Algorithmique et programmation au Lycée

Renée De Graeve Bernard Parisse

4 décembre 2017

Table des matières

Index

Chapitre 1

Introduction

1.1 Apprendre en programmant.

Ce document est principalement destiné aux enseignants qui souhaitent utiliser Xcas pour enseigner l'algorithmique au lycée. Nous espérons qu'il sera aussi consulté par des élèves. Dans sa version HTML [1] consultable depuis un navigateur, certains champs de saisies peuvent être modifiés et testés directement, y compris sur une tablette ou un smartphone, ce qui devrait être un plus par rapport à un cours de programmation papier ou PDF (les fonctions "utilitaires" qui sont appelées plusieurs fois par d'autres fonctions n'ont pas besoin d'être validées par l'utilisateur, elles sont interprétées au chargement).

`Xcas` possède deux interfaces. L'interface classique (Xcas natif) fonctionne sur PC et nécessite une installation, cf. la section A.3. On peut aussi utiliser Xcas pour Firefox sans installation depuis tout appareil disposant d'un navigateur (Firefox recommandé), par exemple depuis un smartphone ou une tablette sans installation, depuis le lien :
http ://www-fourier.ujf-grenoble.fr/˜ parisse/xcasfr.html
(L'accès réseau est nécessaire uniquement lors du chargement de la page).
Il n'est donc pas indispensable d'aller en salle informatique pour faire un exercice d'algorithmique pendant un cours de mathématiques, on peut utiliser les smartphones (en mode avion) ou les tablettes des élèves comme des supercalculatrices (formelles, graphiques, 3d, ... il ne manque que le mode examen...). C'est une raison supplémentaire pour écrire ce document, car faire programmer les algorithmes par les élèves nous parait indispensable pour les motiver par la satisfaction de faire tourner son programme puis de le modifier et de l'améliorer.

1. `http://www-fourier.ujf-grenoble.fr/%7eparisse/irem/algolycee.html`

Faire programmer nous semble bien plus formateur que de demander à un élève de comprendre ce que fait un algorithme non commenté, surtout s'il n'a pas de machine pour le tester (par exemple le jour du bac). En effet l'élève n'a aucun rôle créateur (il ne conçoit pas l'algorithme), il a un rôle analogue à celui d'un professeur corrigeant des copies ce qui n'est pas du tout motivant et a devant les yeux un modèle désastreux d'algorithme puisque non commenté ! De plus, comprendre ce que fait un algorithme non commenté est parfois plus difficile que d'en concevoir un, car les notations de variables, par exemple, ne prennent souvent sens qu'une fois l'algorithme compris. Imagine-t-on donner les étapes de calcul d'une démonstration, sans explications, et demander quel est le théorème qui a été démontré ? Bien sur, la programmation s'apprend de manière progressive, en regardant des exemples de code commentés, puis en complétant des maquettes d'algorithmes avant de se lancer dans l'écriture complète de nouveaux algorithmes.

1.2 Le choix du langage de programmation

La plupart des langages interprétés permettent d'apprendre à programmer les concepts algorithmiques au programme du lycée (test, boucle, variable, affectation, fonction). Pour les élèves, la difficulté principale ce sont les concepts algorithmiques, rarement la syntaxe du langage lui-même, à condition de pouvoir se faire aider par l'enseignant s'ils sont bloqués. C'est donc l'enseignant qui devrait choisir un langage avec lequel il se sent à l'aise, non seulement pour écrire lui-même un programme, mais aussi pour trouver rapidement une erreur de syntaxe ou d'exécution dans le programme d'un de ses élèves.

Pour la grande majorité des élèves, nous pensons qu'il serait souhaitable qu'ils soient confrontés lors des changements de professeur à plusieurs langages au cours de leur scolarité (par exemple `Xcas`, calculatrices, Python, Javascript ...), ce qui leur permettrait de mieux comprendre les concepts universels partagés (l'algorithmique) et les biais et particularités propres à un langage donné (voir en appendice), et faciliterait aussi leur adaptation à d'autres langages plus tard. Pour ceux qui se destinent à des études scientifiques, il nous parait important qu'ils soient aussi confrontés à d'autres types de langages (compilés, fonctionnels ...) au cours de leurs études supérieures, dont au moins un langage de plus bas niveau : les langages interprétés permettent d'utiliser facilement des types ou instructions puissantes, se confronter avec un langage de plus bas niveau permet de mieux comprendre ce qui est basique ou ne l'est pas et ce qui est intrinsèquement rapide ou/et couteux en ressource mémoire ou ne l'est pas (on peut voir ça comme l'analogue entre

faire une démonstration ou admettre un théorème).

Actuellement aucun langage n'est imposé dans les textes pour le lycée, nous espérons que cette situation va perdurer et que certains enseignants résisteront aux pressions de vouloir imposer Python comme le seul et unique langage de programmation au lycée (comme pour les enseignements obligatoires d'informatique en classe préparatoires). Pour les aider dans cette démarche, nous avons ajouté la possibilité de programmer dans Xcas avec une syntaxe compatible avec Python (les personnes connaissant Python peuvent consulter l'appendice B.1 qui décrit plus en détails les différences entre Python et `Xcas`).

Pourquoi choisir `Xcas` ?

`Xcas` est fortement orienté mathématique et de ce fait peut facilement interagir avec les thèmes du programme de maths, tous les types mathématiques au programme du lycée sont directement accessibles (par exemple : les nombres entiers, rationnels, réels, complexes, les nombres approchés, les vecteurs, les polynômes et les matrices).

Le langage de programmation natif de `Xcas` est proche du langage algorithmique ce qui facilite ensuite sa traduction dans d'autre langage, y compris en syntaxe Python (menu Fichier, Exporter, texte python). Nous avons adapté le langage en français pour en faciliter l'apprentissage d'après notre expérience d'enseignement avec des publics divers :

- Les structures sont délimitées par des mots-clefs explicites en français
  (`si . alors . sinon . fsi`),
  (`pour . de . jusque . faire ... fpour`),
  (`tantque . faire ... ftantque`)...
  L'indentation sert à controler qu'on n'a pas fait de faute de syntaxe (non-fermeture d'une parenthèse par exemple). Les diverses interfaces de `Xcas` proposent des assistants pour créer facilement les structures de controle usuelles (fonction, test, boucle).
- Il faut déclarer explicitement les variables locales, ainsi une faute de frappe dans un nom de variable est détectée et un avertissement est affiché si une variable n'est pas déclarée ou n'est pas initialisée.
- Lorsqu'on programme une fonction, on peut lui passer en argument des variables qui sont de type fonction ou expression, ceci facilite l'écriture de certains algorithmes (dichotomie, méthode des rectangles par exemple).
- Les thèmes d'algorithmique abordés au lycée sont presque toujours déjà implémentés dans une commande de `Xcas`, ceci permet de vérifier en comparant le résultat de la fonction qu'on vient de concevoir avec la commande interne.

`Xcas` accepte également l'écriture de programmes en syntaxe compatible

Python, mais en évitant certains pièges du langage par rapport aux mathématiques :

- on peut écrire `^` pour la puissance, `**` est également accepté,
- on peut écrire comme en mathématiques `2a` (inutile de mettre le signe `*`). Par contre il faut mettre le signe `*` dans `2a*(a+1)`,
- on peut travailler naturellement avec les rationnels, `1/2` renvoie une fraction, et non `0` (en Python 2) ou le nombre approché `0.5` (en Python 3),
- on peut aussi faire du calcul exact avec des racines carrées et du calcul littéral avec des variables formelles,
- le nombre complexe i est noté `i` et non `1j`, i^2 s'écrit `i^2` et non `1j**2` et donne comme réponse `-1` et non `-1+0j`,
- les listes servent à représenter des vecteurs, des matrices, des polynômes,
- toutes les commandes de `Xcas` sont accessibles, par exemple les commandes de géométrie analytique (ces commandes sont capables de faire des calculs, pas seulement de représentations graphiques) ou de calcul littéral,

On s'efforcera de donner dans ce document des programmes écrits avec les mots-clefs en français et leur traduction en syntaxe Python, les deux syntaxes sont valides dans `Xcas`. Lorsque la traduction Python n'est pas fournie, on peut l'obtenir de manière automatique depuis Xcas : on saisit le programme écrit en français, puis on l'exporte en texte Python.

Une partie des programmes de ce document écrits en syntaxe Python, peuvent être utilisés dans un interpréteur Python à condition d'y avoir chargé le module giacpy [2] par la commande `from giacpy import *` et d'avoir affecté quelques variables symboliques (par exemple pour `x` par `x=giac('x')`, car il n'y a pas de variables symboliques en Python)

1.3 Structure du document

Les deux premiers chapitres 2 et 3 sont destinés aux débutants, ils présentent les notions de variables, types fondementaux, les structures de programmation et les fonctions avec de nombreux exemples, en partie non mathématiques. Les chapitres suivants donnent de nombreux exemples d'algorithmes dans des thèmes mathématiques allant de la résolution d'équation, à la géométrie et aux statistiques. En annexe on trouvera une aide à l'installation et aux premiers pas avec Xcas, ainsi qu'une discussion plus avancée sur les divers langages de programmation interprétés.

2. `https://www-fourier.ujf-grenoble.fr/%7eparisse/giac_fr.html#python`

Chapitre 2

Types, fonctions

2.1 Types

2.1.1 Entiers, rationnels, réels, complexes et les nombres approchés.

Dans `Xcas` :

- les entiers sont des nombres de $\mathbb{Z}$, par exemple -2,
- les rationnels sont des nombres de $\mathbb{Q}$, par exemple 1/3,
- les nombres approchés sont des nombres décimaux[1], par exemple 3.14. **Attention**, le séparateur entre partie entière et partie fractionnaire est le point `.`
  On peut utiliser la notation scientifique mantisse `e` exposant pour entrer des nombres approchés, par exemple `1.2e-7` signifie `1.2` multiplié par 10^{-7}.
- les nombres réels sont représentés par des nombres décimaux ou par des valeurs symboliques, par exemple $\sqrt{2}$, π, e.
- les nombres complexes sont représentés par $a + ib$ pour a et b réels. Avec les commandes `re`, `im`, `arg` et `abs` pour avoir la partie réelle, la partie imaginaire, l'argument et le module.

Pour avoir une valeur approchée d'un nombre réel on utilise la commande `evalf`, par exemple `evalf(sqrt(2))` ou `evalf(sqrt(2),20)` pour avoir une valeur approchée de $\sqrt{2}$ avec 20 chiffres significatifs.

1. En toute rigueur ce sont des nombres écrits en base 2 et non en base 10 mais on peut l'ignorer au niveau du lycée

```
evalf(sqrt(2))
```

$$1.41421356237$$

```
evalf(sqrt(2),20)
```

$$1.4142135623730950488$$

Pour avoir l'écriture d'un nombre complexe sous la forme $a + ib$ avec a et b réels on utilise la commande `evalc`, par exemple `evalc((1+i*sqrt(2))^2+2)`.

```
(1+i*sqrt(2))^2+2
```

$$\left(1 + i \cdot \sqrt{2}\right)^2 + 2$$

```
evalc((1+i*sqrt(2))^2+2)
```

$$1 + i \cdot 2 \cdot \sqrt{2}$$

2.1.2 Les listes, les séquences et les chaînes de caractères

Définition d'une liste

Qu'est-ce qu'une **liste** ?
C'est une énumération d'objets, dont l'ordre est important. Cela peut servir à représenter les coordonnées d'un point ou d'un vecteur, à contenir une liste de valeurs (observations) en statistiques, ...
Une liste est délimitée par des crochets `[]` et les éléments de la liste sont séparés par une virgule `,`

```
L:=[2,24,1,15,5,10]
```

$$[2, 24, 1, 15, 5, 10]$$

Création d'une liste

– `range`
`range(n)` renvoie la liste `[0,1..n-1]`

`range(a,b)` renvoie la liste `[a,a+1..b[`
`range(a,b,p)` la liste `[a,a+p,a+2p..b[` à condition que `p` soit un entier.

```
range(5)
```

$$[0, 1, 2, 3, 4]$$

```
range(5,11)
```

$$[5, 6, 7, 8, 9, 10]$$

```
range(11,5,-1)
```

$$[11, 10, 9, 8, 7, 6]$$

```
range(1,11,3)
```

$$[1, 4, 7, 10]$$

– `seq` renvoie la liste obtenue lorsqu'on calcule une expression dépendant d'un paramètre `var` qui varie entre `a` et `b` (on peut ajouter un argument optionnel, le pas `p`.
`seq` a 4 ou 5 arguments)
`seq(2^k,k,0,8)`

$$[1, 2, 4, 8, 16, 32, 64, 128, 256]$$

`seq(2^k,k,0,8,2)`

$$[1, 4, 16, 64, 256]$$

`seq(2^k,k,0,8,1/2)`

$$[1, \sqrt{2}, 2, \sqrt{2}\cdot 2, 4, \sqrt{2}\cdot 2^2, 8, \sqrt{2}\cdot 2^3, 16, \sqrt{2}\cdot 2^4, 32, \sqrt{2}\cdot 2^5, 64, \sqrt{2}\cdot 2^6, 128, \sqrt{2}\cdot 2^7, 256]$$

– `makelist` renvoie une liste faite à partir d'une fonction ou d'une constante.

```
makelist(x->x^2,1,10)
```

$$[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]$$

```
makelist(x->x^2,1,10,2)
```

$$[1, 9, 25, 49, 81]$$

```
makelist(x->x^2,1,10,1/2)
```

$$[1, \frac{9}{4}, 4, \frac{25}{4}, 9, \frac{49}{4}, 16, \frac{81}{4}, 25, \frac{121}{4}, 36, \frac{169}{4}, 49, \frac{225}{4}, 64, \frac{289}{4}, 81, \frac{361}{4}, 100]$$

Définition d'une séquence

Qu'est-ce qu'une **séquence** ?
On peut le voir comme une liste sans crochets, ce qui signifie que si on sépare 2 séquences par une virgule, elles sont concaténées, on ne peut donc pas créer une séquence de séquences alors qu'on peut créer une liste de listes. Par exemple les arguments d'une fonction sont regroupés en une séquence.
Une séquence n'est pas délimitée (ou est délimitée par des parenthèses `()`) et les éléments de la séquence sont séparés par une virgule `,`

```
S:=(2,24,1,15,5,10)
```

$$2, 24, 1, 15, 5, 10$$

Attention,
cette syntaxe renvoie un n-uplet en Python, il s'agit d'une liste constante (non modifiable après création).

Création d'une séquence

– `seq` renvoie la séquence obtenue lorsque `var` varie entre `a` et `b` (avec un argument optionnel, le pas=`p`). `seq` a 2 ou 3 arguments.

```
seq(2^k,k=0..8)
```

$$1, 2, 4, 8, 16, 32, 64, 128, 256$$

```
seq(2^k,k=0..8,2)
```

$$[1, 4, 16, 64, 256]$$

```
seq(2^k,k=0..8,1/2)
```

$$[1, \sqrt{2}, 2, \sqrt{2}\cdot 2, 4, \sqrt{2}\cdot 2^2, 8, \sqrt{2}\cdot 2^3, 16, \sqrt{2}\cdot 2^4, 32, \sqrt{2}\cdot 2^5, 64, \sqrt{2}\cdot 2^6, 128, \sqrt{2}\cdot 2^7, 256]$$

– $ est la version infixée de `seq`

```
2^k$10
```

$$2^k, 2^k, 2^k, 2^k, 2^k, 2^k, 2^k, 2^k, 2^k, 2^k$$

```
2^k$(k=0..8)
```

$$1, 2, 4, 8, 16, 32, 64, 128, 256$$

```
2^k$(k=0..8,2)
```

$$1, 4, 16, 64, 256$$

```
2^k$(k=0..8,1/2)
```

$$1, \sqrt{2}, 2, \sqrt{2}\cdot 2, 4, \sqrt{2}\cdot 2^2, 8, \sqrt{2}\cdot 2^3, 16, \sqrt{2}\cdot 2^4, 32, \sqrt{2}\cdot 2^5, 64, \sqrt{2}\cdot 2^6, 128, \sqrt{2}\cdot 2^7, 256$$

Transformation d'une séquence en liste et vice-versa

Si `S` est une séquence alors `[S]` est une liste.
Si `L` est une liste alors `op(L)]` est une séquence.

```
L:=[2,24,1,15,5,10]
```

$$[2, 24, 1, 15, 5, 10]$$

```
S:=(2,24,1,15,5,10)
```

$$2, 24, 1, 15, 5, 10$$

```
op(L)
```

$$2, 24, 1, 15, 5, 10$$

```
[S]
```

$$[2, 24, 1, 15, 5, 10]$$

Définition d'une chaine de caractères

Qu'est-ce qu'une **chaine de caractères** ?
C'est la concaténation de 0, 1 ou plusieurs caractères.
Une chaine de caractères est délimitée par `""`
On concaténe de 2 chaînes avec `+`

```
s:="Bonjour"
```

Bonjour

```
s:="";s:=s+"Bon"+"jour"
```

, Bonjour

2.1.3 Les instructions sur les listes les séquences et les chaînes de caractères

- `dim(L)` ou `len(L)` renvoie le nombre d'éléments de la liste `L`.
  `dim(S)` ou `len(S)` renvoie le nombre d'éléments de la séquence `S`.
  `dim(s)` ou `len(s)` renvoie le nombre de caractères de la chaîne `s`.
- `[]` représente la liste vide et `dim([])` vaut `0`.
  `NULL` représente la séquence vide et `dim(NULL)` vaut `0`.
  `""` représente la chaîne vide et `dim("")` vaut `0`.
- Les éléments de la liste sont numérotés de 0 jusque `dim(L)-1`.
  `L|0]` désigne le premier élément de la liste et `L[dim(L)-1]` ou `L[-1]` désigne le dernier élément de la liste `L`.
  Les éléments de la séquence sont numérotés de 0 jusque `dim(S)-1`.
  `S|0]` désigne le premier élément de la séquence et `S[dim(S)-1]` ou `S[-1]` désigne le dernier élément de la séquence.
  `s|0]` désigne le premier caractère de la chaîne et `s[dim(s)-1]` désigne

le dernier caractère de la chaîne `S`.

- `gauche(L,n)` renvoie les `n` premiers éléments de la liste `L` (c'est le côté gauche de la liste).
  `droit(L,n)` renvoie les `n` derniers éléments de la liste `L` (c'est le côté droit de la liste).
  `gauche(s,n)` renvoie les `n` premiers caractères de la chaîne `s` (c'est le côté gauche de la chaîne).
  `droit(s,n)` renvoie les `n` derniers caractères de la chaîne `s` (c'est le côté droit de la chaîne).
  Par exemple : soient la liste `L:=[2,24,1,15,5,10]` et la chaîne `s:="Bonjour"`.
  `L1:=seq(2*k,k,0,4)` crée la liste `[0,2,4,6,8]`
- `count_eq, count_inf, count_sup, count` : nombre d'éléments égaux, inférieurs, supérieurs ou vérifiant une condition
  `count_sup(2,L1)` ou `count(x->x>2,L1)` compte le nombre d'éléments de `L1` supérieurs à 2.
  `S1:=seq(2*k,k=0..4)` crée la séquence `(0,2,4,6,8)`
  `count(x->x>2,S1)` compte le nombre d'éléments de `S1` supérieurs à 2.
- `sum(L)` renvoie la somme des éléments de la liste `L`.
  `sum(S)` renvoie la somme des éléments de la séquence `S`.
- Quelques exemples :

```
L:=[2,18,1,9,6,54]
```

$$[2, 18, 1, 9, 6, 54]$$

```
n1:=dim(L)
```

$$6$$

```
L[0]
```

$$2$$

```
L[1];L[2]
```

$$18, 1$$

```
L[n1-1]
```

$$54$$

```
gauche(L,2)
```

$$[2, 18]$$

```
droit(L,2)
```

$$[6, 54]$$

```
sum(L)
```

$$90$$

```
S:=(2,18,1,9,6,54)
```

$$2, 18, 1, 9, 6, 54$$

```
n1:=dim(S)
```

$$6$$

```
S[0]
```

$$2$$

```
S[1];S[2]
```

$$18, 1$$

```
S[n1-1]
```

$$54$$

```
sum(S)
```

$$90$$

```
S:=S,3,27
```

$$2, 18, 1, 9, 6, 54, 3, 27$$

```
S:=S,(4,36)
```

$$2, 18, 1, 9, 6, 54, 3, 27, 4, 36$$

```
S:=S,[5,45]
```

$$2, 18, 1, 9, 6, 54, 3, 27, 4, 36, [5, 45]$$

```
s:="Bonjour"
```

Bonjour

```
n2:=dim(s)
```

$$7$$

```
s[0]
```

$$B$$

```
s[1];s[2]
```

$$o, n$$

```
s[n2-1]
```

$$r$$

```
gauche(s,3)
```

Bon

```
droit(s,4)
```

jour

– `append`
`append(L,a)` renvoie la liste où l'élément `a` à été rajouté à la fin de la liste `L`.
On tape :

```
append([1,2,3],4)
```

$$[1, 2, 3, 4]$$

```
L:=[1,2,3]; append(L,4); L
```

$$[1, 2, 3], [1, 2, 3, 4], [1, 2, 3]$$

Attention
Pour ajouter `4` à la fin de la liste `L` on tape :

```
L:=[1,2,3]; L:=append(L,4); L
```

$$[1,2,3],[1,2,3,4],[1,2,3,4]$$

ou comme en Python

```
L:=[1,2,3]; L.append(4); L
```

$$[1,2,3],[1,2,3,4],[1,2,3,4]$$

– Il est facile d'ajouter un élément à une séquence ou de concaténer 2 séquences.
Supposons que `S` soit la séquence `(1,2,3)`(`S:=1,2,3`) et que `L` soit la liste `[1,2,3]` (`L:=[1,2,3]`).
Pour modifier une séquence il suffit par exemple, décrire :
`S:=S,4` (resp `S:=S,4,5,6` ou `S:=S,(4,5,6)`) pour que `S` soit la séquence `(1,2,3,4)` (resp `(1,2,3,4,5,6)`
alors que pour modifier une liste il faut écrire :
`L:=append(L,4)` (resp `L:=append(L,4,5,6)` ou `L:=concat(L,4,5,6)` ou `L:=concat(L,[4,5,6])`) pour que `L` soit la liste `[1,2,3,4]` (resp `[1,2,3,4,5,6]`).
Mais attention !
`L:=append(L,[4,5,6])` renvoie `[1,2,3,[4,5,6]]`
Pour les chaînes de caractères, supposons que `s` soit la chaîne `"abc"` (`s:="abc"`), on peut écrire pour lui rajouter `"def"` :
`s:=append(s,"def")` ou `s:=concat(s,"def")` ou `s:=s+"def"`)

```
s:="abc";s:=s+"def"
```

$$\text{abc}, \text{abcdef}$$

– `prepend`
`prepend(L,a)` renvoie la liste où l'élément `a` à été rajouté au début de la liste `L`.
On tape :

```
prepend([1,2,3],0)
```

$$[0,1,2,3]$$

```
L:=[1,2,3]; prepend(L,0); L
```

$$[1,2,3],[0,1,2,3],[1,2,3]$$

Attention
Pour ajouter `0` au début de la liste `L` on tape :

```
L:=[1,2,3]; L:=prepend(L,0); L
```

$$[1,2,3],[0,1,2,3],[0,1,2,3]$$

ou

```
L:=[1,2,3]; L.prepend(0); L
```

$$[1,2,3],[0,1,2,3],[0,1,2,3]$$

– `concat`
`concat(L1,L2)` renvoie la concaténation des deux listes `L1` et `L2)`.
On tape :

```
concat([1,2,3],[4,5,6,7])
```

$$[1,2,3,4,5,6,7]$$

```
L:=[1,2,3]; concat(L,[4,5,6,7]); L
```

$$[1,2,3],[1,2,3,4,5,6,7],[1,2,3]$$

Attention
Pour ajouter `4,5,6,7` à la fin de la liste `L` on tape :

```
L:=[1,2,3]; L:=concat(L,[4,5,6,7]); L
```

$$[1,2,3],[1,2,3,4,5,6,7],[1,2,3,4,5,6,7]$$

ou

```
L:=[1,2,3]; L.concat([4,5,6,7]); L
```

$$[1,2,3],[1,2,3,4,5,6,7],[1,2,3,4,5,6,7]$$

– `rotate`
`rotate(L)` (resp `rotate(L,n)`) renvoie la liste obtenue en mettant le dernier élément (resp la fin de la liste à partir du `n`-ième élément) en premier (par défaut n=-1).
On tape :

```
rotate([0,1,2,3])
```

$$[3,0,1,2]$$

```
L:=[0,1,2,3]; rotate(L)
```

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

Attention
Pour modifier le contenu de `L`, on tape :

```
L:=[0,1,2,3]; L:=rotate(L); L
```

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

ou

```
L:=[0,1,2,3]; L.rotate(); L
```

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

On tape :

```
rotate([0,1,2,3],2)
```

$$[2,3,0,1]$$

```
L:=[0,1,2,3]; rotate(L,2); L
```

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

Attention
Pour modifier le contenu de `L`, on tape :

```
L:=[0,1,2,3];L:=rotate(L,2);L
```

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

ou

```
L:=[0,1,2,3]; L.rotate(2);L
```

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

– `shift`
`shift(L,n)` : si `n>0` on remplace les `n` premiers éléments de la liste par des `0` et on renvoie ces `0` à la fin de la liste,
si `n<0` renvoie la liste obtenue en remplaçant les `-n` derniers éléments de la liste par `0` et on renvoie ces `0` au début de la liste, puis on renvoie la liste ainsi obtenue. (par défaut n=-1 ce qui signifie que le dernier élément de la liste est supprimé et on met un `0` au début de la liste).
On tape :

```
shift([1,2,3,4],-1),shift([1,2,3,4])
```

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

```
L:=[0,1,2,3];shift(L);shift(L,-2)
```

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

```
shift([1,2,3,4],-2);shift([1,2,3,4],2)
```

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

```
L:=[0,1,2,3];shift(L,2)
```

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

Attention
Pour modifier le contenu de `L`, on tape :

```
L:=[0,1,2,3];L:=shift(L);
```

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

ou

```
L:=[0,1,2,3];L.shift();L
```

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

Pour modifier le contenu de `L`, on tape :

```
L:=[0,1,2,3];L:=shift(L,-2);L
```

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

ou

```
L:=[0,1,2,3];L.shift(-2);L
```

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

Pour modifier le contenu de `L`, on tape :

```
L:=[0,1,2,3];L:=shift(L,2);L
```

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

ou

```
L:=[0,1,2,3];L.shift(2);L
```

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

– `suppress`
`suppress(L,n)` renvoie `L` sans son élément d'indice `n`.
On tape (attention les indices commencent à 0) :

```
suppress([0,1,2,3],2)
```

$$[0, 1, 3]$$

```
L:=[0,1,2,3];suppress(L,2);L
```

$$[0, 1, 2, 3], [0, 1, 3], [0, 1, 2, 3]$$

Attention
Pour modifier le contenu de `L`, on tape :

```
L:=[0,1,2,3];L:=suppress(L,2);L
```

$$[0, 1, 2, 3], [0, 1, 3], [0, 1, 3]$$

ou

```
L:=[0,1,2,3];L.suppress(2);L
```

$$[0,1,2,3],[0,1,3],[0,1,3]$$

– `reverse`
`reverse(L)` ou `revlist(L)` renvoie la liste `L` inversée.
On tape :

```
reverse([1,2,3])
```

$$[3,2,1]$$

```
L:=[1,2,3];reverse(L);L
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \\ 1 & 2 & 3 \end{pmatrix}$$

Attention
Pour modifier le contenu de `L`, on tape :

```
L:=[1,2,3];L:=reverse(L);L
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \\ 3 & 2 & 1 \end{pmatrix}$$

ou

```
L:=[1,2,3];L.reverse();L
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \\ 3 & 2 & 1 \end{pmatrix}$$

2.1.4 Les booléens

Définition

L'ensemble des booléens est un ensemble à 2 éléments :
`vrai` ou `1` et `faux` ou `0`.
Pour faire des tests, on utilise des opérateurs booléens.

Opérateur booléen infixé qui teste l'égalité ==

Pour tester l'égalité entre deux objets (réels, entiers, expressions, listes..) on utilise l'opérateur `==`. Le test renvoie vrai si les deux objets sont égaux, au sens où leur représentation informatique est identique. Pour tester que deux expressions littérales contenues dans les variables `a` et `b` sont mathématiquement équivalentes, il est parfois nécessaire de faire une commande explicite de simplification, au lieu d'écrire `a==b` on écrit `simplify(a-b)==0`.

Exemple :

```
a:=3;b:=5
```

$$3, 5$$

```
b-a==0
```

$$faux$$

```
b-a-2==0
```

$$vrai$$

```
(1+sqrt(2))^2==3+2*sqrt(2);simplify((1+sqrt
(2))^2-3-2*sqrt(2))==0;
```

$$faux, vrai$$

Les différentes significations des signes :=, =,==, #=

Attention

– Il ne faut pas confondre les différentes significations du signe =
Le signe := sert à stocker une valeur dans une variable, le signe = sert à définir une équation (ou à donner la valeur par défaut d'un argument, cf. la section 2.2.4).
Le signe == sert à tester l'égalité. C'est un opérateur booléen infixé. Il renvoie `vrai` ou `faux`.

`Xcas` accepte toutefois = dans certaines situations non ambigües (affectation ou test d'égalité), par exemple :
`a=1;si a=2 alors vrai sinon faux fsi;a;` renvoie `1=1,faux,1`.
Remarque
En Python l'affectation se fait avec le signe =, car Python ne gére pas les équations, c'est pourquoi dans `Xcas` en syntaxe compatible Python, on utilisera le signe #= pour définir une équation. Dans un interpréteur Python avec `giacpy`, il faut utiliser la commande préfixée `equal`.

– `Xcas` travaille par défaut avec des nombres exacts donc :
`3+10^(-16)==3` renverra `faux` ou `false` car il compare 2 nombres rationnels alors qu'avec un logiciel qui ne fait pas de calcul formel, il se peut que `3+10^(-16)==3` renvoie `vrai` ou `true` car il compare 2 nombres flottants i.e. 2 nombres approchés.

Opérateur booléen infixé qui teste la non égalité !=

Le signe != sert à tester la non égalité.
C'est un opérateur booléen infixé. Il renvoie `vrai` ou `faux`.
Exemple :

```
a:=3;b:=5
```

$$3, 5$$

```
b!=a
```

$$vrai$$

```
a+2!=b
```

$$faux$$

Opérateur booléen infixé qui teste l'inégalité `<, >, <=, >=`

Les signes `<, >, <=, >=` servent à tester les inégalités.
Ce sont des opérateurs booléens infixés. Ils renvoient `vrai` ou `faux`.
Exemple :

```
a:=3;b:=5
```

$$3, 5$$

```
b >= a
```

$$vrai$$

```
a+2 > b
```

$$faux$$

```
a+2 >= b
```

$$vrai$$

2.1.5 Expressions, polynômes

Simplification d'une expression avec `normal`

`Xcas` renvoie le résultat d'un calcul littéral sans le simplifier (sauf pour les calculs dans les rationnels).
Il faut utiliser la fonction `normal` ou `simplify` pour obtenir un résulat simplifié.

```
purge(x)
```

Nosuchvariablex

```
(x^2+1)^2+(x^2-1)^2
```

$$\left(x^2+1\right)^2+\left(x^2-1\right)^2$$

```
normal((x^2+1)^2+(x^2-1)^2)
```

$$2\cdot x^4+2$$

Les polynômes

Un polynôme à une indéterminée à coefficients dans $\mathbb{R}$ est déterminé par une séquence $a_n, ..., a_1, a_0$ d'éléments de $\mathbb{R}$, c'est l'expression :
$a_n x^n + ... + a_1 x + a_0$ (ou $a_0 + a_1 x + a_2 x^2 + .. + a_n x^n$).
n est le degré du polynôme.
On dit que l'on a écrit le polynôme selon les puissances décroissantes (ou croissantes).
$a_n, ..a_1, a_0$ sont les coefficients du polynôme et x est la variable ou l'indéterminée du polynôme.
On notera l'ensemble des polynômes à une indéterminée x : $\mathbb{R}[x]$.

Un polynôme à 2 indéterminées x et y à coefficients dans $\mathbb{R}$ est déterminé par une séquence $A_n(y), ..., A_1(y), A_0(y)$ d'éléments de $\mathbb{R}[y]$ et a pour expression :
$A_n(y)x^n + ... + A_1(y)x + A_0(y)$ (ou $A_0(y) + A_1(y)x + A_2(y)x^2 + .. + A_n(y)x^n$)
Par exemple :
si $A_0(y) = y^3 - 2, A_1(y) = -2y, A_2(y) = y^3 + 2 * y + 3$
Le polynôme s'écrit :
$y^3-2-2y*x+(y^3+2*y+3)*x^2 = x^2*y^3+2*x^2*y+3*x^2-2*x*y+y^3-2$
Le degré par rapport à x du polynôme de cet exemple et égal à 2.
Le degré par rapport à y du polynôme de cet exemple et égal à 3.

Coefficients et degré d'un polynôme

`Xcas` représente les polynômes soit sous la forme d'une expression symbolique, soit comme la séquence des coefficients selon les puissances décroissantes (`poly1[1,2,3]` i.e. `[a2,a1,a0]:=[1,2,3]`).
Par exemple, le polynôme $x^2 + 2x + 3$ s'écrit :
soit `x^2+2x+3` où `x` est l'indéterminée mais **Attention** `x` doit être une variable symbolique donc `x` doit être purgée,

soit `poly1[1,2,3]`
Les commandes `symb2poly(x^2+2x+3)` et `poly2symb([1,2,3],x)` passent d'une représentation à l'autre.

- Pour avoir le degré d'une expression polynômiale par rapport à une variable, on utilise l'instruction `degree` qui renvoie le degré d'un polynôme (on peut ajouter un 2ième argument si le polynôme est symbolique en une indéterminée qui n'est pas `x`).
- Pour avoir les coefficients d'un polynôme par rapport à une variable on utilise l'instruction `symb2poly` qui renvoie la liste des coefficients d'un polynôme par rapport au 2ième argument (`x` est la variable par défaut). Ainsi si `L:=symb2poly(P(x))`, `degree(P(x))` est égal à `dim(L)-1`.
  La commande `poly2symb` effectue le calcul inverse, ainsi :
  `symb2poly(y^2+2y+3)` renvoie `poly1[1,2,3]`.
  `poly2symb([1,2,3],y)` renvoie `(y+2)*y+3`.
- Pour avoir le coefficient de degré donné d'un polynôme par rapport à une variable, on utilise l'instruction `coeff`.
- Exemples :
  `purge(x,y):;degree(x^2-2x*y+y^3-2)`

Done, 2

`degree(x^2-2x*y+y^3-2,x)`

2

`degree(x^2-2x*y+y^3-2,y)`

3

`symb2poly(x^2-2x*y+y^3-2)`

$$poly1[1, -2 \cdot y, y^3 - 2]$$

`symb2poly(x^2-2x*y+y^3-2,x)`

$$poly1[1, -2 \cdot y, y^3 - 2]$$

```
symb2poly(x^2-2x*y+y^3-2,y)
```

$$poly1[1, 0, -2 \cdot x, x^2 - 2]$$

```
coeff(x^2-2x*y+y^3-2,x,2)
```

$$1$$

```
coeff(x^2-2x*y+y^3-2,y,1)
```

$$-2 \cdot x$$

2.1.6 Connaître les types et les sous-types

Les types

`Xcas` sait reconnaître le type d'un objet.
Pour avoir le type de l'objet `a` ou le contenu d'une variable `a`, on utilise `type(a)`.
Les types utilisés ici sont :
`real` ou `1` ce qui signifie que `a` contient un nombre flottant.

```
a:=3.14
```

$$3.14$$

```
type(a)
```

$$real$$

```
type(a)+0
```

$$1$$

`integer` ou `2` ce qui signifie que `a` contient un nombre entier,

```
a:=2017
```

$$2017$$

```
type(a)
```

$$integer$$

```
type(a)+0
```

$$2$$

`rational` ou `10` ce qui signifie que `a` contient un nombre rationnel,

```
a:=2/7
```

$$\frac{2}{7}$$

```
type(a)
```

$$rational$$

```
type(a)+0
```

$$10$$

`func` ou `13` ce qui signifie que `a` est le nom d'une fonction,

```
f(x):=2x+3
```

```
 (x)->2*x+3
```

```
type(f)
```

$$func$$

```
type(sin)
```

$$func$$

```
type(sin)+0
```

$$13$$

`vecteur` ou `7` ce qui signifie que `a` contient une liste,

```
L:=[2,24,1,15,5,10]
```

$$[2, 24, 1, 15, 5, 10]$$

```
type(L)
```

$$vecteur$$

```
type(L)+0
```

$$7$$

`string` ou `12` ce qui signifie que `a` contient une chaîne de caractères,

```
s:="Bonjour"
```

$$\text{Bonjour}$$

```
type(s)
```

string

```
type(s)+0
```

12

`expression` ou `8` ce qui signifie que `a` contient une expression,

```
a:=sqrt(2)
```

$$\sqrt{2}$$

```
type(a)
```

expression

```
type(a)+0
```

8

```
type(f(x))
```

expression

```
type(f(x))+0
```

8

`identifier` ou `6` ce qui signifie que `a` contient le nom d'une variable non

affectée.

```
purge(a)
```

$$\sqrt{2}$$

```
type(a)
```

$$identifier$$

```
type(a)+0
```

6

Les sous-types

Certains types de variables peuvent servir à plusieurs usages : par exemple une liste peut représenter les coordonnées d'un point dans l'espace ou les coefficients d'un polynôme ou un ensemble. `Xcas` possède une commande `subtype` permettant de préciser le type d'une variable. Pour avoir le sous-type de la variable `a`, on utilise `subtype(a)`.
Par exemple si `a` contient une liste, `subtype(a)` renvoie `1` pour une séquence, `2` pour un ensemble, `10` pour un polynôme et `0` sinon.

```
subtype(1,2,3)
```

1

```
subtype(set[1,2,3])
```

2

```
subtype(poly1[1,2,3])
```

10

```
subtype([1,2,3])
```

$$0$$

2.2 Les fonctions

On distingue les fonctions ou commandes de `Xcas` et les fonctions définies par l'utilisateur. Pour éviter le risque d'utiliser un nom de fonction de `Xcas`, il est conseillé de nommer les fonctions utilisateurs en utilisant une majuscule comme première lettre. Pour définir des fonctions (utilisateurs), on distinguera :

- les fonctions définies par une expression algébrique. Leur définition peut se faire simplement avec `:=`
- les fonctions qui nécessitent des calculs intermédiaires ou des structures de contrôle (test, boucle). Leur définition se fait au moyen d'un programme, en utilisant les instructions `fonction...ffonction`, `local` et `retourne` et les structures de contrôle qui seront détaillées au chapitre 3 [2], par exemple : `"bon"`.

2.2.1 Quelques fonctions algébriques de `Xcas`

`abs` est la fonction valeur absolue.

```
abs(sqrt(2)-3/2)
```

$$-\sqrt{2}-\frac{-3}{2}$$

`ceil` est le plus grand entier >= à l'argument.
`cos` est la fonction cosinus.

```
cos(pi/6)
```

$$\frac{\sqrt{3}}{2}$$

`floor` est la partie entière i.e.le plus grand entier <= à l'argument.

2. On peut utiliser la syntaxe à la `Python` au sein de `Xcas`, mais il faut déclarer les variables locales avec `# local` (bien mettre un espace entre `#` et `local`) et n'utiliser que `"` (et non `'`) comme délimiteur de chaînes de caractères.

```
floor(-2.42)
```

$$-3$$

```
floor(2.42)
```

$$2$$

`frac` est la partie fractionnaire d'un réel.

```
frac(3/2)
```

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

```
frac(2.42)
```

$$0.42$$

`max` est la fonction maximum pour une séquence de nombres réels.

```
max(sqrt(2),3,5,sqrt(26))
```

$$\sqrt{26}$$

`min` est la fonction minimum pour une séquence de nombres réels.

```
min(sqrt(2),3,5,sqrt(26))
```

$$\sqrt{2}$$

`^` est la fonction puissance.

```
2^10
```

$$1024$$

`round` est la fonction qui arrondit un réel en l'entier (resp le décimal) le plus proche.

```
round(-2.42)
```

$$-2$$

```
round(-2.42,1)
```

$$-2.4$$

```
round(sqrt(2),4)
```

$$1.4142$$

`sign` est la fonction signe de l'argument et renvoie -1, 0 ou +1.

```
sign(-4)
```

$$-1$$

```
sign(4-4)
```

$$0$$

```
sign(4)
```

$$1$$

`sin` est la fonction sinus.

```
sin(pi/3)
```

$$\frac{\sqrt{3}}{2}$$

`sqrt` est la racine carrée.

```
sqrt(2)^2
```

$$2$$

```
(sqrt(3)+1)*(sqrt(3)-1)
```

$$(\sqrt{3}+1)(\sqrt{3}-1)$$

```
normal((sqrt(3)+1)*(sqrt(3)-1))
```

$$2$$

`tan` est la fonction tangente.

```
tan(pi/4)
```

$$1$$

2.2.2 Quelques fonctions aléatoires de Xcas

Nous donnons ci-après les fonctions aléatoires les plus fréquentes : pour des raisons de compatibilité avec le langage `Python` chacune des fonctions ci-dessous peuvent être préfixée par `random.` (par exemple on peut écrire `random.randint()` ou `randint()`)

```
srand randseed RandSeed
```

`srand` ou `randseed` initialise la suite des nombres aléatoires : `srand` ou `randseed`renvoie un entier qui a servi à cette initialisation.
On tape :

`srand`

657599774

`RandSeed`

`RandSeed(n)` initialise les nombres aléatoires selon la valeur de l'entier `n`.
On tape :

```
RandSeed(321)
```

321

`random rand`

`random()` ou `rand()` renvoie un nombre réel (pseudo)-aléatoire de l'intervalle semi-ouvert $[0, 1[$.
On tape :

```
random()
```

0.603756384458

`choice`

`choice(L)` ou `random(L)` ou `rand(L)` renvoie un élément tiré au hasard parmi les éléments de la liste `L`.
On tape :

```
choice([1,2,3,4,5,6])
```

4

`randint`

`randint(a,b)` renvoie un nombre entier (pseudo)-aléatoire compris entre `a` et `b` (bornes incluses).
On tape :

```
randint(1,10)
```

7

`shuffle`

`shuffle(L)` applique une permutation aléatoire à la liste `L`.
On tape :

```
shuffle([1,2,3,4,5,6])
```

$$[3,6,1,4,2,5]$$

`sample` ou `rand`

`sample(L,n)` ou `rand(n,L)` renvoie, si `n<=size(L)`, `n` éléments tirés, au hasard, sans remise parmi les éléments de la liste `L` et sinon renvoie une erreur.
On tape :

```
sample([1,2,3,4,5,6,7,8,9,10],3)
```

$$[3,2,10]$$

Remarque `sample(L,len(L))` ou `rand(dim(L),L)` est identique à `shuffle(L)`.

On tape en syntaxe `Xcas` :

```
fonction testrand()
  local L1,L2,L3,n1,n2,a1,a2;
  n1:=random();
  n2:=random();
  a1:=randint(1,10);
  a2:=randint(1,10)
  L1:=sample([1,2,3,4,5,6],3);
  L2:=sample([1,2,3,4,5,6],3);
  L3:=shuffle([1,2,3,4,5,6]);
  return n1,n2,a1,a2,L1,L2,L3;
ffonction:;
```

```
testrand()
```

$$0.984837350901, 0.310133864637, 3, 9, [4,5,1], [1,3,5], [4,2,1,6,3,5]$$

On tape en syntaxe Python :

```
def testrand() :
    # local L1,L2,L3,n1,n2,a1,a2
    n1=random.random()
    n2=random.random()
    a1=random.randint(1,10)
    a2=randint(1,10)
    L1=random.sample([1,2,3,4,5,6],3)
    L2=random.sample([1,2,3,4,5,6],3)
```

```
    L3=shuffle([1,2,3,4,5,6])
    return n1,n2,a1,a2,L1,L2,L3

  testrand()
```

$0.171089087613, 0.105508357286, 10, 2, [2, 4, 5], [4, 3, 1], [3, 2, 4, 5, 6, 1]$

2.2.3 Définition d'une fonction algébrique d'une variable

Exemple :
On veut définir la fonction f_1 définie pour $x \in \mathbb{R}$, par $f_1(x) = x^2 + 1$.
f est le nom de la fonction et x est le nom de l'argument de f_1 (ici x est un réel), la valeur de la fonction f_1 est $x^2 + 1$.
Remarque : En mathématique on dit que x est une variable.
En syntaxe `Xcas`, on tape simplement :

```
  f1(x):=x^2+1;
```

```
    (x)->x^2+1
```

On pourrait aussi définir `f1` par un programme avec `fonction...ffonction` et `retourne` :

```
  fonction f1(x) retourne x^2+1; ffonction;
```

```
    (x)->return(x^2+1)
```

En syntaxe Python

```
def f1(x):
    return x^2+1

  f1(5)
```

2.2.4 Définition d'une fonction algébrique de 2 variables

Exemple :
On veut définir la fonction f_2 définie pour $(x, y) \in \mathbb{R}^2$, par $f_2(x, y) = x^2 + y^2$.
f_2 est le nom de la fonction et f_2 a 2 arguments réels : x, y (en mathématique on dit que x, y sont les variables de f_2). La valeur de la fonction f_2 est $x^2 + y^2$.
On tape simplement :

```
f2(x,y):=x^2+y^2
```

```
(x,y)->x^2+y^2
```

Ou on tape :

```
fonction f2(x,y)  retourne x^2+y^2; ffonction
```

```
(x,y)->return(x^2+y^2)
```

Valeur par défaut d'une variable
On peut définir la **valeur par défaut** de la variable `y` et ainsi donner le même nom f aux 2 fonctions f_1 et f_2 définies précédement.
Ainsi la fonction f sera définie par :
si elle a une variable x alors $f(x) = x^2 + 1$ et
si elle a 2 variables x, y alors $f(x, y) = x^2 + y^2$.
En syntaxe `Xcas`, on tape alors simplement :

```
f(x,y=1):=x^2+y^2
```

```
(x,y=1)->x^2+y^2
```

Ou on tape :

```
fonction f(x,y=1) retourne x^2+y^2;ffonction
```

```
(x,y=1)->return(x^2+y^2)
```

```
f(3,4)
```

25

```
f(3)
```

10

En syntaxe Python :

```
def f(x,y=1) :
    return x^2+y^2;
```

```
f(3,4)
```

25

```
f(3)
```

10

Autre exemple :
On veut définir la fonction g définie pour $(a, b) \in \mathbb{N}^2$ par $g(a, b) = q, r$ où q, r désigne le quotient et le reste de la division euclidienne de a par b.
On tape simplement :

```
g(a,b):=iquorem(a,b);
```

```
 (a,b)->iquorem(a,b)
```

ou bien avec `fonction...ffonction` et `retourne` :

```
fonction g(a,b) retourne iquorem(a,b); ffonction;
```

```
 (a,b)->return(iquorem(a,b))
```

g est le nom de la fonction, a, b sont les noms des arguments de g (a et b sont des entiers) et `iquorem` renvoie le quotient et le reste de la division euclidienne de a par b sous la forme d'une liste.
On a aussi les instructions :
`iquo(a,b)` qui renvoie le quotient de la division euclidienne de a par b.
`irem(a,b)` qui renvoie le reste de la division euclidienne de a par b.
et on a donc `iquorem(a,b)` est identique à `[iquo(a,b),irem(a,b)]`.

```
g(25,3)
```

$$[8, 1]$$

2.2.5 Définition d'une fonction algébrique sans lui donner un nom

On peut définir une fonction sans la nommer, cela permet de la passer en argument d'une autre commande (par exemple une commande de recherche de racines par dichotomie).
En syntaxe `Xcas`
Par exemple pour définir la fonction $x- > x^2 + 1$, on écrit simplement :
`x->x^2+1`.

```
x->x^2+1
```

```
(x)->x^2+1
```

On tape pour avoir la valeur de cette fonction en $x = 5$:
(x−>x^|2+1)(5)
`(x->x^2+1)(5)`
Si on veut donner un nom à la fonction, on tape :
`f(x):=x^2+1`
puis, on tape :

```
f(5)
```

26

En syntaxe à la `Python`
Par exemple pour définir la fonction $x- > x^2 + 1$, on écrit :
`lambda x : x*x+1`.

Le logiciel `Xcas` le traduit en $x->x^2+1$.
On tape :

```
(lambda x  : x*x+1)(5)
```

26

Si on veut donner un nom à la fonction, on tape :

```
def f(x):
    return x*x+1
```

puis, on tape :

```
f(5)
```

26

Remarque : il n'y a pas de risques de confusion !
Si vous avez une variable `lambda` qui est affectée, l'écriture : `(lambda x : x*x+1)(5)`, ne modifie pas cette variable.
On tape pour définir la fonction `lambda` :

```
lambda(x):=x+1
```

```
 (x)->x+1
```

On tape pour avoir la valeur de la fonction `lambda` en $x=5$:

```
lambda(5)
```

6

On tape pour définir une fonction sans nom :

```
lambda x : x*x+1
```

```
 (x)->x*x+1
```

On tape pour avoir sa valeur en $x=5$:

```
(lambda x : x*x+1)(5)
```

26

2.2.6 Définition d'une fonction algébrique par morceaux avec quand

`quand` a 3 arguments : une condition (même symbolique) et 2 expressions : `quand(Cond,Expr1,Expr2)`
Si la condition `Cond` est vraie alors `quand` renvoie `Expr1`, si la condition `Cond` est fausse alors `quand` renvoie `Expr2`.

Exemple : écriture alternative pour la fonction `Abs1` définie par `Abs1`$(x) = |x-1|-1$, on a :
- si $x > 1$ on a `Abs1`$(x) = x-1-1 = x-2$
- si $x \leq 1$ on a `Abs1`$(x) = -x+1-1 = -x$

On tape :

```
Abs1(x):= quand(x > 1, x-2,-x);
```

```
 (x)->((x>1)? x-2 : -x)
```

ou bien avec `fonction...ffonction` et `retourne` :

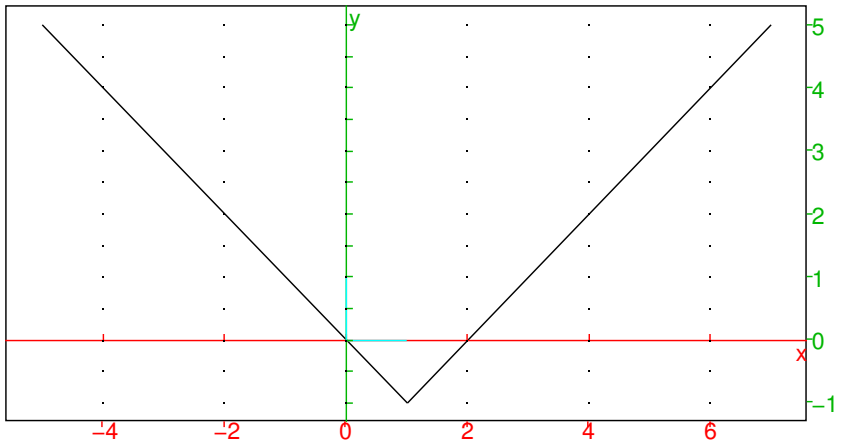
```
fonction Abs1(x) retourne quand(x > 1,x-2,-x
); ffonction;
```

```
 (x)->return(((x>1)? x-2 : -x))
```

```
Abs1(1/2),Abs1(1),Abs1(3/2),Abs1(-2),Abs1(2)
```

$$\frac{-1}{2}, -1, \frac{-1}{2}, 2, 0$$

```
plotfunc(Abs1(x),x=-5..7)
```



Remarque différence entre `ifte` et `quand`

```
f(x):=ifte(x>0,1,0);
```

```
 (x)->if (x>0) 1;
```

```
f(3),f(-2)
```

$$1, 0$$

```
f(x)
```

Ifte : impossibledefaireletestErreur : ValeurArgumentIncorrecte

Pourquoi une erreur ?
Ici `x` n'a pas de valeur : avec `ifte` ou `if then else end_if` il faut que la variable `x` soit affectée pour pouvoir tester la condition (quand on définit une fonction ce qui suit le := n'est pas évalué donc la définition de `f(x)` ne pose pas de problème).
Pour la définition de g avec `when....`, la variable `x` n'a pas besoin dêtre affectée. Il n'y a donc pas d'erreur.

```
g(x):=quand(x>0,1,0)
```

```
 (x)->((x>0)? 1 : 0)
```

```
g(3),g(-2)
```

$$1, 0$$

```
g(x)
```

$$\text{when}\,(x > 0, 1, 0)$$

Chapitre 3

Les instructions de programmation sur des exemples

On donne ici les instructions en syntaxe Xcas, et en syntaxe Python.

3.1 Commentaires et documentation.

3.1.1 Les commentaires dans un programme.

Pour faire un commentaire, on écrit `//` (ou `#` si on a choisi la syntaxe Python), tout ce qui suit est le commentaire.

3.1.2 Documenter une fonction avec la commande `help`

`help(nom_prog)` renvoie la 1ère ligne d'un programme (hors commentaire). On peut donc indiquer à la personne utilisant un programme ce que fait ce programme en écrivant comme première ligne du programme une chaîne de caractères commentant la fonction.
Exemple avec la fonction disciminant `Delta(a,b,c)` qui calcule b^2-4ac.
On tape en syntaxe `Xcas` :

```
fonction Delta(a,b,c)
  // discriminant
  "calcule b^2-4ac";
  return b^2-4a*c;
ffonction:;
```

```
help(Delta)
```

HelponuserfunctionDelta(a, b, c)calculeb^2 − 4ac

On tape en syntaxe Python :

```
def Delta(a,b,c) :
    # discriminant
    "calcule b^2-4ac"
    return b^2-4a*c

help(Delta)
```

HelponuserfunctionDelta(a, b, c)calculeb^2 − 4ac

3.2 Stocker une valeur dans une variable avec :=

L'opérateur infixé `:=` stocke le deuxième argument dans la variable donnée comme premier argument.
On peut aussi stocker plusieurs valeurs dans plusieurs variables, par exemple : En syntaxe `Xcas` :
`a,b,c:=1,2,3` ou `(a,b,c):=(1,2,3)` ou `[a,b,c]:=[1,2,3]`
En syntaxe Python :
`(a,b,c)=(1,2,3)` ou `[a,b,c]=[1,2,3]`

Exemple :
En syntaxe `Xcas` :

```
a:=3
```

3

```
b:=5
```

5

```
a,b:=a+b,b-a
```

8, 2

```
a,b
```

$$8, 2$$

En syntaxe Python :

```
a=3
```

$$8 = 3$$

```
b=5
```

$$2 = 5$$

```
(a,b)=(a+b,b-a)
```

$$8, 2, 10 = -6$$

```
a,b
```

$$8, 2$$

3.3 Enlever une valeur stockée dans une variable avec purge

En syntaxe `Xcas` : l'instruction `purge(a)` permet d'enlever une valeur stockée dans la variable `a`. La variable `a` redevient une variable libre i.e. une variable non affectée.
En syntaxe Python : l'instruction `del a` permet d'enlever une valeur stockée dans la variable `a`.
Exemple :
En syntaxe `Xcas` :

```
a:=3
```

$$3$$

```
b:=5
```

$$5$$

```
purge(a)
```

$$3$$

```
a,b
```

$$a, 5$$

En syntaxe Python :

```
a=3
```

$$a = 3$$

```
b=5
```

$$5 = 5$$

```
del a
```

Nosuchvariablea

```
a,b
```

$$a, 5$$

3.4 Suite d'instructions avec ; ou :;

En syntaxe `Xcas`, pour effectuer une suite d'instructions, il suffit d'écrire ces instructions les unes à la suite des autres, en terminant chaque instruction par ;
En syntaxe `Xcas`, on tape :

```
a:=2; b:=3; a*b;
```

$$2, 3, 6$$

En syntaxe Python, on tape :

```
#a=2 b=3a*b
```

undef

Notez le signe # qui force l'interpréteur à passer en mode compatible Python.

Remarque :
Lorsque la réponse est trop longue, on peut aussi utiliser :; et on obtient `Done` comme réponse.

```
a:=2:; b:=[1,2,3,4,5]:; a*b;
```

$$\text{Done}, \text{Done}, [2, 4, 6, 8, 10]$$

Notez qu'à l'intérieur d'un programme écrit en syntaxe Python le ; n'est pas nécessaire, il suffit de séparer les instructions par un passage à la ligne, en respectant la même indentation que la première instruction.

3.5 L'instruction `retourne` ou `return`

L'instruction `retourne` arrête immédiatement l'exécution du programme et renvoie la valeur de l'instruction située après `retourne`.

Exemple :
En syntaxe `Xcas` :

```
fonction Delta(a,b,c) retourne b^2-4*a*c;
 ffonction;
```

```
    (a,b,c)->return(b^2-4*a*c)
```

```
fonction Abs(x)   si x > 0 alors retourne
 x;fsi;  retourne -x;ffonction;
```

```
    (x)->{
    si x>0 alors return(x) sinon 0
    fsi;
    return(-x);
    }
```

```
   Delta(3,5,2)
```

1

```
   Abs(-3),Abs(3)
```

3, 3

En syntaxe Python :

```
def Delta(a,b,c) :
    return b^2-4*a*c

def Abs(x) :
    if x>0 :
        return x
    return -x
```

```
   Delta(3,5,2)
```

1

```
   (Abs(-3),Abs(3))
```

3, 3

3.6 L'instruction `local`

Notion de variables locales :
Supposons qu'on souhaite définir une fonction h de deux variables a, b (a et b sont des entiers) qui renvoie le numérateur et le dénominateur de la fraction $\frac{a}{b}$ simplifiée.
Pour cela il faut diviser a et b par leur pgcd qui est `gcd(a,b)`.

En syntaxe `Xcas`, on peut écrire **sans utiliser de variables locales** :

```
fonction h0(a,b)
  retourne (a/gcd(a,b),b/gcd(a,b));
ffonction:;
```

En syntaxe Python

```
def h0(a,b) :
     return (a/gcd(a,b),b/gcd(a,b))
```

Mais on observe que cela nécessite de faire deux fois le calcul de `gcd(a,b)`.

Pour éviter de faire faire à l'ordinateur deux fois le même calcul, on va utiliser une variable locale c qui servira à stocker le calcul intermédiaire `gcd(a,b)` avec l'instruction : `c:=gcd(a,b)` (`:=` est le symbole de l'affectation et `gcd(a,b)` renvoie le pgcd de a et b).
Cette variable n'est pas visible à l'extérieur du programme, les modifications faites sur `c` dans le programme n'ont aucun effet sur la variable `c` de la session.
On écrit en syntaxe Xcas `local c;` (ne pas oublier le `;`). Le langage Python déclare implicitement les variables d'une fonction comme variables locales, aussi en syntaxe Python dans Xcas la déclaration de variables locales se fait par un commentaire Python `# local c;`.

D'où la définition de la fonction `h` **en utilisant des variables locales**
En syntaxe `Xcas` :

```
fonction h(a,b)
  local c;
  c:=gcd(a,b);
  retourne (a/c,b/c);
ffonction:;
```

```
   a:=12;b:=22;c:=32;
```

12, 22, 32

```
h0(25,15);
```

$$5, 3$$

```
h(25,15);
```

$$5, 3$$

```
a,b,c
```

$$12, 22, 32$$

On voit ainsi que les valeurs de `a,b,c` n'ont pas été changées par l'exécution des fonctions `h0` ou `h`. En syntaxe Python :

```
def h(a,b) :
    # local c
    c:=gcd(a,b);
    return (a/c),b/c)

a:=12;b:=22;c:=32;
```

$$12, 22, 32$$

```
h0(25,15);
```

$$5, 3$$

```
h(25,15);
```

$$5, 3$$

```
a,b,c
```

$$12, 22, 32$$

Autre exemple :
En syntaxe Xcas :

```
fonction Racine(a,b,c)
  //racine x1,x2 de ax^2+bx+c=0
  "calcule b^2-4ac,x1,x2";
  local d,x1;x2;
  d=b^2-4a*c;
  x1=(-b+sqrt(d))/(2a);
  x2=(-b-sqrt(d))/(2a);
  retourne b^2-4a*c,x1,x2;
ffonction:;
```

```
help(Racine), Racine(2,1,-3)
```

$$\text{Helponuserfunction}\text{Racine}(\text{a}, \text{b}, \text{c})\text{calculeb}\hat{}2 - 4\text{ac}, \text{x1}, \text{x2}, 25, 1, \frac{-3}{2}$$

```
Racine(1,2,3)
```

$$-8, \frac{(-2 + 2 * i \cdot \sqrt{2})}{2}, \frac{(-2 - 2 * i \cdot \sqrt{2})}{2}$$

En syntaxe Python :

```
def Racine(a,b,c) :
    # racine x1,x2 de ax^2+bx+c=0
    # local d,x1,x2
    "calcule b^2-4ac,x1,x2";
    d=b^2-4a*c
    x1=(-b+sqrt(d))/(2a)
    x2=(-b-sqrt(d))/(2a)
    retourne b^2-4a*c,x1,x2
```

```
help(Racine), Racine(2,1,-3)
```

$$\text{HelponuserfunctionRacine(a, b, c)calculeb\^{}2} - 4\text{ac, x1, x2, 25, 1, } \frac{-3}{2}$$

```
Racine(1,2,3)
```

$$-8, \frac{(-2+2*i\cdot\sqrt{2})}{2}, \frac{(-2-2*i\cdot\sqrt{2})}{2}$$

3.7 L'instruction pour

Exemple de pour : la somme des nombres impairs
On veut, dans cet exemple, définir une fonction `Simpair(n)` d'une variable `n` (`n` est un entier) qui calcule la somme des `n` premiers entiers impairs i.e. `1+3+...+2n-1`.
Pour cela, on utilise une variable locale S que l'on initialise à 0 :
`S:=0;`
puis on va faire n étapes en utilisant cette variable locale S.
S va contenir successivement :
étape 1 $S := S + (2 * 1 - 1)$; donc S contient 1 (0+1)
étape 2 $S := S + (2 * 2 - 1)$; donc S contient 4 (1+3)
...
étape k $S := S + (2 * k - 1)$; donc S contient $1 + 3 + ..2k - 1$
...
étape n $S := S + 2 * n - 1$; donc S contient $1 + 3 + ..2n - 1$
En syntaxe `Xcas`, on utilise une boucle `pour` :

```
pour k de 1 jusque n faire S:=S+2*k-1; fpour;
```

En syntaxe Python, on utilise une boucle `for` :

```
for k in range(1,n+1):
    S:=S+2*k-1
```

Dans cette boucle `k` sera successivement égal à `1, 2, 3,...n`.
On dit que l'instruction `S:=S+2*k-1` figurant dans le corps la boucle sera exécuté `n` fois.
Comment fonctionne cette boucle `pour` ?

- la variable `k` est initialisée à `1`,
- les instructions du corps de la boucle sont effectuées (ici il y en a une seule `S:=S+2*k-1`),

- k est est incrémenté automatiquement de 1 (k:=k+1),
- le test k<=n est effectué : si k<=n est vrai, les instructions du corps de la boucle sont à nouveau effectuées etc ...
  sinon on effectue les instructions qui suivent fpour.

En syntaxe Xcas :

```
fonction Simpair(n)
  local S,k;
  S:=0;
  pour k de 1 jusque n faire
    S:=S+2*k-1;
  fpour;
  retourne S;
ffonction:;
```

```
 Simpair(5),Simpair(6),Simpair(100)
```

$$25, 36, 10000$$

En syntaxe Python :

```
def Simpair(n):
    # local S,k
    S=0
    for k in range(1,n+1):
        S=S+2*k-1
    return(S)
```

```
Simpair(5),Simpair(6),Simpair(100)
```

$$25, 36, 10000$$

On pourra vérifier le résultat, en utilisant la commande sum de Xcas :

```
sum(2*k-1,k=1..5),sum(2*k-1,k=1..6),sum(2*k-1,k=1..100
)
```

$$25, 36, 10000$$

Intermède mathématique
1,3,5,7,9,11, Au vue des résultats obtenus pouvez-vous deviner la valeur de s(100) ?
Pouvez-vous deviner et montrer la formule qui donne s(n) ?

On devine : $s(n) = 1 + 3 + ...2n - 1 = n^2$
On sait que pour tout $k \in \mathbb{N}$ on a :
$k^2 - (k-1)^2 = ((k-1)+1)^2 - (k-1)^2 = 2k - 1$ et
$(k+1)^2 - k^2 = 2k + 1$
Donc :
$1 = 1^2 - 0^2$ $(k = 1)$,
$3 = 2^2 - 1^2$ $(k = 2)$,
$5 = 3^2 - 2^2$ $(k = 3)$,
...
$2k - 1 = k^2 - (k-1)^2$
$2k + 1 = (k+1)^2 - (k)^2$
...
$2n - 1 = n^2 - (n-1)^2$
Donc :
$s(n) = 1+3+...2n-1 = 1+(n)^2 = 1+(4-1)+(9-4)...+(n^2-(n-1)^2) = n^2$
En classe de terminales, on peut montrer cette formule par récurrence :
$s(1) = 1$ si $s(n) = (n)^2$ alors on a :
$s(n+1) = s(n) + 2(n+1) - 1 = (n)^2 + 2n + 1 = (n+1)^2$
La formule est donc montrée par récurrence.

3.8 L'instruction `pour` avec un `pas`

Exemple de pour avec un pas : la somme des nombres impairs
Revenons à l'exemple précédent :
définir une fonction `Simpair(n)` d'une variable `n` (`n` est un entier) qui calcule la somme : `1+3+...+2n-1`.
Avec la syntaxe `Xcas`, on peut faire varier la variable `k` avec un pas de 2, en commencant par 1 jusque `2n-1`, on écrit :
`pour k de 1 jusque 2*n-1n pas 2 faire fpour;`
Dans ce cas la valeur de `S` doit être augmentée à chaque pas de la valeur `k`.
On tape en syntaxe `Xcas` :

```
fonction Simpair1(n)
 local S,k;
 S:=0;
 pour k de 1 jusque 2*n-1 pas 2 faire
   S:=S+k;
 fpour;
 retourne S;
ffonction:;
```

```
Simpair(5),Simpair(6),Simpair(7)
```

$$25, 36, 49$$

On tape en syntaxe Python :

```
def Simpair1(n):
    # local S,k
    S=0
    for k in range(1,2*n,2):
       S=S+k
    return(S)

Simpair(5),Simpair(6),Simpair(7)
```

$$25, 36, 49$$

Autre exemple de pour avec un pas : le ticket de caisse

On veut faire le programme d'un ticket de caisse lors d'achats dans un magasin qui ne pratique pas de réduction pour les achats en gros.
Le programme du `ticketcaisse` a comme paramètre une liste `L` donnant le nombre d'un même article suivi du prix de cet article, par exemple :
si `L:=[2,24,1,15,5,10]` cela signifie qu'il y 2 articles à 24 euros, 1 article à 15 euros et 5 articles à 10 euros.
Soit `n:=dim(L)`, dans cet esxemple `n:=6`.
On va parcourir la liste avec une variable `k` : `L[k]` sera le nombre d'articles ayant comme prix `L[ k+1]` : il faut donc, dans cet exemple, que `k` prenne successivement pour valeur `0, 2, 4=n-2`.
Pour cela on initialise la somme à payer avec 0 : `S:=0` puis
on utilise une boucle `pour` avec un `pas` de 2 :
`pour k de 0 jusque n-2 pas 2 faire S:=S+L[k]*L[k+1]; fpour;`
Dans cette boucle `pour`, la variable `k` est initialisée à `0`, puis les instructions du corps de la boucle sont effectuées, puis `k` est incrémenté automatiquement de `2` (`k:=k+2`),
puis on fait le test `k<=n-2` si oui les instructions du corps de la boucle sont à nouveau effectuées etc ... sinon on effectue les instructions qui suivent `fpour`.
En syntaxe `Xcas` :

```
fonction Ticketcaisse(L)
 local S,n,k;
 n:=dim(L);
 S:=0;
 pour k de 0 jusque n-2 pas 2 faire S:=S+L[k]*L[k+1]; fpour;
```

```
 retourne S;
ffonction:;
```

```
Ticketcaisse([1,12,3,5,2,8])
```

43

```
1*12+3*5+2*8
```

43

En syntaxe Python :

```
def Ticketcaisse(L) :
    # local S,n,k
    n=size(L)
    S=0
    for k in range(0,n-1,2) :
        S=S+L[k]*L[k+1]
    retourne S
```

```
Ticketcaisse([1,12,3,5,2,8])
```

43

```
1*12+3*5+2*8
```

43

3.9 L'instruction `si`

Lorsque l'on veut faire des instructions seulement si une condition est réalisée on utilise l'instruction `si`.
Cette instruction a 2 formes :
En syntaxe `Xcas` :
`si <condition> alors <instruction>; fsi;`
et
`si <condition> alors <instruction1>; sinon <instruction2> fsi;`
Comment fonctionne le test `si ... alors ... fsi` ?

- On évalue la condition : une condition a 2 valeurs possibles `vrai` ou `faux` c'est ce que l'on nomme un booléen,
- Si la condition est vraie : on effectue les `<instruction>`, et si la condition est fausse : on ne fait rien,
- On effectue ensuite les instructions qui suivent `fsi;`.

Comment fonctionne le test `si ... alors ...sinon ... fsi` ?

- On évalue la condition : une condition a 2 valeurs possibles `vrai` ou `faux` c'est ce que l'on nomme un booléen,
- Si la condition est vraie : on effectue les `<instruction1>` et si la condition est fausse : on effectue les `<instruction2>`,
- On effectue ensuite les instructions qui suivent `fsi;`.

En syntaxe Python :

```
if <condition> :
   <instruction>
```

et

```
if <condition> :
   <instruction1>
else :
    <instruction2>
```

Exemple de `si alors fsi` : somme des nombres premiers ¡= n

Rappel Un nombre premier est un entier n qui admet 2 diviseurs qui sont 1 et n, par exemple : 17 est un nombre premier et 1 4 ne sont pas des nombres premiers.

On veut définir une fonction `Spremier(n)` d'une variable `n` (`n` est un entier) qui calcule la somme des nombres premiers inférieurs ou égaux à `n` : `2+3+5+7+11...+p` pour `p` premier et `p<=n`. Pour cela on utilisera la fonction booléenne `est_premier` de `Xcas` qui teste si un nombre entier est premier.

Si `est_premier(k)==vrai` et si `k<=n` alors la valeur de `S` doit être augmentée de la valeur `k`.

Pour cela, on utilise une variable locale `S` (la somme) et le test.

En syntaxe `Xcas` :

```
si est_premier(k) alors S:=S+k fsi;
```

En syntaxe Python :

```
if est_premier(k):
    S:=S+k
```

À part le nombre 2 les nombres premiers sont impairs, donc on initialise `S` à 2 puis, on fait varier `k` de 3 jusque `n` avec un pas de 2.
En syntaxe `Xcas` :

```
fonction Spremier(n)
 local S,k;
 S:=2;
 pour k de 3 jusque n pas 2 faire
   si est_premier(k) alors S:=S+k fsi;
 fpour;
 retourne S;
ffonction:;
```

```
   Spremier(10),Spremier(11),Spremier(20)
```

$$17, 28, 77$$

En syntaxe Python :

```
def Spremier(n):
    # local S,k
    S=2
    for k in range(3,n+1,2):
        if est_premier(k) :
            S=S+k
    return(S)
```

```
   Spremier(10),Spremier(11),Spremier(20)
```

$$17, 28, 77$$

Exercice : combien y-a-t-il de nombres premiers ¡= n ?
Modifier l'algorithme précédent pour définir la fonction `Nbpremiers(n)` d'une variable `n` (`n` est un entier) qui calcule le nombre de nombres premiers qui sont inférieurs ou égaux à `n`.
En syntaxe `Xcas` :

```
fonction Nbpremiers(n)
 local S,k,N;
 si n<2 alors retourne 0 fsi;
 N:=1;
 pour k de 3 jusque n pas 2 faire
```

```
   si est_premier(k) alors N:=N+1 fsi;
 fpour;
 retourne N;
ffonction:;

   Nbpremiers(10),Nbpremiers(20),Nbpremiers(100)
```

$$4, 8, 25$$

En syntaxe Python :

```
def Nbpremiers(n):
    # local S,k,N
    if n<2 :
        retourne 0
    N=1
    for k in range(3,n+1,2) :
        if est_premier(k) :
            N=N+1
    return(N)

   Nbpremiers(10),Nbpremiers(20),Nbpremiers(100)
```

$$4, 8, 25$$

On pourra vérifier le résultat, en utilisant la commande `nprimes` de `Xcas` :

```
   nprimes(10),nprimes(20),nprimes(100)
```

$$4, 8, 25$$

Autre exemple du test `si alors sinon fsi` : le ticket de caisse
Dans un magasin on favorise les achats en gros :
si un article `a` a comme prix affiché `P` euros, pour l'achat d'au moins 3 articles `a`, vous avez une réduction de 10%.
On veut, dans cet exemple, définir une fonction `Prix` de 2 variables `n` (`n` est un entier) et `P` un réel qui calcule le prix de `n` article(s).
Pour cela, on utilise :
En syntaxe `Xcas`, une variable locale `S` qui sera la somme à débourser et le test :`si alors sinon fsi;`
En syntaxe Python, on utilise le test :

```
if <condition> :
    <instruction1>
else :
    <instruction2>
```

En syntaxe Xcas :

```
fonction Prix(n,P)
 local S;
 si n>=3 alors S:=n*P*0.9;
 sinon S:=n*P; fsi;
 retourne S;
ffonction:;
```

```
Prix(3,8),Prix(2,8)
```

21.6, 16

```
3*8-3*8*0.1,2*8
```

21.6, 16

En syntaxe Python :

```
def Prix(n,P) :
    # local S
    if n>=3 :
        S:=n*P*0.9
    else :
        S:=n*P
    return S
```

```
Prix(3,8),Prix(2,8)
```

21.6, 16

```
3*8-3*8*0.1,2*8
```

21.6, 16

3.10 Utiliser une fonction utilisateur dans un programme

Exemple : combien de nombres premiers dans [1,b],[b+1,2b]..[(p-1)b+pb] ? On veut définir une fonction `Lnbpremiers(b,p)` de paramètres 2 entiers qui renvoie une liste de `p` entiers qui sont le nombre de nombres premiers compris entre 1 et `b`,`b+1` et `2b` ... `(p-1)*b+1` et `p*b` de nombres premiers entre 0 et `p*b=n`.
Par exemple si `b=10` et `p=3` `Lnbpremiers(b,p)` doit renvoyer : `[4,4,2]` puisque :
entre 1 et 10 il y a 4 nombres premiers (2,3,5,7),
entre 11 et 20 il y a 4 nombres premiers (11,13,17,19),
entre 21 et 30 il y a 2 nombres premiers (23,29).

Pour cela on utilisera la fonction utilisateur `Nbpremiers` écrite précédemment en exercice.
En syntaxe `Xcas` :

```
fonction Lnbpremiers(b,p)
  local L,k,n1,n2,n;
  L:=NULL;
  n:=p*b;
  n1:=Nbpremiers(1);
  pour k de b jusque n pas b faire
     n2:=Nbpremiers(k);
     L:=L,n2-n1;
     n1:=n2;
  fpour;
  retourne [L];
  ffonction:;
```

```
Lnbpremiers(10,30)
```

$$[4, 4, 2, 2, 3, 2, 2, 3, 2, 1, 4, 1, 1, 3, 1, 2, 2, 2, 1, 4, 0, 1, 3, 2, 1, 2, 2, 2, 2, 1]$$

```
Lnbpremiers(100,30)
```

$$[25, 21, 16, 16, 17, 14, 16, 14, 15, 14, 16, 12, 15, 11, 17, 12, 15, 12, 12, 13, 14, 10, 15, 15, 10, 11, 15, 14, 12, 11]$$

En syntaxe Python :

```
def Lnbpremiers(b,p) :
    # local L,k,n1,n2,n
    L=[]
    n=p*b
    n1=Nbpremiers(1)
    for k in range(b,n+1,b) :
        n2=Nbpremiers(k)
        L.append(n2-n1)
        n1=n2
    return L;
```

```
Lnbpremiers(10,30)
```

$$[4, 4, 2, 2, 3, 2, 2, 3, 2, 1, 4, 1, 1, 3, 1, 2, 2, 2, 1, 4, 0, 1, 3, 2, 1, 2, 2, 2, 2, 1]$$

```
Lnbpremiers(100,30)
```

$$[25, 21, 16, 16, 17, 14, 16, 14, 15, 14, 16, 12, 15, 11, 17, 12, 15, 12, 12, 13, 14, 10, 15, 15, 10, 11, 15, 14$$

Autre exemple : le ticket de caisse

On veut faire le programme du ticket de caisse lorsque le magasin pratique l'achat en gros (la liste `L` doit spécifier le nombre n d'un même article de prix P).
En utilisant la fonction `Prix(n,P)` écrite précédemment (cf 3.9), modifier le programme précédent lorsque le magasin pratique l'achat en gros.
En syntaxe `Xcas` :

```
fonction Ticketengros(L)
 local S,n,k;
 n:=dim(L);
 S:=0;
 pour k de 0 jusque n-2 pas 2 faire
   S:=S+Prix(L[k],L[k+1]);
 fpour;
 retourne S;
ffonction:;
```

```
Ticketengros([1,12,3,5,2,8])
```

```
12+3*5*0.9+2*8
```

41.5

En syntaxe Python :

```
def Ticketengros(L) :
    # local S,n,k
    n=dim(L)
    S=0
    for k in range(0,n-1,2) :
        S=S+Prix(L[k],L[k+1])
    return S

Ticketengros([1,12,3,5,2,8])
```

41.5


```
12+3*5*0.9+2*8
```

41.5

3.11 L'instruction tantque

Notion de boucle tantque

On utilise une boucle `tantque` lorsque l'on ne connaît pas à l'avance le nombre d'itérations à effectuer et que l'on arrête les itérations quand une condition devient fausse.

En syntaxe `Xcas` :

```
tantque <condition> faire <instructions> ftantque;
```

En syntaxe Python :

```
while  <condition> :
    <instructions>
```

Comment fonctionne une boucle tantque ?

- On évalue la condition : une condition a 2 valeurs possibles `vrai` ou `faux` c'est ce que l'on nomme un booléen,
- Si la condition est vraie : on effectue `<instructions>` et on revient au test, si la condition est fausse : on passe directement aux instructions qui suivent `ftantque;`.

ou bien on peut dire en langage courant que :
`<condition>` est une condition de continuation de la boucle. tant que la condition est vérifiée, on fait les instructions de la boucle.
Traduction d'une boucle `pour` en une boucle `tantque`
Exemple : somme des éléments d'une liste Soit une liste `L` de nombres réels.
On veut faire la somme des réels de `L`.
En syntaxe `Xcas` :
On tape en utilisant une boucle `pour` :

```
fonction Somme(L)
 local n,j,S;
 n:=dim(L);
 S:=0;
 pour j de 0 jusque n-1 faire
   S:=S+L[j];
 fpour;
 retourne S;
ffonction:;
```

On tape en utilisant une boucle `tantque` :

```
fonction Somme1(L)
 local n,j,S;
 n:=dim(L);
 S:=0;
 j:=0;
 tantque j <= n-1 faire
   S:=S+L[j];
   j:=j+1;
 ftantque;
 retourne S;
ffonction:;
```

On peut aussi écrire `Somme2`, mais **Attention** à l'ordre des instructions de la boucle `tantque` et au test d'arrêt :

```
fonction Somme2(L)
 local n,j,S;
 n:=dim(L);
 j:=0;
 S:=L[0];
 tantque j < n-1 faire
   j:=j+1;
   S:=S+L[j];
 ftantque;
 retourne S;
ffonction:;
```

```
L:=[10,13,22,15,5,27,45,31,78]
```

$$[10, 13, 22, 15, 5, 27, 45, 31, 78]$$

```
Somme(L)
```

$$246$$

```
Somme1(L)
```

$$246$$

```
Somme2(L)
```

$$246$$

En syntaxe Python :
On tape en utilisant une boucle `pour` :

```
def Somme(L) :
    # local n,j,S
    n=dim(L)
    S=0
    for j in range(n) :
        S:=S+L[j]
    return S
```

On tape en utilisant une boucle `tantque` :

```
def Somme1(L) :
    # local n,j,S
    n=dim(L)
    S=0
    j=0
    while j <= n-1 :
        S=S+L[j]
        j=j+1
    return S
```

On peut aussi écrire `Somme2`, mais **Attention** à l'ordre des instructions de la boucle `tantque` et au test d'arrêt :

```
def Somme2(L) :
    # local n,j,S
    n=dim(L)
    j=0
    S=L[0]
    while j < n-1 :
        j=j+1
        S=S+L[j]
    return S;
```

```
L:=[10,13,22,15,5,27,45,31,78]
```

$$[10, 13, 22, 15, 5, 27, 45, 31, 78]$$

```
Somme(L)
```

$$246$$

```
Somme1(L)
```

$$246$$

```
Somme2(L)
```

246

On peut vérifier ces résultats avec la fonction `sum` de `Xcas` :

```
sum([10,13,22,15,5,27,45,31,78]
```

246

Autre exemple : trouver le n-ième nombre premier On veut définir une fonction `Niemeprem(n)` qui renvoie le nième nombre premier.
On utilise, pour cela la fonction booléenne `est_premier` (ou `is_prime`) de `Xcas` qui teste si un nombre entier est premier.
A part le nombre 2, les nombres premiers sont impairs, donc on traite le cas de 2, puis on initialise `k` à 3 et on fait varier `k` avec un pas de 2. En syntaxe `Xcas` :

```
fonction Niemeprem(n)
  local k,p;
  si n==1 alors retourne 2; fsi;
  k:=3;p:=1;
  tantque p<n faire
    si est_premier(k) alors p:=p+1; fsi;
    k:=k+2;
  ftantque;
  retourne k-2;
ffonction:;
```

```
Niemeprem(10),Niemeprem(100)
```

29, 541

En syntaxe `Xcas` avec un `pour` :

```
fonction Niemeprem1(n)
  local k,p;
  si n==1 alors retourne 2; fsi;
  p:=1;
  pour k de 3 jusque 100*n pas 2 faire
```

```
    si est_premier(k) alors p:=p+1; fsi;
    si p==n alors break; fsi;
  fpour;
  retourne k;
ffonction:;
```

```
Niemeprem1(10),Niemeprem1(100)
```

$$29,541$$

En syntaxe Python :

```
def Niemeprem(n) :
    # local k,p
    if n==1 :
        return 2
    k=3
    p=1
    while p<n :
        if isprime(k) :
            p=p+1
        k:=k+2
    return k-2
```

```
Niemeprem(10),Niemeprem(100)
```

$$29,541$$

On pourra vérifier le résultat du programme car cette fonction existe dans `Xcas` c'est `ithprime(n)`

```
ithprime(10),ithprime(100)
```

$$29,541$$

En syntaxe Python avec un `for` :

```
def Niemeprem1(n) :
    # local k,p
    if n==1 :
        return 2
```

```
p=1
for k in range(3,100*n,2) :
    if isprime(k) :
       p:=p+1
    if p==n :
       break;
return k;
```

```
Niemeprem1(10),Niemeprem1(100)
```

$$29, 541$$

3.12 Un exercice et un problème

3.12.1 Exercice

On veut définir une fonction `Snpremiers(n)` d'une variable `n` (`n` est un entier positif) qui calcule la somme des `n` premiers nombres premiers. On utilise, pour cela la fonction booléenne `est_premier` (ou `is_prime`) de `Xcas` qui teste si un nombre entier est premier.
A part le nombre 2, les nombres premiers sont impairs, donc on initialise `S` à 2 et `k` à 3 avec un pas de 2.

En syntaxe `Xcas`, (l'indice `j` compte les nombres premiers, l'indice `k` sert à chercher les nombres qui sont premiers) :

```
fonction Snpremiers(n)
  local S,k,p,j;
  j:=1;
  S:=2;
  k:=3;
  tantque j< n faire
     si est_premier(k) alors
       S:=S+k;
       j:=j+1;
     fsi;
     k:=k+2;
  ftantque;
  retourne S;
```

```
ffonction:;
```

```
Snpremiers(8),Snpremiers(10),Snpremiers(100)
```

$$77, 129, 24133$$

En syntaxe `Xcas`, avec un `pour` :

```
fonction Snpremiers1(n)
  local S,k,p,j;
  j:=1;
  S:=2;
  pour k de 3 jusque 100*n pas 2 faire
     si est_premier(k) alors
       S:=S+k;
       j:=j+1;
     fsi;
     si j==n alors break; fsi
  fpour;
  retourne S;
ffonction:;
```

```
Snpremiers1(8),Snpremiers1(10),Snpremiers1(100)
```

$$77, 129, 24133$$

En syntaxe Python, (l'indice `j` compte les nombres premiers, l'indice `k` sert à chercher les nombres premiers) :

```
def Snpremiers(n) :
    # local S,k,p,j
    j=1
    S=2
    k=3
    while j< n :
        if is_prime(k) :
            S=S+k
            j=j+1
        k=k+2
    return S
```

```
Snpremiers(8),Snpremiers(10),Snpremiers(100)
```

$$77, 129, 24133$$

En syntaxe Python avec un `for` :

```
def Snpremiers1(n) :
    # local S,k,p,j
    j=1
    S=2
    for k in range(3,100*n,2) :
        if is_prime(k) :
            S=S+k
            j=j+1
        if j==n :
            break
    return S
```

3.12.2 Problème : le crible d'Eratosthène

3.12.3 Description

Pour trouver les nombres premiers inférieurs ou égaux à N :

1. On écrit les nombres de 2 à N dans une liste TAB.
2. On met 2 dans la case P .
3. Si $P \times P \leq N$ il faut traiter les éléments de P à N : on barre tous les multiples de P à partir de $P \times P$.
4. On augmente P de 1.
   Si $P \times P$ est strictement supérieur à N, on arrête
5. On met le plus petit élément non barré de la liste dans la case P. On reprend à l'étape 3.
6. on met les éléments non nuls de TAB dans une liste $PREM$.

3.12.4 Écriture de l'algorithme

```
Fonction crible(N)
 local TAB PREM I P
 // TAB et PREM sont des listes
 //=> est le STO des calculatrices
```

```
[] =>TAB
[] =>PREM
//on suppose que les indices d'une liste debutent par 0
pour I de 0 a N faire
 TAB:=concat(TAB, I)
fpour
//On met 0 dans TAB[1] car 1 n'est pas premier
//barrer 1 a ete realise en le remplacant par 0
TAB[1]:=0
//TAB est la liste 0 0 2 3 4 ...N
P:=2
// On a fait les points 1 et 2
tantque P*P <= N faire
 pour I de P jusque E(N/P) faire
   //E(N/P) designe la partie entiere de N/P
   TAB[I*P]:=0
 fpour
 // On a donc barre tous les multiples de P a partir de P*P
 P:=P+1
 //On cherche le plus petit nombre <= N non barre (i.e. non nul)
 // entre P et N
 tantque (P*P <= N) et (TAB[P]=0) faire
   P=P+1
 ftantque
ftantque
//on ecrit le resultat dans une liste PREM
pour I de 2 a N faire
 si TAB[I]!= 0 alors
    PREM:=concat(PREM, I)
 fsi
fpour
retourne PREM
```

3.12.5 En syntaxe Xcas

Première version

```
//renvoie la liste des nombres premiers<=n selon Eratosthene
crible0(n):={
  local tab,prem,p,j;
  tab:=[0,0];
```

```
  prem:=[];
  pour j de 2 jusque n faire
    tab:=append(tab,j);
  fpour;
  p:=2;
  tantque (p*p<=n) faire
    pour j de p jusque floor(n/p) faire
      tab[j*p]:=0;
    fpour
    p:=p+1;
    tantque ((p*p<=n) et (tab[p]==0)) faire
      p:=p+1;
    ftantque;
  ftantque;
  pour j de 2 jusque n faire
    si (tab[j]!=0) alors
      prem:=append(prem,j);
    fsi
  fpour
  retourne(prem);
}:;
```

```
crible0(100)
```

$$[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]$$

```
time(crible0(1000))
```

$$[0.024, 0.0223407622]$$

L'instruction `time` affiche le temps d'exécution de la commande.

Une version plus efficace

1. On utilise `seq` pour définir `tab` et on remplace les affectations `:=` par des affectations par référence `=<`.

   ```
   tab:=[0,0];
   pour j de 2 jusque n faire
    tab:=append(tab,j);
   fpour;
   ```

par

```
tab:=seq(j,j,0,n);
tab[0]=<0; tab[1]=<0;
```

2. On remplace tout d'abord les affectations `:=` d'éléments d'une liste par une affectation par référence `=<` c'est à dire sans recopier la liste à chaque fois qu'on en modifie un élément, ce qui est plus efficace.
3. Puis, lorsque qu'on barre les multiples de `p` premier on remplace les multiplications de `tab[eval(j*p)]:=0;` par des additions on faisant varier `j` avec un pas : On remplace donc

```
pour j de p jusque floor(n/p) faire
   tab[j*p]:=0;
fpour;
```

par

```
pour j de p*p jusque n pas p faire
    tab[j]=<0;
fpour;
```

On obtient :

```
//renvoie la liste des nombres premiers<=n selon Eratosthene
crible(n):={
  local tab,prem,p,j;
  tab:=seq(j,j,0,n);
  tab[0]=<0; tab[1]=<0;
  prem:=[];
  p:=2;
  tantque (p*p<=n) faire
    pour j de p*p jusque n pas p faire
      tab[j]=<0;
    fpour
    p:=p+1;
    tantque ((p*p<=n) et (tab[p]==0)) faire
      p=<p+1;
    ftantque;
  ftantque;
  pour j de 2 jusque n faire
    si (tab[j]!=0) alors
      prem:=append(prem,j);
    fsi
```

```
  fpour
  retourne(prem);
}:;
```

```
   crible(100)
```

$[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]$

```
   time(crible(1000))
```

$$[0.0036, 0.00366153874]$$

3.12.6 En syntaxe Python

Première version

```
# renvoie la liste des nombres premiers<=n selon Eratosthene
def crible0(n) :
    # local tab,prem,p,j
    tab =[0,0]
    prem =[]
    for j in range(2,n+1) :
        tab.append(j)
    p=2
    while p*p<=n :
        for j in range(p,floor(n/p)+1) :
            tab[j*p]=0
        p=p+1
        while p*p<=n and tab[p]==0 :
            p=p+1
    for j in range(2,n+1) :
        if tab[j]!=0 :
            prem.append(j)
    return prem
```

```
   crible0(100)
```

$[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]$

```
time(crible0(1000))
```

$$[0.02, 0.0188385998]$$

Une version plus efficace

```
# renvoie la liste des nombres premiers<=n selon Eratosthene
def crible(n) :
    # local tab,prem,p,j
    tab=range(0,n+1)
    tab[0]=<0; tab[1]=<0
    prem=[]
    p=2
    while p*p<=n :
        for j in range(p*p,n+1,p) :
            tab[j]=<0
        p=p+1
        while p*p<=n and tab[p]==0 :
            p=p+1
    for j in range(2,n+1) :
        if tab[j]!=0 :
            prem.append(j)
    return prem;
```

```
crible(100)
```

$$[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]$$

```
time(crible(1000))
```

$$[0.0022, 0.00220681384]$$

3.13 Autre exemple de boucle `tantque` : le ticket de caisse

En fin de mois, Paul n'a plus qu'une somme `a` dans son porte-monnaie. Paul fait sa liste de courses `Lc` en mettant au début ce qu'il veut vraiement acheter et à la fin de sa liste, il met les achats qu'il doit faire à plus long terme. Dans le magasin (qui ne fait pas de réduction), sa liste de courses `Lc`

devient une liste de prix L.
Dans ce cas, on ne peut pas utiliser une boucle avec pour car on ne sait pas au départ combien de fois on doit effectuer la boucle.
On note S la variable qui stockera successivement la somme des prix des premiers éléments de L : pour cela on utilise la fonction Somme(L) écrite précedemment (cf 3.11).
Il veut faire un programme qui arrête sa liste dès que S>a en coupant L en 2 listes : La liste des objets de ce qu'il achète réellement pour un montant S<=a et Lfin liste des objets qu'il n'achète pas (Lfin est une liste vide lorsque Somme(L)<=a).
Ticketfindemois(L,a) doit renvoyer La,Lfin,P où P est la somme à payer.
"arrêt" se traduit ici par Lfin ==[] ou S>a donc
"continuation" se traduit ici par Lfin!=[] et S<=a.
On teste tout d'abord si Paul a assez d'argent pour payer toute sa liste : pour cela, on utilise le programme Somme précédent (cf 3.11).
Paul a assez d'argent pour payer toute sa liste lorsque Somme(L)<=a et alors on a La :=L, Lfin :=[] et P:=Somme(L).
Si Somme(L)>a, Paul n'a pas assez d'argent donc Lfin!=[] est vrai et la condition d'arrêt est : S<=a. On écrit Ticketfindemois(L,a) pour que k soit le nombre d'articles achetés lorsqu' on sort du tantque.
En syntaxe Xcas :

```
fonction Ticketfindemois(L,a)
 local S,n,k,Lfin,La;
 S:=Somme(L);
 si S <=a alors retourne L,[],S fsi;
 n:=dim(L);
 k:=0;
 S:=L[0];
 tantque S <=a faire
   k:=k+1;
   S:=S+L[k];
  ftantque;
 La:=gauche(L,k);
 Lfin:=droit(L,n-k);
 retourne La,Lfin,S-L[k];
ffonction:;
```

```
   L1:=[15,10,2,7,5,1.2,3,5.5,10,4,5.3]
```

$$[15, 10, 2, 7, 5, 1.2, 3, 5.5, 10, 4, 5.3]$$

```
Ticketfindemois(L1,50)
```

$$[15, 10, 2, 7, 5, 1.2, 3, 5.5], [10, 4, 5.3], 48.7$$

```
L2:=[1,15,2,7.8,5,2.2,4,5.5,2,2.7,1,2.1]
```

$$[1, 15, 2, 7.8, 5, 2.2, 4, 5.5, 2, 2.7, 1, 2.1]$$

```
Ticketfindemois(L2,60)
```

$$[1, 15, 2, 7.8, 5, 2.2, 4, 5.5, 2, 2.7, 1, 2.1], [], 50.3$$

En syntaxe Python :

```
def Ticketfindemois(L,a) :
    # local S,n,k,Lfin,La
    S=Somme(L)
    if S <=a :
        return L,[],S
    n=dim(L)
    k=0
    S=L[0]
    while S <=a :
        k=k+1
        S=S+L[k]
    La:=gauche(L,k)
    Lfin:=droit(L,n-k)
    return La,Lfin,S-L[k]
```

```
L1:=[15,10,2,7,5,1.2,3,5.5,10,4,5.3]
```

$$[15, 10, 2, 7, 5, 1.2, 3, 5.5, 10, 4, 5.3]$$

```
Ticketfindemois(L1,50)
```

$$[15, 10, 2, 7, 5, 1.2, 3, 5.5], [10, 4, 5.3], 48.7$$

```
L2:=[1,15,2,7.8,5,2.2,4,5.5,2,2.7,1,2.1]
```

$$[1, 15, 2, 7.8, 5, 2.2, 4, 5.5, 2, 2.7, 1, 2.1]$$

```
Ticketfindemois(L2,60)
```

$$[1, 15, 2, 7.8, 5, 2.2, 4, 5.5, 2, 2.7, 1, 2.1], [], 50.3$$

On peut aussi écrire mais **Attention** à l'ordre des instructions dans le `tantque`. En syntaxe `Xcas` :

```
fonction Ticketfindemois1(L,a)
 local S,n,k,Lfin,La;
 S:=Somme(L);
 si S <=a alors retourne L,[],S fsi;
 n:=dim(L);
 S:=0;
 k:=0;
 tantque S <=a faire
   S:=S+L[k];
   k:=k+1;
 ftantque;
 La:=gauche(L,k-1);
 Lfin:=droit(L,n-k+1);
 retourne La,Lfin,S-L[k-1];
ffonction:;
```

```
L1:=[15,10,2,7,5,1.2,3,5.5,10,4,5.3]
```

$$[15, 10, 2, 7, 5, 1.2, 3, 5.5, 10, 4, 5.3]$$

```
L2:=[1,15,2,7.8,5,2.2,4,5.5,2,2.7,1,2.1]
```

$$[1, 15, 2, 7.8, 5, 2.2, 4, 5.5, 2, 2.7, 1, 2.1]$$

```
Ticketfindemois1(L1,50)
```

$$[15, 10, 2, 7, 5, 1.2, 3, 5.5], [10, 4, 5.3], 48.7$$

```
Ticketfindemois1(L2,60)
```

$$[1, 15, 2, 7.8, 5, 2.2, 4, 5.5, 2, 2.7, 1, 2.1], [], 50.3$$

Dans `Ticketfindemois1`, c'est `k-1` et non `k`, qui est pas la valeur du nombre d'articles achetés lorsqu'on sort du `tantque`.
En effet, lorsqu'on s'arrête S devient supérieur à a : il ne faut donc pas acheter l'article `L[k]`.
Donc `La:=gauche(L,k-1);` et `Lfin:=droit(L,n-k+1)`.

Remarque
On aurait pu aussi écrire sans utiliser `Somme` mais c'est plus compliqué car la condition du `tantque` porte sur `k` et sur `S`!!! En syntaxe `Xcas` :

```
fonction Ticketfindemois2(L,a)
  local S,n,k,La,Lfin,P;
  n:=dim(L);
  k:=0;
  S:=L[0];
  tantque S<=a et k<n-1 faire
    k:=k+1;
    S:=S+L[k];
  ftantque;
  si S<=a alors retourne L,[],S fsi;
  La:=gauche(L,k);
  Lfin:=droit(L,n-k);
  P:=S-L[k];
 retourne La,Lfin,P;
ffonction:;
```

À chaque étape on a :
Au début, on a :
`k:=0;S:=L[0];` donc `S` est le prix de `1` article.
lorsqu'on fait k fois la boucle on a :
`S:=L[0]+..L[k];` donc `S` est la somme de `k+1` articles.
Quand on sort du `tantque` on a :
soit `S<=a` est vrai, donc `k==n-1` est vrai (puisque `(S<=a et k<n-1)==faux`). `S` est donc la somme de toute la liste `L` i.e. `S` est la somme à payer pour l'achat de `n` articles i.e. Paul peut acheter toute sa liste de courses.
soit `S>a` et `k<=n-1` alors `S` représente la somme des prix des `k+1` premiers articles (`k+1` car on a commencé à l'article `k=0`. Mais Paul ne peut pas acheter le dernier article puisque `S>a`. Le prix `P` représente la somme des prix des `k` premiers articles i.e.des articles `L[0],..L[k-1]` ou encore des artices de la liste `gauche(L,k)` donc `P:=S-L[k]`.

```
L1:=[15,10,2,7,5,1.2,3,5.5,10,4,5.3]
```

$$[15, 10, 2, 7, 5, 1.2, 3, 5.5, 10, 4, 5.3]$$

```
L2:=[1,15,2,7.8,5,2.2,4,5.5,2,2.7,1,2.1]
```

$$[1, 15, 2, 7.8, 5, 2.2, 4, 5.5, 2, 2.7, 1, 2.1]$$

```
Ticketfindemois2(L1,50)
```

$$[15, 10, 2, 7, 5, 1.2, 3, 5.5], [10, 4, 5.3], 48.7$$

```
Ticketfindemois2(L2,60)
```

$$[1, 15, 2, 7.8, 5, 2.2, 4, 5.5, 2, 2.7, 1, 2.1], [], 50.3$$

3.14 Interruption d'une boucle

Si on utilise `retourne` à l'intérieur d'une boucle dans une fonction, celle-ci est interrompue. Ceci permet de transformer des boucles "tantque" en boucle "pour" souvent plus lisibles.

Reprenons l'exemple ci-dessus, on remarque que la boucle tantque utilise un compteur `k` qu'on incrémente à chaque itération comme dans une boucle `pour`. Il est donc naturel d'essayer de réécrire cette fonction avec une boucle `pour`. Il suffira de tester dans le corps de la boucle si la somme (avec le nouvel article) dépasse le contenu du porte-monnaie, dans ce cas il faut s'arrêter sans acheter le nouvel article, on interrompt la boucle et on renvoie les résultats (on calcule la somme au fur et à mesure et donc on n'utilise pas la fonction `Somme`). Dans `Ticketfindemois3`, c'est `k`, qui est pas la valeur du nombre d'articles achetés lorsqu'on sort du `pour`.
En effet, lorsqu'on s'arrête S devient supérieur à a : il ne faut donc pas acheter l'article `L[k]`.
Donc `La:=gauche(L,k);` et `Lfin:=droit(L,n-k)`.
En syntaxe `Xcas` :

```
fonction Ticketfindemois3(L,a)
  local k,n,S,La,Lfin;
  n:=dim(L);
  S:=0;
  pour k de 0 jusque n-1 faire
    si S+L[k]>a alors
      La:=gauche(L,k);
      Lfin:=droit(L,n-k);
      retourne La,Lfin,S;
    fsi;
    S:=S+L[k];
  fpour;
  retourne L,[],S;
ffonction:;
```

```
L1:=[15,10,2,7,5,1.2,3,5.5,10,4,5.3]
```

$$[15, 10, 2, 7, 5, 1.2, 3, 5.5, 10, 4, 5.3]$$

```
L2:=[1,15,2,7.8,5,2.2,4,5.5,2,2.7,1,2.1]
```

$$[1, 15, 2, 7.8, 5, 2.2, 4, 5.5, 2, 2.7, 1, 2.1]$$

```
Ticketfindemois3(L1,50)
```

$$[15, 10, 2, 7, 5, 1.2, 3, 5.5], [10, 4, 5.3], 48.7$$

```
Ticketfindemois3(L2,60)
```

$$[1, 15, 2, 7.8, 5, 2.2, 4, 5.5, 2, 2.7, 1, 2.1], [], 50.3$$

En syntaxe Python :

```
def Ticketfindemois3(L,a):
    # local k,n,S,La,Lfin
    n=dim(L)
    S=0
    k=0
    for k in range(n) :
        if (S+L[k])>a :
            La=gauche(L,k)
            Lfin=droit(L,n-k)
            return La,Lfin,S
        S=S+L[k]
    return(L,[],S)
```

```
L1:=[15,10,2,7,5,1.2,3,5.5,10,4,5.3]
```

$$[15, 10, 2, 7, 5, 1.2, 3, 5.5, 10, 4, 5.3]$$

```
L2:=[1,15,2,7.8,5,2.2,4,5.5,2,2.7,1,2.1]
```

$$[1, 15, 2, 7.8, 5, 2.2, 4, 5.5, 2, 2.7, 1, 2.1]$$

```
Ticketfindemois3(L1,50)
```

$$[15, 10, 2, 7, 5, 1.2, 3, 5.5], [10, 4, 5.3], 48.7$$

```
Ticketfindemois3(L2,60)
```

$$[1, 15, 2, 7.8, 5, 2.2, 4, 5.5, 2, 2.7, 1, 2.1], [], 50.3$$

Cette méthode s'applique pour toute boucle `tantque` dont on peut prévoir à priori un majorant du nombre d'itérations. On peut d'ailleurs aussi l'utiliser si on se fixe un nombre maximal d'itérations qui tient compte du temps d'exécutions, typiquement en `Xcas` de l'ordre du million d'itérations si on veut un résultat en moins de quelques secondes.

Remarque : si on ne veut pas quitter la fonction, il est quand même possible d'interrompre la boucle prématurément en utilisant l'instruction `break`.

3.15 Autre exemple de boucle `tantque` : le ticket de caisse

Pour avoir des clients le dimanche matin, le magasin de Paul offre selon les dimanches une réduction immédiate r qui varie selon le montant a des achats par exemple une réduction de 10 euros dès 60 euros d'achats, ou une réduction de 5 euros dès 50 euros d'achats etc...
Ce magasin ne pratique pas de réduction pour des achats en gros.
Pour être sûr de bénéficier de la réduction, Paul fait sa liste de courses Lc en mettant au début ce qu'il veut vraiement acheter et à la fin de sa liste, il met les achats qu'il doit faire à plus long terme (contrairement au programme précédent, on suppose ici que Paul a suffisamment d'argent).
Attention
On suppose ici que la liste de courses est constituée par une suite des nombres n,P où n est le nombre d'achats du même article de prix P.
Il veut faire un programme qui arrête sa liste dès que $S >= a$ en coupant Lc en 2 listes La liste des objets de ce qu'il achète réellement pour un montant S avant réduction et $Lfin$ liste des objets qu'il n'achète pas ($Lfin$ est éventuellement une liste vide).
Paul veut que son programme ait paramètres Lc, a, r et qu'il renvoie :
La, $Lfin$, S, $S - r$.
Dans ce cas, on ne sait pas au départ combien de fois on doit effectuer la boucle.
Mais on sait quand on doit s'arrêter :
on arrête la boucle lorsque le prix S de la liste complète Lc n'atteint pas le montant a ou dés que le prix S du début de Lc vérifie `S>=a`.

On utilise pour cela une boucle `tantque` :
`tantque <condition> faire <instructions> ftantque;`
Cela veut dire :
tant que "non arrêt", on fait les instructions de la boucle.
"arrêt" se traduit ici par `Lfin ==[] ou S>=a` donc,
"non arrêt" se traduit ici par `Lfin!=[] et S<a`.
Attention la variable k qui va parcourir la liste `L` devra être initialisée (ici `k:=0;`) et modifiée dans le corps de la boucle (ici `k:=k+2;`).
La fonction `Ticketdimanche` a 3 paramètres `L,a,r` et renvoie la liste La des courses qui ont été prises en compte,
la liste $Lfin$ des courses qui n'ont pas été prises en compte (cette liste peut être vide si `S<=a`)
la somme `S` des achats sans la remise et
la somme `S-r` à payer.
En syntaxe `Xcas` :

```
fonction Ticketdimanche(L,a,r)
 local S,n,k,Lfin,La;
 n:=dim(L);
 S:=0;
 k:=0;
 tantque k<n et S<a faire
   S:=S+L[k]*L[k+1];
   k:=k+2;
 ftantque;
 La:=gauche(L,k);
 Lfin:=droit(L,n-k);
 si S<a alors r:=0; fsi;
 retourne La,Lfin,S,S-r;
ffonction:;
```

```
L1:=[1,15,2,7,5,1.2,3,5.5]
```

$$[1, 15, 2, 7, 5, 1.2, 3, 5.5]$$

```
L2:=[1,15,2,7.8,5,2.2,4,5.5,2,2.7,1,2.1]
```

$$[1, 15, 2, 7.8, 5, 2.2, 4, 5.5, 2, 2.7, 1, 2.1]$$

```
Ticketdimanche(L1,50,5)
```

$$[1, 15, 2, 7, 5, 1.2, 3, 5.5], [], 51.5, 46.5$$

```
Ticketdimanche(L2,60,10)
```

$$[1, 15, 2, 7.8, 5, 2.2, 4, 5.5], [2, 2.7, 1, 2.1], 63.6, 53.6$$

En syntaxe Python :

```
def Ticketdimanche(L,a,r) :
    # local S,n,k,Lfin,La
    n=dim(L)
    S=0
    k=0
    while k<n and S<a :
        S=S+L[k]*L[k+1]
        k=k+2
    La=gauche(L,k)
    Lfin=droit(L,n-k)
    if S<a :
        r=0
    return La,Lfin,S,S-r
```

```
L1:=[1,15,2,7,5,1.2,3,5.5]
```

$$[1, 15, 2, 7, 5, 1.2, 3, 5.5]$$

```
L2:=[1,15,2,7.8,5,2.2,4,5.5,2,2.7,1,2.1]
```

$$[1, 15, 2, 7.8, 5, 2.2, 4, 5.5, 2, 2.7, 1, 2.1]$$

```
Ticketdimanche(L1,50,5)
```

$$[1, 15, 2, 7, 5, 1.2, 3, 5.5], [], 51.5, 46.5$$

```
Ticketdimanche(L2,60,10)
```

$$[1, 15, 2, 7.8, 5, 2.2, 4, 5.5], [2, 2.7, 1, 2.1], 63.6, 53.6$$

Traduction du "tantque" en "pour"

On remarque que la boucle "tantque" a un compteur `k`, on peut donc la transformer en boucle "pour" avec sortie prématurée de la boucle lorsque $S > a$.

Attention `gauche(L,k)` représente les `k` premiers éléments de `L` i.e c'est `L[0]..L[k-1]`.

Ici, on sort de la boucle `pour` lorsque `S>=a` donc pour avoir la réduction il faut acheter `L[k]` fois l'article `L[k+1]`, c'est à dire prendre en compte la liste `L[0].. L[k+1]` qui est une liste de `k+2` éléments c'est donc `gauche(L,k+2)`.

En syntaxe `Xcas` :

```
fonction Ticketdimanche1(L,a,r)
 local S,n,k,Lfin,La;
 n:=dim(L);
 S:=0;
 pour k de 0 jusque n-1 pas 2 faire
   S:=S+L[k]*L[k+1];
   si S>=a alors
     La:=gauche(L,k+2);
     Lfin:=droit(L,n-k-2);
     retourne La,Lfin,S,S-r;
   fsi;
 fpour;
 retourne L,[],S,S;
ffonction:;
```

```
L1:=[1,15,2,7,5,1.2,3,5.5]
```

$$[1, 15, 2, 7, 5, 1.2, 3, 5.5]$$

```
L2:=[1,15,2,7.8,5,2.2,4,5.5,2,2.7,1,2.1]
```

$$[1, 15, 2, 7.8, 5, 2.2, 4, 5.5, 2, 2.7, 1, 2.1]$$

```
Ticketdimanche1(L1,50,5)
```

$$[1, 15, 2, 7, 5, 1.2, 3, 5.5], [], 51.5, 46.5$$

```
Ticketdimanche1(L2,60,10)
```

$$[1, 15, 2, 7.8, 5, 2.2, 4, 5.5], [2, 2.7, 1, 2.1], 63.6, 53.6$$

En syntaxe Python :

```
def Ticketdimanche1(L,a,r):
    # local S,n,k,Lfin,La
    n=dim(L)
    S=0
    for k in range(0,n,2):
        S=S+L[k]*L[k+1]
        if S>=a :
            La=gauche(L,k+2)
            Lfin=droit(L,n-k-2)
            return La,Lfin,S,S-r
    return L,[],S,S
```

```
L1:=[1,15,2,7,5,1.2,3,5.5]
```

$$[1, 15, 2, 7, 5, 1.2, 3, 5.5]$$

```
L2:=[1,15,2,7.8,5,2.2,4,5.5,2,2.7,1,2.1]
```

$$[1, 15, 2, 7.8, 5, 2.2, 4, 5.5, 2, 2.7, 1, 2.1]$$

```
Ticketdimanche1(L1,50,5)
```

$$[1, 15, 2, 7, 5, 1.2, 3, 5.5], [], 51.5, 46.5$$

```
Ticketdimanche1(L2,60,10)
```

$$[1, 15, 2, 7.8, 5, 2.2, 4, 5.5], [2, 2.7, 1, 2.1], 63.6, 53.6$$

3.16 Encore un exemple de boucle tantque : le ticket de caisse

Maintenant le magasin de Paul favorise aussi les achats en gros : 10% de réduction lorsque on achète 3 fois le même produit. En plus il offre selon les dimanches une réduction immédiate r qui varie selon le montant a des achats, par exemple une réduction de 10 euros dès 60 euros d'achats.
Modifier les programmes précédents pour tenir compte des achats en gros. On utilise ici la fonction `Prix` écrite précédemment (cf 3.9).
Attention comme précédemment on suppose ici que la liste de courses est constituée par une suite des nombres n,P où n est le nombre d'achats du même article de prix P.
En syntaxe `Xcas` :

```
fonction Ticketdimgros(L,a,r)
 local S,n,k,Lfin,La;
 n:=dim(L);
 S:=0;
 k:=0;
 tantque k < n et S<a faire
   S:=S+Prix(L[k],L[k+1]);
   k:=k+2;
 ftantque;
 La:=gauche(L,k);
 Lfin:=droit(L,n-k);
 si S < a alors r:=0 ;fsi;
 retourne La,Lfin,S,S-r;
ffonction:;
```

```
L:=[1,15,2,7.8,5,2.2,4,5.5,2,2.7,1,2.1]
```

$$[1, 15, 2, 7.8, 5, 2.2, 4, 5.5, 2, 2.7, 1, 2.1]$$

```
Ticketdimgros(L,60,10)
```

$$[1, 15, 2, 7.8, 5, 2.2, 4, 5.5], [2, 2.7, 1, 2.1], 60.3, 50.3$$

```
1*15+2*7.8+5*2.2*0.9+4*5.5*0.9
```

$$60.3$$

En syntaxe Python :

```
def Ticketdimgros(L,a,r) :
    # local S,n,k,Lfin,La
    n=dim(L)
    S=0
    k=0
    while k < n and S<a :
        S=S+Prix(L[k],L[k+1])
        k=k+2
    La=gauche(L,k)
    Lfin=droit(L,n-k)
    if S < a :
        r=0
    return La,Lfin,S,S-r
```

```
L:=[1,15,2,7.8,5,2.2,4,5.5,2,2.7,1,2.1]
```

$$[1, 15, 2, 7.8, 5, 2.2, 4, 5.5, 2, 2.7, 1, 2.1]$$

```
Ticketdimgros(L,60,10)
```

$$[1, 15, 2, 7.8, 5, 2.2, 4, 5.5], [2, 2.7, 1, 2.1], 60.3, 50.3$$

```
1*15+2*7.8+5*2.2*0.9+4*5.5*0.9
```

$$60.3$$

Transformation en boucle pour En syntaxe `Xcas` :

```
fonction Ticketdimgros1(L,a,r)
 local S,n,k,Lfin,La;
 n:=dim(L);
 S:=0;
```

```
 pour k de 0 jusque n-1 pas 2 faire
   S:=S+Prix(L[k],L[k+1]);
   si S>=a alors
     La:=gauche(L,k+2);
     Lfin:=droit(L,n-k-2);
     retourne La,Lfin,S,S-r;
   fsi;
 fpour;
 retourne La,[],S,S;
ffonction:;
```

```
   L:=[1,15,2,7.8,5,2.2,4,5.5,2,2.7,1,2.1]
```

$$[1, 15, 2, 7.8, 5, 2.2, 4, 5.5, 2, 2.7, 1, 2.1]$$

```
   Ticketdimgros1(L,60,10)
```

$$[1, 15, 2, 7.8, 5, 2.2, 4, 5.5], [2, 2.7, 1, 2.1], 60.3, 50.3$$

```
   1*15+2*7.8+5*2.2*0.9+4*5.5*0.9
```

$$60.3$$

En syntaxe Python :

```
def Ticketdimgros1(L,a,r):
    # local S,n,k,Lfin,La
    n=dim(L)
    S=0
    for k in range(0,n,2):
        S=S+Prix(L[k],L[k+1])
        if S>=a :
            La=gauche(L,k+2)
            Lfin=droit(L,n-k-2)
            return La,Lfin,S,S-r
    return La,[],S,S
```

```
   L:=[1,15,2,7.8,5,2.2,4,5.5,2,2.7,1,2.1]
```

$$[1, 15, 2, 7.8, 5, 2.2, 4, 5.5, 2, 2.7, 1, 2.1]$$

```
Ticketdimgros1(L,60,10)
```

$$[1, 15, 2, 7.8, 5, 2.2, 4, 5.5], [2, 2.7, 1, 2.1], 60.3, 50.3$$

```
1*15+2*7.8+5*2.2*0.9+4*5.5*0.9
```

$$60.3$$

3.17 Exercice : Algorithme de tracé de courbe

Soit la fonction `f` définie sur `[a,b]`.
On veut tracer le graphe de cette fonction sur l'intervalle `[a,b]`.
En partageant `[a,b]` en `n` parties égales on obtient :
`a=a0,a1=a+h,a2=a+2h, ...b=a+n*h` avec `h=(b-a)/n`. Le graphe sera obtenu en reliant les points de coordonnées `[a f(a)] [a1 f(a1)]` etc ... par des segments.
En syntaxe `Xcas` :

```
fonction Graphe(f,a,b,n)
 local L,h,k;
 L:=NULL;
 h:=(b-a)/n;
 pour k de 0 jusque n-1 faire
   L:=L,segment(point(a,f(a)),point(a+h,f(a+h)));
   a:=a+h;
 fpour;
 retourne L;
ffonction:;
```

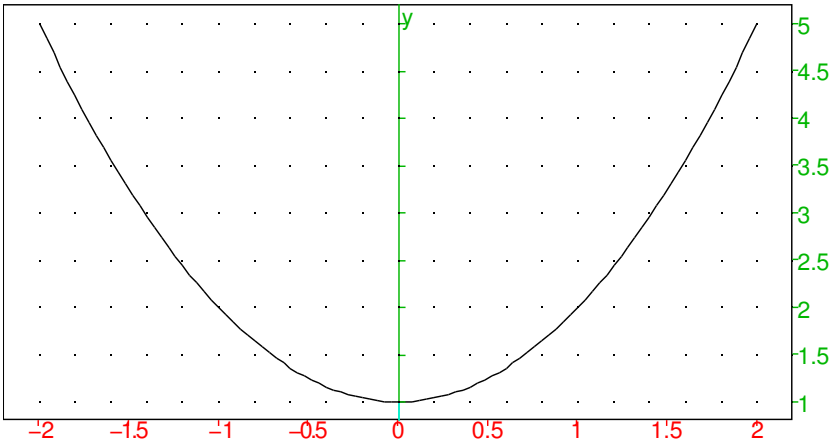
```
f(x):=x^2+1
```

```
 (x)->x^2+1
```

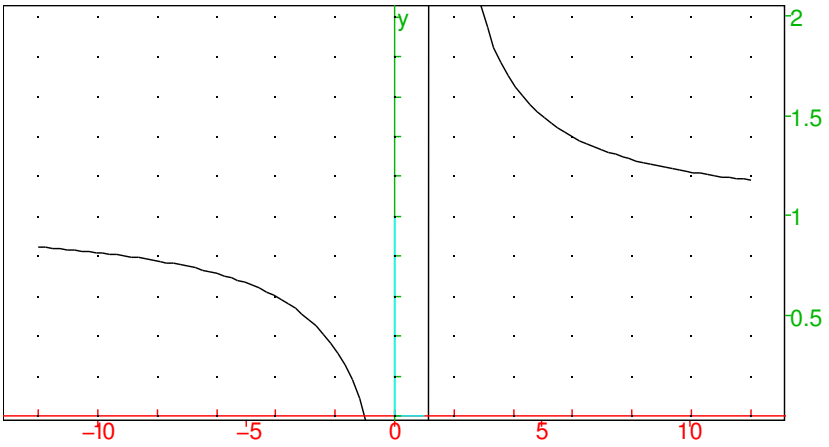
```
g(x):=(x+1)/(x-1)
```

```
 (x)->(x+1)/(x-1)
```

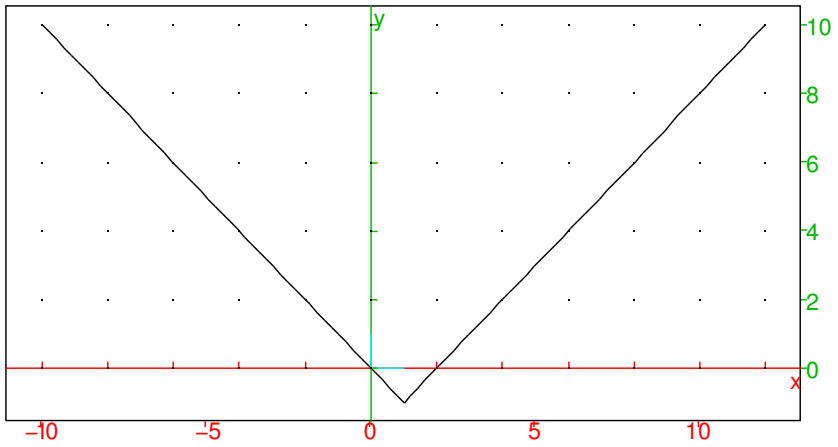
```
Graphe(f,-2,2,100)
```




```
Graphe(g,-12,12,100)
```




```
Graphe(Abs1,-10,12,100)
```



En syntaxe Python :

```
def Graphe(f,a,b,n):
    # local L,h,k
    L=NULL
    h=(b-a)/n
    for k in range(n):
        L=L,segment(point(a,f(a)),point(a+h,f(a+h)))
        a=a+h
    return L
```

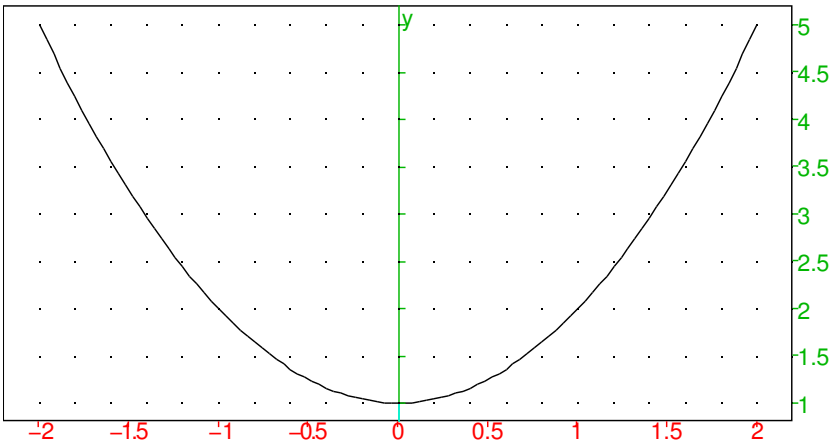
```
f(x):=x^2+1
```

```
 (x)->x^2+1
```

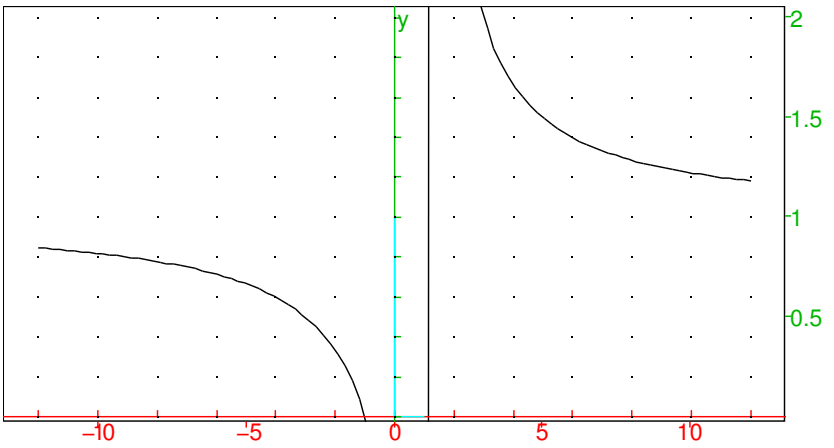
```
g(x):=(x+1)/(x-1)
```

```
 (x)->(x+1)/(x-1)
```

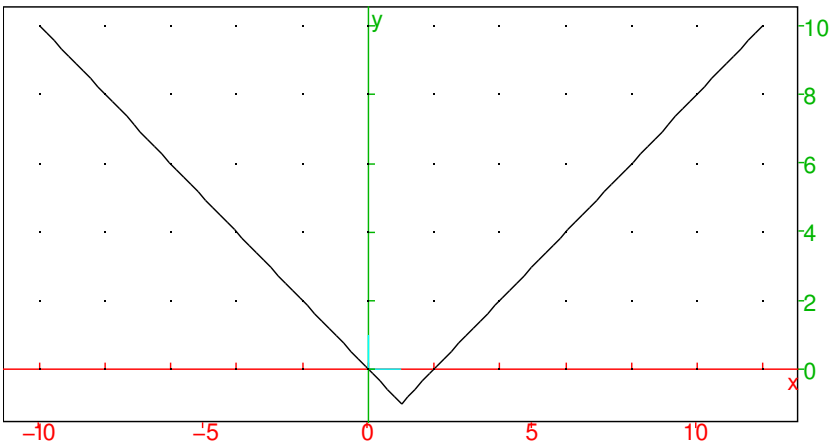
```
Graphe(f,-2,2,100)
```



`Graphe(g,-12,12,100)`



`Graphe(Abs1,-10,12,100)`



3.18 Mettre au point un programme

Lorsqu'on écrit un programme, il y a deux étapes à franchir :

- Écrire un programme syntaxiquement correct. Quand on tape sur le bouton ok, on `compile` le programme (plus préciśement il est interprété). Les erreurs à ce stade sont appelées **erreurs de syntaxe**.
  Certaines interfaces de `Xcas` essaient de placer le curseur à l'endroit de l'erreur, d'autres affichent le numéro de ligne et colonne de l'erreur. Il faut garder en tête que l'erreur peut être située avant l'endroit indiqué par le curseur ou dans le message d'erreur, l'endroit indiqué est le premier endroit où l'interpréteur ne comprend plus la syntaxe.
- S'assurer que le programme fait bien ce qu'on attend de lui. Si ce n'est pas le cas on parle d'**erreur d'exécution** (en anglais runtime error). Pour corriger ce type d'erreur, `Xcas` propose une commande appelée `debug`. Au lieu d'appeler directement le programme, on le place dans `debug()`, par exemple au lieu de taper `Graphe(f,-2,2,100)` on tape `debug(Graphe(f,-2,2,100))` Le debogueur affiche alors la liste des variables locales et leur valeur ainsi que la prochaine ligne du programme qui sera exécutée. On peut alors exécuter ligne par ligne le programme avec des arguments pour lequel on connait la réponse, regarder l'évolution des valeurs des variables et comprendre plus facilement pourquoi le programme ne fait pas ce qui est attendu.

Chapitre 4

Résolution d'équations

4.1 Encadrer une racine d'une équation par dichotomie

Algorithme de dichotomie

On suppose que la fonction f est continue sur l'intervalle $[a, b]$ et que $f(a) < 0$ et $f(b) > 0$ (si $f(a) > 0$ et $f(b) < 0$ on peut se ramener au cas précédent en échangeant a et b). On en déduit que f s'annule pour $x = x_0$ avec $a < x_0 < b$, on cherche une valeur approchée de x_0.

Pour avoir une meilleur approximation de x_0, on cherche le signe de $f(c)$ où $c = \frac{a+b}{2}$ est le milieu de $[a, b]$. Si $f(c) = 0$ alors $x_0 = c$ et on est content ! Si $f(c) < 0$, il est de même signe que $f(a)$, on peut donc remplacer a par c, sinon $f(c) > 0$ est de même signe que $f(b)$, on peut remplacer b par c. On recommence le processus jusquà ce que $f(c) = 0$ ou $|b - a| < 10^{-n}$ (avec par exemple $n = 3$).

```
fonction Dichotomie0(f,a,b,n)
 local c;
 si f(a)*f(b)>0 alors retourne [] fsi;
 si f(a)==0 alors retourne [a] fsi;
 si f(b)==0 alors retourne [b] fsi;
 si f(a)>0 alors a,b:=b,a; fsi; // echange a et b pour avoir f(a)<=0
 tantque abs(b-a)>10^(-n) faire
   c:=evalf(a+b)/2;
   si f(c)=0 alors retourne [c] fsi;
   si f(c)<0 alors a:=c; sinon b:=c; fsi; // on a f(a)<=0
 ftantque
 retourne [c];
ffonction:;
```

Notez le test d'arrêt qui utilise $|b-a|$ car a et b ont peut-être été échangés. Exemple :

```
f(x):=x^2-2
```

```
 (x)->x^2-2
```

```
Dichotomie0(f,1,2,12)
```

[1.41421356237]

```
g(x):=cos(x)-x
```

```
 (x)->cos(x)-x
```

```
Dichotomie0(g,0.,1.,12)
```

[0.739085133215]

On peut rajouter en début de programme un test sur `n` pour que le nombre d'itérations ne soit pas trop grand, par exemple
`si n>12 alors n:=12; fsi;`
On peut aussi utiliser une variable locale pour ne faire qu'une seule fois le calcul de 10^{-n} et de $f(c)$. Ce qui donne le programme suivant :

```
fonction Dichotomie1(f,a,b,n)
 local c,fc,eps;
 si f(a)*f(b)>0 alors retourne [] fsi;
 si f(a)==0 alors retourne [a] fsi;
 si f(b)==0 alors retourne [b] fsi;
 si f(a)>0 alors a,b:=b,a; fsi; // echange a et b pour avoir f(a)<=0
 si n>12 alors n:=12; fsi;
 eps:=10^(-n);
 tantque abs(b-a)>eps faire
   c:=evalf(a+b)/2;
   fc:=f(c);
   si fc=0 alors retourne [c] fsi;
```

```
   si fc<0 alors a:=c; sinon b:=c; fsi; // on a f(a)<0
 ftantque
 retourne [c];
ffonction:;
```

```
   g(x):=cos(x)-x
```

```
    (x)->cos(x)-x
```

```
   Dichotomie1(g,0.,1.,12)
```

[0.739085133215]

Il n'est pas indispensable de tester que $f(a) = 0$, $f(b) = 0$ et $f(c) = 0$ (comme cela n'arrive pratiquement jamais, c'est donc inefficace de le faire), on peut supprimer ces lignes. Il faut alors utiliser $f(c) \leq 0$ comme test (au lieu de $f(c) < 0$) pour conserver comme invariant de boucle $f(a) \leq 0$ et $f(b) > 0$.

```
fonction Dichotomie(f,a,b,n)
 local c,eps;
 si f(a)*f(b)>0 alors retourne [] fsi;
 si f(a)>0 alors a,b:=b,a; fsi; // echange a et b pour avoir f(a)<=0
 si n>12 alors n:=12; fsi;
 eps:=10^(-n);
 tantque abs(b-a)>eps faire
   c:=evalf(a+b)/2; // invariant f(a)<=0 et f(b)>0
   si f(c)<=0 alors a:=c; sinon b:=c; fsi;
 ftantque
 retourne [c];
ffonction:;
```

```
   Dichotomie(g,0.,1.,12)
```

[0.739085133215]

Traduction du "tantque" en "pour"

On observe qu'à chaque itération de la boucle on divise la longueur de l'intervalle par 2, le nombre d'itérations ne peut pas être très grand. On peut donc transformer la boucle "tantque" en boucle "pour" en se fixant à priori

un nombre maximal d'itérations ce qui évitera d'ailleurs d'avoir une boucle qui ne se termine jamais. On montrera plus bas que 2100 itérations suffisent en calcul approché.

```
fonction Dichotomie2(f,a,b,n)
 local c,k,eps;
 eps:=10^-n;
 si f(a)*f(b)>0 alors retourne []; fsi;
 pour k de 1 jusque 2100 faire
   c:=(a+b)/2.0; // invariant f(a)*f(b)<=0
   si b-a<eps alors retourne [c]; fsi;
   si f(a)*f(c)<=0 alors b:=c; sinon a:=c; fsi;
 fpour;
 retourne [c];
ffonction:;
```

```
   g(x):=cos(x)-x
```

```
    (x)->cos(x)-x
```

```
   Dichotomie2(g,0.,1.,12)
```

[0.739085133215]

Exercice : modifier le programme pour ne pas faire le calcul de $f(a)$ dans la boucle.

Remarque avancée : si on connaît les logarithmes, on peut calculer le nombre d'itérations N pour que $|b-a|/2^N < 10^{-n}$ en résolvant cette équation. On peut aussi observer que les calculs se font en approché, dans `Xcas` le plus grand nombre représentable par défaut est `evalf(2^(1024-1)`, donc la taille du plus grand intervalle est (légèrement inférieure à) 2^{1025}. Le plus petit réel strictement positif représentable est `evalf(2^(-1069))`. Comme on divise par 2 la taille de l'intervalle à chaque itération, le nombre maximal d'itérations est au plus 1025+1069=2094 [1]. Au-delà, soit a et b sont représentés par le même nombre flottant (et le test $|b-a| < 10^{-n}$ sera donc vrai) soit ils ne différeront que par leur dernier bit de mantisse, et dans ce cas $c = (a+b)/2$ sera arrondi vers a ou vers b et la boucle `tanque` continuera

1. Il faut ajouter 5 pour un langage traditionnel où la mantisse a 53 chiffres significatifs. Attention, ceci n'est plus valable dans `Xcas` si on modifie la valeur de `Digits`

indéfiniment.
La majoration est le plus souvent très pessimiste, par exemple si $a = 1$ et $b = 2$ ils sont déjà représentés avec le même exposant et le nombre d'itérations sera limité par 48.

```
fonction Dichotomie3(f,a,b,n)
 local c,k,N;
 si f(a)*f(b)>0 alors retourne []; fsi;
 N:=ceil(log((b-a)/10^(-n))/log(2));
 si N>2100 alors N:=2100 fsi;
 pour k de 1 jusque N faire
   c:=(a+b)/2.0; // f(a)*f(b)<=0
   si f(a)*f(c)<=0 alors b:=c; sinon a:=c; fsi;
 fpour;
 retourne [c];
ffonction:;
```

```
   g(x):=cos(x)-x
```

```
    (x)->cos(x)-x
```

```
   Dichotomie3(g,0.,1.,12)
```

[0.739085133216]

Exercice : Modifier la fonction ci-dessus pour calculer f une seule fois par itération, c'est-à-dire qu'on calcule c et $f(c)$ mais qu'on ne recalcule pas $f(a)$.
Première méthode :
On introduit 3 variables locales `fa, fb, fc` contenant les valeurs de $f(a)$, $f(b)$, $f(c)$. On en profite pour tester au passage si $f(c) = 0$.

```
fonction Dichotomie4(f,a,b,n)
 local c,k,N,fa,fb,fc;
 fa:=f(a);
 fb:=f(b);
 si fa*fb>0 alors retourne []; fsi;
 si fa==0 alors retourne [a] fsi;
 si fb==0 alors retourne [b] fsi;
 N:=ceil(log((b-a)/10^(-n))/log(2));
 si N>2100 alors N:=2100 fsi;
```

```
 pour k de 1 jusque N faire
   c:=(a+b)/2.0;
   fc:=f(c);
   si fc==0 alors retourne [c] fsi;
   si fa*fc<0 alors b:=c; sinon a:=c; fa:=fc; fsi;
 fpour;
 retourne [c];
ffonction:;
```

```
   g(x):=cos(x)-x

    (x)->cos(x)-x
```

```
   Dichotomie4(g,0.,1.,12)
```

[0.739085133216]

Deuxième méthode :
On échange le rôle de a et b si $f(a) > 0$.

```
fonction Dichotomie5(f,a,b,n)
 local c,k,N;
 si f(a)*f(b)>0 alors retourne []; fsi;
 N:=ceil(log(abs(b-a)/10^(-n))/log(2));
 si N>2100 alors N:=2100 fsi;
 si f(a)>0 alors a,b:=b,a; fsi;
 pour k de 1 jusque N faire
   c:=(a+b)/2.0; // invariant f(a)<=0 et f(b)>0
   si f(c)<=0 alors a:=c; sinon b:=c; fsi;
 fpour;
 retourne [c];
ffonction:;
```

```
   g(x):=cos(x)-x

    (x)->cos(x)-x
```

```
Dichotomie5(g,0.,1.,12)
```

[0.739085133216]

Ce programme est optimal.

On peut vérifier ces résultats en utilisant la commande `fsolve` de `Xcas` qui effectue la résolution numérique d'une équation :

```
fsolve(cos(x)=x,x,0..1)
```

[0.739085133215]

4.2 Résoudre dans $\mathbb{R}$ une équation se ramenant au premier degré ou au degré 2

On considère une équation qui se ramène au premier ou au deuxième degré.
Si cette équation se ramène au premier degré, elle est de la forme :

`a*x+b=0` avec `a!=0`

donc cette équation a une solution qui est :
`x0:=-b/a`.
Si cette équation se ramène au deuxième degré, elle est de la forme :

`a*x^2+b*x+c=0` avec `a!=0`

donc :

- si Δ=`b^2-4*a*c>0` il y a 2 solutions qui sont :
  `x1:=(-b+`$\sqrt{\Delta}$`)/(2*a)` et `x2:=(-b-`$\sqrt{\Delta}$`)/(2*a)`.
- si Δ=`b^2-4*a*c=0` il y a 1 solution qui est :
  `x1` et `x1:=-b/(2*a)`
- si Δ=`b^2-4*a*c<0` il n'y a pas de solution réelle.

On tape :

```
fonction Solution12(Eq,Var)
 local a,b,c,d;
 Eq:=normal(gauche(Eq)-droit(Eq));
 si degree(Eq,Var)==0 alors
   si (Eq==0) alors retourne "infinite de solution" ;
   sinon retourne "pas de solution" ;
```

```
   fsi;
 fsi;
 si degree(Eq,Var)==1 alors
   //a:=coeff(Eq,Var,1);b:=coeff(Eq,Var,0);
   b:=subst(Eq,Var=0);
   a:=subst(Eq,Var=1)-b;
   retourne normal([-b/a]);
 fsi;
 si degree(Eq,Var)==2 alors
   //a:=coeff(Eq,Var,2);b:=coeff(Eq,Var,1);c:=coeff(Eq,Var,0);
   c:=subst(Eq,Var=0);
   d:=subst(Eq,Var=1);
   b:=(d-subst(Eq,Var=-1))/2;
   a:=d-b-c;
   d:=b^2-4*a*c;
   si d>0 alors retourne simplify([(-b+sqrt(d))/(2*a),(-b-sqrt(d))/(2*a)]);fs
   si d==0 alors retourne simplify([-b/(2*a)]); fsi;
   retourne [];
 fsi;
 retourne "degree >2";
ffonction:;
```

```
   purge(x)
```

Nosuchvariablex

```
   Solution12(2*x^2+2*x+2=x^2+2*x+4,x)
```

$$[\sqrt{2}, -\sqrt{2}]$$

```
   Solution12(2*x^2+2*x+2=2x^2+x+4,x)
```

$$[2]$$

```
   Solution12(2*x^2+2*x+1=-2*x+31,x)
```

$$[3, -5]$$

```
Solution12(2*x^2+2*x+1=-2*x-2,x)
```

[]

4.3 Résoudre un système de deux équations du premier degré à deux inconnues.

On veut résoudre le système de deux équations du premier degré

$$a_1x + b_1y + c_1 = 0, \quad a_2x + b_2y + c_2 = 0$$

à deux inconnues x, y. On notera `a1, b1, c1, a2, b2, c2` les coefficients des équations dans les programmes.
Pour éviter d'étudier des cas particuliers inintéressants, on va supposer que $(a_1, b_1) \neq (0, 0)$ et $(a_2, b_2) \neq (0, 0)$. Dans ce cas $a_1x + b_1y + c_1 = 0$ et $a_2x + b_2y + c_2 = 0$ sont les équations de 2 droites D_1 et D_2.

Solution géométrique

- Si D_1 et D_2 sont concourantes, il y a une seule solution. Pour la déterminer, on peut utiliser la commande `solve` de `Xcas`

  ```
  solve([a1*x+b1*y+c1=0,a2*x+b2*y+c2=0],[x,y])
  ```

  $$\left(\begin{array}{cc} \frac{(\mathrm{b1}\cdot\mathrm{c2}-\mathrm{b2}\cdot\mathrm{c1})}{(\mathrm{a1}\cdot\mathrm{b2}-\mathrm{a2}\cdot\mathrm{b1})} & \frac{(-\mathrm{a1}\cdot\mathrm{c2}+\mathrm{a2}\cdot\mathrm{c1})}{(\mathrm{a1}\cdot\mathrm{b2}-\mathrm{a2}\cdot\mathrm{b1})} \end{array} \right)$$

  On justifiera ce résultat plus bas.
- Si D_1 et D_2 sont parallèles il n'y a pas de solution, sauf si D_1 et D_2 sont confondues, il y a une infinité de solutions. On va montrer que cela se produit si et seulement si $a_1b_2 - a_2b_1 \neq 0$.
  En effet, D_1 et D_2 sont parallèles lorsque les coefficients $[a_1, b_1]$ et $[a_2, b_2]$ sont proportionnels i.e si il existe $k \neq 0$ tel que :
  $[a_1, b_1] = k[a_2, b_2] = [ka_2, kb_2]$ ce qui entraine :

  $$(b_1a_2 - b_2a_1) = kb_2a_2 - kb_2a_2 = 0$$

  Réciproquement, si $(b_1a_2 = b_2a_1)$ alors D_1 et D_2 sont parallèles ou confondues.
  En effet :

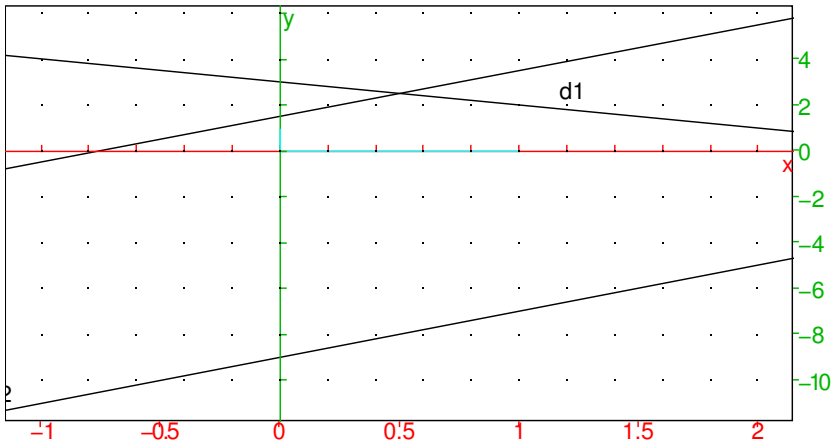
- Si $b_1 = 0$ (resp $a_1 = 0$) alors $b_2 = 0$ (resp $a_2 = 0$) puisque $[a_1, b_1] \neq [0, 0]$ et $(b_1 a_2 = b_2 a_1)$, donc D_1 et D_2 sont parallèles à l'axe des x (resp y).
- Si $b_1 \neq 0$ et $a_1 \neq 0$ alors $b_2/b_1 == a_2/a_1 = k$ ce qui signifie que D_1 et D_2 sont parallèles ou confondues (D_1 et D_2 sont confondues lorsque les coefficients $[a_1, b_1, c_1]$ et $[a_2, b_2, c_2]$ sont proportionnels i.e si il existe $k \neq 0$ tel que :
  $[a_1, b_1, c_1] = k[a_2, b_2, c_2] = [ka_2, kb_2, kc_2]$).

On commence par écrire un programme dans le cas où les droites D_1 et D_2 sont concourrantes.

```
fonction Intersection1(Eq1,Eq2,Var1,Var2)
 local a1,b1,c1,a2,b2,c2;
 Eq1:=normal(gauche(Eq1)-droit(Eq1));
 Eq2:=normal(gauche(Eq2)-droit(Eq2));
 a1:=coeff(Eq1,Var1,1);
 a2:=coeff(Eq2,Var1,1);
 b1:=coeff(Eq1,Var2,1);
 b2:=coeff(Eq2,Var2,1);
 si normal(a1*b2-a2*b1)==0 alors
   retourne "Cas non traite : Eq1 ou Eq2 n'est pas une  "+
          "equation de droite ou droites paralleles";
 fsi;
 c1:=subst(Eq1,[Var1,Var2],[0,0]);
 c2:=subst(Eq2,[Var1,Var2],[0,0]);
 print("droites concourantes");
 retourne [normal((-b2*c1+b1*c2)/(a1*b2-a2*b1)),
           normal((a2*c1-a1*c2)/(a1*b2-a2*b1))];
ffonction:;

   purge(x,y)
```

Nosuchvariablex, Nosuchvariabley

```
d1:=droite(x+y=3);d2:=droite(2x-y=9);d3:=droite
(-4x+2y=3)
```




```
Intersection1(x+y=3,2x-y=9,x,y)
```

$$[4, -1]$$

```
Intersection1(4x-2y+9=0,2x-y=3,x,y)
```

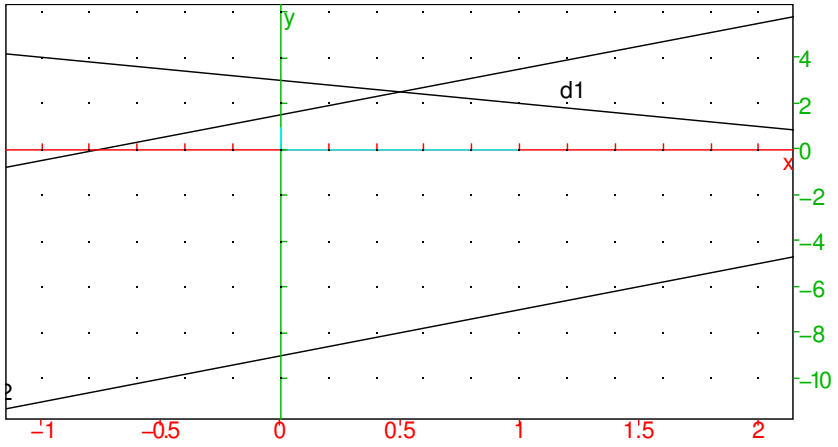
Casnontraite : Eq1ouEq2n′estpasuneequationdedroiteoudroitesparalleles

Exercice : modifiez le programme ci-dessus pour éviter de calculer plusieurs fois $a_1b_2 - a_2b_1$, en stockant sa valeur dans une variable locale.

Correction de l'exercice :

```
fonction Intersection2(Eq1,Eq2,Var1,Var2)
 local a1,b1,c1,a2,b2,c2,d;
 Eq1:=normal(gauche(Eq1)-droit(Eq1));
 Eq2:=normal(gauche(Eq2)-droit(Eq2));
 a1:=coeff(Eq1,Var1,1);
 a2:=coeff(Eq2,Var1,1);
 b1:=coeff(Eq1,Var2,1);
 b2:=coeff(Eq2,Var2,1);
 d:=normal(a1*b2-a2*b1);
 si d==0 alors
   retourne "Cas non traite : Eq1 ou Eq2 n'est pas une"+
    " equation de droite ou droites paralleles ou confondues";
 fsi;
```

```
 c1:=subst(Eq1,[Var1,Var2],[0,0]);
 c2:=subst(Eq2,[Var1,Var2],[0,0]);
 print("droites concourantes");
 retourne [normal((-b2*c1+b1*c2)/d),
           normal((a2*c1-a1*c2)/d)];
ffonction:;

d1:=droite(x+y=3);d2:=droite(2x-y=9);d3:=droite
(-4x+2y=3)
```




```
Intersection2(x+y=3,2x-y=9,x,y)
```

$$[4, -1]$$

```
Intersection2(4x-2y+9=0,2x-y=3,x,y)
```

Casnontraite : Eq1ouEq2n′estpasuneequationdedroiteoudroitesparallelesouconfondues

Voici maintenant un programme qui teste que les équations entrées sont bien des équations de droite et traite aussi le cas des droites parallèles ou confondues :

```
fonction Intersection(Eq1,Eq2,Var1,Var2)
 local a1,b1,c1,a2,b2,c2,d;
 Eq1:=normal(gauche(Eq1)-droit(Eq1));
 si degree(Eq1,Var1)>1 et degree(Eq1,Var2)>1  alors
```

```
  retourne "pas de degre 1";
 fsi;
 Eq2:=normal(gauche(Eq2)-droit(Eq2));
 si degree(Eq2,Var1)>1 et degree(Eq2,Var2)>1 alors
  retourne "pas de degre 1";
 fsi;
 a1:=coeff(Eq1,Var1,1);
 a2:=coeff(Eq2,Var1,1);
 b1:=coeff(Eq1,Var2,1);
 b2:=coeff(Eq2,Var2,1);
 si [a1,b1]==[0,0] ou [a2,b2]==[0,0] alors
  retourne "Eq1 ou Eq2 est nulle";
 fsi;
 c1:=subst(Eq1,[Var1,Var2],[0,0]);
 c2:=subst(Eq2,[Var1,Var2],[0,0]);
 d:=normal(a1*b2-a2*b1);
 si d!=0 alors
  print("droites concourantes");
  retourne [normal((-b2*c1+b1*c2)/d),
            normal((a2*c1-a1*c2)/d)];
 fsi;
 si a1!=0 et a2!=0 alors
  si c1*a2-c2*a1==0 alors
   print("droites confondues");
   retourne [normal(-c1/a1),Var2];
  sinon
   print("droites paralleles");
   retourne [] ;
  fsi;
 fsi;
 si b1!=0 et b2!=0 alors
  si c1*b2==c2*b1 alors
   print("droites confondues");
   retourne [Var1,normal(-c1/b1)];
  sinon
   print("droites paralleles");
   retourne [];
  fsi;
 fsi;
ffonction:;
```

```
Intersection(x+y=3,2x-y=9,x,y)
```

$$[4,-1]$$

```
Intersection(4x-2y+3=0,2x-y=9,x,y)
```

$$[]$$

On vérifie avec `Xcas` :

```
coordonnees(inter_unique(d1,d2))
```

$$[4,-1]$$

```
est_parallele(d2,d3)
```

$$1$$

Justification de la solution lorsque D_1 et D_2 sont concourantes
On a vu que c'était le cas si et seulement si $b_1a_2 - b_2a_1 \neq 0$.

- Si $b_1 \neq 0$ on a $y = (-a_1x - c_1)/b_1$ donc l'abscisse du point d'intersection de D_1 et D_2 vérifie :
  $b_1a_2x + b_2(-a_1x - c_1) + b_1c_2 = 0$
  donc $(b_1a_2 - b_2a_1)x + b_1c_2 - b_2c_1 = 0$ et finalement :

$$x = \frac{b_1c_2 - b_2c_1}{b_2a_1 - b_1a_2}$$

  On remplace dans l'expression de y en fonction de x :

$$y = \frac{1}{b_1}(-a_1\frac{b_1c_2 - b_2c_1}{b_2a_1 - b_1a_2} - c_1) = \frac{1}{b_1}(\frac{-a_1b_1c_2 + a_1b_2c_1 - c_1(b_2a_1 - b_1a_2)}{b_2a_1 - b_1a_2})$$

  Donc

$$y = \frac{-a_1b_1c_2 + c_1b_1a_2}{b_1(b_2a_1 - b_1a_2)} = \frac{-a_1c_2 + c_1a_2}{b_2a_1 - b_1a_2}$$

– Si $b_1 = 0$, on va voir que les mêmes formules s'appliquent. En effet l'abscisse du point d'intersection de D_1 et D_2 vérifie $a_1x + c_1 = 0$ donc

$$x = -\frac{c_1}{a_1} = \frac{b_1c_2 - b_2c_1}{b_2a_1 - b_1a_2}$$

Comme $b_2 \neq 0$, on a

$$y = \frac{-a_2x - c_2}{b_2} = \frac{a_2\frac{c_1}{a_1} - c_2}{b_2} = \frac{a_2c_1 - a_1c_2}{a_1b_2} = \frac{-a_1c_2 + c_1a_2}{b_2a_1 - b_1a_2}$$

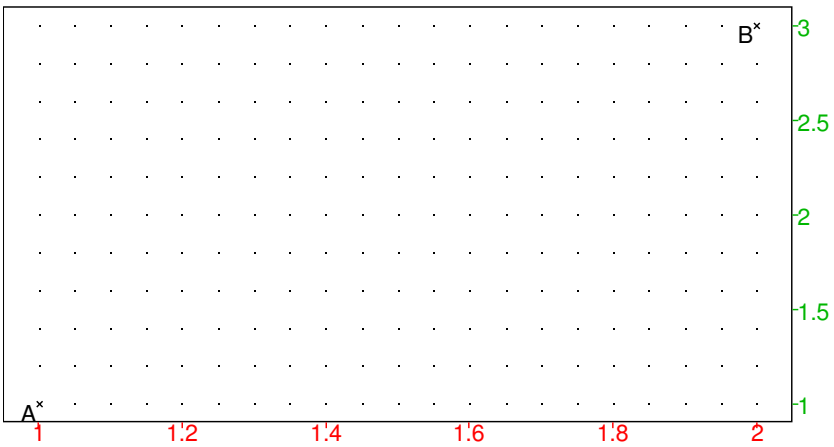
Chapitre 5

Les figures en géométrie plane avec Xcas

5.1 Le point : point et le segment : segment

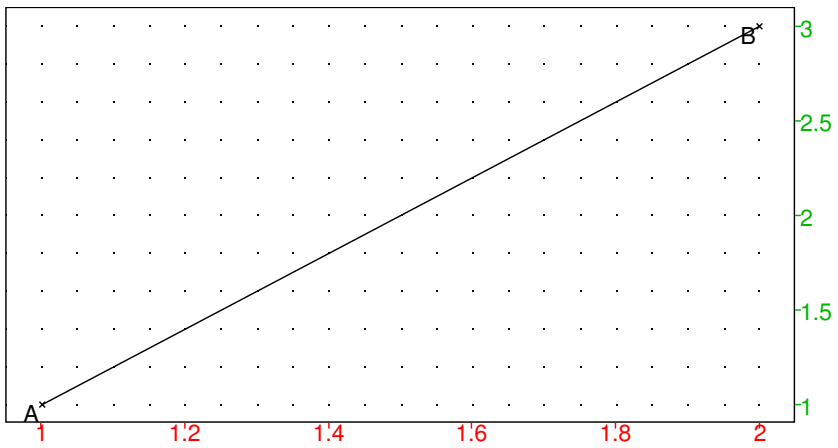
`point` a comme arguments l'abscisse et l'ordonnée du point.
`point` trace le point à l'aide d'une croix sur l'écran de géométrie 2d.
Si on a donné un nom au point (par ex `A:=point(1,1);` ou `A:=point([1,1]);`) ce nom sera affiché à côté de la croix.

```
A:=point(1,1);B:=point(2,3);
```



`segment` a comme argument 2 points.
`segment` trace le segment reliant ces 2 points sur l'écran de géométrie 2d.

```
A:=A;B:=B;segment(A,B);
```



5.2 Les coordonnées d'un point : coordonnees

`coordonnees` a comme argument 1 point.
`coordonnees` renvoie la liste constitué de l'abscisse et de l'ordonnée du point.

```
coordonnees(A)
```

$$[1,1]$$

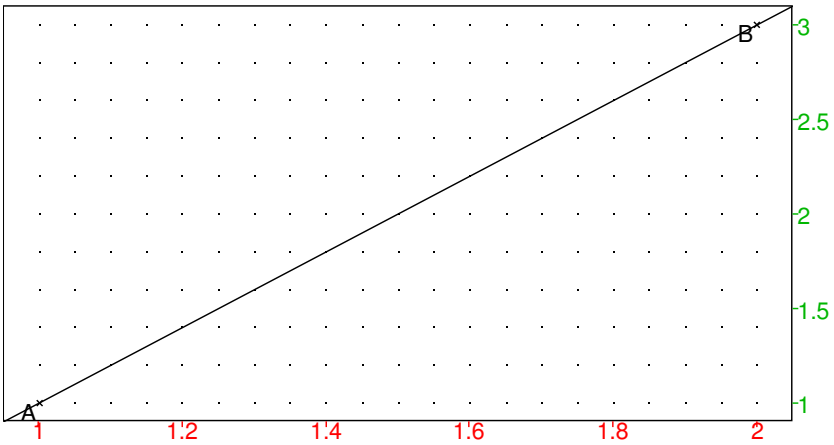
```
coordonnees(B)
```

$$[2,3]$$

5.3 La droite et son équation : droite et equation

`droite` a comme argument 2 points.
`droite` trace la droite passant par ces 2 points sur l'écran de géométrie $2d$.

```
A:=A;B:=B;d:=droite(A,B);
```



`equation` a comme argument une droite.
`equation` renvoie l'équation de cette droite

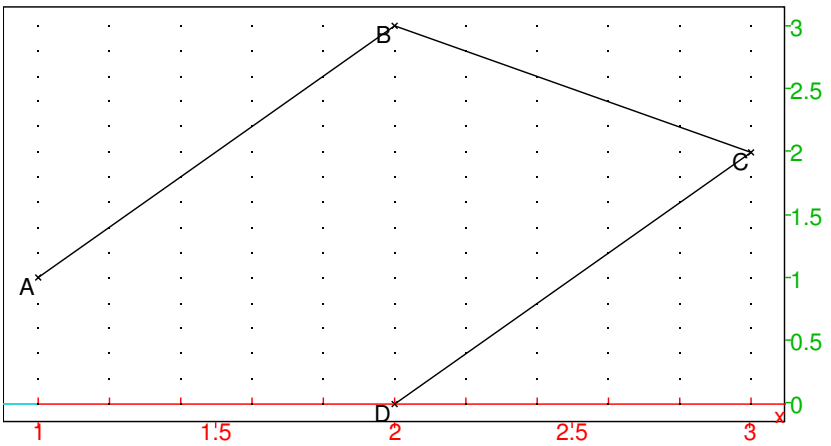
```
equation(d)
```

$$y = (2 \cdot x - 1)$$

5.4 Ligne brisée : polygone_ouvert

`polygone_ouvert` a comme argument une liste `L` de points.
`polygone_ouvert` trace la ligne brisée joignant les points `L[k]` et `L[k+1]` pour `k=0..dim(L)-2`.

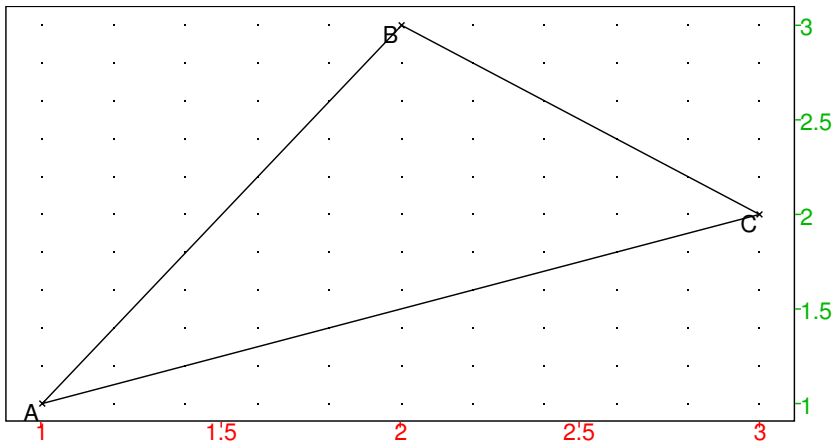
```
A:=point(1,1);B:=point(2,3);C:=point(3,2
);D:=point(2,0);polygone_ouvert(A,B,C,D)
```



5.5 Les polygones : `triangle, carre, polygone`

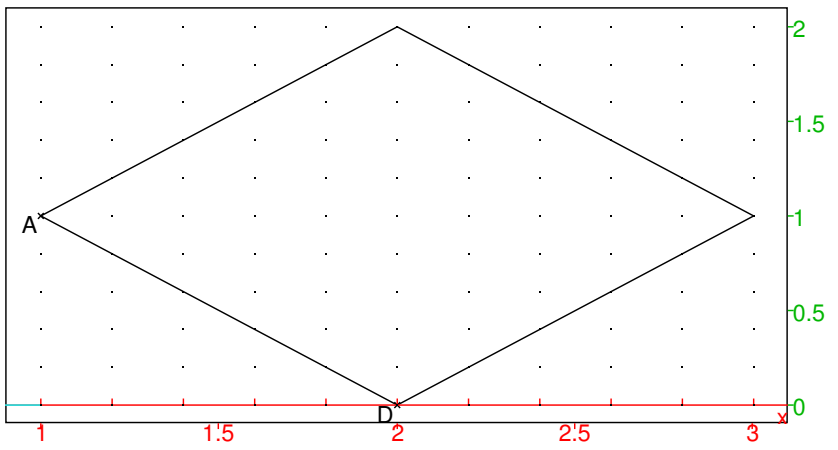
`triangle` a comme argument 3 points.
`triangle` trace le triangle défini par ces 3 points sur l'écran de géométrie $2d$.

```
A:=A;B:=B;C:=C;triangle(A,B,C)
```

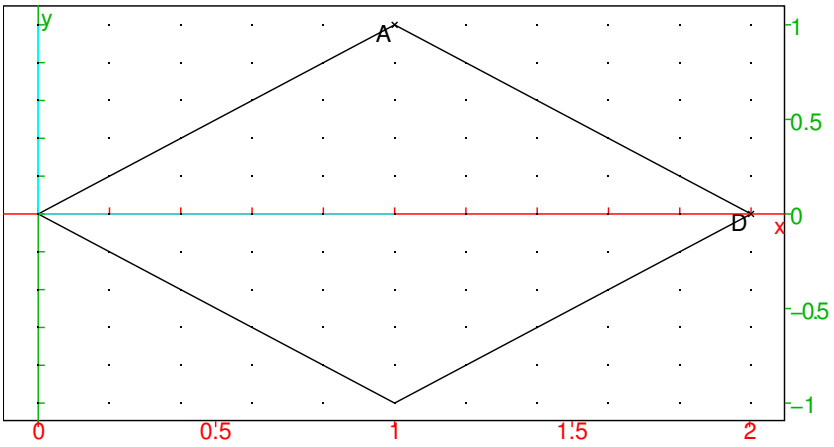


`carre` a comme argument 2 points.
`carre` trace le carré direct défini par ces 2 points sur l'écran de géométrie $2d$.

```
A:=A;D:=D;carre(A,D)
```

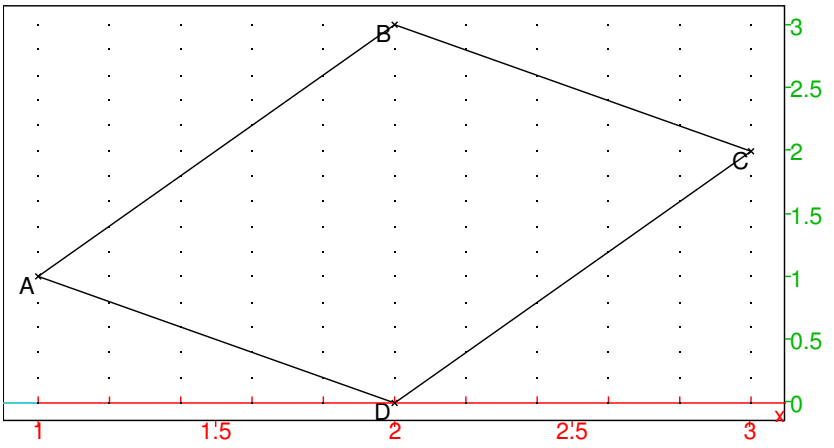



```
A:=A;D:=D;carre(D,A)
```



`polygone` a comme argument une liste de points.
`polygone` trace le polygone fermé défini par cette liste de points sur l'écran de géométrie $2d$.

```
A:=A;B:=B;C:=C;D:=D;polygone(A,B,C,D)
```



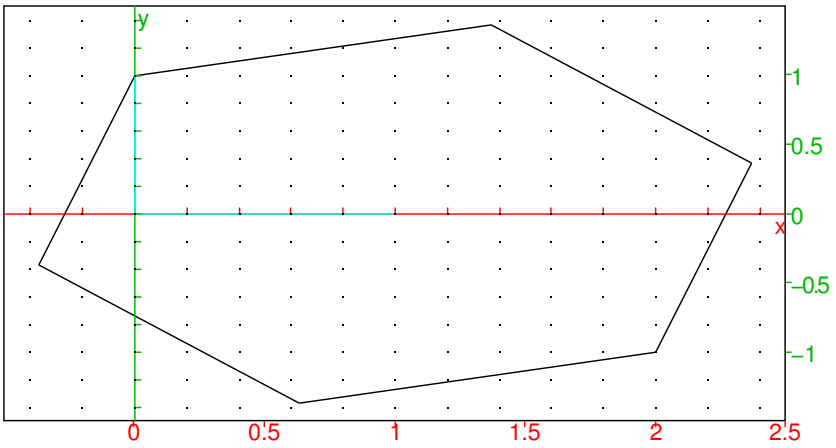
Exercice
Faire un programme qui trace les polygones réguliers connaissant son centre C, un de ses sommets A et le nombre n de côtés.

```
fonction Isopolygonec(C,A,n)
 local L,k;
 L:=NULL;
 L:=A;
 pour k de 0 jusque n-1 faire
```

```
   L:=L,C+(A-C)*exp(2*i*k*pi/n);
 fpour;
 retourne polygone(L);
ffonction:;
```

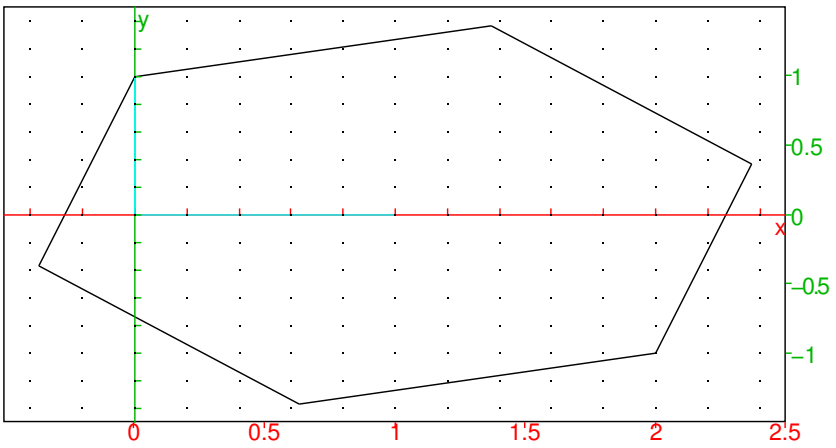
Xcas, la fonction isopolygone a 3 arguments : 2 sommets et n le nombre de côtés ou bien le centre du polygone, un sommet et un nombre négatif $-n$ qui a comme valeur absolue le nombre n de côtés.
Avec Xcas, on tape :

```
isopolygone(point(1),point(i),-6)
```



qui est identique à :

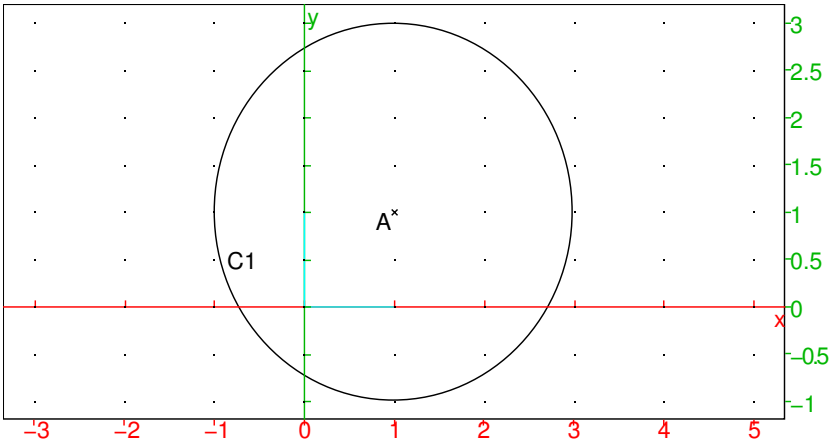
```
Isopolygonec(point(1),point(i),6)
```



5.6 Le cercle et son équation : `cercle` et `equation`

Si le cercle est défini par son centre et son rayon alors `cercle` a pour argument un point et un réel `r`.
`cercle` trace le cercle de centre ce point et de rayon `abs(r)` sur l'écran de géométrie $2d$.

```
A:=point(1,1);C1:=cercle(A,-2)
```

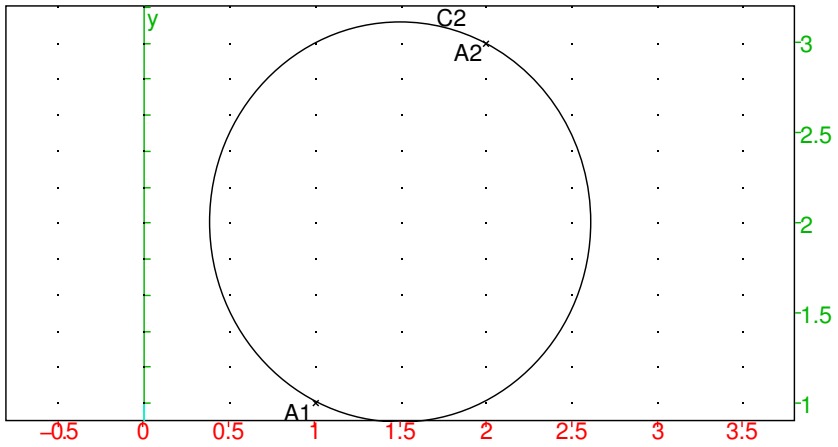



```
equation(C1)
```

$$(x-1)^2+(y-1)^2=4$$

Si le cercle est défini par son diamètre alors `cercle` a pour argument 2 points.
`cercle` trace le cercle de diamètre ces 2 points.

```
A1:=point(1,1);A2:=point(2,3);C2:=cercle(A1,A2)
```



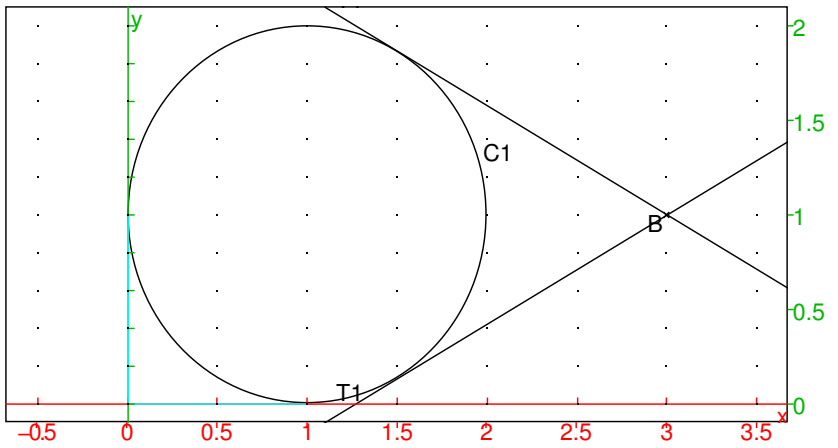

```
equation(C2)
```

$$\left(x-\frac{3}{2}\right)^2+(y-2)^2=(\frac{5}{4})$$

5.7 Les tangentes à un cercle passant par un point et leurs équations

Si `C` est un cercle et `B` un point situé à l'extérieur de (resp sur) `C` alors `tangent(C,B)` trace les (resp la) tangente(s) à `C` passant par `B`.

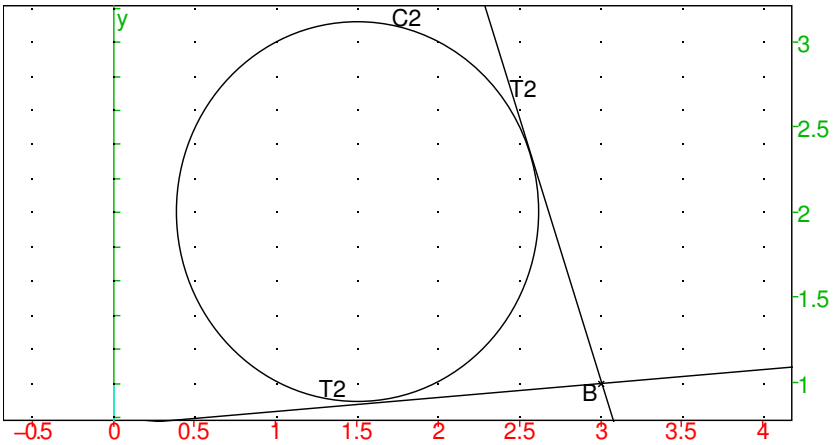
```
B:=point(3,1);C1:=cercle(point(1,1),1);T1:=tangent
(C1,B);
```



```
equation(T1)
```

$$[y = (-\sqrt{3} + 1 + \frac{\sqrt{3}}{3} \cdot x), y = (\sqrt{3} + 1 - \frac{\sqrt{3}}{3} \cdot x)]$$

```
B:=point(3,1);C2:=cercle(A1,A2);T2:=tangent(C2,B);
```




```
equation(T2);
```

$$[y = (\frac{(\sqrt{10} - 3)}{2} \cdot x + \frac{(-3 \cdot \sqrt{10} + 11)}{2}), y = (\frac{(-\sqrt{10} - 3)}{2} \cdot x + \frac{(3 \cdot \sqrt{10} + 11)}{2})]$$

5.8 Exercice : les lunules d'Hippocrate

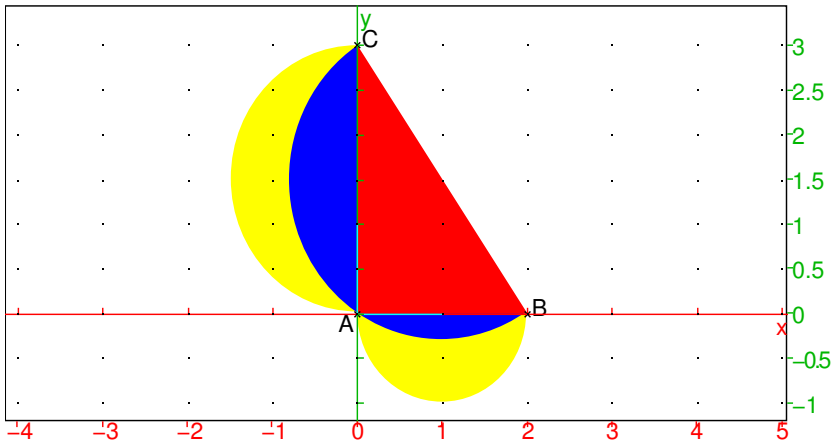
Ce théorème, très ancien, a été démontré par Hippocrate de Chios (-500) (Ne pas le confondre avec Hippocrate de Cos, le médecin), qui étudia aussi la duplication du cube, c'est-à-dire le calcul de la racine cubique de 2.
Hippocrate recherchait alors la quadrature du cercle et pensait que la quadrature de ses lunules allait le rapprocher du but.
Une lunule est une portion de surface délimitée par deux arcs de cercles non concentriques de rayons différents. Ces arcs ont mêmes extrémités et forment un croissant de lune en forme de ménisque : convexe d'un côté et concave de l'autre.

5.8.1 Exercice 1

Soit le triangle ABC rectangle en A et $\mathcal{C}$ le cercle circonscrit à ABC (de diamètre BC).
La lunule L_{AC} est la figure formée par le demi-disque de diamètre AC extérieur au triangle ABC, auquel on enlève son intersection avec le disque délimité par $\mathcal{C}$.
La lunule L_{AB} est la figure formée par le demi-disque de diamètre AB extérieur au triangle ABC, auquel on enlève son intersection avec le disque délimité par $\mathcal{C}$.
Ces deux lunules (en jaune sur la figure ci-dessous) sont appelées **lunules d'Hippocrate**.
Montrer que la somme des aires de ces deux lunules L_{BC} et de L_{BA} (en jaune) est égale à l'aire du triangle ABC (en rouge).
On tape pour faire la figure :

```
fonction Lunule1()
local A,B,C,L;
L:=NULL;
A:=point(0);
B:=point(2,affichage=quadrant1);
C:=point(3*i,affichage=quadrant1);
L:=L,cercle((A+C)/2,3/2,pi/2,3*pi/2,affichage=3+rempli);
L:=L,cercle(A,B,-pi,0,affichage=3+rempli);
L:=L,cercle(B,C,0,pi,affichage=4+rempli);
L:=L,triangle(A,B,C,affichage=1+rempli);
retourne L;
ffonction:;
```

```
Lunule1();A:=point(0);B:=point(2,affichage=quadrant1
);C:=point(3*i,affichage=quadrant1);
```



Solution

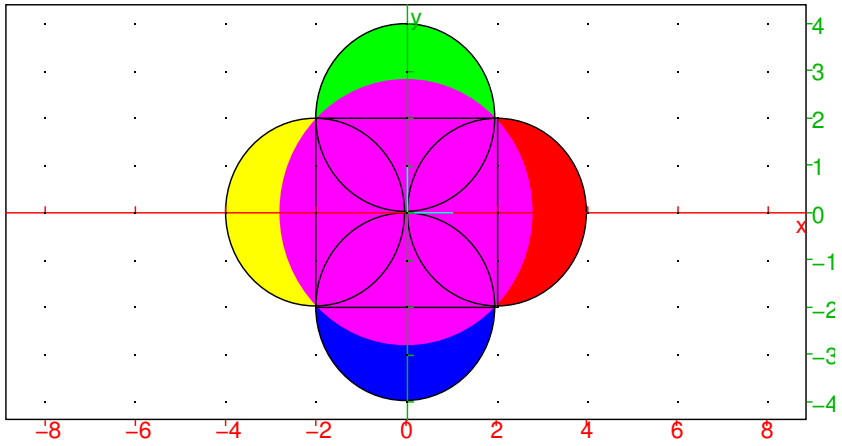
On calcule $S1$ l'aire du triangle ABC :
$S1 = AB * AC/2$ On calcule $S2$ la somme des aires de L_{BC} et de L_{BA} :
$S2$ est obtenue par différence : $S2$ est égale à la somme des aires des demi-cercles de diamètres AB et AC à laquelle on enlève l'aire en bleu.
L'aire en bleu est égale à l'aire du demi-cercle de diamètre BC à laquelle on enlève l'aire $S1$ du triangle ABC :
$S2 = \pi * AB^2/2 + \pi * AC^2/2 - (\pi * BC^2/2 - S1)$
D'après le théorème de Pythagore on a : $AB^2 + AC^2 = BC^2$ donc :
$S2 = S1$.

5.8.2 Exercice 2

Soient un carré de côtés l et les cercles ayant comme diamètre les côtés du carré.
À l'extérieur du carré, ces cercles déterminent avec le cercle circonscrit au carré 4 lunules.
Trouver l'aire des 4 lunules ainsi déterminées.
À l'intérieur du carré, ces cercles en se coupant déterminent 4 "pétales".
Trouver l'aire des 4 "pétales" ainsi déterminées.
On tape pour faire la figure :

```
fonction Lunule2()
local A,B,C,L;
L:=NULL;
L:=L,cercle(point(2),2,-pi/2,pi/2,affichage=1+rempli);
L:=L,cercle(point(2*i),2,0,pi,affichage=2+rempli);
L:=L,cercle(point(-2),2,pi/2,3*pi/2,affichage=3+rempli);
L:=L,cercle(point(0,-2),2,pi,2*pi,affichage=4+rempli);
```

```
L:=L,cercle(0,2*sqrt(2),affichage=5+rempli);
L:=L,cercle(point(0,-2),2);
L:=L,cercle(point(0,2),2);
L:=L,cercle(point(2,0),2);
L:=L,cercle(point(-2,0),2);
L:=L,carre(-2-2*i,2-2*i);
retourne L;
ffonction:;

   Lunule2();
```



Solution Un carré est formé de 2 triangles rectangles donc l'aire des 4 lunules est égale à l'aire du carré (cl le résultat précédent).
La somme de l'aire du carré et de l'aire des "pétales" est égale à l'aire des 4 demi-cercles de rayon $l/2$ (car les 4 demi-cercles qui sont à l'intérieur du carré remplissent le carré et se superposent selon les "pétales") donc l'aire des "pétales"$=\pi * l^2/2 - l^2$.

5.8.3 Exercice 3

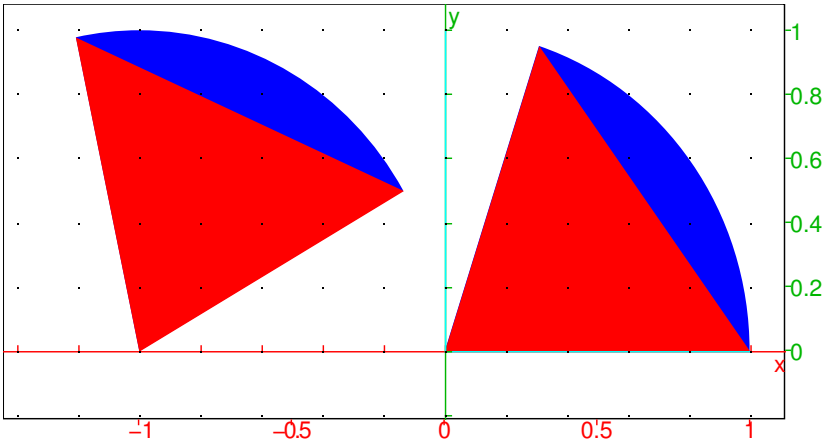
Si la tentative de la quadrature du cercle est un échec, Hippocrate a trouvé la quadrature de plusieurs lunules en partant de la remarque suivante :
deux secteurs circulaires OAB et $O_1A_1B_1$ de rayon r et r_1 semblables i.e. dont les angles au centre a ont la même valeur ont des aires proportionnelles aux carrés des longueurs de leurs cordes.
On tape pour faire la figure :

```
fonction Secteurcirc(O,r,t,a)
```

```
local A,B,L,xO,yO;
L:=NULL;
[xO,yO]:=coordonnees(O);
A:=point(xO+r*cos(t),yO+r*sin(t));
B:=point(xO+r*cos(t+a),yO+r*sin(t+a));
L:=L,cercle(O,r,t,t+a,affichage=4+rempli);
L:=L,triangle(O,A,B,affichage=1+rempli);
retourne L;
ffonction:;
```

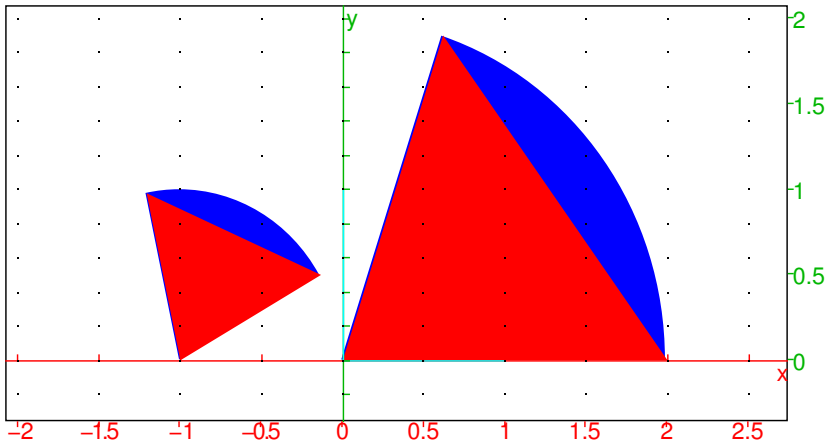
Voici 2 secteurs circulaires identiques :

```
Secteurcirc(-1,1,pi/6,2*pi/5),Secteurcirc
(0,1,0,2*pi/5)
```



Voici 2 secteurs circulaires semblables :

```
Secteurcirc(-1,1,pi/6,2*pi/5),Secteurcirc
(0,2,0,2*pi/5)
```



Chaque secteur circulaire est composé ici d'un triangle (en rouge) et d'une calotte circulaire (en bleu). Les aires des triangles ainsi que les aires des calottes circulaires,sont aussi proportionnelles aux carrés des longueurs des cordes.
Montrer ces résultats.

Solution

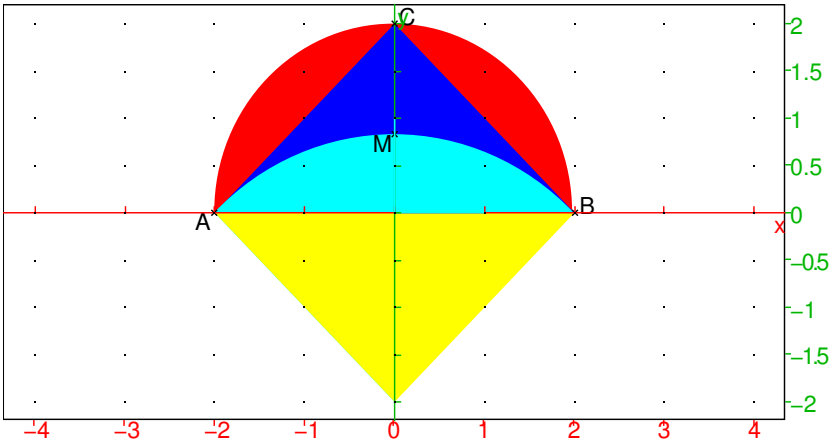
Si deux secteurs circulaires OAB et $O_1A_1B_1$ de rayon r et r_1 ont pour angle au centre a on a :
La longueur AB vaut $2R\sin(a/2$ donc $r = AB/(2\sin(a/2)$
L'aire du secteur circulaire vaut $aR^2/2 = AB^2(a/(8\sin(a/2)^2)$
L'aire du secteur circulaire est donc proportionnelle à AB^2.
L'aire du triangle OAB vaut $R^2\cos(a/2)\sin(a/2) = AB^2(1/(4\tan(a/2))$
L'aire du triangle est donc proportionnelle à AB^2.
L'aire de la calotte circulaire vaut $AB^2((a/(8\sin(a/2)^2) - (1/(4\tan(a/2))$.
L'aire du calotte circulaire est donc proportionnelle à AB^2.

Application

Soit un triangle ABC isocèle et rectangle en C.
Soit D le symétrique de C par rapport à AB.
Le demi-cercle ABC de diamètre AB et le quart de cercle AMB de centre D définissent la lunule $AMBC$ en rouge sur la figure.
On tape pour faire la figure :

```
fonction Lunule3()
local A,B,C,D,L,x0,y0;
L:=NULL;
A:=point(-2);
B:=point(2,affichage=quadrant1);
C:=point([0,2],affichage=quadrant1);
D:=point(0,-2);
```

```
L:=L,affichage(cercle(0,2,0,pi),1+rempli);
L:=L,triangle(A,B,C,affichage=4+rempli);
L:=L,affichage(cercle(D,2*sqrt(2),pi/4,3*pi/4),6+rempli);
L:=L,triangle(A,B,D,affichage=3+rempli);
retourne L;
ffonction:;

Lunule3();A:=point(-2);B:=point(2,affichage=quadrant1
);C:=point(0,2,affichage=quadrant1);M:=point
(0,-2+2*sqrt(2))
```



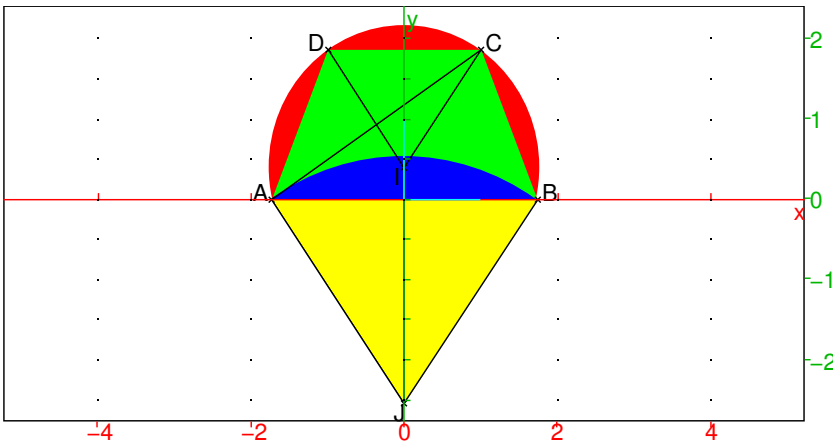
Solution L'aire des 2 calottes circulaires rouges est égale à l'aire de la calotte circulaire cyan puisque $AB^2 = AC^2 + BC^2 = 2AC^2$ (car le triangle ABC est rectangle isocèle de sommet C.
L'aire du triangle ABC (aire bleue+aire de la calotte circulaire cyan) est donc égale à l'aire de la lunule $AMBC$ (aires des calottes circulaires rouges +aire bleue)

5.8.4 Exercice 4

Soit un trapèze isocèle $ABCD$ vérifiant $BC = CD = AD$ et $AB^2 = 3DC^2$.
Soit l'arc de cercle $BCMDA$ de centre I intersection des médiatrices de AC et de DC (où M est un point de l'arc CD).
Soit l'arc de cercle BNA tangent à AC de centre J intersection de la médiatrice de DC et de la perpendiculaire en A à AC.
On tape pour faire la figure :

```
fonction Lunule4()
```

```
local A,B,C,D,I,J,L;
L:=NULL;
A:=point(-sqrt(3.));
B:=point(sqrt(3.));
C:=point(1,sqrt(2*sqrt(3.)));
D:=point(-1,sqrt(2*sqrt(3.)));
//I:=inter_unique(droite(x=0),mediatrice(A,D));
I:=point(0,sqrt(-3+2*sqrt(3.))*sqrt(3.))/3);
//J:=inter_unique(droite(x=0),perpendiculaire(A,droite(A,C)));
J:=point(0,-sqrt(3+2*sqrt(3.)));
L:=L,affichage(cercle(I,evalf(longueur(A,I)),angle(I,I+2,B),2*pi+angle(I,I+2,
L:=L,triangle(A,B,C,affichage=2+rempli);
L:=L,triangle(A,C,D,affichage=2+rempli);
L:=L,affichage(cercle(J,sqrt(6+2*sqrt(3.)),angle(J,J+2,B),angle(J,J+2,A)),4+r
L:=L,affichage(triangle(J,A,B),3+rempli);
L:=L,segment(D,I),segment(C,I),segment(A,C),segment(A,J),segment(J,B),segment
retourne L;
ffonction:;

Lunule4();A:=point(-sqrt(3.),affichage=quadrant2
);B:=point(sqrt(3.),affichage=quadrant1);C:=point
(1,sqrt(2*sqrt(3.)),affichage=quadrant1);D:=point
(-1,sqrt(2*sqrt(3.)),affichage=quadrant2
);I:=point(0,sqrt(-3+2*sqrt(3.))/sqrt(3.
));J:=point(0,-sqrt(3+2*sqrt(3.)))
```



Solution L'aire de la lunule (rouge + vert) $ABCD$ est égale à l'aire du trapèze $ABCD$ (vert+bleu).
En effet, l'aire de la calotte bleue est égale à 3 fois l'aire d'une calotte rouge car

les secteurs circulaires JBA et ICD sont homothétiques et que $AB^2 = 3DC^2$.
On a donc puisque $AD = DC = CB$, l'aire des calottes rouges sont égales.
Donc l'aire des 3 calottes rouges est égale à l'aire bleue et :
aire lunule($ABCD$)=aire trapeze($ABCD$)-aire calotte bleue+3*aire lunule(CD)
Donc
aire lunule($ABCD$)=aire trapeze($ABCD$).

Chapitre 6

La géométrie analytique

Dans ce chapitre, les programmes que l'on va faire ne feront pas de tracés mais renverront des valeurs (coordonnées, coefficients, équations).
On pourra alors faire les figures avec `Xcas` et vérifier les résultats obtenus par ces programmes.

6.1 Les segments

6.1.1 Calculer la distance de deux points connaissant leurs coordonnées

Si les points `A` et `B` ont pour coordonnées `cA:=[xA,yA]` et `cB:=[xB,yB]` le segment `AB` a pour longueur :
`sqrt((xA-xB)^2+(yA-yB)^2)`
On tape :

```
fonction Longueur(cA,cB)
 local xA,xB,yA,yB;
 xA:=cA[0];
 yA:=cA[1];
 xB:=cB[0];
 yB:=cB[1];
 retourne simplify(sqrt((xA-xB)^2+(yA-yB)^2));
ffonction:;
```

```
   cA:=[1,2]
```

$$[1, 2]$$

```
cB:=[-1,-3]
```

$$[-1,-3]$$

```
Longueur(cA,cB)
```

$$\sqrt{29}$$

Vérifions avec `Xcas` :

```
longueur(cA,cB)
```

$$\sqrt{29}$$

6.1.2 Calculer les coordonnées du milieu d'un segment

Si les points `A` et `B` ont pour coordonnées `cA:=[xA,yA]` et `cB:=[xB,yB]`, le milieu de `AB` a pour coordonnées `[(xA+xB)/2,(yA+yB)/2]` :
On tape :

```
fonction Milieu(cA,cB)
 local xA,xB,yA,yB;
 xA:=cA[0];
 yA:=cA[1];
 xB:=cB[0];
 yB:=cB[1];
 retourne [(xA+xB)/2,(yA+yB)/2];
ffonction:;
```

```
cA:=[1,2]
```

$$[1,2]$$

```
cB:=[-1,-3]
```

$$[-1,-3]$$

```
Milieu(cA,cB)
```

$$[0, \frac{-1}{2}]$$

Vérifions avec `Xcas` :

```
coordonnees(milieu(cA,cB))
```

$$[0, \frac{-1}{2}]$$

6.2 Les droites

6.2.1 Équation d'une droite définie par 2 points ou par sa pente et un point

Équation d'une droite définie par 2 points

Si les points `A` et `B` ont pour coordonnées `cA:=[xA,yA]` et `cB:=[xB,yB]`, la droite `d` passant par `A` et `B` a pour équation :
`(xA-xB)*y-(yA-yB)*x-yB*xA+yA*xB=0` ou encore
si `(xA==xB)` alors l'équation de `d` est `x=xA`
si `(xA!=xB)` alors l'équation de `d` est `y=(yA-yB)*(x-xB)/(xA-xB)+yB`

Équation d'une droite définie par sa pente et un point

La droite passant par le point `A` de coordonnées `cA:=[xA,yA]` et de pente `m` a pour équation :
`y=m*(x-xA)+yA`.
On tape :

```
fonction Droite1(cA,cB)
 local xA,xB,yA,yB;
 xA:=cA[0];
 yA:=cA[1];
 xB:=cB[0];
 yB:=cB[1];
 retourne normal((xA-xB)*y-(yA-yB)*x-yB*xA+yA*xB)=0;
ffonction:;
fonction Droite2(cA,m)
 local xA,yA;
```

```
 xA:=cA[0];
 yA:=cA[1];
 retourne y-m*(x-xA)-yA=0;
ffonction:;
```

On peut réunir les 2 programmes en un seul en testant la dimension du deuxième paramètre de `Droite` qui est soit une liste de dimension 2, soit un réél.
On peut aussi faire en sorte que `Droite` accepte 1 seul argument qui soit son équation : pour cela, on donne au deuxième argument une valeur par défaut (ici cette valeur sera arbitraire par exemple 0) (cf 2.2.4).
On tape :

```
fonction Droite(cA,L=0)
 local xA,xB,yA,yB,m;
 si type(cA)==expression alors retourne cA;fsi;
 xA:=cA[0];
 yA:=cA[1];
 si type(L)==vecteur alors
   xB:=L[0];
   yB:=L[1];
   retourne normal((xA-xB)*y-(yA-yB)*x-yB*xA+yA*xB)=0;
 sinon
   m:=L;
   retourne y-m*(x-xA)-yA=0;
 fsi;
ffonction:;
```

Observez qu'on a donné une valeur par défaut 0 au deuxième paramètre `L`, si `Droite` est appelée avec deux arguments tout se passe comme si on avait écrit `L` et non `L=0`, par contre si `Droite` est appelée avec un seul argument, alors `L` prend la valeur 0 au début de la fonction.

```
purge(x,y)
```

Nosuchvariablex, Nosuchvariabley

```
cA:=[1,2]
```

$[1, 2]$

```
cB:=[-1,-3]
```

$$[-1,-3]$$

```
m:=2
```

$$2$$

```
Droite(cA,cB)
```

$$-5\cdot x+2\cdot y+1=0$$

```
Droite(cA,m)
```

$$y-2\cdot(x-1)-2=0$$

```
Droite(x+y=1)
```

$$x+y=1$$

Vérifions avec `Xcas` :

```
equation(droite(point(cA),point(cB)))
```

$$y=(\frac{5}{2}\cdot x-\frac{1}{2})$$

```
equation(droite(point(cA),pente=m))
```

$$y=(2\cdot x)$$

6.2.2 Coefficients (a,b,c) de la droite d'équation ax+by+c=0

Étant donnée l'équation d'une droite `a*x+b*y+c=0`, on va écrire une fonction qui renvoie les coefficients `a`, `b` et `c`.
On utilise tout d'abord `gauche` et `droit` qui renvoie le côté gauche et le côté droit d'une équation.
Par exemple si `Eq:=eq1=eq2` alors `gauche(Eq)` renvoie `eq1` et `droit(Eq)` renvoie `eq2` donc
`gauche(Eq)-droit(Eq)` renvoie `eq` qui est égal à `eq1-eq2`.
On peut alors trouver `a`, `b` et `c` en donnant des valeurs à `x` et `y`.
Posons :

```
c:=subst(eq,[x,y],[0,0])
d1:=subst(eq,[x,y],[1,0])
d2:=subst(eq,[x,y],[0,1])
```

Alors on a `a:=d1-c` et `b:=d2-c`
On tape :

```
fonction Coeffsdroite(Eq)
 local a,b,c,d1,d2,eq;
 eq:=gauche(Eq)-droit(Eq);
 c:=subst(eq,[x,y],[0,0]);
 d1:=subst(eq,[x,y],[1,0]);
 d2:=subst(eq,[x,y],[0,1]);
 retourne normal(d1-c,d2-c,c);
ffonction:;
```

```
Coeffsdroite(y-3*(x-1)-1=0)
```

$$-3, 1, 2$$

```
Coeffsdroite((1+sqrt(2))^2*x-y+3=0)
```

$$2 \cdot \sqrt{2} + 3, -1, 3$$

Remarque
On peut aussi utiliser :
`coeff(P(x,y),x)` (resp `coeff(P(x,y),y)`) qui renvoie la liste des coefficients selon les puissances décroissantes du polynôme `P` par rapport à la variable `x` (resp `y`) et
`coeff(P(x,y),x,n)` (resp `coeff(P(x,y),y,n)`) qui renvoie le coefficient de

`x^n` (resp de `y^n`) du polynôme `P`.
On tape :

```
fonction Coeffdroite(Eq)
 local a,b,c;
 Eq:=gauche(Eq)-droit(Eq);
 a:=coeff(Eq,x,1);
 b:=coeff(Eq,y,1);
 c:=subst(Eq,[x,y]=[0,0]);
 retourne normal(a,b,c);
ffonction:;
```

```
Coeffdroite(y-3*(x-1)-1=0)
```

$$-3, 1, 2$$

```
Coeffdroite(Droite(y-3*(x-1)-1=0))
```

$$-3, 1, 2$$

Vérifions avec `Xcas` :

```
Eq:=y-3*(x-1)-1;[coeff(Eq,x,1),coeff(Eq,y,1
),subst(Eq,[x,y]=[0,0])]
```

$$y - 3 \cdot (x - 1) - 1, [-3, 1, 2]$$

6.2.3 Point d'intersection de 2 droites sécantes

Cette section reprend la section sur la résolution de système de 2 équations à 2 inconnues. Soient deux droites `d1` et `d2` d'équation :
$a_1x + b_1y + c_1 = 0$ et $a_2x + b_2y + c_2 = 0$
Ces 2 droites sont parallèles si $a_1b_2 = a_2b_1$.
Si $a_1b_2 \neq a_2b_1$ `d1` et `d2` sont sécantes.
Les coordonnées de leur point d'intersection sont
$(-c_2 * b_1 + b_2 * c_1)/(-b_2 * a_1 + a_2 * b_1), (c_2 * a_1 - a_2 * c_1)/(-b_2 * a_1 + a_2 * b_1)$
`Interdroite(d1,d2)` renvoie `[]` si `d1` et `d2` sont paralléles et sinon renvoie les coordonées de leur point d'intersection.
On utilise les programmes `Droite` et `Coeffsdroite` précédents (cf 6.2.2 pour avoir les coefficients des équations `Eq1` et `Eq2` de `d1` et de `d2` On tape :

```
fonction Interdroite(d1,d2)
 local a1,a2,b1,b2,c1,c2,gd1,gd2,d;
 (a1,b1,c1):=Coeffsdroite(d1);
 (a2,b2,c2):=Coeffsdroite(d2);
 d:=normal(a2*b1-b2*a1);
 si d==0 alors retourne [];fsi;
 retourne [(b2*c1-b1*c2)/d,(c2*a1-a2*c1)/d];
ffonction:;
```

```
d1:=Droite(2x+3y-1=0)
```

$$2 \cdot x + 3 \cdot y - 1 = 0$$

```
d2:=Droite(-x+y+1=0)
```

$$-x + y + 1 = 0$$

```
d3:=Droite([1,1],[2,3])
```

$$2 \cdot x - y - 1 = 0$$

```
d4:=Droite([1,0],[0,1])
```

$$x + y - 1 = 0$$

```
d5:=Droite([1,2],2)
```

$$y - 2 \cdot (x - 1) - 2 = 0$$

```
Interdroite(d1,d2)
```

$$[\frac{4}{5}, \frac{-1}{5}]$$

```
Interdroite(d3,d4)
```

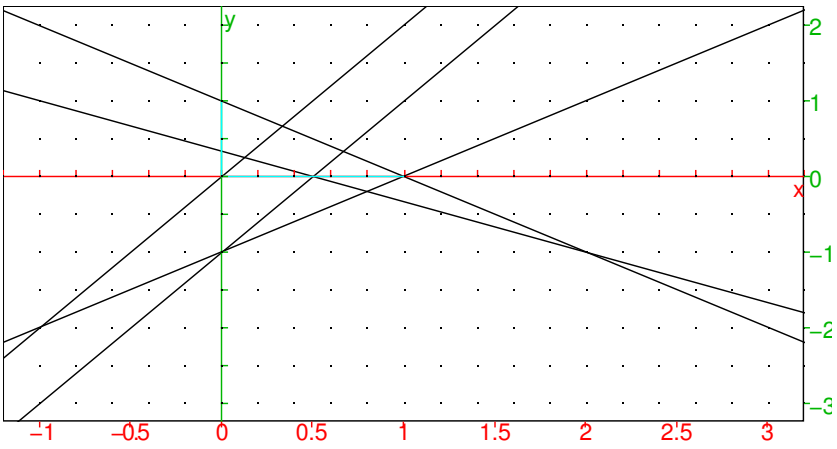
$$[\frac{2}{3},\frac{1}{3}]$$

```
Interdroite(d5,d3)
```

$$[]$$

On fait la figure avec Xcas :

```
droite(d1),droite(d2),droite(d3),droite(d4
),droite(d5)
```



Vérifions avec Xcas :

```
coordonnees(inter_unique(droite(d1),droite(d2)))
```

$$[\frac{4}{5},\frac{-1}{5}]$$

```
coordonnees(inter_unique(droite(d3),droite(d4)))
```

$$[\frac{2}{3},\frac{1}{3}]$$

6.3 Triangles et quadrilatères définis par les coordonnées des sommets

On définit des versions de la commande `polygone` de `Xcas` donc ces 2 programmes vont faire des figures.

```
fonction Triangle(cA,cB,cC)
 retourne polygone(cA,cB,cC);
ffonction:;
fonction Quadrilatere(cA,cB,cC,cD)
 retourne polygone(cA,cB,cC,cD);
ffonction:;
```

```
cA:=[1,-1]
```

$$[1,-1]$$

```
cB:=[1/2,2]
```

$$[\frac{1}{2},2]$$

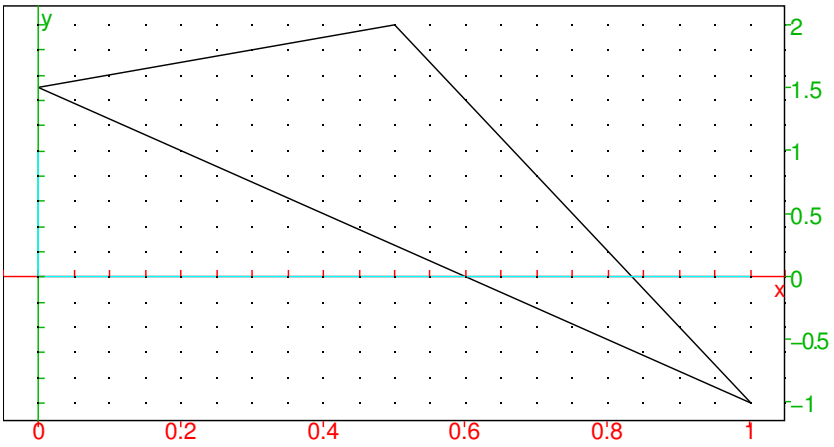
```
cC:=[0,3/2]
```

$$[0,\frac{3}{2}]$$

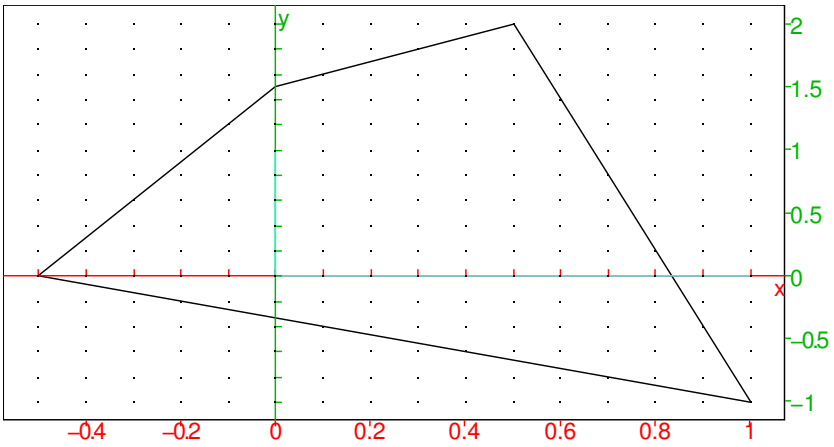
```
cD:=[-1/2,0]
```

$$[\frac{-1}{2},0]$$

```
Triangle(cA,cB,cC)
```




```
Quadrilatere(cA,cB,cC,cD)
```



6.4 Les vecteurs

6.4.1 Les coordonnées d'un vecteur défini par 2 points

Si les coordonnées du point A (resp B) sont `cA:=[xA,yA]` (resp `cB:=[xB,yB]`), les coordonnées du vecteur `AB` sont `[xB-xA,yB-yA]`.
On tape :

```
fonction Vecteur(cA,cB)
 local xA,xB,yA,yB;
 xA:=cA[0];
 yA:=cA[1];
 xB:=cB[0];
```

```
 yB:=cB[1];
 retourne normal([xB-xA,yB-yA]);
ffonction:;
```

ou plus simplement :

```
Vecteur(cA,cB):=normal(cB-cA);
```

```
 (cA,cB)->normal(cB-cA)
```

```
cA:=[1,2]
```

$$[1, 2]$$

```
cB:=[-1,-3]
```

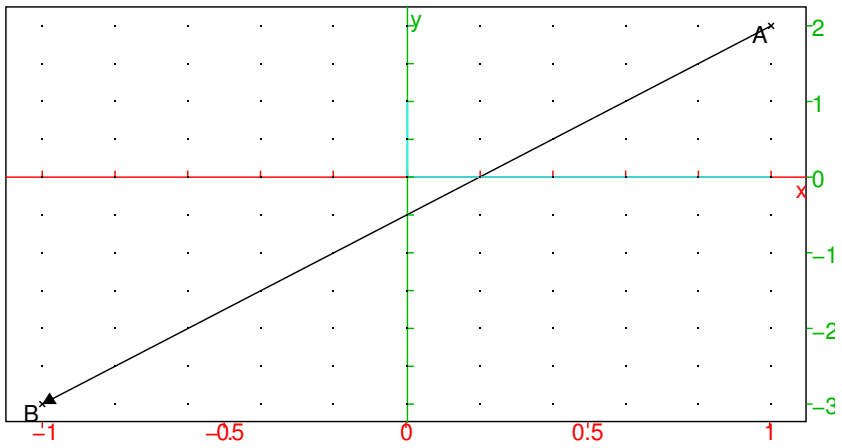
$$[-1, -3]$$

```
Vecteur(cA,cB)
```

$$[-2, -5]$$

On fait la figure avec Xcas :

```
A:=point(cA);B:=point(cB);vecteur(A,B);
```



Vérifions avec Xcas :

```
coordonnees(vecteur(A,B))
```

$$[-2,-5]$$

6.4.2 Calculer les coordonnées de la somme de deux vecteurs dans un repère

Si les vecteurs V1 et V2 ont pour coordonnées cV1:=[x1,y1] et cV2:=[x2,y2], les coordonnées du vecteur V1+V2 sont [x1+x2,y1+y2].
On tape :

```
fonction SumVect(cV1,cV2)
 local x1,x2,y1,y2;
 x1:=cV1[0];
 y1:=cV1[1];
 x2:=cV2[0];
 y2:=cV2[1];
 retourne normal([x1+x2,y1+y2]);
ffonction:;
```

ou plus simplement :

```
SumVect(cV1,cV2):=normal(cV1+cV2);
```

```
 (cV1,cV2)->normal(cV1+cV2)
```

```
cV1:=[1,2]
```

$$[1,2]$$

```
cV2:=[-2,3]
```

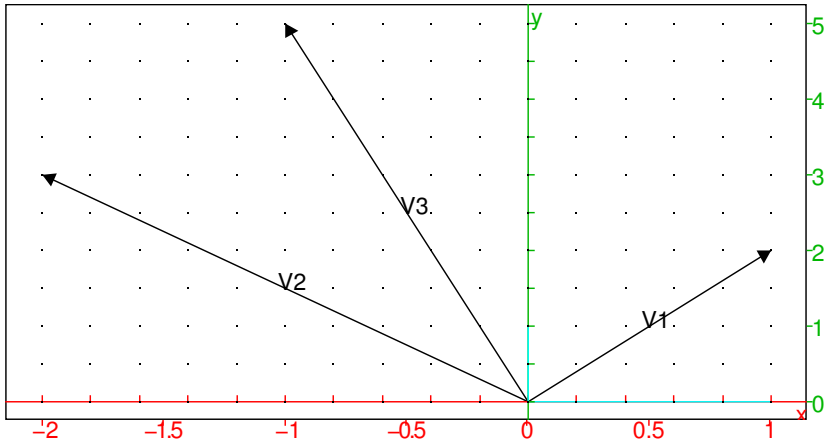
$$[-2,3]$$

```
cV3:=SumVect(cV1,cV2)
```

$$[-1, 5]$$

On fait la figure avec `Xcas` :

```
V1:=vecteur(0,cV1);V2:=vecteur(0,cV2);V3:=vecteur
(0,cV3)
```



Vérifions avec `Xcas` :

```
cV1+cV2
```

$$[-1, 5]$$

6.4.3 Coordonnées de D extrémité du vecteur d'origine C équipollent au vecteur AB

On a $D := C + (B - A)$.
On tape :

```
Translation(cA,cB,cC):=cC+(cB-cA);
```

```
(cA,cB,cC)->cC+cB-cA
```

```
cA:=[1,2]
```

$$[1, 2]$$

```
cB:=[2,3]
```

$$[2, 3]$$

```
cC:=[-1,1]
```

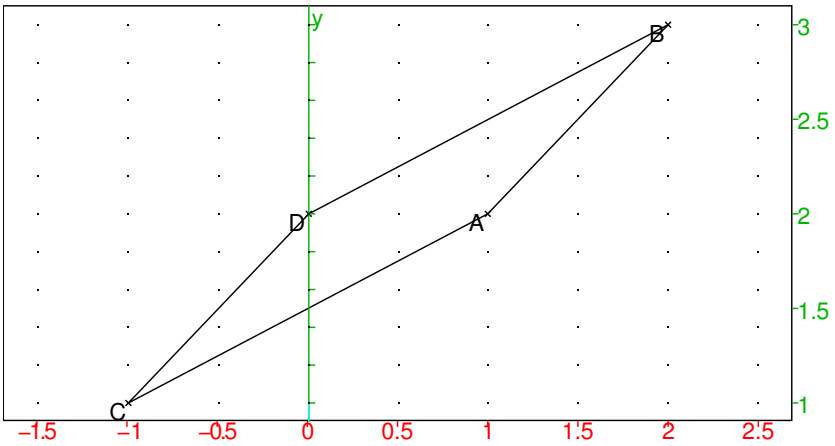
$$[-1, 1]$$

```
cD:=Translation(cA,cB,cC)
```

$$[0, 2]$$

On fait la figure avec `Xcas` :

```
A:=point(cA);B:=point(cB);C:=point(cC);D:=point
(cD);polygone(cA,cB,cD,cC)
```



Vérifions avec `Xcas` :

```
coordonnees(translation(cB-cA,point(cC)))
```

$$[0,2]$$

6.4.4 Norme d'un vecteur

Soit le vecteur `V:=[xV,yV]`, on pose `cV:=[xV,yV]` la liste des coordonnées de V.
La norme de `V` est égale à `sqrt(xV^2+yV^2)`.
On tape :

```
fonction Norme(cV)
 local xV,yV;
 xV:=cV[0];
 yV:=cV[1];
 retourne sqrt(xV^2+yV^2);
ffonction:;
```

```
cA:=[1,2]
```

$$[1,2]$$

```
cB:=[2,3]
```

$$[2,3]$$

```
Norme(cB-cA)
```

$$\sqrt{2}$$

Vérifions avec `Xcas` :

```
norm(vecteur(point(cA),point(cB)))
```

$$\sqrt{2}$$

6.5 Changement de repères

6.5.1 Le problème

Soient 2 repères orthonormés O, Ox, Oy et I, IX, IY.

Notations :

- Soit M un point ou un vecteur.
  On note c_M les coordonnées $[x_M, y_M]$ de M dans le repère Oxy.
  On note C_M les coordonnées $[X_M, Y_M]$ de M dans le repère IXY.
- On note u (resp v) le vecteur unitaire porté par Ox (resp Oy)
- On note U (resp V) le vecteur unitaire porté par OX (resp OY)

Avec ces notations, on a :

- le point I a pour coordonnées $c_I := [x_I, y_I]$ dans le repère O, Ox, Oy donc $OI = x_I u + y_I v$
- le vecteur U a pour coordonnées $c_U := [x_U, yU]$ dans le repère O, Ox, Oy $(x_U^2 + y_U^2 = 1)$ donc $U = x_U u + y_U v$
- le vecteur V a pour coordonnées $c_V := [x_V, y_V]$ dans le repère O, Ox, Oy $(x_V^2 + y_V^2 = 1)$ donc $V = x_V u + y_V v$. L'angle $(U, V) = \pi/2$ donc $x_V := -y_U$ et $y_V := x_U$

6.5.2 Le programme `Changexy2XY(cM,cI,cU)`

On connaît les coordonnées $cM := [x_M, y_M]$ d'un point M dans le repère (O, Ox, Oy) ainsi que les coordonnées $c_I := [x_I, y_I]$ et $c_U := [x_U, y_U]$ de I et de U dans le repère (O, Ox, Oy).
On cherche les coordonnées $C_M := [X_M, Y_M]$ de M dans le repère (I, IX, IY).
On a donc :

$$\begin{aligned} OM = x_M u + y_M v &= OI + IM \\ &= (x_I u + y_I v) + (X_M U + Y_M V) \\ &= x_I u + y_I v + X_M(x_U u + y_U v) + Y_M(x_V u + y_V v) \\ &= (x_I + X_M x_U + Y_M x_V)u + (y_I + X_M y_U + Y_M y_V)v \end{aligned}$$

Donc :

$$x_M = x_I + X_M x_U + Y_M x_V, \quad y_M = y_I + X_M y_U + Y_M y_V$$

on en déduit X_M, Y_M

$$X_M = \frac{(x_M - x_I)y_V - (y_M - y_I)x_V}{x_U y_V - y_U x_V}, \quad Y_M = \frac{(x_M - x_I)y_U - (y_M - y_I)x_U}{x_V y_U - y_V x_U}$$

Or $x_V = -y_U$ et $y_V = x_U$ donc

$$x_U y_V - y_U x_V = x_U^2 + y_U^2 = 1$$

Finalement :

$$X_M = (x_M - x_I)x_U + (y_M - y_I)y_U, \quad Y_M = (x_M - x_I)y_U - (y_M - y_I)x_U$$

On tape :

```
fonction Changexy2XY(cM,cI,cU)
local xM,xI,xU,xV,yM,yI,yU,yV,l;
xM:=cM[0];
yM:=cM[1];
xI:=cI[0];
yI:=cI[1];
xU:=cU[0];
yU:=cU[1];
l:=xU^2+yU^2;
si l!=1 alors l:=sqrt(l);xU:=xU/l;yU:=yU/l;fsi;
xV:=-yU;
yV:=xU;
retourne normal([((xM-xI)*xU+(yM-yI)*yU),(-(xM-xI)*yU+(yM-yI)*xU)]);
ffonction:;
```

Remarque
Dans le programme ci-dessus, on teste si le vecteur `U` est unitaire, si ce n'est pas le cas, on le rend unitaire avec :
`l:=xU^2+yU^2;si l!=1 alors l:=sqrt(l);xU:=xU/l;yU:=yU/l;fsi;`
On tape :

```
cM:=[5/2,9/2]
```

$$[\frac{5}{2}, \frac{9}{2}]$$

```
cI:=[2,3]
```

$$[2, 3]$$

```
cU:=[sqrt(2)/2,sqrt(2)/2]
```

$$[\frac{\sqrt{2}}{2},\frac{\sqrt{2}}{2}]$$

```
cV:=[-sqrt(2)/2,sqrt(2)/2]
```

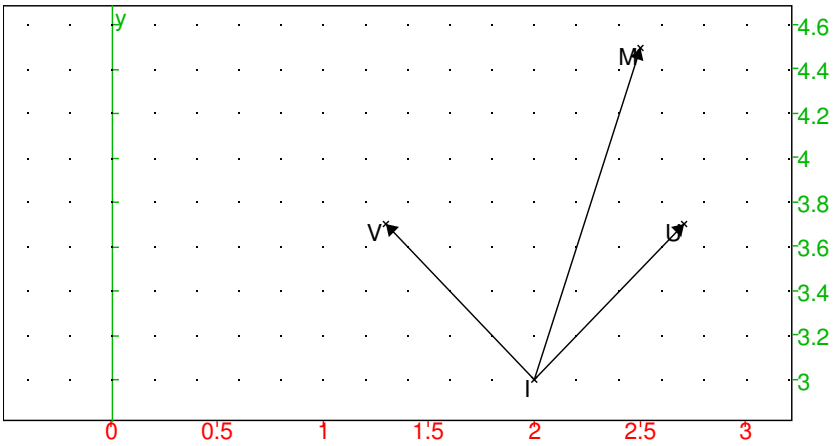
$$[-\frac{\sqrt{2}}{2},\frac{\sqrt{2}}{2}]$$

```
Changexy2XY(cM,cI,cU)
```

$$[\sqrt{2},\frac{\sqrt{2}}{2}]$$

La figure avec `Xcas` :

```
O:=point(0);M:=point(cM);I:=point(cI);U:=point
(cI+cU);V:=point(cI+cV);vecteur(I,U);vecteur
(I,V);vecteur(I,M)
```



6.5.3 Le programme `ChangeXY2xy(CM,cI,cU)`

Il s'agit du programme inverse du précédent : on connaît les coordonnées $C_M := [X_M, Y_M]$ d'un point M dans le repère (I, IX, IY) ainsi que les coordonnées $c_I := [x_I, y_I]$ et $c_U := [x_U, y_U]$ de I et de U dans le repère (O, Ox, Oy). On cherche les coordonnées $c_M := [x_M, y_M]$ de M dans le repère (O, Ox, Oy).

On a vu précédemment que :

$$x_M = x_I + X_M x_U + Y_M x_V, \quad y_M = y_I + X_M y_U + Y_M y_V$$

On tape :

```
fonction ChangeXY2xy(CM,cI,cU)
 local XM,xI,xU,xV,YM,yI,yU,yV,l;
 XM:=CM[0];
 YM:=CM[1];
 xI:=cI[0];
 yI:=cI[1];
 xU:=cU[0];
 yU:=cU[1];
 l:=xU^2+yU^2;
 si l!=1 alors l:=sqrt(l);xU:=xU/l;yU:=yU/l;fsi;
 xV:=-yU;
 yV:=xU;
 retourne normal([xI+XM*xU+YM*xV,yI+XM*yU+YM*yV]);
ffonction:;
```

Remarque
Dans le programme ci-dessus, si le vecteur `U` n'est pas unitaire, on le rend unitaire :
`l:=xU^2+yU^2;si l!=1 alors l:=sqrt(l);xU:=xU/l;yU:=yU/l;fsi;`
On tape :

```
CM:=[sqrt(2),sqrt(2)/2]
```

$$[\sqrt{2}, \frac{\sqrt{2}}{2}]$$

```
cI:=[2,3]
```

$$[2, 3]$$

```
cU:=[sqrt(2)/2,sqrt(2)/2]
```

$$[\frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}]$$

```
cV:=[-sqrt(2)/52,sqrt(2)/2]
```

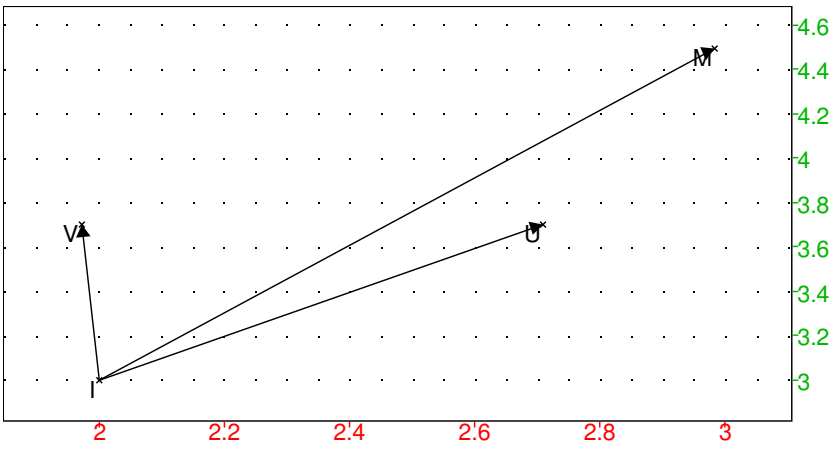
$$[-\frac{\sqrt{2}}{52},\frac{\sqrt{2}}{2}]$$

```
ChangeXY2xy(CM,cI,cU)
```

$$[\frac{5}{2},\frac{9}{2}]$$

La figure avec `Xcas` :

```
O:=point(0);M:=point(cI+CM[0]*cU+CM[1]*cV
);I:=point(cI);U:=point(cI+cU);V:=point(cI+cV
);vecteur(I,U);vecteur(I,V);vecteur(I,M)
```



6.5.4 Exercices

En se servant des programmes précédents, faire les programmes :

1. calculant les coordonnées c_C du sommet C d'un triangle équilatéral direct ABC connaissant les coordonnées c_A de A et c_B de B,
2. calculant les coordonnées c_C et c_D des sommets C et D d'un carré direct $ABCD$ connaissant les coordonnées c_A de A et c_B de B.

Solution
1/ Soit un triangle équilatéral direct ABC. Dans le repère orthonormé Oxy, A a pour coordonnées $c_A := [x_A, y_A]$ et B a pour coordonnées $c_B := [x_B, y_B]$. Cherchons les coordonnées $C_C = [X_C, Y_C]$ du point C dans le repère d'origine A et d'axe des X dirigé selon le vecteur AB. On pose :

$$l := \sqrt{(x_B - x_A)^2 + (y_B - y_A)^2}$$

On a :

$$c_U := [\frac{x_B - x_A}{l}, \frac{y_B - y_A}{l}], \quad C_C := [\frac{l}{2}, \frac{l\sqrt{3}}{2}]$$

Et les coordonnées de C dans le repère orthonormé Oxy sont :
`ChangeXY2xy(CC,cA,cU)` (fonction écrite précédemment cf 6.5.3).
On tape :

```
fonction Coordequi(cA,cB)
 local xA,yA,xB,yB,cU,l,CC;
 xA:=cA[0];
 yA:=cA[1];
 xB:=cB[0];
 yB:=cB[1];
 l:=sqrt((xB-xA)^2+(yB-yA)^2);
 cU:=[(xB-xA)/l,(yB-yA)/l];
 CC:=[l/2,l*sqrt(3)/2];
 retourne ChangeXY2xy(CC,cA,cU);
ffonction:;
```

```
cA:=[1,2]
```

$$[1, 2]$$

```
cB:=[-1,-3]
```

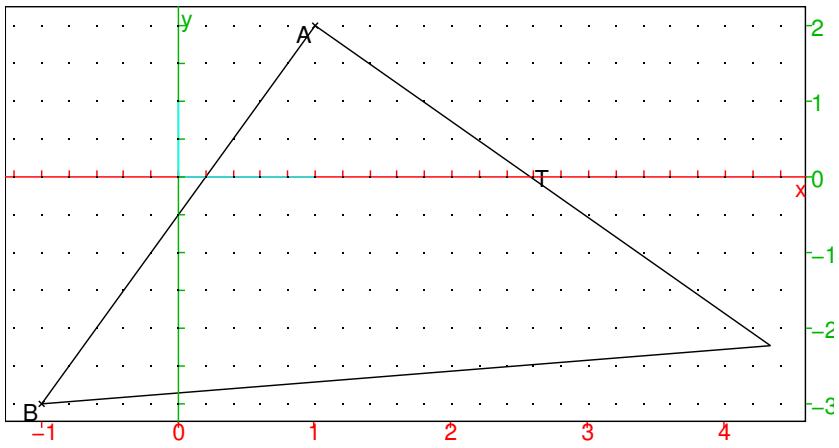
$$[-1, -3]$$

```
cC:=Coordequi(cA,cB)
```

$$[\frac{5 \cdot \sqrt{3}}{2}, \frac{(-2 \cdot \sqrt{3} - 1)}{2}]$$

On fait la figure avec Xcas :

```
A:=point(cA);B:=point(cB);T:=triangle_equilateral
(A,B);
```



Vérifions avec Xcas :

```
normal(coordonnees(sommets(T)[2]))
```

$$[\frac{5\cdot\sqrt{3}}{2},\frac{(-2\cdot\sqrt{3}-1)}{2}]$$

2/ Soit un carré direct $ABCD$, avec A de coordonnées $c_A := [x_A, y_A]$ et B de coordonnées $c_B := [x_B, y_B]$ dans le repère orthonormé Oxy.
Cherchons les coordonnées $C_C := [X_C, Y_C]$ du point C et $C_D := [X_D, Y_D]$ des points C et D dans le repère d'origine A et d'axe des X dirigé selon le vecteur AB. On pose :

$$l := \sqrt{(x_B - x_A)^2 + (y_B - y_A)^2}$$

On a :

$$c_U := [\frac{x_B - x_A}{l}, \frac{y_B - y_A}{l}], \quad c_C = [l, l], \quad c_D = [0, l]$$

Donc les coordonnées de C dans le repère orthonormé Oxy sont :
`ChangeXY2xy(CC,cA,cU)`
les coordonnées de D dans le repère orthonormé Oxy sont :
`ChangeXY2xy(CD,cA,cU)` (fonction écrite précédemment cf 6.5.3).

```
fonction Coordcarre(cA,cB)
 local xA,yA,xB,yB,CC,CD,cU,l;
 xA:=cA[0];
 yA:=cA[1];
 xB:=cB[0];
 yB:=cB[1];
 l:=sqrt((xB-xA)^2+(yB-yA)^2);
 cU:=[(xB-xA)/l,(yB-yA)/l];
 CC:=[l,l];
 CD:=[0,l];
 retourne ChangeXY2xy(CC,cA,cU),ChangeXY2xy(CD,cA,cU);
ffonction:;
```

```
cA:=[1,2]
```

$$[1, 2]$$

```
cB:=[-1,-3]
```

$$[-1, -3]$$

```
cC:=Coordcarre(cA,cB)[0]
```

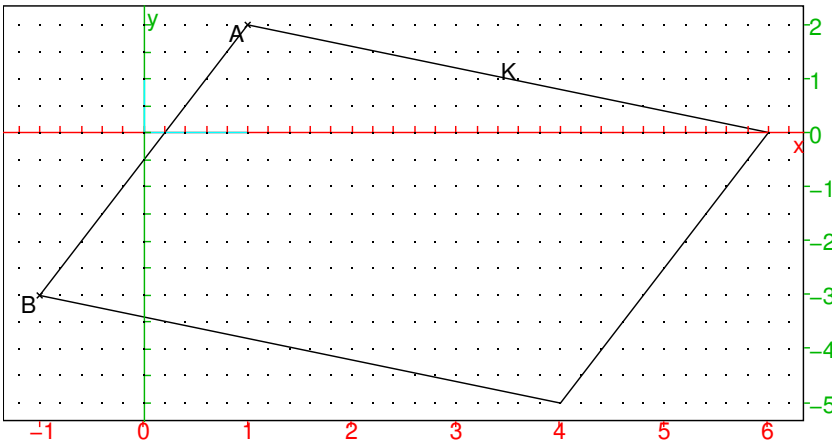
$$[4, -5]$$

```
cD:=Coordcarre(cA,cB)[1]
```

$$[6, 0]$$

On fait la figure avec Xcas :

```
A:=point(cA);B:=point(cB);K:=carre(A,B)
```



On vérifie avec `Xcas` :

```
normal(coordonnees(sommets(K)[2],sommets(K)[3]))
```

$$\begin{pmatrix} 4 & -5 \\ 6 & 0 \end{pmatrix}$$

6.6 Cercles, Tangentes à un cercle

6.6.1 Équation d'un cercle défini par son centre et son rayon

Le cercle C défini par son centre A de coordonnées $[x_A, y_A]$ et de rayon r a pour équation :

$$(x - x_A)^2 + (y - y_A)^2 = r^2$$

ou encore

$$x^2 + y^2 - 2x_A x - 2y_A y + x_A^2 + y_A^2 - r^2 = 0$$

On va écrire une procédure Cercle qui renvoie une liste constituée des coordonnées de son centre, de son rayon et de son équation.
On tape :

```
fonction Cercle1(cA,r)
 //cercle defini par son centre et son rayon
 local xA,yA;
 xA:=cA[0];
 yA:=cA[1];
 retourne [cA,r,x^2+y^2-2*xA*x-2*yA*y+xA^2+yA^2-r^2=0];
ffonction:;
```

6.6.2 Équation d'un cercle défini par son diamètre

Si les points A et B ont pour coordonnées $[x_A, y_A]$ et $[xB, yB]$, le cercle C de diamètre AB a pour équation :
si `M:= Milieu(A,B)`, si `xM:=M[0]`, si `yM:=M[1]` et si `r:=Longueur(A,B)/2` : `(x-xM)^2+(y-yM)^2=r^2` ou encore
`x^2+y^2-2*xM*x-2*yM*y+xM^2+yM^2-r^2=0`
On va écrire une procédure Cercle qui renvoie une liste constituée des coordonnées de son centre, de son rayon et de son équation.
On tape :

```
fonction Cercle2(cA,cB)
 //cercle d\'efini par son diam\`etre
 local xA,xB,xM,yA,yB,yM,M,r;
 xA:=cA[0];
 yA:=cA[1];
 xB:=cB[0];
 yB:=cB[1];
 cM:=Milieu(cA,cB):
 xM:=cM[0];
 yM:=cM[1];
 r:=Longueur(cA,cB):
 retourne [cM,r,x^2+y^2-2*xM*x-2*yM*y+xM^2+yM^2-r^2=0];
ffonction:;
```

6.6.3 Équation d'un cercle défini par son centre et son rayon ou par son diamètre

On peut reunir les 2 programmes en un seul en testant la dimension du deuxième paramètre de `Cercle` qui est soit une liste de dimension 2 (cercle défini par son diamètre), soit un réel (cercle défini par centre et rayon).
On tape :

```
fonction Cercle(cA,L)
 local cB,xA,xB,xM,yA,yB,yM,cM,r;
 xA:=cA[0];
 yA:=cA[1];
 si type(L)==vecteur alors
   xB:=L[0];
   yB:=L[1];
   cB:=[xB,yB]
   cM:=Milieu(cA,cB);
```

```
   xM:=cM[0];
   yM:=cM[1];
   r:=Longueur(cA,cB)/2;
   retourne  [cM,r,x^2+y^2-2*xM*x-2*yM*y+xM^2+yM^2-r^2=0];
 sinon
   r:=L;
   retourne [cA,r,x^2+y^2-2*xA*x-2*yA*y+xA^2+yA^2-r^2=0];
 fsi;
ffonction:;
```

```
   cA:=[1,2]
```

$$[1,2]$$

```
   cB:=[-1,-3]
```

$$[-1,-3]$$

```
   r:=2
```

$$2$$

```
   Cercle(cA,cB)
```

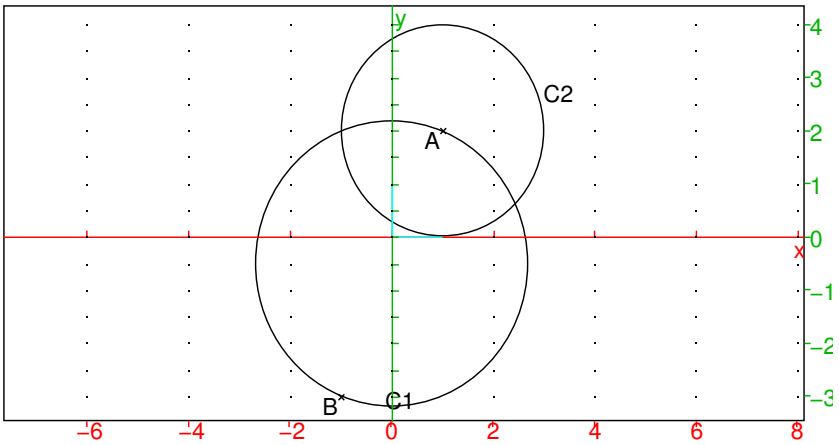
$$[[0,\frac{-1}{2}],\frac{\sqrt{29}}{2},x^2+y^2+y+\frac{1}{4}-\left(\frac{\sqrt{29}}{2}\right)^2=0]$$

```
   Cercle(cA,r)
```

$$[[1,2],2,x^2+y^2-2\cdot x-4\cdot y+1=0]$$

On fait la figure avec `Xcas` :

```
A:=point(cA);B:=point(cB);C1:=cercle(A,B
);C2:=cercle(A,r)
```



On vérifie avec Xcas :

```
normal(equation(C1));normal(equation(C2))
```

$$x^2+y^2+y+\frac{1}{4}=(\frac{29}{4}), x^2-2\cdot x+y^2-4\cdot y+5=4$$

6.6.4 Centre et rayon d'un cercle donné par son équation

On utilise ici les commandes Xcas `gauche, droit ,coeff, subst`
On tape :

```
fonction Centrerayon(Eq)
 local k,a,b,c;
 Eq:=gauche(Eq)-droit(Eq);
 k:=coeff(Eq,x,2);
 si k!=coeff(Eq,y,2) alors retourne "ce n'est pas un cercle";fsi;
 Eq:=Eq/k;
 a:=-coeff(Eq,x,1)/2;
 b:=-coeff(Eq,y,1)/2;
 c:=subst(Eq,[x,y],[0,0]);
 retourne [a,b], normal(sqrt(a^2+b^2-c));
ffonction:;
```

```
Centrerayon(2x^2+2y^2-4x+2y=0)
```

$$[1,\frac{-1}{2}],\frac{\sqrt{5}}{2}$$

On vérifie avec Xcas :

```
coordonnees(centre(2x^2+2y^2-4x+2y=0)),rayon
(2x^2+2y^2-4x+2y=0)
```

$$[1, \frac{-1}{2}], \sqrt{5}\frac{1}{2}$$

6.6.5 Construire la tangente à un cercle en l'un de ses points

Soit C un cercle de centre I de coordonnées $c_I = [x_I, y_I]$ et de rayon r. Soit A un point de C de coordonnées $c_A = [x_A, y_A]$. la tangente au cercle C en A est perpendiculaire à IA, donc a pour pente $m = -(x_A - x_I)/(y_A - y_I)$ L'équation de cette tangente est donc : `Droite(cA,m)` (fonction `Droite` a été écrite précédemment cf 6.2.1).
On utilise aussi, ici, les fonctions `Longueur` (cf 6.1.1) et `Cercle` (cf 6.6.3).
On tape :

```
fonction Tangent1(C,cA)
 local I,r,m,xI,yI,xA,yA,cI;
 cI:=C[0];
 r:=C[1];
 si Longueur(cA,cI)!=r alors retourne "A n'est pas sur C"; fsi;
 xI:=cI[0];
 yI:=cI[1];
 xA:=cA[0];
 yA:=cA[1];
 si yA!=yI alors
   m:=-(xA-xI)/(yA-yI);
   retourne Droite(cA,m);
 fsi
 retourne x=xA;
ffonction:;
```

```
   cI:=[0,1]
```

$$[0, 1]$$

```
   cA:=[1,0]
```

$$[1, 0]$$

```
r:=sqrt(2)
```

$$\sqrt{2}$$

```
C:=Cercle(cI,r)
```

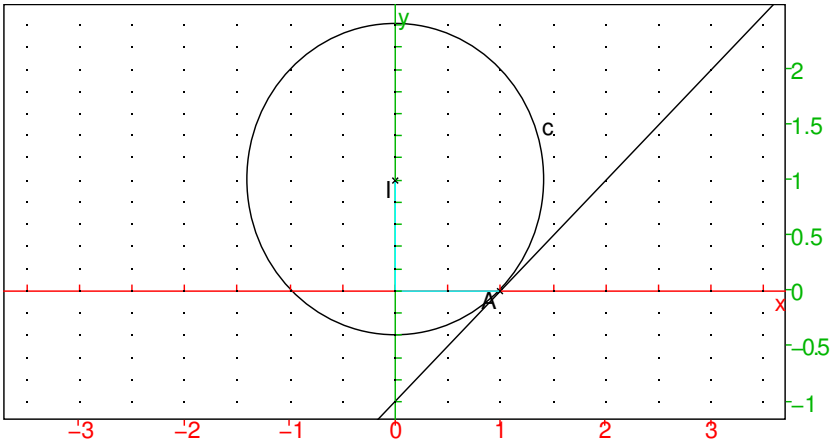
$$[[0,1],\sqrt{2},x^2+y^2-2\cdot y-1=0]$$

```
T1:=Tangent1(C,cA);
```

$$y-x+1=0$$

On fait la figure avec `Xcas` :

```
I:=point(cI);c:=cercle(I,r);A:=point(cA);droite
(T1);
```



On vérifie avec `Xcas` :

```
t:=tangent(c,A)::equation(t)
```

$$\text{Done}, y=(x-1)$$

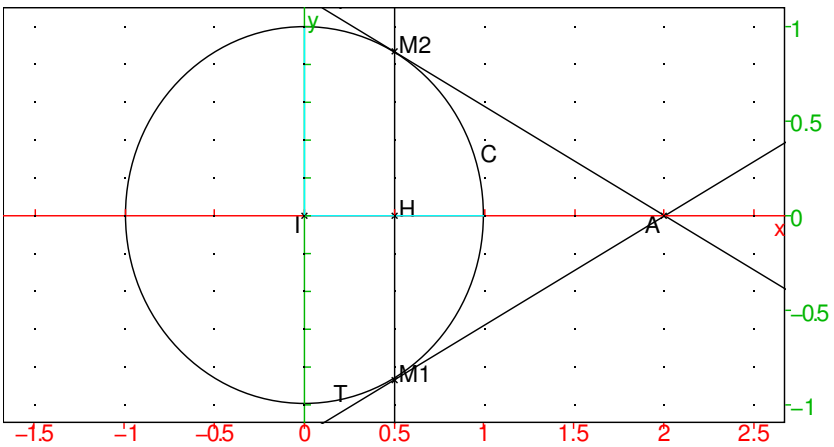
6.6.6 Construire les tangentes à un cercle passant par un point

Soit C un cercle de centre I de coordonnées $c_I := [x_I, y_I]$ et de rayon r.
Soit A un point du plan de coordonnées $c_A := [x_A, y_A]$.
Si A est à l'intérieur de C il n'y a pas de tangente à C passant par A.

Le cas simple

On suppose qu'on a choisi comme repère, le repère IXY d'origine le centre I du cercle C et tel que A est sur l'axe des X (i.e. de coordonnées $[X_A, 0]$) et à l'extérieur de C.
On peut mener par A, 2 tangentes T_1 et T_2 à C.
Soient M_1 et M_2 les 2 points de tangeance de T_1 et T_2.
Les triangles IAM_1 (resp IAM_2) sont rectangles en M_1 (resp M_2) et M_1M_2 est perpendiculaire à AI.
Soit H l'intersection de M_1M_2 avec AI.
On fait la figure avec `Xcas` :

```
I:=point(0);C:=cercle(I,1);A:=point(2,0);T:=tangent
(C,A);M1:=inter_unique(C,T[0]);M2:=inter_unique
(C,T[1]);droite(M1,M2);M12:=droite(M1,M2
):;H:=inter_unique(M12,droite(A,I))
```



On a :

$$r^2 = IM_1^2 = IM_2^2 = IA \times IH$$

Donc dans le repère IXY, A a pour abscisse $X_A =$ Longueur(cI,cA) et pour ordonnée 0.

H a pour abscisse $\frac{r^2}{X_A}$ et pour ordonnée 0.
M_1 et M_2 ont la même abscisse :

$$X_{M_1} = X_{M_2} = \frac{r^2}{X_A}$$

M_1 et M_2 ont des ordonnées opposées :

$$Y_{M_1} > 0, \quad Y_{M_1}^2 = r^2 - \frac{r^4}{X_A^2} = r^2(1 - \frac{r^2}{X_A^2}), \quad Y_{M_2} = -Y_{M_1}$$

$$Y_{M_1} = r\frac{\sqrt{X_A^2 - r^2}}{X_A}, \quad Y_{M_2} = -Y_{M_1}$$

Les tangentes sont donc :

$$\text{Droite}([X_A, 0], [X_{M_1}, Y_{M_1}]) \text{ et } \text{Droite}([X_A, 0], [X_{M_2}, Y_{M_2}])$$

Si A est sur C (i.e. $X_A == r$) alors on peut mener par A, une tangente T_1 à C qui est $\text{Droite}(x = X_A)$.

Le cas général

On se ramène au cas précédent par changement de repère.
Soit Oxy le repère. Dans le repère Oxy, I (resp A) a pour coordonnées c_I (resp c_A).
On fait un changement de repère en prenant le centre I du cercle comme origine et IA comme axe des X.
On note X_M et Y_M les coordonnées d'un point M ou d'un vecteur M dans le nouveau repère IXY et x_M et y_M les coordonnées de M dans le repère Oxy.
Dans le repère IXY, A a pour abscisse $X_A = \text{Longueur}(c_A, c_I)$ et pour ordonnée 0.
H a pour abscisse $\frac{r^2}{X_A}$ et pour ordonnée 0.
Si U est le vecteur unitaire de IX on a :
$x_U := (x_A - x_I)/X_A$ et $y_U := (y_A - y_I)/X_A$ avec $X_A := \text{Longueur}(c_A, c_I)$.
On a vu que (cf 6.5.2) :

$$x_M = x_I + X_M x_U + Y_M x_V, \quad y_M = y_I + X_M y_U + Y_M y_V$$

$$x_V := -y_U, \quad y_V := x_U$$

Donc :

$$x_M = x_I + X_M x_U - Y_M y_U, \quad y_M = y_I + X_M y_U + Y_M x_U$$

En remplaçant x_U et y_U par leur valeur, on a :

$$x_M = x_I + \frac{X_M(x_A - x_I) - Y_M(y_A - y_I)}{\text{Longueur}(c_A, c_I)} \quad y_M = y_I + \frac{X_M(y_A - y_I) + Y_M(x_A - x_I)}{\text{Longueur}(c_A, c_I)}$$

Dans le repère IXY les coordonnées de M_1 et M_2 sont :

$$X_{M_1} = X_{M_2} = r^2/x_A$$
$$Y_{M_1} = r\frac{\sqrt{X_A^2 - r^2}}{X_A}, \quad Y_{M_2} = -Y_{M_1}$$

Donc :

$$\begin{aligned}
x_{M_1} &= x_I + \frac{r^2/x_A(x_A - x_I) - r\sqrt{1 - r^2/x_A}(y_A - y_I)}{\text{Longueur}(c_A, c_I)} \\
y_{M_1} &= y_I + \frac{r^2/x_A(y_A - y_I) + r\sqrt{1 - r^2/x_A}(x_A - x_I)}{\text{Longueur}(c_A, c_I)}, \\
x_{M_2} &= x_I + \frac{r^2/x_A(x_A - x_I) + r\sqrt{1 - r^2/x_A}(y_A - y_I)}{\text{Longueur}(c_A, c_I)}, \\
y_{M_2} &= y_I + \frac{r^2/x_A(y_A - y_I) - r\sqrt{1 - r^2/x_A}(x_A - x_I)}{\text{Longueur}(c_A, c_I)}
\end{aligned}$$

On écrit la fonction `Tangent(C,cA)` qui renvoie une liste (éventuellement vide) contenant la ou les équations des tangentes au cercle `C` passant par le point A de coordonnées `cA`, en utilisant `Milieu` (cf 6.1.2), `Longueur` (cf 6.1.1), `Droite`(cf 6.2.1) et `Cercle` (cf 6.6.3).
On tape :

```
fonction Tangent(C,cA)
 local cI,r,xI,yI,xA,yA,XM1,YM1,xM1,yM1,XM2,YM2,xM2,yM2,l;
 cI:=C[0];
 xI:=cI[0];
 yI:=cI[1];
 r:=C[1];
 l:=Longueur(cA,cI);
 xA:=cA[0];
 yA:=cA[1];
 si l < r alors
   print("A n'est pas a l'exterieur de C");
   retourne [];
 fsi;
 si l==r et yA==yI alors
```

```
   retourne [x-xA=0];
 fsi;
 si l==r et (yA-yI)!=0 alors
   retourne [normal(Droite([xA,yA],-(xA-xI)/(yA-yI)))];
 fsi;
 XM1:=r^2/l;
 YM1:=r*sqrt(1-r^2/l^2);
 xM1:=normal(xI+(XM1*(xA-xI)-YM1*(yA-yI))/l);
 yM1:=normal(yI+(XM1*(yA-yI)+YM1*(xA-xI))/l);
 XM2:=r^2/l;
 YM2:=-r*sqrt(1-r^2/l^2);
 xM2:=normal(xI+(XM2*(xA-xI)-YM2*(yA-yI))/l);
 yM2:=normal(yI+(XM2*(yA-yI)+YM2*(xA-xI))/l);
 retourne [Droite([xM1,yM1],[xA,yA]),Droite([xM2,yM2],[xA,yA])];
ffonction:;
```

```
   cI:=[0,1]
```

$$[0, 1]$$

```
   cA:=[2,1]
```

$$[2, 1]$$

```
   r:=sqrt(2);
```

$$\sqrt{2}$$

```
   C1:=Cercle(cI,r)
```

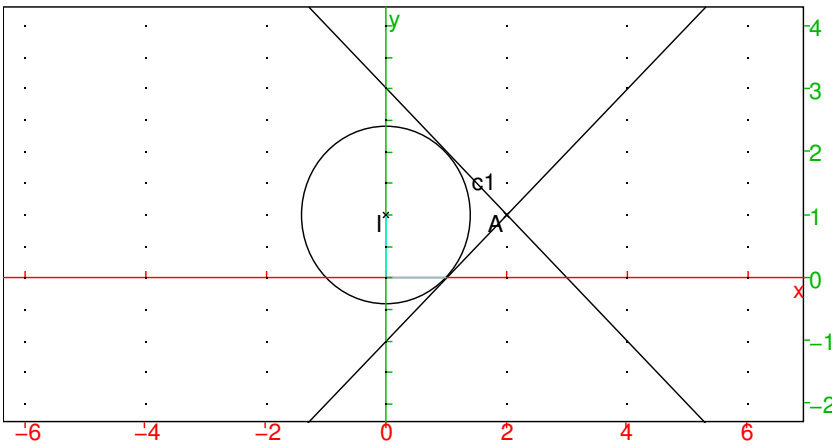
$$[[0, 1], \sqrt{2}, x^2 + y^2 - 2 \cdot y - 1 = 0]$$

```
   T1:=Tangent(C1,cA)
```

$$[-x - y + 3 = 0, x - y - 1 = 0]$$

On fait la figure avec Xcas :

```
I:=point(cI);A:=point(cA);c1:=cercle(I,r
);droite(T1[0]);droite(T1[1])
```

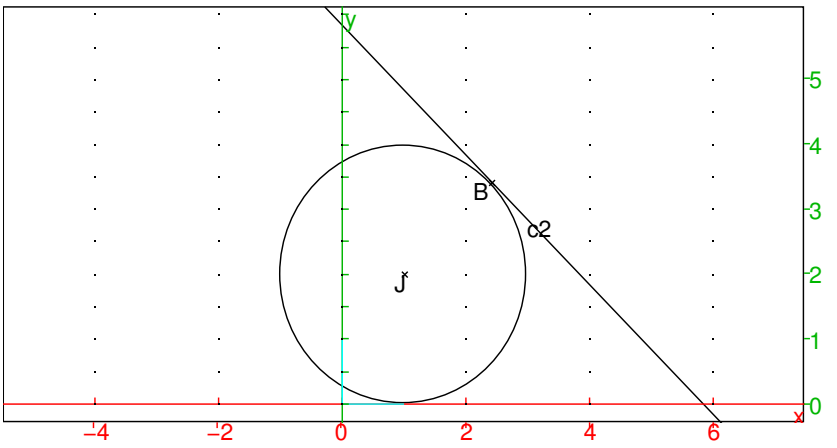



```
C2:=Cercle([1,2],2);T2:=Tangent(C2,[1+sqrt
(2),2+sqrt(2)])
```

$$[[1,2],2,x^2+y^2-2\cdot x-4\cdot y+1=0],[x+y-2\cdot\sqrt{2}-3=0]$$

On fait la figure avec Xcas :

```
J:=point(1,2);B:=point(1+sqrt(2),2+sqrt(2
));c2:=cercle(J,2);droite(T2[0]);
```



On vérifie avec Xcas :

```
equation(tangent(c1,A))
```

$$[y=(x-1),y=(-x+3)]$$

```
equation(tangent(c2,B))
```

$$y=(2\cdot\sqrt{2}+3-x)$$

On peut utiliser non seulement Milieu (cf 6.1.2), Longueur (cf 6.1.1), Droite(cf 6.2.1) et Cercle (cf 6.6.3), mais aussi le programme de changement de repère écrit précedemment ChangeXY2xy (cf 6.5.3 :

```
fonction Tangentes(C,cA)
 local cI,r,xI,yI,xA,yA,XA,CM1,CM2,cM1,cM2,cU;
 cI,r:=C;
 xI,yI:=cI;
 xA,yA:=cA;
 XA:=Longueur(cA,cI);
 cU:=[xA-xI,yA-yI]/XA;
 si XA >r alors
   CM1:=[r^2/XA,r*sqrt(1-r^2/XA^2)];
   CM2:=[r^2/XA,-r*sqrt(1-r^2/XA^2)];
   cM1:=ChangeXY2xy(CM1,cI,cU);
   cM2:=ChangeXY2xy(CM2,cI,cU);
   retourne [Droite(cM1,[xA,yA]),Droite(cM2,[xA,yA])];
 fsi;
 si XA==r alors
   CM1:=[r,1];
   cM1:=ChangeXY2xy(CM1,cI,cU);
   retourne [Droite(cM1,[xA,yA])];
 fsi
 retourne [];
ffonction:;
```

```
   cI:=[0,1]
```

$$[0,1]$$

```
cA:=[2,1]
```

$$[2,1]$$

```
r:=sqrt(2);
```

$$\sqrt{2}$$

```
C:=Cercle(cI,r)
```

$$[[0,1],\sqrt{2},x^2+y^2-2\cdot y-1=0]$$

```
Tangentes(C,cA)
```

$$[-x-y+3=0,x-y-1=0]$$

```
Tangentes(Cercle([1,2],1),[1,3])
```

$$[-y+3=0]$$

On vérfie avec `Xcas` :

```
equation(tangent(cercle(point(cI),r),point(cA)))
```

$$[y=(x-1),y=(-x+3)]$$

```
equation(tangent(cercle(point([1,2]),1),point
([1,3])))
```

$$y=3$$

6.6.7 Solution analytique des tangentes à un cercle

On peut aussi faire une résolution analytique pour construire la (ou les) tangente(s) à un cercle C passant par un point A.
On pourra se servir des programmes écrits précédement : `Longueur` (cf 6.1.1), `Droite`(cf 6.2.1), `Cercle` (cf 6.6.3) et `Solution12` (cf 4.2).

On considère un repère orthonormé dans lequel le centre I du cercle C de rayon r a pour coordonnées $[x_I, y_I]$.
C a pour équation $(x - x_I)^2 + (y - y_I)^2 = r^2$ soit

$$x^2 + y^2 - 2x_I x - 2y_I y + x_I^2 + y_I^2 - r^2 = 0$$

Soit A un point de coordonnées $[x_A, y_A]$.
Si A se trouve à l'extérieur du cercle C, on peut mener par A, deux tangentes. Les points de contact M_1 et M_2 de ces tangentes avec C sont aussi les points d'intersection de C et du cercle de diamètre IA.
Le cercle de diamètre IA a pour centre K et rayon R où K est le milieu de IA de coordonnées $[x_K, y_K] = c_K$=Milieu(cI, cA) et R =Longueur$(cI, cA)/2$.
Ce cercle a donc comme équation $(x - x_K)^+(y - y_K)^2 = R^2$ i.e.

$$x^2 + y^2 - 2x_K x - 2y_K y + x_K^2 + y_K^2 - R^2 = 0$$

Il faut donc résoudre le système d'inconnues x, y

$$\begin{aligned} x^2 + y^2 - 2xIx - 2yIy + xI^2 + yI^2 - r^2 &= 0 \\ x^2 + y^2 - 2x_K x - 2y_K y + x_K^2 + y_K^2 - R^2 &= 0 \end{aligned}$$

qui est équivalent à :

$$\begin{aligned} x^2 + y^2 - 2xIx - 2yIy + xI^2 + yI^2 - r^2 &= 0 \\ 2(xI - x_K)x + 2(yI - y_K)y + x_K^2 + y_K^2 + r^2 - xI^2 - yI^2 - R^2 &= 0 \end{aligned}$$

On a $2(xI - x_K) = (xI - x_A)$ et $2(yI - y_K) = (yI - y_A)$
Or $A \neq I$ donc $x_A \neq x_I \neq x_K$ ou $y_A \neq y_I \neq y_K$. Si $(yI - y_K) \neq 0$ (resp $(xI - x_A) \neq$) alors on connaît y (resp x) en fonction de x (resp y) et il faut résoudre une équation de degré 2 en x (res y).
Si A est sur le cercle C, on peut mener par A, une tangente (c'est une droite passant par A et qui est perpendiculaire à IA.
Si A est à l'intérieur du cercle C, il n'y a pas de tangente passant par A.
On utilise encore `Longueur` (cf 6.1.1), `Droite` (cf 6.2.1) et `Cercle` (cf 6.6.3), mais aussi le programme `Solution12` écrit précedemment (cf 4.2 : On tape :

```
fonction Tangenteq(C,cA)
```

```
 local cI,r,l,xI,yI,xA,yA,xK,yK,Eq,xM,yM,xM1,yM1,xM2,yM2,R,m;
 cI:=C[0];
 xI:=cI[0];
 yI:=cI[1];
 r:=C[1];
 l:=Longueur(cA,cI);
 xA:=cA[0];
 yA:=cA[1];
 R:=l/2;
 si l >r alors
  xK:=(xI+xA)/2;
  yK:=(yI+yA)/2;
  si (yI-yA)!=0 alors
    yM:=(xK^2+yK^2+r^2-xI^2-yI^2-R^2+2*(xI-xK)*x)/(yA-yI);
    Eq:=x^2+yM^2-2xI*x-2yI*yM+xI^2+yI^2-r^2=0;
    [xM1,xM2]:=Solution12(Eq,x);
    yM1:=subst(yM,x=xM1);
    yM2:=subst(yM,x=xM2);
  sinon
    xM:=(xK^2+yK^2+r^2-xI^2-yI^2-R^2+2*(yI-yK)*y)/(xA-xI);
    Eq:=xM^2+y^2-2xI*xM-2yI*y+xI^2+yI^2-r^2=0;
    [yM1,yM2]:=Solution12(Eq,y)
    xM1:=subst(xM,y=yM1);
    xM2:=subst(xM,y=yM2);
  fsi;
 retourne [Droite(cA,[xM1,yM1]),Droite(cA,[xM2,yM2])];
 fsi;
 si l==r alors
   si (yI-yA)!=0 alors
     m:=normal(-(xA-xI)/(yA-yI));
     retourne [Droite([xA,yA],m)];
   sinon
     retourne [Droite([xA,yA],[xA,yA+1])];
   fsi;
 fsi;
 retourne [];
ffonction:;

   cI:=[0,1]
```

$$[0,1]$$

```
cA:=[2,1]
```

$$[2,1]$$

```
r:=sqrt(2);
```

$$\sqrt{2}$$

```
C:=Cercle(cI,r)
```

$$[[0,1],\sqrt{2},x^2+y^2-2\cdot y-1=0]$$

```
Tangenteq(C,cA)
```

$$[x+y-3=0,-x+y+1=0]$$

```
Tangenteq(Cercle([1,2],1),[1,3])
```

$$[y-3=0]$$

On vérifie avec `Xcas` :

```
equation(tangent(cercle(point([0,1]),r),point
(cA)))
```

$$[y=(x-1),y=(-x+3)]$$

```
equation(tangent(cercle(point([1,2]),1),point
([1,3])))
```

$$y=3$$

Chapitre 7

Quelques tests géométriques

7.1 Test d'alignement de 3 points

Établir que trois points sont alignés ou non alignés.
Si les 3 points sont confondus `Estaligne(A,B,C)` renvoie 2
Si les 3 points sont alignés `Estaligne(A,B,C)` renvoie 1
Si les 3 points ne sont pas alignés `Estaligne(A,B,C)` renvoie 0
On tape :

```
fonction Estaligne(cA,cB,cC)
 local xA,yA,xB,yB,xC,yC;
 si cA==cB et cA==cC alors retourne 2; fsi;
 si cA==cB ou cA==cC alors retourne 1; fsi;
 xA:=cA[0];
 yA:=cA[1];
 xB:=cB[0];
 yB:=cB[1];
 xC:=cC[0];
 yC:=cC[1];
 si normal((xB-xA)*(yC-yA)-(xC-xA)*(yB-yA))==0 alors
   retourne 1;
 fsi;
 retourne 0;
ffonction:;
```

```
   Estaligne([1,2],[1,2],[1,2])
```

2

```
Estaligne([1,2],[3,6],[1,2])
```

1

```
Estaligne([1,2],[3,6],[-1,-2])
```

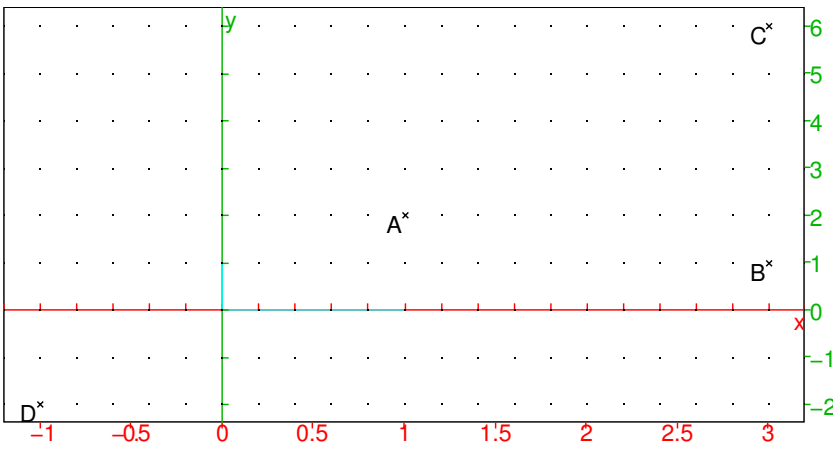
1

```
Estaligne([1,2],[1,6],[-1,-2])
```

0

On fait la figure avec Xcas :

```
A:=point(1,2);B:=point(3,1);C:=point(3,6
);D:=point(-1,-2);
```



On vérifie avec Xcas :

```
est_aligne([1,2],[3,6],[-1,-2])
```

1

```
est_aligne([1,2],[3,1],[-1,-2])
```

$$0$$

7.2 Test de parallélisme de 2 droites

Soient deux droites D_1 et D_2 d'équation :
$a_1x + b_1y + c_1 = 0$ et $a_2x + b_2y + c_2 = 0$
Ces 2 droites sont parallèles si $a_1b_2 = a_2b_1$.
On utilise la fonction `Coeffsdroite` (cf 6.2.2 qui calcule les coefficients `a`, `b` et `c` de l'équation d'une droite `ax+by+c=0` : On tape :

```
fonction Estparallele(d1,d2)
 local a1,a2,b1,b2,d,c1,c2;
 a1,b1,c1:=Coeffsdroite(d1);
 a2,b2,c2:=Coeffsdroite(d2);
 d:=normal(a2*b1-b2*a1);
 si d==0 alors retourne 1; fsi;
 retourne 0;
ffonction:;
```

```
D1:=Droite(2*x-3*y=1)
```

$$2 \cdot x - 3 \cdot y = 1$$

```
D2:=Droite(-4*x+6*y=23)
```

$$-4 \cdot x + 6 \cdot y = 23$$

```
Estparallele(D1,D2)
```

$$1$$

```
D3:=Droite(-5*x+15*y=16)
```

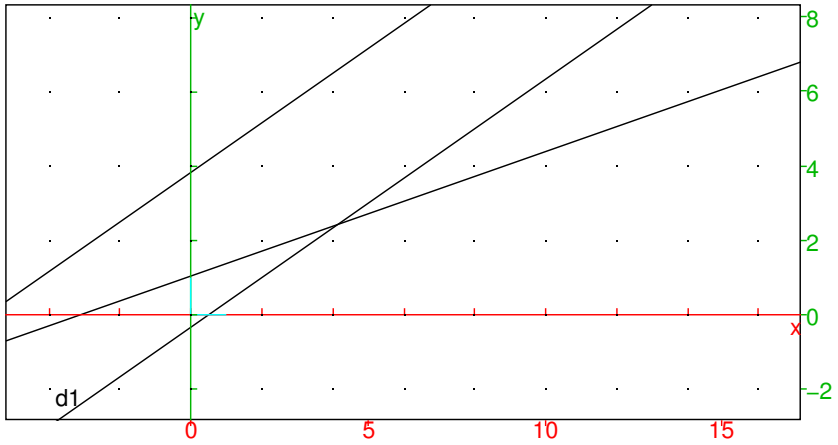
$$-5 \cdot x + 15 \cdot y = 16$$

```
Estparallele(D1,D3)
```

0

On fait la figure avec Xcas :

```
d1:=droite(2*x-3*y=1);d2:=droite(-4*x+6*y=23
);d3:=droite(-5*x+15*y=16)
```



On vérifie avec Xcas :

```
est_parallele(d1,d2)
```

1

```
est_parallele(d1,d3)
```

0

7.3 Caractériser alignement et parallélisme par la colinéarité

Trois points A, B et C sont alignés si les vecteurs AB et AC sont colinéaires.
Deux droites AB et CD sont parallèles si les vecteurs AB et CD sont colinéaires.

Chapitre 8

Statistiques

8.1 Calcul de moyenne et écart-type

Exercice : Écrire une fonction prenant en argument la liste des données et renvoyant sa moyenne et son écart-type, au moyen d'une boucle `pour` (sans utiliser `mean` ou `stddev` qui sont les commandes `Xcas` pour moyenne et écart-type).

```
fonction Stats(L)
  local j,Lj,n,s,s2;
  n:=dim(L);
  s:=0; s2:=0;
  pour j de 0 jusque n-1 faire
    Lj := L[j];
    s := s+Lj
    s2 := s2+Lj^2;
  fpour
  retourne s/n,sqrt(s2/n-(s/n)^2);
ffonction:;
```

```
L:=[7,9,11,10,7,14,11];
```

$$[7, 9, 11, 10, 7, 14, 11]$$

```
Stats(L);
```

$$\frac{69}{7}, \frac{7 \cdot \sqrt{258}}{49}$$

On vérifie avec `Xcas` :

```
mean(L); stddev(L);
```

$$\frac{69}{7}, \frac{7 \cdot \sqrt{258}}{49}$$

Exercice : Écrire une fonction prenant en argument la liste des données et renvoyant sa médiane.
Indication : Pour déterminer une médiane, il faut au préalable trier les données, ce qui est la partie difficile de l'algorithme, le reste de l'algorithme est simple. Au niveau du lycée on peut utiliser la commande `Xcas sort` qui trie une liste par ordre croissant, puis on renvoie l'élément d'indice la taille de la liste/2.

```
fonction Median(L)
  local n;
  n:=dim(L);
  L:=sort(L);
  si irem(n,2)==1 alors
    retourne L[(n-1)/2];
  sinon
    retourne (L[n/2-1]+ L[n/2])/2.;
  fsi;
ffonction:;
```

```
L1:=[148,143,142,146,144,146,142,147,140];
```

$$[148, 143, 142, 146, 144, 146, 142, 147, 140]$$

```
Median(L1)
```

$$144$$

```
L2:=[143,142,146,144,146,142,147,140];
```

$$[143, 142, 146, 144, 146, 142, 147, 140]$$

```
Median(L2)
```

143.5

On vérifie avec Xcas :

```
median(L1)
```

144.0

Exercice : Écrire une fonction qui calcule la moyenne et l'écart-type et qui a comme argument soit la liste alternant valeur et effectif soit 2 listes.

```
fonction Statexo1(L)
  local j,lj,n,n1,s,s2,vj,ej;
  n:=dim(L);
  si type(n)!=vecteur alors retourne "erreur"; fsi;
  s:=0;
  s2:=0;
  n1:=n[0];
  n:=sum(col(L,1));
  pour j de 0 jusque n1-1 faire
    lj := L[j];
    vj := lj[0];
    ej := lj[1];
    s := s+vj*ej
    s2 := s2+vj^2*ej;
  fpour;
  retourne s/n,sqrt(s2/n-(s/n)^2);
ffonction:;
fonction Statexo2(L1,L2)
  local j,vj,ej,n,n1,n2,s,s2;
  n1:=dim(L1);
  n2:=dim(L2);
  si n1!=n2 alors retourne "erreur"; fsi;
  n:=sum(L2);
  s:=0;
  s2:=0;
```

```
  pour j de 0 jusque n1-1 faire
    vj := L1[j];
    ej := L2[j];
    s := s+vj*ej
    s2 := s2+vj^2*ej;
  fpour;
  retourne s/n,sqrt(s2/n-(s/n)^2);
ffonction:;
```

On a mesuré la taille en cm (arrondie à l'entier le plus proche) de 200 fossiles de la même espèce. On a obtenu pour `k=0..12` une taille `L1[k]` d'effectif de `L2[k]`.
Calculer la moyenne et l'écart-type de cette distribution.

```
L1:=[0,1,2,3,4,5,6,7,8,9,10,11,12]
```

$$[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]$$

```
L2:=[1,16,31,37,41,30,23,13,6,1,0,1,0]
```

$$[1, 16, 31, 37, 41, 30, 23, 13, 6, 1, 0, 1, 0]$$

```
L12:=seq([L1[k],L2[k]],k,0,12)
```

$$\begin{pmatrix} 0 & 1 \\ 1 & 16 \\ 2 & 31 \\ 3 & 37 \\ 4 & 41 \\ 5 & 30 \\ 6 & 23 \\ 7 & 13 \\ 8 & 6 \\ 9 & 1 \\ 10 & 0 \\ 11 & 1 \\ 12 & 0 \end{pmatrix}$$

```
Statexo1(L12)
```

$$4, \frac{2 \cdot \sqrt{365}}{20}$$

On a mesuré la taille en cm (arrondie à l'entier le plus proche) d'une autre espèce de fossiles. On a obtenu pour `k=0..12` une taille `L1[k]` d'effectif de `L3[k]`.
Calculer la moyenne et l'écart-type de cette distribution.

```
L1:=[0,1,2,3,4,5,6,7,8,9,10,11,12]
```

$$[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]$$

```
L3:=[2,11,21,52,64,77,62,46,41,11,8,5,0]
```

$$[2, 11, 21, 52, 64, 77, 62, 46, 41, 11, 8, 5, 0]$$

```
Statexo2(L1,L3)
```

$$\frac{1053}{200}, \frac{200 \cdot \sqrt{182991}}{40000}$$

On vérifie avec `Xcas` :

```
mean(L1,L2),stddev(L1,L2)
```

$$4, \frac{2 \cdot \sqrt{365}}{20}$$

```
mean(L1,L3),stddev(L1,L3)
```

$$\frac{1053}{200}, \frac{200 \cdot \sqrt{182991}}{40000}$$

8.2 Simulation d'un échantillon

Exercice : Générer aléatoirement n valeurs 0 ou 1, la probabilité d'avoir 1 étant fixée à p (pour faire cela, on comparera avec p le résultat de la commande `Xcas alea(0,1)` qui renvoie un réel entre 0 et 1). Calculer la fréquence de 1 observée.

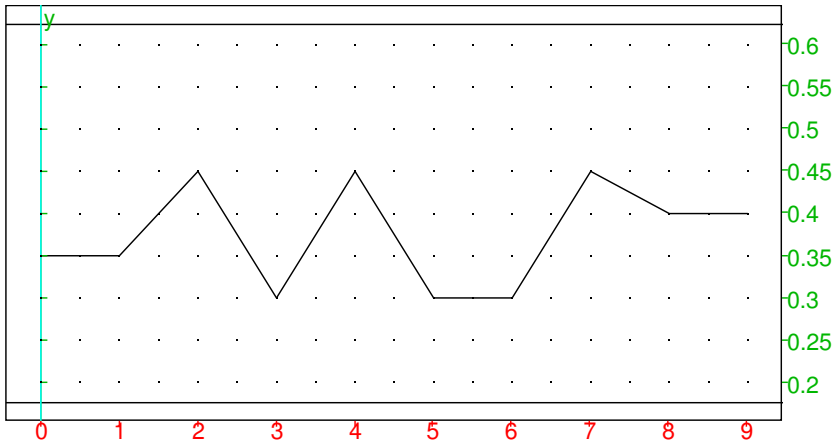
```
fonction Simu(n,p)
  local j,a,n1;
  n1:=0;
  pour j de 1 jusque n faire
    a:=alea(0,1);
    si a<=p alors n1:=n1+1; fsi;
  fpour
  retourne n1/n;
ffonction:;
```

```
L:=seq(Simu(20.,.4),k,1,10);
```

$$[0.35, 0.35, 0.45, 0.3, 0.45, 0.3, 0.3, 0.45, 0.4, 0.4]$$

On peut visualiser les fréquences obtenues avec la commande `plotlist([y0,...,yn])` de `Xcas` qui trace la ligne polygonale reliant les points d'abscisse `k` et d'ordonnée `yk` pour `k=0..n`.

```
plotlist(L);droite(y=0.4-1/sqrt(20)); droite
(y=0.4+1/sqrt(20))
```



Commandes `Xcas` permettant de faire le calcul directement :

```
n:=20.; p:=.4; v:=randvector(n,binomial,1,p
); sum(v)/n;
```

$$20.0, 0.4, [0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0], 0.2$$

La commande `randvector` génère ici n valeurs aléatoires selon la loi binomiale de paramètres 1 et p, on en fait ensuite la moyenne. On peut bien sur utiliser d'autres lois que la loi binomiale, par exemple la loi uniforme (`uniform`), la loi normale (`normald`), la loi exponentielle (`exponential`), ...

8.3 Intervalle de fluctuation

Écrire un algorithme effectuant N simulations d'échantillons, en utilisant la fonction précédente, renvoyer la liste des fréquences ainsi que la proportion de fréquences situées en-dehors de l'intervalle $[p - 1/\sqrt{n}, p + 1/\sqrt{n}]$.

Rappelons qu'on s'intéresse à cet intervalle, parce qu'une simulation suit une loi binomiale de moyenne np et d'écart-type $\sigma = \sqrt{np(1-p)}$ donc les fréquences suivent une loi binomiale de moyenne p et d'écart-type $\tilde{\sigma} = \sigma/n = \sqrt{p(1-p)/n}$. Or

$$4p(1-p) = 1 - (2 * p - 1)^2 \Rightarrow 4p(1-p) \leq 1$$

donc

$$2\tilde{\sigma} = 2\frac{\sigma}{n} = \sqrt{\frac{4p(1-p)}{n}} \leq \frac{1}{\sqrt{n}}$$

Pour n grand, la loi binomiale s'approche d'une loi normale ayant cet écart type.

On utilise la fonction `Simu` écrite précédemment (cf 8.2).

```
fonction Fluctuat(N,n,p)
  local j,L,out;
  L:=[];
  out:=0;
  pour j de 0 jusque N-1 faire
    L[j]:=Simu(n,p);
    si abs(L[j]-p)>1/sqrt(n) alors out:=out+1; fsi;
  fpour;
  retourne out/N,L;
ffonction:;
```

On lance 20 fois une pièce de de monnaie mal équilibrée, la probabilité d'obtenir face étant égale à 0.4.
`Simu(20,0.4)` renvoie donc la fréquence du nombre de faces observées. On effectue plusieurs fois 100 simulations :

```
L:=Fluctuat(100.,20,.4);L1:=L[1]:;
```

$$0.05, [\frac{1}{2}, \frac{3}{10}, \frac{7}{20}, \frac{1}{4}, \frac{9}{20}, \frac{1}{5}, \frac{7}{20}, \frac{9}{20}, \frac{13}{20}, \frac{3}{10}, \frac{9}{20}, \frac{3}{10}, \frac{7}{20}, \frac{9}{20}, \frac{1}{2}, \frac{2}{5}, \frac{1}{2}, \frac{2}{5}, \frac{2}{5}, \frac{2}{5}, \frac{2}{5}, \frac{1}{2}, \frac{7}{20}, \frac{7}{20}, \frac{7}{20}, \frac{3}{5}, \frac{3}{10}$$

```
LL:=Fluctuat(100.,20,.4);LL1:=LL[1]:;
```

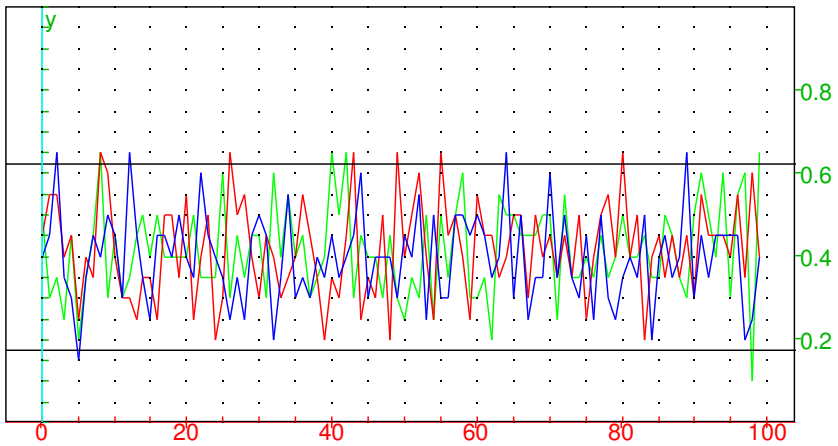
$$0.06, [\frac{9}{20}, \frac{11}{20}, \frac{11}{20}, \frac{2}{5}, \frac{9}{20}, \frac{1}{4}, \frac{2}{5}, \frac{7}{20}, \frac{13}{20}, \frac{3}{5}, \frac{2}{5}, \frac{3}{10}, \frac{3}{10}, \frac{1}{4}, \frac{7}{20}, \frac{7}{20}, \frac{1}{4}, \frac{1}{2}, \frac{1}{2}, \frac{7}{20}, \frac{11}{20}, \frac{1}{4}, \frac{2}{5}, \frac{1}{2}, \frac{1}{5}, \frac{3}{10}, \frac{13}{20}$$

```
LLL:=Fluctuat(100.,20,.4);LLL1:=LLL[1]:;
```

$$0.05, [\frac{2}{5}, \frac{9}{20}, \frac{13}{20}, \frac{7}{20}, \frac{3}{10}, \frac{3}{20}, \frac{7}{20}, \frac{9}{20}, \frac{2}{5}, \frac{1}{2}, \frac{9}{20}, \frac{3}{10}, \frac{13}{20}, \frac{9}{20}, \frac{7}{20}, \frac{1}{4}, \frac{9}{20}, \frac{9}{20}, \frac{2}{5}, \frac{1}{2}, \frac{2}{5}, \frac{7}{20}, \frac{3}{5}, \frac{9}{20}, \frac{2}{5}, \frac{7}{20}$$

On peut visualiser la répartition de ces fréquences :

```
gl_y=0..1;purge(x,y);plotlist(L1,affichage=vert
);plotlist(LL1,affichage=rouge);plotlist
(LLL1,affichage=bleu);droite(y=0.4-1/sqrt
(20)); droite(y=0.4+1/sqrt(20))
```



Avec les commandes de Xcas

On génère une séquence de N moyennes de vecteurs de n valeurs aléatoires selon la loi binomiale de paramètres 1 et p :

```
N:=100.; n:=20; p:=.4;l:=seq(sum(randvector
(n,binomial,1,p))/n,j,1,N);
```

$$100.0, 20, 0.4, [\frac{11}{20}, \frac{3}{10}, \frac{1}{2}, \frac{9}{20}, \frac{9}{20}, \frac{7}{20}, \frac{2}{5}, \frac{1}{4}, \frac{7}{20}, \frac{9}{20}, \frac{11}{20}, \frac{2}{5}, \frac{3}{10}, \frac{2}{5}, \frac{3}{10}, \frac{7}{20}, \frac{1}{4}, \frac{9}{20}, \frac{1}{4}, \frac{3}{10}, \frac{1}{2}, \frac{7}{20}, \frac{2}{5}, \frac{3}{10}, \frac{2}{5}, \frac{2}{5},$$

On compte les moyennes dont l'écart à p est plus grand que $1/\sqrt{n}$:

```
count(x->abs(x-p)>1/sqrt(n),l)/N;
```

$$0.03$$

On peut accélérer le calcul en utilisant le générateur aléatoire selon la loi binomiale de paramètres n et p

```
N:=100.; n:=20; p:=.4;l:=randvector(N,binomial,n,p
)/n;
```

$$100.0, 20, 0.4, [\frac{1}{2}, \frac{9}{20}, \frac{3}{20}, \frac{3}{10}, \frac{3}{10}, \frac{3}{20}, \frac{7}{20}, \frac{9}{20}, \frac{1}{2}, \frac{3}{5}, \frac{3}{10}, \frac{1}{2}, \frac{9}{20}, \frac{3}{10}, \frac{1}{2}, \frac{3}{20}, \frac{2}{5}, \frac{11}{20}, \frac{3}{10}, \frac{9}{20}, \frac{7}{20}, \frac{2}{5}, \frac{3}{10}, \frac{3}{4}, \frac{7}{20},$$

```
count(x->abs(x-p)>1/sqrt(n),l)/N;
```

$$0.05$$

On peut enfin tracer le graphe de l'évolution de la proportion de fréquences en-dehors de l'intervalle $[p-1/\sqrt{n}, p+1/\sqrt{n}]$ lorsque N augmente. On génère une liste de fréquences :

```
N:=10000.; n:=20; p:=.4;l:=randvector(N,binomial,n,p
)/n:;
```

$$10000.0, 20, 0.4, \text{Done}$$

puis on compte les fréquences en-dehors de l'intervalle et on stocke les fréquences dans une liste

```
L:=[]; compteur:=0; a:=p-1/sqrt(n); b:=p+1/sqrt(n);
pour j de 0 jusque N-1 faire
  si l[j]<a ou l[j]>b alors compteur++; fsi;
  L[j]:=compteur/(j+1);
fpour;
gl_y=0..0.1;plotlist(L)
```

On a utilisé ici la fonction d'incrémentation, `compteur++` est équivalent à `compteur=compteur+1`.

La probabilité théorique d'être dans l'intervalle $[p-1/\sqrt{n}, p+1/\sqrt{n}]$ est donnée par la commande `Xcas` :

```
binomial_cdf(n,p,n*(p-1/sqrt(n)),n*(p+1/sqrt(n)))
```

$$0.963009909732$$

Il y a donc une probabilité faible (de l'ordre de 4%) de ne pas être dans l'intervalle de fluctuation.
La probabilité d'être dans l'intervalle plus précis $[np-\sigma/\sqrt{n}, np+\sigma/\sqrt{n}]$ est donnée par :

```
sigma:=stddev(binomial(n,p));binomial_cdf
(n,p,n*p-2*sigma,n*p+2*sigma)
```

$$2.19089023002, 0.963009909732$$

Elle est ici identique parce que les bornes de l'intervalle ont la même partie entière.

8.4 Évolution d'une fréquence.

On illustre l'évolution d'une fréquence vers sa moyenne théorique (espérance).

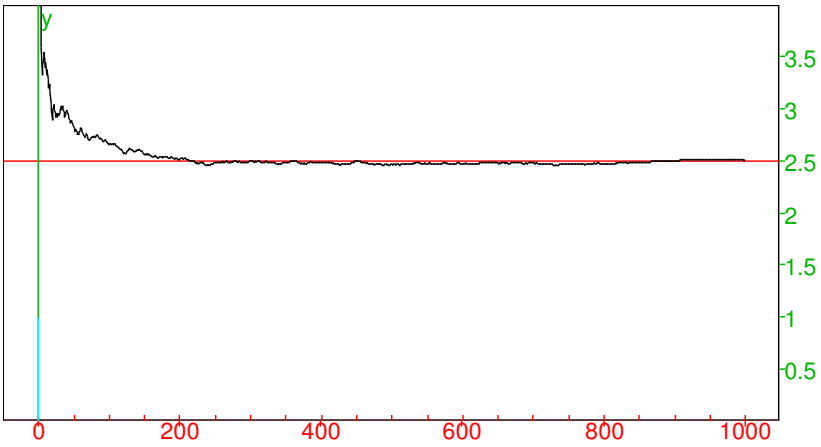
Expérience : on tire un dé à 6 faces, si le résultat est 1 on renvoie 1, si c'est 2 à 4 on renvoie 2, si c'est 5 ou 6 on renvoie 4. Simulation, en utilisant la fonction `piecewise` :

```
fonction experience()
  local a;
  a:=alea(6)+1; // simule un de 6
  return piecewise(a<2,1,a<5,2,4);
ffonction:;
```

On effectue N expériences et on regarde l'évolution de la moyenne des résultats.

```
fonction evolutionmoyenne1(N)
  local l,s,j,a;
  l:=[];
  s:=0;
  pour j de 1 jusque N faire
    s:=s+experience();
    l:=append(l,s/j);
  fpour;
  return l;
ffonction:;
```

```
purge(x,y);droite(y=2.5,couleur=rouge);gl_y=0..4;plotlist
(evolutionmoyenne1(1000))
```



Dans la version HTML interactive, cliquez plusieurs fois sur le bouton ok pour voir différents comportements.

On peut écrire une fonction plus générale en passant la fonction `experience` en argument

```
fonction evolutionmoyenne(experience,N)
  local l,s,j,a;
  l:=[];
  s:=0;
  pour j de 1 jusque N faire
    s:=s+experience();
    l:=append(l,s/j);
```

```
  fpour;
  return l;
ffonction:;
```

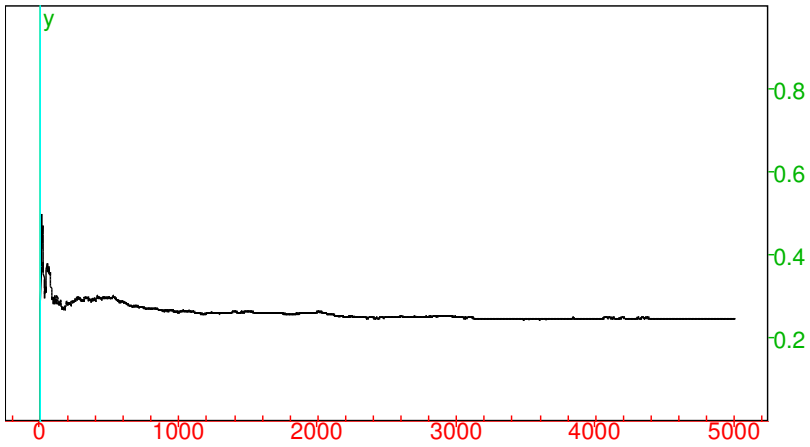
8.5 Triangles de spaghettis

On prend un spaghetti qu'on casse en 3 morceaux, peut-on construire un triangle avec ces 3 morceaux ? On fixe l'unité de longueur au spaghetti entier. On a donc 3 longueurs a, b et $c = 1 - a - b$ et il faut vérifier les inégalités triangulaires $a < b + c, b < a + c, c < a + b$ (voir la section 9.4).

Première méthode de coupe : on tire au hasard x et y entre 0 et 1. Si $x < y$, on pose $a = x, b = y - x$ (et donc $c = 1 - y$) sinon on échange x et y ($a = y$, $b = x - y$ et $c = 1 - x$). Simulation, `correct` vérifie l'inégalité triangulaire, `spag1` renvoie 1 si on peut faire un triangle et 0 sinon :

```
correct(a,b,c):=a<b+c et b<a+c et c<a+b:;
fonction spag1()
  local x,y,c;
  x:=alea(0,1);
  y:=alea(0,1);
  si y>x alors
    return correct(x,y-x,1-y);
  sinon
    return correct(y,x-y,1-x);
  fsi;
ffonction:;
fonction spagn(n)
  local L,s;
  L:=[]; // liste des frequences de succes
  s:=0; // nombre de succes
  pour j de 1 jusque n faire
    si spag1() alors s++ fsi;
    L:=append(L,s/j);
  fpour;
  return L;
ffonction:;
```

```
  gl_y=0..1;plotlist(spagn(5000))
```



Exercice : modifier `spag1` pour couper le spaghetti en prenant au hasard, puis en coupant le morceau le plus long au hasard.

```
fonction spag1()
  local x,y,c;
  x:=alea(0,1);
  si x>1/2 alors
    y:=x; // on casse x
    x:=x*alea(0,1);
  sinon // on casse 1-x
    y:=x+(1-x)*alea(0,1);
  fsi;
  return correct(x,y-x,1-y);
ffonction:;
```

Voir l'étude à la fin du manuel Algorithmes et simulation de `Xcas`.

8.6 Les aiguilles de Buffon

Le naturaliste Buffon, en 1777 a posé le problème de l'aiguille en ces termes : "Je suppose que dans une chambre, dont le parquet est simplement divisé par des joints parallèles, on jette en l'air une baguette et que l'un des joueurs parie que la baguette ne croisera aucune des parallèles du parquet..."

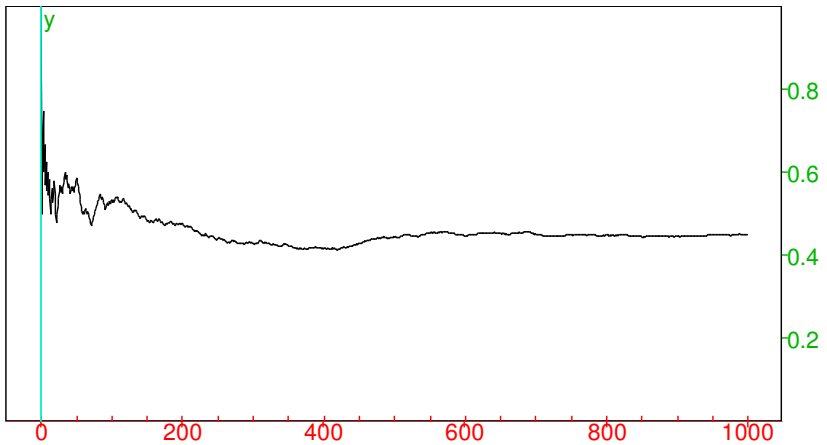
On peut modéliser le problème en supposant que la distance entre les paralléles est de 1, et que les parallèles sont horizontales, d'équations $y = k, k \in \mathbb{Z}$. On suppose que la longueur de l'aiguille est l. Lorsqu'elle tombe sur le parquet, on suppose qu'une de ses extrémités est en (x, y) et qu'elle fait un angle θ avec l'horizontale, son autre extrémité est donc en $(x+l\cos(\theta), y+l\sin(\theta))$. Elle coupe une des parallèle si l'intervalle $[y, y + l\sin(\theta)]$ contient

un entier c'est-à-dire si la partie entière de y diffère de celle de $y + l\sin(\theta)$. On observe que la valeur de x n'a pas d'importance, et que la valeur de y modulo 1 non plus. Pour effectuer la simulation, on suppose donc que y est un réel aléatoire selon la loi uniforme entre 0 et 1, et θ un réel aléatoire selon la loi uniforme entre $-\frac{\pi}{2}$ et $\frac{\pi}{2}$

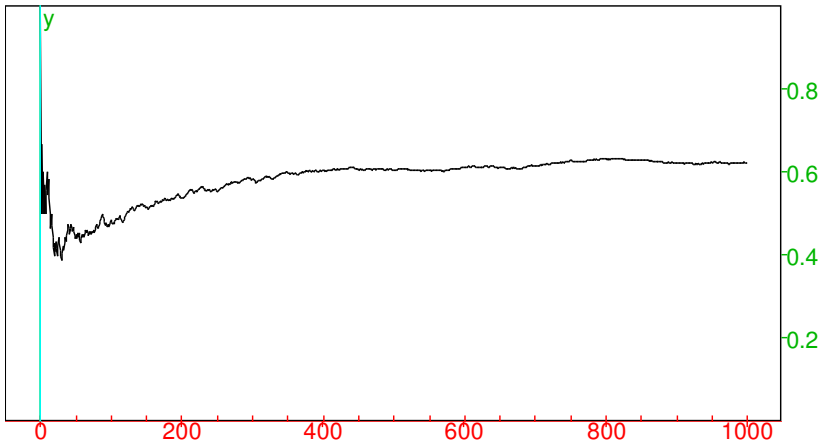
```
fonction buffonpi(l)
  local y,theta;
  y:=alea(0,1);
  theta:=alea(-.5,.5)*pi;
  si floor(y+l*sin(theta))!=0 alors return 1; sinon return 0; fsi;
ffonction:;
```

Tests pour quelques valeurs de l

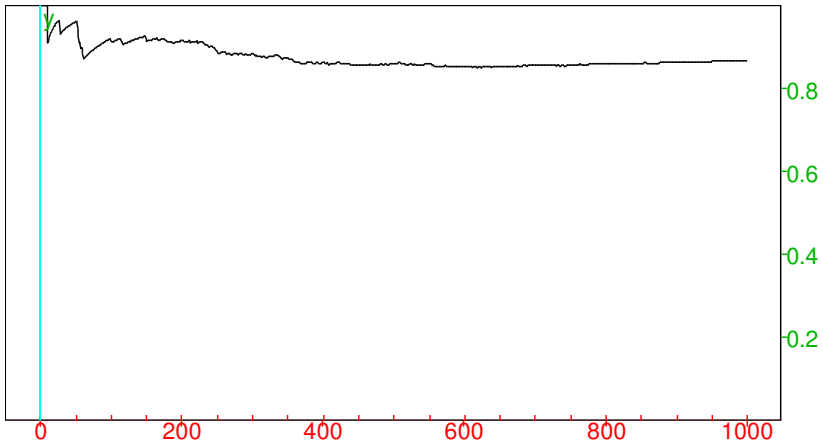
```
experience():=buffonpi(0.7);gl_y=0..1;plotlist
(evolutionmoyenne(experience,1000))
```




```
experience():=buffonpi(1);gl_y=0..1;plotlist
(evolutionmoyenne(experience,1000))
```

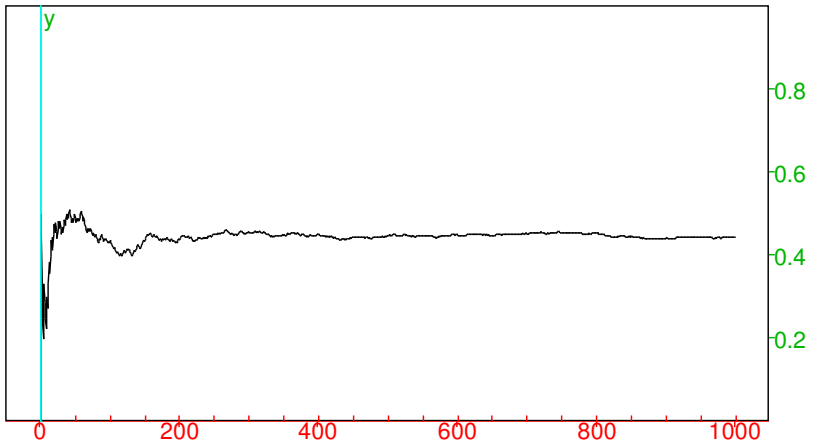



```
experience():=buffonpi(2.5);gl_y=0..1;plotlist
(evolutionmoyenne(experience,1000))
```



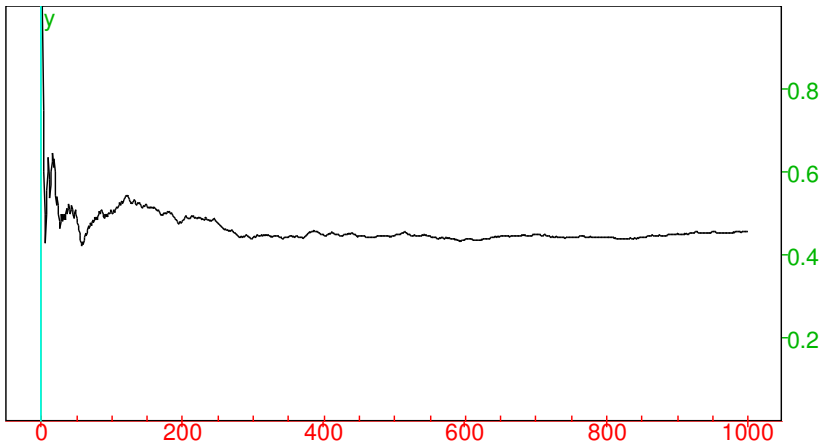
On peut aussi générer la direction aléatoire **sans utiliser le nombre** π en tirant au hasard deux réels entre 0 et 1 et si le point obtenu est dans le disque unité, on prend la direction obtenue.

```
fonction buffonsanspi(l)
  local y,X,Y;
  y:=alea(0,1);
  repeter X:=alea(0,1); Y:=alea(0,1) jusqua X*X+Y*Y<=1;
  si floor(y+l*Y/sqrt(X^2+Y^2))!=0 alors return 1; sinon return 0; fsi;
ffonction:;

experience():=buffonsanspi(0.7);gl_y=0..1;plotlist
(evolutionmoyenne(experience,1000))
```



On peut enfin définir algébriquement

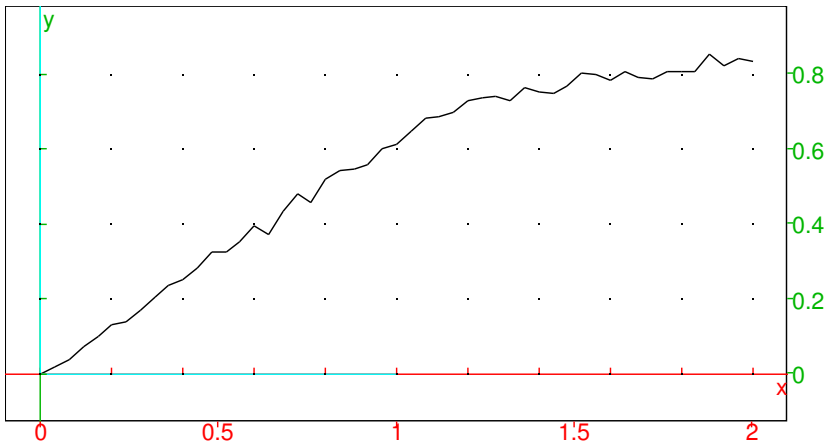
```
buffon(l):=floor(alea(0,1)+l*sin(pi*alea
(-.5,.5)))!=0;experience():=buffon(0.7);gl_y=0..1;plotlist
(evolutionmoyenne(experience,1000))
```



Pour conjecturer le comportement de la probabilité en fonction de l on va l'estimer par la fréquence observée au bout de N d'expériences, par exemple on prendra $N = 1000$ (pour éviter un temps d'exécution trop long)

```
fonction probabuffon(l,N)
  local s,j;
  s:=0;
  pour j de 1 jusque N faire
    s := s+buffonpi(l);
  fpour;
  return s/N;
ffonction:;
```

```
plot('probabuffon(l,1000)',l=0..2,xstep=0.04)
```



Pour l petit, le comportement est linéaire, pour l grand on observe une asymptote horizontale (qui est $y = 1$ car lorsque la taille l de l'aiguille est de plus en plus grande, la probabilité de ne pas couper une horizontale tend vers 0 puisque la direction θ doit être de plus en plus proche). On peut démontrer que le coefficient de proportionnalité pour l petit vaut $\frac{2}{\pi}$, (cf. le manuel Exercices de `Xcas` pour $l = 1$).

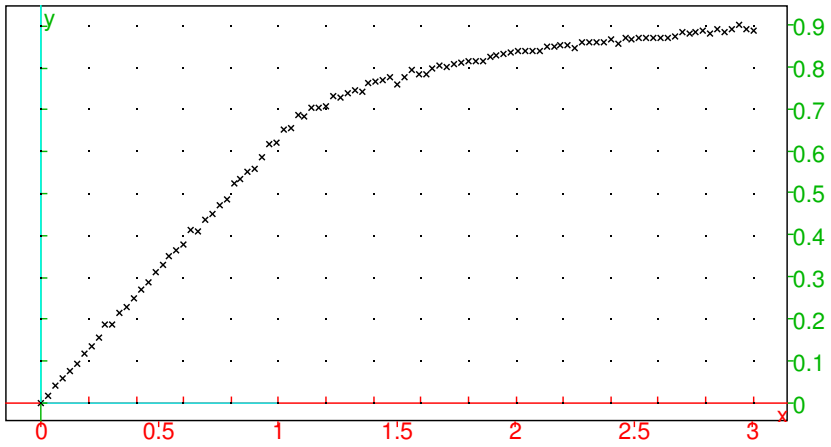
Pour obtenir un tracé plus précis sans attendre trop longtemps, il faut trouver une astuce pour accélérer le calcul dans `probabuffon`. Pour cela on part de la version algébrique de `buffon` qui se prête au calcul appliqué en parallèle à une liste (`sin` appliqué à une liste renvoie la liste des sinus des éléments de la liste, `+` effectue la somme composante par composante, etc.). On doit renvoyer 0 si le nombre de parallèles traversées est nul et 1 sinon, donc si on fait N expériences, la proportion est donnée par :

```
Buffon(l,N):=1-count_eq(0,floor(randvector
(N,uniformd,0,1)+l*sin(pi*randvector(N,uniformd,-.5,.5
))))/N;
```

```
(l,N)->1-count_eq(0,floor(randvector(N,'uniformd',0,1)+l*sin(pi*randvector(N,'un
```

On peut maintenant tracer le graphe de manière plus précise, par exemple avec $N = 5000$ et un pas de 0.03 sur $[0, 3]$, sans patienter trop longtemps :

```
seq(point(l,Buffon(l,5000)),l,0,3,0.03)
```



Ce type d'optimisation est fréquent dans les langages interprétés, on essaie d'utiliser les commandes natives du logiciel pour effectuer les boucles les plus internes au lieu de les faire exécuter par l'interpréteur, forcément plus lent que les commandes natives qui sont compilées.

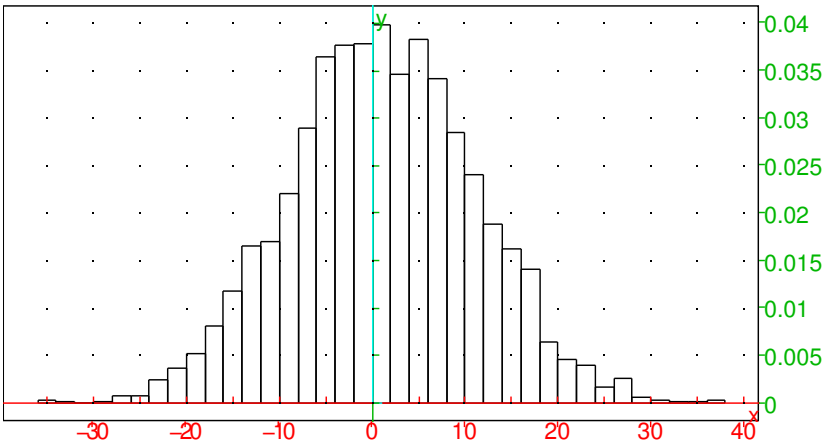
8.7 Marche aléatoire à 1 dimension.

Un marcheur ivre se déplace sur un trottoir, en faisant un pas aléatoirement vers la droite ou vers la gauche. On simule sa position après n pas (on suppose qu'il part de l'origine).

```
fonction marche(n)
  local j,x;
  x:=0;
  pour j de 1 jusque n faire
    si alea(2)==1 alors x++; sinon x--; fsi;
  fpour;
  return x;
ffonction:;
```

On effectue N marches aléatoires et on trace l'histogramme des positions, par exemple pour $n = 100$ et $N = 3000$

```
histogram(seq(marche(100),3000),-50,2)
```

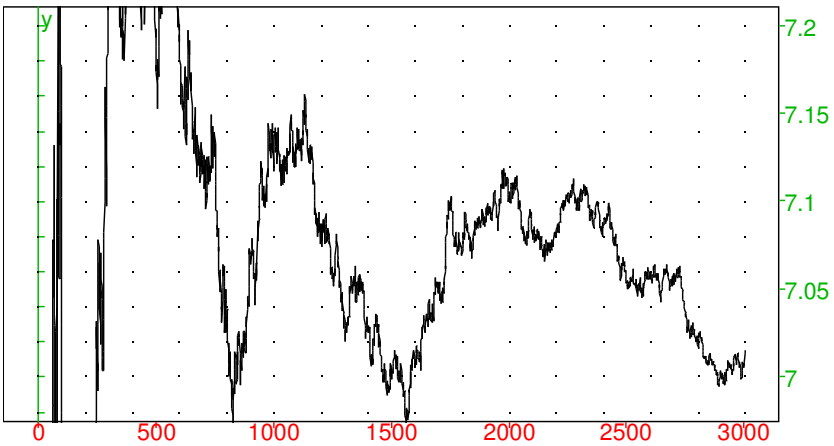


On peut aussi s'intéresser au nombre de passage par l'origine.

```
fonction retour(n)
  local j,x,r;
  x:=0; r:=0;
  pour j de 1 jusque n faire
    si alea(2)==1 alors x++; sinon x--; fsi;
    si x==0 alors r++; fsi;
  fpour;
  return r;
ffonction:;
```

et conjecturer son espérance par observation d'un nombre important de marches

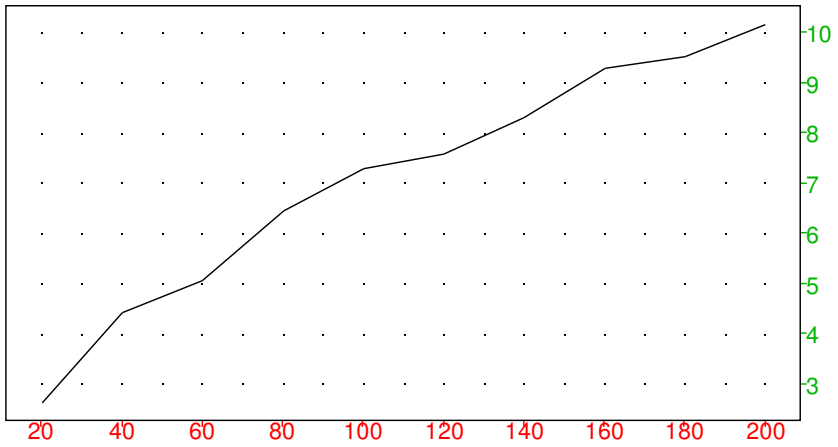
```
experience():=retour(100);plotlist(evolutionmoyenne
(experience,3000))
```



On peut aussi observer l'espérance en fonction du nombre de pas. Pour cela, on approche l'espérance par exemple par la moyenne de 1000 nombre de retours.

```
fonction moyenneretour1(n)
  local s,j;
  s:=0;
  pour j de 1 jusque 1000 faire
    s := s+retour(n);
  fpour;
  return s/1000;
ffonction:;
```

```
plot('moyenneretour1(n)',n=20..200,xstep=20)
```



Comme pour les aiguilles de Buffon, le temps de calcul est long (c'est pour cela que l'on a mis un pas de 20 dans la commande précédente), mais il est possible d'accélérer en faisant plusieurs optimisations :

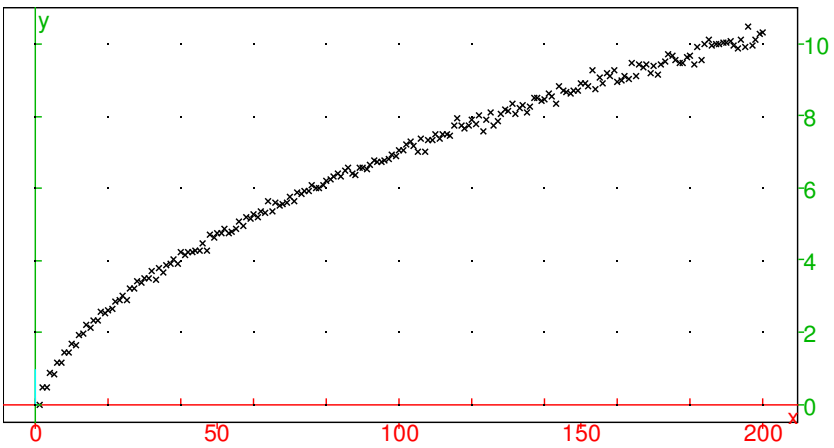
- au lieu de décaler de 1 ou -1, on peut décaler de 1 ou 0 en décidant que l'origine se décale elle-même de 1/2.
- le passage par l'origine ne peut s'effecteur qu'après un nombre pair de pas, on peut donc faire 2 pas à la fois avant de tester. À chaque itération, on fait donc deux pas de valeur 1 ou 0 et l'origine est décalée de 1.
- on remplace les boucles les plus internes par du calcul sur des listes.

```
fonction moyenneretour(N,n) // N experiences, marche de n
  local k,pos,r;
  pos:=seq(0,N); // N marcheurs a l'origine
```

```
  r:=0;
  n:=iquo(n,2);
  pour k de 1 jusque n faire
    pos:=pos+randvector(N,2)+randvector(N,2);
    r:=r+count_eq(k,pos); // nombre de passage origine translatee
  fpour;
  retourne r/N;
ffonction:;
```

On peut maintenant faire le tracé avec une bonne précision en quelques secondes :

```
seq(point(n,moyenneretour(2000,n)),n,1,200)
```



8.8 Les urnes de Polya

On place dans une urne r boules rouges et b boules bleues, indiscernables au toucher. On tire une boule, on note sa couleur, et on remet dans l'urne cette boule ainsi qu'une boule de même couleur (le nombre total de boules augmente donc de 1). On recommence, on effectue en tout n tirages.

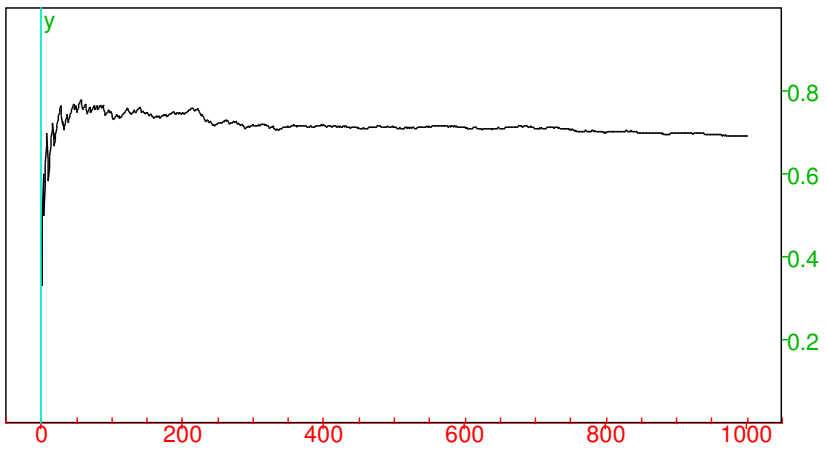
On fait une simulation, on va renvoyer la liste des proportions de boules rouges dans l'urne au cours de l'expérience. Il y a r boules rouges parmi $r+b$, on convient donc que la boule tirée est rouge si un tirage d'entier dans $[0, r+b[$ est strictement inférieur à r.

```
fonction polya(r,b,n)
  local L,j;
  L:=[r/(r+b)];
```

```
  pour j de 1 jusque n faire
    si alea(r+b)<r alors r++; sinon b++; fsi;
    L[j]:=r/(r+b);
  fpour;
  return L;
ffonction:;
```

Exemple :
Simulation avec 1000 tirages et 1 boule rouge et 1 boule bleue au début :

```
gl_y=0..1;plotlist(polya(1,1,1000))
```

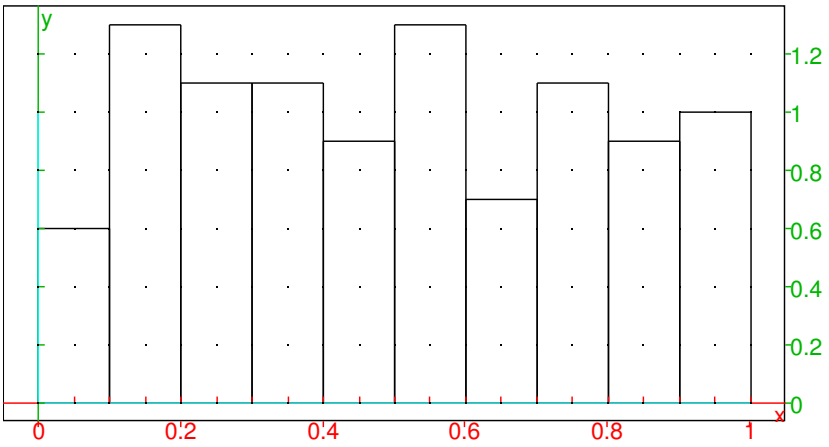


Dans la version HTML, si on clique plusieurs fois, on observe que la valeur limite est très variable. On peut s'intéresser à la répartition de cette proportion limite de boules rouges (obtenue ici au bout de 1000 étapes). On modifie la fonction précédente pour ne renvoyer que la dernière proportion.

```
fonction polya1(r,b,n)
  local j;
  pour j de 1 jusque n faire
    si alea(r+b)<r alors r++; sinon b++; fsi;
  fpour;
  return r/(r+b);
ffonction:;
```

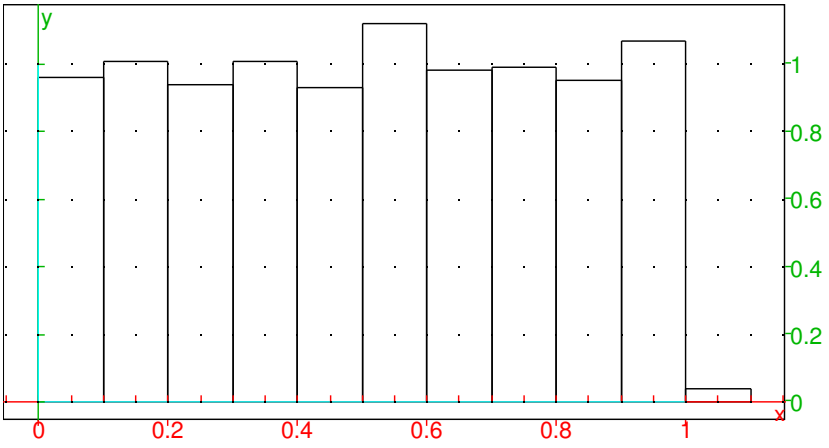
Puis on fait l'histogramme des fréquences pour N expériences

```
N:=100;histogramme(seq(polya1(1,1,1000),N),0,0.1);
```



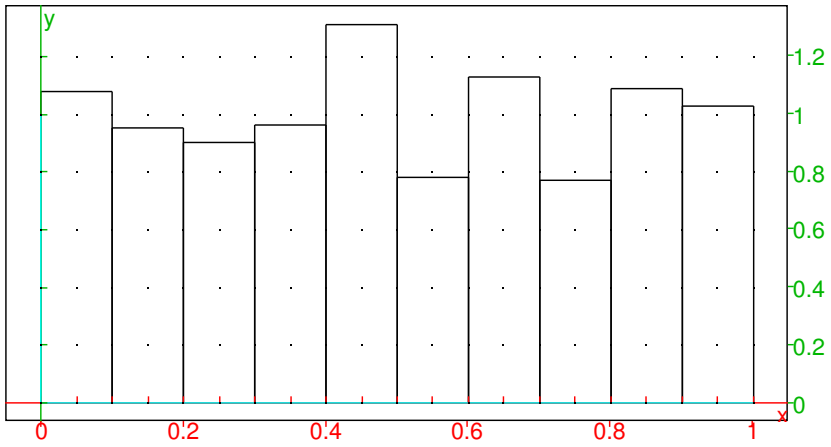
Le temps de calcul commence à s'allonger, on peut essayer d'optimiser un peu. On observe que le nombre total de boules augmenté de 1 à chaque tirage, il suffit donc de suivre la valeur de r

```
fonction polya2(r,b,n)
  local j,total,totalalafin;
  totalalafin:=r+b+n-1;
  pour total de r+b jusque totalalafin faire
    si alea(total)<r alors r++; fsi;
  fpour;
  return r/(total-1);
ffonction:;

N:=1000;histogramme(seq(polya2(1,1,1000),N
),0,0.1);
```



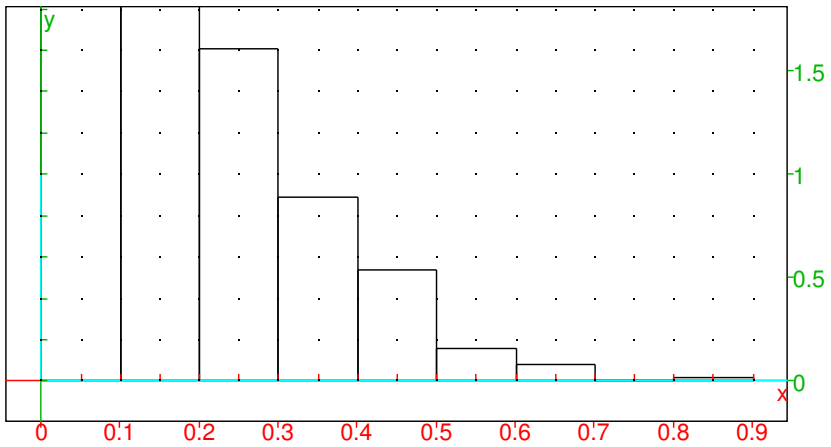
Une autre façon d'optimiser consiste à observer que `j` ne sert pas dans la boucle `pour` de `polya1`.

```
fonction polya3(r,b,n)
  seq(when(alea(r+b)<r,r++, b++),n);
  return r/(r+b);
ffonction:;

N:=1000;histogramme(seq(polya3(1,1,1000),N
),0,0.1);
```



Pour $N = 1000$, on s'aperçoit que la distribution semble uniforme. Ceci n'est plus le cas pour d'autres valeurs de r et b, dans la version HTML, vous pouvez tester diverses valeurs initiales, par exemple 5,5 ou 1,5.

```
N:=1000;histogramme(seq(polya3(1,5,1000),N
),0,0.1);
```



Chapitre 9

Les algorithmes du document ressource Python d'eduscol

On propose dans ce chapitre des versions `Xcas` de programmes Python donnés dans ce document [1]

Pour les statistiques, voir le chapitre 8.

Pour la dichotomie, voir 4.1.

9.1 Arithmétique

On renvoie au manuel de programmation de `Xcas` (menu Aide, Manuels), on donne ici rapidement les équivalents `Xcas` du document eduscol.

9.1.1 Euclide

On calcule le PGCD de a et b en observant que si b est nul c'est a, sinon c'est le PGCD de b et du reste de la division euclidienne de a par b (`irem(a,b)`)

```
fonction pgcd(a,b)
  tantque b!=0 faire
    a,b:=b,irem(a,b);
  ftantque;
  return a;
ffonction:;
```

1. http://cache.media.eduscol.education.fr/file/Mathematiques/73/3/Algorithmique_et_programmation

```
pgcd(25,15)
```

5

9.1.2 Écriture en base 2

L'écriture de n en base 2 est l'écriture de $n/2$ à laquelle on ajoute le reste de la division n par 2.

```
fonction base2(n)
  local l;
  l:=[];
  tantque n>0 faire
    l:=append(l,irem(n,2));
    n:=iquo(n,2);
  ftantque;
  return revlist(l);
ffonction:;
```

Il faut inverser l'ordre des éléments dans la liste avant de la renvoyer, parce qu'on calcule en premier le dernier bit de l'écriture de n en base 2.

```
l:=base2(12345)
```

$$[1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1]$$

L'opération inverse se fait en utilisant le même principe : lorsqu'on ajoute un bit à la fin de l'écriture de n en base 2, cela revient à multiplier n par 2 et lui ajouter la valeur de ce bit.

```
fonction base10(l)
  local n,x;
  n:=0;
  pour x in l faire
    n=2*n+x;
  fpour;
  return n;
ffonction:;
```

```
base10(l)
```

12345

La taille de l'écriture en base 2 s'obtient par `dim(l)` ou directement par le programme suivant

```
fonction taille2(n)
  local l;
  l:=0;
  tantque n>0 faire
    n:=iquo(n,2);
    l++;
  ftantque;
  return l;
ffonction:;
```

```
   taille2(12345)
```

14

9.1.3 Test de primalité

On teste si n est divisible par un entier $d \leq \sqrt{n}$.

```
fonction estpremier(n)
  local d;
  si n<2 alors return false; fsi;
  d:=2;
  tantque d*d<=n faire
    si irem(n,d)=0 alors return false; fsi;
    d++;
  ftantque;
  return true;
ffonction:;
```

```
   estpremier(1234567); isprime(1234567);
```

$$faux, faux$$

Remarque : cet algorithme est qualifié de naïf car il ne permet pas de tester la primalité de grands entiers.

9.1.4 Factorisation d'entiers

Au lieu de s'arrêter et d'indiquer que le nombre n'est pas premier lorsqu'on a une division exacte par d, on ajoute d à la liste des facteurs et on divise n par d. Lorsqu'on sort de la boucle extérieure, n vaut 1 ou est premier, il ne faut pas oublier de l'ajouter à la liste des facteurs.

```
fonction facto(n)
  local l,d;
  l:=[]; d:=2;
  tantque d*d<=n faire
    tantque irem(n,d)=0 faire
      l:=append(l,d);
      n:=n/d;
    ftantque;
    d++;
  ftantque;
  si n>1 alors l:=append(l,n); fsi;
  return l;
ffonction:;
```

```
facto(1234567); time(facto(1234567))
```

$$\begin{pmatrix} 127 & 9721 \\ 0.000425 & 0.0003800064125 \end{pmatrix}$$

```
ifactor(1234567); time(ifactor(1234567))
```

$$127 \cdot 9721, [2.8e-05, 2.36098496e-05]$$

```
facto(1234577); time(facto(1234577))
```

$$[1234577], [0.0028, 0.0023730052]$$

```
ifactor(1234577); time(ifactor(1234577))
```

$$1234577, [2.8e-05, 2.41935466e-05]$$

Exercice :
Comparer l'efficacité de l'algorithme précédent avec l'algorithme du document eduscol et compter le nombre d'itérations lorsque n est premier dans chacun de ces 2 programmes.

```
fonction facto1(n)
  local l,d;
  l:=[]; d:=2;
  tantque n>1 faire
    si irem(n,d)=0 alors
      tantque irem(n,d)=0 faire
        l:=append(l,d);
        n:=n/d;
```

```
      ftantque;
    fsi;
    d++;
  ftantque;
  return l;
ffonction:;
```

```
facto1(1234567); time(facto1(1234567))
```

$$\begin{pmatrix} 127 & 9721 \\ 0.02 & 0.0168335585 \end{pmatrix}$$

```
facto1(1234577); time(facto1(1234577))
```

$$[1234577], [2.26, 1.940780903]$$

Remarque : ce type d'algorithme est qualifié de naïf, car il ne permet pas de factoriser des nombres ayant de grands facteurs premiers.

9.2 Longueur d'un arc de courbe

On cherche la longueur approchée d'un arc de courbe d'équation $y = f(x)$ entre les points d'abscisses a et b. Pour cela on approche l'arc de courbe par une ligne polygonale composée de n petits segments reliant les points de la courbe d'abscisses $a + kh$ et $a + (k + 1)h$ pour $k \in [0, n - 1]$ et $h = \frac{b-a}{n}$.

```
fonction longueurcourbe(f,a,b,n)
  local l,k,h;
  h:=evalf((b-a)/n);
  l:=0;
  pour k de 0 jusque n-1 faire
    l:=l+distance(point(a+k*h,f(a+k*h)),point(a+(k+1)*h,f(a+(k+1)*h)));
  fpour;
  return l;
ffonction:;
```

```
f(x):=sqrt(1-x^2); longueurcourbe(f,0,1,100)
```

```
(x)->sqrt(1-x^2)                                    ,1.57064916467
```

Exercice :
Cette fonction `longueurcourbe` est relativement inefficace, car on calcule

plusieurs fois les mêmes quantités. Utilisez des variables locales pour la rendre plus efficace.

Solution :

```
fonction longueurcourbe(f,a,b,n)
  local l,k,h,x,M,N;
  h:=evalf((b-a)/n);
  l:=0;
  x:=a;
  M:=point(x,f(x));
  pour k de 0 jusque n-1 faire
    x:=x+h;
    N:=point(x,f(x));
    l:=l+distance(M,N);
    M:=N;
  fpour;
  return l;
ffonction:;
```

```
   f(x):=sqrt(1-x^2); longueurcourbe(f,0,1,100)
```

```
(x)->sqrt(1-x^2)                                    , 1.57064881543
```

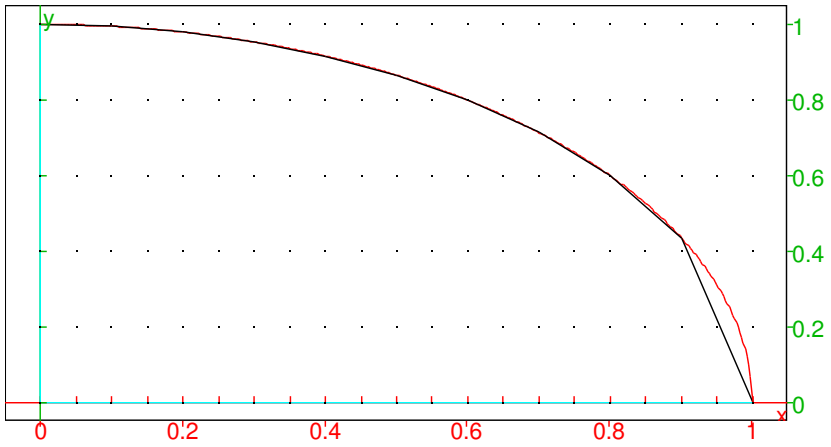
Remarque : h est approché, donc en effectuant la boucle on va accumuler des erreurs d'arrondis. Si l'erreur est faite par excès, la dernière valeur de x risque d'être un peu supérieure à 1, et $f(x)$ devient alors complexe, avec une partie imaginaire très petite. Ceci n'a pas d'importance dans `Xcas`, car l'instruction `point(x,y)` renvoie le point d'affixe $x + iy$. Dans d'autres langages, par exemple Python, cela peut provoquer une erreur.

Exercice :
Faire afficher la ligne polygonale dont on calcule la longueur.

Solution :

```
fonction approxcourbe(f,a,b,n)
  local L,k,h,x,M,N;
  h:=evalf((b-a)/n);
  x:=a;
  M:=point(x,f(x));
  L:=[M];
  pour k de 0 jusque n-1 faire
    x:=x+h;
    N:=point(x,f(x));
```

```
    L:=append(L,N);
    M:=N;
  fpour;
  return polygone_ouvert(L);
ffonction:;

f(x):=sqrt(1-x^2); plot(f(x),x=0..1,affichage=rouge
);approxcourbe(f,0,1,10)
```



9.3 Réfraction et recherche de minimas

On se donne une courbe d'équation $y = f(x)$ séparant deux milieux d'indices n_1 et n_2 et on cherche le chemin lumineux reliant deux points $A(x_A, y_A)$ et $B(x_B, y_B)$ situés de part et d'autre de la courbe, il s'agit de deux segments de droite AM et BM où M est le point de la courbe tel que la somme $n_1 AM + n_2 BM$ est minimal. Comme M a pour coordonnées $(x, f(x))$, on peut calculer cette somme en fonction de x

$$g(x) = n_1\sqrt{(x - x_A)^2 + (f(x) - y_A)^2} + n_2\sqrt{(x - x_B)^2 + (f(x) - y_B)^2}$$

Il s'agit donc de déterminer le minimum de la fonction g (en supposant connue la fonction f).

9.3.1 Minimum d'une fonction

On commence par la recherche approchée du minimum d'une fonction g. On se donne un point de départ x et un pas h, on regarde si on diminue la valeur de g en se déplaçant vers la gauche. Si c'est le cas, on continue à se

déplacer vers la gauche jusqu'à ce que g ne diminue plus, on a alors atteint un minimum local relativement à la discrétisation choisie. Si ce n'est pas le cas, on essaie de se déplacer vers la droite. [2]

```
fonction minih(g,x,h)
  // recherche d'un minimum local de g a partir de x en se deplacant
  // par pas de h vers la gauche ou vers la droite
  si g(x-h)<g(x) alors
    tantque g(x-h)<g(x) faire x:=x-h; ftantque;
  sinon
    tantque g(x+h)<g(x) faire x:=x+h; ftantque;
  fsi;
  retourne x;
ffonction:;
```

Exemple :
Recherche du minimum de la fonction $(x^2-2)^2$ en partant de 2.0 (prendre le carré d'une expression, ici x^2-2, et en chercher le minimum est une méthode de recherche d'une solution approchée)

```
g(x):=(x^2-2)^2:; minih(g,1.0,0.1)
```

Done, 1.4

Attention ! ce programme peut tourner en boucle indéfiniment si g décroit indéfiniment. Un programme plus robuste fixera un nombre maximal d'itérations, par exemple ici 1000 par défaut.

```
fonction minih(g,x,h,N=1000)
  local j;
  si g(x-h)<g(x) alors
    pour j de 1 jusque N faire
      si g(x-h)>g(x) alors return x; fsi;
      x:=x-h;
    fpour;
  sinon
    pour j de 1 jusque N faire
      si g(x+h)>g(x) alors return x; fsi;
      x:=x+h;
    fpour;
  fsi;
```

2. Cette méthode se généralise en dimension plus grande, on cherche la direction de plus grande pente et on se déplace dans cette direction d'un pas constant

```
  retourne "non trouve";
ffonction:;
```

Exercice :
Améliorer ce programme pour calculer g une seule fois par itération.

Solution :
Pour avoir une valeur de x avec une précision fixée à l'avance, on commence avec un pas de h (par défaut 0.1) et on divise le pas par 10.

```
fonction mini(g,x,h=0.1,eps=1e-10)
  // recherche d'un minimum local avec une precision de eps
  tantque h>eps faire
    x:=minih(g,x,h);
    h:=h/10;
  ftantque;
  retourne minih(g,x,h);
ffonction:;
```

On effectue un dernier appel à `minih` pour assurer que la précision est meilleure que `1e-10`.

```
  g(x):=(x^2-2)^2:; mini(g,1.0)
```

Done, 1.4142135624

Exercice :
expliquer pourquoi il est plus rapide d'appeler `mini(g,1.0,0.1,1e-10)` que `minih(g,1.0,1e-10)`.

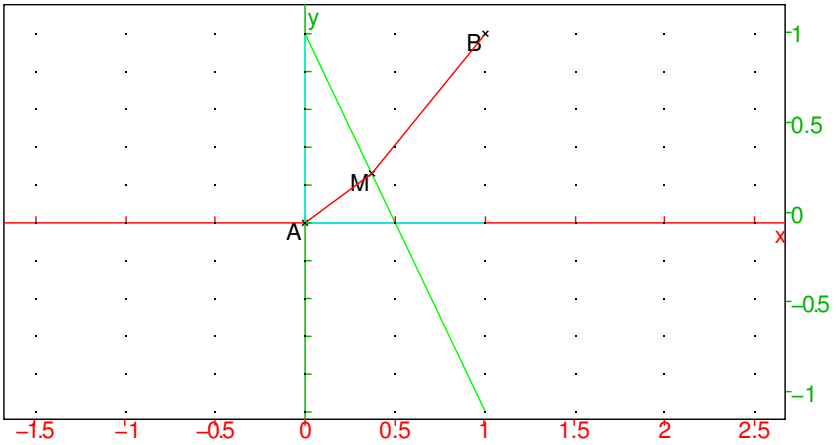
9.3.2 Application à la réfraction

Appliquons maintenant au problème de la réfraction. La fonction `refraction` prend en arguments la fonction f définissant la courbe séparatrice, les points A et B et les indices de réfraction, elle renvoie le point M.

```
fonction refraction(f,A,B,n1,n2)
  local g,x,tst;
  si f(abscisse(A))<ordonnee(A) ou f(abscisse(B))>ordonnee(B) alors
    return "A et B sont du meme cote du graphe!";
  fsi;
  g(x):=n1*distance(A,point(x,f(x)))+n2*distance(B,point(x,f(x)));
  x:=mini(g,abscisse(milieu(A,B)));
  return point(x,f(x));
ffonction:;
```

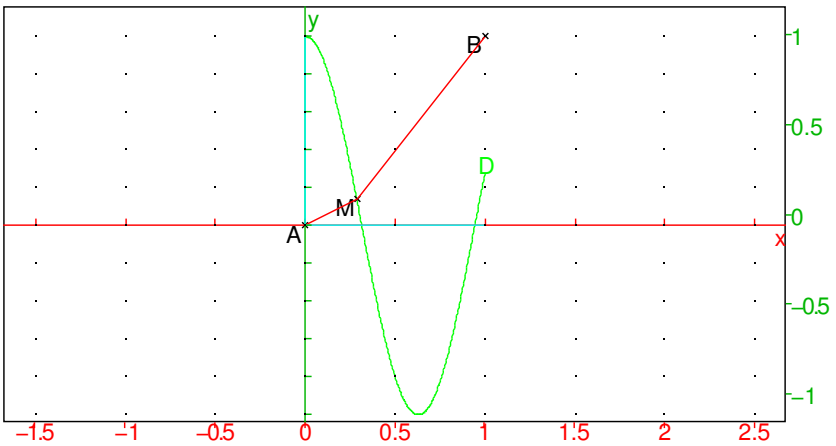
Illustration avec une fonction linéaire $1-2x$, la séparatrice est la droite D d'équation $y = 1-2x$

```
f(x):=1-2x;gl_ortho=true;D:=plot(f(x),x=0..1,couleur=vert
);A:=point(0,0); B:=point(1,1); M:=refraction
(f,A,B,2.5,1);segment(A,M,couleur=rouge);
 segment(M,B,couleur=rouge)
```



Pour tester avec une courbe non linéaire, par exemple avec la fonction $f(x) = \cos(5x)$, il suffit de modifier la définition de f :

```
f(x):=cos(5x);gl_ortho=true;D:=plot(f(x),x=0..1,couleur=vert
);A:=point(0,0); B:=point(1,1); M:=refraction
(f,A,B,2.5,1);segment(A,M,couleur=rouge);
 segment(M,B,couleur=rouge)
```



9.3.3 Discussion

L'application à la réfraction est certainement trop difficile pour des élèves débutants, on définit en effet ici une fonctionnelle (fonction dont un argument est une fonction), et on définit une fonction dans une autre fonction (la fonction g). De plus la définition de g ci-dessus n'est pas recommandée, car elle dépend de la valeur de paramètres n_1, n_2, A et B qui ne sont pas passés en argument (g n'est pas une fonction pure, i.e. indépendante du contexte). Pour rendre g indépendante du contexte, il faut remplacer dans son expression `n1*distance(A,M)+n2*distance(B,M)`
n_1, n_2, A, B par leurs valeurs. Dans `Xcas`, on peut le faire en utilisant le moteur de calcul formel. On utilise une variable symbolique `x` pour calculer l'expression de g, c'est le rôle de la commande `purge(x)` ci-dessous, et on utilise la commande `unapply` qui permet de définir une fonction à partir d'une expression.

```
fonction refraction(f,A,B,n1,n2)
  local g,M,x,tst,xM;
  tst:=(f(abscisse(A))-ordonnee(A))*(f(abscisse(B))-ordonnee(B));
  si tst>=0 alors return "A et B sont du meme cote du graphe!"; fsi;
  purge(x);
  M:=point(x,f(x));
  g:=unapply(n1*distance(A,M)+n2*distance(B,M),x);
  xM:=mini(g,abscisse(milieu(A,B)));
  return point(xM,f(xM));
ffonction:;
```

On a aussi amélioré le test que les deux points A et B sont bien de part et d'autre de la courbe en introduisant une variable locale `tst` qui doit être négative.

Dans des langages non formels, pour faire les choses proprement, il faut passer explicitement les paramètres `A,B,n1,n2` à la fonction `g`, en les regroupant (dans une liste dans un langage de script, par un pointeur vers une structure en C). Le prototype de la fonction `g` est alors `g(x,params)`, celui des fonctions `mini` est `mini(g,param,x,h,eps)`. Mais ceci complique bien sur la présentation au niveau du lycée.

9.3.4 Autre méthode de recherche de minimum

On suppose ici qu'on cherche le minimum d'une fonction g sur un intervalle $[a, b]$ avec g d'abord décroissante puis croissante. On suppose donc

qu'on a un triplet a, c, b tel que $g(c)$ est plus petit que $g(a)$ et $g(b)$

$$a < c < b, \quad g(c) < g(a), g(c) < g(b)$$

On effectue ensuite une sorte de dichotomie. On se fixe une précision ϵ

1. On effectue une boucle tant que $b - a > \epsilon$
2. On calcule la valeur de g au milieu u de $[a, c]$. Si $g(u)$ est plus petit que $g(a)$ et $g(c)$ on peut utiliser le nouveau triplet a, u, c, on passe à l'itération suivante.
3. Sinon, on calcule la valeur de g au milieu v de $[c, b]$. Si $g(v)$ est plus petit que $g(c)$ et $g(b)$ on peut utiliser le nouveau triplet c, v, b, on passe à l'itération suivante.
4. Sinon, $g(u)$ est plus grand que $g(c)$ ou que $g(a)$ donc est plus grand que $g(c)$ puisque $g(c) < g(a)$, et de même $g(v)$ est plus grand que $g(c)$, donc on utilise le nouveau triplet u, c, v et on passe à l'itération suivante.

Le programme se termine (En effet, si on applique la troisième règle, $b - a$ est divisé par 2. Si on applique uniquement la 1ère et la 2ème règle, si on applique deux fois de suite la première règle b est remplacé par c puis par le u précédent donc on gagne un facteur 2, de même pour la deuxième règle, si on applique la première puis la deuxième b est remplacé par c et a par u donc on gagne aussi un facteur 2).

```
fonction mini2(g,a,b,c,eps=1e-10)
  local u,v;
  si c<=a ou c>=b alors return "erreur"; fsi;
  si g(c)>=g(a) ou g(c)>=g(b) alors return "erreur"; fsi;
  tantque b-a>eps faire
    u:=(a+c)/2;
    si g(u)<g(a) et g(u)<g(c) alors a,c,b:=a,u,c; continue; fsi;
    v:=(b+c)/2;
    si g(v)<g(c) et g(v)<g(b) alors a,c,b:=c,v,b; continue; fsi;
    a,c,b:=u,c,v;
  ftantque;
  return a,b,c;
ffonction:;
```

```
 mini2(cos,1.0,6.0,3.0)
```

$$3.14159265156, 3.14159265165, 3.14159265161$$

Exercice :
Rendre ce programme plus efficace en calculant le moins de fois possible les images par g de a, b, c, u, v.

9.4 Solveur de triangle

On cherche à construire un triangle ABC dont les cotés sont donnés par $a = BC, b = AC, c = AB$ où a, b, c vérifient les inégalités triangulaires

$$a < b + c, \quad b < a + c, \quad c < a + b$$

On peut par translation se ramener au cas où A est l'origine, puis par rotation à $B(c, 0)$, il s'agit donc de déterminer les coordonnées (x, y) de C.

```
c:=3; x:=1; y:=2;
A:=point(0,0,legende="A(0,0)");
B:=point(c,0,legende="B(c,0)");
C:=point(x,y,legende="C(x,y)");
triangle(A,B,C);
```

On a deux équations

$$a^2 = BC^2 = (x - c)^2 + y^2, \quad b^2 = AC^2 = x^2 + y^2$$

En en faisant la différence, on obtient

$$a^2 - b^2 = (x - c)^2 - x^2 = -2cx + c^2$$

d'où l'on tire la valeur de x

$$x = \frac{c^2 - (a^2 - b^2)}{2c}$$

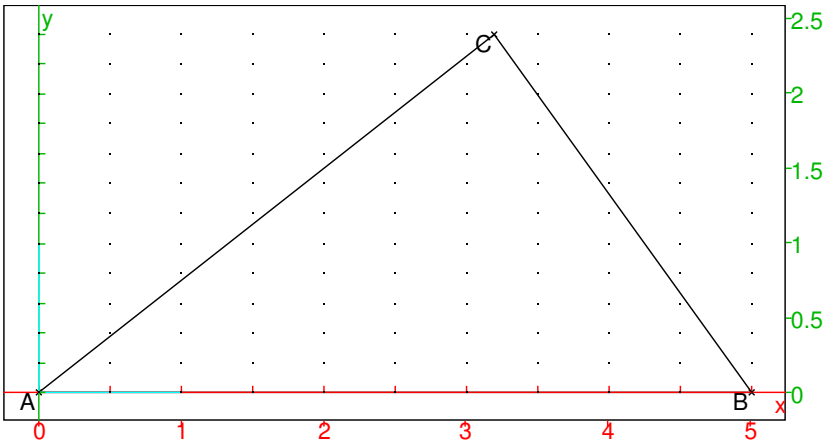
puis la deuxième équation nous donne deux valeurs pour y (correspondant à deux triangles symétriques)

$$y = \pm\sqrt{b^2 - x^2}$$

Remarque :
Puisque a, b, c vérifient les inégalités triangulaires $a < b+c, \quad b < a+c, \quad c < a + b$ si $x = \frac{c^2-(a^2-b^2)}{2c}$ alors $b^2 - x^2 \geq 0$, en effet :
$4c^2x^2 = (b^2 + c^2 - a^2)^2$ donc
$4c^2(b^2 - x^2) = (2cb)^2 - (b^2 + c^2 - a^2)^2 = (2cb + b^2 + c^2 - a^2)(2cb - b^2 - c^2 + a^2)$.
Donc $b^2 - x^2$ a le même signe que $((b + c)^2 - a^2)(a^2 - (b - c)^2)$. Ce signe est positif puisque d'après les inégalités triangulaires on a :
$0 < a < b + c$ donc $a^2 < (b + c)^2$ et
$b - c < a$ et $c - b < a$ donc $|b - c| < a$ donc $(b - c)^2 < a^2$.

```
fonction trisolve(a,b,c)
  local x,y;
  x:=(c^2+b^2-a^2)/(2*c);
  y:=sqrt(b^2-x^2);
  return point(0,0),point(c,0),point(x,y);
ffonction:;
```

```
A,B,C:=trisolve(3,4,5); triangle(A,B,C)
```



Remarque :
Le document ressource Python eduscol utilise une méthode de résolution approchée pour déterminer le triangle. Cela nous semble peu adapté, car la résolution exacte est accessible au niveau lycée et est beaucoup plus simple.

Chapitre 10

Aide

10.1 Les fonctions usuelles avec Xcas

`abs` : valeur absolue ou le module de l'argument.
`ceil` : renvoie le plus petit entier ¿= à l'argument.
`cos` : renvoie le cosinus de l'argument.
`Digits` : pseudo-variable pour modifier le nombre `n` de chiffres significatifs (par ex `Digits:=n`).
`evalf` : évaluation numérique du premier argument (le nombre de digits peut être donné comme second argument).
`floor` : renvoie la partie entière de l'argument (le plus grand entier ¡= à l'argument.
`frac` : partie fractionnaire de l'argument.
`max` : renvoie le maximum des éléments d'une séquence ou d'une liste de réels.
`min` : renvoie le minimum des éléments d'une séquence ou d'une liste de réels.
`+` : renvoie la concaténation de 2 chaînes ou addition terme à terme de 2 expressions ou 2 listes (opérateur infixé).
`round` : renvoie l'argument réel arrondi à l'entier (ou le décimal) le plus proche.
`sin` : renvoie le sinus de l'argument.
`sign` : renvoie le signe (-1,0,+1) de l'argument.
`sqrt` : renvoie la racine carrée de l'argument.
`tan` : renvoie la tangente de l'argument.

10.2 Les fonctions Xcas de calcul formel utilisées

`coeff(P,var,[n]` : renvoie la liste des coefficients d'un polynôme `P` par rapport à la variable `Var` ou le coefficient de degré `n`.
`count(f,L)` : applique la fonction `f` aux éléments de la liste `L` et en renvoie la somme.
`degree(P,var` : renvoie le degré du polynôme `P` par rapport à la variable `var`.
`dim` : renvoie la longueur d'une liste, d'une séquence ou d'une chaîne de caractères.
`droit` : renvoie le côté droit d'une équation, d'un intervalle, d'une liste ou d'une chaîne .
`fsolve` ou `resoudre_numerique` : renvoie la solution numérique d'une équation.
`gauche` : renvoie le côté gauche d'une équation, d'un intervalle, d'une liste ou d'une chaîne.
`iquo` : renvoie le quotient euclidien de 2 entiers.
`iquorem` : renvoie la liste du quotient et du reste euclidien de 2 entiers.
`irem` : renvoie le reste euclidien de 2 entiers.
`normal` : renvoie une simplification de l'argument.
`purge(a)` : enlève la valeur stockée dans la variable `a`.
`simplify` ou `simplifier` : renvoie une simplification de l'argument.
`solve` ou `resoudre` : renvoie la solution d'une équation.
`sommets` : renvoie la liste des sommets d'un polygone.
`subst` ou `substituer` : remplace dans une expression, une variable non affectée par une valeur.
`sum` ou `somme` : somme des éléments d'une liste (ou séquence).
`subtype` : renvoie 1 pour une séquence, 2 pour un ensemble, 10 pour un polynôme et 0 sinon ce qui définit le sous-type de l'argument.
`type` : renvoie n dans [1..12] définissant le type de l'argument.

10.3 Les fonctions Xcas de géométrie utilisées

`affichage` : employé comme option de commandes graphiques permet de faire des dessins en couleur (par ex `A:=point([1,1],affichage=rouge)`).
`carre` : renvoie et dessine le carré direct donné par 2 points.
`centre` : renvoie et dessine le centre du cercle donné en argument.
`cercle` : renvoie et dessine le cercle donné par 1 point et 1 réel (son centre et son rayon) ou par 2 points (son diamètre) ou par son équation.
`demi_droite` : renvoie et dessine la demi-droite donnée par son origine et 1 point ou par son origine et son vecteur directeur [1,m].

`droite` : renvoie et dessine la droite donnée par 2 points ou par 1 point et son vecteur directeur [1,m].
`est_aligne` : renvoie 1 si les points sont alignés, 2 si les points sont confondus et 0 sinon.
`est_parallele` : renvoie 1 si 2 droites sont parallèles et 0 sinon.
`equation` : renvoie l'équation de l'argument.
`inter` dessine et renvoie la liste des points d'intersection de 2 objets géométriques.
`inter_unique` renvoie et dessine un des points d'intersection de 2 objets géométriques.
`longueur` : renvoie la longueur du segment défini par 2 points ou par les coordonnées de ces points. `norm` : renvoie la norme l2 d'un vecteur :
`norm`$([x_1,..x_n])$: renvoie $\sqrt{(}x_1^2+...x_n^2)$
`point` : renvoie et dessine le point de coordonnées l'argument.
`polygone` : renvoie et dessine le polygone donné par une liste de points.
`polygone_ouvert` : renvoie et dessine la ligne polygonale donnée par une liste de points.
`rayon` :renvoie le rayon du cercle donné en argument.
`segment` : renvoie et dessine soit le segment donné par 2 points ou le vecteur donné par un point et son vecteur directeur. **Bien voir** la différence entre `segment(point(cA),point(cB))` (c'est le segment (A,B)) et `segment(cA,cB)` qui est aussi `segment(point(cA),point(cA+cB))`.
`sommets` : renvoie la liste des sommets du polygone donné en argument et les dessine.
`translation(B-A,C)` : renvoie et dessine le translaté de `C` dans la translation de vecteur `AB`.
`triangle` : renvoie et dessine le triangle donné par 3 points.
`vecteur` : renvoie et dessine le vecteur donné par 2 points ou par un point et son vecteur directeur. **Bien voir** la différence entre `vecteur(point(cA),point(cB))`(c'est le vecteur (A,B)) et `vecteur(cA,cB)` qui est aussi `vecteur(point(cA),point(cA+cB))`.

10.4 Les fonctions Xcas de programmation utilisées

`break` : pour interrompre une boucle (tantque ¡cond¿ faire inst1 ; si ¡cond¿ alors inst2 ;break ;fsi ;ftantque ;).
`fpour`
`fsi`
`ftantque`
`jusque`
`local`
`pas`

```
pour <k> de <k1> jusque <k2> faire <instructions> pas <p> fpour
quand
retourne
si <condition> alors <instructions1> sinon <instructions2> fsi
tantque <condition> faire <instructions> ftantque
```

10.5 Les fonctions Xcas utilisées en statistiques

`alea(n1,n2)` : renvoie un réel aléatoire de [n1,n2[.
`alea(n)` : renvoie un entier aléatoire entre 0 et `n-1`.
`binomial` : peut être employé comme option de la commande `randvector`.
`binomial_cdf(n,p,x1,x2)` : renvoie Proba(x1¡=X¡=x2) quand X suit la loi binomiale B(n,p).
`mean` : renvoie la moyenne d'une liste ou d'une liste pondérée par le deuxième argument.
`median` : renvoie la médiane d'une liste ou d'une liste pondérée par le deuxième argument.
`plotlist` : lorsque l'argument est $L = [y_0, ..., y_n]$, trace la ligne polygonale reliant les points d'abscisse k et d'ordonnée y_k pour $k = 0..n$.
`randvector` : renvoie une liste de nombres aléatoires de taille n constituée d'entiers aléatoires distribués selon la loi indiquée.
`stddev` : renvoie l'écart-type d'une liste ou d'une liste pondérée par le deuxième argument.

Annexe A

Premiers pas avec les interfaces de Xcas

A.1 Les différentes intefaces

Vous pouvez utiliser Giac/Xcas avec plusieurs interfaces différentes :

- Xcas natif sur PC Windows ou Linux et sur Mac
  Pour cela il faudra installer Xcas sur votre machine en suivant les instructions de la sous section A.3.
- Xcas depuis votre navigateur (Firefox recommandé)
  Pour cela il suffira de se connecter une seule foisà Internet pour pouvoir ensuite travailler dans Firefox en suivant les instructions de la sous section A.2.
- depuis un interpréteur Python avec le module giacpy, cf. la section A.5
- En écrivant des documents interactifs comme celui-ci en LaTeX [1].

A vous de choisir !

A.2 Avec Xcas pour Firefox

Depuis votre navigateur (il est conseillé d'utiliser Firefox pour des performances optimales) ouvrez le lien
http ://www-fourier.ujf-grenoble.fr/˜ parisse/xcasfr.html
Lors de la première utilisation, vous êtes invités à choisir entre la compatibilité de syntaxe Python ou la syntaxe Xcas native. Ceci peut être modifié ultérieurement en cliquant sur le bouton Config en haut à gauche.

1. https://www-fourier.ujf-grenoble.fr/~parisse/tice.html

La ligne de commande se trouve en bas de page, au-dessus se trouve une ligne de boutons assistants

- 123 permet d'afficher un clavier scientifique
- ? affiche une aide sur la commande ou le début de commande à l'endroit du curseur
- deux flèches de direction pour déplacer le curseur sur les smartphones ou tablettes
- sel pour sélectionner la ligne de commande
- math qui affiche des assistants mathématique
- prog qui affiche des assistants de programmation
  - test : assistant pour créer un test (si alors sinon finsi)
  - boucle : assistant pour créer une boucle (tantque ou pour)
  - fonct : assistant pour créer une fonction
  - debug : insère la commande debug(pour mettre au point un programme
  - // ou # pour commencer un commentaire sur la ligne actuelle
  - \n pour passer à la ligne
  - del pour effacer un caractère
  - ->/ pour indenter le source
- ok (ou touche Entree du clavier) pour valider la ligne de commande. Pour passer à la ligne sur un clavier physique on peut taper simultanément sur shift et Entree. La ligne de commande est alors évaluée et le résultat affiché en face ou en-dessous. Il est possible de modifier une ligne de commande existante et de la réexécuter.

Vous pouvez sauvegarder une session, vous pouvez aussi l'envoyer par mail. Le lien Clone permet de

- de cloner une session (par exemple en cas de crash)
- en utilisant clic droit, de recopier le lien correspondant sur une page web, on peut ainsi indiquer le lien vers un début de session à des élèves.
- toujours en utilisant clic droit, de recopier la session vers Xcas natif. Dans Xcas natif sur PC, cliquer sur le numéro de niveau où vous voulez insérer la session de Xcas pour Firefox (par exemple sur 1 pour le niveau 1) puis faites la copie en tapant simultanément sur Ctrl et V.

Pour plus de détails, vous pouvez cliquer sur le bouton Tutoriel en haut à gauche en-dessous du bouton Config.

A.3 Xcas natif sur PC Windows et Linux et sur Mac.

Pour installer Xcas suivez les instructions depuis :
http ://www-fourier.ujf-grenoble.fr/˜ parisse/install_fr.html

Vous pouvez ensuite taper des commandes dans les lignes de commande en utilisant les menus, en particulier le menu Outils qui contient les commandes mathématiques de calcul les plus utilisées, à défaut le menu Cmd qui les contient classées par thèmes, le menu Graph pour les commandes de type graphes de fonction, le menu Geo pour la géométrie, pour plus de détails voir le menu Aide, Débuter en calcul formel, Tutoriel.

Pour programmer, il est plus confortable d'ouvrir un niveau de programmes depuis le menu Prg, nouveau programme. Un assistant de création de fonction s'ouvre immédiatement (vous pouvez l'ignorer en tapant sur la touche Echap). Un niveau de programme apparait avec son propre menu et des boutons assistants Test, Boucle et Fonction pour vous aider à saisir les structures de base de l'algorithmique. Une fois votre programme saisi, vous l'interprétez en tapant sur le bouton OK (raccourci clavier F9). S'il est correct, vous pouvez tester votre fonction depuis une ligne de commande.

Vous pouvez cloner une session simple vers Xcas pour Firefox à partir du menu Fich, Clone, Online.

A.4 Conseils pour adapter des scripts Python.

De nombreuses ressources Python d'algorithmique lycée n'utilisent pas de sortie graphique et fonctionneront sans changement avec `Xcas`. Si ce n'est pas le cas :

- Vérifiez que les délimiteurs de chaines de caractères sont bien des `"` et non des `'`, si nécessaire remplacez-les (N.B. si une chaine délimitée par des `'` contient des `"`, remplacez-les par `\"`).
- Les commandes agissant sur les listes ou les chaines de caractères sont presque toutes compatibles. Attention toutefois `L:=[1,2];5*L` renvoie `[5,10]` en `Xcas` (utiliser `flatten(seq(L,5))`), et `L+l` renvoie `[2,4]` (utiliser `concat(L,l)`)
- Les commmandes d'`import` sont ignorées par Xcas, sauf la commande `from turtle import *` qui affecte dans des variables de nom les raccourcis de commandes en anglais leurs équivalents français. Mais il est sans doute plus simple d'utiliser directement les commandes en français !

– Si le script utilise les modules math, cmath, random, il n'est pas nécessaire de les importer, les commandes sont déjà connues dans `Xcas`, vous pouvez donc simplifier le script.

Pour les scripts utilisant des commandes graphiques du module matplotlib, il faudra utiliser à la place des commandes graphiques de Xcas que vous trouverez depuis le menu `Graphe` ou `Geo`, par exemple :

– `point(x,y)` trace le point de coordonnées (x, y)
– `droite(A,B)` ou `droite(equation)` trace une droite, `segment(A,B)` un segment, avec A et B 2 points (ou deux affixes de point)
– `plotlist(l)` affiche une ligne polygonale reliant les points d'abscisse 0, 1, ... et d'ordonnées les éléments de la liste l, `plotlist(X,Y)` relie les points d'abscisses de la liste X et d'ordonnée de la liste Y, `polygonscatterplot(X,Y)` fait de même et marque les points
– `histogram` et `bar_plot` servent à faire des graphes statistiques
– `plot` trace le graphe d'une fonction
– ...

A.5 Python et giacpy

Une partie des programmes en syntaxe Python de ce document peuvent s'exécuter dans un interpréteur Python, après avoir chargé le module `giacpy`. Il faut d'abord installer ce module sur votre PC, en suivant les instructions ici :
https ://www-fourier.ujf-grenoble.fr/˜ parisse/giac_fr.html#python
Ensuite le chargement se fait par la ligne de commande

```
from giacpy import *
```

Les commandes de `Xcas` en anglais sont en général accessibles depuis l'interpréteur Python, mais il faut parfois faire des ajustements :

– Pour avoir une variable symbolique, par exemple `x`, on écrira `x=giac('x')`
– Pour afficher un objet géométrique ou un graphe de fonction, on utilise la méthode `qcas()`, par exemple `c=circle(0,1)` puis `c.qcas()`
– Il ne faut pas oublier que la puissance est `**` et les règles propres à Python pour la division d'entiers

Annexe B

Xcas, Python et Javascript.

B.1 Le mode de compatibilité Python de Xcas

Xcas accepte une syntaxe compatible avec le langage `Python`. La principale différence entre la programmation en français et Python, c'est qu'il n'y a pas de fin de bloc explicite en Python, c'est l'indentation qui joue ce rôle. L'indentation consiste à ajouter au début de chaque ligne d'un même bloc le même nombre d'espaces. Il est recommandé d'indenter les programmes en mode Xcas, en mode Python c'est obligatoire. Il faut donc faire attention :

- lorsqu'on copie un programme `Python` avec un copier-coller : cela risque de modifier l'indentation,
- lorsqu'on écrit un texte contenant un programme `Python` de faire en sorte que ce programme apparaisse sur une seule page (ce qui n'est pas toujours possible pour un programme contenant beaucoup d'instructions !).

Attention, le mode de compatibilité Python de Xcas n'est pas identique à Python. Seules les constructions de base sont acceptées (pas de programmation orientée objet) :

- `def` pour définir une fonction,
- `if` pour un test,
- `for` pour une boucle pour,
- `while` pour une boucle tantque.

Certaines commandes Xcas sont compatibles avec Python, par exemple `range`, `assert`, `random`, etc. Notez que :

- il est recommandé de déclarer explicitement les variables locales (avec `# local <nom-var>`).
- les chaînes de caractères sont délimitées par `"` car dans `Xcas` on réserve `'` pour quoter un argument c'est à dire pour ne pas évaluer un argument,

par exemple :
`"a"` désigne la lettre `a` (c'est une chaîne de 1 caractère), et si `a:=2+2` (ou en Python `a=2+2`) `a` renvoie `4` alors que `'a'` désigne la variable `a` non évaluée et donc `'a'` renvoie le nom `a` de la variable `a` (ce n'est pas une chaîne de 1 caractère, c'est un identificateur).
On aura par exemple :
`"a"+3` renvoie `"a3"` mais
si `a:=4` alors `'a'+3` renvoie `3+a` et `a+3` renvoie `7`
`type("a")` renvoie `string` (chaîne de caractères)
`type(a)` renvoie `integer` (`a` contient un entier)
`type('a')` renvoie `identifier` (`'a'` est un nom de variable).

- les commandes graphiques des librairies Python ne sont pas implentées, il faut utiliser les commandes de Xcas (par exemple `plot`, `plotlist`, `histogram`, ...)
- les chaines de caractères peuvent être modifiées caractère par caractère, comme les listes.
- la taille des listes s'ajuste automatiquement lors d'une affectation à un indice au-delà de la taille actuelle de la liste
- Si on écrit un programme dans lequel l'utilisateur doit saisir des données, on utilise soit la commande `saisir(a)` (ou `input(a)`) pour saisir une expression, soit `saisir_chaine(s)` (ou `textinput(s)`) pour saisir une chaîne de caractères, alors que'en Python `input` renvoie une chaîne de caractères qu'il faut affecter dans une variable
- il n'y a pas besoin d'importer de bibliothèques (import math, import random, etc...),

Exemple : On veut faire un programme qui teste si un nombre est parfait : un entier naturel n est parfait si il est égal à la somme de ses diviseurs (1 compris, n non compris).

On écrit la fonction `Parfait` qui a comme argument un entier `n`.
On définit les variables locales à la fonction :

- `s` qui recevra la somme des diviseurs de `n`. au début `s` est nul.
- `j` qui parcourt les diviseurs potentiels,

Pour tous les entiers `j` entre `1` et `n-1` on teste si `j` divise `n` et pour cela on cherche le reste de la division de `n` par `j`, c'est `irem(n,j)` ou `n%j` qui renvoie ce reste. Si le reste est nul, c'est que `j` divise `n` donc on ajoute `j` à `s`. Quand `j` a parcouru tous les entiers jusque `n-1` (i.e. quand la boucle `pour` est terminée), on teste si `s` est égal à `n`. Si il y a égalité alors `n` est parfait, sinon `n` n'est pas parfait.

Une implémentation en syntaxe Xcas en français :

```
fonction Parfait(n)
```

```
  local j,s;
  "Parfait(n) renvoie vrai si n est la somme de ses diviseurs";
  s:=0;
  pour j de 1 jusque n-1 faire
    si irem(n,j)==0 alors
      s:=s+j; // j divise n, on l'ajoute a la somme
    fsi;
  fpour;
  si s==n alors // on pourrait aussi recrire return s==n
    retourne vrai;
  sinon
    retourne faux;
  fsi;
ffonction:;

   Parfait(6)
```

vrai

L'indentation permet de visualiser facilement la forme du programme, ici on a 3 instructions : 1 affectation, 1 boucle `pour` qui contient une instruction (1 test si) et 1 test si/sinon. L'indentation sert aussi à contrôler qu'on n'a pas fait de faute de syntaxe (non-fermeture d'une parenthèse par exemple). En syntaxe compatible Python, le programme `Parfait` précédent s'écrit :

```
def Parfait(n):
    # local j,s
    '''Parfait(n) determine si l'entier n est la somme de ses diviseurs'''
    s=0
    for j in range(1,n):
        if n
            s=s+j # j divise n, on l'ajoute
    if s==n:
        return True
    else:
        return False

   Parfait(24)
```

faux

B.2 Xcas et Javascript

Javascript est le langage des navigateurs, il est donc extrêmement répandu et peut être utilisé sans installation sur PC/Mac. De ce fait, un programme écrit en javascript pourra facilement être intégré dans une petite interface utilisateur en HTML (voir l'exemple ci-dessous), et exécuté depuis n'importe quel autre navigateur, ce qui peut être assez motivant.
Du point de vue des types de variables utilisables, on dispose entre autres de nombres approchés, de chaines de caractères et de listes, il n'y a pas de type particulier pour les entiers. Les structures de controle ont une syntaxe proche du langage C, point-virgule en fin d'instruction, délimiteurs de blocs explicites `{}`, test en `if () { ... } else { ... }` et boucle en `for(init;condition;incrementation){...}` ou `while(condition){...}`, déclaration de variable par `var` et retour de fonction par `return`.
Si vous connaissez javascript, vous pouvez utiliser une syntaxe très proche dans `Xcas`. Exemple, syntaxe utilisable dans les deux langages pour le calcul du n-ième nombre de Fibonacci :

```
function fibo(n){
  var u,v,w,j;
  u=1; v=1;
  if (n<2) return 1;
  for (j=2;j<=n;j++){
    w=u+v; u=v; v=w; // ou en Xcas (u,v)=(v,u+v);
  }
  return v;
}
```

Pour avoir une interface HTML minimaliste pour cette fonction, on crée un fichier nommé disons `fibo.html` (que l'on consultera depuis un navigateur) contenant :

```
<script>
function fibo(n){
  var u,v,w,j;
  u=1; v=1;
  if (n<2) return 1;
  for (j=2;j<=n;j++){
    w=u+v; u=v; v=w;
  }
  return v;
}
```

```
</script>
Le
<textarea cols="5" rows="1" id="input"
onkeypress="if (event.keyCode!=13) return true;
document.getElementById('output').value=fibo(eval(value));
return false;">5</textarea>
i&egrave;me nombre de Fibonacci est
<textarea rows="1" id="output"></textarea>
```

On peut utiliser les commandes de `Xcas` depuis Javascript, c'est exactement ce que fait l'interface `Xcas` pour Firefox.

B.3 Xcas et Python

Python est un langage assez populaire dans les milieux scientifiques et est facile à installer. Il possède un type spécifique pour les entiers. Sa principale différence par rapport aux autres langages est que les structures de controle n'ont pas de délimiteur explicite de fin de bloc, c'est l'indentation qui joue ce role. L'obligation d'indenter correctement a certes des vertus pédagogiques, mais pas l'absence de délimiteur explicite de fin de bloc. Ceci rend aussi l'échange de code source Python avec des applications orientées texte (copier/coller, mail, forums, traitement de texte) plus fragile [1], ou même la lecture difficile si le code est affiché sur plus qu'une page. De plus, la déclaration de variables locales dans une fonction est implicite, ce qui n'est pas très pédagogique. D'autre part, la syntaxe de la boucle for en Python utilise des objets plus abstraits (itérateurs) ou qui peuvent être inutiles (liste d'entiers en Python 2), elle s'éloigne de celle de la grande majorité des autres langages, y compris la traduction en langage machine. Ainsi l'exemple de Fibonacci donne

```
def fibo(n):
    u=1
    v=1
    if n<2:
        return 1
    for j in range(2,n+1):
        w=u+v  # ou directement u,v=v,u+v
```

1. Ainsi, l'exercice de spécialité du sujet de bac S 2017 centres etrangers demande aux candidats d'interpréter un algorithme écrit en syntaxe Python sans délimiteurs de blocs explicites ... et l'un des blocs est mal indenté, très probablement au cours d'un passage par un logiciel de traitement de texte

```
        u=v
        v=w
    return v
```

B.4 Comparaison rapide

La principale différence entre les 3 langages, c'est que `Xcas` est un logiciel de calcul formel et permet de travailler avec des variables symboliques et des expressions. Dans `Xcas`, une variable globale peut ne pas avoir de valeur (son évaluation la renvoie), ce n'est pas le cas en Javascript ou en Python (l'évaluation d'une variable non initialisée renvoie une erreur). De ce fait, il ne faut pas oublier de déclarer les variables locales d'une fonction pour éviter des erreurs. En contrepartie pour réaliser des fonctions comme la dichotomie ou la recherche d'une valeur approchée d'intégrale par la méthode des rectangles, on peut passer une expression symbolique en paramètre et l'évaluer en un point, ce qui est conceptuellement plus simple que de passer une fonction en argument à une autre fonction. Avec `Xcas`, on peut aussi faire du calcul symbolique à l'intérieur d'une fonction, par exemple dériver une expression, calculer des valeurs exactes ou approchée, et dans ce dernier cas travailler avec un nombre arbitraire de décimales.

Des trois langages, Javascript est le plus rapide, puis Python puis `Xcas`.

Annexe C

Les biais des langages interprétés

Les ordinateurs n'exécutent pas directement les instructions données dans un langage comme `Xcas`, Javascript, Python, C, Pascal, etc., ils exécutent uniquement du code machine spécifique au micro-processeur de l'ordinateur. Il y a deux façons de traduire le langage en code machine : soit une fois pour toutes lors d'une étape appelée compilation, soit au moment de l'exécution de l'instruction du langage par l'interpréteur (qui est lui-même un programme écrit dans un langage compilé), on parle alors de langage interprété (en toute rigueur, il existe une troisième possibilité un peu intermédiaire qui consiste à compiler au moment où on exécute l'instruction du langage).

Les langages interprétés sont par nature plus faciles à mettre en oeuvre (pas de compilation) et de ce fait rendent l'apprentissage plus simple. Mais la traduction au moment de l'exécution a bien entendu un impact sur le temps d'exécution, parce que les boucles sont traduites autant de fois qu'elles sont exécutées (des optimisations peuvent y remédier plus ou moins, il est d'ailleurs fort possible que la syntaxe de la boucle `for` en Python soit liée à des considérations d'optimisation). De plus, les langages interprétés utilisent des variables dont le type est déterminé au moment de l'exécution du code, ce qui a un impact aussi bien pour la taille occupée par la variable que pour l'exécution d'une opération qui doit commencer par déterminer le type des arguments pour agir en conséquence. Enfin, lorsqu'on utilise un langage compilé, on a accès aux types de données directement manipulables par le microprocesseur, dont la place occupée en mémoire est optimale et le temps d'exécution très rapide, par exemple on peut multiplier deux entiers dont le produit est inférieur à 2^{63} en moins d'un milliardième de seconde.

Bien entendu, on peut améliorer le temps d'exécution avec un langage interprété si certaines taches répétitives sont codées dans une instruction

du langage, par exemple l'instruction effectuant le produit de deux matrices n'est pas programmé dans le langage interprété lui-même mais est compilé une fois pour toutes dans le code exécutable de l'interpréteur. Le programmeur qui souhaite avoir un code suffisamment rapide va donc adopter un style de programmation qui favorisera l'utilisation de commandes du langage effectuant en une seule instruction ce qui nécessiterait une ou plusieurs boucles imbriquées si on le programmait uniquement avec les instructions de base d'un langage compilé. C'est pour cette raison que les programmeurs expérimentés travaillant avec un langage interprété écrivent souvent dans un style "algébrique" avec le moins possible de structures de controle explicites et des types de données souvent assez complexes. Par exemple pour écrire un produit scalaire de deux listes en `Xcas`, on utilisera la commande `dot`, si elle n'existait pas, on pourrait écrire `sum(x[j]*y[j],j,0,size(x)-1)` pour éviter de faire une boucle `pour`, le chapitre statistiques 8 contient quelques exemples d'optimisation de ce type. Pour des commandes plus compliquées, on utilisera sans doute des commandes `seq` imbriquées au lieu de boucles. Mais ce style présente un risque, celui de calculer plusieurs fois la même quantité : une expression algébrique n'a pas de variables locales pour stocker de résultats intermédiaires. Il faut donc prendre garde au biais de ne pas calculer des quantités intermédiaires. Ce n'est pas le seul biais, il faut également faire attention à l'utilisation d'instructions faisant des boucles implicitement et à l'utilisation de types de données puissants qui risquent de masquer la complexité des opérations nécessaires et pourraient être remplacés par des types plus simples, par exemple utiliser un dictionnaire (ou annuaire) alors qu'un tableau ferait l'affaire. D'ailleurs d'un point de vue pédagogique il est toujours bon de savoir ce qu'il y a derrière une boite noire lorsque c'est accessible au niveau de l'élève ou de l'étudiant. De plus le style algébrique est certes compact, mais il est souvent plus difficile à maintenir (aussi bien à relire qu'à corriger), et il peut aussi générer une occupation mémoire inutile : par exemple en simulation, générer une grosse matrice de nombre aléatoires alors qu'une boucle agissant sur des lignes de cette matrice générées une par une et effacées après usage est incomparablement moins gourmand en mémoire.

Enfin, l'optimisation d'un code destiné à être vraiment utilisé nécessite souvent la compilation des parties critiques, et l'optimisation de ces parties critiques se fait souvent de manière différente et parfois opposée aux habitudes acquises en optimisant du code interprété. Illustrons cela plus en détails sur un exemple très utile en mathématique : le produit scalaire de deux vecteurs et le produit d'une matrice par un vecteur. Il s'agit d'une boucle `pour` que l'on code par

```
fonction ps(u,v)
 local r,j,n
 r:=0;
 pour j de 0 jusque n-1 faire
  r:=r+u[j]*v[j];
 fpour;
 return r;
ffonction
```

On peut optimiser en utilisant l'instruction d'incrémentation `r += u[j]*v[j];` au lieu de `r:=r+u[j]*v[j];`.

Traduit dans un langage compilé, ce code n'est pas loin d'être optimal, par exemple en C++ :

```
double ps(const vector<double> & u,const vector<double> & v){
  size_t j,n=u.size();
  double r=0;
  for (j=0;j<n;j++){
    r += u[j]*v[j];
  }
  return r;
}
```

Les variables `r,j,n` sont stockées dans des régistres du microprocesseur. Par itération, il y a 2 additions d'entiers 64 bits (adresse de base des vecteurs et valeur de l'indice), 2 lectures en mémoire de 8 octets (`u[j]` et `v[j]`), puis une multiplication de flottants, une addition de flottant, une incrémentation d'entier 64 bits (l'indice), une comparaison entre 2 entiers 64 bits, un branchement. Ce qui prend le plus de temps c'est la lecture en mémoire des double des vecteurs `u` et `v` et le branchement/test de fin de boucle.

En langage interprété, ce code sera beaucoup plus lent, car à chaque itération, pour déterminer la valeur de `u[j]` et `v[j]`, il faut évaluer la variable d'indice `j` (lue dans l'annuaire des variables affectées), de même pour les variables `u` et `v`, s'apercevoir que ce sont bien des tableaux, comparer `j` à la taille de `u` et de `v` (erreur si l'indice est trop grand), extraire la j-ième case du tableau. Ensuite on fait le produit, ce qui nécessite de tester le type des variables, on ajoute le résultat à `r` ce qui nécessite à nouveau de tester le type des arguments de `+=`. Puis on incrémente `j` de 1 et on compare à `n` (qu'il faut évaluer). Pour diminuer les opérations de gestion de la boucle, certains langages interprétés ont une instruction de boucle spécifique pour itérer sur les éléments d'une liste, par exemple si on veut afficher les éléments d'une liste au lieu de faire

```
fonction aff(u)
 local j,n
 pour j de 0 jusque n-1 faire
  afficher(u[j]);
 fpour;
ffonction;
```

on écrira

```
fonction aff(u)
 local j;
 pour j in u faire
  afficher(j);
 fpour;
ffonction;
```

Cette boucle est implémentée en interne par une variable de longueur de la liste `n`, une autre d'indice, disons `k`, et `j` est évalué par calcul de `u[k]`, mais les opérations de gestion de la boucle sont pré-compilées et donc plus rapides. Dans ce type de boucle, le code est plus compact et ne pose aucun problème de maintenance. Par contre, pour le produit scalaire, faire la même chose nécessite d'introduire des objets plus complexes : on peut imaginer générer la liste des paires `u[j],v[j]` puis itérer sur cette liste. Mais si on ne veut pas créer inutilement en mémoire une liste de paires, cette liste doit être générée de manière virtuelle (par exemple au moyen d'une paire de pointeurs sur les tableaux `u` et `v`) et il faut disposer d'une commande capable de prendre en argument une liste virtuelle. Par exemple en Python on pourrait écrire `sum(i*j for i,j in zip(u,v))`. Un mécanisme de création de liste virtuelle peut bien entendu avoir un intérêt intrinsèque, mais il faut avoir conscience que les objets que l'on manipule sont plus complexes que les listes (qui sont déjà des objets non triviaux), et que l'utilisation de cet objet dans l'exemple du produit scalaire pour optimiser une boucle est un artéfact de langage interprété, et qu'il sera difficile d'optimiser plus avec ce style de programmation. Il sera d'ailleurs plus facile d'optimiser en langage compilé sans utiliser ce type d'objets.

Ainsi, il est possible d'optimiser la boucle du produit scalaire, on peut pour n grand regrouper plusieurs itérations, et gagner du temps sur les parties test/branchement, par exemple 2 par 2

```
double ps(const vector<double> & u,const vector<double> & v){
  size_t j,n=u.size();
  double r=0;
```

```
  n--;
  for (j=0;j<n;j+=2){
    r += u[j]*v[j]+u[j+1]*v[j+1];
  }
  n++;
  for (;j<n;j++){
    r += u[j]*v[j];
  }
  return r;
}
```

Et si on effectue le produit d'une matrice M par un vecteur v, on peut optimiser le temps d'exécution en mutualisant la lecture des coefficients de v, on fait plusieurs produits scalaires avec v simultanément.

```
double ps(const vector<double> & u1,
  const vector<double> & u2,const vector<double> & v){
  size_t j,n=u1.size();
  double r1=0,r2=0;
  n--;
  for (j=0;j<n;j+=2){
    double vj=v[j],vj1=v[j+1];
    r1 += u1[j]*vj+u1[j+1]*vj1;
    r2 += u2[j]*vj+u2[j+1]*vj1;
  }
  n++;
  for (;j<n;j++){
    r1 += u1[j]*v[j];
    r2 += u1[j]*v[j];
  }
  return r;
}
```

Ici en faisant 2 produits scalaires simultanément, on fait 6 lectures en mémoire au lieu de 8 pour 2 itérations.

Il nous parait donc essentiel pour un scientifique d'apprendre aussi un langage pas trop éloigné de la machine, ou au moins d'être capable de coder avec des objets de base.