Introduction à Maple

MPSI 2 Lycée Fermat

A. Soyeur

2000-2001

Table des matières

Chapitre 1

Introduction, affectation

1.1 Calcul numérique

Après avoir lancé le programme MAPLE (version 4 ou 5), on se trouve devant un « prompt » (signe `>`). On entre une commande terminée par un point virgule. Lorsque l'on appuie sur la touche Entrée, MAPLE évalue notre commande et affiche un résultat :

Maple
```
> 2 + 2;
    4
> 2 ^ 3;
    8
> 100!;
```

Si l'on remplace le point virgule à la fin de la commande par deux points (`:`), MAPLE évalue la commande, mais n'affiche pas de résultat. C'est utile pour ne pas encombrer l'écran de résultats intermédiaires.
Voici quelques fonctions mathématiques que nous utiliserons cette année :

- `sin`, `cos`, `tan`, `arcsin`, `arccos`, `arctan` : fonctions trigonométriques ;
- `sinh`, `cosh`, `tanh` : fonctions de trigonométrie hyperbolique ;
- `exp`, `ln` exponentielle et logarithme népérien. `log` : logarithme en base 10, `log[a]` : logarithme en base a.
- `sqrt` : racine carrée, `abs` : valeur absolue.
- `irem` : le reste de la division euclidienne d'un entier par un autre.
- `gcd` : le pgcd de deux entiers.
- `floor`, `ceil` : la partie entière inférieure et supérieure d'un réel.

Une fonction s'applique à un (ou plusieurs) *arguments* placés entre parenthèses :

Maple
```
> sin( exp(1) * log[2](5) );
```

Il existe quelques constantes prédéfinies (certaines commencent par une majuscule) :

- `Pi` pour le nombre π ;
- `I` pour le complexe i ($i^2 = -1$) ;
- `infinity` pour représenter l'infini ∞, qui peut être précédé d'un signe : `+infinity` ou `-infinity` ;
- Le nombre de Neper E n'est pas défini. Il faut utiliser `exp(1)`.

Par défaut, Maple effectue des calculs formels. Si l'on souhaite une valeur numérique approchée d'un résultat, on utilise la fonction `evalf` (f comme *float*). On peut préciser le nombre de chiffres significatifs que l'on désire grâce à un argument supplémentaire de cette fonction :

Maple
```
> evalf( sin(Pi / 5) );
> evalf( Pi, 200 );
```

Exercice 1. *Déterminez une valeur approchée à 10^{-10} près du nombre e.*

Exercice 2. *Déterminez une valeur approchée à 10^{-3} près du nombre complexe $(1+\pi i)^4$.*

Maple garde en mémoire l'historique des calculs effectués. Le symbole `"` pour la version 4 de Maple ou `%` pour la version 5, permet d'accéder au dernier calcul effectué. Attention : il s'agit du dernier calcul effectué dans l'historique et non pas du dernier calcul affiché dans la feuille de calculs :

Maple
```
> sin(Pi / 5) * 3;
> exp( % );
> evalf( % );
```

Le symbole `%%` (ou `""`) permet d'accéder à l'avant dernier résultat calculé.

Exercice 3. *Déterminez avec 20 chiffres significatifs, une valeur numérique approchée de la quantité*

$$\sqrt{\left|\ln\left(\sin(\pi/7)\right)\right|}$$

Pour cela, décomposez les calculs, et utilisez le symbole %.

1.2 Identificateurs en Maple, affectation de variables

Maple manipule des nombres, mais également des identificateurs. Un identificateur est une suite de caractères, pris parmi `A-Z`, `a-z`, `0-9` et le caractère souligné, `_`. Le premier caractère doit être une lettre de l'alphabet. Voici quelques identificateurs valides : `x`, `Mon_identificateur`, `variable123`, `Ensemble_vide`. Remarquez qu'un identificateur correspondant à une lettre grecque est affiché avec le symbole correspondant :

Maple
```
> alpha + beta - gamma + theta - mu + phi;
```

donne à l'affichage $\alpha+\beta-\gamma+\theta-\mu+\phi$. Les identificateurs servent comme noms d'objets formels, dans des « expressions algébriques » :

Maple
```
expand( (x + y) ^ 3 );
```

Nous reviendrons plus tard sur la signification d'une expression algébrique.

Exercice 4. *Expliquez le résultat de*

Maple
```
> evalf(pi);
```

Un identificateur peut servir comme une « variable », que l'on peut voir comme une boîte qui peut contenir une expression. On *affecte* une expression à une variable à l'aide de l'opérateur d'affectation `:=` , à ne pas confondre avec une égalité mathématique. Cet opérateur, dans la description d'algorithmes est noté `<-` :

Maple
```
> A := x + 2;
> B := x + y;
> mon_expression := A + B;
```

Dans la variable de nom `A`, on a stocké l'expression $x+2$, dans la variable de nom `B`, on a stocké l'expression $x+y$, et dans la variable de nom `mon_expression`, on a stocké l'expression $A+B$. Il est indispensable de comprendre le mécanisme d'évaluation et d'affectation :

- Pour évaluer une expression :
  1. Maple remplace toutes les variables affectées d'une expression par leur valeur.

2. Comme les valeurs des variables affectées peuvent elles-mêmes contenir des variables affectées, Maple réitère les substitutions jusqu'à ce que l'expression ne contienne plus de variables affectées.
3. Maple utilise ses règles de simplification sur les objets terminaux (nombres et variables non-affectées), et affiche le résultat.

– Pour affecter une expression à une variable : `G := D`
  1. Maple commence par évaluer le membre droit, l'expression `D`
  2. Il affecte ensuite cette valeur au membre gauche, la variable `G`.

Le mécanisme d'affectation se passe donc en deux temps, ce qui explique la construction suivante que nous utiliserons beaucoup en programmation :

Maple
```
> A := 0;
> A := A + 1;
> A := A + 1;
```

À la première ligne, on affecte la valeur 0 à la variable A. À la deuxième ligne, Maple commence par évaluer le membre droit à la valeur 1, puis affecte cette nouvelle valeur à la variable A. Comme A contient maintenant la valeur 1, à la troisième ligne, le résultat de l'évaluation du membre droit est 2, et c'est cette valeur que contient maintenant la variable A. Considérez l'exemple suivant :

Maple
```
> y := 2 * x;    # y contient 2*x
> x := 1;        # x contient 1
> y;             # y contient 2
> z := x ^ 2;    # z contient 1
> x := 2;        # x contient 2
> y;             # y a changé et contient 4
> z;             # z n'a pas changé et contient toujours 1
```

Au départ, la variable y est définie à partir d'une variable non-affectée x, alors que la variable z est définie à partir d'un membre droit qui a une valeur numérique (il n'y a plus de variable `x` à l'affectation de `z`). Ceci explique le comportement précédent lors du changement de la valeur de x.

Remarque 1. On peut affecter une variable V avec une expression qui fait intervenir cette même variable, à condition qu'elle aît été préalablement affectée :

Maple
```
> unassign('V');
> V := V + 1;  # impossible
> V := 0;
> V := V + 1;  # correct
```

On veut échanger le contenu de deux variables A et B. La séquence d' instructions suivante est incorrecte :

Maple
```
> B := A;
> A := B;
```

En effet, supposons que la variable A contienne une expression x, et la variable B une expression y. Voici les contenus des variables après les deux instructions :

instruction	A	B
	x	y
`B := A`	x	x
`A := B`	x	x

Il faut utiliser une variable supplémentaire, notée traditionnellement `temp` (comme temporaire) :

Maple
```
temp := A;
A := B;
B := temp;
```

instruction	A	B	temp
	x	y	
`temp := A`	x	y	x
`A := B`	y	y	x
`B := temp`	y	x	x

`Exercice 5.` *Sans utiliser Maple, trouvez le contenu des variables A et B après les instructions suivantes, en supposant qu'initialement A et B contiennent des nombres x et y :*

1. `A := A + B`
2. `B := A - B`
3. `A := A - B`

Une fois affectée, une variable ne peut plus servir comme nom dans une expression algébrique. Il est souvent utile de désaffecter une ou toutes les variables.

```
Maple
restart;
```

désaffecte toutes les variables affectées. On utilise cette commande à la fin d'une session de travail avant de commencer une nouvelle session indépendante.

```
Maple
x := 'x';
```

En affectant la variable x avec son nom (entre apostrophes sous la touche 4), on désaffecte la variable x, et elle peut à nouveau servir comme nom dans une expression formelle.
Pour désaffecter simultanément plusieurs variables, utiliser la fonction `unassign` :

```
Maple
> a := 1; b := 2;
> unassign('a', 'b');
```

Pour définir des expressions compliquées, ou réutiliser des parties d'expressions, on utilise des variables. Par exemple, pour définir l'expression suivante :

$$\frac{\sqrt{\ln(x+1) * \sin(x+y)}}{\ln(\sqrt{x^2+1}+2)}$$

on peut procéder ainsi :

```
Maple
> num := ln(x + 1) * sin( x + y);
> nom := sqrt(num);
> denom := sqrt(x^2 + 1) + 2;
> denom := ln(denom);
> expression := num / denom;
```

Le principal intérêt est de contrôler au fur et à mesure les sous-expressions intermédiaires, et de minimiser les fautes de frappe.
La constante système `Digits` (remarquez la majuscule) détermine le nombre de chiffres significatifs exacts dans les calculs effectués par Maple. Si l'on veut effectuer tous les calculs avec 100 chiffres significatifs, il suffit d'affecter cette variable :

```
Maple
> Digits := 100;
> evalf( sin( Pi / 6) );
```

1.3 Packages

Pour des raisons d'économie et d'efficacité, au lancement de Maple, seules les fonctions les plus utilisées sont automatiquement chargées en mémoire. Si vous désirez utiliser une fonction plus « exotique », il faut d'abord charger en mémoire le fichier qui la définit :

```
Maple
> readlib(randpoly);
```

Maple est également organisé en « bibliothèques » (packages en Anglais). Chaque bibliothèque regroupe plusieurs fonctions spécialisées dans un domaine précis. Par exemple, le package `plots` contient de nombreuses fonctions destinées aux graphiques 2D et 3D. Pour charger toutes les fonctions d'un package, utiliser la fonction `with` :

```
Maple
> with(plots); # charge les fonctions du package plots
```

`Exercice 6.` *Charger toutes les fonctions de la bibliothèque* student *(regroupant des fonctions utiles aux étudiants).*

1.4 Un mot sur la présentation

Pour rendre les entrées Maple plus lisibles, nous adopterons les conventions suivantes :

- Tout ce qui suit le symbole # est ignoré par Maple, ce qui permet d'introduire des commentaires (surtout utile en programmation) ;
- Chaque opérateur (+, ∗, /, :=, ...) est entouré d'espace :

```
Maple
> 2+4; # mauvais
> 2 + 4; # bon
> A:=x^2; # mauvais
> A := x ^ 2 # bon
```

- Chaque *séparateur*, (`,`, `;`), est suivi d'un espace :

```
Maple
> f(2,3); # mauvais
> f(2, 3); # bon
> 2+2 ;3+3; # mauvais
> 2 + 2; 3 + 3; # bon
```

L'opérateur point `.`, permet de concaténer deux noms de variables :

```
Maple
> ma.variable;
```

C'est utile principalement pour définir un grand nombre de variables à l'aide d'une boucle (nous verrons les boucles plus tard) :

```
Maple
> for i from 1 to 10 do f.i := i od;
> f4;
```

Chapitre 2

Calcul formel

2.1 Expressions

Avant de faire du calcul formel, voyons comment Maple travaille avec les « expressions algébriques ». Une expression algébrique est définie à partir de *symboles terminaux* (des nombres et des variables non-affectées) ainsi que des opérateurs comme `+`, `*` ... et des fonctions comme `sin`, `exp` ...
Toute expression algébrique peut se représenter par un *arbre syntaxique*. Par exemple, l'expression

$$A := (1+2) * \Big(x - \sin\big(\sqrt{y}\big)\Big) * \Big(2 * x + \sin(y)\Big)$$

se représente par l'arbre suivant :

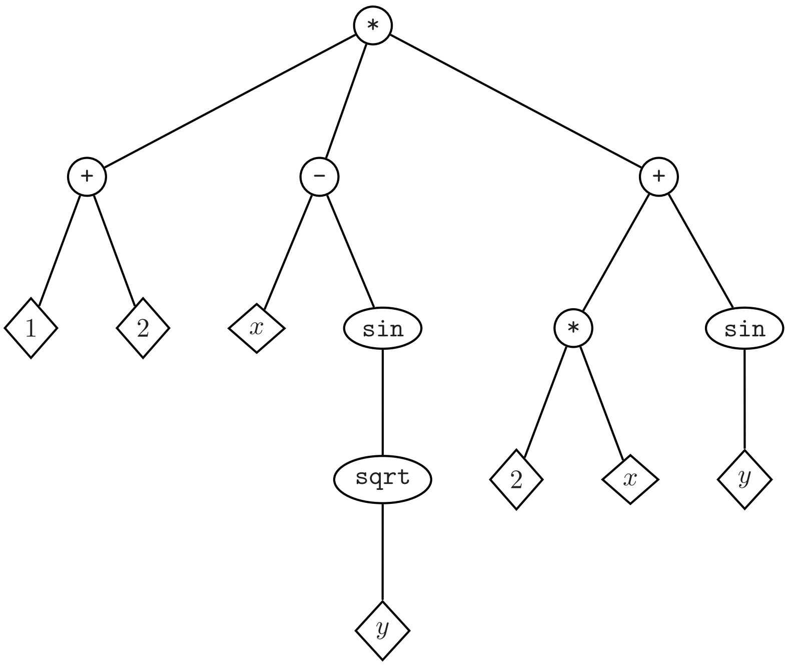

FIG. 2.1 – *Un arbre de syntaxe*

Les symboles terminaux (nombres et variables non-affectées) sont représentées aux « feuilles » de l'arbre (dessinées sur la figure avec un losange), et les opérateurs et fonctions sont représentés aux « noeuds » de l'arbre (dessinés avec des ronds). Chaque expression algébrique possède un *type* que l'on peut obtenir à l'aide de la commande `whattype` :

Maple
```
> whattype(A);
```

Le type retourné correspond à la racine de l'arbre. L'opérateur op permet de sélectionner une sous-expression d'une expression algébrique. Par exemple,

Maple
```
> B := op(2, A);
```

retourne l'expression algébrique correspondant au deuxième sous-arbre de l'expression A. Sur notre exemple, $B = x - \sin(\sqrt{y})$.

Maple
```
> C := op(2, B);
> D := op(C);
```

En prenant le deuxième sous-arbre de l'expression B, on trouve $C = \sin(\sqrt{y})$. Comme l'expression C se représente par un arbre avec un seul sous arbre, on utilise l'opérateur op à un seul argument pour récupérer l'expression $\sqrt{y}$.

Exercice 7. *Définir (en procédant en plusieurs étapes) l'expression algébrique*

$$\frac{\sin(x+y) - \cos(\sqrt{y})}{\exp(x^{y^z})}$$

Dessiner son arbre syntaxique, et utiliser sous Maple l'opérateur op *pour récupérer la sous-expression* y^z.

Il est possible dans une expression algébrique de substituer une sous-expression par une autre en utilisant la fonction subs :

subs(sous_expr = remplacement, expr)

Elle substitue chaque occurrence de la sous-expression sous_expression par l'expression remplacement dans l'expression expr.

Maple
```
> A := x^2 + sin(exp(y) + exp(z));
> subs(exp(y) + exp(z) = sqrt(x), A);
```

Remarquez le symbole = qui désigne ici une égalité mathématique, à ne pas confondre avec le symbole d'affectation :=. Supposons que l'on manipule l'expression algébrique

Maple
```
> A := x^2 + 1;
```

On veut savoir la valeur de cette expression lorsque x vaut 3 :

Maple
```
> subs(x = 3, A);
```

L'utilisation de subs est plus propre que d'affecter x, puis de le désaffecter ensuite :

Maple
```
> x := 3;
> A;
> x := 'x';
```

2.2 L'aide en ligne

Il n'est pas nécessaire de connaître beaucoup de fonctions pour travailler avec Maple, par contre il est important de savoir utiliser l'aide en ligne pour examiner toutes les possibilités d'une fonction, ou pour trouver sans aide extérieure la fonction qui permet de résoudre un problème donné.
Pour obtenir l'aide sur une fonction dont on connaît le nom, il suffit de faire précéder le nom de cette fonction par un point d'interrogation. Une fenêtre d'aide sur la fonction apparaît alors.

Maple
```
> ?expand
```

La fenêtre d'aide comprend plusieurs sections :

1. **Calling sequence :** la description formelle de la syntaxe de la fonction ;
2. **Parameters :** ce que représentent les paramètres de la fonction ;
3. **Description :** une description détaillée de ce que peut accomplir la fonction ;
4. **Examples :** une série d'exemples d'utilisation de cette fonction ;
5. **See also :** les fonctions ayant un lien avec cette page d'aide.

L'autre moyen d'obtenir de l'aide en partant de zéro est d'utiliser le menu help, et de visiter les fonctions classées par catégories.

`Exercice 8.` *Quelles sont les fonctions qui manipulent les nombres complexes ? Déterminez le module et l'argument du complexe* $z = 1 + \sqrt{3}i$.

`Exercice 9.` *L'entier* 28037 *est-il premier ? Si ce n'est pas le cas, trouvez à l'aide de Maple sa décomposition en facteurs premiers.*

2.3 Un peu de calcul formel

Nous verrons dans cette section les commandes les plus intéressantes pour effectuer du calcul formel avec Maple.

2.3.1 Transformations d'expressions

- `simplify( expr )` : c'est un peu une fonction fourre-tout, qui simplifie (en un sens trop compliqué à définir précisément) des expressions. Voici quelques exemples :

  Maple
  ```
  > simplify( (x + y) ^ 2 - x ^ 2 - y ^ 2 );
  ```

  Maple
  ```
  > (sin(x)) ^ 4 - (cos(x)) ^ 4;
  > simplify(%);
  ```

  Maple
  ```
  > simplify( sqrt(x ^ 2) );
  ```

  Maple simplifie cette dernière expression en `csgn(x) x`. La fonction `csgn(x)` dépend du signe de x et si x est réel ou complexe … Pour simplifier cette expression si vous savez que x est un réel positif, il faut faire part à Maple de cette information :

  Maple
  ```
  > assume(x > 0);
  > simplify( sqrt(x ^ 2) );
  ```

  La fonction `assume` permet de faire des hypothèses sur les variables. Notez l'apparition d'un tilde devant la variable x, qui signifie qu'une assomption a été faite sur cette variable. On peut utiliser `about` pour voir l'hypothèse faite sur une variable :

  Maple
  ```
  > about(x);
  ```

  Pour enlever les hypothèses sur les variables, le plus simple est de désaffecter la variable :

  Maple
  ```
  > unassign('x');
  ```

- `expand( expr )` : développe une expression :

  Maple
  ```
  > A := (x + y) ^ 3;
  > expand(A);
  ```

Maple
```
> cos(a + b);
> expand(%);
```

– combine(expr, option) : c'est en gros l'inverse d' expand. L'option peut être trig, exp, ln, power.

Maple
```
> combine(sin(p) * sin(q));
```

Maple
```
> combine(exp(x) * exp(y));
```

Maple
```
> A :=  ln(x) + ln(y) ;
> combine(A);
```

Maple refuse de combiner cette dernière entrée, car x et y ne sont pas à priori des réels positifs (le logarithme Maple est défini sur les nombres complexes ...). Il faut faire une hypothèse sur les variables :

Maple
```
> assume(x > 0, y > 0);
> combine(A);
> unassign('x', 'y'); # enlève les hypothèses par la suite
```

– normal(expr) : Réduit au même dénominateur, puis simplifie une fraction rationnelle.

Maple
```
> num := x ^ 2 + 2 * x + 1;
> den := x^2 + 3 * x + 2;
> f := num / den;
> normal(f);
```

– factor(expr) : essaie de factoriser une expression polynômiale ou rationnelle, en conservant des coefficients rationnels. Le problème de la factorisation est beaucoup plus compliqué que le développement, et Maple n'y arrive pas toujours :

Maple
```
> A := x^3 - x^2 - 4;
> factor(A);
```

Maple
```
> factor(x ^ 2 - 2);
```

Cette expression est factorisable, mais pas avec des coefficients rationnels ! Pour la factoriser quand même, il faut utiliser la fonction split (qui n'est pas chargée au démarrage) :

Maple
```
> readlib(split);  # charge la fonction split
> split(A, x);
```

Vous obtiendrez souvent des résultats faisant intervenir Root_Of, qui désigne l'ensemble des racines d'un polynôme. Pour remplacer explicitement ces racines (ce qui n'est pas toujours possible), il faut convertir le résultat :

Maple
```
> convert(%, radical);
```

– convert(expr, option) : convertit une expression d'une forme à une autre.
  – Pour décomposer des fractions rationnelles en éléments simples (voir le cours d'intégration cette année) :

Maple
```
> F := (x ^ 4 + 1) / (x^ 2 + 1) ^ 2;
> convert(F, parfrac, x) # décomposition en éléments simples dans R
> convert(F, parfrac, x, complex) # dans C
```

  - Pour exprimer les fonctions usuelles en fonction d'exponentielles et logarithmes :

```
Maple
> A := cos(x);
> convert(A, expln);
```

  - Voir l'aide sur convert pour d'autres possibilités.

- `collect(expr, expr_2)` : exprime l'expression expr par rapport aux puissances de expr_2 :

```
Maple
> P := exp(x + y) * (x + y) ^ 2 - exp(2 * x) * (x + y) ^ 3;
> collect(P, x);
> collect(P, y);
> collect(P, exp(x));
```

  Les résultats ne sont pas triés. Pour cela, on peut utiliser la fonction sort qui réordonne les monômes d'un polynôme :

```
Maple
> P := y^4 * x^2 + y^2 * x^4 - x + x * y;
> sort(P, x);
> sort(P, y);
```

Exercice 10. *Considérons l'expression*

$$P = (x^2y - xy^2e^x)^3$$

En utilisant les fonctions de manipulation d'expressions, obtenir la forme suivante de P (ordonnée selon les puissances de e^x) :

$$P = -y^6x^3(e^x)^3 + 3y^5x^4(e^x)^2 - 3y^4x^5e^x + y^3x^6$$

Exercice 11. *Considérons la fraction rationnelle :*

$$F = \frac{(x^2 + x - 1)^2}{(x + 1)^3}$$

1. *Décomposer cette fraction en éléments simples dans $\mathbb{R}$, et affecter une variable A avec le résultat.*
2. *En utilisant les fonctions de manipulation, transformer A pour retrouver F.*

Exercice 12. *Factoriser le polynôme*

$$P = x^3 - 2x$$

Exercice 13. *Transformer l'expression $\sin(x)^3$ en une expression en $\sin x$ et $\sin(3x)$.*

Exercice 14. *Transformer l'expression $\sin(3x)$ en un polynôme en $\sin x$*

2.3.2 Opérations mathématiques

- `diff(expr, var)` : permet d'obtenir l'expression dérivée par rapport à la variable var :

```
Maple
> A := x^2 + y * x^3;
> diff(A, x);
> diff(A, y);
```

On peut dériver plusieurs fois :

```
Maple
> B = y * sin(x^2);
> diff(B, x);
> diff(B, y, x);
> diff(B, x$5);
```

Le signe dollar permet de répéter plusieurs fois la commande.

- `int(expr, var)` : permet de trouver l'expression primitive de expr, par rapport à la variable var :

```
Maple
> A := x^2 + y * x;
> int(A, x);
```

Pour calculer une intégrale définie, utiliser à la place de var un intervalle (caractérisé par deux points) :

```
Maple
> A := sin(t);
> int(A, t = 0 .. Pi);
```

Il est également possible de définir une primitive sans la calculer explicitement (pour effectuer des manipulations formelles par la suite) en utilisant la *forme inerte* de la fonction `int`. Une forme inerte commence par une majuscule. Pour évaluer ensuite cette forme inerte, utiliser `value` , ,

```
Maple
> F := int( sin(t), t);
> value(F);
```

- `solve(equation, var)` : résout (lorsque c'est possible) une équation :

```
Maple
> eq := x^2 + y = 0; # une équation est stockée dans la variable eq
> solve(eq, x);
> solve(eq, y);
```

On peut également résoudre des inéquations :

```
Maple
> solve(x^2 + x > 3, x);
```

Lorsque les équations à résoudre ne sont pas polynômiales, il faut se méfier des résultats obtenus avec solve : on peut ne pas obtenir de solution, n'obtenir qu'une solution parmi plusieurs, ou même obtenir des solutions fausses !

Pour obtenir une résolution numérique approchée d'une équation, utiliser `fsolve` :

```
Maple
> fsolve( x * exp(x) = 1, x);
```

- `limit(expr, var = a)` : détermine la limite (lorsqu'elle existe) de la fonction correspondant à l'expression lorsque la variable tend vers a :

```
Maple
> A := sin(x) / x;
> limit(A, x = 0);
```

On peut demander des limites à gauche et à droite d'un point :

```
Maple
> limit(1 / t, t = 0);
> limit(1 / t, t = 0, right);
> limit(1 / t, t = 0, left);
```

- `series(expr, var = a, ordre)` : détermine un développement limité de la fonction correspondant à l'expression au voisinage du point a à l'ordre donné. Voir le cours de mathématique pour la définition d'un développement limité.

```
Maple
> A := sin(x);
> B := series(A, x = 0, 6);
```

On peut récupérer l'expression correspondant à la partie régulière (le polynôme sans le reste en $O(x^n)$) grâce à la fonction convert :

```
Maple
> convert(B, polynom);
```

- plot(expr, var = intervalle) : permet de tracer la courbe représentative de la fonction correspondant à l'expression sur l'intervalle spécifié :

```
Maple
> A := sin(t);
> plot(A, t = 0 .. 2 * Pi);
```

- sum(expr, ind = intervalle) : calcule une somme. expr est une expression qui dépend d'un indice ind :

```
Maple
> sum( 1 / i, i = 1 .. 100);
> sum(i ^ 2, i = 1 .. n);
> factor(%);
> sum( 1 / n ^ 2, n = 1 .. infinity);
```

Si le nom utilisé comme indice est une variable déjà affectée, on peut l'entourer d'apostrophes :

```
Maple
> i := 10;
> sum( 'i' ^ 2, 'i' = 1 .. n);
> unassign('i');
```

Il est possible également de définir une somme sans la calculer en utilisant la *forme inerte* de la fonction sum (comme pour int) :

```
Maple
>  Sum( i ^ 3, i = 1 .. 100);
> value(%);
> factor(%);
```

De même, on définit des produits finis ou infinis :

```
Maple
> product( 1 - 1 / 'i', 'i' = 1 .. 10);
```

Exercice 15. *On se propose d'étudier la fonction définie par*

$$f(x) = (x^2 - 1)e^x$$

1. *Désaffecter toutes les variables ;*
2. *Stocker dans une variable f l'expression $(x^2 - 1)e^x$;*
3. *Dans une variable df, stocker l'expression correspondant à la dérivée $f'(x)$;*
4. *Déterminer les solutions des équations $f(x) = 0$ puis $f'(x) = 0$. Résoudre les inéquations $f'(x) < 0$ et $f'(x) > 0$;*
5. *Déterminer la limite $\lim_{x \to +\infty} f(x)$;*
6. *Déterminer les valeurs $f(1/2)$, $f'(1/2)$ et donner des valeurs numériques approchées de ces nombres ;*
7. *Tracer sur une feuille le tableau de variations de la fonction f en obtenant toutes les données intéressantes (maximas, minimas, limites ...) avec Maple ;*
8. *Tracer le graphe de f entre 0 et 1 ;*
9. *Déterminer une primitive de la fonction f, que l'on stockera dans une variable F ;*
10. *Vérifier que $F' - f = 0$ (développer l'expression obtenue) ;*
11. *Déterminer un développement limité à l'ordre 3 au voisinage de 0 de la fonction f.*

2.4 Fonctions

En mathématiques, une fonction est la donnée d'un ensemble de départ, d'un ensemble d'arrivée et d'une *correspondance* qui à chaque élément de l'ensemble de départ associe un unique élément de l'ensemble d'arrivée. En informatique, une fonction est la donnée d'un *type* de départ (int, float ...), d'un type d'arrivée et d'une correspondance. Pour définir une fonction en Maple, il suffit de définir la correspondance grâce à l'opérateur `->` :

```
Maple
> f =: x -> x ^ 2;
```

Le membre droit de l'affectation est une fonction, qui à une variable var associe l'expression en cette variable. Comme en mathématique, l'objet fonction est indépendant du nom x utilisé pour le définir : on dit que la variable x est muette. On aurait pu définir la *même* fonction de la façon suivante :

```
Maple
> f := t -> t ^ 2;
```

Cet objet de type fonction est ensuite stocké dans la variable f. L'intérêt d'une fonction est que l'on peut *l'appliquer* à un nombre ou un nom de variable, pour obtenir un nombre ou une expression :

```
Maple
> f(3);
> f(y);
```

Remarque 2. Une faute courante de débutant consiste à confondre fonction et expression :

```
Maple
> f(x) := x ^ 2;
> f(x);
> f(2);
```

À la première ligne, `x^2` est une *expression*, que l'on affecte dans la variable de nom `f(x)` (les parenthèses font partie du nom de la variable). À la deuxième ligne, Maple affiche le contenu de la variable `f(x)`, c'est à dire l'expression `x^2`. À la troisième ligne, Maple affiche le contenu de la variable de nom `f(2)`, qui n'est pas affectée ! À COMPRENDRE ABSOLUMENT !!

`Exercice 16.` *1. Regarder la section* synopsis *de l'aide sur la fonction* `diff` *;*

2. Expliquer les résultats renvoyés par Maple avec :

```
Maple
> f := x -> x ^ 2;
> g := x ^ 2;
> diff(g, x);
> diff(f(x), x);
> diff(f, x);
> diff(g(x), x);
```

`Exercice 17.` *La fonction* `D(fonction)` *renvoie une fonction : la fonction dérivée de* `fonction`*:*

```
Maple
> f := x -> x ^ 3;
> g := D(f);
> g(t);
```

La variable g est de type fonction, et cette fonction est la fonction $t- > 3t^2$: la dérivée de la fonction f.

1. Voir le synopsis de la page d'aide de la fonction `D` *;*

2. *Expliquer les résultats renvoyés par Maple :*

```
Maple
> f := x -> x ^ 2;
> g := x ^ 2;
> df := D(f);
> dg := D(g);
> f(t);
> g(t);
```

À partir d'une fonction f, on obtient une expression en une variable x en *appliquant* la fonction f au nom x :

```
Maple
> f := x -> x ^ 2;  # la fonction f
> f(x);  # l'expression f(x)
```

Réciproquement, à partir d'une expression en la variable x, on peut définir une fonction d'une variable grâce à la fonction `unapply(expr, var)` :

```
Maple
> expr := x ^ 2;
> f := unapply(expr, x);
> f(t);
```

Il est préférable (pour des raisons compliquées d'évaluation) d'utiliser `unapply` plutôt que d'utiliser la forme suivante :

```
Maple
> expr := x ^ 2;
> f := x -> expr;
```

Pour composer des fonctions d'une variable, on utilise l'opérateur `@` :

```
Maple
> f := x - > x ^ 2;
> g := x -> sin(x);
> f_g := f @ g; # f_g contient la fonction fog
> g_f := g @ f;
> f_g(t);
> g_f(y);
```

On peut également définir des fonctions de plusieurs variables :

```
Maple
> f := (x, y) -> x * sin(y);
> f(1, Pi / 2);
> f(a, b ^ 2);
> g := unapply(x * y, (x, y));
> g(a, b);
```

Exercice 18. *On considère la fonction de deux variables définie par :*

$$f(\lambda,x) = \lambda x(1-x)$$

1. *Définir en Maple cette fonction ;*
2. *Tracer le graphe de $f(1,x)$ et $f(4,x)$ pour $x \in [0,1]$;*
3. *Dans une variable f_1, stocker la* fonction $x->f(1,x)$ *(utiliser* `unapply`*) ;*
4. *Déterminer l'*expression $f_1^{(2)}(t) = f \circ f(t)$*. C'est un polynôme en t que l'on ordonnera par puissances décroissantes ;*
5. *Dans la variable* `df`*, stocker la* fonction *de deux variables $\dfrac{\partial f}{\partial x}$ (dérivée de f par rapport à la variable x).*

2.5 Séquences, listes, ensembles

2.5.1 Séquence

Une séquence (ou suite) est une collection d'objets Maple séparés par des virgules. La séquence suivante est stockée dans la variable ma_sequence. Son premier élément est une variable non-affectée, le deuxième une expression et le troisième une fonction.

Maple
```
> ma_sequence := a, x ^ 2, x -> x ^ 3;
```

On accède au i^e élément d'une séquence à l'aide de crochets :

Maple
```
> ma_sequence[2];
```

Il n'est pas possible de modifier un élément d'une séquence une fois celle-ci crée.

2.5.2 Ensembles

Un ensemble est formé d'éléments tous distincts, dans un ordre indéfini. On définit un ensemble en Maple en entourant une séquence d'accolades :

Maple
```
> ens := {1, 2, x ^ 2, 2, 1};
```

Remarquez que les éléments 2 et 1 ne sont pas répétés. Comme l'ordre des éléments est indéfini, il est impossible d'accéder au i^e élément d'un ensemble. Les opérations mathématiques sur les ensembles sont les suivantes :

- `intersect` : réalise l'intersection de deux ensembles ;

  Maple
  ```
  > A := {1, 2, 3, 4};
  > B := {2, 4, 5, 6};
  > A inter B;
  ```

- `union` : réalise l'union de deux ensembles ;

  Maple
  ```
  > A union B;
  ```

- `minus` : réalise le complément d'un ensemble dans un autre :

  Maple
  ```
  > A minus B;
  ```

Exercice 19. *Définissez une fonction stockée dans une variable* sigma *qui prend deux ensembles* A *et* B *comme arguments et qui renvoie l'ensemble* $A \triangle B = (A \cup B) \setminus (A \cap B)$ *(différence symétrique). Par exemple, si* $A = \{1,2,3,4\}$ *et* $B = \{2,4,5,6\}$, $A \triangle B = \{1,3,5,6\}$.

Les ensembles sont utilisés par plusieurs fonctions Maple, pour remplacer un argument par plusieurs. Par exemple, pour tracer le graphe d'une expression, on utilise la fonction plot de la façon suivante :

`plot(expr, var = intervalle)`

Si l'on désire tracer sur un même graphe plusieurs expressions, on remplace l'unique expression expr par un ensemble d'expressions :

Maple
```
> plot( {sin(x), sin(2 * x), sin(3 * x)}, x = 0 .. 2 * Pi);
```

On peut de même effectuer des substitutions multiples dans une expression en utilisant un ensemble. C'est une méthode très commode pour obtenir des applications numériques à partir de formules abstraites : il suffit de collecter toutes les valeurs numériques des variables dans un ensemble :

```
Maple
> F := g * sin(lambda * t);
> G := g * lambda / t;
> app_numerique_1 := {g = 9.8, lambda = Pi, t = 2};
> app_numerique_2 := {g = 9.8, lambda = 2 * Pi, t = 1};
> subs(app_numerique_1, F);
> simplify(%);
> subs(app_numerique_1, G);
> simplify(%);
> subs(app_numerique_2, G);
> simplify(%);
```

`Exercice 20.` *Cherchez de l'aide sur la fonction* `solve` *(la description uniquement), et résolvez le système d'équations suivant :*

$$\begin{cases} x + y = a \\ x - y = b \end{cases}$$

où les inconnues sont les variables x, y et où les variables a, b sont des paramètres.

`Exercice 21.` *Tracer sur un même graphe la représentation de la fonction* $\sin$, *et les parties régulières de ses développements limités aux ordres* 2, 4 *et* 6. *On ne recopiera rien, mais on affectera des variables et on utilisera des fonctions déjà rencontrées.*

2.5.3 Listes

Une liste contient des éléments, qui peuvent être répétés, et dans un ordre bien déterminé (un peu comme une séquence). Pour définir une liste en Maple, il suffit d'entourer une séquence de crochets :

```
Maple
> l := [1, 3, 2, 3];
> l[2];
> l[5];
```

On accède au ième élément d'une liste comme pour une séquence. La structure de liste est également très utilisée dans les fonctions classiques de Maple. Par exemple, pour tracer plusieurs graphes d'expressions simultanément et imposer une couleur pour chacun, on utilise la fonction `plot` de la façon suivante :

```
Maple
> plot([sin(x), sin(2 * x), sin(3 * x), x = 0 .. 2 * Pi, color = [blue, red,
green] );
```

Il faut ici utiliser une liste à la place d'un ensemble, car l'ordre des expressions à tracer importe. Remarquez l'utilisation d'une liste pour spécifier les couleurs du tracé.

Remarque 3. Une liste en Maple ressemble plus à un tableau qu'à une liste chaînée que l'on retrouve dans d'autres langages de programmation : on peut accéder directement à un élément quelconque.

On transforme une liste (ou un ensemble) en une séquence, grâce à l'opérateur `op` :

```
Maple
> l := [x, 2 * x, 3 * x];
> e := {x, 2 * x};
> op(l);
> op(e);
```

Réciproquement, pour transformer une séquence en une liste (ensemble), il suffit d'entourer cette séquence de crochets (accolades) :

Maple
```
> s := x, x ^ 2, 3 * x;
> l := [s];
> e := {s};
```

Pour ajouter un élément en fin de liste, on utilise les instructions suivantes :

Maple
```
> l := [op(l), nouvel_element];
```

On part d'une liste l (éventuellement vide), que l'on transforme en séquence grâce à op, on rajoute un élément à cette séquence, avec une virgule et le nouvel élément, puis on transforme cette nouvelle séquence en liste en l'entourant de crochets. On réaffecte enfin cette nouvelle séquence à l.

Maple
```
> l := [];
> l := [op(l), x];
```

Si l'on répète plusieurs fois la deuxième instruction, on ajoute au fur et à mesure l'élément x en fin de liste. Nous utiliserons plus tard ce procédé de construction en programmation : on part de la liste vide, et l'on rajoute (dans une boucle) progressivement de nouveaux éléments en fin de liste.
On peut appliquer une fonction à chaque élément d'une liste (plus généralement à chaque sous-expression d'une expression) en utilisant la fonction `map(fonc, liste)` :

Maple
```
> l := [1, 2, 3];
> f := x -> x ^ 2;
> map(f, l);
```

2.5.4 Construction automatique de séquences

Pour construire automatiquement des séquences (et donc des listes ou des ensembles), on utilise la fonction `seq` :
`seq(expr, ind = intervalle)`
où expr est une expression qui dépend d'un indice `ind`.

Maple
```
s := seq( (x + y) ^ n, n = 1 .. 10);
```

Transformons cette séquence en liste, et appliquons à chaque élément de cette liste la fonction expand :

Maple
```
> l := [s];
> map(expand, l);
```

`Exercice 22.` 1. *Cherchez de l'aide sur les fonctions associées aux nombres premiers (prime numbers en Anglais) ;*
2. *Produire la liste des* 100 *premiers nombres premiers*
3. *Produire à partir de cette liste, la liste des restes de ces* 100 *nombres dans la division euclidienne par* 5.

Chapitre 3

Programmation

3.1 Corps d'une procédure Maple

On peut créer des programmes sous Maple qui utilisent les facilités du calcul formel, contrairement à d'autres langages de programmation (CAML, PASCAL, C, ...). Malheureusement, la syntaxe trop peu rigoureuse ne destine Maple qu'à de petits scripts orientés mathématique ...
Un programme (ou procédure) réunit plusieurs instructions effectuées les unes après les autres dans l'ordre où elles apparaissent. Une procédure est en fait une *fonction* qui utilise 0 ou plusieurs *arguments* (données fournies par l'utilisateur lors de l'appel de la procédure). La structure d'un programme est la suivante :

Maple
```
> var := proc(arg_1, ... , arg_n)  # procédure affectée dans var
    local var_1, ... , var_n;      # variables locales
    global var_1, ..., var_n;      # variables globales
    instr_1;                       # les instructions
    ...
    instr_n;                       # résultat renvoyé
    end;                           # fin de la procédure
```

La procédure est affectée à une variable var. L'utilisateur appelle cette fonction comme une fonction ordinaire :

Maple
```
var(arg_1, ... , arg_n);
```

Remarque 4.

- Il n'y a pas de point virgule à la fin de la première ligne ;
- La deuxième ligne permet de définir des *variables locales*, c'est à dire des variables que l'on pourra utiliser à l'intérieur du programme (pour stocker des résultats intermédiaires par exemple). Ces variables locales n'existent pas à l'extérieur du programme. Si une variable locale porte le nom d'une variable existante à l'extérieur du programme, mais non déclarée comme variable globale, cette variable locale à l'intérieur du programme n'aura rien à voir avec la variable extérieure ;
- La troisième ligne faculative permet de déclarer des variables extérieures au programme. On peut utiliser le contenu de variables déjà affectées en dehors de la procédure à condition de les déclarer comme *variables globales* ;
- Vient ensuite le *corps* du programme : plusieurs instructions séparées par des point virgules. A chaque appel de la procédure, le programme effectuera dans l'ordre ces instructions ;
- Le résultat renvoyé correspond au résultat renvoyé par la dernière instruction ;
- La fin du programme est déclarée avec le mot clé end.

Dans les instructions du corps de la procédure, on peut *lire* la valeur des arguments et l'utiliser, mais *on ne peut pas affecter les arguments*. Par contre on peut affecter et lire le contenu des variables locales (c'est leur intérêt).

Pour taper un programme, ne pas utiliser la touche *Entrée* pour passer à la ligne : il faut utiliser la combinaison `MAJ + Entrée` (tenir la touche Majuscule enfoncée puis appuyer sur Entrée). Seulement après `end;` appuyer sur `Entrée` pour valider le programme. Pour éviter des problèmes, il est conseillé de taper la première et dernière ligne (avec `MAJ+Entrée` entre chaque ligne), puis `Entrée`. Ensuite, revenir avec la souris à la deuxième ligne, et taper les instructions du programme.
Un exemple de programme que l'on va stocker dans la variable `essai` : il utilise un argument a et une variable locale S :

Maple
```
> essai:= proc(a)
    local S;
    S := a;
    S := S + a;
    S;
  end;
```

Si l'on tapait les trois instructions du programme, le résultat de la dernière instruction vaudrait $2a$. C'est ce que fait ce programme : il renvoie le double de l'argument a :

Maple
```
> essai(5);
      10
```

3.2 La boucle FOR

L'intérêt d'utiliser des procédures est de répéter un certain nombre de fois une instruction, ce qui serait fastidieux à faire à la main. On utilise pour cela une instruction de boucle :

```
for ind from debut to fin by incr do
   instr_1;
   ...
   instr_n;
od;
```

qui se comprend de la façon suivante : au départ, la variable `ind` vaut `debut`. Si `ind` $\leq$ `fin`, on effectue les instructions 1 à n, sinon c'est terminé et on sort de la boucle. Ensuite, la variable `ind` est incrémentée de la valeur `incr` : $ind \leftarrow ind + incr$. Si la nouvelle valeur de `ind` est inférieure ou égale à b, on exécute à nouveau les instructions, sinon on sort de la boucle. On recommence ainsi, jusqu'à ce que `ind` $> b$. Remarquer le mot clé `od` (l'anagramme de `do`) qui termine la boucle.
Par défaut, l'incrément vaut 1, et l'on recontre le plus souvent une boucle `for` sans `by` :

```
for ind from debut to fin do
  instr_1;
  ...
  instr_n;
od;
```

Une boucle for représente *une* instruction dans une procédure.

3.2.1 Calcul de sommes et produits

Ecrivons par exemple un programme qui prend comme argument un entier $n \geq 1$ et qui va renvoyer la somme

$$\sum_{i=1}^{n} i = 1 + 2 + \cdots + (n-1) + n$$

Nous allons utiliser une variable locale S, qui va être initialisée au départ avec 0 et créer une boucle qui sera parcourue n fois. Après le premier passage, S vaudra 1, après le deuxième passage, S vaudra $1 + 2$, après le ième passage, S vaudra $1 + \cdots + i$, et après le nième passage, S contiendra le résultat voulu $S = 1 + 2 + \cdots + n$.

Il suffira dans la dernière instruction de notre programme d'afficher le contenu de S. A chaque passage dans la boucle, l'opération à faire consiste à rajouter la valeur de l'indice de boucle i à la variable S, ce qu'on peut schématiser par :

$$S \leftarrow S + i$$

Le programme correspondant s'écrit :

Maple
```
> somme := proc(n)
>   local S,i;
>   S := 0;                 # initialisation de la variable locale
>   for i from 1 to n do    # une boucle parcourue n fois
>        S := S + i;        # instruction effectuée a chaque passage
>   od;                     # fin de la boucle
>   S;                      # la derniere instruction : le programme
> end;                          #renvoie la valeur de S
```

Pour faciliter la lisibilité du programme, on *indente* (deux espaces supplémentaires par rapport à la colonne courante) les instructions dans la boucle.

Exercice 23. *Ecrire une procédure* `somme(f,n)` *qui prend en arguments une fonction, un entier et qui renvoie la valeur*

$$\sum_{k=1}^{n} f(k)$$

Exercice 24. *Ecrire une procédure* `produit(f,n)` *qui prend comme arguments une fonction, un entier et qui renvoie la valeur*

$$\prod_{k=1}^{n} f(k)$$

Utiliser cette procédure pour calculer $20!$.

3.2.2 Calcul des termes d'une suite récurrente.

Une autre utilisation classique d'une boucle for : le calcul des termes d'une suite récurrente. Considérons une suite de réels définie par les relations :

$$u_0 = a \quad u_{n+1} = f(u_n)$$

où $f : \mathbb{R} \mapsto \mathbb{R}$ est une fonction. Connaissant $u_0 \in \mathbb{R}$, la relation de récurrence permet de calculer u_n pour $n \in \mathbb{N}$ quelconque. Nous allons écrire un programme qui prend comme arguments la fonction f, le réel a, un entier n et qui renvoie la valeur du n$^{\text{e}}$ terme de la suite, u_n :

Maple
```
> iteration := proc(f, a, n)
    local U, i;
    U := a;                  # U contient u_0
    for i from 1 to n do
       U := f(U);            # U contient u_i
    od;
    U;                       # U contient u_n
  end;
```

Utilisons ce programme pour calculer u_{10} où la suite u_n est définie par :

$$u_0 = 0.1 \quad u_{n+1} = u_n - u_n^2$$

puis pour $u_0 = 2$.

```
Maple
> fonc := x -> x - x ^ 2;    # la fonction associee a la suite
> iteration(fonc, 0.1, 10);
> iteration(fonc, 2.0, 10);
```

Remarquez que la procédure `iteration` réclame un deuxième argument de type réel. C'est pour cela qu'il faut écrire 2.0 (ou 2.) et non 2 qui est considéré comme un objet formel :

```
Maple
> iteration(fonc, 2, 10);
```

Nous allons modifier ce programme pour qu'il renvoie la liste $[u_0,u_1,\ldots,u_n]$. On définit une variable locale, `liste` initialisée avec la liste vide `[]` et l'on rajoute l'élément u_i à la fin de cette liste à chaque passage dans la boucle :

```
Maple
> formelist := proc(f, a, n)
    local S, list, i;
    list := [];
    S := a;
    for i from 0 to n do
       list := [op(list), S];  # list contient [u_0, ... ,u_i]
       S := f(S);              # S contient u_{i+1}
    od;
    list;                      # list contient [u_0, ... , u_n]
  end;
```

`Exercice 25.` *Il existe une méthode très efficace pour trouver une valeur approchée de $\sqrt{a}$ $(a > 0)$. On considère pour cela la suite récurrente définie par :*

$$u_0 = 1 \quad u_{n+1} := \frac{1}{2}\left(u_n + \frac{a}{u_n}\right)$$

On montre que cette suite converge vers $\sqrt{a}$ de manière quadratique*, c'est à dire qu'à partir d'un certain rang, le nombre de décimales communes entre u_n et $\sqrt{a}$ double à chaque itération ! Notons ϵ_n l'erreur commise en approximant $\sqrt{a}$ par u_n :*

$$\epsilon_n := |u_n - \sqrt{a}|$$

Ecrire une procédure `quadratique` *qui prend comme arguments le réel a, l'entier n et qui renvoie la liste $[\epsilon_1,\ldots,\epsilon_n]$. Vérifier sur cette liste que la convergence est quadratique. Pour cela, faire les calculs avec 50 chiffres significatifs en tapant l'instruction*

```
Maple
  > Digits := 50;
```

avant d'appeler la procédure (remarquer la majuscule comme pour toute constante Maple)

`Exercice 26.` *La suite de Fibonacci est définie par*

$$F_0 = 1, \quad F_1 = 1, \quad \forall n \geq 1, \quad F_{n+1} = F_n + F_{n-1}$$

1. *Ecrire une procédure* `fibo` *qui prend comme argument un entier n et qui renvoie l'entier F_n ;*
2. *Écrire une procédure* `fibo_list` *qui renvoie la liste $[F_0,F_1,\ldots,F_n]$.*

3.3 La boucle while

Il arrive que l'on écrive un programme qui doit répéter les mêmes opérations un certain nombre de fois, sans savoir à l'avance le nombre de répétitions nécessaires. On utilise alors une boucle `while` :

Maple
```
> while condition do
    instr_1;
    ...
    instr_n;
  od;
```

où cond est une condition qui dépend des données du programme : c'est un objet de type booléen (par exemple $x = 0$, $x <> 1$, $x + 1 <= y - 3$, $(x = 1) and (y = 2)$, $(x = 0) or (y = 0)$, $isprime(i)$ ...). L'instruction se comprend de la façon suivante : tant que la condition cond est vraie, effectuer les instructions $instr_1, \dots, instr_n$. Dès que la condition est fausse, sortir de la boucle.
Par exemple, considérons la suite récurrente

$$u_0 = a > 0 \quad u_{n+1} = \sqrt{u_n}$$

On sait qu'elle est convergente et on veut trouver une valeur approchée de sa limite. On va itérer cette suite jusqu'à ce que deux termes consécutifs soient très proches : $|u_{n+1} - u_n| \leq \varepsilon$, où ε est une précision donnée par l'utilisateur. Pour cela, utilisons une variable locale S initialisée avec $S \leftarrow a$. Tant que $|S - \sqrt{S}| > \varepsilon$, on réaffecte S : $S \leftarrow \sqrt{S}$. A la fin, il suffit d'afficher le contenu de la variable locale S. Ce programme s'écrit simplement :

Maple
```
> iter := proc(a,epsilon)
    local S;
    S := a;
    while abs(S - sqrt(S)) > epsilon do
       S := sqrt(S);
    od;
    S;
   end;

 > iter(2.0, 0.0001);
```

Retenons qu'on utilisera une boucle `for` dans un programme si l'on peut prévoir à l'avance le nombre de passages dans la boucle. Sinon, on utilisera une boucle `while`. La boucle `while` est plus dangereuse que la boucle `for` car on n'est pas sûr que le programme s'arrêtera (la condition de boucle peut être toujours vérifiée !). Il faut donc une preuve humaine de l'algorithme employé pour garantir la terminaison.

`Exercice 27.` *Reprenons la suite d'un exercice précédent :*

$$u_0 = 1, \quad \forall n \in \mathbb{N}, u_{n+1} = \frac{1}{2}\left(u_n + \frac{a}{u_n}\right)$$

Ecrire un programme qui prend pour arguments un réel positif a, un réel positif ϵ et qui renvoie le premier entier n tel que

$$\left|u_{n+1} - \sqrt{a}\right| < \epsilon$$

3.4 Les tests

Dans un programme, il se peut que suivant les données, il faille exécuter une instruction ou une autre. On utilise pour cela une instruction de test :

Maple
```
if condition then instr fi;
```

où `condition` est de type booléen. Remarquer le mot clé `fi` qui est l'anagramme de `if`. À la place de `instr`, on peut mettre plusieurs instructions séparées par des points virgule. On peut également effectuer deux opérations différentes selon que *condition* est vraie ou fausse :

Maple
```
if condition then instr_1 else instr_2
```

Si cond s'évalue à vrai, on effectue `instr_1` sinon, on effectue `instr_2`.
Définissons par exemple la fonction suivante :

$$f : \begin{cases} \mathbb{R} & \longrightarrow & \mathbb{R} \\ x & \mapsto & \begin{cases} 0 & \text{si } x \leq 0 \\ e^{-\dfrac{1}{x^2}} & \text{si } x > 0 \end{cases} \end{cases}$$

Maple
```
> f := proc(x)
    if x > 0 then exp(-1 / x ^ 2) else 0 fi;
  end;
```

On peut utiliser plusieurs tests imbriqués :

Maple
```
if cond_1 then instr_1
elif cond_2 then instr_2
...
elif cond_n then instr_n
else instr_defaut
fi;
```

Si `cond_1` est vraie, on exécute `instr_1`. Sinon, et si `cond_2` est vraie, on exécute `instr_2` ... Enfin si aucune des n conditions précédentes n'est vérifiée, c'est l'instruction `instr_defaut` qui est exécutée.

Remarque 5. En général on commence par traiter le cas général du problème qui correspond à la condition 1 vraie. Ensuite, on étudie les cas de plus en plus particuliers.

Par exemple, écrivons un programme qui prend comme argument un entier n, et qui renvoie $2n$ si $n = 3k$, $3n$ si $n = 3k + 1$ et $4n$ si $n = 3k + 2$. Nous allons utiliser la fonction `mod(a, b)` qui renvoie le reste de la division euclidienne de a par b :

```
> f := proc(n)
    local r;
    r := n mod 3;
    if r = 0 then 2 * n
       elif r = 1 then 3 * n
       else 4 * n
    fi;
 end;
```

`Exercice 28.` *Ecrire un programme qui prend comme arguments deux réels a,b et qui renvoie le réel solution de l'équation $ax = b$ si elle est unique ou alors les chaines de caractères* `‘pas de solutions‘` *ou* `‘tout x est solution‘` *dans les autres cas.*

3.5 Conception de programmes

Nous disposons des deux outils principaux, les boucles et les tests. Il s'agit maintenant de concevoir des programmes clairs. Tout d'abord on précise les arguments de la procédure et les variables locales à utiliser. Ensuite on décrit le programme en complétant le plan suivant :

1. Arguments et leurs types ;

2. Variables locales et leurs types;
3. Initialisation des variables locales ;
4. Corps du programme (en général une boucle) ;
5. Dernière instruction (ce que renvoie la procédure).

Par exemple, le programme qui calcule le nième terme de la suite

$$u_0 = a \quad u_{n+1} = f(u_n)$$

est décrit par :

1. **Arguments :**
   - f : une *fonction* (la fonction à itérer) ;
   - a: de type *float* (la valeur de u_0) ;
   - n: de type *int* (le nombre d'itérés) ;
2. **Variables locales :**
   - S : de type *float*
3. **Initialisation des variables locales**
$$S \leftarrow a$$
4. **Corps de la procédure**
   Pour i variant de 1 à n faire :
$$S \leftarrow f(S)$$
5. **Fin :** Afficher le contenu de S.

`Exercice 29.` *Décrire puis implémenter une procédure qui prend un entier n comme argument et qui renvoie la quantité*

$$\sum_{i=1}^{n} \sum_{j=1}^{n} e^{ij}$$

3.6 Mise au point des programmes

Lorsqu'un programme se plante, ou retourne un résultat erroné, il est intéressant de suivre pas à pas les opérations qu'il effectue pour détecter l'erreur. On utilise pour cela la commande `debug(nom)` (`trace(nom)` pour la version 4) où `nom` est le nom de la procédure. Maple affiche alors tous les résultats intermédiaires. Pour désactiver cette fonctionnalité, utiliser `undebug(nom)` (respectivement `untrace(nom)`).

3.7 Tableaux

Maple permet de définir des tableaux à une ou plusieurs dimensions grâce à la commande array :

```
Maple
> a := array(1..10);
> a[1] := 3; a[4] := Pi;
> print(a);
```

Maple utilise ces tableaux en algèbre linéaire. Nous allons faire un peu d'algorithmique classique sur les tableaux. Pour cela, nous définirons deux variables globales : un entier n qui désigne le nombre d'éléments d'un tableau et un tableau

$$t = [t_0, \dots ,t_{n-1}]$$

En algorithmique, on a l'habitude d'utiliser les indices 0 à $n-1$ pour les éléments d'un tableau.

```
Maple
> n := 50;
> a := array(0..n);
```

La fonction rand(a..b) renvoie une procédure qui permet d'obtenir des entiers aléatoires compris entre a et b :

```
Maple
> alea := rand(0..100);
```

Servons-nous de ce générateur de nombres aléatoires pour initialiser notre tableau t :

```
Maple
init_tab := proc()
  global n, t;
  for i from 0 to (n - 1) do
      t[i] := alea();
  od;
end;
```

On peut vérifier que le tableau a été rempli avec des nombres aléatoires :

```
Maple
> print(t);
```

Écrivons une première procédure qui détermine le plus grand élément du tableau t :

```
Maple
max_tab := proc()
  global n, t;
  local M;
  M := t[0];
  # M = t[0]
  for i from 1 to (n-1) do
     if t[i] > M then M := t[i] fi;
     # INV : M = max(t0, ... , ti)
  od;
  # M = max(t0, ... , t(n-1))
  M;
end;
```

Nous voulons maintenant réorganiser les éléments du tableau t pour qu'ils soient triés par ordre croissant. Il existe plusieurs algorithmes classiques de tris. Dans ces algorithmes, on évite d'utiliser un tableau auxiliaire (imaginez que la taille du tableau t soit immense !). On veut trier *sur place* le tableau. Écrivons d'abord une fonction echange(i, j) qui échange les éléments t_i et t_j du tableau t :

Maple
```
echange := proc(i, j)
  global t;
  local temp;
  temp := t[i];
  t[i] := t[j];
  t[j] := temp;
end;
```

Le tri par sélection

L'idée du tri par sélection est simple. Supposons placés par ordre croissant les i premiers éléments du tableau t au début : $[t_0, \ldots, t_{i-1}]$. On recherche le plus petit élément parmi $t_i, \ldots, t_{n-1}$. Il se trouve à l'indice j, et l'on échange t_i avec t_j. Ainsi, les $(i+1)$ plus petits éléments du tableaux t seront placés en tête et triés par ordre croissant. Il suffit de répéter cette étape pour i variant de 0 à $(n-2)$.

Maple
```
tri_selection := proc()
  global n, t;
  local ind_min;
  for i from 0 to (n - 2) do
    # Recherche de min(t(i), ..., t(n-1))
    ind_min := i;
    for j from (i + 1) to (n - 1) do
       if t[j] < t[ind_min] then ind_min := j fi;
    od;
    # INV : i <= k <= n - 1 ==> t[ind_min] <= t[k]
    echange(i, ind_min);
    # INV : t[0] <= ... <= t[i] et i<k<n-1 ==> t[k] >= t[i]
  od;
  # t[0] <= ... <= t[n-1]
end;
```

Déterminez le nombre de comparaisons nécessaires (dans le pire des cas) pour trier le tableau t avec cette méthode.

Le tri bulle

Dans cet algorithme, on parcourt le tableau de gauche à droite, et l'on échange deux éléments successifs s'ils ne sont pas dans le bon ordre. Après le premier passage, le plus grand élément du tableau se trouve placé dans la case $n-1$. On recommence de 0 à $n-2$ et ainsi de suite.

Maple
```
tri_bulle := proc()
  global n, t;
  for d from (n-2) to 0 by -1 do
     for i from 0 to d do
        if t[i] > t[i+1] then echange(i, i + 1) fi;
     od;
     # INV : t[d+1] <= ... <= t[n-1] et 0 <= k <= d ==> t[k] <= t[d+1]
  od;
end;
```

3.8 Récursivité

On peut écrire des fonctions qui « s'appellent elle-mêmes » avec de nouveaux arguments. La théorie des fonctions récursive est proche de la récurrence mathématique. Par exemple, on peut définir par récurrence :

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \times (n-1)! & \text{si } n \geq 1 \end{cases}$$

La fonction récursive correspondante s'écrit :

Maple
```
fact := proc(n)
  if n = 0 then 1 else
    n * fact(n - 1);
  fi;
end;
```

Le problème de la terminaison d'une fonction récursive est crucial. Il faut vérifier que la fonction ne s'appelle récursivement que sur des arguments « plus petits » jusqu'à aboutir au cas d'arrêt en un nombre fini d'étapes. Pour calculer a^n, on peut utiliser l'algorithme d'exponentiation rapide :

$$a^n = \begin{cases} 1 & \text{si } n = 0 \\ y * y & \text{si } n = 2p \text{ avec } y = a^p \\ a * y * y & \text{si } n = 2p+1 \text{ avec } y = a^p \end{cases}$$

Maple
```
expo_rapide := proc(a, n)
  local y;
  if n = 0 then 1
  elif irem(n, 2) = 0 then
     y := expo_rapide(n, iquo(n, 2));
     y * y;
  else
     y := expo_rapide(n, iqui(n, 2));
     a * y * y;
  fi;
end;
```

Exercice 30. *Pour calculer le* pgcd *de deux entiers strictement positifs* n *et* m*, on utilise l'algorithme d'Euclide : Si* r *est le reste de la division euclidienne de l'entier* n *par l'entier* m*, on montre que* $\text{pgcd}(n,m) = \text{pgcd}(m,r)$*. Utiliser ce résultat pour écrire une fonction récursive qui calcule le* pgcd *de deux entiers.*

Chapitre 4

Graphiques

4.1 La fonction plot

Nous avons déjà vu l'utilisation standard de la fonction `plot` pour tracer le graphe d'une expression :

`plot(expr, var = intervalle, options);`

Par exemple :

```
Maple
> plot(sin(x), x = 0 .. 2 * Pi, color = red);
```

Voyez la page d'aide de `plot` pour les différentes options. Il est possible de tracer plusieurs graphes $y = f(x)$ simultanément, en utilisant une structure de liste :

`plot( liste_expressions, var = intervalle, options);`

Par exemple :

```
Maple
> plot( [sin(t), sin(2 * t), cos(t)], t = 0 .. 2 * Pi, color =
    [blue, red, green]);
```

On peut également tracer une ligne polygônale joignant les points $A_1, \ldots, A_n$ de coordonnées $A_i = (x_i, y_i)$. Les coordonnées de chaque point sont placées dans une liste $[x_i, y_i]$, et l'on forme la liste de ces listes :

```
Maple
> liste_points := [ [0, 0], [2, 1], [1, 3], [4, -1] ];
> plot(liste_points);
```

Si l'on veut tracer uniquement le nuage de point, il suffit d'utiliser l'option `style = point` :

```
Maple
> plot( [ [0, 1], [2, 3], [1, -2] ], style = point );
```

La même fonction `plot` permet également de tracer des courbes paramétrées avec la syntaxe :

`plot( [ expr_x, expr_y, var = intervalle ] );`

où `expr_x` et `expr_y` sont des expressions en la variable `var`. Remarquez l'utilisation d'une liste dans cette syntaxe.

```
Maple
> plot( [ sin(t), cos(t), t = 0 .. Pi] );
```

`Exercice 31.` *Une roue de bicyclette de rayon R roule sans glisser sur une route. On considère un point situé sur la périphérie de la roue. Il s'agit de tracer sa trajectoire. On prendra comme paramètre l'angle θ du schéma. Déterminer la relation entre l et θ, puis les coordonnées de C en fonction de θ et enfin les coordonnées de M en fonction de θ. Tracer ensuite la trajectoire du point pour $\theta \in [0{,}4 * Pi]$.*

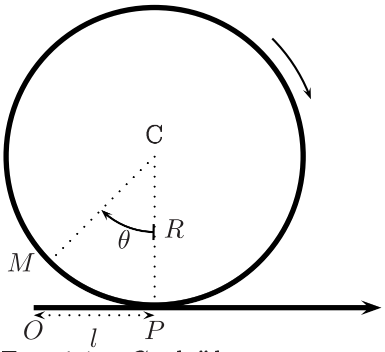

FIG. 4.1 – *Cycloïde*

Exercice 32. *Sur un oscilloscope, si on utilise une certaine fréquence de balayage horizontale et une autre fréquence de balayage vertical, le point lumineux décrit une courbe paramétrée de la forme*

$$x(t) = \sin(\omega_1 t), \quad y(t) = \sin(\omega_2 t)$$

Représenter quelques courbes (dites de Lissajoux) pour différentes fréquences. Utiliser un grand intervalle de temps ($t \in [0,200]$). Qu'observe-t-on si le rapport $\frac{\omega_1}{\omega_2}$ est rationnel ? S'il n'est pas rationnel ? (On peut montrer alors que la courbe est « dense » dans le carré $[-1,1] \times [-1,1]$).

4.2 Superposition de graphiques

La fonction plot permet de superposer plusieurs courbes $y = f(x)$ sur un même graphique, définies sur un même intervalle. Pour superpose des graphiques quelconques, on utilise une méthode plus évoluée : la fonction display qui se trouve dans le package plots :

Maple
```
> with(plots);
> graph_1 := plot(sin(x), x = -Pi .. Pi):
> graph_2 := plot([sin(t), 2 * cos(t), t = 0 .. Pi]):
> display([graph_1, graph_2]);
```

Après avoir chargé le package plot, on définit deux graphiques, que l'on stocke dans les variables graph_1 et graph_2. Remarquez l'utilisation du symbole : à la place du point virgule pour effectuer les calculs sans afficher les résultats. En effet, la structure d'un objet graphique en Maple est longue et peu intéressante (remplacez les deux points par un point virgule pour vous en convaincre). On superpose ensuite les deux graphiques placés dans une liste grâce à la fonction display.

4.3 Autres graphiques 2D

Un point $M(x,y)$ du plan peut être représenté par ses coordonnées polaires (ρ,θ), avec

$$x = \rho\cos(\theta), \quad y = \rho\sin(\theta)$$

Une courbe polaire est l'ensemble des points du plan vérifiant une relation de la forme

$$\rho = f(\theta)$$

On peut tracer une courbe polaire dans un secteur angulaire limité par les angles $[\theta_0,\theta_1]$ avec la commande polarplot(expr, angle = intervalle
Cette commande fait partie du package plots, qu'il faut donc chargée au préalable :

Maple
```
> with(plots);
> polarplot( cos(2 * u), u = 0 .. 4 * Pi);
```

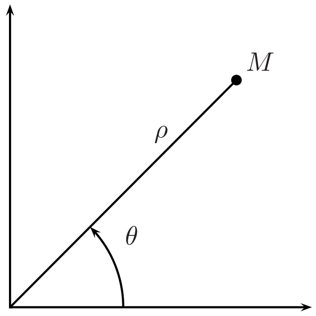

FIG. 4.2 – *Coordonnées polaires*

Exercice 33. *Tracer la cardioïde d'équation polaire*

$$\rho = 1 + \cos(\theta), \quad \theta \in [0,2\pi]$$

4.4 Graphiques 3D

4.4.1 Courbes paramétrées dans l'espace

Ce sont des courbes définies par des équations paramétriques

$$x = x(t), \quad y = y(t), \quad z = z(t)$$

où t est un paramètre qui varie dans l'intervalle $[t_0,t_1]$. On les trace avec la commande
`spacecurve( [expr_x, expr_y, expr_z, var = intervalle] );`
où `expr_x`, `expr_y` et `expr_z` sont des expressions en la variable `var`. La fonction `spacecurve` se trouve dans le package `plots`. Remarquer comme pour les courbes paramétrées planes l'utilisation du type liste. On peut utiliser l'option `numpoints` pour augmenter le nombre de points du tracé.

Maple
```
> with(plots);
> spacecurve([t, cos(t), sin(2 * t)], t = 0 .. 2 *  Pi);
```

4.4.2 Champs de vecteurs

En chaque point du plan, on associe un vecteur (représentant une force par exemple). On obtient alors un *champ de vecteurs*, c'est à dire une application $\overrightarrow{F} : \mathbb{R}^2 \mapsto \mathbb{R}^2$, avec $\overrightarrow{F} = (X(x,y),Y(x,y))$. On trace ce champ de vecteurs grâce à la commande `fieldplot` qui se trouve dans le package `plots` :
`fieldplot( [expr_X, expr_Y], var_1 = intervalle_1,  var_2 = intervalle_2)`
où `expr_X` et `expr_Y` sont des expressions en les deux variables `var_1` et `var_2`.

Maple
```
> with(plots);
> fieldplot([cos(x * y), sin(x + y)], x = -Pi .. Pi, y = -Pi .. Pi);
```

Exercice 34. *Tracer le champ de force correspondant à un champ dérivant du potentiel scalaire :*

$$V(x,y) = \frac{1}{r} = \frac{1}{\sqrt{x^2 + y^2}}$$

(c'est à dire $\overrightarrow{F}(x,y) = \overrightarrow{\nabla V}(x,y) = \begin{pmatrix} \dfrac{\partial V}{\partial x} \\ \dfrac{\partial V}{\partial y} \end{pmatrix}$*)*

4.4.3 Surfaces $z = f(x,y)$

Considérons un domaine du plan $D = [a,b] \times [c,d]$, et une fonction $f : D \mapsto \mathbb{R}$. En chaque point (x,y) du domaine D, on associe la hauteur $f(x,y)$, ce qui représente une surface

$$\Sigma = \{(x,y,z) \in \mathbb{R}^3 \text{ tq } z = f(x,y)\}$$

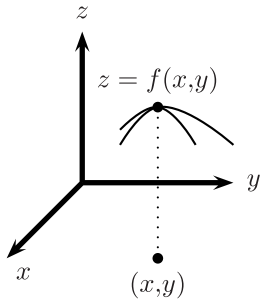

FIG. 4.3 – *surface* $z = f(x,y)$

On trace ces surfaces avec la commande

```
plot3d( expr, var_1 = intervalle_1, var_2 = intervalle_2)
```

où *expr* est une expression en les deux variables `var_1` et `var_2`.

```
Maple
> plot3d( sin(x * y), x = -Pi .. Pi, y = -Pi .. Pi);
```

Cliquer avec la souris sur le graphique, et en maintenant le bouton enfoncé, déplacer le cube formé. Appuyer ensuite sur l'icône R (redessiner). On modifie ainsi le point de vue de la surface. Essayer les différentes icônes de la fenêtre graphique $3D$.
On peut augmenter la précision du tracé avec l'option `numpoints`:

```
Maple
> plot3d( sin(x * y), x = -Pi .. Pi, y = -Pi .. Pi, numpoints = 2000);
```

4.4.4 Courbes de niveau

Etant donnée une surface $z = f(x,y)$, $x \in [x_0,x_1]$, $y \in [y_0,y_1]$ et une hauteur c, si l'on coupe la surface par le plan horizontal $z = c$, on obtient une courbe tracée dans ce plan : la *courbe de niveau* de hauteur c. On trace cette courbe en utilisant la commande (qui se trouve dans le package `plots`) :

```
implicitplot( equation, var_1 = intervalle_1, var_2 = intervalle_2);
```

où `equation` est une équation en les variables `var_1` et `var_2`.

```
Maple
> with(plots);
> implicitplot(x ^ 2 + y ^ 4 = 1, x = -1 .. 1, y = -1 .. 1);
```

`Exercice 35.` *Tracer l'ellipse d'équation cartésienne :*

$$\frac{x^2}{2} + y^2 = 1 \quad x \in [-2..2], y \in [-1..1]$$

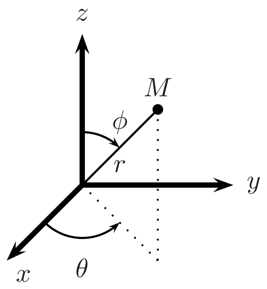

FIG. 4.4 – *Coordonnées sphériques*

4.4.5 Surfaces en coordonnées sphériques

Un point de l'espace peut être défini par ses coordonnées sphériques : r,θ,ϕ avec

$$x = r\sin\phi\cos\theta, \quad y = r\sin\phi\sin\theta, \quad z = r\cos\phi$$

Une surface de l'espace peut être décrite par une équation $r = f(\theta,\phi)$, avec $\theta \in [\theta_0,\theta_1]$, $\phi \in [\phi_0,\phi_1]$. On trace une telle surface avec la commande `sphereplot` (qui se trouve dans le package `plots`) :

`sphereplot( expr, angle_1 = intervalle_1, angle_2 = intervalle_2)`

où expr est une expression en les variables `angle_1` (l'angle θ) et `angle_2` (l'angle ϕ).

Maple
```
> with(plots);
> sphereplot(cos(2 * u) * sin(v), u = 0 .. 2 * Pi, v = 0 .. Pi);
```

`Exercice 36.` *Tracer les orbitales atomiques de l'atome d'hydrogène correspondant à la surface sphérique définie par*

$$r = \cos^2\phi\sin^2\theta$$

$\theta \in [0,2\pi]$,$\phi \in [-\pi/2,\pi/2]$.

4.4.6 Surfaces paramétrées

Ce sont des surfaces de l'espace définies par des équations paramétriques :

$$x = x(u,v), \quad y = y(u,v) \quad z = z(u,v)$$

où u,v sont deux paramètres indépendants, $u \in [u_0,u_1]$, $v \in [v_0,v_1]$.
On trace ces surfaces avec la commande

`plot3d([expr_X, expr_Y, expr_Z],  var_1 = intervalle_1, var_2 = intervalle_2);`

où expr_X, expr_Y, expr_Z sont des expressions en les deux variables `var_1` et `var_2`.

Maple
```
> plot3d( [sin(u * v), cos(v), u], u = 0 .. Pi, v = 0 .. Pi);
```

`Exercice 37.` *Tracer le ruban de Moebius : la surface paramétrée définie par les équations*

$$\begin{cases} x = \big(5 + u\cos(v/2)\big)\cos v \\ y = \big(5 + u\cos(v/2)\big)\sin v \\ z = u\sin(v/2) \end{cases}$$

avec $-1 \leq u \leq 1$ *et* $0 \leq v \leq 2 * Pi$.

Chapitre 5

Intégration

5.1 Changement de variables

Nous avons déjà vu la fonction int qui permet de trouver une primitive ou de calculer une intégrale définie :

Maple
```
> int( sin(x), x);              # primitive
> int( sin(x), x = 0 .. Pi);    # intégrale définie
```

Nous nous intéressons ici à la *forme inerte* de cette fonction qui permet de définir une primitive (ou une intégrale définie) sans que Maple cherche à la calculer :

Maple
```
> F := Int(sin(x) / (1 + cos(x)), x);
```

Cette forme inerte permet d'effectuer des manipulations formelles de primitive : changemenent de variables et intégration par parties. Il nous faut d'abord charger le package student :

Maple
```
> with(student);
```

La fonction qui permet d'effectuer un changement de variables s'appelle

`changevar(equation, forme_inerte, nouvelle_variable)`

Par exemple, pour effectuer le changement de variable $y = \cos(x)$ dans la primitive précédente :

Maple
```
> F := changevar(y = cos(x), F, y);
> value(F);
```

Exercice 38. *On considère l'intégrale définie*

$$F = \int_0^{\pi/4} \frac{\sin x + \cos x}{\sqrt{1 + \sin x \cos x}} \, \mathrm{d}x$$

1. *Essayez de calculer directement cette intégrale ;*
2. *Stockez dans une variable F la forme inerte de cette intégrale ;*
3. *Effectuez le changement de variables $t = x + /pi/4$ et stockez la nouvelle intégrale dans une variable L.*
4. *Développez L, et substituez $\sin^2 t$ par $1 - \cos^2 t$;*
5. *Effectuez un nouveau changement de variables $u = \cos t$;*
6. *Calculez cette dernière intégrale.*

5.2 Intégration par parties

La fonction qui permet d'effectuer une intégration par partie s'appelle
`intparts(forme_inerte, expression_a_deriver)`
Par exemple :

```
Maple
> F := Int(x * ln(x), x);
> Fdef := Int(x * ln(x), x = 1 .. 2);
> G := intparts(F, ln(x));
> Gdef := intparts(Fdef, ln(x));
> value(G);
> value(Gdef);
```

Chapitre 6

Equations différentielles

6.1 Définition d'une équation différentielle

Pour définir une équation différentielle, utiliser `diff`. Par exemple, considérons l'équation différentielle du second ordre suivante :

$$y''(t) + ty'(t) = e^t$$

On la définit par :

Maple
```
> eq := diff(y(t), t$2) + t * diff(y(t), t) = exp(t) %$
```

L'objet équation a été stocké dans la variable eq.

Exercice 39. *Définir et stocker dans une variable* eqdiff *l'équation différentielle du premier ordre suivante :*

$$t * y'(t) + \sin(y(t)) = \cos(t)$$

6.2 Résolution formelle d'une équation différentielle

Comme pour le calcul de primitives, il est rare que l'on sache résoudre explicitement une équation différentielle. Par contre, il existe une classe d'équations différentielles, les équations différentielles *linéaires* que l'on sait résoudre. Maple utilise la fonction

`dsolve(eq, expr)`

où la variable eq contient une équation différentielle, et où `expr` représente l'expression de la fonction inconnue :

Maple
```
> eq := diff(y(t), t) + y(t) = t;
> dsolve(eq, y(t));
```

Le résultat est donné avec une constante d'intégration C_1. Il existe donc une infinité de solutions à cette équation différentielle du premier ordre. On peut préciser des données initiales, pour obtenir une solution unique (voir cours de math. cette année). Par exemple, pour une équation différentielle du premier ordre, si l'on impose la valeur $y(0)$, il y a unicité de la solution. Résolvons par exemple l'équation différentielle :

$$\begin{cases} y'(t) + y(t) = t \\ y(0) = 1 \end{cases}$$

Maple
```
> eq := diff(y(t), t) + y(t) = t;
> dsolve({eq, y(0) = 1}, y(t));
```

Remarquez l'utilisation d'un ensemble pour définir l'équation différentielle et sa condition initiale.
Pour une équation du deuxième ordre, il faut imposer *deux* conditions initiales pour avoir unicité de la solution. Par exemple :

$$\begin{cases} y''(t) + y'(t) + y(t) = 0 \\ y(0) = 0 \\ y'(0) = 1 \end{cases}$$

Maple
```
> eq := diff(y(t), t$2) + diff(y(t), t) + y(t) = 0
> dsolve({eq, y(0) = 0, D(y)(0) = 1}, y(t));
```

Remarquez l'utilisation de l'opérateur `D` (qui prend une fonction en argument et qui renvoie la fonction dérivée).

`Exercice 40.` *Résoudre avec Maple l'équation différentielle*

$$xy'(x) + x^2y(x) = x^2$$

`Exercice 41.` *Résoudre l'équation différentielle avec données initiales suivante :*

$$\begin{cases} y'' - 2y' + y = \sin t \\ y(0) = 1 \\ y'(0) = 0 \end{cases}$$

6.3 Résolution numérique

Comme on ne sait pas résoudre explicitement la majorité des équations différentielles non-linéaires de la physique, il est intéressant d'utiliser des méthodes numériques pour déterminer des solutions approchées et les tracer. On obtient ainsi de précieuses informations sur le comportement qualitatif des solutions. Les fonctions qui permettent de représenter les courbes des solutions se trouvent dans le package `DEtools` :

Maple
```
> with(DEtools);
```

6.3.1 Équations du premier ordre

Pour représenter les graphe d'une solution d'une équation du premier ordre avec la donnée initiale $y(0) = a$, on utilise la fonction `DEplot` :

`DEplot(eq, y(t), t = intervalle, [ [y(0) = a] ], options);`

où `intervalle` représente l'intervalle de temps sur lequel on cherche les solutions. Remarquez l'utilisation d'une liste de liste pour définir la donnée initiale. Voir l'aide pour les différentes options de tracé disponibles. On peut superposer plusieurs courbes solutions correspondant à plusieurs données initiales $y(0) = a_0, \ldots, y(0) = a_n$:

`DEplot(eq, y(t), t = intervalle, [ [y(0) = a], ... , [y(0) = a_n] ], options);`

Pour augmenter la précision de la solution approchée, on *diminue* le pas de temps T utilisé dans la méthode numérique. Pour cela, rajouter l'option `stepsize = T`.

Maple
```
> eq := diff(y(t), t) + y^2(t) = t;
> DEplot(eq, y(t), t = 0 .. 1, [[y(0) = 1], [y(0) = .5]], stepsize = 0.01);
```

6.3.2 Équations du second ordre

Pour tracer des solutions approchées d'équations différentielles du second ordre, on utilise la même fonction DEplot. Il faut *deux* données initiales par courbe intégrale. La syntaxe est la suivante :

```
DEplot(eq, y(t), t = intervalle, [[y(0) = a, D(y)(0) = a']], options)
```

et pour tracer plusieurs courbes sur le même graphique :

```
DEplot(eq, y(t), t = intervalle, [[y(0) = a_0, D(y)(0) = a_0'], ...
  [y(0) = a_n, D(y)(0) = a_n']], options)
```

Par exemple :

```
Maple
> eq := diff(y(t), t$2) + diff(y(t), t) + y(t)^2 = 0
> DEplot(eq, y(t), t = 0 .. 4 * Pi, [[y(0) = 0, D(y)(0) = 1],
    [y(0) = 0, D(y)(0) = .5]], stepsize = 0.01);
```

Exercice 42. *L'équation différentielle qui régit le mouvement d'un pendule simple est non-linéaire :*

$$y''(t) + \sin(y(t)) = 0$$

où $y(t)$ représente l'angle du pendule avec la verticale.

1. *Essayez de résoudre formellement cette équation différentielle ;*
2. *Lorsque l'on s'intéresse à de petites oscillations autour de la position d'équilibre ($y(t)$ petit), les physiciens font l'approximation $\sin(y(t)) \sim y(t)$ et l'équation différentielle linéarisée correspondante s'écrit*

   $$y''(t) + y(t) = 0$$

   On veut vérifier la validité de cette approximation en représentant sur le même graphe la solution de l'équation non-linéaire et la solution de l'équation linéarisée correspondant aux mêmes données initiales. On utilisera un pas de temps de 0.1 ;
3. *Dans une variable* sol_lin*, stocker le graphique de la solution de l'équation linéaire correspondant aux données initiales $y(0) = 0.1$ et $y'(0) = 0$, pour $t \in [0,2\pi]$. Dans une variable* sol_non_lin*, stocker le graphique de la solution de l'équation non-linéaire correspondant aux mêmes données initiales ;*
4. *Superposer les deux graphes ;*
5. *Refaire la même manipulation pour des données initiales $y(0) = \pi/4$, $y'(0) = 1$ et $t \in [0,8\pi]$. Qu'observez-vous ?*
6. *En utilisant l'aide sur la fonction* DEplot*, reprenez les tracés précédents avec une couleur rouge pour la solution non-linéaire et une couleur bleue pour la solution linéaire.*

Index