Découverte de MuPAD

Guillaume CONNAN - Lycée Jean Perrin (Rezé)

24 mai 2005

I. Introduction

MuPAD est un logiciel conçu initialement par des chercheurs de l'université allemande de Paderborn. Ce n'est pas tout à fait un logiciel libre comme nous allons le voir, mais presque. En effet, le logiciel est composé de deux grands ensembles : un noyau (kernel) qui dépend de chaque ordinateur et qui n'est pas accessible aux utilisateurs (il gère par exemple le mode d'affichage selon le système d'exploitation) et une bibliothèque de fichiers décrivant chaque algorithme de calcul en mode texte, c'est à dire affichable et modifiable sur n'importe quel éditeur de texte. Ainsi, chaque utilisateur peut vérifier, réparer, améliorer la partie mathématique du logiciel et c'est ce qui nous importe vraiment, nous qui sommes plus mathématiciens qu'informaticiens.
Par exemple, nous pouvons regarder comment MuPAD calcule le reste de la division euclidienne de a par b à l'aide de l'instruction `irem(a,b)` en éditant le fichier correspondant dans le répertoire `/usr/local/MuPAD/share/lib`

```
 $Date: 2000/06/16 07:34:55 $ $Author: tonner $ $Revision:1.3$
 irem(b,a) liefert bei Eingabe zweier ganzer Zahlen a und b den ganzzahligen (natuerlichen) Rest
bei Division von b durch a.
 irem:= proc(b,a) begin
 if args(0) <> 2 then error("expecting two arguments")
 end_if;
 if domtype(b) = DOM_INT then
 if domtype(a) = DOM_INT then
 return( b - a*specfunc::trunc(b/a) )
 elif testtype( a,Type::Numeric ) then error("Integer expected")
else return( procname(b,a) )
 end_if  elif testtype( b,Type::Numeric ) or domtype(a) <> DOM_INT and testtype(a,Type::Numeric)
then
 error("Integer expected")
 else return( procname(b,a) ) end_if
end_proc:
```

Certains connaissent déjà le minimum vital en programmation et voient en gros de quoi il retourne. Pour les autres, nous allons étudier les boucles très rapidement. Mais tout le monde peut comprendre qu'il est possible de modifier ce texte pour le personnaliser. Par exemple, on peut remplacer la ligne

```
 error("Integer expected")
```

par

```
 error("Bougre d'âne ! Il faut renter deux entiers")
```

qui est plus conviviale.
On peut aussi y trouver un intérêt mathématique (qui est assez limité dans l'exemple précédent) : par exemple, on peut se demander comment MuPAD teste si un entier est premier à l'aide de `isprime(n)` qui renvoie `TRUE` ou `FALSE`. Cette question peut être étudiée avec des élèves comme un point de départ pour l'étude de la théorie des nombres en terminale.

Voilà pour la liberté. Mais comme d'habitude, cette liberté entraîne quelques efforts de notre part (je ne dirai pas qu'elle a un prix, mais vous voyez ce que je veux dire). Il faut faire quelques efforts pour installer MuPAD de manière conviviale sur son ordinateur.
Tout d'abord il faut télécharger la version 3.1 pour linux : elle est gratuite et se présente sous la forme d'un fichier rpm récupérable à l'adresse

`http://research.mupad.de/download.html.`

Une fois installé, il restera à s'occuper du logiciel chargé de tracer les graphiques, mais nous verrons ça la prochaine fois.
Une fois votre PATH complété en tapant par exemple

```
ln -sf /usr/local/MuPAD/share/bin/* /usr/local/bin
```

vous pouvez lancer l'application de plusieurs manières :

- par la commande `mupad` à partir d'un terminal
- par la commande `xmupad` que nous utiliserons aujourd'hui et qui permet de travailler de manière plus conviviale sous X avec une barre de menu assez spartiate (germanique...), mais bon, c'est déjà pas mal. Un seul souci : MuPAD est allemand et ne connait donc pas les accents. Les barres de menus seront donc infestées de?? ,» et autres signes caballistiques. Pour vous en débarrasser, il suffit d'utiliser un éditeur de texte pour modifier le script de `xmupad` qui se trouve sur mon ordi dans `/usr/local/MuPAD/share/bin/xmupad` sans oublier de modifier les droits de propriété de `bin` et `xmupad`. Vous y trouverez une ligne
  `export LC_CTYPE=C`
  que vous remplacerez par
  `export LC_CTYPE="fr_FR"`
- vous pouvez utilisez des éditeurs de textes externes comme par exemple TeXmacs qui, si MuPAD est installé, contient dans le menu `texte` un sous-menu `Session` où vous trouverez une option `MuPAD`. Vous serez alors dans un environnement très convivial, avec toutes les options d'édition imaginables.
- il existe également l'environnement Mupacs qui permet l'utilisation de MuPAD dans le célèbre emacs. Vous trouverez des renseignements sur le site `mupacs.sf.net`

Il existe encore d'autres possibilités, mais celles-ci suffiront pour commencer à travailler (enfin !)
Il reste à débrider la mémoire du noyau en allant chercher une license sur le site
`https://www.mupad.org/muptan/muptan.php`
Il suffira ensuite, une fois sous xmupad, de taper l'instruction suivante pour la license du lycée Jean Perrin par exemple :

```
>> register("7BSE91","112100-44A20-C176F-.....-.....")
```

Bien, nous supposons maintenant que tout va bien et que vous avez une version de MuPAD exploitable et correctement installée.
Nous allons bientôt découvrir les fonctionnalités utiles à des élèves de lycée (ou plus vieux) d'un logiciel de calcul formel appelé aussi CAS (Computer Algebra System) comme l'est MuPAD. Les niveaux de chacun étant divers, les activités sont de niveaux variés pour que chacun y trouve son compte.

II. La prise en mains

Avant tout, nous aurons besoin de connaître quelques outils techniques. Il faut savoir que MuPAD est doté d'un fichier d'aide au format dvi extrèmement complet et clair, même s'il est écrit en anglais. On y accède par une icône ou en tapant

```
>> ?truc;
```

pour avoir un renseignement sur `truc`. Le petit ";" est là pour dire à MuPAD que l'on veut que le résultat de l'instruction s'affiche. Si l'on veut que MuPAD reste muet, on terminera par ":".
Vous pouvez accéder à une aide plus personnelle en me contactant
`descopau@yahoofr`
ou, si le problème est plus ardu, en vous inscrivant à la liste d'aide en français à l'adresse
`http://www.aful.org/wws/compose_mail/mupad/`

II.1. Sauvons nos fichiers

Pour faire de l'import-export de sessions mupad, le plus souple est de les enregistrer en mode texte pour pouvoir utiliser MuPAD au choix à partir d'un terminal, de xmupad ou d'un éditeur de texte.
Pour cela, vous pouvez par exemple créer un répertoire DocMuPAD et spécifier à MuPAD les chemins d'accés et de sortie en tapant pour les sorties

```
>> WRITEPATH:="/home/moi/DocMuPAD";
```

ou simplement

```
>> WRITEPATH:="DocMuPAD";
```

et pour les entrées

```
>> READPATH:="/home/moi/DocMuPAD";
```

ou

```
>> READPATH:="DocMuPAD";
```

Ensuite, pour écrire on tapera

```
>> machin:=[diff(cos(x)/(1+x^2),x)];
```

```
>> write(Text,"test",machin);
```

Maintenant, redémarrer le noyau et taper

```
>> read("test");
```

Sinon, ouvrez MuPAD dans un terminal en vous plaçant dans le fichier de sauvegarde que vous aurez créé et un Ctrl S sauvera les fichiers dedans.
Vous pouvez même exporter vos résultats au format LaTeX

```
>> truc:=generate::TeX(%);
```

```
>> write(Text,"/home/moi/DocTeX/test.tex",truc)
```

Et comme j'écris sous LaTeX, voyez le résultat :

$$\left[-\frac{\sin(x)}{x^2+1}-\frac{2\ x\ \cos(x)}{(x^2+1)^2}\right]$$

ou tout simplement

```
>> generate::TeX(3/2);
```

II.2. Saisie des instructions

On peut copier, coller comme sur toute machine unix en sellectionnant avec la souris par exemple puis en appuyant sur le bouton du milieu de la souris. On peut répéter une instruction tapée précédemment en utilisant la flèche haute, seulement si l'on est dans un terminal. On peut rappeler le dernier résultat donné par MuPAD en tapant %, le n-ième résultat précédent en tapant %n. MuPAD reste sourd tant qu'on a pas validé son résultat par la touche « entrée ». Si l'on veut passer à la ligne sans que MuPAD n'enregistre tout de suite notre résultat, on appuie simultanément sur « shift » et « entrée ».

```
>> 1+2
```

```
>> 1+2;
```

```
>> 1+2:
```

```
>> %*5;
```

```
>> 1+2
```

```
>>  3+4
```

```
>> 1+2;3+4;5+6:7*9;
```

```
>> %3^5
```

Nous verrons bien d'autres caractères-clefs en temps utile.

II.3. variable et type

Une variable est le nom d'un emplacement permettant de stocker un objet d'un type donné (nombre entier, vecteur, courbe, ...la liste est longue !). On utilise pour cela la saisie « := » Quand on écrit :

```
>>  a:=3/4;b:=3*x^3-2*x+7;
```

on affecte à la variable « a » le nombre $3/4$ et à « b » le polynôme $3x^3 - 2x + 7$. Tant que ces variables n'auront pas été libérées de leur affectation elles conserveront ces valeurs .

```
>> 2*a; b+4*a;
```

Pour les libérer on écrit :

```
>> delete(a);
```

```
>> 2*a; b+4*a;
```

On peut placer des commentaires en fin de ligne qui ne seront pas lus par MuPAD mais par vos étudiants pour les aider à comprendre un programme

```
>> 32 + # ceci est le symbole de l'addition # 3 ;
```

```
>> 32 / 8 ; \\Toujours plus fort : une division !
```

```
>> 2^5; 2**6; \\deux manières de désigner l'opération puissance
```

On peut toutes les libérer simultanément en faisant `reset()`,mais il est recommandé de bien réfléchir avant... On retiendra qu' écrire `a:=x` ne signifie pas donc pas « a égale x » mais « à a on affecte la valeur x ». La nuance réside dans le fait qu'il n'y a pas symétrie. Vérifiez le rapidement en tapant successivement

```
>>  r:=20;
```

puis

```
>> 32:=q;
```

En mathématiques , il n'est pas possible d'appliquer n'importe quel traitement à n'importe quelle variable (par exemple , on ne peut extraire la racine carrée d'un vecteur !), il en va de même en informatique . Avant d'appliquer une commande MuPAD à un groupe de variables , il faudra donc se demander

1) si elles sont bien du type qui convient
2) contrôler la syntaxe employée.

Pour cela, on peut interroger MuPAD sur le type d'une variable donnée

```
>>  type(3); type(3.0); type(6/2); type(1+2);type(3/2);type(I);type([I])
```

Le domaine `DOM` renvoyé est le domaine informatique du caractère rentré. Vous aurez peut-être l'occasion un jour de faire la différence avec `Dom` qui renvoie un domaine ayant des propriétés mathématiques et permettra de calculer avec des matrices à coefficients entiers, ou de calculer modulo 13 par exemple.

II.4. Travail sur les expressions numériques

Amusons-nous un peu :

```
>> 3^(4^5);
```

```
>> sqrt(3);
```

Pas très parlant peut-être...alors utilisons la commande `float` qui renvoie l'approximation décimale d'un nombre

```
>> float(sqrt(3));
```

Et si l'on veut être plus précis

```
>> DIGITS:=1515; float(sqrt(3));
```

La valeur par défaut de `DIGITS`, qui donne le nombre de chiffres significatifs d'une approximation numérique est 10.

```
>> float(PI);
```

Observez maintenant

```
>> reset();
```

```
>> (2+sqrt(3))^32);
```

Merci MuPAD, dîtes-vous. Très puissant ce logiciel ! Mais il suffit de lui demander ce que vous voulez vraiment

```
>> expand(%);
```

Il n'est pas la peine de rappeler aux brillants anglicistes que vous êtes la signification du terme `expand`.
Nous verrons d'autres fonctions du même type au moment de survoler les outils algébriques.

II.5. Résolution d'équations

L'outil général est `solve`

```
>> solve(x^2-4*x+3=0,x); solve(x^2-x+3=0,x);
```

C'est assez classique et on retrouve cet outil sur les calculatrices de base. Ce qui l'est moins, c'est la capacité de résoudre des équations dépendant de paramètres.

```
>> S:=solve(x^2 - s*x^2+p=0,x):
```

Vous avez remarqué que les solutions des premiers exemples sont affichées entre accolades. Interrogeons MuPAD :

```
>> type(solve(x^2-4*x+3=0,x));
```

La réponse `DOM_SET` nous indique qu'il s'agit d'un ensemble. Nous verrons dans le paragraphe suivant consacré au calcul de 7! que ceci a une grande importance. Pour en revenir à notre équation du deuxième degré, l'ensemble des solutions comportera deux éléments, ou plutôt opérandes en langage MuPAD. Nous pouvons demander à notre logiciel préféré le premier élément de cet ensemble grâce à la commande

```
>> op(S,1);
```

qui affiche le premier opérande de l'ensemble `S`.
Nous pouvons être également amenés à résoudre des équations non pas sur $\mathbb{C}$ mais sur un intervalle donné. Par exemple, si nous voulons résoudre l'équation $x^2 = 4$ sur $\mathbb{R}^+$, nous indiquerons à MuPAD que notre x doit être positif

```
>> assume(x>0);
```

```
>> solve(x^2=4,x);
```

On peut sinon utiliser les ensembles habituels avec les notations suivantes

$\mathbb{N}$	`NonNegInt`	$\mathbb{N}^*$	`PosInt`	$-\mathbb{N}^*$	`NegInt`
$\mathbb{Z}$	`Integer`	$2\mathbb{Z}$	`Even`	$2\mathbb{Z}+1$	`Odd`
$\mathbb{Q}$	`Rational`	$\mathbb{Q}^{*+}$	`PosRat`	$\mathbb{Q}^{*-}$	`NegRat`
$\mathbb{R}$	`Real`	$\mathbb{R}^+$	`NonNegative`	$\mathbb{Q}^+$	`NonNegRat`
$\mathbb{R}*+$	`Positive`	$\mathbb{R}*-$	`Negative`	$\mathbb{C}^*$	`NonZero`
$i\mathbb{R}^*$	`Imaginary`	$i\mathbb{Z}^*$	`IntImaginary`	$\mathbb{P}$	`Prime`

```
>> assume(x in Type::Positive): solve(x^2-4=0,x);
```

On peut résoudre également des systèmes linéaires

```
>> reset():solve({x+2*y+a*z=-1,a*x+y+z=2,2*x+a*y-z=3},{x,y,z});
```

Il existe des outils d'algèbre linéaire plus adaptés que nous verrons plus tard.
On peut résoudre des inéquations

```
>> solve(x^2-2*x-4>=2*x,x);
```

des équations trigonométriques

```
>> solve(cos(x)=sin(x),x);
```

À vous d'imaginer d'autres situations...
Un « étudiant » taquin m'a par exemple proposé

```
>> solve(cosh(x)=I*sinh(x),x);
```

II.6. Simplifications et ré-écritures

Voici un petit test booléen qui utilise la commande `is` et ici l'égalité (à distinguer de l'affectation `:=`)

```
>> is(1+1=2);
```

Continuons

```
>> is((x-1)*(x+1)=x^2-1);
```

Tu parles d'un logiciel de calcul formel...
En fait, MuPAD ne comparera deux polynômes que s'ils sont sous forme développée. Les choses se compliquent s'il s'agit de polynômes à plusieurs indéterminées et dépendant de paramètres. Nous verrons ça plus tard...
Observons plutôt

```
>> y^2 +a*(x+z)*(x-z)+(x^2-y^2);
```

```
>> x+a*(x+z)-(a-1)*(x+z);
```

Ces problèmes viennent du codage informatique de certaines expressions. Nous n'entrerons pas dans les détails.
Voyons plutôt quelques commandes utiles. Nous avons déjà vu `expand` qui développe une expression et qui peut être utile ici

```
>> expand(%);
```

```
>> expand((x+a+b)^3);
```

Pour nous, x est de manière conventionnelle la variable et a et b des paramètres. Ce n'est pas aussi évident pour MuPAD. Il suffit de lui expliquer que nous voulons privilégier x grâce à la commande `collect`

```
>> collect(%,x);
```

Notons, tant que nous y sommes que la commande `expand` a d'autres talents cachés

```
>> expand(cos(a+b)); expand(sinh(3*x));
```

En fait, ce n'est pas une surprise : MuPAD développe des formules du binôme en exponentielle.
Il existe parallèlement `factor` qui se comprend bien. Explorez à l'aide de l'aide la commande `combine` aux multiples talents.

```
>> combine(arctan(2*x)+arctan(y)+arctan(z),arctan);
```

Dans le même ordre d'idée, `partfrac` décompose une fraction en éléments simples.
Notez enfin que la commande `simplify` peut parfois s'avérer utile car elle combine plusieurs des fonctions précédentes.
Il y a beaucoup à dire sur les polynômes car de nombreuses fonctions utiles existent, par exemple `coeff, degreevec, ground, lcoeff, ldegree, lmonomial, lterm, monomials, nterms, nthcoeff, nthmonomial, nthterm, poly, poly2list, tcoeff,...`. En arithmétique des polynômes aussi `gcd,lcm,gcdex,divide`.
Ouvrons une petite parenthèse sur `factor` : tout le monde sait que la factorisation de X^2-2 ne sera pas la même sur $\mathbb{Q}[X]$ et sur $\mathbb{R}[X]$. MuPAD, lui, factorise sur $\mathbb{Q}[X]$, parfois sur $\mathbb{Q}[a,X]$, sachez-le.

```
>> factor(X^2-2);
```

Pour ruser, on peut utiliser `solve`

```
>> solve(x^2-2=0,x);
```

Si nous ne pouvons pas factoriser dans $\mathbb{R}[X]$, on peut le faire dans $\mathbb{Z}[X]$ par exemple en changeant le « domaine ».
Nous n'en parlerons pas tout de suite...

II.7. Nombres complexes

Nous nous contenterons d'un rapide survol de commandes qui se comprennent d'elles-mêmes.
La vedette du paragraphe est le nombre `I` qui vérifie `I*I=-1`
Observez

```
>> expand((3+4*I)^2); Re(3+4*I); Im(3+4*I); conjugate(3+4*I); abs(3+4*I);
```

```
>> exp(I*PI/3); rectform(exp(I*PI/3)*(1+I)^2); arg(6+2*I*sqrt(3));
```

```
>> float(exp(I*PI/3))
```

II.8. Calculs en analyse

Les fonctions usuelles sont notées selon les conventions internationales et se comprennent bien

```
>> cos(PI); sin(PI/4)^2; tan(-x); arctan(tan(32*PI)); arctan(tan(a));
```

Un petit rappel :

```
>> assume(-PI/2<a<PI/2): arctan(tan(a));
```

```
>> E-exp(1); ln(32*E); expand(ln(32*E)); float(ln(32*E))
```

Pour être complet, sachez que MuPAD utilise le logarithme complexe $z \mapsto \ln_{\mathbb{R}} |z| + i \arg z$.

```
>> ln(I);
```

C'est pourquoi on observe

```
>> exp(ln(x)); ln(exp(x));
```

```
>> log(2,8); log(10,100000);
```

Puisqu'on en parle, la commande `sqrt` est aussi définie sur $\mathbb{C}$

```
>> sqrt(32); sqrt(-32);
```

et MuPAD fait mieux que certains élèves de lycée

```
>> (u^(1/4))^4; (u^4)^(1/4);
```

Quelques commandes utiles :

```
>> abs(-32); floor(-1.2); frac(-1.2); sign(-1.2); trunc(-1.2); round(1.3); round(1.6); max(1,2,3);
min(1,2,3);
```

Enfin, nous pouvons calculer sommes et produits

```
>> sum(k,k=0..32); sum(k, k=0..n); partfrac(%); sum(1/n^2, n=1..infinity); product((1-1/k),k=2..32);
```

Ici, la notation `0..32` symbolise l'intervalle entier $[\![0,32]\!]$.

II.9. Limites

Pour les limites, pas de grand mystère

```
>>  limit(sin(x)/x,x=0); limit(x/exp(x),x=infinity); limit(x/exp(x),x=-infinity);
```

```
>> limit(sin(x)/x,x=infinity); limit((1-1/n)^n, n=infinity)
```

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

Pour le dernier calcul, il faut être plus précis

```
>> limit(1/x,x=0,Left); limit(tan(x),x=PI/2,Right);
```

Pour information, si vous avez le courage, vous pouvez vérifier dans la librairie que MuPAD calcule les limites à partir de développements en séries.
Vous aurez sans doute remarqué que MuPAD attend une expression en argument de la commande `limit` et non pas une fonction.
Il faudra bien faire la différence entre

```
>> f:= x+ln(x);
```

où `f` est une expression et

```
>> g:=x->x+ln(x);
```

où `g` est une fonction. En effet

```
>> f(1); g(1); f(2); g(2); x:=2: f; f(2); g(E);
```

MuPAD peut transformer une expression en fonction à l'aide de la commande `fp::unapply(expr,var)`

```
>> h:=fp::unapply(f,x); h(32);
```

```
>> limit(g(x),x=0);
```

Pourquoi ?

```
>> g(x); delete(x): g(x); limit(g(x),x=0); limit(g(x),x=0,Right);
```

Après les limites, la continuité, ou plutôt la discontinuité avec la commande `discont(f(x),x)`

```
>> discont(1/sin(x),x); discont(1/sin(x), x=-10..10);
```

Vous aurez remarqué que `-10..10` représente l'intervalle réel $[-10,10]$.

II.10. Dérivation

Là encore, il faudra distinguer expression et fonction, la commande n'étant pas la même

```
>> D(tan); D(tan)(x); diff(tan(x),x);
```

Pour les dérivées successives

```
>> diff(tan(x),x,x); diff(tan(x),x$5); collect(%,tan(x)); (D@5)(tan);
```

Et si l'on a oublié certaines formules

```
>> D(u*v); D(u/v); D(u@v);
```

sachant que `@` est l'opérateur de composition de fonctions.
On a accés de la même manière aux dérivées partielles

```
>> diff(x^2*y,x);diff(x^2*y,y);diff(x^2*y,x,x);diff(x^2*y,y,x);
```

Enfin, dans le cas de fonctions d'une seule variable, on peut utiliser l'apostrophe `'`

```
>> tan'(x);
```

II.11. Primitives, calcul intégral

La commande-clef est `int`

```
>> int(ln(x),x); int(1/sin(t),t);
```

MuPAD n'est pas omniscient

```
>> int(exp(x^3),x);
```

On peut calculer des intégrales sur des segments

```
>> int(1/(1+t^2), t=0..1); int(exp(x^3),x=0..1); float(%);
```

```
>> int(ln(t^2),t=-1..1); int(ln(t),t=0..1);
```

ou sur des intervalles non bornés

```
>> int(1/t^2,t=1..infinity); int(1/t,t=-1..1);
```

Notez que MuPAD calcule les intégrales au sens des intégrales impropres et non pas au sens de Lebesgue

```
>> int(sin(t)/t,t=0..infinity);
```

On peut calculer des intégrations par parties quand MuPAD semble bloqué ou effectuer des changements de variables. Ces commandes ne sont pas dans la librairie standard. Il faut aller les charger dans la librairie `intlib` en tapant

```
>> intlib::byparts(int(expression,variable),ordre)
```

ou, si on est amené à utiliser cette librairie à plusieurs reprises, on la charge en tapant

```
>> export(intlib);
```

Enfin, on peut calculer des intégrales multiples

```
>> int(int(2*r*cos(t)^2,t=0..2*PI),r=0..R);
```

II.12. Développements limités et généralisés

On peut utiliser `taylor`

```
>> taylor(tan(x),x=0,15); taylor(1/(1+x^2),x=-2,15); taylor(f(x),x=0,5);
```

Notez que MuPAD utilise des grands O.
On peut effectuer des opérations sur les développements

```
>> taylor(sin(x),x=0,15)/taylor(cos(x),x=0,15);
```

effectuer des développements généralisés

```
>> taylor(sqrt(x+sqrt(x^2+1))-sqrt(x+sqrt(x^2-1)),x=infinity,5);
```

Si vous n'avez besoin que de la partie polynomiale du développement, utilisez la commande `expr`

```
>> expr(taylor(tan(x),x=0,9));
```

II.13. Équations différentielles

Nous ne ferons qu'effleurer les possibilités de MuPAD en ce domaine. La résolution se fait en deux étapes qu'on peut d'ailleurs combiner.
Tout d'abord, il s'agit de saisir l'EDO à l'aide de la commande

`ode({équation,CondInit},FonctionInconnue(Variable)`

On peut le rentrer de manière naturelle

```
>> ode(y''(t)+4*y(t)=sin(t),y(t));
```

et MuPAD traduit sous la forme $f(y'(t),y(t),t) = 0$

```
>> ode(- sin(t) + 4 y(t) + diff(y(t), t, t), y(t))
```

Il ne reste plus qu'à résoudre à l'aide de la commande `solve`

```
>> solve(%);
```

Nous nous contenterons de cette approche algébrique.

II.14. Les nombres entiers

Ici encore, nous nous contenterons d'un survol. Des activités spécifiques sont proposées dans les chapitres suivants.
Les commandes de base sont `a div b` et `a mod b` qui renvoient respectivement le quotient et le reste de la division euclidienne de `a` par `b`.
Il y a aussi `gcd(a,b)` et `lcm(a,b)` qui donnent respectivement le pgcd et le ppcm des entiers `a` et `b`.
On peut obtenir la factorisation en produit de facteurs premiers à l'aide de `factor(n)`

```
>> factor(123456789);
```

On peut tester si un entier est premier

```
>> isprime(123456789); isprime(3803);
```

Il y a aussi

```
>> nextprime(123456789);
```

Il y a surtout un grand nombre de fonctions dans la librairie `numlib` que je vous conseille d'explorer.

II.15. Algèbre linéaire

Ce domaine est abondamment exploré par MuPAD. Chacun pourra l'approfondir selon ses besoins. Donnons quelques lignes générales.
Tout d'abord, avant de construire des matrices, MuPAD a besoin de savoir sur quel anneau vous voulez travailler. Par exemple, en rentrant

```
>> Mq:=Dom::Matrix(Dom::Rational);
```

vous indiquez que vous allez travailler avec des coefficients rationnels, la dimension de votre espace n'entrant pas en ligne de compte.

Nous verrons que faire du `Mq` ensuite. Énumérons d'abord différents domaines. L'anneau de base peut être

- `Dom::Integer` pour $\mathbb{Z}$
- `Dom::IntegerMod(13)` pour $\mathbb{Z}/13\mathbb{Z}$
- `Dom::Rational` pour $\mathbb{Q}$
- `Dom::Real` pour des réels sans approximation
- `Dom::Float` pour les approximations numériques des réels
- `Dom::Complex` pour les complexes
- `Dom::DistributedPolynomial([x],Dom::Integer)` pour travailler dans l'anneau des polynômes à coefficients entiers.

Pour travailler dans un anneau quelconque, et si l'on veut utiliser des paramètres, il suffira de rentrer

```
>> M:=Dom::Matrix()
```

Maintenant, il reste à renter les matrices.
On peut la rentrer « à la main » par lignes en rappelant le domaine dans lequel on travaille.
Ici, nous travaillerons par exemple dans le domaine `Mq` précédemment défini.

```
>> A:=Mq([[1,2,3],[4,5,6],[7,8,9]]);
```

```
>> B:=Mq([[1,0,1],[0,1,0],[1,1,2]]);
```

puis effectuer les opérations usuelles

```
>> A+B; A*B; 1/B; 1/A; B^10; A*B-B*A;
```

On peut créer des matrices de tailles diverses

```
>> a:=Mq(4,1,[1,2,3,4]);
```

Les matrices sont en fait des listes[1] de listes, donc en particuliers des listes. On les traitera donc comme telles. Nous étudierons plus précisément les listes au chapitre suivant, mais sachez que vous pouvez appliquer une fonction à tous les coefficients d'une matrice à l'aide des commandes `map(A,f)` et `zip(A,B,f)`

```
>> M:=Dom::Matrix(); C:=M([[1,0,2],[0,3,0],[4,5,6]]); F:=map(C,cos); G:=zip(C,F,(x,y)->2*x^2-y);
```

Pour la suite, nous aurons besoin de charger la volumineuse librairie `linalg`

```
>> export(linalg);
```

Impossible de parler de toutes les commandes. Faites-vous une idée en tapant

```
>> ?linalg;
```

On peut au moins parler du déterminant, du polynôme caractéristique

```
>> det(A); det(B); charpoly(A,X);
```

des valeurs propres

```
>> eigenvalues(F); # pour rigoler...#
```

Accordons quelques instants aux résolutions de systèmes linéaires. Plusieurs méthodes d'entrée sont possibles. Tout d'abord, nous pouvons rentrer les systèmes de manière naturelle grâce à la commande `expr2Matrix(liste des équations, liste des variables)` (en anglais expréchione tou matrix) qui transforme comme son nom l'indique une expression en matrice. On peut rentrer en troisième argument optionnel le domaine des coefficients, par exemple `Dom::Integer`. Ainsi

```
>> S:=expr2Matrix([x+2*y+3*z=4,y+7*z=6,x+y+z=3],[x,y,z]);
```

puis

```
>> matlinsolve(S);
```

affiche le triplet solution, s'il existe.
Nous aurions pu sinon rentrer une matrice carrée représentant le système linéaire et une matrice colonne afin de résoudre l'équation $AX = B$.

```
>> A:=M([[1,2,3],[0,1,7],[1,1,1]]); B:=M(3,1,[4,6,3]); matlinsolve(A,B);
```

1. Suite d'éléments ordonnés écrits entre crochets

II.16. Et le reste

Il est impossible d'être exhaustif. Nous n'avons exploré qu'à peine 1% des commandes de MuPAD. Explorez l'aide, les librairies, pour vous rendre compte que MuPAD est très riche et contient des modules exploitables bien au-delà du premier cycle universitaire. Notons simplement que nous n'avons pas évoqué la librairie `stats` pourtant très importante que les spécialistes pourront décortiquer.

III. Quelques exercices de MAO (maths assistées par ordinateur)

Exercice 1 Noyau, image

Qu'est-ce que l'endomorphisme de $\mathbb{R}^2$ d'expression analytique canonique

$$\begin{cases} x' &= -\frac{1}{5}x - \frac{2}{5}y \\ y' &= \frac{3}{5}x + \frac{6}{5}y \end{cases}$$

Vous aurez besoin de déterminer la matrice A associée, le noyau de A grâce à `nullspace(A)` qui en renvoie une base, l'image avec `basis(A)`. Ensuite, à vous d'essayer des petits calculs sur A pour deviner sa nature.

Exercice 2 Puissances de matrices

1) Soit $A = \frac{1}{3}\begin{pmatrix} 0 & -2 & -2 \\ 2 & 0 & -1 \\ 2 & 1 & 0 \end{pmatrix}$.
   a) Calculer A^2,A^3 et A^4.
   b) Montrer que $\{A,A^2,A^3,A^4\}$ est un groupe pour le produit matriciel de $\mathcal{M}_3(\mathbb{R})$.
2) Pour chacune des matrices A suivantes, calculer A^n pour tout $n \in \mathbb{N}$.
   a) $\begin{pmatrix} 1 & a \\ 0 & 1 \end{pmatrix}$ avec $a \in \mathbb{R}$.
   b) La matrice J_p de $\mathcal{M}_p(\mathbb{R})$ dont tous les coefficients sont égaux à 1.

Exercice 3 Inverse d'une matrice

Soit $A = \begin{pmatrix} 1 & 0 & a \\ 1 & a & -1 \\ a & 0 & 1 \end{pmatrix}$. Pour quelles valeurs de a, la matrice A est-elle inversible. Calculez alors l'invesre de A.

Exercice 4 EDO

Testez MuPAD sur les équations suivantes

1) $(x+1)y' - xy = 0$
2) $(1+x^2)y' + xy = 3x^3 + 3x$ $\qquad$ $(1+x^2)y' + xy = 1$
3) $xy' + 2y = x/(1+x^2)$
4) $y' = |y|$
5) $\sqrt{1-x^2}y' - y^2 - 1 = 0$
6) $y'' - 3y' + 2y = xe^x + \sin(x)$
7) $x^2y'' + 3xy' + y = 0$

Exercice 5 Division euclidienne de polynômes

1) On voudrait conjecturer à l'aide de MuPAD le reste de la division euclidienne de A par B sachant que $A = ((\sin t)X + \cos t)^n$ par $B = X - 1$.
   Déterminez ce reste pour des valeurs particulières de n. Conjecture? Prouvez votre conjecture...
   Reprenez le même exercice avec $B = X^2 + 1$. Pour vous aider dans votre conjecture, vous pourrez utiliser la fonction `combine(expr,sincos)` qui permet de linéariser une expression trigonométrique et la fonction `coeff(p,x,n)` qui permet d'isoler le coefficient du polynôme p en x^n.
2) Montrez que $A = X + 3X^2 + 5X^3 + 5X^4 + 3X^5 + X^6$ divise $B = (X+1)^{6n+1} - X^{6n+1} - 1$. Vous pourrez utilisez au choix la fonction `solve(equation,inconnue)` ou `factor(polynome)` pour trouver les racines de A.

Exercice 6 Avec sommation

Trouvez la valeur de $\displaystyle\sum_{n=1}^{+\infty} \frac{1}{1^2 + 2^2 + \cdots + n^2}$

Exercice 7 Trigo

Résolvez l'équation $\arccos(x) = \arcsin(2x)$

Exercice 8 Études de fonctions

1) Étudiez la fonction $x \mapsto x \arctan(x)$ en portant une attention particulière au comportement asymptotique.
2) Étudiez $x \mapsto \sqrt[5]{(x^2-1)^2(x+11/3)}$.

Exercice 9 Ipp

On pose $I_n = \int_0^{\pi/2} \sin^n t \, \mathrm{d}t$. Exprimez I_{n+2} en fonction de I_n.

Exercice 10 Système

Résoudre $\begin{cases} x + ay + z + t &= 1 \\ x + y + az + t &= b \\ x + y + z + at &= 1 \end{cases}$

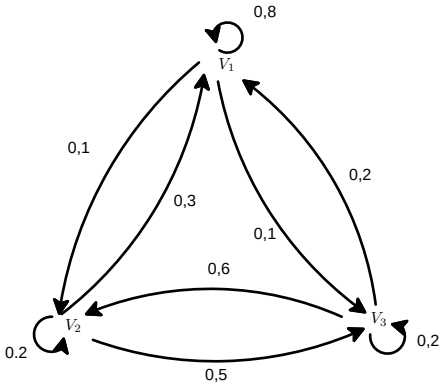
Exercice 11 Mini-chaînes de Markov...

Aidons la mafia

Les chaînes de Markov sont issues de la théorie des probabilités et utilisent des outils d'algèbre linéaire qui nous intéressent aujourd'hui. Elles permettent de simuler des phénomènes aléatoires qui évoluent au cours du temps. Nous allons les découvrir à travers l'étude d'un exemple simple.
Zlot, Brzxxz et Morzgniouf sont trois villes situées respectivement en Syldavie, Bordurie et Bouzoukstan. Des traficants de photos dédicacées du groupe ABBA prennent leur marchandise le matin dans n'importe laquelle de ces villes pour l'apporter le soir dans n'importe quelle autre. On notera pour simplifier V_1, V_2 et V_3 ces villes et p_{ij} la probabilité qu'une marchandise prise le matin dans la ville V_i soit rendue le soir dans la ville V_j. La matrice $(p_{ij})_{1 \leqslant i \leqslant 31 \leqslant j \leqslant 3}$ est appelée *matrice de transition* de la chaîne de Markov. Que s'attend-on à observer sur les colonnes d'une matrice de transition?
Supposons que P soit connue et vaille

$$P = \begin{pmatrix} 0{,}8 & 0{,}3 & 0{,}2 \\ 0{,}1 & 0{,}2 & 0{,}6 \\ 0{,}1 & 0{,}5 & 0{,}2 \end{pmatrix}$$

Les traficants se promenant de ville en ville, il peut être utile de visualiser leurs déplacements par le *diagramme de transition* suivant



On notera $x_i^{(k)}$ la proportion de traficants qui se trouvent au matin du jour k dans la ville V_i. En probabilités, on appelle *vecteur d'état* tout élément $(x_1, \cdots, x_n)$ de $\mathbb{R}^{+n}$ tel que $x_1 + \cdots + x_n = 1$.
Ainsi, $x^{(k)} = \left(x_1^{(k)}, x_2^{(k)}, x_3^{(k)}\right)$ est un vecteur d'état.
On montre que les vecteurs d'état de la chaîne sont liés par la relation

$$x^{(k)} = P \cdot x^{(k-1)}$$

et donc

$$x^{(k)} = P^k \cdot x^{(0)}$$

Supposons que le chef de la mafia locale dispose de 1000 traficants qui partent tous le matin du jour 0 de la ville de Zlot. Quelle sera la proportion de traficants dans chacune des villes au bout d'une semaine? d'un mois? d'un an?

Le parrain voudrait que la proportion moyenne de traficants soit stable d'un jour sur l'autre. Il recherche donc les vecteurs d'état x vérifiant l'équation $P \cdot x = x$. Vous apprendrez après l'été à résoudre de manière systématique ce genre de problème. Nous allons pour l'heure nous débrouiller sans appui théorique mais avec Maple et la fonction `linsolve(A,B)` qui permet de résoudre les équations du type AX=B avec A une matrice, B un vecteur connu et X le vecteur inconnu. Comment procéder?

Météo

À Morzgniouf, les jours sont soit secs, soit pluvieux. On désigne par E_1 l'état sec et E_2 l'état pluvieux et par p_{ij} la probabilité qu'un jour soit dans l'état E_i sachant que le jour précédent était dans l'état E_j. Les scientifiques Bouzouks ayant observé les phénomènes météorologiques des trent-deux dernières années à Morzgiouf ont établi la matrice de transition suivante

$$P = \begin{pmatrix} 0{,}750 & 0{,}338 \\ 0{,}250 & 0{,}662 \end{pmatrix}$$

Sachant qu'il fait beau aujourd'hui, quelle est la probabilité qu'il pleuve dans dix jours?

Il est temps de passer aux choses sérieuses avec cette activité de niveau Terminale.

IV. Initiation à la programmation : 7 manières de calculer 7!

1) Bien sûr, on peut commencer par faire

   ```
   >> 7!;
   ```

   mais le calcul de 7! n'est qu'un prétexte pour découvrir sur un exemple simple la syntaxe de programmation MuPAD.

2) On pourrait alors taper

   ```
   >> 1*2*3*4*5*6*7;
   ```

   mais les choses pourraient se compliquer au moment de calculer 3232!

3) Nous allons donc utiliser une boucle *for* avec indice numérique : nous partons de 1, puis nous multiplions par 2, puis nous multiplions le produit précédent par 3, etc. Cela donne

   ```
   >> p:=1: for k from 1 to 7 step 1 do p:=p*k end_for;
   ```

   Ce programme n'est pas optimum. En fait, dans une boucle `for`, l'instruction `step` est facultative : elle vaut par défaut 1. Ici, on peut se contenter de

   ```
   >> p:=1 for k from 1 to 7 do p:=p*k end_for;
   ```

4) Nous aurions pu créer la liste des entiers de 1 à 7 puis utiliser une boucle *for* indexée cette fois par les éléments de la liste. C'est plus compliqué, mais cela nous permet de découvrir comment MuPAD traite les listes. Il faudra bien distinguer

   ▷ les **ensembles** (`set`) qui sont des collections *non ordonnées* d'expressions toutes différentes séparées par des virgules et encadrées d'accolades.

   ```
   >> ens1:={2,4,1}; ens2:={2,5,8,5};
   ```

   On peut effectuer les opérations usuelles sur les ensembles

   ```
   >> ens1 intersect ens2; ens1 union ens2; ens1 minus ens2;
   ```

   l'ensemble vide se note

   ```
   >> { }
   ```

   ▷ les **suites** (`sequence`) qui sont des collections *ordonnées* d'expressions, différentes ou non, séparées par des virgules et encadrées ou non par des parenthèses.

   ```
   >> 5,7,5,1,2,3;
   ```

   On peut aussi utiliser l' opérateur `$` pour des suites définies par une formule explicite

   ```
   >> p^2 $ p=1..5;

   >> m $ 5;

   >> j^2 $ j=1..n;
   ```

   pose problème.

   ▷ les **listes** (`list`) qui sont des collections *ordonnées* d'expressions séparées par des virgules et encadrées par des crochets. La différence, c'est qu'une suite, en tant que juxtaposition d'expressions, est en quelque sorte « en lecture seule », alors qu'une liste est une expression en elle-même et pourra donc « subir » des opérations algébriques.
   Notez au passage quelques fonctions utiles

   ```
   >> s1:=i $ i=-2..2; s2:=a,b,c,d,e;

   >> l1:=[s1]; l2:=[s2]; # une liste est une suite entre crochets#

   >> nops(l2); # nombre d'opérandes #

   >> l2[3]; # extrait le 3ème opérande #

   >> l2[3..5];

   >> select (l1,x->x>0)
   ```

```
>> subs(l2,b=32); subsop(l1,2=z); subsop(l2,5=null()); # pour substituer ou supprimer
un opérande, null() étant la liste vide #

>> map(l2,cos);

>> zip(l1,l2,(x,y)->x+y);

>> l3:=l1.l2; # pour concaténer#
```

Cela donne

```
>> reset();

>> l:=[k $ k=1..7]: p:=1: for i from 1 to 7 do p:=p*l[i] end_for: p;
```

5) Nous pouvons utiliser une boucle `while`

```
>> reset();

>> p:=1: k:=1: while k<7 do k:=k+1: p:=p*k end_while: p;
```

6) C'est bien beau, mais que ferons-nous quand il faudra calculer 32! ou 3232! et tous les autres ? Il faudrait créer un programme (on dira une **procédure**) qui donne $n!$ pour tout entier naturel n.

```
>> reset(); # je ne vous le dirai plus #

>> fac:=proc(n)
local p,k; # nous aurons besoin de variables locales i.e. internes à la procédure, comme des
variables d'intégrations à l'intérieur d'une intégrale #
begin
p:=1;
for k to n do p:=p*k end_for;
end_proc: # termine la procédure #

>> fac(32);
```

7) Le meilleur pour la fin : la **procédure récursive**, qui s'appelle elle-même

```
>> factr:=proc(n)
begin
if n=0 then 1 # une boucle if...then...else pour régler le cas de 0! #
else n*factr(n-1)
end_if; # symbolise la fin de la boucle #
end_proc:

>> factr(32);
```

En fait, ce mécanisme correspond à une suite numérique qui s'écrirait mathématiquement $u_n = n \times u_{n-1}$.

V. Des petits exercices de programmation

V.1. Les énoncés

Exercice 1

Déterminez une procédure `E:=proc(x)` qui, à un réel positif x, associe sa partie entière.

Exercice 2

Déterminez une procédure `ab:=proc(x)` qui, à un réel x, associe sa valeur absolue.

Exercice 3

Déterminez une procédure permettant de calculer la moyenne des éléments d'une famille de nombres réels.

Exercice 4

Pour $n \in \mathbb{N}^*$, on considère $a_n = 1 + 1/n$ et $b_n = a_n^{a_n^{\cdot^{\cdot^{\cdot^{a_n}}}}}$, avec « n fois a_n ».
Par exemple, $b_2 = (3/2)^{3/2}$.

1) Déterminez une procédure qui calcule b_n à partir de n.
2) On admet que la suite (b_n) est décroissante et converge vers 1. Déterminez une instruction qui permette de déterminer le plus petit entier n tel que $b_n \leqslant 1{,}001$.

Exercice 5

Vous savez peut-être que la suite $(S_n)_{n\in\mathbb{N}}$ de terme général

$$S_n = \sum_{k=0}^{n} \frac{1}{k!}$$

est croissante et converge vers e. Nous l'admettrons dans cet exercice.

1) Déterminez une procédure `S=proc(n)` qui, à un entier naturel n, associe S_n. N'oubliez pas les gages habituels : vous n'utiliserez ni la fonction prédéfinie `sum`, ni les procédures calulant $n!$ vues précédemment.
2) Déterminez une procédure `seuil:=proc(p)` qui, à un entier naturel p, associe le plus petit entier naturel n tel que $|S_n - e| \leqslant 10^{-p}$. Vous aurez besoin de savoir que e se dit `exp(1)` en Maple.

Exercice 6

Déterminez une procédure `sol:=proc(a,b,c)` qui, à une équation $ax^2 + bx + c = 0$, associe son ensemble des solutions.

Exercice 7

Déterminez une procédure `test:=proc(l)`, `l` étant une liste d'entiers, qui teste si ses éléments forment une suite croissante.

Exercice 8

Vous connaissez tous la formule du triangle de Pascal. Déterminez donc une procédure récursive `c:=proc(n,p)` qui calcule les coefficients du binôme. Testez avec `c(32,3)` et `c(3,32)`.

V.2. Des solutions

Les solutions proposées sont loin d'être les seules. Vous avez vu par exemple sept méthodes pour calculer 7!, et il en existe d'autres. Et puis, les solutions proposées ne sont pas forcément les meilleures : n'hésitez pas à faire part de vos idées.

Exercice 1

```
>> E:=proc(x) begin local n; n:=0; while n<x do n:=n+1 end_while: n-1; end_proc:
```

Exercice 2

```
>> ab:=proc(x) begin if x>=0 then x else -x end_if; end_proc:
```

Exercice 3

```
>> moy:=proc(l) local s,k; begin s:=0; for k from 1 to nops(l) do s:=s+l[k] end_for: s/nops(l);
end_proc:
```

Notez qu'il faut rentrer une liste en argument, donc entre crochets. Vous remarquerez que nous sommes confrontés à un petit problème technique : nous avons plutôt l'habitude d'utiliser des approximations numériques pour les moyennes. Il existe pour cela la fonction `float(nombre)`. Le nombre de chiffres par défaut est 10. On peut le faire varier grâce à la variable `DIGITS:= nombre`. Remarquez enfin que MuPAD fait la différence entre `1/3` et `1/3.0`. Par exemple, comparez `(1/3+2/3)-1` et `1/3+2/3.0)-1`

Exercice 4

```
>> bn:=proc(n) local a,b,k; begin a:=1+1/n; b:=1; for k from 1 to n do b:=a^b end_for: b end_proc:
```

Le résultat est un peu illisible. On peut utiliser la fonction `simplify(nombre)`, mais le résultat est un peu faiblard. Mieux vaut faire intervenir `float` dans la procédure.

Exercice 5

" ...le plus petit entier tel que..." nous fait penser à une boucle while

```
>> m:=1: while bn(m)>1.001 do m:=m+1 end_while: m;
```

Exercice 6

La ruse est d'introduire deux variables en plus de l'indice : t pour le terme et s pour la somme

```
>> S:=proc(n) local s,t,k; begin
s:=1: t:=1: for k from 1 to n do t:=t/k: s:=s+t
end_for: s end_proc:
```

Par curiosité, essayez

```
>> float(S(32)-exp(1));
```

Comment y remédier ?
Passons à la question suivante :

```
>> DIGITS=1000:
seuil:=proc(s) local n; begin
n:=0: while float(exp(1)-S(n))>float(1/10^s) do n:=n+1 end_while: n; end_proc:
```

Exercice 7

Rien de très difficile mathématiquement parlant, mais cela nous permet de découvrir la fonction elif, contraction de else if qui permet d'accéder à une sous-boucle if.

```
>> sol:=proc(a,b,c) local d,x1,x2: begin
d:=b^2-4*a*c:
if d>0 then x1:=(-b-sqrt(d))/(2*a): x2:=(-b+sqrt(d))/(2*a): elif d=0 then x1:=-b/(2*a): x2:=x1
: else x1:=(-b-I*sqrt(-d))/(2*a): x2:=(-b+I*sqrt(-d))/(2*a) : end_if; if d<>0 then print("Les solutions
sont ".expr2text(x1). "et " .expr2text(x2)): else print ("La solution est ".expr2text(x1)): end_if;
end_proc:
```

Il y a bien sûr plus court

```
>> solv:=proc(a,b,c) begin
solve(a*x^2+b*x+c=0,x); end_proc:
```

VI. Premier exemple d'application en terminale : méthode de Newton et dichotomie

Vous savez de quoi il s'agit, donc je me contente de donner le résultat :
On construit un programme dépendant de la donnée d'une fonction `f`, d'une précision `e` et d'un premier terme `u0`.

```
>> Newton:=proc(f,e,u0)
>>  local un,aun,fp,k;
>>  begin
>>  fp:=D(f);# fp est la dérivée de f
>>  k:=0; # On règle le compteur des itérations à zéro
>> aun:=u0; # l'ancien un vaut au départ u0
>>  un:=u0-f(u0)/fp(u0);# calcul du terme suivant
>>  while abs(un-aun)>e do # tant que la précision n'est pas atteinte, on réitère
>> aun:=un; un:=un-f(un)/fp(un);
>>  k:=k+1;# un tour de plus au compteur
>>  end_while;
>> print(Unquoted,expr2text(float(un))." est la solution trouvée à ".expr2text(e)." près après
".expr2text(k)." itérations");
>> end_proc:
```

On regarde ce que cela donne

```
>> Newton(x->x^2-2,10^(-4),2);
```

Pour la dichotomie, la procédure dépend toujours de `f` et `e` et aussi des bornes `a` et `b` de l'intervalle

```
>> dicho:=proc(f,e,a,b)
>>  local aa,bb,k;
>>  begin
>>  aa:=a;bb:=b; # les anciennes bornes valent au départ a et b
>>  k:=0; # le compteur est mis à zéro
>> while (bb-aa)>e do
>> if sign((f((bb+aa)/2)))=sign((f(bb))) then bb:=((aa+bb)/2):
>> else aa:=((aa+bb)/2):
>> end_if:
>>  k:=k+1;
>>  end_while;
>>  print(expr2text(float((bb+aa)/2))." est la solution trouvée à ".expr2text(e)." près après
".expr2text(k)." itérations");
>>  end_proc:
```

Il ne reste plus qu'à appliquer et à comparer

```
>> dicho(x->x^2-2,10^(-4),1,2);
```

VII. Deuxième exemple d'application en terminale : arithmétique

Voici des extraits d'activités que j'ai proposées en terminale cette année.

VII.1. Tests et cribles

Nous pouvons tester si un nombre est premier :

```
>> test1:=proc(n)
>>  local L;
>>  begin
>>  L:=[]; # On crée une liste, vide au départ, pour y placer les éventuels diviseurs de n#
>>  for i from 2 to floor(sqrt(n)) do # floor signifie partie entière #
>>  if n mod i = 0 then L:=L.[i]: end_if # si le reste est nul, on ajoute i à la liste #
>>  end_for;
>>  if L=[] then print(expr2text(n). " est premier" ); # si L est vide, n est premier #
>>  else print(expr2text(n). " n'est pas premier" ); end_if
>>  end_proc:
>> test1(1321235158);
"1321235158 n'est pas premier"
>> test1(101!+1);
Error: Computation aborted; during evaluation of 'test1'
```

Ce test a donc des limites et nécessitera de nombreuses améliorations.

Mais nous pouvons faire mieux : nous pouvons obtenir rapidement une liste des *petits* entiers premiers inférieurs à un nombre donné n : on écrit les entiers de 2 à n et on barre les multiples des nombres premiers inférieurs à $\sqrt{n}$. Les entiers restant sont premiers. Nous ne sommes pas les premiers à avoir eu cette idée, Eratosthène l'a eu avant nous en 240 avant JiCé.

```
>> Erato:=proc(n)
>>  local x,i,y,P;
>>  begin
>>  x := array(1..n); # On crée une liste de n nombres #
>>  for i from 1 to n do
>>  x[i]:=1; #ces nombres valent tous 1 au départ#
>>  end_for;
>> x[1]:=0; #1 n'est pas premier donc on le "raye" en lui associant 0#
>>  for y from 2 to floor(sqrt(n)) do #on teste les entiers jusqu'à Vn (floor = partie entière)#
>> if(x[y]=1) then #si le yème nombre n'est pas barré#
>> for i from 2 to floor(n/y) do
>> x[i*y]:=0; #on barre tous ses multiples#
>> end_for;
>> end_if;
```

```
>> end_for;
>> P:=[]; #on crée une liste vide#
>> for i from 1 to n do
>> if(x[i]=1) then P:= P.[i]; #on ajoute tous les entiers non barrés#
>> end_if;
>> end_for;
>> P; #on affiche la liste des entiers restant : ils sont premiers#
>> end_proc:
```

Cela nous donne :

```
>> Erato(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]

VII.2. Décomposition des entiers en produit de nombres premiers

Pour obtenir un programme donnant la décomposition d'un entier donné, nous allons utiliser la procédure `Erato` mise au point précédemment qui renvoie la liste des entiers premiers inférieurs à un nombre donné.

```
>> decompo:=proc(n)
>>  local L,l,D,N;
>> begin
>>  D:=[]: L:=Erato(n): l:=op(L): N:=n:
```

l est l'ensemble des nombres premiers inférieurs à n

```
>>  while not(contains(l,N)) do
```

tant que N n'est pas premier

```
>>  for k from 1 to nops(L) do
```

nops(L)= nb d'éléments de L

```
>>  if N mod L[k]=0 then D:=D.[L[k]]; # si le kième premier divise N, on le rajoute à la liste#
>>  N:=N div L[k]; # on divise N par ce nb premier #
>>  break; # on recommence à la 5ème ligne des fois que le kième premier divise encore n#
>>  end_if; end_for; end_while;
>>  D:=D.[N]; # on n'oublie pas n si n est premier
>>  end_proc:
>> decompo(50);
>>  [2, 5, 5]
```

Pour être honnête, il existe la fonction `factor` qui donne directement la décomposition en produit de facteurs premiers.

```
>> factor(50);
>>  2 5^2
```

VII.3. Petit théorème de Fermat

Nous avons mis au point sur le modèle du crible d'Eratosthène une méthode permettant de tester si un entier est premier ou non. Cela marche assez bien pour des petits nombres, mais cette méthode devient impraticable s'il s'agit de tester un entier d'une centaine de chiffres. Nous allons nous occuper dans cette section de deux problèmes imbriqués : d'une part trouver des méthodes permettant de vérifier rapidement si un nombre est premier et d'autre part réfléchir à d'éventuelles méthodes permettant de « fabriquer » des nombres premiers aussi grands que l'on veut.

« *A thing of beauty is a joy for ever. Its loveliness increases; it will never Pass into nothingness* » écrivait John Keats 217 ans après la naissance de Pierre de Fermat, dont le Petit Théorème demeure un joyau de la théorie des nombres, malgré les innombrables découvertes effectuées depuis dans ce domaine. S'il est qualifié de petit, c'est qu'une conjecture célèbre du même Fermat, restée indémontrée pendant des siècles, s'est accaparée le titre de *Grand Théorème de Fermat*.

$$x^n + y^n = z^n \quad \text{n'a pas de solution dans } \mathbb{N}^3 \quad \text{pour } n > 2$$

Dans la marge d'un manuscrit, Fermat prétendait en avoir trouvé la démonstration mais manquer de place pour l'écrire. Il a pourtant fallu attendre 1995 pour qu'Andrew Wiles le démontre en utilisant des outils surpuissants : l'entêtement d'une multitude de chercheurs a abouti à la démonstration de ce théorème, mais surtout a permis de développer d'importants outils en théorie des nombres. Quant à lui trouver des applications, le problème reste ouvert.

Au contraire, notre petit théorème, bien moins « médiatique » et à la démonstration abordable, a eu de très nombreuses et importantes conséquences : nous allons en étudier quelques-unes.

Mais tout d'abord, comment l'idée de son théorème est venu à l'esprit de l'ami Pierre ?
Au cours de ses nombreuses recherches, Fermat s'est intéressé aux nombres de Mersenne [2] qui sont les nombres de la forme $M_n = 2^n - 1$, avec n un entier premier. On savait déjà que M_{11} était composé

$$2^{11} - 1 = 2047 = 23 \times 89 \ (i)$$

Mais c'est Fermat qui remarqua que

$$2^{23} - 1 = 8388607 = 47 \times 178841 \ (ii)$$

Il sauta aux yeux de Fermat qui vivait parmi les entiers, que $47 = 2 \times 23 + 1$ en observant (ii) ce qui lui mit la puce à l'oreille pour observer que $23 = 2 \times 11 + 1$ dans (i). Qu'en dire ? Que le plus petit diviseur premier d'un nombre composé $2^p + 1$ est de la forme $2p + 1$?
Le problème, c'est que $2^{29} - 1 = 536870911 = 233 \times 1103 \times 2089$, et donc, ça ne marche pas. D'ailleurs, ce n'est pas ce qui attira l'attention de Fermat. Il remarqua en effet que dans (i), non seulement $23 = 2 \times 11 + 1$ mais aussi $89 = 8 \times 11 + 1$. De même dans (ii), $47 = 2 \times 23 + 1$ et $178\,481 = 7\,760 \times 23 + 1$. Il conjectura donc que tout diviseur premier de $2^p - 1$ est congru à 1 modulo p.

Mais l'ami Pierre ne s'arrêta pas là. On a $2^{11} \equiv 1[23]$, mais on a le même résultat avec 2^{22}, 2^{33}, etc. De même, 2^{11}, 2^{22},...,2^{88}, etc. sont tous congrus à 1 modulo 89. Et puis 2^{23}, 2^{46}, etc. sont congrus à 1 modulo 47 et enfin 2^{23},...,$2^{178\,480}$, etc. sont aussi tous congrus à 1 modulo 178 481.

Oui, et alors. Et bien, pour ces quatre nombres premiers, on a $2^m \equiv 1[p]$, mais surtout, dans chaque cas, l'un de ces m est $p - 1$.
En bref, pour les nombres premiers 23, 89, 47 et 178481, on a $2^{p-1} \equiv 1[p]$. Est-ce vrai pour *tous* les entiers ? Nous qui avons un bel ordinateur, menons une petite enquête :

```
>> fer:=proc(a,n) local k;
>> begin
>> for k from 1 to n do
>> [ithprime(k),pmod(a,ithprime(k)-1,ithprime(k))]; # ithprime(k) donne le kème nb premier
>> end_for; end_proc:
```

2. Nous en reparlerons plus abondamment un peu plus loin

```
>> fer(3,15);
```

[[2, 1], [3, 0], [5, 1], [7, 1], [11, 1], [13, 1], [17, 1], [19, 1],
[23, 1], [29, 1], [31, 1], [37, 1], [41, 1], [43, 1], [47, 1]]

Occupons nous maintenant du théorème lui-même, énoncé pour la première fois par Fermat en 1640 dans une lettre adressée au fameux Frère Marin Mersenne.

Théorème -1

Soit p un nombre premier et a un entier non divisible par p. Alors $a^{p-1}-1$ est divisible par p, ou, en d'autres termes

$$a^{p-1} \equiv 1[p]$$

Comme d'habitude, Fermat affirma avoir la démonstration mais ne pas avoir la place de l'écrire à son correspondant [3]

Il existe de très nombreuses démonstrations de ce théorème. Nous en verrons trois, une maintenant, une à titre d'exercice et une comme cas particulier d'un théorème plus général, le théorème d'Euler.

La plus rapide consiste à montrer que $a^{p-1}(p-1)! = a \times 2a \times 3a \times \cdots \times (p-1)a$ est congru à $(p-1)!$ modulo p puis d'en déduire le résultat. Nous aurons besoin d'établir deux petites propriétés [4].

Propriété -1

Soit p un nombre premier et a un entier, alors p divise a OU p et a sont premiers entre eux.

Propriété -2

Soit p un nombre premier et a et b deux entiers. Si p divise ab, alors p divise a ou p divise b.

Considérons donc les multiples successifs de a jusqu'à $(p-1)a$ et appelons r_k le reste de la division de ka par p.

- ▷ Montrons d'abord qu'aucun r_k n'est nul : en effet, si p divisait ka, alors il diviserait a ou k, or il ne divise pas a par hypothèse et il ne divise pas k car $k \leqslant p-1$.
- ▷ Montrons maintenant que ces restes sont deux à deux distincts : s'il existe deux entiers k et k' tels que $r_k = r_{k'}$, alors $ka \equiv k'a[p]$, puis $a(k-k') \equiv 0[p]$, ce qui veut dire que p divise $a(k-k')$. On peut donc conclure comme tout à l'heure.
- ▷ On obtient donc que $a^{p-1}(p-1)! = a \times 2a \times 3a \times \cdots \times (p-1)a \equiv r_1 \times r_2 \times \cdots \times r_{p-1}[p]$. Or les r_k sont $p-1$ entiers distincts compris entre 1 et $p-1$: il s'agit donc exactement des entiers de 1 à $p-1$ à l'ordre près, donc leur produit est égal à $(p-1)!$
  On obtient donc que $a^{p-1}(p-1)! \equiv (p-1)![p]$
- ▷ Pour conclure, on montre facilement que p est premier avec $(p-1)!$. Or p est premier avec $(a^{p-1}-1)\big((p-1)!\big)$ et on utilise une troisième fois le théorème de Gauss pour conclure que p divise $a^{p-1}-1$.

Et maintenant...une petite illustration MuPAD, évidemment...
On utilisera la fonction `n mod d` qui renvoie le reste de la division de `n` par `d`.

```
>> 2^(97-1) mod 97;
```


3. Fermat était-il malhonnête ou avare de son encre?
4. Un petit exercice d'entraînement consiste à les démontrer

Choisissons maintenant un nombre premier quelconque grâce à la fonction `nextprime`

```
>> p:=nextprime(10^20);
```

100000000000000000039
puis un nombre entier quelconque inférieur à 10^{12}

```
>> random();
```

321110693270
calculons a^{p-1} modulo p

```
>> a^(p-1) mod p;
```

Error: Overflow/underflow in arithmetical operation

C'est trop gros : serions-nous donc encore une fois condamnés aux petits nombres ? Eh bien non, il est temps d'utiliser notre cerveau ! Et aussi l'algorithme d'exponentiation rapide vu précédemment.

```
>> pmod:=proc(a,n,p) local B,A;
>> begin
>>  A:=a mod p: B:=1; # Pour simplifier car on travail modulo p
>> while n<>0 do
>>  if n mod 2=0 then A:=(A*A mod p); n:=n/2 # si n pair, on élève au carré et on divise la puissance par 2
>>  else B:=(B*A mod p); n:=n-1 # sinon, on enlève une puissance pour obtenir une puissance paire
>> end_if end_while;
>> B
>>  end_proc:
```

Cette fois-ci on a la réponse, et en plus, instantanément !

```
>> pmod(321110693270,100000000000000000039-1,100000000000000000039);
```

1
Comme quoi, quand on arrive aux limites des possibilités calculatoires d'un ordinateur, rien ne vaut un bon cerveau.

C'est bien joli tout ça, amis à quoi ça peut servir ?
Prenons un grand nombre, assez compliqué pour avoir l'air premier, par exemple

```
>> n:=10^100*random()^3+32!+32*random()+13;
```

n:=1741699500705224901995212664744181760000000000000000000000000000000000000000000 000000000000000000000000
02631308369336935301672419082577229
Est-ce que ce nombre de 136 chiffres est premier ? Si tel est le cas, alors $2^{p-1} \equiv 1\,[p]$ [5]
Jetons donc un coup d'œil

```
>> pmod(2,n-1,n);
```

1092918599640366815513230164517167793897330855635903160391509055400325096371123855972621318700635448744642860847610628382268616793816279
Ouh la la, ça ne fait pas 1 ! Donc n n'est pas premier, et la réponse a été donnée immédiatement, en quelques mili-secondes ! C'est quand même assez fort : on arrive, grâce au *petit* théorème de Fermat, à montrer qu'un entier de 136 chiffres pris au hasard n'est pas premier [6].
Maintenant, que se passe-t-il si le résultat est effectivement 1 ? Cela fait-il de n un nombre premier ?
Par exemple, vérifiez que $2^{341-1} \equiv 1\,[341]$, et pourtant $341 = 11 \times 31$. On dit que 341 est *pseudopremier*.
Voici un petit programme pour trouver les premiers nombres pseudo premiers :

5. pourquoi ?
6. Ça peut paraître impressionnant comme ça, mais MuPAD s'avérera ridiculement minable quand il s'agira de tester des nombres d'un million de chiffres. Mais bon, c'est un début...

```
>> pseudo:=proc(a,n) local i,L;
>>  begin
>>  L:=[];
>>  for i from 3 to n step 2 do if not(isprime(i)) and pmod(a,i-1, i)=1 then L:=L.[i];
>>  end_if; end_for:
>>  L;
>>  end_proc:
>>  pseudo(2,2000);
```

[341, 561, 645, 1105, 1387, 1729, 1905]

```
>> pseudo(3,3000);
```

[91, 121, 671, 703, 949, 1105, 1541, 1729, 1891, 2465, 2665, 2701, 2821]

Pour des nombres plus grands, il n'est pas facile de savoir s'ils sont réellement premiers. Puisqu'on ne sait pas a priori s'ils le sont, on les appelle des nombres probablement premiers de base a. Certains sont PP d'une seule base. Par exemple

```
>> pmod(2,340,341)
```

1

```
>> pmod(3,340,341);
```

56

Ainsi, nous pouvons multiplier les bases pour éliminer les candidats. Malheureusement, il existe encore une infinité de nombres qui sont PP dans toutes les bases ,tout en étant composés :

Définition -1
Un entier composé n est un ***nombre de Carmichael*** *si $a^{n-1} \equiv 1[p]$ pour tout entier a premier avec n*

Pour se donner un ordre de grandeur, il y a un peu plus d'un million de nombres premiers inférieurs à 25 milliards, mais seulement 21 853 pseudo premiers et parmi eux 2 163 nombres de Carmichael. C'est peu, mais c'est toujours trop !
Malheureusement, les choses se compliquent un peu par la suite et nous devons jeter l'éponge à notre niveau faute d'outils algébriques.
Nous pouvons seulement démontrer quelques lemmes menant à des tests fort puissants mais pour l'instant hors d'atteinte.

VII.4. Système RSA

Nos amis Ronald **R**ivest, Adi **S**hamir et Leonard **A**dleman mirent au point en 1977 un système de codage alors qu'ils essayaient au départ de montrer que ce qu'ils allaient développer étaient une impossibilité logique : aah, la beauté de la recherche...Et savez-vous quel est la base du système ? Je vous le donne en mille : le petit théorème de Fermat ! Mais replaçons dans son contexte les résultats de R, S et A. Jusqu'il y a une trentaine d'années, la cryptographie était l'apanage des militaires et des diplomates. Depuis, les banquiers, les mega business men et women, les consommateurs internautes, les barons de la drogue et j'en passe ont de plus en plus recours aux messages codés. Oui, et alors ? Le problème, c'est que jusqu'ici, l'envoyeur et le destinataire partageait une même clé secrète qu'il fallait faire voyager à l'insu de tous : les états et l'armée ont la valise diplomatique, mais les autres ?

C'est ici que Whitfield Diffie et Martin Hellman apparaissent avec leur idée de principe qu'on peut résumer ainsi : votre amie Josette veut recevoir de vous une lettre d'amour mais elle a peur que le facteur l'intercepte et la lise. Elle fabrique donc dans son petit atelier une clé et un cadenas. Elle vous envoie le cadenas, ouvert mais sans la clé, par la poste, donc à la merci du facteur : le cadenas est appelé *clé publique*. Vous recevez le cadenas et l'utilisez pour fermer une boîte contenant votre lettre : le facteur ne peut pas l'ouvrir car il n'a pas la clé. Josette reçoit donc la boîte fermée : elle utilise sa clé, qu'elle seule possède, pour ouvrir la boîte et se pâmer devant vos élans epistolaires.

En termes plus mathématiques, cela donne : je fabrique une fonction π définie sur $\mathbb{N}$ qui possède une réciproque σ. On suppose qu'on peut fabriquer de telles fonctions mais que si l'on ne connaît que π, il est (quasiment) impossible de retrouver σ. La fonction ${}_pi$ est donc la clé publique : vous envoyez π(message) à Josette. Celle-ci calcule $\sigma\big(\pi(\text{message})\big) = \text{message}$. Josette est la seule à pouvoir décoder car elle seule connaît σ.
Tout ceci était très beau en théorie, mais Diffie et Hellman n'arrivèrent pas à proposer de telles fonctions. Mais voici qu'arrivent Ronald, Adi et Leonard...

Je vais vous présenter la méthode en effectuant les calculs grâce à MuPAD. Il ne vous restera plus qu'à prouver mathématiquement que tout ceci fonctionne de manière cohérente...

Nous aurons tout d'abord besoin d'un programme `num` qui transforme un message écrit en majuscule en un nombre entier et un programme `alph` qui effectuera la démarche inverse. Je vous les livre sans explications car leur fabrication ne nous intéresse pas arithmétiquement parlant [7].

```
>> num:=proc(m) local l,L;

>> begin l:=numlib::toAscii(m): L:="":

>> for i from 1 to nops(l) do L:=L.expr2text(l[i]) end_for:

>> text2expr(L); end_proc:

>> num("TU ES LE MEILLEUR");
```

848532698332766932776973767669858 2

```
>> alph:=proc(n) local l,L;

>> begin L:=[]: l:=expr2text(n):

>> for i from 0 to length(l)-1 step 2 do L:=L.[text2expr(substring(l,i,2))] end_for:

>> numlib::fromAscii(L); end_proc:

>> alph(676976653270657384327679787184697780833281856932746932766932836573833277 9\
78328065858682693280698473843284697269838373 78);
```

"CELA FAIT LONGTEMPS QUE JE LE SAIS MON PAUVRE PETIT TEHESSIN"

Ceci étant dit, je vais vous montrer comment quelqu'un peut m'envoyer un message crypté en utilisant le protocole RSA, que personne ne peut comprendre sauf moi, puis comment je vais décrypter ce message.
Avant de m'envoyer un message, on doit connaître ma *clé publique*, connue de tous et constituée d'un couple de nombres (n,e).
Et avant de connaître cette clé, il faut que je la fabrique - secrètement - de la manière suivante :

- ▷ je commence par fabriquer un nombre premier quelconque mais suffisamment grand

  ```
  >> p:=nextprime(85745232134567894561534867515378531351354531313543515 3);
  ```

  857452321345678945615348675153785313513545313135435373

7. Les curieux pourront toujours me demander des éclaircissements

▷ puis un autre

```
>> q:=nextprime(957413878646574313748313698454836415683846163846834364863 4);
```

$$957413878646574313748313698454836415683846163846834364864 3$$

Ces deux nombres ne sont connus que de moi !

▷ je forme le produit de ces deux nombres

```
>>  n:=p*q;
```

$$820936752734075304110522989015826163858292652874432299289707809102955590 20\backslash$$
$$564744753100917126450827310051456488 39$$

Ce nombre n est mon module public. Il reste à former le nombre e qui est ma puissance de cryptage publique. Là encore, cela se fait par étapes.

▷ je commence par former le nombre phi_n (tiens, tiens...) de la manière suivante

```
>> phi_n:=(p-1)*(q-1);
```

$$820936752734075304110522989015826163858292652874432298332208185224246708 56\backslash$$
$$277221420865737704931075472236665648 24$$

▷ c'est là qu'il faut jouer avec la chance : je dois choisir un nombre e premier avec phi_n. En pratique, je choisis un grand nombre premier au hasard :

```
>> e:=nextprime(4568531538443156358743168741353);
```

$$456853153844315635874316874144 7$$

Vérifions quand même que e et phi_n sont premiers entre eux

```
>> gcd(e,phi_n);
```

$$1$$

Ouf!... [8]

▷ il me reste à calculer ma puissance de décryptage secrète : je l'obtiens en cherchant l'inverse de e modulo phi_n, c'est à dire, comme vous commencez à le savoir puisque nous l'avons vu plusieurs fois, en déterminant un coefficient de Bézout grâce à l'algorithme d'Euclide étendu.
Le problème, c'est que nos nombres sont un peu grands pour faire l'algorithme à la main, mais...heureusement, il y a MuPAD. Il existe en effet une fonction `igcdex(x,y)` qui renvoie le PGCD de x et y, ainsi que des coefficients de Bézout u et v qui vérifient donc $ux + vy = x \wedge y$

```
>> igcdex(e,phi_n);
```

$$1,$$
$$-336156780915232966390080366314943856192209886396520797258981056750505725 6\backslash$$
$$485280235971052358778352121022620510929$$
$$, 18707200650453019624510714300 61$$

Ceci veut dire que $ue + v$ phi_n$= 1$, avec u le deuxième nombre renvoyé par MuPAD et v le troisième. Ainsi $ue \equiv 1$[phi_n], mais u n'est pas forcément l'inverse de e modulo phi_n, car il n'appartient pas forcément à l'intervalle entier $[\![0$,phi_n$-1]\!]$. Il suffit de calculer sa classe modulo phi_n

```
>> d:=-336156780915232966390080366314943856192209886396520797258981056750505725 6\
485280235971052358778352121022620510929 mod phi_n;
```

$$484779971818842337720442622700882307666082766477911501073227128473740982 91\backslash$$
$$424419061155214117147554262010460538 95$$

8. Pouvait-on en être sûr ?

Je suis maintenant prêt à recevoir des messages cryptés selon le protocole RSA et vous êtes prêt(e) à m'en envoyer car j'ai rendu publique ma clé (e,n).

▷ d'abord vous « numérisez » votre message.

```
>> text_num:=num("ESSAYE DONC DE DECRYPTER CE MESSAGE");
```

$$69838365896932687978673268693268696782898084698232676932776983836571 69$$

▷ comme nous allons calculer modulo n, il faut vérifier que text_num est plus petit que n, en utilisant par exemple `length`[9]

```
>> length(n);
```

$$112$$

```
>> length(text_num);
```

$$70$$

▷ ensuite, vous calculez ce nombre puissance e et vous regardez quelle est sa classe modulo n grâce à un petit programme mis au point précédemment.

```
>> crypte:=pmod(text_num,e,n);
```

$$35469233150717093648369621564213357837080328406430784459914432556683228305\backslash$$
$$48957393046529574279084728520248267373$$

▷ je reçois ce message et je le décrypte grâce à ma clé secrète d que je suis le seul à connaître dans tout l'univers

```
>> decrypt:=pmod(crypte,d,n);
```

$$69838365896932687978673268693268696782898084698232676932776983836571 69$$

▷ il ne me reste plus qu'à retrouver mon texte original grâce à la procédure `alph`

```
>> alph(decrypt);
```

"ESSAYE DONC DE DECRYPTER CE MESSAGE"

Ça marche !

Oui, bon, mais je devine que mille questions se bousculent déjà dans vos têtes : comment ça marche? Pourquoi ça marche? Comment casser le système? etc.
Pour vous aider, voici quelques questions intermédiaires.

1) Que vaut $\varphi(pq)$ si p et q sont premiers?
2) Où intervient le théorème d'Euler ? Pourquoi a-t-on le droit de l'utiliser ? Vous pourrez montrez que si $a^u \equiv a[p]$ et $a^u \equiv a[q]$, alors $a^u \equiv a[pq]$ en étudiant $a^u - a$.
3) À quelle opération mathématique correspond le « cassage » naturel du système? Est-ce un problème facile?
4) Supposons qu'un même texte en clair t est envoyé à deux personnes ayant choisi le même nombre n et des exposants e_1 et e_2 premiers entre eux : on trouve facilement u et v tels que $ue_1 + ve_2 = 1$. Alors les deux messages chiffrés sont $c_1 \equiv t^{e_1}[n]$ et $c_2 \equiv t^{e_2}[n]$. Montrez que, connaissant c_1, c_2, e_1, e_2 et n, on peut en déduire t, ce qui est fort gênant...

9. Et qu'est-ce qu'on peut faire si text_num est trop grand?