INFO 153 : Calcul Formel et Sciences de la Matière

Université Lille 1
François Boulier

26 août 2008

Table des matières

Chapitre 1

Introduction

Équipe pédagogique

François	Boulier
Éric	Cochin
Jean–Claude	Depannemaecker
Raouf	Dridi
Mimoun	Ismaili
Damien	Jacob
Laurent	Labonnote
Éric	Louvergneaux
Augustin	van Groenendael
Maurice	Monnerville
Éric	Wegrzynowski
Léopold	Weinberg

Le cours « Calcul Formel et Sciences de la Matière » est d'abord un cours de programmation à l'usage d'étudiants se destinant à des études non informatiques : physique, chimie . . . On s'efforce tout au long du cours d'écrire de petits programmes.

Dans ce cours, on étudie des algorithmes scientifiques importants (pivot de Gauss, méthode de Newton, d'Euler etc . . .) et on développe un thème qui peut varier au fil des ans. Cette année, le thème choisi est l'analyse qualitative des systèmes d'équations différentielles.

Il s'agit aussi d'un cours de calcul formel : on utilise un logiciel (MAPLE) permettant de mener des calculs symboliques et numériques. Cet outil constitue un atout très important dans l'enseignement des algorithmes scientifiques : il permet d'illustrer très concrètement les rappels mathématiques, de bien mettre en évidence la différence entre les erreurs de méthode et les erreurs d'arrondis et enfin de mener des démonstrations assistées par ordinateurs. En particulier, on montre comment déterminer formellement l'ordre de plusieurs méthodes (Newton, Euler) en déléguant tous les calculs symboliques au logiciel.

Dans la première partie du cours, on rappelle les principes de base de la programmation et on montre comment ils se réalisent dans le langage de programmation du logiciel MAPLE : principaux

types et structures de données, structures de contrôle, écriture de fonction.

La deuxième partie constitue une introduction au paquetage d'algèbre linéaire de MAPLE. On y apprend aussi les techniques élémentaires de résolution de systèmes d'équations linéaires : le pivot de Gauss (pour les systèmes qui ont au moins autant d'inconnues que d'équations) et la méthode des moindres carrés (pour les systèmes qui ont plus d'équations que d'inconnues). On conclut par une introduction à la diagonalisation de matrices, qui constitue (entre autres) un outil très important pour la résolution de systèmes différentiels linéaires.

Dans la troisième partie du cours, on étudie deux grandes méthodes de résolution numérique d'équations : la méthode de Newton pour les équations non différentielles et la méthode d'Euler pour les équations différentielles. On généralise la méthode d'Euler à un système d'équations différentielles. On manipule également le solveur MAPLE d'équations différentielles qui est fondé sur des méthodes plus efficaces que la méthode d'Euler. Dans les deux cas (méthodes de Newton et d'Euler), on utilise les possibilités de calcul symbolique de MAPLE pour analyser formellement les vitesses de convergence des méthodes.

La quatrième partie est consacrée à l'analyse qualitative des équations différentielles autonomes. On commence par l'analyse qualitative d'une équation en une variable. On termine le cours par l'analyse qualitative d'un système de deux équations différentielles linéaires en deux variables. Ce dernier cours s'appuie fortement sur la diagonalisation de matrices étudiée préalablement.

Des prolongements naturels du cours seraient l'analyse qualitative des points fixes des systèmes différentiels non linéaires. On pourrait y introduire des techniques de résolution exacte de systèmes algébriques (pour la recherche des points fixes) et les principes de la linéarisation d'un système différentiel au voisinage d'un point (de façon à pouvoir appliquer les techniques vues précedemment). La résolution exacte de systèmes algébriques est étudiée dans le cadre de l'UE INFO 251 « systèmes polynomiaux : que signifie résoudre ? ».

En ce qui concerne l'algorithmique numérique, ce support de cours a fortement bénéficié des ouvrages [1] pour les méthodes de Newton et d'Euler (nombreux raisonnements élémentaires), [6, 5] pour les méthodes d'intégration d'équations différentielles (le support de cours est plus simple que le livre et rédigé en Français) et [2] pour des considérations historiques. Pour le calcul formel, on suggère les ouvrages [3], qui fournit une introduction en Français aux algorithmes de base, [4] qui décrit en détail plusieurs algorithmes importants du logiciel MAPLE ainsi que [8], plus récent.

La plupart des photographies incluses dans le support de cours proviennent du site [7] de l'université de Saint Andrews.

Chapitre 2

Programmation en MAPLE

On rappelle dans ce cours les principes fondamentaux de la programmation impérative et on montre comment les mettre en œuvre en MAPLE.

2.1 Nombres et structures de données

Le logiciel MAPLE permet de manipuler différents types de nombres. On passe en revue les principaux types. On s'intéresse pour chacun d'eux au problème du *coût* des quatre opérations en ressources de la machine (temps et mémoire) et au problème du test d'égalité à zéro.

Fondamentalement, ce test est la seule opération importante : un programmeur qui souhaiterait implanter un nouveau type de nombre pourrait toujours réaliser les quatre opérations trivialement (en ne simplifiant jamais les expressions). Par exemple, rien ne force à « simplifier » le produit $\sqrt{2} \cdot \sqrt{3}$ en $\sqrt{6}$. Mais ne pas simplifier les expressions ne fait que reporter les difficultés sur le test d'égalité à zéro puisqu'il faudra bien que ce test reconnaisse que $\sqrt{2} \cdot \sqrt{3} - \sqrt{6} = 0$ est une expression vraie.

2.1.1 Les nombres inexacts

On les appelle aussi « nombres à virgule flottante ». C'est le type de nombre le plus répandu en informatique. On le rencontre dans tous les langages de programmation. Les quatre opérations sont assez simples à implanter. Elles sont même câblées dans de nombreux microprocesseurs. Elles souffrent du problème des erreurs d'arrondi. Le test d'égalité à zéro n'a pas de sens en pratique en raison de ces erreurs. Il est donc souvent remplacé par un test « inférieur à ε » où ε est un réel positif dont le choix est souvent laissé à l'utilisateur.

Une particularité de MAPLE : le nombre de décimales peut être fixé arbitrairement grand.

```
a := sqrt (2.);
                    a := 1.414213562
a^2;
                      1.999999999
Digits := 30:
```

```
sqrt (2.);
            1.41421356237309504880168872421
```

2.1.2 Les nombres exacts

Les entiers et les rationnels

On rencontre le type « nombre entier » dans tous les langages de programmation mais il s'agit presque tout le temps de nombres entiers de taille limitée. Le logiciel MAPLE fournit des entiers et des rationnels arbitrairement grands. Ces nombres sont presque toujours représentés de façon interne dans une base de numération. Au moins lorsque les nombres sont petits, les ordinateurs appliquent les quatre opérations telles que nous les avons apprises à l'école primaire (pour les grands nombres, il existe des algorithmes de multiplication et de division sophistiqués).

Les quatre opérations prennent un temps qui augmente (non linéairement) avec le nombre de chiffres de leurs opérandes. Penser à la multiplication de deux nombres à n chiffres pratiquée avec l'algorithme bien connu : elle nécessite en gros n^2 multiplications de chiffres et donc un temps de calcul qui croît à peu près proportionnellement à n^2.

Le test d'égalité à zéro est facile dans une base de numération.

Les nombres algébriques

Un nombre est dit « algébrique » s'il est racine d'un polynôme en une indéterminée et à coefficients rationnels. Les nombres $\sqrt{2}$ et i sont algébriques, tous les rationnels aussi. Les nombres π et e ne le sont pas.

Il n'y a pas de représentation informatique standard pour les représenter. Une solution consiste à représenter un nombre algébrique α par un polynôme P de degré minimal dont α est racine plus un intervalle (ou un couple d'intervalles dans le cas de nombres algébriques complexes) permettant de distinguer α des autres racines de P. Anecdotiquement, avec une telle représentation, le produit de deux nombres est moins coûteux que leur somme.

Le test d'égalité à zéro existe mais il est si coûteux que le logiciel MAPLE ne le met pas systématiquement en œuvre bien qu'il en dispose.

```
sqrt (2);
                          1/2
                         2
sqrt (2)^2;
                          2

# Une réponse surprenante !
if sqrt (2) * sqrt (3) = sqrt (6) then
        OUI
else
        NON
fi;
                          NON
```

```
# Pourtant, le logiciel « connaît » la bonne réponse
simplify (sqrt (2) * sqrt (3) - sqrt (6));
                          0
```

Les autres

Passés les nombres algébriques, en général, on ne dispose plus d'algorithme permettant de tester l'égalité à zéro. Voici un exemple frappant (bien qu'il s'agisse de fonctions et non de nombres).

Théorème 1 *(Caviness, Richardson et Matijasevic, 1968, 1970)*
Soit $\mathscr{F}$ l'ensemble de toutes les formules qu'on peut construire avec des rationnels, π, un symbole x, les opérations $+$, $\times$, sin et abs. Par exemple

$$\sin(2\pi + 3), \quad |x| + 1/2, \quad \ldots$$

Il ne peut exister aucun algorithme qui prenne en entrée deux formules quelconques de $\mathscr{F}$ et décide si ces formules représentent la même fonction de $\mathbb{R}$ dans $\mathbb{R}$.

2.1.3 Calcul symbolique

Le logiciel MAPLE permet de manipuler des expressions symboliques (calcul formel). En MAPLE, un symbole est une variable à laquelle aucune valeur n'est affectée.

```
P := (a + b)^2;
                                2
                     P := (a + b)
expand (P);
                     2             2
                    a  + 2 a b + b

Q := subs (a = 3, P);
                                2
                     Q := (3 + b)

P;
                              2
                       (a + b)
```

2.1.4 Principales structures de données

Il s'agit des « séquences », « listes » et « ensembles ».

Définition 1 *Une séquence est une suite d'objets séparés par des virgules. La virgule est l'opérateur de concaténation des séquences. La séquence vide est nommée NULL.*

La fonction *seq* permet de construire des séquences.

```
s1 := 3, x + 1, 2;
                s1 := 3, x + 1, 2
s2 := 2, x*y;
                  s2 := 2, x*y
seq (i^2, i = 1 .. 5);
               1, 4, 9, 16, 25
s2, NULL;
                     2, x*y
s3 := s1, s2;
             s3 := 3, x + 1, 2, 2, x*y
```

Définition 2 *Une liste est une séquence délimitée par des crochets.*

La fonction *op* permet d'obtenir les opérandes d'une liste (la séquence de ses éléments). Le fonction *nops* permet d'obtenir le nombre d'opérandes (d'éléments) d'une liste.

```
L3 := [ s3 ];
            L3 := [3, x + 1, 2, 2, x*y]
nops (L3);
                       5
L3 [2];
                     x + 1
op (L3);
                 3, x + 1, 2, 2, x*y
```

Définition 3 *Un ensemble est une séquence encadrée par des accolades. Deux ensembles qui contiennent les mêmes éléments sont égaux.*

Les ensembles MAPLE ressemblent beaucoup aux ensembles mathématiques donnés en extension : l'ordre des éléments n'est pas spécifié et chaque élément n'apparaît qu'une fois. Lorsqu'on convertit une séquence en ensemble, l'ordre des éléments de la séquence peut donc être modifié et les doublons sont supprimés. Une précision sur l'ordre des éléments d'un ensemble : si les variables E et F contiennent deux ensembles égaux alors l'ordre des éléments de E et de F est le même mais cet ordre peut changer d'une session MAPLE à une autre. Les fonctions *op* et *nops* s'appliquent aussi aux ensembles.

```
# L'ordre des éléments changé et le doublon a été retiré.
E3 := { s3 };
              E3 := { 2, x + 1, 3, x*y }

nops (E3);
                       4

op (E3);
                 2, x + 1, 3, x*y

E3 [2];
                      x + 1
```

```
# le premier paramètre est un ensemble de deux équations,
# le second un ensemble de deux indéterminées,
# la réponse est un ensemble de deux expressions.

solve ( { 2*x+3*y=5, x+y=7 }, { x, y } );

                {y = -9, x = 16}
```

Les fonctions *op* et *nops* permettent d'accéder aux opérandes d'expressions générales.

```
expr := f(g(x+1,5),17);
             expr := f(g(x + 1, 5), 17)
nops (expr);
                          2
op (expr);
                  g(x + 1, 5), 17

op (1, op (1, expr));
                      x + 1
```

2.2 Boucles for

L'instruction *seq*, vue au cours précédent, permet de réaliser des *itérations*. Un calcul « itératif » est un calcul « en boucle ». Il existe des itérations que *seq* ne permet pas de faire (facilement). Exemple. On souhaite calculer les 4 premiers termes de la suite définie par

$$u_0 = 1, \quad u_n = 2\,u_{n-1} + 3.$$

Pourquoi *seq* est inadapté : il faut réutiliser à l'étape n (à la enième « itération ») le résultat du calcul effectué à l'étape $n-1$. Une solution consiste à calculer les termes les uns après les autres en utilisant des variables indicées :

```
u[0] := 1;
u[1] := 2*u[0] + 3;
u[2] := 2*u[1] + 3;
u[3] := 2*u[2] + 3;
```

Cette suite d'instruction peut se résumer au moyen d'une boucle *for*.

```
u[0] := 1;                  # initialisation
for n from 1 to 3 do
   u[n] := 2*u[n-1] + 3     # corps de la boucle
od
```

Exemple. Affecter à une variable S la somme des éléments d'une liste L. On peut modéliser ce problème sous la forme d'une suite définie par récurrence :

$$u_1 = 0, \quad u_{n+1} = u_n + L[n] \quad (1 \leq n \leq \textit{nops}(L))$$

```
u[1] := 0;
for n from 1 to nops (L) do
    u[n+1] := u[n] + L[n]
od;
S := u[nops(L) + 1];
```

Cette solution est maladroite parce que toutes les sommes intermédiaires sont stockées dans des variables. Voici une autre solution, sans variables indicées, qui évite cette difficulté :

```
S := 0;
for n from 1 to nops (L) do
    S := S + L[n]
od;
```

On peut représenter l'exécution de la boucle sous la forme d'un tableau. Prenons $L = [3, 1, 7]$. Chaque ligne du tableau contient une valeur de n et une valeur de S. On y lit la valeur de S *au début* de l'étape n. Pour pouvoir lire la valeur de S à la fin du calcul, on ajoute une ligne « fictive » $n = 4$ à la fin du tableau. En effet, la valeur de S *à la fin* de l'étape $n = 3$ est égale à la valeur de S *au début* de l'étape fictive $n = 4$, c'est–à–dire 11.

	n	valeur de S au début de l'étape n
	1	0
	2	3
	3	4
arrêt au début de l'étape $n = 4$	4	11

Définition 4 *(invariant de boucle for) On appelle* invariant de boucle for *une propriété qui est vraie au début de chaque itération (une propriété satisfaite à chaque ligne du tableau précédent).*

Voici un invariant de la boucle donnée en exemple :

$$S = \sum_{i=1}^{n-1} L_i.$$

Cet invariant est « utile » parce qu'il permet de prouver la boucle : elle s'arrête au début de l'étape $n = \mathit{nops}\,(L) + 1$. D'après l'invariant, S contient la somme de tous les éléments de la liste L : la boucle est correcte[1]. Variante de boucle *for* :

```
S := 0;
for n from nops (L) to 1 by -1 do
    S := S + L[n]
od;
```

[1] Par comparaison, $1 + 1 = 2$ est aussi un invariant de boucle mais beaucoup moins utile.

2.3 Procédures et fonctions

Le langage de programmation MAPLE offre la possibilité de définir des fonctions. Par exemple la fonction f dont voici une *spécification*

$$\begin{array}{rccc} f: & \mathbb{R} & \rightarrow & \mathbb{R} \\ & x & \mapsto & 3\,x^2 + 2\,x + 1 \end{array}$$

peut être *implantée* (*codée, réalisée, mise en œuvre*) en MAPLE par

```
f := proc (x)
    3*x^2 + 2*x + 1
end;
```

Sur l'exemple, *f* est l'*identificateur* de la fonction, *x* en est le *paramètre formel* et l'expression `3*x^2 + 2*x + 1` en est le *corps*. Une fois définie, une fonction peut être *appelée*.

```
f (1);
          6
```

L'expression $f(1)$ est un *appel de fonction* (appel à la fonction *f*). Le paramètre 1 est le *paramètre effectif* de cet appel de fonction. Le nombre 6 est la *valeur retournée* (ou encore le *résultat*) de l'appel de fonction.

2.3.1 Variables globales et paramètres formels

On appelle *variables globales* les variables qu'on peut modifier interactivement dans la feuille de travail lorsqu'on utilise MAPLE. Il n'y a aucun lien entre le paramètre formel *x* de la fonction *f* et la variable globale *x*. Preuve.

```
# La variable globale x vaut 3
x := 3:
x;
          3
# La variable locale x est modifiée par l'appel
f (1);
          6
# La variable globale n'a pas changé de valeur
x;
          3
```

La définition adoptée ci–dessus est particulière à MAPLE. La notion de variable globale existe dans la plupart des langages de programmation, avec des définitions différentes bien sûr.

2.3.2 Spécification et implantation

On distingue la *spécification* d'une fonction de son *implantation*. La *spécification* d'une fonction décrit ce que la fonction calcule en fonction des paramètres formels. C'est une réponse à la question *quoi ?*. Une spécification doit être claire, synthétique, facile à comprendre ...

L'*implantation* d'une fonction décrit comment le calcul est réalisé. C'est une réponse à la question *comment ?*. Le problème ici est de faire exactement ce que la spécification décrit. On essaie parfois d'obtenir une fonction rapide à l'exécution ou économe en mémoire. Parfois, on cherche plutôt à réaliser une fonction facile à relire et donc à modifier. Voici une deuxième implantation de la fonction f (mettant en œuvre ce qu'on appelle un schéma de Hörner) qui effectue moins d'opérations.

```
f := proc (x)
    ((3 * x) + 2) * x + 1
end;
```

2.3.3 Variables locales

Pour des fonctions plus compliquées que f, le calcul du résultat peut nécessiter l'utilisation de variables. Considérons la fonction

$$\begin{array}{rccc} \text{puissance} : & \mathbb{R} \times \mathbb{N} & \to & \mathbb{R} \\ & (n,\, s) & \mapsto & n^s. \end{array}$$

Voici une implantation possible :

```
puissance := proc (n, s)
    local k, resultat;
    resultat := 1;
    for k from 1 to s do
        resultat := resultat * n
    od;
    resultat
end;
```

La valeur retournée par une fonction est le résultat de la dernière instruction exécutée (ici, c'est le contenu de la variable *resultat*). Le mot-clef *local* indique que les variables *k* et *resultat* sont locales à la fonction (et même à l'appel de fonction). En d'autres termes, comme pour les paramètres formels, les variables locales *k* et *resultat* n'ont rien de commun avec les variables globales *k* et *resultat*. Un exemple suffit à le prouver.

```
resultat := 18:
# La variable globale "resultat" vaut 18
resultat;
           18
# La variable locale "resultat" est modifiée par l'appel
puissance (2, 5);
           32
# La variable globale "resultat" n'a pas changé de valeur
resultat;
           18
```

Construction de la fonction puissance. Ce paragraphe s'adresse aux étudiants qui manquent d'intuition pour écrire du code. Comment peut–on construire la fonction *puissance* si on manque d'intuition ? Eh bien on peut s'aider en choisissant un invariant de boucle. On part de l'idée naturelle qu'on va effectuer s multiplications par n (en gros : ça peut être $s-1$ ou $s+1$) et accumuler le résultat peu à peu dans une variable *resultat*. On aura besoin d'un indice de boucle k. Et on choisit un invariant. Par exemple :

$$\textit{resultat} = n^k.$$

On part ensuite d'un schéma « vide » de boucle *for* :

```
resultat := ???;
for k from 1 to ??? do
    ???
od;
```

L'idée, c'est que les parties manquantes sont imposées par l'invariant.

L'invariant doit être vrai au début de chaque itération. Il doit être vrai en particulier initialement, pour $k = 1$. La valeur initiale de *resultat* est donc $n^1 = n$. Quelle doit être la valeur finale pour k dans la boucle *for* ? Appelons–la k'. La boucle va s'arrêter à la fin de l'étape $k = k'$ et donc au début de l'étape $k = k' + 1$. À la fin de la boucle, pour que la variable *resultat* vaille n^s, sachant qu'elle vaut $n^{k'+1}$ d'après l'invariant, il faut que k' vaille $s - 1$. Quelle doit être l'instruction du corps de la boucle ? Au début de l'étape k, on a *resultat* $= n^k$. Au début de l'étape suivante, k est incrémenté de 1. Il faut donc multiplier *resultat* par n.

On obtient la solution suivante, qu'il resterait ensuite à « habiller » pour obtenir une fonction MAPLE :

```
resultat := n;
for k from 1 to s-1 do
    resultat := resultat * n
od;
```

Cette solution est un peu moins bonne que la précédente (elle ne couvre pas le cas $s = 0$). Mais mieux vaut une solution imparfaite que pas de solution du tout ! À titre d'information, un invariant correspondant à la première solution est

$$\textit{resultat} = n^{k-1}.$$

2.3.4 L'utilisation de fonctions simplifie l'écriture des programmes

Les deux exemples ci–dessous montrent comment écrire des programmes compliqués par composition de programmes plus simples.

Généralisation de la fonction puissance

On souhaite écrire une fonction *puissance2* qui généralise la fonction *puissance* écrite ci–dessus, de façon à autoriser les exposants négatifs. En voici une spécification :

$$\begin{array}{rccc} \text{puissance2}: & \mathbb{R}\times\mathbb{Z} & \to & \mathbb{R} \\ & (n,\, s) & \mapsto & n^s. \end{array}$$

Remarque : la fonction n'est pas définie dans le cas où $n = 0$ et $s < 0$. En voici une implantation (on utilise la structure de contrôle « if » définie en section 2.4) :

```
puissance2 := proc (n, s)
    local resultat;
    if s >= 0 then
        resultat := puissance (n, s)
    else
        resultat := puissance (1/n, -s)
    fi;
    resultat
end:
```

Autre exemple

On s'intéresse à la célèbre fonction « zéta » de Riemann

$$\zeta(s) = \sum_{n=1}^{\infty} \frac{1}{n^s}\cdot$$

On souhaite approximer $\zeta(s)$ i.e. implanter la fonction spécifiée ci–dessous :

$$\begin{array}{rccl} \text{zeta}: & \mathbb{N}\times\mathbb{N} & \to & \mathbb{R} \\ & (s,\, m) & \mapsto & 1 + \dfrac{1}{2^s} + \cdots + \dfrac{1}{m^s}\cdot \end{array}$$

Idée. On utilise une variable *resultat*. On effectue m itérations (en gros) en faisant varier un indice k. On choisit pour invariant

$$\textit{resultat} = \sum_{j=1}^{k-1} \frac{1}{j^s} = \frac{1}{1^s} + \frac{1}{2^s} + \cdots + \frac{1}{(k-1)^s}\cdot$$

Par des raisonnements comparables à ceux tenus pour la fonction *puissance*, on trouve la solution suivante :

```
zeta := proc (s, m)
    local k, resultat;
    resultat := 0;
    for k from 1 to m do
        resultat := resultat + 1 / puissance (k, s)
    od;
    resultat
end;
```

On remarque que la fonction *zeta* et la fonction *puissance* ont toutes deux des variables locales k et *resultat*. De même qu'il n'y a aucun lien entre une variable globale et une variable locale de même nom, il n'y a aucun lien entre deux variables locales de même nom appartenant à deux fonctions différentes.

2.3.5 Fonctions définies avec une flèche

Les fonctions dont le corps se résume à une instruction peuvent s'écrire simplement grâce à l'opérateur « flèche ». Les deux implantations de f ci–dessous sont équivalentes.

```
f := proc (x)
    3 * x^2 + 2*x + 1
end;

f := x -> 3 * x^2 + 2*x + 1;
```

Voici un autre exemple avec une fonction de deux variables.

```
f := proc (x,y)
    3 * x^2 + x * y
end;

f := (x,y) -> 3 * x^2 + x * y;
```

2.3.6 Fonctions et expressions

Ne pas confondre *fonctions* et *expressions*. Dans l'exemple suivant, f est une fonction et p est une expression. La fonction *D* permet de dériver une fonction, *diff* permet de dériver une expression, *subs* permet d'évaluer une expression.

```
f := x -> 2*x^2 + 3*x + 1;
           f := x -> 2 x^2 + 3 x + 1

p := 2*x^2 + 3*x + 1;
           p := 2 x^2  + 3 x + 1

# correct
f(1);
           6

# incorrect : cette notation ne convient pas aux expressions
p(1);
           2 x(1)^2  + 3 x(1) + 1

# correct
subs (x=1, p);
           6

# incorrect : diff ne s'applique pas aux fonctions
diff (f, x);
           0

# correct
D (f);
           x -> 4 x + 3
```

```
# correct
diff (p, x);
           4 x + 3
```

La conversion d'une fonction en expression peut se faire par une évaluation. La conversion inverse se fait grâce à *unapply*.

```
# conversion d'une fonction en une expression
f (y);
           2 y^2  + 3 y + 1

# conversion d'une expression en une fonction
unapply (p, x);
           x -> 2 x^2 + 3 x + 1
```

2.4 La structure de contrôle if

Prenons l'exemple d'une fonction définie par morceaux

$$\begin{array}{rccll} f: & \mathbb{R} & \to & \mathbb{R} & \\ & x & \mapsto & 0 & \text{si } x \in]-\infty, 0], \\ & & & x & \text{si } x \in]0, 2], \\ & & & x^2 - 2 & \text{si } x \in]2, +\infty[. \end{array}$$

Pour traduire la spécification ci–dessus en une fonction MAPLE, il suffit de faire l'effort de traduire la spécification en une phrase en Français à base de « si », « alors » et « sinon ». Il vaut mieux éviter les tests inutiles. Voici ce qu'on peut obtenir sur l'exemple : si $x \leq 0$ alors le résultat est 0 sinon, si $x \leq 2$ alors le résultat est x sinon le résultat est $x^2 - 2$. Voici deux réalisations :

```
f := proc (x)
    local resultat;
    if x <= 0 then
        resultat := 0
    elif x <= 2 then
        resultat := x
    else
        resultat := x^2 - 2
    fi;
    resultat
end;
```

```
f := proc (x)
    local resultat;
    if x <= 0 then
        resultat := 0
    else
        if x <= 2 then
            resultat := x
```

```
        else
            resultat := x^2 - 2
        fi
    fi;
    resultat
end;
```

2.4.1 Un problème type : déterminer le maximum d'une liste

On cherche à réaliser une fonction *maximum* paramétrée par une liste non vide et qui retourne son plus grand élément.

$$\begin{array}{rrcl} \text{maximum}: & \text{Listes de réels} & \to & \mathbb{R} \\ & L = [a_1, \ldots, a_n] & \mapsto & \max a_j \ (1 \leq j \leq n) \\ & (n \geq 1) & & \end{array}$$

Idée. Utiliser une variable m (pour « maximum »). Faire varier un indice k de 2 à n. Prendre pour invariant : $m = \max a_j$ $(1 \leq j \leq k-1)$. À l'étape k, on compare m et a_k. Si $m < a_k$ alors on remplace l'ancienne valeur de m par a_k sinon on ne fait rien. Lors du codage, l'expression « sinon on ne fait rien » ne se traduit pas (ou encore se traduit par l'absence de partie « sinon »).

```
maximum := proc (L)
    local m, k;
    m := L [1];
    for k from 2 to nops (L) do
        if m < L [k] then
            m := L [k]
        fi
    od;
    m
end;
```

2.5 La structure de contrôle while

On emprunte 5000 euros à 7 pour cent par an. On rembourse 200 euros par mois. Combien de mois faut–il pour rembourser l'emprunt ? Notons d_n la somme due après n mois. Notons la mensualité $m = 200$ et le taux annuel $t = 7/100$. On a

$$d_0 = 5000, \quad d_{n+1} = (d_n - m)(1 + t/12).$$

Le problème consiste à déterminer le plus petit entier n tel que $d_n < m$. Ce problème ne se résout pas naturellement avec un *for* parce qu'on ne connaît pas à l'avance le nombre d'itérations à effectuer. La boucle *while* est une structure de contrôle à laquelle on indique à quelle condition le calcul doit continuer. Sur l'exemple, on continue à rembourser des mensualités complètes tant que la somme due est supérieure à la mensualité.

```
t := 7/100;
m := 200;
n := 0;
d[n] := 5000.;
remb := 0;
while d[n] > m do
    remb := remb + m;
    d[n+1] := (d[n] - m)*(1 + t/12);
    n := n + 1
od;
remb := remb + d[n];
```

À l'exécution on obtient :

```
    remb := 200
d[1] := 4828.000000
       n := 1
    remb := 400
d[2] := 4654.996667
       n := 2
    remb := 600

        ...

d[25] := 385.0959315
      n := 25
    remb := 5200
d[26] := 186.1756578
      n := 26
remb := 5386.175658
```

La boucle s'est arrêtée avec $n = 26$: il va falloir rembourser 26 mensualités complètes plus une dernière, plus petite. Dans le cas général, rien ne garantit qu'une boucle *while* s'arrête. Sur l'exemple, si on fournit une mensualité m trop petite, la boucle ne s'arrête pas : c'est le problème du surendettement.

```
# Même calcul avec m = 20
                                    ...
                                                  7
                          d[1306] := .3091746339 10
                                   n := 1306
                                remb := 26140
                                    ...
```

On dispose pour les boucles *while* d'une notion d'invariant équivalente à celle introduite pour les boucles *for*. On en donnera un exemple un peu plus loin.

Définition 5 *(invariant de boucle while)*

Un invariant de boucle while est une propriété vraie à chaque fois que la condition du while est évaluée (y compris lorsque la condition est fausse, c'est–à–dire lorsque la boucle s'arrête).

Tout calcul réalisable avec *for* est réalisable avec *while*. La réciproque est fausse. Prenons l'exemple de la suite définie par

$$u_0 = 1, \quad u_{n+1} = 2\,u_n + 3.$$

Le calcul de u_{10} peut se réaliser des deux façons ci–dessous :

```
u[0] := 1;
for n from 1 to 10 do
    u[n] := 2*u[n-1] + 3
od;
```

```
u[0] := 1;
n := 1;
while n <= 10 do
    u[n] := 2*u[n-1] + 3;
    n := n + 1
od;
```

2.5.1 Un problème type : tester l'appartenance à une liste

Étant donnés un nombre x et une liste de nombres L, on souhaite déterminer si x appartient à L. Plus précisément, on cherche à écrire une fonction

$$\begin{array}{rrcll} \text{appartient :} & \mathbb{R} \times \text{Listes de réels} & \to & \mathbb{B} & \\ & (x,\, L) & \mapsto & \textit{true} & \text{si } x \in L \\ & & & \textit{false} & \text{sinon} \end{array}$$

Une telle fonction, qui retourne *vrai* ou *faux* est appelée *fonction booléenne* (ou parfois *prédicat*). On commence par réaliser une première version (maladroite) de la fonction avec une boucle *for*. On corrige ensuite la maladresse au moyen d'un *while*.

Idée. On se donne un indice k pour parcourir la liste et une variable booléenne *trouve*. On adopte l'invariant de boucle *for* suivant (l'invariant dépend de k) : *la variable* trouve *vaut* true *si et seulement si x figure parmi les éléments de L d'indice strictement inférieur à k.* On obtient :

```
appartient := proc (x, L)
    local k, trouve;
    trouve := false;
    for k from 1 to nops (L) do
        if x = L[k] then
            trouve := true
        fi
    od;
    trouve
end
```

FIG. 2.1 – George Boole (1815–1864).

La solution est maladroite parce qu'elle parcourt toute la liste même si x figure en début de liste L. On obtient une meilleure solution avec une boucle *while*. Il suffit d'appliquer le mécanisme de traduction de *for* en *while* puis de sophistiquer le test. L'invariant de la boucle *while* est le même que précédemment.

```
appartient := proc (x, L)
    local k, trouve;
    trouve := false;
    k := 1;
    while k <= nops (L) and not trouve do
        if x = L[k] then
            trouve := true
        fi;
        k := k + 1
    od;
    trouve
end
```

2.6 Éléments d'algèbre de Boole

En termes mathématiques, les conditions apparaissant dans les « if » et les « while » sont des expressions *booléennes*, du nom du mathématicien George Boole.

Définition 6 *Une* expression booléenne *est une expression qui s'évalue en* vrai *ou* faux

Les valeurs *vrai* et *faux* se notent *true* et *false* en MAPLE. Par exemple,

$$x \geq 3, \quad x \geq 3 \text{ et } 3x \neq 2, \quad x \text{ est un nombre pair}$$

FIG. 2.2 – Augustus De Morgan (1806–1871).

sont des expressions booléennes. Par contre

$$x + 7, \quad \text{la moitié d'un nombre pair}$$

ne sont pas des expressions booléennes. Les opérateurs $\leq$, $<$, $=$, $\neq$, $\geq$, $>$ sont appelés *opérateurs relationnels*. Ils servent à construire des expressions booléennes élémentaires. Ils sont notés en MAPLE $<=$, $<$, $=$, $<>$, $>=$ et $>$. Les opérateurs *et*, *ou*, *non* sont appelés *opérateurs logiques*. Ils servent à construire des expressions booléennes compliquées à partir d'expressions booléennes élémentaires. Ils sont notés *and*, *or* et *not* en MAPLE. Voici leur table de vérité :

and	vrai	faux
vrai	vrai	faux
faux	faux	faux

or	vrai	faux
vrai	vrai	vrai
faux	vrai	faux

not	
vrai	faux
faux	vrai

Les trois opérateurs logiques décrits ci–dessus sont liés par les *lois de De Morgan*, du nom du mathématicien Augustus De Morgan.

Proposition 1 *(lois de De Morgan)*
Quelles que soient les expressions booléennes a et b on a

$$\begin{aligned} \textit{non } (a \textit{ et } b) &= (\textit{non } a) \textit{ ou } (\textit{non } b) \\ \textit{non } (a \textit{ ou } b) &= (\textit{non } a) \textit{ et } (\textit{non } b) \end{aligned}$$

Exemple. Reprenons l'exemple de la fonction *appartient*. On cherche, en effectuant une boucle, si un élément appartient à une liste. La boucle s'arrête si

k dépasse le nombre d'éléments de L ou le booléen *trouve* vaut *vrai*.

La condition du *while* est la négation de l'expression booléenne ci–dessus. Elle s'obtient grâce aux lois de De Morgan : les calculs continuent tant que

$$\underline{k \text{ ne dépasse pas le nombre d'éléments de } L} \text{ et } \underline{\text{le booléen } \textit{trouve} \text{ vaut } \textit{faux}}.$$

Remarque. L'opérateur logique *and* de MAPLE n'évalue pas son second opérande si le premier s'évalue en *faux*. Cette particularité permet d'écrire la fonction *appartient* de la façon suivante.

```
appartient := proc (x, L)
    local k;
    k := 1;
    while k <= nops (L) and x <> L[k] do
        k := k + 1
    od;
    evalb (k <= nops (L))
end
```

Cette écriture aurait été incorrecte dans la cas d'un *and* évaluant systématiquement tous ses opérandes puisque la fonction aurait cherché à déterminer $L[k]$ pour $k > \textit{nops}\,(L)$.

Chapitre 3

Systèmes linéaires

Ce chapitre comporte trois cours. Dans les deux premiers on s'intéresse à la résolution de systèmes d'équations linéaires. Dans le premier, on considère le cas de systèmes ayant plus (respectivement autant) d'inconnues que d'équations, qui ont en général une infinité de solutions (respectivement une unique solution). On introduit pour cela l'algorithme du pivot de Gauss et sa variante dite de « Gauss–Jordan ». Dans le deuxième, on considère le cas où le système a plus d'équations que d'inconnues. En général, de tels systèmes n'ont aucune solution mais on peut chercher une « presque solution » aussi proche que possible d'une « vraie solution ». On introduit la méthode des moindres carrés. Le troisième cours introduit une notion très importante : la diagonalisation de matrices. Elle nous servira dans le dernier chapitre. On en profite pour comparer diagonalisation et pivot de Gauss.

Ces trois cours doivent être considérés comme des rappels d'algèbre illustrés informatiquement. Dans la plupart des situations réelles apparaissent des problèmes que nous passons sous silence (problèmes de conditionnement de matrices lorsque les calculs se font en virgule flottante, détermination des racines de polynômes de degré élevé) qui amènent à employer des algorithmes plus sophistiqués que ceux que nous présentons.

Aux étudiants qui souhaitent en savoir davantage, on suggère la lecture[1] du chapitre IV du polycopié [5].

3.1 Manipulations élémentaires sur les matrices

On introduit ci–dessous les fonctions du paquetage *LinearAlgebra* de MAPLE qui nous seront les plus utiles. Un paquetage MAPLE est un ensemble de fonctions. Les fonctions de *LinearAlgebra* ne sont pas automatiquement chargées en mémoire lorsqu'on ouvre une session de MAPLE. La commande *with* permet de les charger.

```
with (LinearAlgebra):

Matrix (2,3);
```

[1] le simple survol est déjà très instructif !

```
                                  [0    0    0]
                                  [           ]
                                  [0    0    0]

Vector[column] (3);
                                       [0]
                                       [ ]
                                       [0]
                                       [ ]
                                       [0]

Vector[row] (4);
                                  [0, 0, 0, 0]
```

Le paquetage *LinearAlgebra* offre une notation alternative permettant de définir des matrices et des vecteurs. L’opérateur « virgule » permet d’ajouter des lignes à une matrice. L’opérateur « barre verticale » permet d’ajouter des colonnes (penser à la notation traditionnelle utilisée pour border une matrice avec un vecteur).

```
# Les éléments sont ajoutés les uns après les autres en formant à
# chaque fois une nouvelle ligne : on obtient un vecteur colonne.
<1,2,3>;

                                       [1]
                                       [ ]
                                       [2]
                                       [ ]
                                       [3]

# Les éléments sont ajoutés les uns après les autres en formant à
# chaque fois une nouvelle colonne : on obtient un vecteur ligne.
<1|2|3>;
                                    [1, 2, 3]

# L’opérateur , permet de border une matrice avec un vecteur ligne
M := <<1|2|3>, <4|5|6>>;
                                       [1    2    3]
                                 M := [           ]
                                       [4    5    6]

W := <7|8|9>;
                                   W := [7, 8, 9]

<M, W>;
                                  [1    2    3]
                                  [           ]
                                  [4    5    6]
                                  [           ]
                                  [7    8    9]
```

```
# L'opérateur | permet de border une matrice avec un vecteur colonne
W := <10, 11>;
                                    [10]
                                W := [  ]
                                    [11]

<M | W>;
                              [1    2    3    10]
                              [                  ]
                              [4    5    6    11]
```

L'opérateur « + » permet d'additionner deux matrices. L'opérateur « ∗ » permet de multiplier une matrice par un scalaire. L'opérateur « . » permet de multiplier deux matrices entre elles. L'opérateur « accent circonflexe » permet d'élever une matrice à une certaine puissance. Lorsque l'exposant vaut -1 on obtient l'inverse de la matrice.

```
M + x * M;
                         [ 1 + x     2 + 2 x    3 + 3 x]
                         [                             ]
                         [4 + 4 x    5 + 5 x    6 + 6 x]

N := M . Transpose (M);
                                   [14    32]
                               N := [        ]
                                   [32    77]

Nmu := N^(-1);
                                   [77     -16 ]
                                   [--     --- ]
                                   [54     27  ]
                             Nmu := [           ]
                                   [-16        ]
                                   [---    7/27]
                                   [27         ]
```

Il est souvent utile en programmation de pouvoir déterminer les dimensions d'une matrice ou d'accéder à des éléments à partir de leurs indices de ligne et de colonne.

```
Dimension (M);
                                  2, 3

RowDimension (M);
                                    2

ColumnDimension (M);
                                    3
# Élément ligne 1 et colonne 3.
M [1,3];
                                    3
```

3.2 Résolution de systèmes d'équations linéaires

On s'intéresse au problème suivant : étant donné un système linéaire, décrire l'ensemble de ses solutions. Voici un exemple de système linéaire :

$$\begin{cases} 2\,x_1 - x_2 + 4\,x_3 - 2\,x_4 &= 0, \\ -2\,x_1 + 2\,x_2 - 3\,x_3 + 4\,x_4 &= 0, \\ 4\,x_1 - x_2 + 8\,x_3 + x_4 &= 1 \end{cases}$$

Résoudre le système revient à résoudre le système

$$A\,x = b$$

où la matrice $A \in \mathbb{R}^{m\times n}$ des coefficients du système et le vecteur $b \in \mathbb{R}^m$ des membres gauches des équations sont donnés et où $x = {}^t(x_1 \ \dots \ x_n)$ désigne le vecteur des inconnues.

$$\begin{pmatrix} 2 & -1 & 4 & -2 \\ -2 & 2 & -3 & 4 \\ 4 & -1 & 8 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}.$$

Trois cas peuvent se produire : le système n'a aucune solution, le système a exactement une solution, le système a une infinité de solutions. La résolution peut se mener automatiquement par l'algorithme du « pivot de Gauss » ou sa variante, le « pivot de Gauss–Jordan ».

Le pivot de Gauss

Définition 7 *Deux systèmes qui ont même ensemble de solutions sont dits* équivalents.

On transforme un système en un système équivalent, plus simple, sur lequel on lit toutes les informations désirées. L'algorithme du pivot de Gauss repose sur la proposition suivante.

Proposition 2 *Les opérations suivantes ne changent pas les solutions d'un système (il faut les faire à la fois sur les membres gauche et droit des équations).*

1. *Multiplier une ligne par un scalaire non nul.*
2. *Ajouter à une ligne un multiple d'une autre ligne.*
3. *Échanger deux lignes.*

Ces opérations ne changent pas les solutions d'un système parce qu'elles sont « réversibles ». Le « pivot de Gauss » applique les opérations précédentes sur le système dans le but d'obtenir un système équivalent où chaque équation (on les lit du bas vers le haut) introduit au moins une nouvelle inconnue. Reprenons l'exemple qui précède. Après pivot de Gauss :

$$\begin{cases} 2\,x_1 - x_2 + 4\,x_3 - 2\,x_4 &= 0, \\ x_2 + x_3 + 2\,x_4 &= 0, \\ -x_3 + 3\,x_4 &= 1 \end{cases}$$

FIG. 3.1 – Carl Friedrich Gauss (1777–1855) en 1803. Il invente la méthode des moindres carrés pour déterminer l'orbite de Cérès en 1801. Il l'applique à nouveau vers 1810 pour l'astéroïde Pallas et publie à cette occasion la méthode qui lui a permis de résoudre le système des « équations normales » ; méthode qu'on appelle aujourd'hui le « pivot de Gauss » [2].

L'équation du bas introduit x_3 et x_4. L'équation du milieu introduit x_2. L'équation du haut introduit x_1. Plutôt que de manipuler le système d'équations, on manipule la matrice $(A \mid b)$ formée de la matrice A des coefficients du système, bordée par le vecteur des seconds membres.

$$(A \mid b) = \left(\begin{array}{cccc|c} 2 & -1 & 4 & -2 & 0 \\ -2 & 2 & -3 & 4 & 0 \\ 4 & -1 & 8 & 1 & 1 \end{array}\right)$$

La matrice $(A' \mid b')$ obtenue après pivot de Gauss :

$$(A' \mid b') = \left(\begin{array}{cccc|c} 2 & -1 & 4 & -2 & 0 \\ 0 & 1 & 1 & 2 & 0 \\ 0 & 0 & -1 & 3 & 1 \end{array}\right)$$

Le fait que chaque équation introduit au moins une inconnue se traduit par le fait que la diagonale principale est formée d'éléments non nuls (ce sont les « pivots ») sous lesquels toutes les entrées de la matrice sont nulles. Remarque : sur une matrice quelconque, on pourrait très bien avoir une diagonale avec des « décrochements ». La variante de Gauss–Jordan est un pivot de Gauss « poussé au maximum » où on impose que les pivots soient égaux à 1 et que tous les éléments d'une colonne contenant un pivot (sauf le pivot bien sûr) soient nuls. Voici le système et la matrice $(A'' \mid b'')$ obtenus après pivot de Gauss–Jordan :

$$\left\{\begin{array}{l} x_1 + 15/12\, x_4 = 5/2 \\ x_2 + 5\, x_4 = 1 \\ x_3 - 3\, x_4 = -1 \end{array}\right. \qquad (A'' \mid b'') = \left(\begin{array}{cccc|c} 1 & 0 & 0 & 15/12 & 5/2 \\ 0 & 1 & 0 & 5 & 1 \\ 0 & 0 & 1 & -3 & -1 \end{array}\right)$$

FIG. 3.2 – Marie Ennemond Camille Jordan (1838–1922). Son nom est associé à deux algorithmes d'algèbre linéaire qu'il ne faut pas confondre : le pivot de « Gauss–Jordan » et la « mise sous forme de Jordan ».

On peut maintenant décrire l'ensemble des solutions du système en se servant de x_4 comme paramètre.

$$\left\{ \begin{pmatrix} 5/2 - 15/12\, x_4 \\ 1 - 5\, x_4 \\ -1 + 3\, x_4 \\ x_4 \end{pmatrix}, \quad x_4 \in \mathbb{R} \right\}$$

Proposition 3 *Soient $(A \mid b)$ un système et $(A' \mid b')$ un système équivalent sous forme de Gauss ou de Gauss–Jordan.*

1. *Le système est sans solutions si et seulement s'il y a un pivot dans la colonne b'.*
2. *Le système a une unique solution si et seulement si toutes les colonnes sauf b' contiennent un pivot.*
3. *Le système a une infinité de solutions si et seulement si b' et au moins une colonne de A' ne contiennent pas de pivot. Dans ce cas, on peut décrire l'ensemble des solutions du système en se servant des inconnues correspondant aux colonnes sans pivots comme paramètres.*

Utilisation en MAPLE

On traite l'exemple précédent avec le paquetage *LinearAlgebra* de MAPLE.

```
# Chargement du paquetage en mémoire
with (LinearAlgebra):
```

```
# Le système d'équations
S := [  2*x1 -   x2 + 4*x3 - 2*x4 = 0,
       -2*x1 + 2*x2 - 3*x3 + 4*x4 = 0,
        4*x1 -   x2 + 8*x3 +   x4 = 1 ]:

# La liste des inconnues
X := [x1, x2, x3, x4]:

# GenerateMatrix retourne une séquence formée de la
# matrice des coefficients de S suivie du vecteur des seconds membres
# L'ordre des inconnues dans la liste X donne l'ordre des colonnes
# dans la matrice des coefficients

# Observer la technique utilisée pour affecter la séquence résultat de
# l'appel de fonction aux deux variable A et b.

A, b := GenerateMatrix (S, X);
                                [ 2    -1     4    -2]  [0]
                                [                    ]  [ ]
                          A, b := [-2     2    -3     4], [0]
                                [                    ]  [ ]
                                [ 4    -1     8     1]  [1]

# Ab s'obtient en bordant A avec b.

Ab := <A | b>;
                                  [ 2    -1     4    -2    0]
                                  [                        ]
                            Ab := [-2     2    -3     4    0]
                                  [                        ]
                                  [ 4    -1     8     1    1]

# LinearSolve retourne le même résultat que celui obtenu dans la
# section précédente (l'inconnue _t0[4] correspond à x4).

LinearSolve (Ab);
                                  [5/2 - 15/2 _t0[4]]
                                  [                 ]
                                  [  1 - 5 _t0[4]   ]
                                  [                 ]
                                  [  -1 + 3 _t0[4]  ]
                                  [                 ]
                                  [     _t0[4]      ]

# La fonction GaussianElimination applique le pivot de Gauss.
GaussianElimination (Ab);
                             [2    -1     4    -2    0]
                             [                        ]
                             [0     1     1     2    0]
                             [                        ]
                             [0     0    -1     3    1]
```

```
# La variante de Gauss-Jordan s'appelle ReducedRowEchelonForm.
# On retrouve le résultat obtenu dans la section précédente.
# La fonction ReducedRowEchelonForm est appelée par LinearSolve.

Ab2 := ReducedRowEchelonForm (Ab);
                              [1    0    0    15/2    5/2]
                              [                          ]
                        Ab2 := [0    1    0     5       1 ]
                              [                          ]
                              [0    0    1     -3     -1 ]

# GenerateEquations effectue l'opération inverse de GenerateMatrix.
# Elle permet ici d'interpréter facilement le résultat obtenu.

GenerateEquations (Ab2, X);
                       15 x4
                [x1 + ----- = 5/2, x2 + 5 x4 = 1, x3 - 3 x4 = -1]
                        2
```

Algorithme du pivot de Gauss

On procède colonne par colonne. On commence à la colonne 1. On cherche un élément non nul (un pivot). On choisit l'élément en haut à gauche de la matrice. On ajoute un multiple de la première ligne à chacune des lignes qui suivent de façon à faire apparaître des zéros sous le pivot. On obtient

$$\begin{pmatrix} 2 & -1 & 4 & -2 & 0 \\ 0 & 1 & 1 & 2 & 0 \\ 0 & 1 & 0 & 5 & 1 \end{pmatrix}$$

On passe à la colonne 2. On cherche un élément non nul sur cette colonne qui ne soit pas sur la même ligne que le pivot précédent (on veut en fait que sur la ligne du nouveau pivot il y ait un zéro au niveau de la première colonne). On choisit le 1 sur la deuxième ligne. On ajoute un multiple de la deuxième ligne à la troisième de façon à faire apparaître des zéros sous le pivot.

$$\begin{pmatrix} 2 & -1 & 4 & -2 & 0 \\ 0 & 1 & 1 & 2 & 0 \\ 0 & 0 & -1 & 3 & 1 \end{pmatrix}$$

C'est fini pour le pivot de Gauss.

Pour obtenir un système sous forme de Gauss–Jordan, on peut continuer comme suit. On ajoute un multiple de la troisième ligne aux deux qui précèdent pour faire apparaître des zéros sur la troisième colonne

$$\begin{pmatrix} 2 & -1 & 0 & 10 & 4 \\ 0 & 1 & 0 & 5 & 1 \\ 0 & 0 & -1 & 3 & 1 \end{pmatrix}$$

On ajoute un multiple de la deuxième ligne à la première pour faire apparaître un zéro sur la deuxième colonne

$$\begin{pmatrix} 2 & 0 & 0 & 15 & 5 \\ 0 & 1 & 0 & 5 & 1 \\ 0 & 0 & -1 & 3 & 1 \end{pmatrix}$$

On multiplie enfin chaque ligne par un scalaire approprié pour que les pivots vaillent 1.

$$\begin{pmatrix} 1 & 0 & 0 & 15/2 & 5/2 \\ 0 & 1 & 0 & 5 & 1 \\ 0 & 0 & 1 & -3 & -1 \end{pmatrix}$$

Dans le cas général, il peut arriver qu'on ne puisse pas trouver de pivot sur la colonne en cours de traitement. Dans ce cas, on passe à la colonne suivante.

Remarque : la matrice obtenue à la fin du pivot de Gauss n'est pas définie de façon unique puisqu'à chaque étape, le choix du pivot influe sur les calculs. Certaines propriétés du système (rang, ensemble des solutions) sont toutefois indépendantes du choix du pivot.

En calcul scientifique, les entrées de la matrice sont très souvent des nombres inexacts. Dans ce cas, des problèmes supplémentaires apparaissent et il est essentiel de choisir le pivot avec soin. Cette question est abordée page 90 du chapitre IV du polycopié [5]. Une notion liée très importante est celle de la « condition » ou du « conditionnement » d'une matrice, développée page 93 de ce même polycopié.

3.3 La méthode des moindres carrés

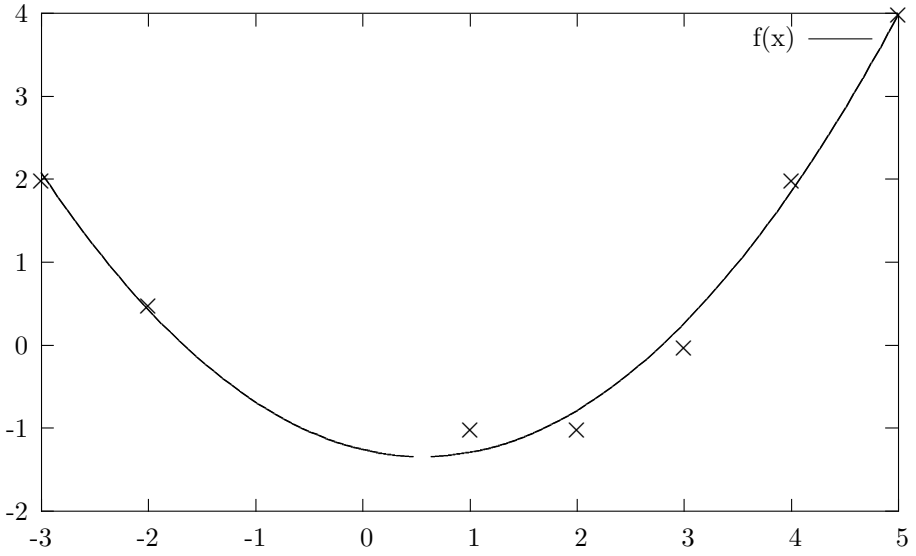
Présentons le problème sur un exemple.

i	1	2	3	4	5	6	7
x_i	-3	-2	1	2	3	4	5
y_i	2	0.5	-1	-1	0	2	4

On suppose que les sept points ci–dessus du plan $(x,\, y)$ ont été mesurés par un certain procédé et qu'en théorie, ils devraient tous appartenir au graphe d'une même parabole, dont on cherche à identifier les coefficients $a_2,\, a_1,\, a_0$:

$$y = f(x) = a_2\, x^2 + a_1\, x + a_0.$$

En d'autres mots, on cherche les coefficients d'une parabole qui passe au plus près des sept points :



La méthode des moindres carrés résout ce problème.

La méthode

On considère un système $\mathscr{S}$ de m équations linéaires à n inconnues $a_1, \ldots, a_n$ avec $m > n$. Il y a plus d'équations que d'inconnues. Les inconnues sont les a_i (comme dans l'exemple). Les quantités x_{ij} et y_i sont supposées connues.

$$\mathscr{S} \begin{cases} a_1\, x_{11} + \cdots + a_n\, x_{1n} &= y_1 \\ a_1\, x_{21} + \cdots + a_n\, x_{2n} &= y_2 \\ & \vdots \\ a_1\, x_{m1} + \cdots + a_n\, x_{mn} &= y_m \end{cases}$$

En général $\mathscr{S}$ n'a pas de solution mais on peut chercher une solution approchée $(\bar{a}_1, \ldots, \bar{a}_n)$ telle que la somme de carrés

$$e_1^2 + \cdots + e_m^2$$

soit minimale où

$$e_i \underset{\text{def}}{=} \bar{a}_1\, x_{i1} + \cdots + \bar{a}_n\, x_{in} - y_i, \quad 1 \leq i \leq m$$

désigne l'écart entre la valeur y_i mesurée et la valeur $\bar{a}_1\, x_{i1} + \cdots + \bar{a}_n\, x_{in}$ fournie par la solution approchée. On peut montrer que cette solution minimale existe et est unique. Matriciellement, $\mathscr{S}$ s'écrit $X\, a = y$ c'est–à–dire

$$\begin{pmatrix} x_{11} & \cdots & x_{1n} \\ x_{21} & & x_{2n} \\ \vdots & & \vdots \\ x_{m1} & \cdots & x_{mn} \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix}.$$

On multiplie à gauche les deux membres de l'égalité par la transposée ${}^tX$ de la matrice X c'est–à–dire

$$\begin{pmatrix} x_{11} & x_{21} & \cdots & x_{m1} \\ \vdots & & & \vdots \\ x_{1n} & x_{2n} & \cdots & x_{mn} \end{pmatrix}$$

On obtient un nouveau système ${}^tX\,X\,a = {}^tX\,y$ de n équations à n inconnues, appelé « système des équations normales ». La matrice ${}^tX\,X$ des coefficients est symétrique. On peut montrer que la solution de ce nouveau système (qui peut s'obtenir par pivot de Gauss par exemple) est la solution approchée $(\bar{a}_1, \ldots, \bar{a}_n)$ désirée.

Application à l'exemple

On cherche à identifier les coefficients $a_2,\ a_1,\ a_0$ de la parabole $y = f(x) = a_2\,x^2 + a_1\,x + a_0$. On cherche donc un triplet $(\bar{a}_2,\ \bar{a}_1,\ \bar{a}_0)$ qui minimise la somme des carrés

$$e_1^2 + \cdots + e_7^2$$

où les quantités

$$e_i \underset{\text{def}}{=} \bar{a}_2\,x_i^2 + \bar{a}_1\,x_i + \bar{a}_0 - y_i, \quad 1 \leq i \leq 7$$

désignent les écarts entre les ordonnées y_i mesurées et les ordonnées $f(x_i)$ des points de la courbe d'abscisse x_i. Résolution avec l'aide du paquetage *LinearAlgebra* de MAPLE.

```
# Chargement du paquetage LinearAlgebra
with (LinearAlgebra):

# La liste des points mesurés
P := [[-3,2],[-2,1/2],[1,-1],[2,-1],[3,0],[4,2],[5,4]]:

# La matrice X
X := Matrix ([seq ([P[i,1]^2, P[i,1], 1], i = 1 .. nops (P)) ]);
                                  [ 9    -3    1]
                                  [             ]
                                  [ 4    -2    1]
                                  [             ]
                                  [ 1     1    1]
                                  [             ]
                             X := [ 4     2    1]
                                  [             ]
                                  [ 9     3    1]
                                  [             ]
                                  [16     4    1]
                                  [             ]
                                  [25     5    1]

# Le vecteur des coefficients
y := Vector ([ seq (P [i,2], i = 1 .. nops (P)) ]);
```

```
                                    [ 2 ]
                                    [   ]
                                    [1/2]
                                    [   ]
                                    [-1 ]
                                    [   ]
                                y := [-1 ]
                                    [   ]
                                    [ 0 ]
                                    [   ]
                                    [ 2 ]
                                    [   ]
                                    [ 4 ]
# Le système normal s'obtient en multipliant X et y à gauche
# par la transposée de X.

tXX := Transpose (X) . X;
                                  [1076    190    68]
                                  [                 ]
                           tXX := [ 190     68    10]
                                  [                 ]
                                  [  68     10     7]
tXy := Transpose (X) . y;
                                        [147 ]
                                        [    ]
                                  tXy := [ 18 ]
                                        [    ]
                                        [13/2]

# Résolution du système normal avec Gauss-Jordan
M := ReducedRowEchelonForm (< tXX | tXy >);

                                  [               4311  ]
                                  [1    0    0    ----- ]
                                  [               15974 ]
                                  [                     ]
                                  [               -4861 ]
                           M := [0    1    0    ----- ]
                                  [               15974 ]
                                  [                     ]
                                  [               -20101]
                                  [0    0    1    ------]
                                  [               15974 ]

# Interprétation de la solution sous la forme d'un système
# d'équations de la forme : coeff de la parabole = valeur

GenerateEquations (M, [a2, a1, a0]);
                              4311       -4861       -20101
                      [a2 = -----, a1 = -----, a0 = ------]
                             15974       15974       15974
```

La méthode des moindres carrés est également décrite à la page 100 du chapitre IV du polycopié [5]. On y trouve les explications précédentes avec de nombreux développements. Cette méthode est parfois appelée « méthode des moindres carrés *linéaire* » pour la distinguer de sa généralisation dans le cas où les inconnues n'apparaissent pas linéairement dans les équations. Une introduction à la « méthode des moindres carrés *non linéaire* » est décrite page 141 du chapitre VI du polycopié [5] (avec une jolie application pour amateurs de randonnées en montagne). Elle s'appuie sur la méthode de Newton que nous étudierons au chapitre suivant.

3.4 Diagonalisation de matrice

Soit M une matrice 2×2 (pour faire simple, mais tous les raisonnements sont valables pour des matrices carrées de dimension quelconque). Pour calculer M^2, M^3 etc ... il suffit d'appliquer l'algorithme bien connu du produit de matrices. Le paquetage *LinearAlgebra* permet de calculer ces puissances de matrices très facilement. Mais comment obtenir une expression pour M^n en laissant n sous forme symbolique ? Et comment obtenir M^n avec n non entier ? La réponse donnée par le paquetage n'est pas très satisfaisante (voir ci–dessous) ! Ce sont des questions plus concrètes qu'il n'y paraît : certaines propriétés d'un câble coaxial de n mètres se lisent dans une certaine matrice M (qui dépend du câble) élevée à la puissance n : il n'y a aucune raison que n soit un nombre entier.

```
# Une matrice M
M := <<1 | 2>, <3 | 2>>;
                                        [1    2]
                                   M := [      ]
                                        [3    2]

# Aucun problème pour calculer M^n quand n est un entier
M^2;
                                      [7     6]
                                      [       ]
                                      [9    10]

# La réponse du paquetage quand n n'est pas entier est peu utilisable !
M^n;
                                      [1    2]n
                                      [      ]
                                      [3    2]
M^(1.5);
                                     [1    2]1.5
                                     [      ]
                                     [3    2]
```

La diagonalisation de matrice répond à cette question.

À première vue, déterminer une expression symbolique de M^n semble difficile sauf dans le cas très particulier où la matrice est diagonale, c'est–à–dire où elle ne comporte des éléments non nuls

que sur la diagonale :

$$\begin{pmatrix} a & 0 \\ 0 & b \end{pmatrix}^n = \begin{pmatrix} a^n & 0 \\ 0 & b^n \end{pmatrix}.$$

Diagonaliser une matrice carrée M quelconque, c'est chercher une matrice diagonale J et une matrice P telles que

$$M = P\,J\,P^{-1}.$$

Diagonaliser M simplifie considérablement le problème de l'élévation à la puissance. Un calcul rapide montre en effet que

$$M^n = P\,J^n\,P^{-1}.$$

Diagonaliser M n'est pas toujours possible. Il existe toutefois une opération plus générale que la diagonalisation, qu'on appelle la mise sous forme de Jordan et qui se réduit à la diagonalisation quand on l'applique à une matrice diagonalisable. Dans le paquetage *LinearAlgebra*, la diagonalisation s'obtient par la fonction *JordanForm*. Cette fonction retourne normalement la matrice J. Elle retourne la matrice P si on lui fournit comme paramètre supplémentaire « *output='Q'* ».

```
M := <<1 | 2>, <3 | 2>>;
                                    [1    2]
                                M := [      ]
                                    [3    2]

J := JordanForm (M);
                                    [-1    0]
                                J := [       ]
                                    [ 0    4]

P := JordanForm (M, output='Q');
                                  [3/5     2/5]
                              P := [            ]
                                  [-3/5    3/5]

# On vérifie la théorie

P . J. Pˆ(-1);
                                  [1    2]
                                  [      ]
                                  [3    2]

# On calcule une expression de Mˆn

Jn := << (-1)ˆn | 0 >, < 0 | 4ˆn >>;
                                    [    n      ]
                                    [(-1)     0 ]
                              Jn := [            ]
                                    [           n]
                                    [  0      4 ]
# La voici
```

```
Mn := P . Jn . P^(-1);
                        [       n      n              n      n]
                        [ 3 (-1)    2 4        2 (-1)    2 4 ]
                        [ ------- + ----     - ------- + ----]
                        [    5       5            5       5  ]
                  Mn := [                                     ]
                        [        n      n           n      n ]
                        [  3 (-1)    3 4      2 (-1)    3 4  ]
                        [- ------- + ----     ------- + ---- ]
                        [     5       5          5       5   ]

# Vérification pour n = 2
subs (n = 2, Mn);
                                    [7     6]
                                    [       ]
                                    [9    10]
```

3.4.1 Compléments

Le contenu de ce paragraphe sera utilisé dans le cadre de l'analyse qualitative des systèmes différentiels. Lorsque une matrice M est diagonalisable, les éléments de la matrice diagonale J sont des « valeurs propres » λ_1, λ_2 de M et les colonnes de la matrice P sont des « vecteurs propres » V_1, V_2 de M.

Définition 8 *Un scalaire λ est une « valeur propre » de M s'il existe un vecteur V non nul tel que $M\,V = \lambda\,V$. Le vecteur V est appelé un « vecteur propre » associé à λ.*

Valeur propre se dit « eigenvalue » en Anglais. Vecteur propre se dit « eigenvector ». La fonction *Eigenvalues* du paquetage *LinearAlgebra* retourne les valeurs propres d'une matrice. La fonction *Eigenvectors* retourne une séquence formée du vecteur des valeurs propres suivi d'une matrice formée d'un vecteur propre par valeur propre. Les vecteurs propres apparaissent en colonne. Ils sont donnés dans le même ordre que les valeurs propres.

```
# Les deux valeurs propres de M
Eigenvalues (M);
                                    [ 4]
                                    [  ]
                                    [-1]
# Les valeurs propres et un vecteur propre par valeur propre
Eigenvectors (M);
                                [ 4]  [2/3    -1]
                                [  ], [         ]
                                [-1]  [ 1      1]

# On vérifie la théorie.
M . <2/3, 1>;
                                    [8/3]
                                    [   ]
                                    [ 4 ]
```

```
4 * <2/3, 1>;
                                    [8/3]
                                    [   ]
                                    [ 4 ]
M . <-1, 1>;
                                    [ 1]
                                    [  ]
                                    [-1]

-1 * <-1, 1>;
                                    [ 1]
                                    [  ]
                                    [-1]
```

Proposition 4 *Les valeurs propres de M sont les racines du polynôme caractéristique de M c'est–à–dire du polynôme $\det(M - \lambda\, I)$ où λ désigne l'inconnue et I la matrice identité de même dimension que M.*

```
# On vérifie la proposition
M - lambda * IdentityMatrix (2);
                           [1 - lambda        2     ]
                           [                        ]
                           [    3         2 - lambda]

# Le polynôme caractéristique de M
Determinant (%);
                                                 2
                              -4 - 3 lambda + lambda

# Ses racines sont les valeurs propres de M
solve (%, lambda);
                                    4, -1
```

La proposition suivante implique immédiatement que l'ensemble des vecteurs propres associés à une valeur propre forme un espace vectoriel (appelé « espace propre »).

Proposition 5 *Si $V_1, \ldots, V_k$ sont des vecteurs propres associés à une même valeur propre λ alors toute combinaison linéaire $c_1\, V_1 + \cdots + c_k\, V_k$ est encore un vecteur propre associé à λ.*

3.4.2 Considérations algorithmiques

La principale difficulté lors de la diagonalisation consiste à calculer les valeurs propres. Le polynôme caractéristique d'une matrice M de dimension $n \times n$ est un polynôme de degré n. Déterminer les racines d'un polynôme de degré n est difficile (même en théorie) dès que $n \geq 5$. Par contre, déterminer une base de l'espace propre d'une valeur propre λ connue est facile (au moins en théorie) : il suffit par exemple d'appliquer l'algorithme du pivot de Gauss à la matrice $M - \lambda\, I$. Voici comment procéder sur l'exemple.

Calcul d'un vecteur propre associé à $\lambda = 4$

On résout le système $M - \lambda I = 0$ pour $\lambda = 4$.

```
# M4 := M - lambda * I pour lambda = 4
M4 := M - 4 * IdentityMatrix (2);
                                    [-3     2]
                                M4 := [        ]
                                    [ 3    -2]

# V := Les solutions de M4 . x = 0 (x vecteur d'inconnues qq).
V := LinearSolve (< M4 | <0, 0> >);
                                  [  _t0[1]  ]
                                V := [          ]
                                  [3/2 _t0[1]]

# Le vecteur propre V dépend d'un paramètre _t0[1]
# V est un vecteur propre de M de valeur propre 4
M . V;
                                  [4 _t0[1]]
                                  [        ]
                                  [6 _t0[1]]
```

Calcul d'un vecteur propre associé à $\lambda = -1$

La même manipulation, effectuée pour la valeur propre $\lambda = -1$ fournit une description de l'ensemble des vecteurs propres qui lui sont associés :

```
# M_1 := M - lambda * I pour lambda = -1
M_1 := M - (-1) * IdentityMatrix (2);
                                    [2    2]
                                M_1 := [      ]
                                    [3    3]

# V := les solutions de M_1 . x = 0 (x vecteur d'inconnues qq)
V := LinearSolve ( < M_1 | <0, 0 >> );
                                    [-_t0[2]]
                                V := [       ]
                                    [_t0[2] ]

# On vérifie que V est un vecteur propre de M de valeur propre -1.
M . V;
                                   [_t0[2] ]
                                   [       ]
                                   [-_t0[2]]
```

Le calcul des valeurs propres et des vecteurs propres d'une matrice est un sujet important en calcul scientifique. La méthode directe (résolution du polynôme caractéristique) est très rarement employée lorsque les entrées des matrices sont des nombres inexacts. Le chapitre VI du polycopié [5] est entièrement consacré à ces questions.

3.4.3 Pivot de Gauss contre diagonalisation

Attention à ne pas confondre les deux opérations qui retournent toutes les deux des matrices diagonales quand on les applique à des matrices carrées (et que tout marche bien). La matrice produite par le pivot de Gauss s'obtient en multipliant la matrice d'entrée à gauche par une matrice inversible. La matrice produite par le procédé de diagonalisation s'obtient en multipliant la matrice d'entrée à gauche par une matrice inversible et à droite par son inverse.

Chapitre 4

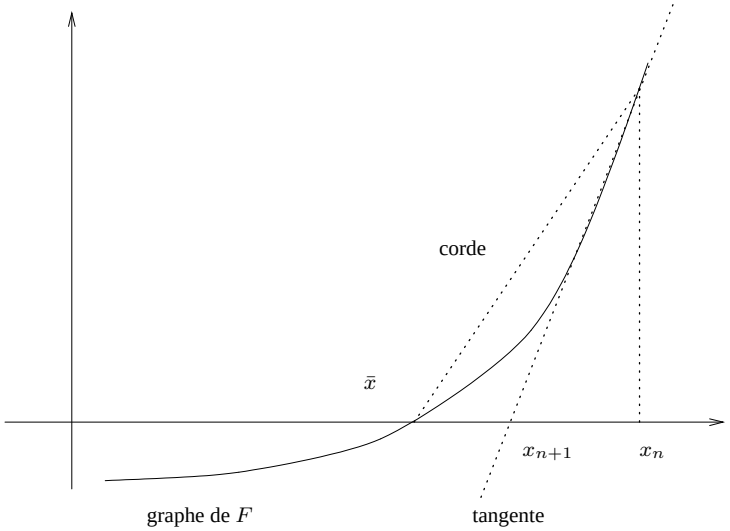
Méthodes numériques de résolution d'équations

4.1 La méthode de Newton

On cherche à déterminer une solution $\bar{x}$ de l'équation $F(x) = 0$ où F est une fonction deux fois dérivable. On part d'une solution approchée $x = a$. La méthode d'Isaac Newton consiste à calculer les premiers termes d'une suite (x_n) définie par

$$x_0 = a, \quad x_{n+1} = x_n - \frac{F(x_n)}{F'(x_n)}\cdot$$

On arrête les calculs dès que $|F(x_n)| < \varepsilon$ où ε est un réel positif fixé à l'avance. Graphiquement, x_{n+1} est donné par l'intersection de l'axe des x avec la tangente à la courbe en $(x_n,\, F(x_n))$.



Explication. D'après le théorème des accroissements finis, il existe un point $z \in [\bar{x},\, x_n]$ tel que la tangente à la courbe en z soit parallèle à la corde qui va de $(x_n,\, F(x_n))$ à $(\bar{x},\, 0)$. En d'autres

FIG. 4.1 – Isaac Newton en 1726 (1643–1727)

termes, tel que

$$F'(z) = \frac{F(x_n) - F(\bar{x})}{x_n - \bar{x}} = \frac{F(x_n)}{x_n - \bar{x}}.$$

Si x_n et $\bar{x}$ sont proches alors $x_n \simeq z$ et on obtient :

$$\bar{x} \simeq x_n - \frac{F(x_n)}{F'(x_n)}.$$

Exemple. On applique la méthode de Newton pour calculer la racine positive de l'équation $x^2 - 4 = 0$, c'est–à–dire le nombre 2. On prend pour valeur initiale $x_0 = 3$.

```
Digits := 30:
F := x -> xˆ2 - 4;
                                        2
                                F := x -> x  - 4
Fprime := D(F);
                                Fprime := x -> 2 x
x[0] := 3.:
for n from 0 to 4 do
    x[n+1] := x[n] - F(x[n])/Fprime(x[n])
od;
                    x[1] := 2.166666666666666666666666667

                    x[2] := 2.006410256410256410256410256

                    x[3] := 2.000010240026214467109035799 13

                    x[4] := 2.000000000026214400000171798 69
```

```
x[5] := 2.000000000000000000000017179869
```

La vitesse de convergence est impressionnante : le nombre de décimales correctes double approximativement à chaque itération. On verra ci–dessous que c'est lié au fait que 2 est une racine simple de l'équation utilisée.

Exemple. Appliquons maintenant la méthode de Newton à l'équation $(x-2)^2 = 0$ dont 2 est racine double. La vitesse de convergence est nettement moins bonne !

```
F := x -> (x-2)^2;
                                        2
                              F := x -> (x - 2)
Fprime := D(F);
                            Fprime := x -> 2 x - 4

x[0] := 3.:
for n from 0 to 4 do
    x[n+1] := x[n] - F(x[n])/Fprime(x[n])
od;
                    x[1] := 2.500000000000000000000000000000

                    x[2] := 2.250000000000000000000000000000

                    x[3] := 2.125000000000000000000000000000

                    x[4] := 2.062500000000000000000000000000

                    x[5] := 2.031250000000000000000000000000
```

4.1.1 Convergence de la méthode

Dans cette section, on utilise MAPLE pour analyser théoriquement la relation entre la vitesse de convergence et la simplicité de la racine. La méthode de Newton peut diverger ou boucler et, même dans le cas où elle converge, elle peut ne pas converger vers la racine $\bar{x}$ désirée. Cependant, lorsqu'on étudie l'ordre d'une méthode numérique de résolution, on considère toujours une racine $\bar{x}$ et on suppose que la méthode converge bien vers elle. Voici une autre remarque importante : dans les méthodes numériques de résolution d'équations, il faut distinguer les erreurs de méthode des erreurs d'arrondis. On ne s'intéresse dans cette section qu'aux erreurs de méthode (erreurs qui persistent même si on mène tous les calculs avec des nombres exacts, ce qui est possible en MAPLE). Les erreurs dues aux arrondis sont beaucoup plus difficiles à estimer.

Ordre d'une méthode de résolution d'équations numériques

On définit l'erreur à l'étape n par $e_n \underset{\text{def}}{=} x_n - \bar{x}$.

Définition 9 *Une méthode de résolution d'équations numériques est dite d'ordre p si*

$$\lim_{n\to+\infty} \frac{e_{n+1}}{e_n^p} = \text{cste non nulle.}$$

Dire qu'une méthode est d'ordre un, c'est dire que le nombre de décimales correctes augmente linéairement avec n : à chaque étape, on gagne k nouvelles décimales correctes (pour une certaine constante k). Dire qu'une méthode est d'ordre deux, c'est dire que le nombre de décimales correctes augmente quadratiquement avec n : à chaque étape, le nombre de décimales correctes double (approximativement). Dans ce qui suit, on suppose que la méthode converge bien vers une racine $\bar{x}$.

Proposition 6 *Si $\bar{x}$ est une racine simple de F alors la méthode de Newton est d'ordre deux (au moins) sinon la méthode est d'ordre un.*

On montre avec MAPLE que la méthode de Newton est d'ordre deux dans le cas d'une racine simple. Comme x_n est proche de $\bar{x}$, on peut approximer le graphe de F par une parabole.

```
F := x -> a*x^2 + b*x + c;
```

$$F := x \to a\,x^2 + b\,x + c$$

```
Fp := D(F);
```

$$Fp := x \to 2\,a\,x + b$$

```
xbar := (- b - sqrt (b^2-4*a*c))/(2*a);
```

$$xbar := \frac{1}{2}\,\frac{-b - \sqrt{b^2 - 4\,a\,c}}{a}$$

On exprime x_{n+1} en fonction de x_n.

```
x(n+1) := x(n) - F(x(n))/Fp(x(n));
```

$$x(n+1) := x(n) - \frac{a\,x(n)^2 + b\,x(n) + c}{2\,a\,x(n) + b}$$

On porte la dynamique sur l'erreur : on cherche à exprimer e_{n+1} en fonction de e_n :

```
e(n+1) := x(n+1) - xbar;
```

$$e(n+1) := x(n) - \frac{a\,x(n)^2 + b\,x(n) + c}{2\,a\,x(n) + b} - \frac{1}{2}\,\frac{-b - \sqrt{b^2 - 4\,a\,c}}{a}$$

```
e(n+1) := subs (x(n) = e(n) + xbar, e(n+1)):
e(n+1) := simplify (e(n+1));
```

```
                                        2
                                     e(n)  a
e(n + 1) := - --------------------------
                                        2         1/2
                          -2 e(n) a + (b  - 4 a c)
```

Si la racine est simple, c'est–à–dire si $b^2 - 4\,a\,c > 0$ alors le rapport e_{n+1}/e_n^2 tend vers une constante non nulle quand e_n tend vers zéro.

```
simplify (e(n+1)/e(n)^2);
                                     a
                           - --------------------------
                                      2         1/2
                             -2 e(n) a + (b  - 4 a c)
subs (e(n) = 0, %);
                                     a
                                - --------------
                                   2         1/2
                                  (b  - 4 a c)
```

La méthode de Newton se généralise pour des systèmes d'équations non linéaires. On peut consulter à ce sujet le chapitre VI (page 138) du polycopié [5].

4.2 Intégration d'équations différentielles

Soient $f : [a, b] \times \mathbb{R} \to \mathbb{R}$ une fonction continue de deux variables réelles et $\alpha \in \mathbb{R}$ un réel. On cherche à résoudre des systèmes de la forme suivante (une équation différentielle ordinaire et une condition initiale) :

$$\mathscr{S} \begin{cases} x(a) = \alpha \\ x'(t) = f(t, x(t)). \end{cases}$$

4.2.1 Intégration symbolique

La fonction *dsolve* du logiciel MAPLE permet d'intégrer certaines de ces équations *symboliquement*. Lors d'une intégration symbolique, on peut ne pas préciser de condition initiale. Une constante arbitraire apparaît alors dans la solution. On peut aussi fixer une condition initiale. La constante arbitraire est alors déterminée par la condition initiale.

```
f := (t,x) -> t + x - 1;
                            f := (t, x) -> t + x - 1

edo := diff (x(t), t) = f (t, x(t));
                                d
                          edo := -- x(t) = t + x(t) - 1
                                dt
# intégration sans fixer de condition initiale
dsolve (edo, x(t));
                             x(t) = -t + exp(t) _C1
```

```
# intégration en fixant une condition initiale
dsolve ( { edo, x(1)=3 }, x(t) );
                                          exp(t)
                              x(t) = -t + 4 ------
                                          exp(1)
```

4.2.2 Champs de pentes et courbes intégrales

Malheureusement, les équations différentielles ordinaires ne peuvent pas être intégrées symboliquement pour la plupart. Cela ne signifie pas que ces équations n'ont pas de solution : elles en ont dès que la fonction f satisfait certaines conditions assez naturelles mais ces solutions ne peuvent pas se noter au moyen de formules finies faisant intervenir des logarithmes, des exponentielles etc ... Considérons par exemple l'équation différentielle

$$x'(t) = \frac{1}{|x(t)| + 0.1} - (t-2)\,\sin(x(t)).$$

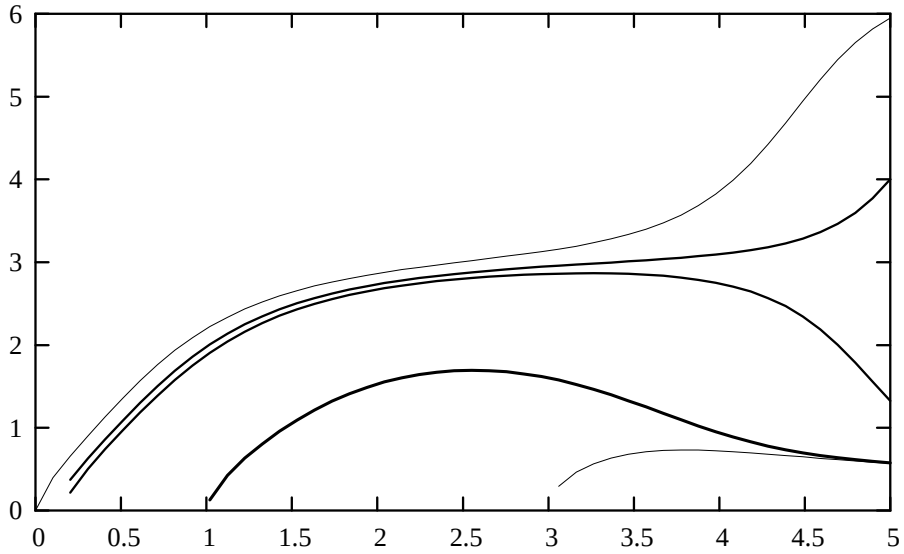
La fonction *dsolve* de MAPLE ne parvient pas à en calculer la solution, même pour une condition initiale fixée.

```
f := (t,x) -> 1 / (abs (x) + 0.1) - (t - 2) * sin (x);
                                     1
                  f := (t, x) -> ---------- - (t - 2) sin(x)
                                 | x | + .1

edo := diff (x(t),t) = f(t,x(t));
                       d              1
              edo := -- x(t) = ------------ - (t - 2) sin(x(t))
                       dt       | x(t) | + .1

dsolve ( { edo, x(0)= 1 } , x(t) );
# Pas de réponse
```

Cette équation différentielle a pourtant des solutions. Voici les graphes de cinq de ses solutions, c'est–à–dire de cinq *courbes intégrales* de l'équation, calculées (on verra plus tard comment) en fixant cinq conditions initiales différentes. À chaque condition initiale correspond une courbe intégrale. Le temps t figure en abscisse : on s'est restreint à $t \in [a,\, b] = [0,\, 5]$. L'axe des ordonnées correspond à $x(t)$. Bien qu'on n'ait tracé que cinq courbes, l'équation différentielle a une infinité de solutions. Il faut imaginer la bande $[a,\, b] \times \mathbb{R}$ complètement tapissée de courbes qui ne se coupent pas.



Les courbes intégrales ci–dessus ont été tracées par logiciel (voir les fonctions *odeplot* et *display* du paquetage *plots*) mais comment les a–t–on obtenues ? La seule information dont on dispose, c'est la fonction f. En tout point $(t,\, x) \in [a,\, b] \times \mathbb{R}$, le réel $f(t, x)$ (qui est lui, facile à calculer), nous donne la dérivée de la courbe intégrale qui passe par $(t,\, x)$, c'est–à–dire la pente de la tangente à cette courbe. Pour visualiser cette pente, on peut tracer en tout point de la bande $[a,\, b] \times \mathbb{R}$ le vecteur directeur de la tangente à la courbe, c'est–à–dire un vecteur de coordonnées

$$\begin{pmatrix} \Delta\, t \\ \Delta\, x \end{pmatrix} = \begin{pmatrix} 1 \\ f(t,\, x) \end{pmatrix}.$$

En pratique, on ne trace que quelques vecteurs (il y en a une infinité) et on ajuste leur longueur pour garder un graphique lisible. Voici le graphique obtenu sur l'exemple. Ce graphique représente ce qu'on appelle le *champ de pentes* de l'équation différentielle (voir la fonction *fieldplot* du paquetage *plots*).

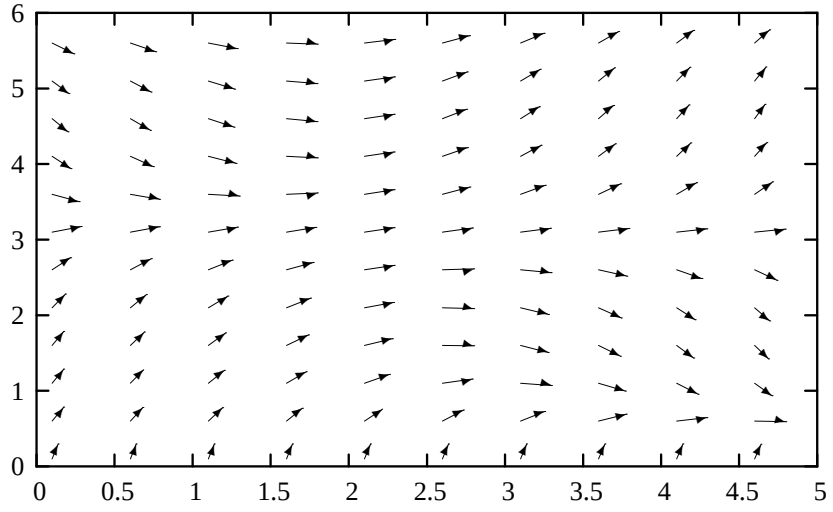
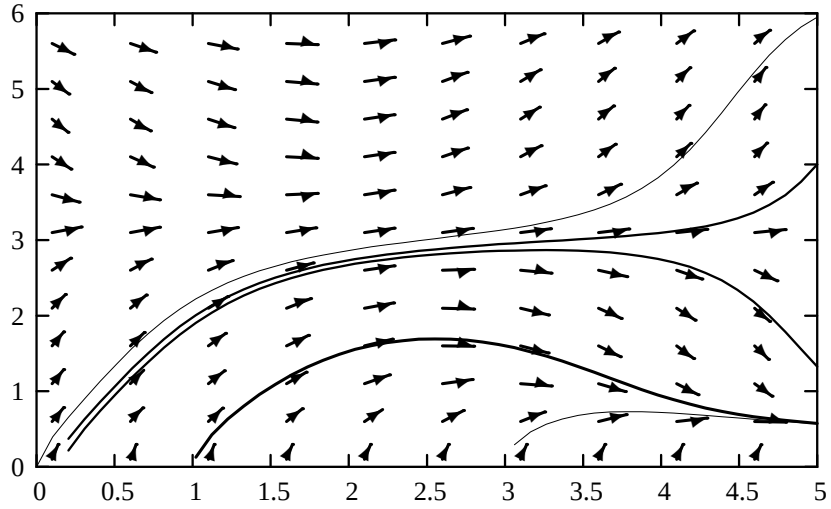


FIG. 4.2 – Leonhard Euler (1707–1783).

Si on superpose les deux graphiques, on vérifie visuellement que les vecteurs donnent les tangentes aux courbes intégrales. Dit autrement, de façon plus imagée, on voit que chaque courbe intégrale s'obtient « en plaçant la pointe d'un crayon n'importe où sur la bande et en suivant les flèches ». C'est cette idée qui est appliquée de manière plus ou moins sophistiquée par toutes les méthodes numériques d'intégration d'équations différentielles. La plus simple d'entre elles est la méthode de Leonhard Euler. On comprend au passage pourquoi les courbes intégrales ne se coupent pas : si deux courbes se coupaient, il y aurait deux vecteurs différents au point d'intersection des deux courbes[1].



[1]Cet argument suffit à montrer que deux courbes intégrales ne peuvent pas se couper « franchement » mais il ne suffit pas à montrer que deux courbes intégrales ne peuvent pas se couper « tangentiellement » (ce qui est vrai aussi). Il faut en toute rigueur utiliser le théorème 2, page 55.

4.2.3 Intégration numérique : la méthode d'Euler

On cherche à intégrer numériquement un système différentiel

$$\mathscr{S} \left\{ \begin{array}{l} x(a) = \alpha \\ x'(t) = f(t,\, x(t)). \end{array} \right.$$

sur un intervalle de temps $t \in [a,\, b]$. On suppose que la fonction $f\ : [a,\, b] \times \mathbb{R} \to \mathbb{R}$ satisfait certaines propriétés assez naturelles (elle doit être continue et ne pas tendre vers l'infini sur l'intervalle considéré) dont on reparlera ultérieurement.

Intégrer numériquement une équation différentielle munie d'une condition initiale, c'est chercher à approximer le graphe de la courbe intégrale de $x(t)$ sous la forme d'une liste finie de N points (le nombre de points est à fixer par l'utilisateur) :

$$(t_i,\, x_i) \quad 0 \leq i \leq N.$$

Attention à la distinction entre $x(t_i)$ et x_i ! On note $x(t_i)$ le « vrai » point de la courbe (celui qu'on cherche) et x_i le point calculé par la méthode. La méthode d'Euler consiste à subdiviser l'intervalle $[a,\, b]$ en N sous–intervalles de longueur

$$h = \frac{b-a}{N}$$

et à poser

$$t_i \underset{\text{def}}{=} a + h\, i.$$

Le premier point $(t_0,\, x_0)$ est donné par la première équation du système

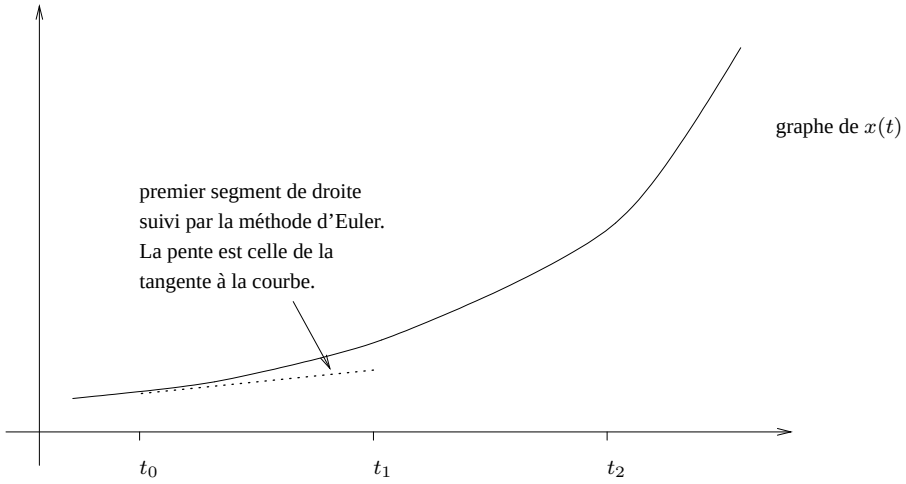
$$(t_0,\, x_0) = (a,\, \alpha).$$

Pour déterminer les points suivants, Euler a l'idée de considérer que sur un petit intervalle de temps h la courbe $x(t)$ recherchée est très proche de sa tangente. La pente de la tangente à la courbe en $(t_0,\, x_0)$ est donnée par la deuxième équation du système

$$x'(t_0) = f(t_0,\, x(t_0)) = f(t_0,\, x_0).$$

On obtient donc un deuxième point

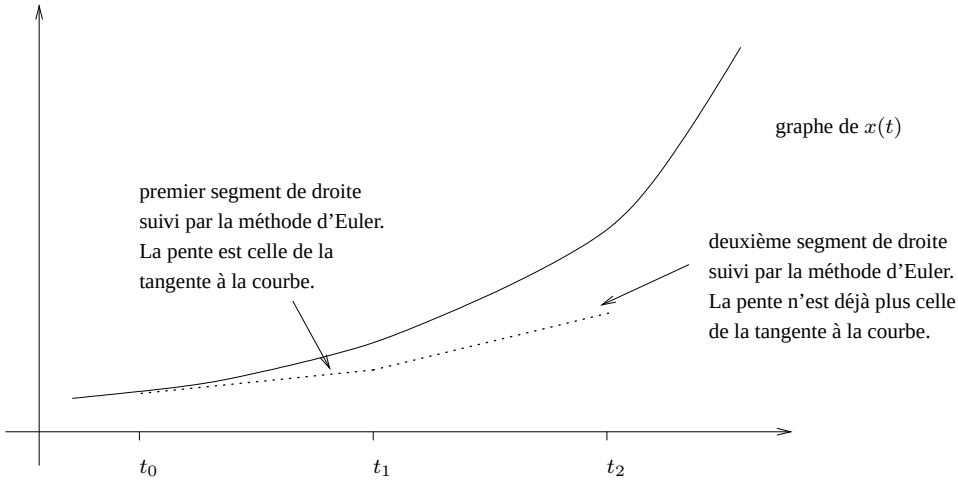
$$(t_1,\, x_1) \underset{\text{def}}{=} (t_0 + h,\, x_0 + h\, f(t_0,\, x_0)).$$



Aux étapes suivantes, Euler applique la même formule :

$$(t_{i+1},\, x_{i+1}) \underset{\text{def}}{=} (t_i + h,\, x_i + h\, f(t_i,\, x_i)).$$

La formule est la même mais son interprétation est un peu différente. Considérons par exemple le calcul du point d'indice $i = 2$. Le point $(t_1,\, x_1)$ dont on part n'est déjà plus exactement sur la courbe : on a sauté d'une courbe intégrale à une autre courbe intégrale de l'équation différentielle. Pour cette raison, comme $x_1 \neq x(t_1)$, la « pente de la tangente » $f(t_1,\, x_1)$ dont on se sert dans la formule n'est pas non plus exactement la « vraie » pente $f(t_1,\, x(t_1))$. La méthode cumule donc deux erreurs.



Exemple

On illustre la méthode d'Euler sur un exemple extrêmement simple. On s'intéresse au système suivant, sur l'intervalle de temps $[a,\, b] = [0,\, 1]$. Un calcul rapide montre que $x(1) = 1$.

$$x(a) = \frac{1}{e}, \quad x'(t) = x(t).$$

Les instructions ci–dessous appliquent la méthode d'Euler pour $N = 5$.

```
a := 0: b := 1:
f := x -> x:
x[0] := evalf (1/exp(1));
                              x[0] := .3678794412
N := 5: h := (b - a)/N:
for i from 1 to N do
    x[i] :=  x[i-1] + h*f(x[i-1])
od:
seq ([a+h*i, x[i]], i = 0 .. N);
    [0, .3678794412], [1/5, .4414553294], [2/5, .5297463953],

    [3/5, .6356956744], [4/5, .7628348093], [1, .9154017712]
```

On voit que l'approximation x_N de $x(1)$ n'est pas très bonne (une décimale correcte). On constate empiriquement que pour obtenir une décimale correcte supplémentaire, il faut multiplier le nombre de pas par 10. Voici les valeurs de x_N pour différentes valeurs de N.

N	x_N
5	$.9154017712$
50	$.9901799003$
500	$.9990018316$
5000	$.9999000225$

Ordre d'une méthode d'intégration d'équations différentielles

On définit l'erreur en t_i par

$$e_i \underset{\text{def}}{=} x(t_i) - x_i.$$

Définition 10 *Une méthode d'intégration d'équations différentielles est dite d'ordre p s'il existe une constante K telle que $|e_i| \leq K\, h^p$.*

Dans le cas où la constante K vaut 1, dire qu'une méthode est d'ordre p c'est dire qu'à chaque fois qu'on divise la longueur du pas h par 10, on gagne p décimales correctes supplémentaires pour x_N.

L'exemple traité ci–dessus suggère que la méthode d'Euler est d'ordre 1. En effet, à chaque fois qu'on multiplie N par 10, le pas h est divisé par 10. Comme l'erreur est elle–aussi divisée par 10, c'est que l'exposant p vaut 1. Une démonstration rigoureuse sera faite ultérieurement.

Remarque. Le même mot « ordre » est utilisé deux fois dans ce cours : une fois pour les équations numériques et une fois pour les équations différentielles. Dans les deux cas l'idée est la même : c'est une mesure de l'efficacité théorique de la méthode (plus l'ordre est élevé, plus la méthode est efficace). Mais les définitions sont en fait techniquement différentes et ne doivent pas être confondues.

4.2.4 Le solveur de MAPLE

La méthode d'Euler n'est jamais utilisée en pratique parce qu'elle est d'ordre trop petit. Parmi les méthodes les plus populaires, on trouve les méthodes dites de Runge–Kutta. On en rencontre d'ordre 2 à 8. Le solveur numérique de MAPLE utilise une variante d'une méthode de Runge–Kutta d'ordre quatre. Voici un exemple. L'équation différentielle ne semble pas pouvoir être intégrée symboliquement.

```
eq := diff (x(t),t) = 3*t*x(t)^2 + sin(x(t)) - 1;
                          d                   2
                 eq := -- x(t) = 3 t x(t)  + sin(x(t)) - 1
                          dt

dsolve ( { eq, x(0)=1 }, x(t) );
# Pas de réponse
```

On peut par contre l'intégrer numériquement. L'intégrateur numérique de MAPLE retourne une fonction $t \mapsto [t, x_t]$ qui met en œuvre une variante d'une méthode de Runge–Kutta d'ordre quatre[2]. La solution retournée par le solveur numérique de MAPLE peut être tracée à l'écran en utilisant la fonction *odeplot* du paquetage *plots*.

```
sol := dsolve ( { eq, x(0) = 1 }, x(t), numeric );
                          sol := proc (x_rfk45) ... end

seq (sol(i/10), i = 0 .. 6);

  [t = 0., x(t) = 1.],

        [t = 0.100000000000000, x(t) = 0.998887400703896367],

        [t = 0.200000000000000, x(t) = 1.02983892519049691],

        [t = 0.300000000000000, x(t) = 1.10213372260592268],

        [t = 0.400000000000000, x(t) = 1.23679532944053761],

        [t = 0.500000000000000, x(t) = 1.48114655648673788],

        [t = 0.600000000000000, x(t) = 1.95832848633099865]

# Pour obtenir la tracé de la courbe intégrale
> with (plots):
> odeplot (sol, 0..1);
```

[2] Précisément, il s'agit un schéma de type Runge–Kutta–Fehlberg à *pas adaptatif* : la longueur du pas h varie au cours de l'intégration en fonction de la difficulté à suivre la courbe intégrale. L'estimation de l'erreur commise à chaque pas (et donc de la difficulté) se fait grâce à une autre formule, « emboîtée », qui est ici d'ordre cinq.

4.2.5 Intégration d'un système différentiel

La méthode d'Euler et les méthodes de type Runge–Kutta se généralisent facilement pour intégrer des systèmes d'équations différentielles. Dans le texte qui suit, on note les indices en exposant. Soient $f^1,\ f^2\ : \mathbb{R}^2 \to \mathbb{R}$ deux fonctions de deux variables réelles et $\alpha,\ \beta \in \mathbb{R}$ deux réels. On cherche à intégrer numériquement le système suivant sur un intervalle $t \in [a,\ b]$:

$$x(a) = \begin{pmatrix} x^1(a) \\ x^2(a) \end{pmatrix} = \begin{pmatrix} \alpha \\ \beta \end{pmatrix}, \quad x'(t) = \begin{pmatrix} x^{1\prime}(t) \\ x^{2\prime}(t) \end{pmatrix} = \begin{pmatrix} f^1(t,\, x^1(t),\, x^2(t)) \\ f^2(t,\, x^1(t),\, x^2(t)) \end{pmatrix}.$$

La méthode d'Euler, appliquée à ce système consiste à fixer un nombre de pas N, à poser $h = (b-a)/N$ et à calculer les N premiers termes de la suite définie par :

$$x_0 = \begin{pmatrix} \alpha \\ \beta \end{pmatrix}, \quad x_{n+1} = \begin{pmatrix} x^1_{n+1} \\ x^2_{n+1} \end{pmatrix} = \begin{pmatrix} x^1_n + h\, f^1(t,\, x^1_n,\, x^2_n) \\ x^2_n + h\, f^2(t,\, x^1_n,\, x^2_n) \end{pmatrix}.$$

Dans le cas particulier des systèmes différentiels linéaires, les fonctions f^1 et f^2 peuvent être représentées par une matrice A. Par exemple, le système

$$x'(t) = \begin{pmatrix} x^{1\prime}(t) \\ x^{2\prime}(t) \end{pmatrix} = \begin{pmatrix} x^1(t) + x^2(t) \\ x^1(t) - 2\, x^2(t) \end{pmatrix}$$

peut être noté

$$x'(t) = A\, x(t)$$

avec

$$x'(t) = \begin{pmatrix} x^{1\prime}(t) \\ x^{2\prime}(t) \end{pmatrix}, \quad x(t) = \begin{pmatrix} x^1(t) \\ x^2(t) \end{pmatrix}, \quad A = \begin{pmatrix} 1 & 1 \\ 1 & -2 \end{pmatrix}.$$

Dans ce cas, la suite de points construite par la méthode d'Euler peut être notée

$$x_0 = \begin{pmatrix} \alpha \\ \beta \end{pmatrix}, \quad x_{n+1} = x_n + h\, A\, x_n.$$

```
A := <<1 | 1>, <1 | -2 >>;
```

```
                                    [1     1]
                              A := [        ]
                                    [1    -2]
```

```
h := 0.01:
x[0] := <1, 1>;
```

```
                                      [1]
                                x[0] := [ ]
                                      [1]
```

```
for n from 0 to 5 do
   x [n+1] := x [n] + h * A . x[n]
od;
```

```
                                [1.02000000000000002 ]
                          x[1] := [                    ]
                                [0.989999999999999992]
```

```
                    [1.04010000000000002 ]
           x[2] := [                     ]
                    [0.980399999999999938]

                    [1.06030500000000005 ]
           x[3] := [                     ]
                    [0.971192999999999973]

                    [1.08061998000000004 ]
           x[4] := [                     ]
                    [0.962372189999999961]

                    [1.10104990169999994 ]
           x[5] := [                     ]
                    [0.953930945999999946]

                    [1.12159971017699989 ]
           x[6] := [                     ]
                    [0.945862826096999898]
```

Le logiciel MAPLE permet aussi de résoudre numériquement des systèmes linéaires. Comme dans le cas d'une variable, c'est une méthode de type Runge–Kutta qui est appliquée, d'ordre plus élevé que la méthode d'Euler.

```
eq1 := diff (x1(t),t) = x1(t) + x2(t);
                              d
                        eq1 := -- x1(t) = x1(t) + x2(t)
                              dt

eq2 := diff (x2(t),t) = x1(t) - 2*x2(t);
                             d
                       eq2 := -- x2(t) = x1(t) - 2 x2(t)
                             dt

sol := dsolve ({eq1, eq2, x1(0)=1, x2(0)=1}, {x1(t), x2(t)}, numeric);
                        sol := proc(rkf45_x)  ...  end

sol (0.1);
        [t = .1, x1(t) = 1.205826386250551, x2(t) = .9189092967731726]
```

4.2.6 Convergence de la méthode d'Euler

Revenons au cas d'une seule équation différentielle avec une condition initiale. Il est utile de rappeler la remarque faite au sujet de la méthode de Newton : dans les méthodes d'intégration numérique, il faut distinguer les erreurs de méthode des erreurs d'arrondis. On ne s'intéresse dans cette section qu'aux erreurs de méthode (erreurs qui persistent même si on mène tous les calculs avec des nombres exacts, ce qui est possible en MAPLE). Les erreurs dues aux arrondis sont beaucoup plus difficiles à estimer. Pour commencer, il faut préciser certaines hypothèses sur la fonc-

FIG. 4.3 – Augustin Louis Cauchy (1789–1857). « *M. Cauchy annonce, que, pour se conformer au vœu du Conseil, il ne s'attachera plus à donner, comme il l'a fait jusqu'à présent, des démonstrations parfaitement rigoureuses.* » (Conseil d'instruction de l'École Polytechnique, 24 novembre 1825).

tion f faites informellement dans les sections précédentes. Le théorème suivant est dû à Augustin Cauchy.

Théorème 2 *S'il existe une constante L telle que, quels que soient $t \in [a,\, b]$ et $x_0,\, x_1 \in \mathbb{R}$ on ait*

$$|f(t,\, x_0) - f(t,\, x_1)| \leq L\,|x_0 - x_1|$$

alors le système

$$\mathscr{S} \left\{ \begin{array}{l} x(a) = \alpha \\ x'(t) = f(t,\, x(t)). \end{array} \right.$$

admet une unique solution au moins une fois dérivable.

Une fonction f qui satisfait la condition du théorème est dite « lipschitzienne » en x, en l'honneur du mathématicien Rudolf Lipschitz. Si la dérivée partielle $\partial f/\partial x$ est bornée en valeur absolue sur la bande $[a,\, b] \times \mathbb{R}$ alors f est lipschitzienne en x. Cela se montre assez facilement avec le théorème des accroissements finis.

Revenons à la méthode d'Euler. On rappelle qu'on définit l'erreur en t_i par

$$e_i \underset{\text{def}}{=} x(t_i) - x_i.$$

FIG. 4.4 – Rudolf Lipschitz (1832–1903).

Théorème 3 *Supposons vérifiées les conditions du théorème 2. Supposons de plus que, sur l'intervalle* $[a,\, b]$*, la solution* $x(t)$ *du système* $\mathscr{S}$ *est au moins deux fois dérivable et admet une dérivée seconde* $x''(t)$ *bornée en valeur absolue.*

Alors la méthode d'Euler est d'ordre 1*. En d'autres termes, il existe une constante* K *telle que* $|e_i| \leq K\, h$*.*

Le théorème est un théorème de convergence. Il prouve en effet que si h tend vers zéro alors l'erreur tend vers zéro aussi. On démontre le théorème à l'aide de MAPLE. On porte la dynamique sur l'erreur : on cherche à exprimer e_{n+1} en fonction de e_n puis à résoudre la relation de récurrence. Les calculs sont faits avec MAPLE. Il ne nous reste qu'à tenir les raisonnements. C'est, pour le théorème 3, une « démonstration assistée par ordinateur ».

D'après la formule de Taylor et les hypothèses du théorème, la fonction $x(t)$ solution de l'équation différentielle admet un développement à l'ordre 2 au voisinage de t_n de la forme

$$x(t_{n+1}) \underset{\text{def}}{=} x(t_n + h) = x(t_n) + h\, x'(t_n) + \frac{h^2}{2}\, x''(\tau) = x(t_n) + h\, f(t_n,\, x(t_n)) + \frac{h^2}{2}\, x''(\tau)$$

où $0 \leq \tau \leq h$. En notant M_2 la borne sur la dérivée seconde $x''(\tau)$ on obtient une majoration

```
Taylor := x(t(n+1)) <= x(t(n)) + h*f(t(n), x(t(n))) + M2/2*h^2:
Taylor;
                                                                  2
            x(t(n + 1)) <= x(t(n)) + h f(t(n), x(t(n))) + 1/2 M2 h
```

La formule de la méthode d'Euler nous donne une relation

```
Euler := x(n+1) = x(n) + h*f(t(n), x(n)):
Euler;
                  x(n + 1) = x(n) + h f(t(n), x(n))
```

En soustrayant les deux relations terme à terme on obtient une majoration

```
erreur := Taylor - Euler:
erreur;

  -x(n + 1) + x(t(n + 1)) <=
                                                                          2
    -x(n) - h f(t(n), x(n)) + x(t(n)) + h f(t(n), x(t(n))) + 1/2 M2 h
```

Sachant que e_n est par définition égale à $x(t_n) - x_n$, on peut réécrire la relation précédente (je ne connais pas de façon très élégante de le faire)

```
erreur := subs (x(t(n+1)) = e(n+1) + x(n+1),
                  x(t(n)) = e(n) + x(n),
                  f(t(n), e(n) + x(n)) = f(t(n), x(t(n))), erreur):
erreur;
                        2
      e(n + 1) <= 1/2 h  M2 + h f(t(n), x(t(n))) - h f(t(n), x(n)) + e(n)
```

On reporte l'hypothèse du théorème 2 c'est–à–dire

$$|f(t_n,\, x(t_n)) - f(t_n,\, x_n)| \leq L\,|x(t_n) - x_n| \underset{\text{def}}{=} L\,|e_n|.$$

La constante L apparaît. Remarquer qu'en toute rigueur, il faudrait faire apparaître des symboles de valeur absolue.

```
erreur := subs (f(t(n), x(t(n))) =
                     L*(x(t(n)) - x(n)) + f(t(n), x(n)), erreur):
erreur := subs (x(t(n)) = e(n) + x(n), erreur):
erreur := collect (erreur, e(n)):
erreur;
                                                          2
                    e(n + 1) <= (h L + 1) e(n) + 1/2 h  M2
```

On a obtenu la relation désirée. La solution de la relation de récurrence obtenue en remplaçant le « $\leq$ » par un « $=$ » fournit une majoration de l'erreur. La condition initiale $e_0 = 0$ traduit le fait qu'en $t = t_0$ l'erreur est nulle.

```
recurrence := lhs (erreur) = rhs (erreur):
recurrence;
                                                          2
                     e(n + 1) = (h L + 1) e(n) + 1/2 h  M2
erreur := e(n) <= rsolve ( { recurrence, e(0) = 0 }, e(n) ):
erreur;
                                              n
                                M2 h (h L + 1)        M2 h
                    e(n) <= 1/2 -------------- - 1/2 ----
                                          L              L
```

Le calcul n'est pas fini : on ne peut pas laisser le n en exposant. L'astuce consiste à remarquer que, quel que soit n on a

$$(1 + L\,h)^n \leq e^{L\,n\,h} \leq e^{L(b-a)}.$$

La deuxième inégalité est évidente puisque $n\,h \leq b - a$. Pour la première, il suffit d'observer que $1 + L\,h \leq e^{L\,h}$, en procédant par exemple à un calcul de développement limité

```
taylor (exp (L*h), h, 2);
                                    2
                          1 + L h + O(h )
```

On obtient ainsi une majoration de l'erreur de la forme $K\,h$. On a même une expression pour K.

```
erreur := subs ((L*h+1)^n = exp(L*(b-a)), erreur):
erreur := collect (erreur, h):
erreur;
                      /    M2 exp(b - a)        M2 \
                      |1/2 ------------- - 1/2 ----| h
                      \          L               L  /
```

4.2.7 Méthode d'Euler d'ordre deux

L'idée consiste à considérer que sur un petit intervalle de temps h la courbe $x(t)$ recherchée peut être approximée par une parabole au lieu d'une droite. Pour mener les calculs on a besoin d'une formule pour la dérivée seconde de $x(t)$. On l'obtient en dérivant la seconde équation de $\mathscr{S}$ et en appliquant la formule de la dérivée des fonctions composées :

$$f'(t,\,x) = \frac{\partial f}{\partial t}(t,\,x)\,t' + \frac{\partial f}{\partial x}(t,\,x)\,x'$$

ce qui nous donne, en remplaçant t' par 1 et x' par sa valeur

$$x''(t) = \left(\frac{\partial f}{\partial t} + \frac{\partial f}{\partial x}\,f\right)\,(t,\,x(t)).$$

En appliquant la même idée que pour la méthode d'Euler on obtient une formule

$$x_{i+1} \underset{\text{def}}{=} x_i + h\,f(t_i,\,x_i) + \frac{h^2}{2}\,\left(\frac{\partial f}{\partial t} + \frac{\partial f}{\partial x}\,f\right)\,(t_i,\,x_i).$$

Théorème 4 *Supposons vérifiées les conditions du théorème 2. Supposons de plus que, sur l'intervalle $[a,\,b]$, la solution $x(t)$ du système $\mathscr{S}$ soit au moins trois fois dérivable et que la fonction*

$$\left(\frac{\partial f}{\partial t} + \frac{\partial f}{\partial x}\,f\right)\,(t,\,x)$$

soit lipschitzienne en x. Alors il existe une constante K telle que l'erreur

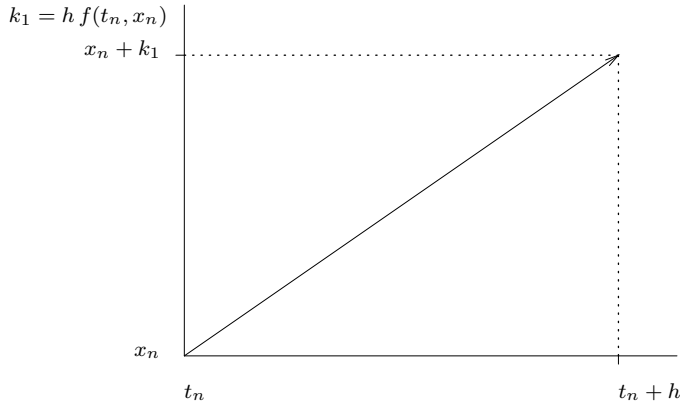
$$|e_n| \underset{\text{def}}{=} |x(t_n) - x_n| \leq K\,h^2.$$

La preuve du théorème met en œuvre les mêmes raisonnements et majorations que pour la méthode d'Euler. Elle est laissée en exercice. On obtient ainsi une méthode d'ordre deux.


FIG. 4.5 – Carle David Tolmé Runge (1856–1927) et Martin Wilhelm Kutta (1867–1944).

4.2.8 Méthodes de Runge et Kutta

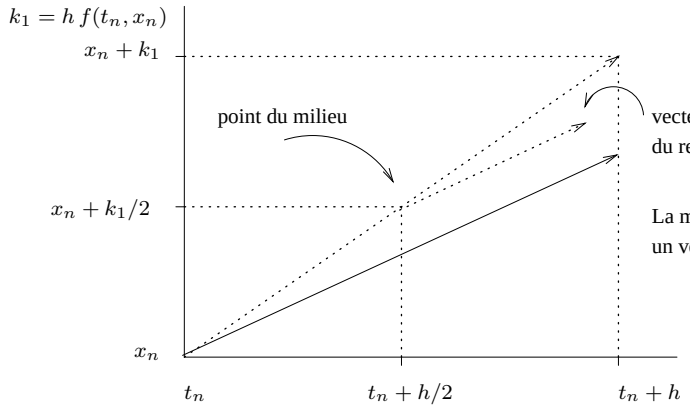
Les méthodes de Runge–Kutta ont été inventées vers 1900 par les mathématiciens allemands Carle Runge et Martin Wilhelm Kutta. La méthode d'Euler consiste à suivre le vecteur de pente $f(t_n,\, x_n)$ sur un pas de temps h à partir du point $(t_n,\, x_n)$.



Un pas de la méthode d'Euler définit un rectangle

Pourquoi ce vecteur–là et pas un autre ? Une meilleure approximation est obtenue en suivant le vecteur de pente $f(t_n + h/2,\, x_n + k_1/2)$ situé au milieu du rectangle défini par la méthode d'Euler. On obtient ainsi une méthode de Runge–Kutta d'ordre deux (celle inventée par Runge en 1895) :

$$\begin{aligned} k_1 &= h\, f(t_n,\, x_n) \\ k_2 &= h\, f\left(t_n + \frac{h}{2},\, x_n + \frac{k_1}{2}\right) \\ x_{n+1} &= x_n + k_2. \end{aligned}$$



Il existe des méthodes de Runge–Kutta d'ordre quelconque. Voici une formule possible pour l'ordre quatre[3] :

$$\begin{aligned} k_1 &= h\,f(t_n,\, x_n) \\ k_2 &= h\,f\left(t_n + \frac{h}{2},\, x_n + \frac{k_1}{2}\right) \\ k_3 &= h\,f\left(t_n + \frac{h}{2},\, x_n + \frac{k_2}{2}\right) \\ k_4 &= h\,f(t_n + h,\, x_n + k_3) \\ x_{n+1} &= x_n + \frac{1}{6}\,(k_1 + 2\,k_2 + 2\,k_3 + k_4). \end{aligned}$$

On voit que la nouvelle valeur x_{n+1} est obtenue en suivant sur un pas de temps h un vecteur dont la pente est la moyenne pondérée de plusieurs pentes : au facteur h près, k_1 est la pente d'un vecteur situé au début de l'intervalle $[t_n,\, t_n + h]$, les nombres k_2 et k_3 sont les pentes de vecteurs pris au milieu de l'intervalle, k_4 est la pente d'un vecteur situé à la fin de l'intervalle. Les pentes prises au milieu de l'intervalle ont un coefficient plus élevé que celles des points aux extrémités. Ces coefficients ne sont pas choisis au hasard : ils assurent que la méthode est bien d'ordre 4. Pourtant, ils ne sont pas définis de façon unique : il existe d'autres jeux de coefficients possibles définissant une méthode d'ordre 4.

L'intégration numérique des équations différentielles ordinaires est en fait la spécialité de E. Hairer. Pour cette raison, on recommande la lecture (même superficielle) du très intéressant chapitre III du polycopié [5]. On pourra aussi lire le chapitre I, consacré aux calculs d'intégrales, qui lui est fortement lié (un calcul d'intégrale est un cas particulier d'intégration d'équation différentielle).

[3] Le schéma de Runge–Kutta donné pour l'ordre deux a deux « étages » (il y a deux k_i calculés). De même, le schéma donné pour l'ordre quatre a quatre étages. Ce phénomène ne se généralise pas : à partir de $p = 5$, il faut plus que p étages pour obtenir un schéma de Runge–Kutta d'ordre p. Voir [6, Theorem 5.1, page 173].

Chapitre 5

Analyse qualitative d'équations différentielles

Ce cours est fortement inspiré des pages accessibles à l'URL `http://www.sosmath.com`. Beaucoup de phénomènes dynamiques peuvent se modéliser au moyen d'équations différentielles mais les équations ainsi obtenues ne peuvent généralement pas s'intégrer formellement. On peut les intégrer numériquement pour des conditions initiales données (cf. cours précédents) mais ce n'est pas suffisant. On a souvent besoin d'avoir une idée intuitive du comportement (de la « forme ») des courbes intégrales et de la façon dont ce comportement varie avec les conditions initiales. C'est cette analyse qualitative des solutions d'équations différentielles qui est abordé dans ce cours.

5.1 Analyse qualitative d'une équation en une variable

5.1.1 Équations différentielles autonomes

Dans ce cours et dans le suivant, on se restreint à des équations différentielles *autonomes* c'est–à–dire à des équations différentielles

$$x'(t) = f(x(t)) \qquad \text{ou encore} \qquad x' = f(x)$$

dont le second membre ne dépend pas explicitement de t. Il s'agit d'un cas particulier du cas étudié lors d'un cours précédent.

5.1.2 Fonctions constantes solutions

Elles jouent un rôle essentiel lors de l'analyse qualitative. On les appelle aussi *équilibres* ou encore *points fixes* de l'équation différentielle. On cherche les solutions de l'équation différentielle de la forme

$$x(t) = \textit{constante}.$$

En dérivant la relation ci–dessus, on voit qu'elle satisfont la relation $x'(t) = 0$ (quel que soit t).

Proposition 7 *Les solutions constantes sont les solutions de l'équation* $f(x) = 0$.

Lien avec la méthode d'Euler. Cette méthode, appliquée à l'équation différentielle, calcule une suite de points

$$x_{n+1} = x_n + h\, f(x_n)$$

où $h > 0$ est un certain pas de temps. Dire que $x(t)$ est une fonction constante, c'est dire que $x_0 = x_1 = \cdots = x_n = \cdots$ et donc que $f(x_n) = 0$.

5.1.3 La première chose à faire : chercher les solutions constantes

On commence par montrer sur des exemples que ces solutions ont « toujours » une signification particulière. Dans certaines limites, on sait bien que plus une population est importante plus elle croît vite. Cette affirmation étant trop imprécise pour permettre la mise en équation de la croissance de la population, on peut faire une hypothèse supplémentaire : on pose que sur tout intervalle de temps suffisamment petit, le nombre de naissances est proportionnel à la population existante. En notant $k > 0$ la constante de proportionnalité et $x(t)$ le nombre d'individus à l'instant t on obtient un modèle de croissance de poulation décrit par l'équation (modèle exponentiel) :

$$x' = k\, x.$$

La solution de cette équation différentielle est une famille d'exponentielles

$$x(t) = C\, e^{k\, t}$$

où la constante $C \geq 0$ est donnée par la condition initiale $x(0) = C$. La seule solution constante est la solution $x(t) = 0$. Elle correspond au cas où il n'y a aucun individu initialement. Comme il n'y a pas génération spontanée de population, c'est le cas où l'espèce est éteinte.

On voit que le modèle exponentiel n'est pas très réaliste puisque la croissance d'une population est souvent limitée par des contraintes écologiques. On utilise donc souvent un autre modèle (modèle logistique) pour pallier cet inconvénient. L'équation dépend de deux constantes $k,\, M > 0$:

$$x' = k\, x\, \left(1 - \frac{x}{M}\right).$$

Les solutions constantes sont les solutions $x(t) = 0$ et $x(t) = M$. La solution $x(t) = 0$ correspond au cas d'une espèce éteinte. La solution $x(t) = M$ correspond au cas d'une espèce dont la population est maximale. Les conditions initiales qui donnent ces solutions sont bien sûr $x(0) = 0$ et $x(0) = M$.

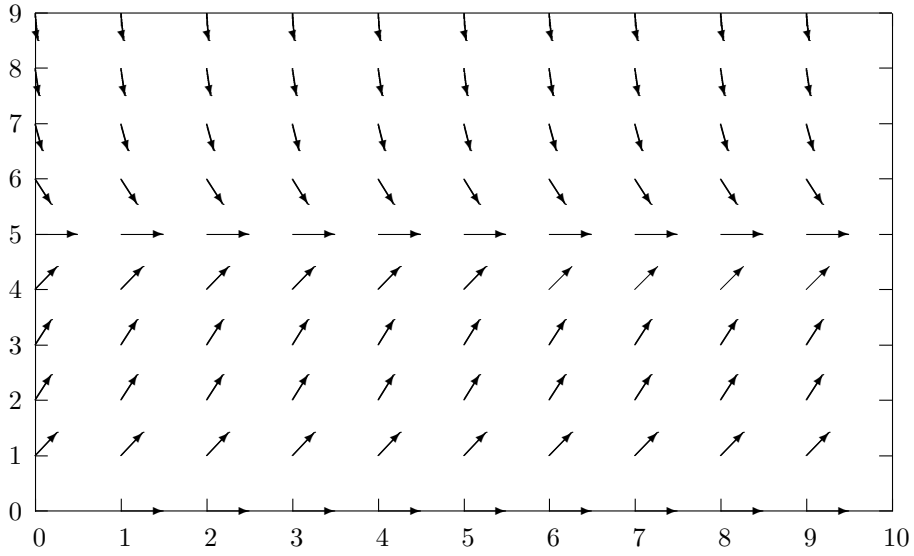
5.1.4 La seconde chose à faire : étudier le signe de f

Mais comment se comportent les solutions non constantes de l'équation différentielle ? On se retreint naturellement au cas où f est une fonction continue. Il suffit d'étudier le signe de f entre ses racines.

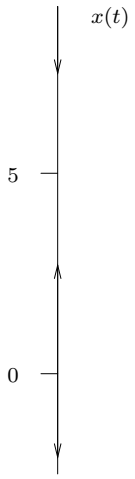
Proposition 8 *Entre deux racines $f(x)$ garde un signe constant.*

Corollaire 1 *Soient $\alpha < \beta$ deux racines consécutives de f et $\alpha < x(0) < \beta$ une condition initiale. Si $f(x(0)) > 0$ la solution $x(t)$ de condition initiale $x(0)$ est croissante et tend asymptotiquement vers β. Si $f(x(0)) < 0$ alors la solution $x(t)$ de condition initiale $x(0)$ est décroissante et tend asymptotiquement vers α.*

Revenons à l'exemple du modèle logistique. Si la condition initiale $0 < x(0) < M$ alors $f(x(0)) > 0$, la solution $x(t)$ croît et tend vers la valeur limite M sans jamais l'atteindre. Si la condition initiale $M < x(0)$ alors $f(x(0)) < 0$, la solution $x(t)$ décroît et tend vers la valeur limite M sans jamais l'atteindre. La constante M est appelée la « capacité maximale ». Enfin si la condition initiale $x(0) < 0$ (cas sans signification physique) alors $f(x(0)) < 0$, la solution $x(t)$ décroît et tend vers $-\infty$. On dit que la solution constante $x(t) = M$ est un équilibre « attractif » alors que la solution constante $x(0) = 0$ est un équilibre « répulsif ». Sur le diagramme suivant, l'axe horizontal correspond à t et l'axe vertical à x. En chaque point, on a tracé le vecteur de coordonnées $(t,\ f(x))$. On a pris $k = 2$ et $M = 5$. La méthode d'Euler consiste, partant d'un point, à suivre le vecteur en ce point sur un intervalle de temps h.



Ce diagramme est invariant par translation horizontale. C'est normal puisque l'équation différentielle est autonome. On peut le représenter plus économiquement en projetant les vecteurs sur l'axe vertical (l'axe des x donc). On obtient ce qu'on appelle une *ligne de phase*. Les solutions constantes apparaissent comme des points (d'où l'appellation de *points fixes*).



5.1.5 Cas d'un système

Cette section 5.1.5 est donnée pour titiller la curiosité des étudiants. Elle ne fait pas partie du programme du cours.

Dans beaucoup de cas on cherche plusieurs fonctions du temps. Considérons par exemple le système suivant, qui décrit l'évolution de deux populations (des loups et des renards par exemple) en lutte pour une même proie. On note $x(t)$ le nombre de loups et $y(t)$ le nombre de renards à l'instant t.

$$\begin{cases} x' = x\,(1-x) - x\,y, \\ y' = 2\,y\,(1-y/2) - 3\,x\,y. \end{cases} \qquad \text{ou encore} \qquad \begin{cases} x' = f(x,\,y), \\ y' = g(x,\,y). \end{cases}$$

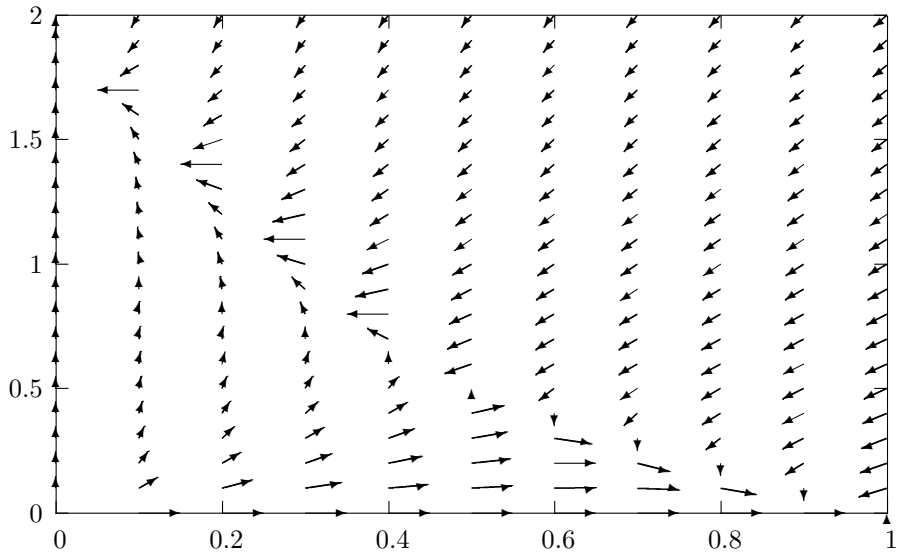
On reconnaît un modèle logistique pour chaque espèce plus un terme non linéaire qui modélise le fait que les renards et les loups s'entretuent à chaque fois qu'ils se rencontrent. La probabilité de rencontre est faible quand les espèces sont peu nombreuses ou quand l'une des deux espèces est presque éteinte. Les solutions constantes du système différentiel sont les solutions du système algébrique

$$\begin{cases} 0 = x\,(1-x) - x\,y, \\ 0 = 2\,y\,(1-y/2) - 3\,x\,y. \end{cases}$$

Sur l'exemple, ce sont les quatre couples de fonctions constantes $(x(t),\,y(t)) = (0,\,0)$, $(0,\,2)$, $(1/2,\,1/2)$ et $(1,\,0)$.

Pour se faire une idée du comportement du système, on représente graphiquement le *plan de phase* défini par le système. Ce graphique est la généralisation en deux variables de la ligne de phase tracée à la fin de la section précédente : en un point $(x,\,y) \in \mathbb{R}^2$ donné, on place le vecteur de coordonnées $(f(x,\,y),\,g(x,\,y))$. On a deux informations : l'orientation du vecteur et sa norme (sauf qu'ici, pour des raisons de lisibilité, on a imposé une longueur maximale arbitraire). Sur beaucoup de logiciels, les vecteurs sont dessinés avec des longueurs fixes ; on représente leur norme en faisant varier leur couleur.

Fonction MAPLE : *dfieldplot* du paquetage *DEtools*.



Pour bien comprendre le graphique, il faut prendre conscience qu'il est la projection sur un plan d'un graphique en trois dimensions (le troisième axe représente le temps). Les quatre fonctions constantes solutions du système différentiel apparaissent comme quatre points. Ces quatre points fixes ont des comportements très différents. Les points $(0,\, 0)$ et $(1/2,\, 1/2)$ sont répulsifs. Le point $(0,\, 0)$ est ce qu'on appelle une « source ». Le point $(1/2,\, 1/2)$ est ce qu'on appelle un « col ». les deux autres points fixes sont attractifs. Ce sont ce qu'on appelle des « puits » mais les trajectoires ne s'en approchent pas de la même façon. L'analyse qui précède peut être faite de façon rigoureuse en approximant le système différentiel par un système différentiel linéaire au voisinage de chacun des points fixes et en étudiant ces derniers (le système linéaire varie avec le point fixe). On voit que le comportement qualitatif d'un système de deux variables est nettement plus riche que celui d'un système d'une variable. Les systèmes de trois variables et plus sont encore plus riches. C'est d'ailleurs à partir de trois variables qu'on commence à constater des phénomènes « chaotiques ».

5.2 Résolution formelle des systèmes différentiels linéaires

On s'intéresse aux systèmes différentiels linéaires de la forme

$$\begin{cases} x_1' = a\,x_1 + b\,x_2 \\ x_2' = c\,x_1 + d\,x_2 \end{cases}$$

qu'on peut noter matriciellement

$$\begin{pmatrix} x_1' \\ x_2' \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \quad \text{ou encore} \quad x' = A\,x.$$

Bien qu'on se restreigne dans ce cours au cas de deux variables, une bonne partie des raisonnements restent valables pour des systèmes en n variables. De tels systèmes linéaires apparaissent naturellement dans au moins deux cas : lorsqu'on linéarise un système non linéaire au voisinage

d'un point (fixe éventuellement cf. cours précédent) ; lorsqu'on étudie une équation différentielle linéaire d'ordre deux et à coefficients constants. L'équation suivante par exemple modélise un système masse–ressort :

$$m\,x_1'' + c\,x_1' + k\,x_1 = 0$$

où $x_1(t)$ représente la distance de la masse m à la position d'équilibre et c et k sont deux réels positifs (c est une constante de frottement et k la constante de raideur du ressort). En nommant $x_2(t) = x_1'(t)$ on transforme l'équation d'ordre deux en un système de deux équations d'ordre un :

$$\begin{pmatrix} x_1' \\ x_2' \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ -k/m & -c/m \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}.$$

On remarque que cette technique de renommage permet de transformer n'importe quelle équation différentielle d'ordre p (non nécessairement linéaire d'ailleurs) en un système de p équations différentielles d'ordre un.

Cas des matrices diagonales

Le cas où la matrice A est diagonale est particulièrement simple.

$$A = \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix}$$

La résolution du système $x' = A\,x$ c'est–à–dire du système

$$\begin{cases} x_1' = \lambda_1\,x_1, \\ x_2' = \lambda_2\,x_2 \end{cases}$$

est élémentaire. On trouve pour ensemble de solutions

$$\begin{pmatrix} x_1(t) \\ x_2(t) \end{pmatrix} = \begin{pmatrix} c_1\,e^{\lambda_1 t} \\ c_2\,e^{\lambda_2 t} \end{pmatrix} \quad \text{où} \quad c_1,\,c_2 \in \mathbb{R}.$$

Dans le cas général, la matrice A n'est bien sûr pas diagonale mais elle est parfois diagonalisable.

Cas des matrices diagonalisables

Supposons A diagonalisable. On a alors $x' = P\,J\,P^{-1}\,x$. Multiplions les deux membres de l'égalité, à gauche, par P^{-1}. On trouve $P^{-1}\,x' = J\,P^{-1}\,x$. Procédons à un changement de variable et posons $z = P^{-1}\,x$. Le système différentiel devient

$$z' = J\,z.$$

Ses solutions sont

$$\begin{pmatrix} z_1(t) \\ z_2(t) \end{pmatrix} = \begin{pmatrix} c_1\,e^{\lambda_1 t} \\ c_2\,e^{\lambda_2 t} \end{pmatrix} \quad \text{où} \quad c_1,\,c_2 \in \mathbb{R}.$$

Les solutions du système $x' = A\,x$ s'obtiennent en procédant au changement de variable inverse :

$$\begin{pmatrix} x_1(t) \\ x_2(t) \end{pmatrix} = P \begin{pmatrix} c_1\,e^{\lambda_1 t} \\ c_2\,e^{\lambda_2 t} \end{pmatrix} = c_1\,e^{\lambda_1 t}\,V_1 + c_2\,e^{\lambda_2 t}\,V_2 \quad \text{où} \quad c_1,\,c_2 \in \mathbb{R}.$$

Proposition 9 *Soit $x' = A\,x$ un système différentiel linéaire avec $A \in \mathbb{R}^{2\times 2}$. Si A est diagonalisable alors les solutions du système sont de la forme suivante, où V_1 est un vecteur propre de A de valeur propre λ_1, où V_2 est un vecteur propre de A de valeur propre λ_2 et où $c_1,\ c_2$ désignent deux constantes arbitraires :*

$$c_1\, e^{\lambda_1\, t}\, V_1 + c_2\, e^{\lambda_2\, t}\, V_2.$$

Vérifions la proposition à l'aide du logiciel MAPLE.

```
# Les deux équations
eq1 := diff (x1(t),t) - x1(t) - 2*x2(t);
                              /d       \
                       eq1 := |-- x1(t)| - x1(t) - 2 x2(t)
                              \dt      /

eq2 := diff (x2(t),t) - 3*x1(t) - 2*x2(t);
                             /d       \
                      eq2 := |-- x2(t)| - 3 x1(t) - 2 x2(t)
                             \dt      /

# La matrice des coefficients du système de deux équations
A := <<1 | 2>, <3 | 2>>;
                                    [1    2]
                                A := [      ]
                                    [3    2]

# Diagonalisation
J := JordanForm (A);
                                   [-1    0]
                               J := [       ]
                                   [ 0    4]
P := JordanForm (A, output='Q'):

# Les vecteurs x et x'
x := <x1(t), x2(t)>;
                                    [x1(t)]
                                x := [     ]
                                    [x2(t)]
xprime := map (diff,x,t);
                                     [d       ]
                                     [-- x1(t)]
                                     [dt      ]
                            xprime := [        ]
                                     [d       ]
                                     [-- x2(t)]
                                     [dt      ]



# Les vecteurs z et z' s'obtiennent par chgt de variable.
z := P^(-1) . x;
```

```
                              [x1(t) - 2/3 x2(t)]
                         z := [                  ]
                              [  x1(t) + x2(t)  ]

zprime := P^(-1) . xprime;
                                [/d       \       /d       \]
                                [|-- x1(t)| - 2/3 |-- x2(t)|]
                                [\dt      /       \dt      /]
                     zprime := [                            ]
                                [  /d       \   /d       \  ]
                                [  |-- x1(t)| + |-- x2(t)|  ]
                                [  \dt      /   \dt      /  ]

# Les solutions [z1(t), z2(t)] après chgt de variable sont évidentes

solz := < c1 * exp (-t), c2 * exp (4*t) >;
                                      [c1 exp(-t) ]
                               solz := [           ]
                                      [c2 exp(4 t)]

# Les solutions [x1(t), x2(t)] s'obtiennent par le chgt de variable inverse
solx := P. solz;
                            [3/5 c1 exp(-t) + 2/5 c2 exp(4 t) ]
                     solx := [                                 ]
                            [-3/5 c1 exp(-t) + 3/5 c2 exp(4 t)]

# Vérification du résultat
simplify (subs (x1(t)=solx[1], x2(t)=solx[2], [eq1, eq2]));
                                     [0, 0]
```

Cas des matrices non diagonalisables

Dans le cas où la matrice A n'est pas diagonalisable, les solutions sont aussi des couples de combinaisons linéaires d'exponentielles de la forme $e^{\lambda t}$ mais avec des polynômes en t pour coefficients. Cette partie sort du cadre du cours.

Valeurs propres non réelles

Les solutions d'un système différentiel linéaire $x' = A\,x$ sont des combinaisons linéaires d'exponentielles avec les valeurs propres de A en exposant. On ne considère que des matrices à coefficients réels mais même dans ce cas, les valeurs propres peuvent être des nombres complexes. Les solutions du système sont alors des fonctions à image complexe, ce qui est surprenant !

Explication. La résolution que nous avons menée ne prend pas en compte le fait que les conditions initiales qui nous intéressent sont réelles (les solutions complexes correspondent en fait à des conditions initiales complexes). Au prix de quelques calculs supplémentaires, on donne une autre expression des solutions, plus intuitive peut–être, où le nombre imaginaire i n'apparaît plus

explicitement. On a trouvé dans une section précédente :

$$\begin{pmatrix} x_1(t) \\ x_2(t) \end{pmatrix} = P \begin{pmatrix} c_1\, e^{\lambda_1 t} \\ c_2\, e^{\lambda_2 t} \end{pmatrix} = (V_1\ V_2) \begin{pmatrix} c_1\, e^{\lambda_1 t} \\ c_2\, e^{\lambda_2 t} \end{pmatrix}.$$

Les deux valeurs propres sont conjuguées puisque la matrice (donc son polynôme caractéristique) a des coefficients réels. Notons–les $\lambda = a \pm i\, b$. Les colonnes de la matrice P sont formées de deux vecteurs propres, qu'on peut prendre conjugués eux–aussi.

$$P = \begin{pmatrix} a_1 + i\, b_1 & a_1 - i\, b_1 \\ a_2 + i\, b_2 & a_2 - i\, b_2 \end{pmatrix}.$$

On se rappelle que $e^{(a \pm i\, b)t} = e^{at}\,(\cos(bt) \pm i \sin(bt))$. On procède à un changement de variable (inversible) sur les constantes :

$$c_1 = \frac{d_1 + i\, d_2}{2}, \quad c_2 = \frac{d_1 - i\, d_2}{2}.$$

On trouve

$$\begin{pmatrix} x_1(t) \\ x_2(t) \end{pmatrix} = e^{a\, t} \begin{pmatrix} (a_1\, d_1 - b_1\, d_2)\cos(b\, t) - (a_1\, d_2 + b_1\, d_1)\sin(b\, t) \\ (a_2\, d_1 - b_2\, d_2)\cos(b\, t) - (a_2\, d_2 + b_2\, d_1)\sin(b\, t) \end{pmatrix}.$$

Vérification en MAPLE.

```
P := <<a1+I*b1 | a1-I*b1>, <a2+I*b2 | a2-I*b2>>;
```

```
                          [a1 + I b1    a1 - I b1]
                      P := [                     ]
                          [a2 + I b2    a2 - I b2]
```

```
z := <c1*exp(a*t)*(cos(b*t)+I*sin(b*t)),
      c2*exp(a*t)*(cos(b*t)-I*sin(b*t)) >;
```

```
                        [c1 exp(a t) (cos(b t) + sin(b t) I)]
                    z := [                                  ]
                        [c2 exp(a t) (cos(b t) - sin(b t) I)]
```

```
x := P . z:
x := subs (c1=(d1+I*d2)/2,c2=(d1-I*d2)/2, x):
x := map (factor, x);
```

```
  x :=

    [-exp(a t)

    (-a1 d1 cos(b t) + a1 d2 sin(b t) + b1 d1 sin(b t) + b1 d2 cos(b t))]

    [-exp(a t)

    (-a2 d1 cos(b t) + a2 d2 sin(b t) + b2 d1 sin(b t) + b2 d2 cos(b t))]
```

5.3 Analyse qualitative des systèmes différentiels linéaires

On s'intéresse aux systèmes différentiels linéaires de la forme

$$\begin{cases} x_1' = a\,x_1 + b\,x_2 \\ x_2' = c\,x_1 + d\,x_2 \end{cases}$$

qu'on peut noter matriciellement

$$\begin{pmatrix} x_1' \\ x_2' \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \quad \text{ou encore} \quad x' = A\,x.$$

5.3.1 Point fixe du système

Les solutions constantes (ou points fixes) du système sont les solutions du système d'équations linéaires

$$0 = A\,x.$$

Dans le cas général (cas où $\det A \neq 0$) le point fixe est unique : c'est l'origine.

5.3.2 Vecteurs propres et trajectoires en ligne droite

Une courbe intégrale du système $x' = A\,x$ est une courbe dans un espace à trois dimensions $(t,\, x_1(t),\, x_2(t))$. Projetons cette courbe sur le plan $(x_1(t),\, x_2(t))$.

Proposition 10 *La projection de la trajectoire sur le plan de phase est une ligne droite si et seulement si le vecteur directeur de la droite est un vecteur propre de A.*

Pour s'en convaincre, on peut penser à la méthode d'Euler, appliquée au système $x' = A\,x$. À partir d'une condition initiale $x(0)$ et d'un pas de temps h, cette méthode construit une suite de de points définie par (chaque x_i est un vecteur de $\mathbb{R}^2$) :

$$x_0 = x(0), \quad x_{n+1} = x_n + A\,x_n\,h.$$

Dire que les points sont tous alignés, c'est dire que chaque vecteur x_{n+1} est un multiple du vecteur x_n et donc que tous ces vecteurs sont des vecteurs propres de A (ce sont tous des vecteurs associés à la même valeur propre). Voici un exemple avec MAPLE.

```
A := <<-1 | 1>, <0 | 1>>;
                                    [-1    1]
                                A := [       ]
                                    [ 0    1]
Eigenvectors (A);
                                  [-1]  [1    1/2]
                                  [  ], [       ]
                                  [ 1]  [0     1 ]
h := 1:
```

```
x [0] := <1, 2>;
                                        [1]
                                x[0] := [ ]
                                        [2]
for n from 0 to 2 do
    x[n+1] := x[n] + h * A . x[n]
od;
                                        [2]
                                x[1] := [ ]
                                        [4]

                                        [4]
                                x[2] := [ ]
                                        [8]

                                      [ 8]
                               x[3] := [  ]
                                      [16]
```

5.3.3 Analyse qualitative

On suppose que la matrice A est diagonalisable et que ses deux valeurs propres sont distinctes l'une de l'autre. Il y a plusieurs cas à considérer : le cas où les deux valeurs propres sont réelles (il y a alors une sous–discussion suivant leur signe) et le cas où elles sont complexes (il y a alors une sous–discussion suivant le signe de leur partie réelle).

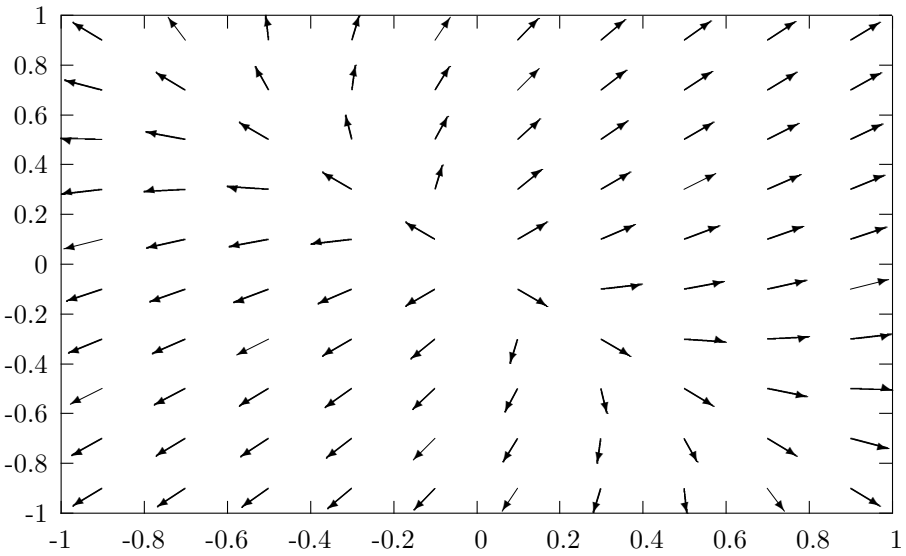
En effet, les valeurs propres λ_1 et λ_2 sont les racines du polynôme caractéristique $\det(A - \lambda\, I)$ de la matrice A. Comme A est de dimension 2×2, ce polynôme est de degré deux. Il s'écrit $a\,\lambda^2 + b\,\lambda + c$ pour certains $a,\, b,\, c \in \mathbb{R}$. Si son discriminant $\Delta = b^2 - 4\,a\,c$ est positif, les deux valeurs propres sont réelles. S'il est négatif, les deux valeurs propres sont des nombres complexes conjugués (et donc de même partie réelle). S'il est nul, la matrice n'a qu'une seule valeur propre réelle mais on ne considère pas ce cas–là ici.

Cas de deux valeurs propres réelles

Cas de deux valeurs propres positives. Quand $\lambda > 0$ on a

$$\lim_{t \to -\infty} c_i\, e^{\lambda_i\, t} = 0 \quad \text{et} \quad \lim_{t \to +\infty} c_i\, e^{\lambda_i\, t} = +\infty.$$

Les trajectoires partent du point fixe et vont vers l'infini. Quand t tend vers $+\infty$ elles deviennent de plus en plus semblables à des droites parallèles aux vecteurs propres associés à la plus grande des deux valeurs propres. On dit que le point fixe est une *source*. Voici un exemple type de champ de vecteurs :



Voici la matrice A qui a servi à tracer l'exemple ci–dessus avec ses deux valeurs propres et un vecteur propre chacune.

```
A := <<2 | 1>, <1 | 2>>;
```

```
                                    [2    1]
                                A := [      ]
                                    [1    2]
```

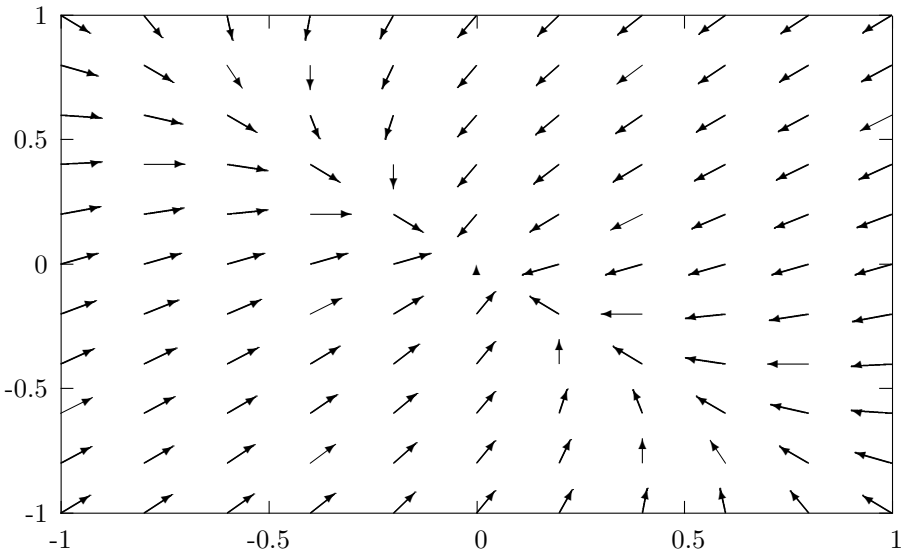
```
Eigenvectors (A);
```

```
                                  [3]  [1    -1]
                                  [ ], [       ]
                                  [1]  [1     1]
```

Cas de deux valeurs propres négatives. Quand $\lambda < 0$ on a

$$\lim_{t\to-\infty} c_i\, e^{\lambda_i\, t} = +\infty \quad \text{et} \quad \lim_{t\to+\infty} c_i\, e^{\lambda_i\, t} = 0.$$

Les trajectoires viennent de l'infini et convergent vers le point fixe. Quand t tend vers $-\infty$ elles ressemblent à des droites parallèles aux vecteurs propres associés à la valeur propre de plus grande valeur absolue. On dit que le point fixe est un *puits*. Voici un exemple type de champ de vecteurs :



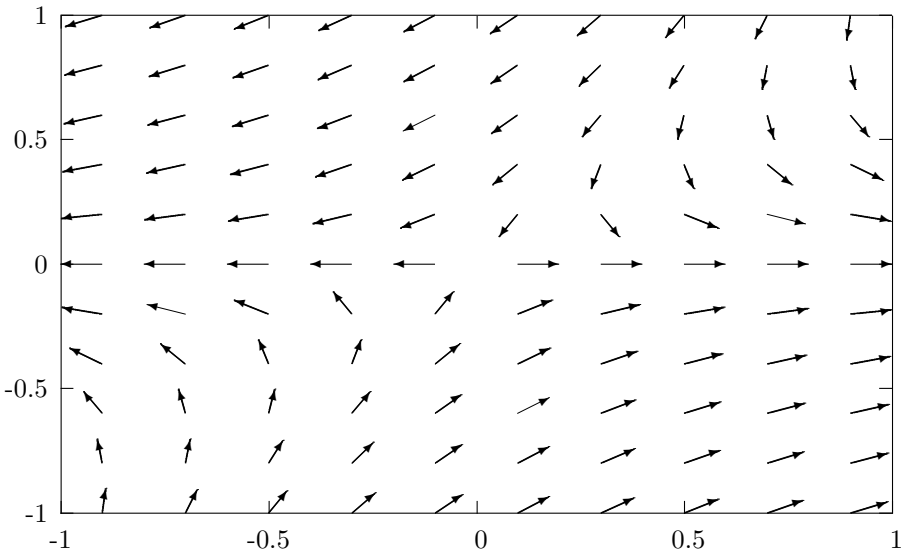
Voici la matrice A qui a servi à tracer l'exemple ci–dessus avec ses deux valeurs propres et un vecteur propre chacune.

```
A := <<-2 | -1>, <-1 | -2 >>: Eigenvectors (A);
                                [-3]  [1    -1]
                                [  ], [       ]
                                [-1]  [1     1]
```

Cas de valeurs propres de signes différents. Supposons $\lambda_1 < 0 < \lambda_2$ et considérons la formule qui donne les solutions du système :

$$\begin{pmatrix} x_1(t) \\ x_2(t) \end{pmatrix} = P \begin{pmatrix} c_1\, e^{\lambda_1 t} \\ c_2\, e^{\lambda_2 t} \end{pmatrix} \quad \text{où} \quad c_1,\, c_2 \in \mathbb{R}.$$

Dans le cas général (cas où $c_1 \neq 0$ et $c_2 \neq 0$) on voit que $e^{\lambda_1 t}$ domine quand t tend vers $-\infty$ (les trajectoires sont proches de droites parallèles aux vecteurs propres associés à λ_1) et que $e^{\lambda_2 t}$ domine quand t tend vers $+\infty$ (les trajectoires sont alors proches de droites parallèles aux vecteurs propres associés à λ_2). On dit que le point fixe est un *col*. Il y a exactement deux trajectoires particulières, l'une convergeant vers l'origine, l'autre partant de l'origine. Elles correspondent aux cas où l'un des coefficients c_i est nul. Voici un exemple type de champ de vecteurs :



Voici la matrice A qui a servi à tracer l'exemple ci–dessus avec ses deux valeurs propres et un vecteur propre chacune.

```
A := <<1 | -1>, <0 | -1>>: Eigenvectors (A);
                                [-1]  [1/2    1]
                                [  ], [        ]
                                [ 1]  [ 1     0]
```

Cas d'une valeur propre nulle. C'est un cas particulier qui sort du cadre du cours.

Cas de deux valeurs propres non réelles

Les valeurs propres λ_1 et λ_2 sont nécessairement des nombres complexes conjugués, c'est–à–dire des nombres complexes

$$\lambda_i = a \pm i\, b$$

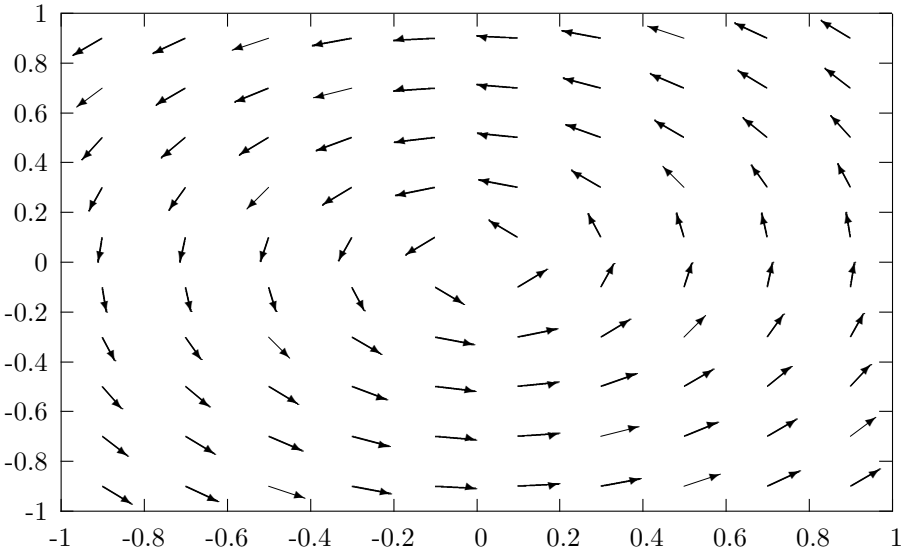
ayant même partie réelle a. Le cas d'une valeur propre réelle et l'autre non ne se pose donc pas pour les systèmes de deux variables.

On a vu dans une section précédente que les solutions du système différentiel peuvent toujours s'exprimer comme des combinaisons linéaires sur $\mathbb{R}$ des deux fonctions suivantes

$$s_1(t) = e^{a\,t} \begin{pmatrix} \alpha_1 \cos(b\,t) - \beta_1 \sin(b\,t) \\ \alpha_2 \cos(b\,t) - \beta_2 \sin(b\,t) \end{pmatrix} \quad \text{et} \quad s_2(t) = e^{a\,t} \begin{pmatrix} \alpha_1 \sin(b\,t) + \beta_1 \cos(b\,t) \\ \alpha_2 \sin(b\,t) + \beta_2 \cos(b\,t) \end{pmatrix}$$

où les réels α_i, β_i « viennent » des vecteurs propres de A.

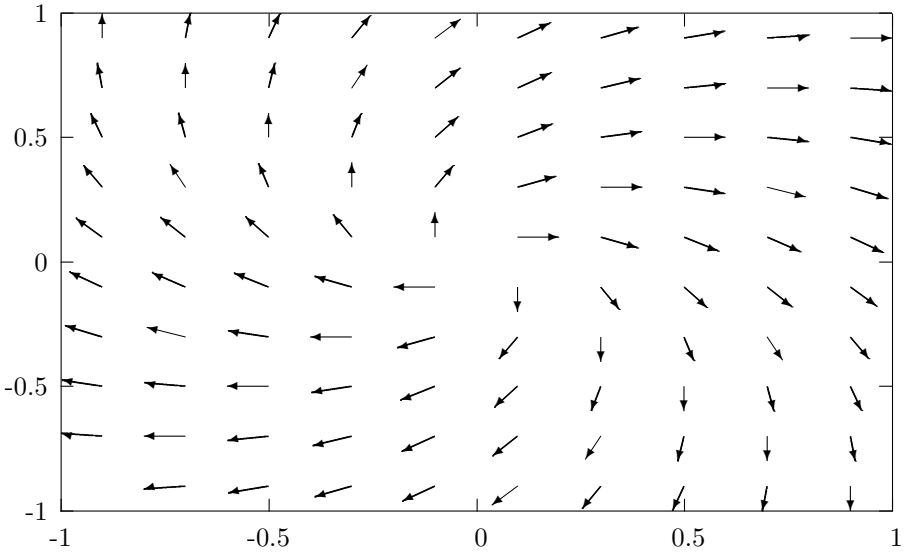
Cas d'une partie réelle nulle. Si $a = 0$ alors $e^{a\,t} = 1$ et les fonctions $s_1(t)$ et $s_2(t)$ sont périodiques de période $2\,\pi/b$. Les trajectoires des solutions se referment sur elles–mêmes. On dit que ce sont des « cycles ». Le point fixe est ce qu'on appelle un « centre ».



Voici la matrice A qui a servi à tracer l'exemple ci–dessus avec ses deux valeurs propres et un vecteur propre chacune.

```
A := <<0 | -1>, <1 | 0>>: Eigenvectors (A);
                                  [I ]  [I    -I]
                                  [  ], [       ]
                                  [-I]  [1    1 ]
```

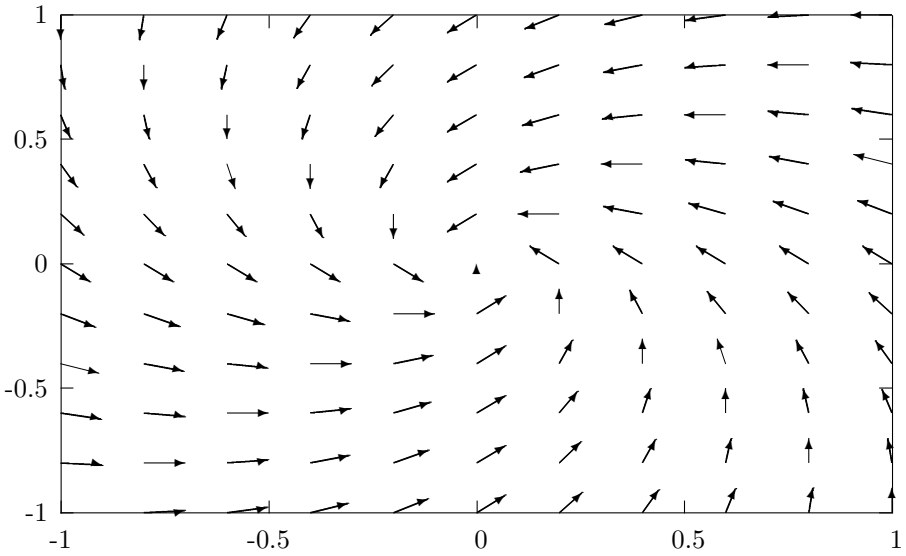
Cas d'une partie réelle non nulle. Suivant que a est positif ou non, le facteur $e^{a\,t}$ tend vers $+\infty$ ou vers zéro. Dans le premier cas, les trajectoires des solutions s'éloignent du point fixe en suivant une spirale. Le point fixe est appelé une « source spirale » (le terme « foyer » est parfois utilisé).



Voici la matrice A qui a servi à tracer l'exemple ci–dessus avec ses deux valeurs propres et un vecteur propre chacune.

```
A := <<1 | 1>, <-1 | 1>>: Eigenvectors (A);
                                [1 + I]  [-I    I]
                                [     ], [       ]
                                [1 - I]  [1     1]
```

Dans le deuxième cas, les trajectoires des solutions convergent vers le point fixe en suivant une spirale. Le point fixe est appelé un « puits spirale ».



Voici la matrice A qui a servi à tracer l'exemple ci–dessus avec ses deux valeurs propres et un vecteur propre chacune.

```
A := <<-1 | -1>, <1 | -1>>: Eigenvectors (A);
                                [-1 + I]  [I    -I]
                                [      ], [       ]
                                [-1 - I]  [1    1 ]
```

Index

Bibliographie

[1] Jacques Baranger. *Introduction à l'analyse numérique*. Hermann, Paris, 1977.

[2] Jean-Luc Chabert, Évelyne Barbin, Michel Guillemot, Anne Michel-Pajus, Jacques Borowczyk, Ahmed Djebbar, and Jean-Claude Martzloff. *Histoire d'algorithmes. Du caillou à la puce*. Belin, Paris, 1994.

[3] James H. Davenport, Yvon Siret, and Évelyne Tournier. *Calcul formel, Systèmes et algorithmes de manipulations algébriques*. études et recherches en informatique. Masson, Paris, 1987.

[4] Keith O. Geddes, Stephen R. Czapor, and George Labahn. *Algorithms for Computer Algebra*. Kluwer Academic Publishers, 1992.

[5] Ernst Hairer. Polycopié du cours "Analyse Numérique". Accessible sur `http://www.unige.ch/~hairer`, octobre 2001.

[6] Ernst Hairer, Syvert Paul Norsett, and Gerhard Wanner. *Solving ordinary differential equations I. Nonstiff problems*, volume 8 of *Springer Series in Computational Mathematics*. Springer–Verlag, New York, 2nd edition, 1993.

[7] John J. O'Connor and Edmund F. Robertson. The MacTutor History of Mathematics archive. `http://www-history.mcs.st-andrews.ac.uk/history`, 2006.

[8] Joachim von zur Gathen and Jürgen Gerhard. *Modern Computer Algebra*. Cambridge University Press, United Kingdom, 1999.