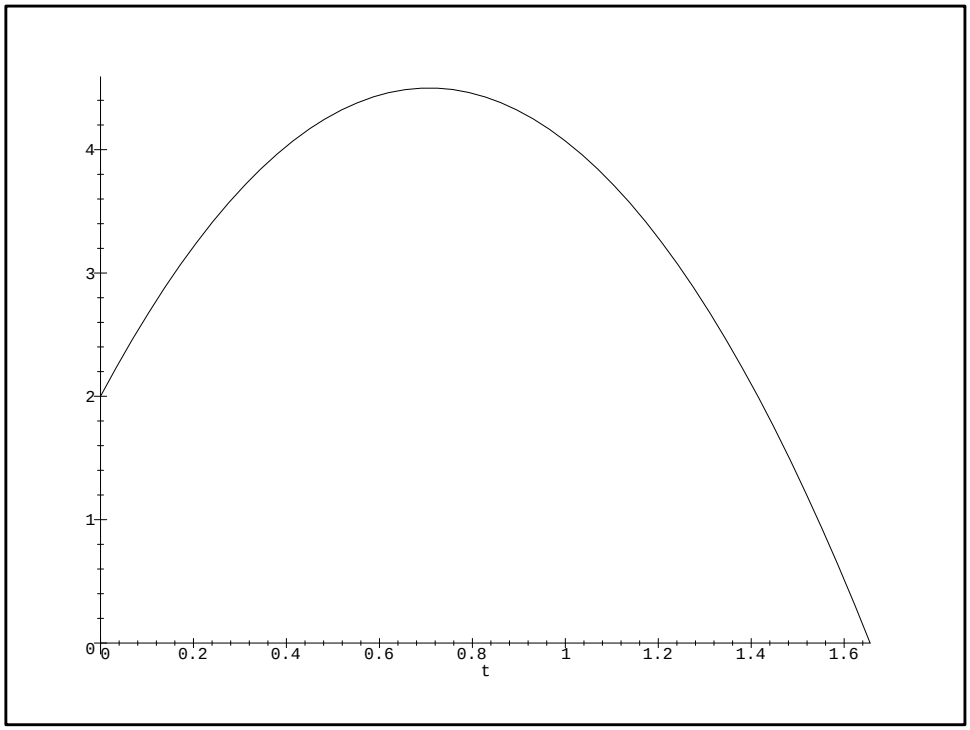


FIG. 5 –

Question 2. On note $(x_0(t), y_0(t))$ la position du satellite au temps t, (x_1, y_1) celle de la terre et (x_2, y_2) celle de la lune. Les équations de la trajectoire en x et en y s'écrivent :

```
equations:={diff(x0(t),t$2)=m1*(x1-x0(t))/(distance(x0(t),y0(t),x1,y1)^3)+
                            m2*(x2-x0(t))/(distance(x0(t),y0(t),x2,y2)^3),
            diff(y0(t),t$2)=m1*(y1-y0(t))/(distance(x0(t),y0(t),x1,y1)^3)+
                            m2*(y2-y0(t))/(distance(x0(t),y0(t),x2,y2)^3)};
```

On note v_0 la vitesse initiale du satellite, R le rayon de la terre. Les conditions initiales s'écrivent en **Maple** de la façon suivante :

```
conditionsInitiales:={x0(0)=-R,y0(0)=0,
                      D(x0)(0)=0,D(y0)(0)=v0};
```

Question 3. On effectue la résolution numérique de l'équation :

```
equ:=subs({x1=0,y1=0,x2=10,y2=0,v0=2.61,m1=4,m2=1,R=1},
          equations union conditionsInitiales):
f:=dsolve(equ,{x0(t),y0(t)},type=numeric):
```

Maintenant nous pouvons connaître la position (approchée) du satellite pour t quelconque. Par exemple en $t = 1$ on a :

```
> f(1);
                                                d
      [t = 1, x0(t) = .5202001930014002, ---- x0(t) = 1.317769483335385,
                                               dt

                                           d
          y0(t) = .3595330701882939, ---- y0(t) = -2.443607838512814]
                                          dt
```

Question 4. Nous traçons la trajectoire à l'aide de **odeplot** (Fig. 6). Selon le même principe, la capsule Apollo 8 a contourné la lune.

```
plots[odeplot](f,[t,x0(t),y0(t)],0..30,numpoints=200,labels=[`t`,`x0(t)`,`y0(t)`]);
```

2.2.5 Moindres carrés

Question 1.

```
# calcule le carre de la distance euclidienne entre un point p defini
# par point et une droite l definie par droite.
carredistance:=proc(p,l)
  local ix;
  ix:=(-l[2]*l[1]+p[1]+p[2]*l[1])/(l[1]^2+1);
  normal((p[1]-ix)^2+(p[2]-(l[1]*ix+l[2]))^2);
end;
```

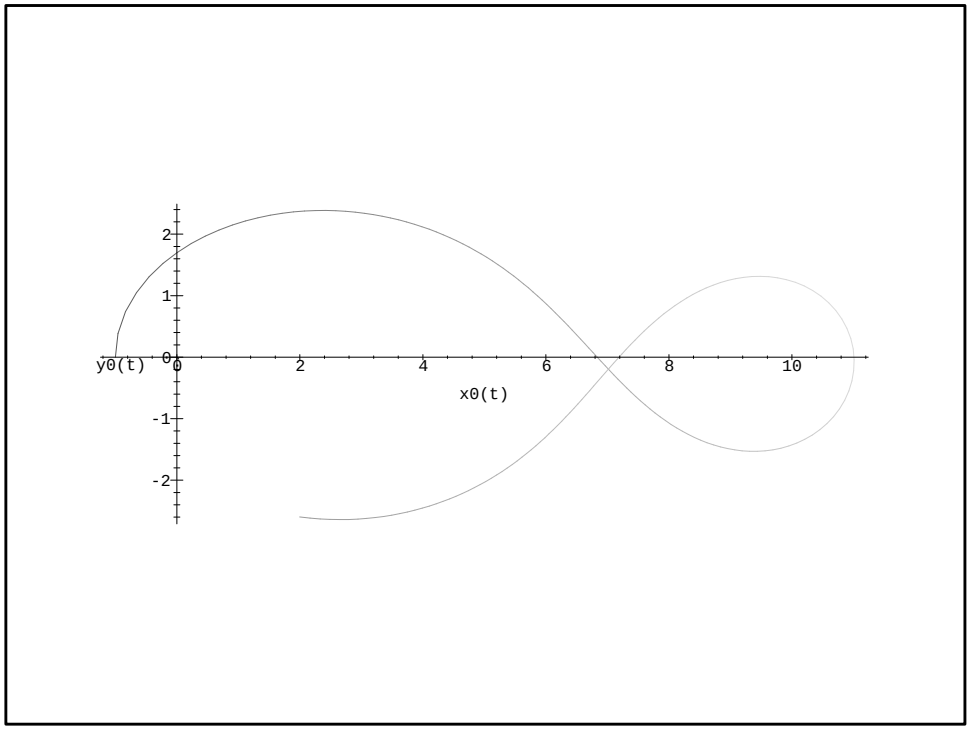


FIG. 6 – *Trajectoire du satellite*

Question 2. On cherche `b` tel que
`sum(carredistance(point(x[i],y[i]),droite(a,b)),i=1..N)` soit minimal. Une condition nécessaire pour qu'une fonction soit minimale en un point est que la dérivée soit nulle en ce point.

```
> normal(solve(diff(sum(carredistance(point(x[i],y[i]),droite(a,b)),
                        i=1..N),b),b));
```

$$\frac{\sum_{i=1}^{N}\left(a\,x_i - y_i\right)}{N}$$

La solution est unique.

Question 3.

```
# retourne la droite de moindres carres pour une liste L de point definis par
# point. La liste de points doit contenir au moins deux points.
moindres_carres:=proc(L:list)
  local sxy,sx,sx2,sy,n,i,xi,yi,a,b;
  sxy:=0;
  sx:=0;
  sx2:=0;
  sy:=0;
  n:=nops(L);
  if n=1 then
    ERROR(`nops(l)<2`);
  fi;
  for i to n do
    xi:=abscisse(L[i]);
    yi:=ordonnee(L[i]);
    sxy:=sxy+xi*yi;
    sx:=sx+xi;
    sy:=sy+yi;
    sx2:=sx2+xi^2
  od;
  a:=(n*sxy-sx*sy)/(n*sx2-sx^2);
  b:=(sy-a*sx)/n;
  droite(a,b)
end;
```

Pour pouvoir tester facilement la procédure, on définit la procédure suivante, qui permet de tracer sur un même graphe un ensemble de points et la droite de moindres carrés qui lui est associée :

```
affiche:=proc(liste_points:list)
  d1:=plot(liste_points,style=point):
  a:=moindrescarres(liste_points):
  d2:=plot(a[1]*x+a[2],x=0..10):
  plots[display]([d1,d2]);
end;
```

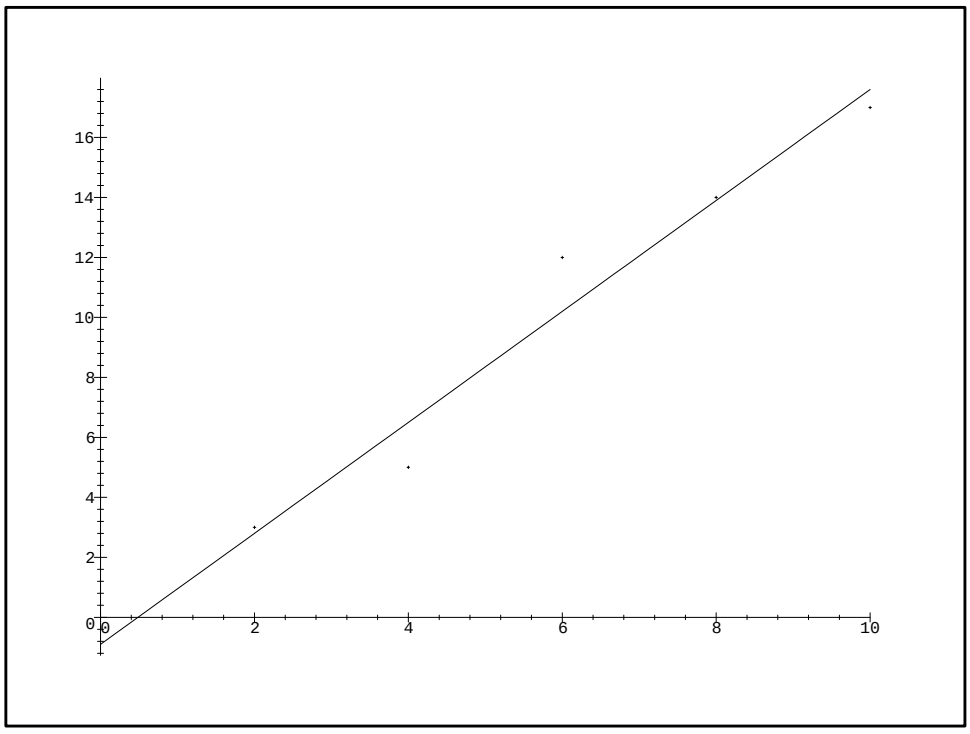


FIG. 7 –

La figure 7 montre le résultat de l'appel suivant :

```
affiche([point(2,3),point(10,17),point(4,5),point(6,12),point(8,14)]);
```

Question 4. L'exemple (Fig. 8) montre un inconvénient de la méthode. Les points jouent tous la même importance dans le calcul de la droite de moindres carrés. Les trois points très proches, qui représente quasiment la même mesure, attirent fortement la droite. Un seul point attirerait moins la droite. Le résultat dépend donc de la façon dont sont prises les mesures. Pour remédier à cet inconvénient, il faut soit imposer des conditions sur la prise des mesures (le même intervalle entre les mesures), soit modifier la méthode pour pondérer l'importance des différents points (par exemple en fonction de l'intervalle entre la mesure

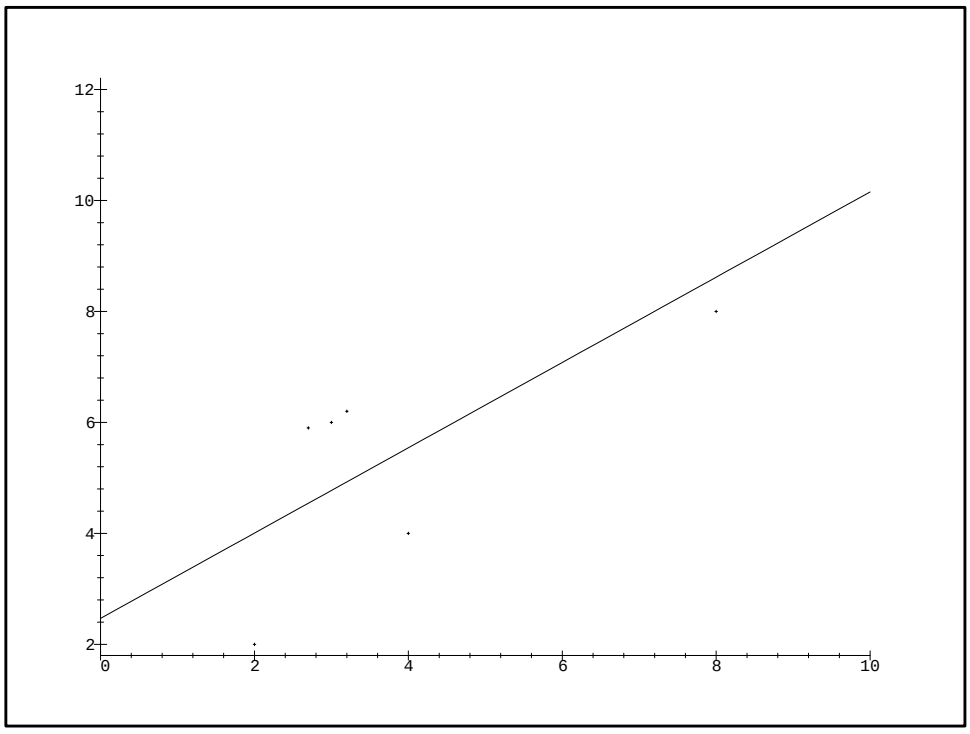


FIG. 8 –

précédente et la suivante). La méthode n'est pas bien adaptée au cas ou l'on connaît certaines conditions sur la droite, par exemple qu'elle passe par zéro.

2.2.6 Équation de Van der Pool

Question 1. On définit tout d'abord l'équation :

```
> equation_vdp:=diff(y(t),t$2)+mu*(y(t)^2-1)*diff(y(t),t)+y(t)=0;

                    /   2      \
                    |  d       |           2      /  d      \
     equation_vdp := |----- y(t)| + mu (y(t)  - 1) |---- y(t)| + y(t) = 0
                    |   2      |                  \ dt      /
                    \ dt       /
```

On définit les conditions initiales :

```
> conditions:=(D@@1)(y)(0)=0,D(y)(0)=0,y(0)=1;

                conditions := D(y)(0) = 0, D(y)(0) = 0, y(0) = 1
```

On résout l'équation avec l'option **series**

```
> dsolve({subs(mu=3,equation_vdp),conditions},y(t),type=series);
                                  2          4          5      6
                 y(t) = 1 - 1/2 t  + 1/24 t  - 3/20 t  + O(t )
```

Pour afficher le résultat, il faut convertir la série en polynôme :

```
d1:=plot(convert(subs(",y(t)),polynom),t=0..2):
d1;
```

Question 2. On résout l'équation avec l'option **numeric**.

```
> res3:=dsolve({subs(mu=3,equation_vdp),conditions},y(t),type=numeric):
```

On obtient une procédure **res3** qui permet de calculer la solution de l'équation en tout point. Par exemple en $t = 1$:

```
> res3(1);
                                                d
       [t = 1, y(t) = .3768595013236257, ---- y(t) = -1.738454725640297]
                                                dt
```

On utilise **odeplot** pour tracer la solution :

```
> d2:=plots[odeplot](res3,[t,y(t)],0..2,numpoints=200):
> d2;
```

Question 3. Pour afficher sur le même graphe les résultats précédents (Fig. 9) :

```
> plots[display]([d1,d2]);
```

On remarque que les résultats ne sont pas les mêmes. La solution utilisant **series**(la courbe qui tend vers $-\infty$) n'est en effet valable qu'au voisinage de zéro. Pour afficher la solution sur de longues périodes, il faut donc utiliser l'option **numeric**.

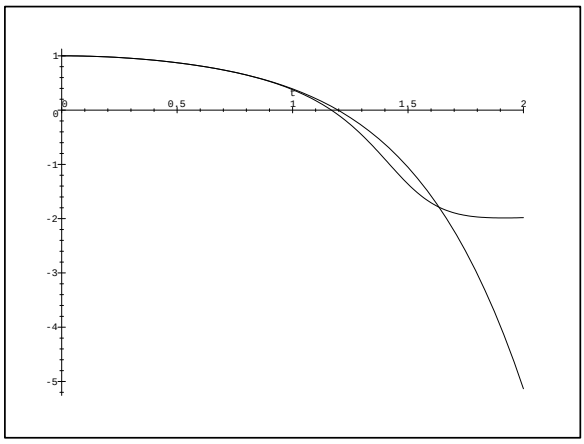


FIG. 9 – $\mu = 3, t \in [0..2]$, *options* `series` *et* `numeric`

Question 4. On définit une procédure **`affiche`** qui permet de tracer la solution pour différentes valeurs de μ (Fig. 11 et 10) :

```
affiche:=proc(val)
  res:=dsolve({subs(mu=val,equation_vdp),conditions},y(t),type=numeric):
  plots[odeplot](res,[t,y(t)],0..40,numpoints=200);
end;
affiche(10);
affiche(1);
affiche(0.1);
```

Plus le frottement est important, et plus l'oscillateur prend vite son régime permanent.

2.2.7 Interaction électrostatique

Question 1. On commence par définir une représentation pour les particules. On choisit ici de représenter chaque particule par une liste dont le premier élément est la charge de la particule, le second sa position, la position étant une liste dont le premier élément correspond à l'abscisse, le second à l'ordonnée de la particule représentée.

```
# set_particule definit une representation pour une particule de charge q
# et de position (x,y) dans le plan
set_particule:=proc(q,x,y)
  [q,[x,y]];
end;

# charge retourne la charge d'une particule definie a l'aide de set_particule
```

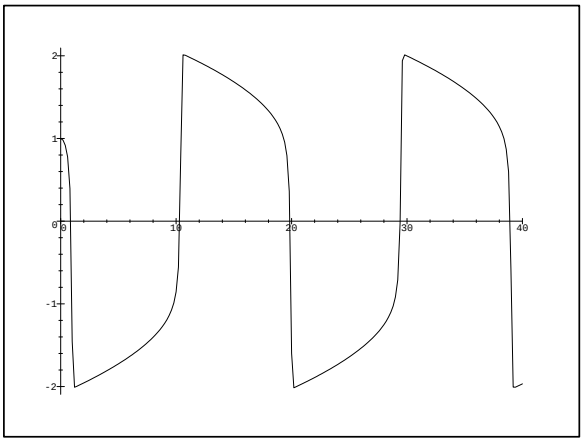


FIG. 10 – $\mu = 10, t \in [0..40]$

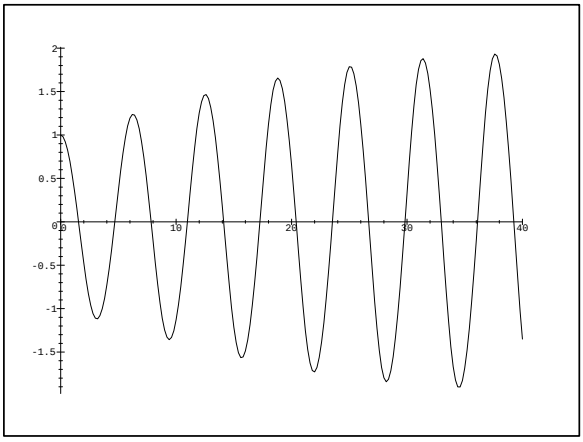


FIG. 11 – $\mu = 0.1, t \in [0..40]$

```
charge:=proc(p)
  p[1];
end;

# pos_x retourne l'abscisse d'une particule definie a l'aide de set_particule
pos_x:=proc(p)
  p[2][1];
end;

# pos_y retourne l'ordonnee d'une particule definie a l'aide de set_particule
pos_y:=proc(p)
  p[2][2];
end;
```

La procédure **potentiel_electrostatique** s'écrit alors :

```
# potentiel_electrostatique calcule la valeur numerique du champ
# electrostatique en un point (x,y) defini par une liste de
# particules. Les elements de la liste particules doivent etre definis
# a l'aide de set_particules. Le potentiel ne peut etre calcule aux
# points ou sont placees les particules.
potentiel_electrostatique:=proc(particules,x,y)
   n:=nops(particules);
   s:=0;
   for i to n do
      rx:=x-pos_x(particules[i]);
      ry:=y-pos_y(particules[i]);
      r:=sqrt(rx*rx+ry*ry);
      s:=evalf(s+charge(particules[i])/r);
   od;
end;
```

Dans le cas d'une particule de charge -1 placée en (-1,0) et d'une particule de charge 1 placée en (1,0), on peut s'attendre aux résultats suivants :

- Le potentiel électrostatique doit être positif quand on est plus près de la charge positive que de la charge négative :

```
mes_particules:=[set_particule(1,1,0),set_particule(-1,-1,0)];
> potentiel_electrostatique(mes_particules,1,1);
                                  .5527864044
> potentiel_electrostatique(mes_particules,-3,0);
                                  -.2500000000
```

- Le potentiel électrostatique doit être nul aux points équidistants des deux particules :

```
>  potentiel_electrostatique(mes_particules,0,1);
                                        0
```

- La valeur du potentiel est d'autant plus élevée que l'on est proche des particules :

```
> potentiel_electrostatique(mes_particules,1,0.5);
                                  1.514928750
```

On peut effectuer des tests similaires avec un nombre plus important de particules.

```
> potentiel_electrostatique([set_particule(1,1,0),set_particule(-1,-1,0),
> set_particule(1,0,1),set_particule(-1,0,-1)],1,1);
                                1.105572808
```

Question 2. On définit une procédure `borne` qui permet de borner un résultat :

```
# borne retourne vmin si res est inferieur a vmin, et vmax si res est
# superieur a vmax.
borne:=proc(res)
   max(vmin,min(res,vmax));
end;
```

Le tracé du potentiel se fait simplement à l'aide de `plot3d`.

- cas 1 : une particule de charge -1 placée en (-1,0) et une particule de charge 1 placée en (1,0). Pour obtenir la forme générale de la courbe de potentiel électrostatique (Fig. 12) :

```
plot3d(borne(potentiel_electrostatique([set_particule(1,1,0),
                                        set_particule(-1,-1,0)],
                                       x,y)),
       x=-2..2,y=-2..2,
       grid=[50,50],
       style=PATCH,
       labels=[`x`,`y`,`V(x,y)`],
       axes=FRAME,
       shading=ZGREYSCALE,
       title=`cas 1 : potentiel electrostatique `);
```

Pour afficher les courbes d'équipotentiel (Fig. 13) :

```
plot3d(borne(potentiel_electrostatique([set_particule(1,1,0),
                                        set_particule(-1,-1,0)],
                                       x,y)),
       x=-2..2,y=-2..2,
       grid=[50,50],
       style=PATCHCONTOUR,
       labels=[`x`,`y`,`V(x,y)`],
       axes=FRAME,
       contours=50,
       shading=ZGREYSCALE,
       title=`cas 1 : courbes d'equipotentiel`);
```

- cas 2, 3 et 4 : On procède de la même façon.

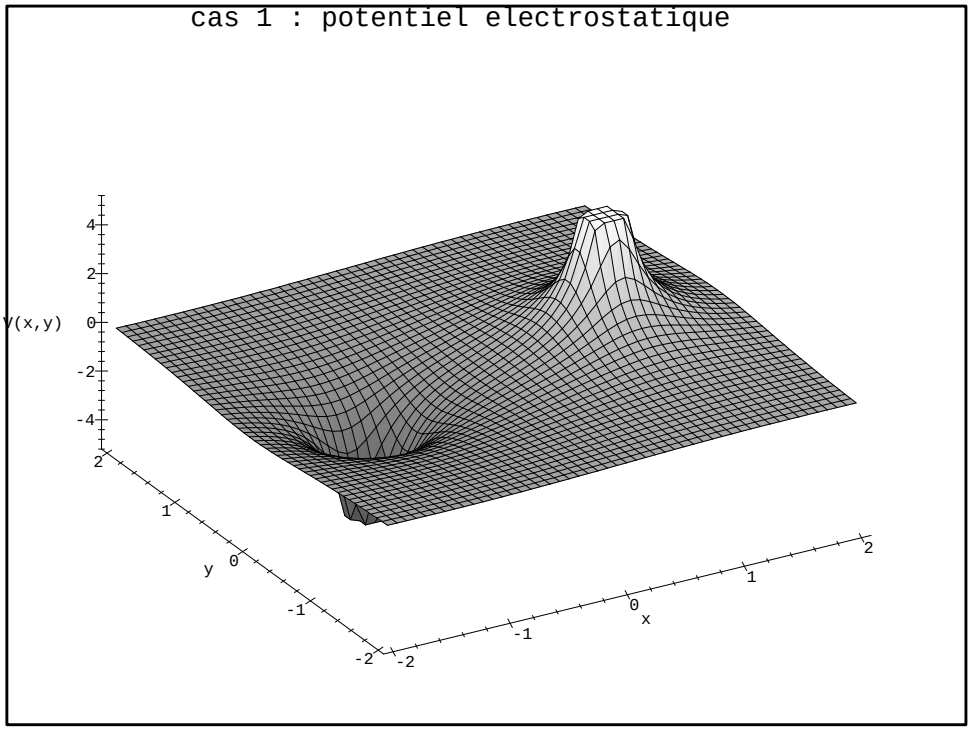


FIG. 12 –

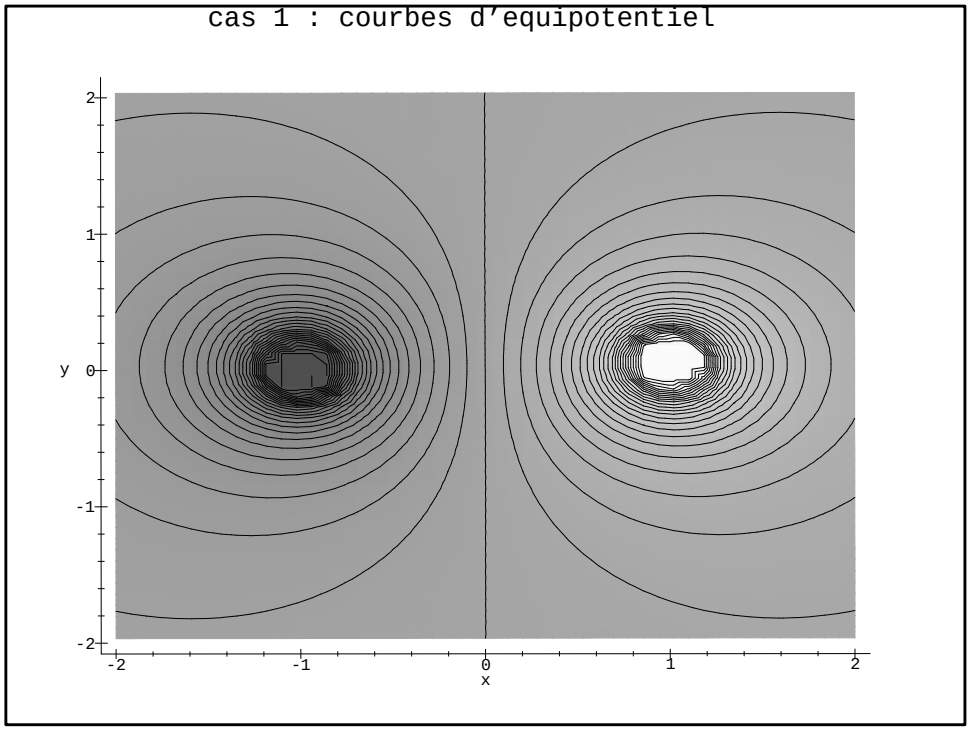


FIG. 13 –

Question 3. On définit la procédure `champ_electrostatique` :

```
# champ_electrostatique calcule la valeur numerique du champ
# electrostatique en un point (x,y) defini par une liste de particules
# particules. Les elements de la liste particules doivent etre definis
# a l'aide de set_particules. La valeur du vecteur champ est retournee
# dans une liste dont le premier element correspond a la premiere composante
# et le second a la seconde composante du vecteur champ.
# Le vecteur champ ne peut etre calcule aux points ou sont placees les
# particules.
champ_electrostatique:=proc(particules,x,y)
   n:=nops(particules);
   sx:=0;
   sy:=0;
   for i to n do
      rx:=x-pos_x(particules[i]);
      ry:=y-pos_y(particules[i]);
      r:=sqrt(rx*rx+ry*ry);
      sx:=evalf(sx+charge(particules[i])*rx/r^3);
      sy:=evalf(sy+charge(particules[i])*ry/r^3);
   od;
   [sx,sy];
end;
```

Question 4. On écrit la procédure de normalisation des vecteurs champ calculés :

```
# associe a un vecteur v (defini par une liste dont le premier element
# correspond a la premiere composante et le second a la seconde composante
# du vecteur), un vecteur (sous forme de liste) de norme 1 et de meme
# direction que v
normalise:=proc(v)
   local norme;
   norme:=sqrt(v[1]^2+v[2]^2);
   [v[1]/norme,v[2]/norme];
end;
```

On utilise `fieldplot` pour tracer les directions du champ électrostatique (Fig. 14) :

```
plots[fieldplot](normalise(champ_electrostatique([set_particule(1,1,0),
                                                  set_particule(-1,-1,0)],
                                                 x,y)),
                 x=-2..2,y=-2..2);
```

Question 5. On remarque que les lignes de champ sont perpendiculaires aux lignes d'équipotentiel. La direction des lignes de champ en un point correspond à celle de plus grande pente du potentiel électrostatique.

3 Questions fréquemment posées

3.1 Comment faire pour interrompre un très long calcul

Sous windows, appuyer sur le bouton stop de la barre d'icônes. Si cela ne marche pas, tuez `Maple` et recommencez tout...

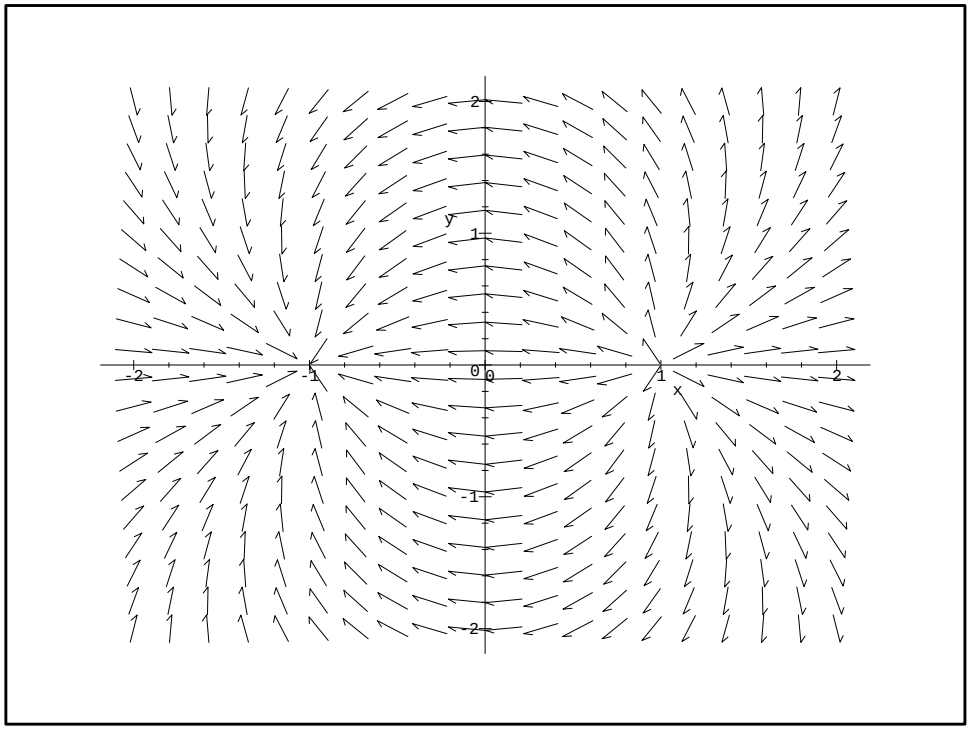


FIG. 14 – *Lignes de champ électrostatique*

3.2 Plus rien ne s'évalue. Que faut-il faire ?

Vous avez certainement oublié de fermer une procédure ou une boucle. Tapez `end;`

3.3 Faut-il utiliser une table ou un tableau ?

Si les index des éléments à stocker sont entiers et que vous connaissez le nombre maximal d'éléments à stocker, utilisez un tableau (`array`). Sinon utilisez une table.

3.4 Faut-il utiliser une boucle `for`, `seq`, ou `sum` ?

Si le résultat est sous forme numérique, utilisez de préférence `for` ou `convert([seq(..,..)],'+')`. Dans le cas d'un calcul symbolique – vous ne connaissez pas le nombre d'éléments à sommer par exemple –, utilisez `sum`.

3.5 Faut-il utiliser `op(i,l)` ou `l[i]` pour récupérer le i^{eme} élément d'une liste ?

Les deux sont corrects, mais préférez `l[i]` qui est plus lisible. Dans un cas on récupère le ième élément de la liste, dans l'autre le ième opérande d'une expression.

3.6 Quand faut-il mettre ; ou : ?

Il faut mettre un ; ou : pour séparer deux instructions consécutives d'un même bloc d'instructions. Il ne faut donc pas en mettre après `proc()`, `do` et `then`. On tolère ; ou : après la dernière instruction d'un bloc d'instructions (pour permettre l'ajout de nouvelles instructions en fin de bloc), c'est à dire avant `od`, `fi` ou `end`.

3.7 Comment "récupérer" les valeurs retournées par `solve` ?

Utilisez `subs`. Par exemple :

```
> solve({a+2*b=25,a+b=7},{a,b});
                              {a = -11, b = 18}
> subs(",a);
                                     -11
```

3.8 Comment tracer le résultat d'un appel à `series` ?

Utilisez `convert(...,polynom)`

```
> series(x/(1-x-x^2), x=0);
                          2      3      4      5      6
                     x + x  + 2 x  + 3 x  + 5 x  + O(x )
> plot(convert(",polynom),x=0..2);
```

Références

[1] Bruce W. Char, Benton L. Leong, Keith O. Geddes, and Gaston H. Gonnet. *First leaves : a tutorial introduction to Maple V.* Springer, 1992.

[2] Jack-Michel Cornil and Philippe Testud. *Maple : introduction raisonnée l'usage de l'étudiant, de l'ingénieur et du chercheur.* Springer Verlag, 1995.

[3] James Harold Davenport, Yvon Siret, and Evelyne Tournier. *Calcul formel : systèmes et algorithmes de manipulations algbriques.* Masson, 1987.

[4] Claude Gomez, Bruno Salvy, and Paul Zimmermann. *Calcul formel : mode d'emploi : exemples en MAPLE.* Masson, 1995.

Unit´e de recherche INRIA Lorraine, Technopˆole de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhˆone-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399