Ce document propose un petit guide de référence de Xcas, puis des énoncés de TP pour 6 séances de 1h30 :

1. TP1 : apprentissage de xcas
2. TP2 : écriture sous forme matricielle des problemes d'algebre lineaire)
3. TP3,4 : programmation du pivot de Gauss et applications : inverse, base du noyau/de l'image,
4. TP5 : polynome minimal et recherche des espaces propres
5. TP6 : codes correcteurs (peut etre donné sous forme de mini-projets ?)

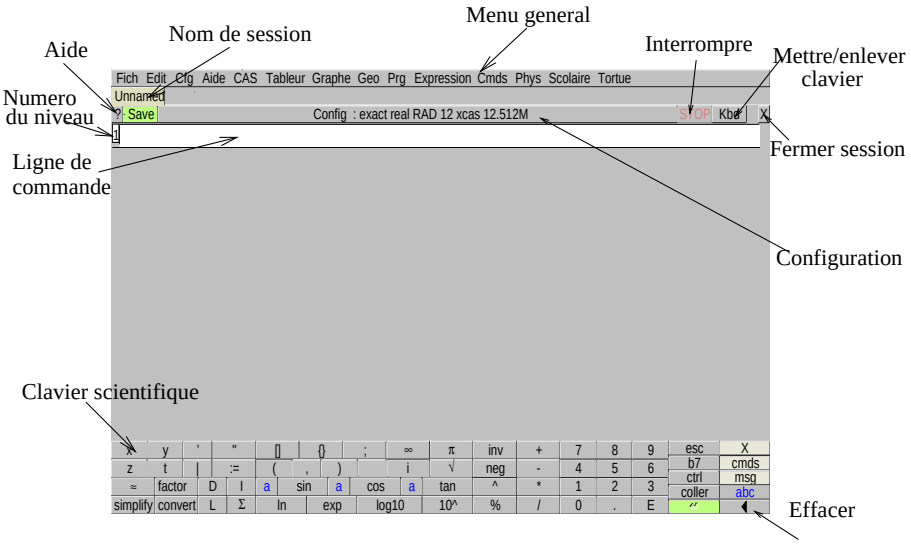
1 Premiers pas avec Xcas

Pour télécharger Xcas, allez sur le site
`http://www-fourier.ujf-grenoble.fr/~parisse/install_fr.html`
Pour traiter les exemples, il est conseillé d'ouvrir Xcas :

- Sous Windows en installation locale, on clique sur l'icone `xcasfr` du bureau.
- Sous Linux avec Gnome, on clique sur Xcas du menu Education. Sinon, ouvrir un terminal et taper `xcas &`.
- sur Mac, cliquez sur Xcas dans le menu Applications du Finder.

Lors de la première utilisation, choisissez `Xcas` lorsqu'on vous demande de choisir une syntaxe (sauf si vous connaissez le langage Maple). Nous donnons ici seulement le minimum de l'interface à connaitre pour commencer à programmer. On consultera plutot le manuel Débuter en calcul formel ou les autres manuels (menu Aide) pour apprendre à utiliser les fonctionnalités de Xcas en calcul formel, géométrie, tableur, etc..

L'interface apparaît comme suit au lancement de `Xcas`.



Vous pouvez la redimensionner. De haut en bas cette interface fait apparaître :

- un barre de menu gris cliquable : `Fich`, `Edit`, `Cfg`, `Aide`, `CAS`, `Tableur`, `Graphe`, `Geo`,...
- un onglet indiquant le nom de la session, ou `Unnamed` si la session n'a pas encore été sauvegardée,
- une zone de gestion de la session avec un bouton `?` pour ouvrir l'index de l'aide un bouton `Save` pour sauvegarder la session, un bouton `Config: exact real ...` affichant la configuration et permettant de la modifier, un bouton `STOP` permettant d'interrompre un calcul trop long, un bouton `Kbd` pour faire apparaitre un clavier ressemblant à celui d'une calculatrice, qui peut faciliter vos saisies, et un bouton `x` pour fermer la session
- une zone rectangulaire blanche numérotée 1 (première ligne de commande) dans laquelle vous pouvez taper votre première commande (cliquez si nécessaire pour faire apparaitre le curseur dans cette ligne de commande) : `1+1`, suivi de la touche "Entrée" ("Enter" ou "Return" selon les claviers). Le résultat apparaît au-dessous, et une nouvelle ligne de commande s'ouvre, numérotée 2.

Vous pouvez modifier l'aspect de l'interface et sauvegarder vos modifications pour les utilisations futures (menu `Cfg`).

Vous n'avez pour l'instant qu'à entrer des commandes dans les lignes de commandes successives. Si vous utilisez la version html de ce cours, vous pouvez copier-coller les commandes proposées depuis votre navigateur. Chaque ligne de commande saisie est exécutée par la touche "Entrée". Essayez par exemple d'exécuter les commandes suivantes.

```
1/3+1/4
sqrt(2)^5
resoudre(x+3=1,x)
50!
```

Toutes les commandes sont gardées en mémoire. Vous pouvez donc remonter dans l'historique de votre session pour modifier des commandes antérieures. Essayez par exemple de changer les commandes précédentes en :

```
1/3+3/4
sqrt(5)^2
resoudre(2*x+3=0)
500!
```

Notez que

- pour effacer une ligne de commande, on clique dans le numéro de niveau à gauche de la ligne de commande, qui apparait alors en surbrillance. On appuie ensuite sur la touche d'effacement. On peut aussi déplacer une ligne de commande avec la souris.
- Le menu `Edit` vous permet de préparer des sessions plus élaborées qu'une simple succession de commandes. Vous pouvez créer des groupes de lignes de commandes (sections), ajouter des commentaires ou fusionner des niveaux en un seul niveau.
- Le menu `Prg` contient la plupart des instructions utiles pour programmer.

2 Les objets du calcul formel

2.1 Les nombres

Les nombres peuvent être exacts ou approchés. Les nombres exacts sont les constantes prédéfinies, les entiers, les fractions d'entiers et plus généralement toute expression ne contenant que des entiers et des constantes, comme `sqrt(2)*e^(i*pi/3)`. Les nombres approchés sont notés avec la notation scientifique standard : partie entière suivie du point de séparation et partie fractionnaire (éventuellement suivie de `e` et d'un exposant). Par exemple, `2` est un entier exact, `2.0` est la version approchée du même entier ; `1/2` est un rationnel, `0.5` est la version approchée du même rationnel. `Xcas` peut gérer des nombres entiers en précision arbitraire : essayez de taper `500!` et comptez le nombre de chiffres de la réponse.

On passe d'une valeur exacte à une valeur approchée par `evalf`, on transforme une valeur approchée en un rationnel exact par `exact` Les calculs sont effectués en mode exact si tous les nombres qui interviennent sont exacts. Ils sont effectués en mode approché si un des nombres est approché. Ainsi `1.5+1` renvoie un nombre approché alors que `3/2+1` renvoie un nombre exact.

```
sqrt(2)
evalf(sqrt(2))
sqrt(2)-evalf(sqrt(2))
exact(evalf(sqrt(2)))*10^9
exact(evalf(sqrt(2)*10^9))
```

Pour les nombres réels approchés, la précision par défaut est proche de 14 chiffres significatifs (la précision relative est de 53 ou 45 bits pour les réels flottants normalisés selon les versions de Xcas). Elle peut être augmentée, en donnant le nombre de décimales désiré comme second argument de `evalf`.

```
evalf(sqrt(2),50)
evalf(pi,100)
```

On peut aussi changer la précision par défaut pour tous les calculs en modifiant la variable `Digits`.

```
Digits:=50
evalf(pi)
evalf(exp(pi*sqrt(163)))
```

La lettre `i` est réservée à $\sqrt{-1}$ et ne peut être réaffectée ; en particulier on ne peut pas l'utiliser comme indice de boucle.

```
(1+2*i)^2
(1+2*i)/(1-2*i)
e^(i*pi/3)
```

`Xcas` distingue l'infini non signé `infinity` (∞), de `+infinity` ($+\infty$) et de `-infinity` ($-\infty$).

```
1/0; (1/0)^2; -(1/0)^2
```

Constantes prédéfinies	
`pi`	$\pi \simeq 3.14159265359$
`e`	$e \simeq 2.71828182846$
`i`	$i = \sqrt{-1}$
`infinity`	∞
`+infinity`	$+\infty$
`-infinity`	$-\infty$

2.2 Les caractères et les chaînes

Une chaîne est parenthésée par des guillemets (`"`). Un caractère est une chaîne ayant un seul élément.

```
s:="azertyuiop"
size(s)
s[0]+s[3]+s[size(s)-1]
concat(s[0],concat(s[3],s[size(s)-1]))
head(s)
tail(s)
mid(s,3,2)
l:=asc(s)
ss:=char(l)
string(123)
expr(123)
expr(0123)
```

Chaînes	
`asc`	chaîne->liste des codes ASCII
`char`	liste des codes ASCII->chaîne
`size`	nombre de caractères
`concat` ou `+`	concaténation
`mid`	morceau de chaîne
`head`	premier caractère
`tail`	chaîne sans le 1ier caractère
`string`	nombre ou expression->chaîne
`expr`	chaîne->nombre (base 10 ou 8) ou expression

2.3 Les variables

On dit qu'une variable est formelle si elle ne contient aucune valeur : toutes les variables sont formelles tant qu'elles n'ont pas été affectées (à une valeur). L'affectation est notée `:=`. Au début de la session `a` est formelle, elle devient affectée après l'instruction `a:=3`, `a` sera alors remplacé par 3 dans tous les calculs qui suivent, et

`a+1` renverra 4. `Xcas` conserve tout le contenu de votre session. Si vous voulez que la variable `a` après l'avoir affectée, soit à nouveau une variable formelle, il faut la "vider" par `purge(a)`. Dans les exemples qui suivent, les variables utilisées sont supposées avoir été purgées avant chaque suite de commandes.

Il ne faut pas confondre

- le signe `:=` qui désigne l'affectation
- le signe `==` qui désigne une égalité booléenne : c'est une opération binaire qui retourne 1 ou 0 (1 pour true qui veut dire Vrai et 0 pour false qui veut dire Faux)
- le signe `=` utilisé pour définir une équation.

```
a==b
a:=b
a==b
solve(a=b,a)
solve(2*a=b+1,a)
```

On peut faire certains types d'hypothèses sur une variable avec la commande `assume`, par exemple `assume(a>2)`. Une hypothèse est une forme spéciale d'affectation, elle efface une éventuelle valeur précédemment affectée à la variable. Lors d'un calcul, la variable n'est pas remplacée mais l'hypothèse sera utilisée dans la mesure du possible, par exemple `abs(a)` renverra `a` si on fait l'hypothèse `a>2`.

```
sqrt(a^2)
assume(a<0)
sqrt(a^2)
assume(n,integer)
sin(n*pi)
```

La fonction `subst` permet de remplacer une variable dans une expression par un nombre ou une autre expression, sans affecter cette variable.

```
subst(a^2+1,a=1)
subst(a^2+1,a=sqrt(b-1))
a^2+1
```

Remarque : pour stocker une valeur dans une variable par référence, par exemple pour modifier une valeur dans une liste (un vecteur, une matrice), sans recréer une nouvelle liste mais en modifiant en place la liste existante, on utilise l'instruction `=<` au lieu de `:=`. Cette instruction est plus rapide que l'instruction `:=`, car elle économise le temps de copie de la liste.

2.4 Les expressions

2.4.1 Définition

Une expression est une combinaison de nombres et de variables reliés entre eux par des opérations : par exemple `x^2+2*x+c`.

Lorsqu'on valide une commande, `Xcas` remplace les variables par leur valeur si elles en ont une, et exécute les opérations.

```
(a-2)*x^2+a*x+1
a:=2
(a-2)*x^2+a*x+1
```

Certaines opérations de simplification sont exécutées automatiquement lors d'une évaluation :

- les opérations sur les entiers et sur les fractions, y compris la mise sous forme irréductible
- les simplifications triviales comme $x + 0 = x$, $x - x = 0$, $x^1 = x \ldots$
- quelques formes trigonométriques : $\cos(-x) = \cos(x)$, $\cos(\pi/4) = \sqrt{2}/2$, $\tan(\pi/4) = 1 \ldots$

Nous verrons dans la section 2.4.2 comment obtenir plus de simplifications.

2.4.2 Développer et simplifier une expression

En-dehors des règles de la section précédente, il n'y a pas de simplification systématique. Il y a deux raisons à cela. La première est que les simplifications non triviales sont parfois coûteuses en temps, et le choix d'en faire ou non est laissé à l'utilisateur ; la deuxième est qu'il y a en général plusieurs manières de simplifier une même expression, selon l'usage que l'on veut en faire. Les principales commandes pour transformer une expression sont les suivantes :

- `expand` : développe une expression en tenant compte uniquement de la distributivité de la multiplication sur l'addition et du développement des puissances entières.
- `normal` et `ratnormal` : d'un bon rapport temps d'exécution-simplification, elles écrivent une fraction rationnelle (rapport de deux polynômes) sous forme de fraction irréductible développée ; `normal` tient compte des nombres algébriques (par exemple comme `sqrt(2)`) mais pas `ratnormal`. Les deux ne tiennent pas compte des relations entre fonctions transcendantes (par exemple comme `sin` et `cos`).
- `factor` : un peu plus lente que les précédentes, elle écrit une fraction sous forme irréductible factorisée.
- `simplify` : elle essaie de se ramener à des variables algébriquement indépendantes avant d'appliquer `normal`. Ceci est plus coûteux en temps et "aveugle" (on ne contrôle pas les réécritures intermédiaires). Les simplifications faisant intervenir des extensions algébriques (par exemple des racines carrées) nécessitent parfois deux appels et/ou des hypothèses (`assume`) pour enlever des valeurs absolues avant d'obtenir la simplification souhaitée.
- `tsimplify` essaie de se ramener à des variables algébriquement indépendantes mais sans appliquer `normal` ensuite.

Dans le menu `Math` du bandeau supérieur, les 4 sous-menus de réécriture contiennent d'autres fonctions, pour des transformations plus ou moins spécialisées.

```
b:=sqrt(1-a^2)/sqrt(1-a)
ratnormal(b)
normal(b)
tsimplify(b)
```

```
simplify(b)
simplify(simplify(b))
assume(a<1)
simplify(b)
simplify(simplify(b))
```

La fonction `convert` permet de passer d'une expression à une autre équivalente, sous un format qui est spécifié par le deuxième argument.

```
convert(exp(i*x),sincos)
convert(1/(x^4-1),partfrac)
convert(series(sin(x),x=0,6),polynom)
```

Transformations	
`simplify`	simplifier
`tsimplify`	simplifier (moins puissant)
`normal`	forme normale
`ratnormal`	forme normale (moins puissant)
`expand`	développer
`factor`	factoriser
`assume`	rajout d'hypothèses
`convert`	transformer en un format spécifié

2.5 Les fonctions

2.5.1 Fonctions usuelles

De nombreuses fonctions sont déjà définies dans `Xcas`, en particulier les fonctions classiques. Les plus courantes figurent dans le tableau ci-après ; pour les autres, voir le menu `Math`.

Fonctions classiques	
`abs`	valeur absolue
`round`	arrondi
`floor`	partie entière (plus grand entier $\leq$)
`ceil`	plus petit entier $\geq$
`abs`	module
`arg`	argument
`conj`	conjugué
`sqrt`	racine carrée
`exp`	exponentielle
`log`	logarithme naturel
`ln`	logarithme naturel
`log10`	logarithme en base 10
`sin`	sinus
`cos`	cosinus
`tan`	tangente
`asin`	arc sinus
`acos`	arc cosinus
`atan`	arc tangente
`sinh`	sinus hyperbolique
`cosh`	cosinus hyperbolique
`tanh`	tangente hyperbolique
`asinh`	argument sinus hyperbolique
`acosh`	argument cosinus hyperbolique
`atanh`	argument tangente hyperbolique

2.5.2 Fonctions algébriques définies par l'utilisateur

Pour créer une nouvelle fonction, il faut la déclarer à l'aide d'une expression contenant la variable. Par exemple l'expression $x^2 - 1$ est définie par `x^2-1`. Pour la transformer en la fonction f qui à x associe $x^2 - 1$, trois possibilités existent :

```
f(x):= x^2-1
f:=x->x^2-1
f:=unapply(x^2-1,x)
f(2); f(a^2)
```

On peut définir des fonctions de plusieurs variables à valeurs dans $\mathbb{R}$ comme `f(x,y):=x+2*y` et des fonctions de plusieurs variables à valeurs dans $\mathbb{R}^p$ par exemple `f(x,y):=(x+2*y,x-y)`

2.5.3 Distinguer expression et fonction

Si `f` est une fonction d'une variable et `E` est une expression, `f(E)` est une autre expression. Il est essentiel de ne pas confondre fonction et expression. Si on définit : `E:=x^2-1`, alors la variable `E` contient l'expression $x^2 - 1$. Pour avoir la valeur de cette expression en $x = 2$ il faut écrire `subst(E,x=2)` et non `E(2)` car `E` n'est pas une fonction. Lorsqu'on définit une fonction, le membre de droite de l'affectation n'est

pas évalué. Ainsi l'écriture `E:=x^2-1; f(x):=E` définit la fonction $f : x \mapsto E$ car `E` n'est pas évalué. Par contre `E:= x^2-1; f:=unapply(E,x)` définit bien la fonction $f : x \mapsto x^2 - 1$ car `E` est évalué.

La fonction `diff` permet de calculer la dérivée d'une expression par rapport à une ou plusieurs de ses variables. Pour dériver une fonction f, on peut appliquer `diff` à l'expression $f(x)$, mais alors le résultat est une expression. Si on souhaite définir la fonction dérivée, il faut utiliser `function_diff`.

```
E:=x^2-1
diff(E)
f:=unapply(E,x)
diff(f(x))
f1:=function_diff(f)
```

Il ne **faut pas** définir la fonction dérivée par `f1(x):=diff(f(x))`, car `x` aurait dans cette définition deux sens incompatibles : c'est d'une part la variable formelle de dérivation et d'autre part l'argument de la fonction `f1`. D'autre part, cette définition évaluerait `diff` à chaque appel de la fonction (car le membre de droite d'une affectation n'est jamais évalué), ce qui serait inefficace. Il faut donc soit utiliser `f1:=function_diff(f)`, soit `f1:=unapply(diff(f(x)),x)`.

2.5.4 Opérations sur les fonctions

On peut ajouter et multiplier des fonctions, par exemple `f:=sin*exp`. Pour composer des fonctions, on utilise l'opérateur `@` et pour composer plusieurs fois une fonction avec elle-même, on utilise l'opérateur `@@`.

```
f:=x->x^2-1
f1:=f@sin
f2:=f@f
f3:=f@@3
f1(a)
f2(a)
f3(a)
```

2.6 Listes, séquences, ensembles

`Xcas` distingue plusieurs sortes de collections d'objets, séparés par des virgules :

- les listes (entre crochets)
- les séquences (entre parenthèses)
- les ensembles (entre pourcentage-accolades)

```
liste:=[1,2,4,2]
sequence:=(1,2,4,2)
ensemble:=%{1,2,4,2%}
```

Les listes peuvent contenir des listes (c'est le cas des matrices), alors que les séquences sont plates (un élément d'une séquence ne peut pas être une séquence). Dans un ensemble, l'ordre n'a pas d'importance et chaque objet est unique. Il existe une autre structure, appelée table, dont nous reparlerons plus loin.

Il suffit de mettre une séquence entre crochets pour en faire une liste ou entre accolades précédées de `%` pour en faire un ensemble. On passe d'une liste à sa séquence associée par `op`, d'une séquence à sa liste associée en la mettant entre crochets ou avec la fonction `nop`. Le nombre d'éléments d'une liste est donné par `size` (ou `nops`).

```
se:=(1,2,4,2)
li:=[se]
op(li)
nop(se)
nops(se)
%{se%}
size([se])
size(%{se%})
```

Pour fabriquer une liste ou une séquence, on utilise des commandes d'itération comme `$` ou `seq` (qui itèrent une expression) ou `makelist` (qui définit une liste à l'aide d'une fonction).

```
1$5
k^2 $ (k=-2..2)
seq(k^2,k=-2..2)
seq(k^2,k,-2..2)
[k^2$(k=-2..2)]
seq(k^2,k,-2,2)
seq(k^2,k,-2,2,2)
makelist(x->x^2,-2,2)
seq(k^2,k,-2,2,2)
makelist(x->x^2,-2,2,2)
```

La séquence vide est notée `NULL`, la liste vide `[]`. Pour ajouter un élément à une séquence il suffit d'écrire la séquence et 'élément séparés par une virgule. Pour ajouter un élément à une liste on utilise `append`. On accède à un élément d'une liste ou d'une séquence grâce à son indice mis entre crochets, le premier élément étant d'indice 0.

```
se:=NULL; se:=se,k^2$(k=-2..2); se:=se,1
li:=[1,2]; (li:=append(li,k^2))$(k=-2..2)
li[0],li[1],li[2]
```

Les polynômes sont souvent définis par une expression, mais ils peuvent aussi être représentés par la liste de leurs coefficients par ordre de degré décroissant, avec comme délimiteurs `poly1[` et `]`. Il existe aussi une représentation pour les polynômes à plusieurs variables. Les fonctions `symb2poly` et `poly2symb` permettent de passer de la représentation expression à la représentation par liste et inversement, le deuxième argument détermine s'il s'agit de polynômes en une variable (on met le nom de la variable) ou de polynômes à plusieurs variables (on met la liste des variables).

Séquences et listes	
`E$(k=n..m)`	créer une séquence
`seq(E,k=n..m)`	créer une séquence
`[E$(k=n..m)]`	créer une liste
`makelist(f,k,n,m,p)`	créer une liste
`op(li)`	passer de liste à séquence
`nop(se)`	passer de séquence à liste
`nops(li)`	nombre d'éléments
`size(li)`	nombre d'éléments
`sum`	somme des éléments
`product`	produit des éléments
`cumSum`	sommes cumulées des éléments
`apply(f,li)`	appliquer une fonction à une liste
`map(li,f)`	appliquer une fonction à une liste
`poly2symb`	polynôme associé à une liste
`symb2poly`	coefficients d'un polynôme

2.7 Instructions graphiques

Toutes les instructions du menu `Geo` ont un résultat graphique, par exemple `point(1,2)` affiche le point de coordonnées 1 et 2, `droite(A,B)` la droite passant par deux points A et B définis auparavant. On peut donner des attributs graphiques aux objets graphiques en ajoutant à la fin de l'instruction graphique l'argument `affichage=...` dont la saisie est facilitée par le menu `Graphic->Attributs`.

Lorsqu'une ligne de commande contient une instruction graphique, le résultat est affiché dans un repère 2-d ou 3-d selon la nature de l'objet généré. On peut controler le repère avec les boutons situés à droite du graphique, par exemple orthonormaliser avec le bouton `_|_`. Si une ligne de commande contient des instructions graphique et non graphiques, c'est la nature de la dernière instruction qui décide du type d'affichage.

L'instruction `A:=click()` permet de définir une variable contenant l'affixe d'un point du plan que l'on clique avec la souris.

2.8 Aide en ligne

Les commandes de Xcas sont regroupées par thèmes dans les menus du bandeau gris supérieur : `CAS`, `Graphic`, `Geo`, `Cmds`, `Phys`, ... Lorsqu'on sélectionne une commande dans un menu,

- soit l'index de l'aide s'ouvre à la commande sélectionnée (par exemple pour les commandes du menu `CAS`). Cliquez sur le bouton `Details` pour afficher la page du manuel correspondant à la commande dans votre navigateur.
- soit une boite de dialogue s'ouvre vous permettant de spécifier les arguments de la commande (par exemple pour tracer une courbe depuis le menu `Graphic`)
- soit la commande est recopiée dans la ligne de commande. Pour connaitre la syntaxe de cette commande, appuyez sur le bouton `?` en haut à gauche, ou faites afficher la zone de `Messages` (en utilisant le menu `Cfg`,). Vous pouvez aussi configurer Xcas (menu `Cfg` puis `Configuration generale` puis

cocher la case `Aide HTML automatique`) pour que la page correspondante du manuel s'ouvre automatiquement dans votre navigateur.

Le menu `Aide` contient les différentes formes d'aide possible : un guide de l'utilisateur (interface), un guide de référence (`Manuels->Calcul formel`, aide detaillée sur chaque commande), un `Index` (liste des commandes classées par ordre alphabétique avec une ligne d'entrée permettant de se déplacer facilement) et une recherche par mots clefs.

Si vous connaissez déjà le nom d'une commande et que vous désirez vérifier sa syntaxe (par exemple `factor`), vous pouvez saisir le début du nom de commande (disons `fact`) puis taper sur la touche de tabulation (située à gauche de la touche A sur un clavier français) ou cliquer sur le bouton `?` en haut à gauche. L'index des commandes apparaît alors dans une fenêtre, positionné à la première complétion possible, avec une aide succinte sur chaque commande. Par exemple, vous voulez factoriser un polynôme, vous supposez que le nom de commande commence par `fact`, vous tapez donc `fact` puis la touche de tabulation, vous sélectionnez à la souris `factor` (ou un des exemples) puis vous cliquez sur OK.

Vous pouvez aussi saisir `?factor` pour avoir l'aide succinte en réponse. Si le nom que vous avez saisi n'est pas reconnu, des commandes proches vous sont suggérées.

2.9 Temps de calcul, place mémoire

Le principal problème du calcul formel est la complexité des calculs intermédiaires. Elle se traduit à la fois par le temps nécessaire à l'exécution des commandes et par la place mémoire requise. Les algorithmes implémentés dans les fonctions de `Xcas` sont performants, mais ils ne peuvent pas être optimaux dans tous les cas. La fonction `time` permet de connaître le temps d'exécution d'une commande (si ce temps est très court, `Xcas` exécute plusieurs fois la commande pour afficher un résultat plus précis). La mémoire utilisée apparaît dans les versions Unix dans la ligne d'état (en rouge à bas à gauche). Si le temps d'exécution d'une commande dépasse quelques secondes, il est possible que vous ayez commis une erreur de saisie. N'hésitez pas à interrompre l'exécution (bouton orange `stop` en bas à droite, il est conseillé de faire une sauvegarde de votre session auparavant).

3 Programmation

Comme le texte définissant un programme ne tient en général pas sur une ou deux lignes, il est commode d'utiliser un éditeur de programmes. Pour cela, on utilise le menu `Prg->Nouveau programme` de Xcas. Le menu `Prg->Ajouter` facilite aussi la saisie des principales structures de controle de programmation.

3.1 Tests

On peut tester l'égalité de 2 expressions en utilisant l'instruction `==`, alors que `!=` teste si 2 expressions ne sont pas égales. On peut aussi tester l'ordre entre 2 expressions

avec <, <=, >, >=, il s'agit de l'ordre habituel sur les réels pour des données numériques ou de l'ordre lexicographique pour les chaines de caractères.

Un test renvoie 1 s'il est vrai, 0 s'il est faux. On peut combiner le résultat de deux tests au moyen des opérateurs logiques && (et logique), || (ou logique) et on peut calculer la négation logique d'un résultat de test ! (négation logique). On utilise ensuite souvent la valeur du test pour exécuter une instruction conditionnelle `si ... alors ... sinon ... fsi`.
N.B. : Xcas admet aussi une syntaxe compatible avec le langage C
`if (condition) { bloc_vrai } else { bloc_faux }`.

Par exemple, on pourrait stocker la valeur absolue d'un réel x dans y par :

```
si x>0 alors y:=x; sinon y:=-x; fsi;
```

(on peut bien sur utiliser directement `y:=abs(x)`).

Exemples : Tester si un triangle dont on fait cliquer les 3 sommets à l'utilisateur est rectangle.

3.2 Boucles

On peut exécuter des instructions plusieurs fois de suite en utilisant une boucle définie (le nombre d'exécutions est fixé au début) ou indéfinie (le nombre d'exécutions n'est pas connu). On utilise en général une variable de controle (indice de boucle ou variable de terminaison).

- Boucle définie
  `for(init;condition;incrementation){ instructions }`
  `for ... from ... to ... do ... od`
  Exemple, calcul de 10 !
  ```
  f:=1; for (j:=1;j<=10;j++){ f:=f*j; }
  f:=1; for j from 1 to 10 do f:=f*j; od;
  ```
  Attention, vous ne pouvez pas utiliser `i` comme indice de boucle, car `i` est prédéfini (comme $\sqrt{-1}$)
- Boucle indéfinie
  `while (...) { ... }`
  Exemple, algorithme d'Euclide
  ```
  while (b!=0){ r:=irem(a,b); a:=b; b:=r;}
  ```

Xcas accepte aussi l'arrêt de boucle en cours d'exécution (`if (...) break;`) dont l'usage peut éviter l'utilisation de variables de controle compliquées.

3.3 Fonctions (non algébriques)

La plupart des fonctions ne peuvent avoir une définition par une formule algébrique. On doit souvent calculer des données intermédiaires, faire des tests et des boucles. Il faut alors définir la fonction par une suite d'instructions, délimitées par `{ ... }`. La valeur calculée par la fonction est alors la valeur calculée par la dernière instruction ou peut être explicitée en utilisant le mot-clef `return` suivi de la valeur à renvoyer (N.B. : l'exécution de `return` met un terme à la fonction même s'il y a encore des instructions après).

Pour éviter que les données intermédiaires n'interfèrent avec les variables de la session principale, on utilise un type spécial de variables, les variables locales, dont la valeur ne peut être modifiée ou accédée qu'à l'intérieur de la fonction. On utilise à cet effet le mot-clef `local` suivi par les noms des variables locales séparés par des virgules. Si une fonction calcule plusieurs données on peut les renvoyer dans une liste.

Exemple : le PGCD

```
pgcd(a,b):={
  local r;
  while (b!=0){
    r:=irem(a,b);
    a:=b;
    b:=r;
  }
  return a;
}
```

On clique ensuite sur le bouton OK, si tout va bien, le programme `pgcd` est défini et on peut le tester dans une ligne de commande par exemple par `pgcd(25,15)`.

Dans la section suivante, on va voir comment exécuter en mode pas à pas un programme, ce qui peut servir à comprendre le déroulement d'un algorithme, mais aussi à corriger un programme erroné.

3.4 Exécuter en pas à pas et mettre au point

La commande `debug` permet de lancer un programme en mode d'exécution pas à pas. Elle ouvre une fenêtre permettant de diriger l'exécution du programme passé en argument. Par exemple, on entre le programme :

```
carres(n):={
  local j,k;
  k:=0;
  for (j:=1;j<n+1;j++) {
    k:=k+j^2;
  }
  return k;
}:;
```

On tape pour debugger le programme `carres` ci-dessus :

```
debug(carres(5))
```

cela ouvre la fenêtre du debugger. En cliquant sur le bouton `sst` on peut exécuter pas à pas le programme en visualisant l'évolution des valeurs des variables locales et des paramètres du programme. Cela permet de détecter la grande majorité des erreurs qui font qu'un programme ne fait pas ce que l'on souhaite. Pour des programmes plus longs, le debugger permet de controler assez finement l'exécution du programme en placant par exemple des points d'arrêt.

Exercice : exécuter en mode pas à pas le programme `pgcd` pour quelques valeurs des arguments.

4 TP1 prise en main

1. Écrire le polynôme $(x+3)^7 \times (x-5)^6$ selon les puissances décroissantes de x.
2. Simplifier les expressions suivantes :
$$\sqrt{3+2\sqrt{2}}, \quad \frac{1+\sqrt{2}}{1+2\sqrt{2}}, \quad e^{i\pi/6}, \quad 4\text{atan}(\frac{1}{5}) - \text{atan}(\frac{1}{239})$$
3. Factoriser :
$$x^8 - 3x^7 - 25x^6 + 99x^5 + 60x^4 - 756x^3 + 1328x^2 - 960x + 256$$
$$x^6 - 2x^3 + 1, \quad (-y+x)z^2 - xy^2 + x^2y$$
4. Calculez les intégrales et simplifiez le résultat :
$$\int \frac{1}{e^x - 1}\, dx, \quad \int \frac{1}{x\ln(x)} \ln(\ln(x))dx, \quad \int e^{x^2}\, dx, \quad \int x\sin(x)e^x\, dx$$
Vérifiez en dérivant les expressions obtenues.
5. Déterminer la valeur de :
$$\int_1^2 \frac{1}{(1+x^2)^3}, \quad \int_1^2 \frac{1}{x^3+1}\, dx$$
6. Calculer les sommes suivantes
$$\sum_{k=1}^{N} k, \sum_{k=1}^{N} k^2, \sum_{k=1}^{\infty} \frac{1}{k^2}$$
7. Développer $\sin(3x)$, linéariser l'expression obtenue et vérifier qu'on retrouve l'expression initiale.
8. Calculer le développement de Taylor en $x = 0$ à l'ordre 4 de :
$$\ln(1+x+x^2), \quad \frac{\exp(\sin(x)) - 1}{x + x^2}, \quad \sqrt{1+e^x}, \quad \frac{\ln(1+x)}{\exp(x) - \sin(x)}$$
9. Trouver les entiers n tels que le reste de la division euclidienne de $123n$ par 256 soit 17.
10. Déterminer la liste des diviseurs de 45768.
Factoriser 100 !
11. Résoudre le système linéaire :
$$\begin{cases} x & + & y & + & az & = & 1 \\ x & + & ay & + & z & = & 2 \\ ax & + & y & + & z & = & 3 \end{cases}$$
12. Déterminer l'inverse de la matrice :
$$A = \begin{pmatrix} 1 & 1 & 1 & a \\ 1 & 1 & a & 1 \\ 1 & a & 1 & 1 \\ a & 1 & 1 & 1 \end{pmatrix}$$

5 TP 2 : ramener un problème d'algèbre linéaire à un pivot de Gauss

Les instructions de Xcas correspondantes sont dans le menu `Cmds->Alglin->Gauss`.

Exercice 1 Écrire le système linéaire sous forme matricielle :

$$\begin{cases} x & + & y & + & az & = & 1 \\ x & + & ay & + & z & = & 2 \\ ax & + & y & + & z & = & 3 \end{cases}$$

puis le résoudre avec l'instruction de réduction sous forme échelonnée `rref`.

Exercice 2

Calcul de la somme de deux sous-espaces vectoriels.
On donne deux sous-espaces vectoriels E_1 et E_2 de $\mathbb{R}^n$ par deux familles génératrices (c'est-à-dire une liste de vecteurs), il s'agit d'écrire un algorithme permettant

- de trouver une base de $E_1 + E_2$
- de trouver une écriture d'un élément de $E_1 + E_2$ comme somme d'un élément de E_1 et d'un élément de E_2

On pourra utiliser les fonctions `rref` ou/et `ker` de Xcas. Tester avec deux sous-espaces de dimension 2 de $\mathbb{R}^5$. N'oubliez pas de rédiger une justification mathématique de la méthode mise en oeuvre.

Exercice 3

Calcul de l'intersection de deux sous-espaces vectoriels.
On donne deux sous-espaces vectoriels E_1 et E_2 de $\mathbb{R}^n$ par deux familles génératrices (c'est-à-dire une liste de vecteurs), il s'agit d'écrire un algorithme permettant de trouver une base de $E_1 \cup E_2$. On pourra utiliser les fonctions `rref` ou/et `ker` de Xcas. Tester avec 2 sous-espace de dimension 3 de $\mathbb{R}^4$. N'oubliez pas de rédiger une justification mathématique de la méthode mise en oeuvre.

Exercice 4

Mise en oeuvre du théorème de la base incomplète. On donne une famille de vecteurs de $\mathbb{R}^n$ (liste de vecteurs), il s'agit d'écrire un algorithme permettant d'extraire une base du sous-espace engendré, puis de compléter par des vecteurs afin de former une base de $\mathbb{R}^n$ tout entier, et enfin d'écrire un vecteur de $\mathbb{R}^n$ comme combinaison linéaire des vecteurs de la base complétée. On pourra utiliser les fonctions `rref` ou/et `ker` de Xcas. Tester avec une famille de 3 vecteurs de $\mathbb{R}^5$. N'oubliez pas de rédiger une justification mathématique de la méthode mise en oeuvre.

6 TP3,4 : pivot de Gauss et applications

6.1 Programmation

On rappelle qu'en mode Xcas, les indices commencent à 0 (en mode maple les indices commencent à 1). Etant donné une matrice M ayant L lignes et C colonnes, on demande de programmer l'algorithme du pivot de Gauss que l'on rappelle :

1. Initialiser la ligne courante l et la colonne courante c à 0
2. Tant que $l < L$ et $c < C$ faire
3. Chercher dans la colonne c à partir de la ligne l un coefficient non nul (appelé pivot)
4. S'il n'y en a pas, incrémenter c et revenir à l'étape 2
5. S'il y en a un, échanger la ligne l avec la ligne du pivot (`rowSwap(matrice,l1,l2)`)
6. Pour les lignes j variant de $l+1$ à $L-1$ ou de 0 à $L-1$ à l'exclusion de la ligne l (selon que l'on effectue une réduction sous-diagonale ou complète), remplacer la ligne L_j par $L_j - \alpha L_l$ où α est calculé pour annuler le coefficient de la colonne c de L_j (`mRowAdd(coeff,matrice,l1,l2)`)
7. Incrémenter l et c et revenir à l'étape 2.
8. Normaliser à 1 le premier coefficient non nul de chaque ligne (en divisant la ligne par ce coefficient)

6.2 Inverse d'une matrice

Pour calculer l'inverse d'une matrice M carrée de taille n, on peut résoudre simultanément n systèmes linéaires du type $Mx_k = y_k$ où y_k représente (en colonne) les coordonnées du k-ième vecteur de la base canonique pour k variant de 1 à n. On écrit donc la matrice M puis les colonnes des coordonnées de ces n vecteurs, donc la matrice identité de taille n. On réduit complètement la matrice par l'algorithme du pivot de Gauss. Si M est inversible, les n premières colonnes après réduction doivent être la matrice identité de taille n, alors que les n colonnes qui suivent sont les coordonnées des x_k donc ces n colonnes constituent M^{-1}. Écrire un programme de calcul d'inverse de matrice par cet algorithme.

6.3 Noyau d'une application linéaire

Soit M la matrice d'une application linéaire dont on veut calculer une base du noyau. On réduit complètement M par le pivot de Gauss. On enlève les lignes de 0 finales de M. Puis on insère des lignes de 0 pour placer les pivots (premier coefficient non nul de chaque ligne) sur la diagonale principale. On obtient ainsi une matrice carrée M_r de taille le nombre de colonnes de M. On parcourt la diagonale de M_r, et chaque fois qu'on rencontre un 0, on ajoute un vecteur à la base du noyau, ce vecteur ayant pour coordonnées la colonne actuelle où on a remplacé le 0 de la diagonale par -1.

6.4 Algorithme de Gauss-Bareiss

Lorsque les coefficients de la matrice $M = (m_{j,k})_{0 \leq j < L, 0 \leq k < C}$ sont entiers, on peut souhaiter éviter de faire des calculs dans les rationnels, et préférer utiliser une combinaison linéaire de lignes ne faisant intervenir que des coefficients entiers. On peut par exemple effectuer l'opération (où l, c désignent la ligne et colonne du pivot)

$$L_j \leftarrow m_{l,c} L_j - m_{j,c} L_l$$

Tester la taille des coefficients obtenus pour une matrice carrée aléatoire de taille 5 puis 10. L'idée est-elle bonne ?

On peut montrer qu'on peut toujours diviser par le pivot utilisé pour réduire la colonne précédente (initialisé à 1 lors de la réduction de la première colonne)

$$L_j \leftarrow \frac{1}{p_{\text{prec}}}(m_{l,c} L_j - m_{j,c} L_l)$$

Tester à nouveau sur des matrices carrées de taille 5, 10, vérifier que les calculs sont bien effectués dans $\mathbb{Z}$. Comparer le dernier coefficient en bas à droite avec le déterminant de la matrice (si vous avez vu les propriétés des déterminants, démontrez ce résultat. Plus difficile : en déduire qu'on peut bien diviser par le pivot de la colonne précédente en considérant des déterminants de matrices extraites)

7 TP5 : polynome minimal et caractéristique.

On propose ici quelques algorithmes de calcul du polynôme minimal M et/ou caractéristique P d'une matrice carrée A de taille n. On peut bien sur calculer le polynôme caractéristique en calculant directement le déterminant (par exemple avec l'algorithme de Gauss-Bareiss pour éviter les fractions de polynômes), mais c'est souvent plus couteux que d'utiliser un des deux algorithmes ci-dessous.

7.1 Interpolation de Lagrange

On sait que le degré de P est n, il suffit donc de connaitre la valeur de P en $n+1$ points distincts pour connaitre P. Par exemple, on calcule $P(k)$ pour $k = 0, ..., n$, et en utilisant l'instruction `lagrange` de Xcas, on en déduit le polynôme caractéristique. Programmer cet algorithme et testez votre programme en comparant le résultat de votre programme et de l'instruction `charpoly` de Xcas sur quelques matrices.

7.2 Algorithme probabiliste.

Cet algorithme permet de calculer le polynôme caractéristique C dans presque tous les cas, en recherchant le polynôme minimal M de A (on note m le degré de M). Cf. le DM 3 pour un exemple de mise en oeuvre.

On sait que $M(A) = 0$, donc pour tout vecteur v, $M(A)v = 0$. Si $M(x) = \sum_{k=0}^{m} M_k x^k$, alors

$$0 = M(A)v = \sum_{k=0}^{m} M_k A^k v$$

On va donc rechercher les relations linéaires qui existent entre les $n+1$ vecteurs $v, ..., A^n v$. Cela revient à déterminer le noyau K de l'application linéaire dont la matrice a pour colonnes $v, ..., A^n v$. On sait que le vecteur $(M_0, ..., M_m, 0, ..., 0) \in \mathbb{R}^{n+1}$ des coefficients de M (complété par des 0 si $m < n$) appartient à ce noyau K. Si le noyau K est de dimension 1, alors $m = n$, et les coefficients de M sont proportionnels aux coefficients du vecteur de la base du noyau calculé. On en déduit alors le polynôme minimal M et comme $m = n$, et aussi le polynôme caractéristique $C = M$.

Programmer cet algorithme en prenant un vecteur v aléatoire. Attention, pour calculer $A^k v$ on formera la suite récurrente $v_k = Av_{k-1}$, pourquoi ?

Si on n'a pas de chances dans le choix de v, on trouvera un noyau de dimension plus grande que 1 bien que le polynome minimal soit de degré n. On peut alors recommencer avec un autre vecteur. On peut aussi chercher la relation de degré minimal pour un v donné (elle apparait automatiquement comme premier vecteur de la base dans l'algorithme de calcul du noyau donné au TP2), prendre le PPCM P des polynomes pour 2 ou 3 vecteurs aléatoires et conclure s'il est de degré n. Programmer cette variante.

Si le polynome minimal est de degré $m < n$, on peut tester si le PPCM P est le polynome minimal en calculant $P(A)$ mais ce calcul est couteux. On peut aussi faire confiance au hasard, supposer que le polynome minimal est bien $M = P$ et essayer de déterminer C/M par d'autres moyens, par exemple la trace si $n = m + 1$. On dispose alors d'un moyen de vérification en calculant l'espace caractéristique correspondant à la valeur propre double. Programmer cette variante.

8 TP6 : codes correcteurs linéaires.

La transmission (ou la lecture) d'information de manière fiable a pris une importance grandissante avec le développement des réseaux informatiques, la numérisation des données (y compris audio et vidéo) mais aussi la communication à grande distance (par exemple sonde spatiale sur Mars). Il a fallu trouver des moyens économiques :

- de détecter des erreurs de lecture ou de transmission
- de corriger une erreur de lecture ou de transmission sans avoir besoin de la relire ou de la réémettre (pour éviter une coupure dans la lecture d'un flux audio ou vidéo, ou un délai trop long par exemple pour communiquer avec une sonde spatiale)

On présente dans ce TP quelques méthodes de détection et correction d'erreurs qui utilisent de l'algèbre linéaire et des polynômes dont les coefficients au lieu d'être des réels ou des complexes sont des entiers modulo un nombre premier, en général 2, qui forment un corps. Dans Xcas, on représente les nombres ou les vecteurs ou les polynômes modulo 2 en ajoutant `mod 2` ou `%2`, par exemple `1%2`, `[1,2] %2`, `(x^3+3x+1)%2`

On appellera symbole d'information l'unité de base transmise, qu'on supposera appartenir à un corps fini K, par exemple pour un bit un élément de $K = \mathbb{Z}/2\mathbb{Z}$ (pour un octet on pourrait utiliser un élément du corps à 256 éléments $K = GF(2, 8)$ mais l'étude de ce corps est hors programme).

On veut coder un message de longueur k avec des possibilités de détection et de correction d'erreurs, pour cela on rajoute des symboles calculés à partir des précédents,

et on envoie un mot ayant n symboles (avec $n > k$).

8.1 Le bit de parité.

On prend $k = 7$ bits et $n = 8$ bits. On compte le nombre de 1 parmi les 7 bits envoyés, si ce nombre est pair, on envoie 0 comme 8ième bit, sinon 1. Au final le nombre de bits à 1 de l'octet (1 octet=8 bits) est pair. On peut ainsi détecter une erreur de transmission si à la réception le nombre de bits d'un octet est impair, mais on ne peut pas corriger d'erreurs. On peut aussi dire que l'octet représente un polynôme à coefficients dans $\mathbb{Z}/2\mathbb{Z}$ divisible par $X + 1$.

Exercice : Écrire un programme Xcas permettant de rajouter un bit de parité à une liste composée de 7 bits. Puis un programme de vérification qui accepte ou non un octet selon sa parité. On peut convertir un entier en une liste de bits par `convert(j,base,2)` (le bit de poids fort est à droite) puis en un polynome avec `symb2poly`. On peut effectuer la vérification de deux manières, en comptant le nombre de 1 ou avec l'instruction `rem`.

8.2 Codes linéaires

Définition : On multiplie le vecteur v des k symboles à transmettre par une matrice M à coefficients dans K de taille $n \times k$ et on transmet l'image Mv. Pour assurer qu'on peut identifier un antécédent unique à partir d'une image, il faut que M corresponde à une application linéaire injective (ce qui entraine $n \geq k$). On dit qu'un vecteur de n symboles est un mot du code s'il est dans l'image de l'application linéaire.

Pour assurer l'injectivité tout en facilitant le décodage, on utilise souvent une matrice identité k, k comme sous-bloc de la matrice n, k, par exemple on prend l'identité pour les k premières lignes de M, on ajoute ensuite $n - k$ lignes.

Pour savoir si un vecteur w est un mot de code, il faut vérifier qu'il est dans l'image de M. On peut par exemple vérifier qu'en ajoutant la colonne de ses coordonnées à M, on ne change pas le rang de M (qui doit être k) ou, lorsque M a comme premier sous-bloc une matrice identité, on teste si w est bien l'image de l'antécédent calculé à partir des k premières coordonnées de w.

Exercice : créez une matrice M de taille 7,4 injective. Puis un programme qui teste si un vecteur w est un mot de code et en extrait alors la partie avant codage, c'est-à-dire le vecteur v tel que $Mv = w$. Vérifiez votre programme : si on prend un vecteur $v \in K^k$ et $w = Mv$, on doit retrouver v à partir de w.
Instructions utiles : `idn` (matrice identité) `ker` (noyau d'une application linéaire), `rank` (rang), `tran` (tranposée), ... Pour créer une matrice, on peut coller les lignes de 2 matrices A et B par `[op(A),op(B)]` ou avec `blockmatrix`.

8.3 Codes polynomiaux

Définition : Il s'agit d'un cas particulier de codes linéaires. On se donne un polynôme $g(x)$ de degré $n - k$, On représente le message de longueur k à coder par un polynôme P de degré $k - 1$ dont les coefficients sont les symboles du message. On multiplie P par x^{n-k}, on calcule le reste R de la division euclidiennt de Px^{n-k} par g. On émet

alors $Px^{n-k} - R$ qui est divisible par g. Un des intérêts des codes polynomiaux est que la vérification d'appartenance au code est très simple : les mots de code correspondent à des polynômes divisibles par g.

Exercice : écrire de cette facon le codage du bit de parité. Puis une procédure Xcas de codage utilisant $g = X^7 + X^3 + 1$ (ce polynôme était utilisé par le Minitel).

8.4 Détection et correction d'erreur

Si le mot recu n'est pas dans l'image de l'application linéaire il y a eu erreur de transmission. Sinon, il n'y a pas eu d'erreur *détectable* (il pourrait y avoir eu plusieurs erreurs qui se "compensent").

Plutôt que de demander la réémission du mot mal transmis (ce qui serait par exemple impossible en temps réel depuis un robot en orbite autour de Mars), on essaie d'ajouter suffisamment d'information pour pouvoir corriger des erreurs en supposant que leur nombre est majoré par un entier N.

Remarque :
Si les erreurs de transmissions sont indépendantes, la probabilité d'avoir au moins $N+1$ erreurs dans un message de longueur p est $\sum_{k=N+1}^{p} \binom{p}{k} \epsilon^k (1-\epsilon)^{p-k}$, où ϵ est la probabilité d'une erreur de transmission. Par exemple, pour un message de 10^3 caractères, chacun ayant une probabilité d'erreur de transmission de 10^{-3}, si on prend $N = 3$, alors la probabilité d'avoir au moins 4 erreurs est de 0.019 (arrondi par excès) :

```
P(N,eps,p):=sum(comb(p,k)*eps^k*(1-eps)^(p-k),k,N+1,p):;
P(3,1e-3,10^3)
```

Selon la nature du message, on acceptera une probabilité d'avoir au moins $N+1$ erreurs plus ou moins petite.

Exemple : On ne peut pas corriger d'erreur avec le bit de parité.

8.5 Distances

La distance de Hamming de 2 mots est le nombre de symboles qui diffèrent. (il s'agit bien d'une distance au sens mathématique, elle vérifie l'inégalité triangulaire).

Exercice : écrire une procédure de calcul de la distance de Hamming de 2 mots (la fonction Xcas correspondante s'appelle `hamdist`).

La distance d'un code est la distance de Hamming minimale de 2 mots différents du code. Pour un code linéaire, la distance est aussi le nombre minimal de coefficients non nuls d'un vecteur non nul de l'image. Pour un code polynomial, la distance du code est le nombre minimal de coefficients non nuls d'un multiple de g de degré inférieur à n.

Exercice : quelle est la distance du code linéaire que vous avez créé plus haut ?

Majoration de la distance du code :
La distance minimale d'un code linéaire est inférieure ou égale à $n - k + 1$: en effet on écrit en ligne les coordonnées des images de la base canonique (ce qui revient à transposer la matrice) et on réduit par le pivot de Gauss, comme l'application linéaire est injective, le rang de la matrice est k, donc la réduction de Gauss crée $k - 1$ zéros dans chaque ligne, donc le nombre de coefficients non nuls de ces k lignes (qui sont toujours des mots de code) est au plus de $n - k + 1$.

Exercice : si votre code linéaire n'est pas de distance 3, modifiez les 3 dernières lignes pour réaliser un code de distance 3. On ne peut pas obtenir une distance $n-k+1=4$ avec $n=7$ et $k=4$ dans $\mathbb{Z}/2\mathbb{Z}$, essayez ! Essayez sur $\mathbb{Z}/3\mathbb{Z}$ et $\mathbb{Z}/5\mathbb{Z}$.

N.B. : Pour les codes non polynomiaux, par exemple convolutifs, la distance n'est pas forcément le paramètre le mieux adapté à la correction d'erreurs.

8.6 Correction au mot le plus proche

Une stratégie de correction basée sur la distance consiste à trouver le mot de code le plus proche d'un mot donné. Si la distance d'un code est supérieure ou égale à $2t+1$, et s'il existe un mot de code de distance inférieure ou égale à t au mot donné, alors ce mot de code est unique. On corrige alors le mot transmis en le remplacant par le mot de code le plus proche.

Exercice : écrivez un programme permettant de corriger une erreur dans un mot dans votre code linéaire.

On dit qu'un code t-correcteur est parfait si la réunion des boules de centre un mot de code et de rayon t (pour la distance de Hamming) est disjointe et recouvre l'ensemble des mots (K^n).

Exercice : votre code linéaire sur $\mathbb{Z}/2\mathbb{Z}$ (celui de distance 3) est-il un code 1-correcteur parfait ?

Remarque :

Pour avoir un système de codage efficace, la correction d'une erreur au mot le plus proche doit pouvoir être effectuée plus rapidement que par recherche d'un mot de code parmi les mots ayant 1, 2, ..., t symboles différents du mot recu. Certains codes polynomiaux le permettent. Il en est ainsi pour les codes de Reed-Solomon, utilisés par exemple pour corriger des erreurs de lecture sur les CD audio, l'algorithme de recherche du mot de code est une version modifiée de l'algorithme de Bézout pour les polynômes (avec des coefficients dans le corps fini à 256 éléments).