Algorithmique (Agrégation interne)

Bernard.Parisse@ujf-grenoble.fr

2010, 2017

Table des matières

N.B. : ce texte est disponible en ligne [1] (interactif) ou en PDF [2] (mort). La version HTML vous permet de tester avec ou sans modifications certains calculs ou algorithmes.

1 Algorithmique

Dans cette section, on passe en revue les notions de bases en algorithmique au programme (B.2), en commencant par travailler en ligne de commande (notion de variable, affectation), puis en complexifiant au fur et à mesure la ligne de commande (test, boucle) et on termine par l'écriture de fonctions. Les commandes des sections qui suivent peuvent être saisies et exécutées dans une ligne de commande de Xcas, pour des programmes qui dépassent une ou deux lignes, il est conseillé d'utiliser le menu `Prg`, et les assistants `Test` et `Boucle` (voir la section 1.3.2).

On pourra trouver d'autres tutoriels introduisant l'algorithmique, par exemple Algorithmique et programmation ay lycée [3] La syntaxe sera donnée en Xcas (très proche du langage algorithmique en français) ou Python, avec les éléments de syntaxe permettant d'adapter à d'autres languages utilisables à l'oral (Python, Scilab, Maxima), on ne reprendra par contre pas Free Pascal qui n'a pas d'interpréteur en mode interactif et se prête donc moins à la démarche proposée ici, ni Carmetal dont la version 3 permet d'écrire des programmes en javascript mais qui est trop spécialisé en géométrie par rapport aux thèmes abordés ici (sauf dans la section 1.2.2), ni bien sur les logiciels de géométrie ne permettant pas de programmer (en-dehors de macro-constructions). Quelques remarques sur ces logiciels et langages :

- Xcas est un logiciel de calcul formel, avec des fonctionnalités de géométrie (2-d et 3-d) et de tableur.
  On peut programmer en Xcas en français (langage algorithmique) ou en syntaxe compatible avec Python.
  Il existe plusieurs interfaces pour utiliser Xcas : Xcas pour PC/Mac/Linux ou Xcas pour Firefox utilisable sur tout matériel muni d'un navigateur compatible avec Firefox (tablette, smartphone).
- Python est un langage de programmation généraliste interprété, il dispose donc d'une librairie mathématique de base beaucoup moins fournie, ce qui peut être selon la situation un plus ou un moins. C'est maintenant le langage imposé au lycée, c'est pour cette raison que Xcas accepte maintenant la programmation en syntaxe Python.

1. `https://www-fourier.ujf-grenoble.fr/~parisse/agregint.html`
2. `https://www-fourier.ujf-grenoble.fr/~parisse/agregint.pdf`
3. `https://www-fourier.ujf-grenoble.fr/~parisse/algolycee.html`

– Scilab est un logiciel de calcul scientifique (non formel) utilisé dans l'industrie et l'enseignement supérieur, il est adapté pour des illustrations et algorithmes reliées à l'algèbre linéaire numérique et l'analyse numérique, mais beaucoup moins pour tout ce qui touche à l'arithmétique, à commencer par l'absence de type entier (on est obligé d'utiliser des flottants pour les représenter).
– Enfin, Maxima est un logiciel de calcul formel de niveau comparable à Xcas, mais sans tableur ni géométrie.

1.1 Données, variables, affectation

Pour stocker une donnée en mémoire, on utilise une variable. Une variable possède un nom, qui est composé d'au moins un caractère alphabétique (de A à Z ou de a à z) suivi ou non d'autres caractères alphanumériques, par exemple `a`, `var`, `b12`. Xcas est sensible à la différence entre majuscules et minuscules (c'est aussi le cas de Python, Maxima, Scilab). Une variable peut contenir n'importe quel type de donnée.

Pour donner une valeur à une variable

– on écrit un nom de **variable**, suivi par l'instruction d'**affectation** `:=` suivi par la valeur. Par exemple
```
a := 1.2
```
$$1.2$$
(en utilisant = en Python et Scilab, avec `:` en Maxima).
– ou (dans un programme interactif) on utilise l'instruction `saisir` suivi du nom de variable que l'on met entre parenthèses. Par exemple `saisir(a)` ou encore `saisir(a,b)` ou encore pour faciliter la saisie
```
saisir("Entrez votre nom",a,"Entrez votre age",b)
```
L'équivalent en Python et Scilab est `a=input("Entrez une valeur"),`.
Remarque : on évitera d'utiliser `saisir/afficher`, il vaut presque toujours mieux créer une fonction qu'un programme interactif, et ceci correspond aux changements dans le programme 2017 d'algorithmique de seconde.

Pour utiliser la valeur d'une variable

– on tape une ligne contenant le nom de la variable et la touche Entrée, la variable sera automatiquement remplacée par sa valeur. Par exemple si on tape `a:=2` puis `a+1` on obtient 3.
```
a:=2; a+1
```
$$2, 3$$
Avec Xcas et Maxima une variable non affectée n'est pas remplacée et reste symbolique (calcul formel), en Python et Scilab, cela déclenche une erreur.
– on peut aussi afficher la valeur d'une variable au cours de l'exécution d'un programme en utilisant la commande `afficher` (avec Maxima, Xcas `print("texte",variable)`, en Python 2 `print "texte",variable`, en Scilab `disp(variable)`). Cela permet d'afficher un résultat intermédiaire dans une zone située avant la ligne contenant le résultat du programme dans Xcas ou en bas de page dans

Xcas pour Firefox. Par exemple `afficher(a+1)`

$$0$$

affichera `3`.

Xcas peut manipuler différents types de données dont :

- les entiers, les fractions, les nombres flottants. La différenciation se fait pour les nombres par la présence d'un point séparateur décimal par exemple `1`, `2/3`, `1.1`. On peut utiliser la notation scientifique mantisse, exposant comme dans `1e-7` (préférable à `0.0000001`).
  `1; 2/3; 1.1`

$$1, \frac{2}{3}, 1.1$$

- les paramètres formels, par exemple `x`, `t`, `z`,
- les expressions, par exemple `x+1`, `x^2+y^2`,
- les fonctions, par exemple algébriques
  `f(x):=x^2; f(5)`

$$x \to x^2, 25$$

,

- les chaines de caractères, par exemple
  `s:="bonjour"`

$$\text{bonjour}$$

On accède à un caractère d'une chaine en donnant le nom de la chaine indicié par un entier compris entre 0 et la taille de la chaine moins 1 (même convention qu'en langage C), par exemple
`s[0]`

$$b$$

- les listes et les vecteurs, par exemple
  `a:=[1,2,3]`

$$[1, 2, 3]$$

On accède à un élément d'une liste en donnant le nom de la liste indicié par un entier compris entre 0 et la taille de la liste moins 1 (même convention qu'en langage C), par exemple
`a[0]`

$$1$$

On peut créer des listes de taille donnée avec une formule de remplissage, par

exemple `seq(0,10)`

$$[0,0,0,0,0,0,0,0,0,0,0]$$

`seq(j^2,j,1,10)`

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

(voir aussi `makelist`, `ranm`). Ce qui permet en particulier de faire des simulations sans avoir forcément besoin d'écrire un programme.

– des objets géométriques ou plus généralement graphiques (point, droite, cercle, polygone, plan, sphère, courbe représentative d'une fonction, etc.) obtenus par une instruction graphique (cf. section 1.2.2)

Maxima et Python peuvent manipuler des entiers, flottants, chaines de caractères, et listes, avec les mêmes délimiteurs que Xcas. Pour adresser un élément d'une liste, on utilise le nom de variable suivi de l'indice entre `[]`, les indices commencent à 0 en Xcas, Scilab, Python et à 1 en Maxima. Attention, Scilab ne propose pas de type entier (on peut représenter un entier par un flottant sans erreur de représentation s'il est inférieur en valeur absolue à 2^{53}).

Il faut être bien conscient du type de données que l'on manipule, par exemple en Xcas `7/2; type(7/2);`

$$\frac{7}{2}, rational$$

renvoie un rationnel, en Python2 `7/2` renvoie 3 le quotient euclidien des deux entiers, alors que `7./2.` renvoie 3.5 (le quotient des deux flottants). Attention, en Python3, `7/2` renvoie 3.5, le quotient en flottant, donc `4/2` est un flottant et pas un entier (il faut utiliser `//` pour obtenir le quotient euclidien sur les versions 2 et 3 de Python).

Exercices (Xcas) :

– Calculer `3*(1/3.)` et `3*(1/3)`.
– Définir la fonction `f(x):=(x+1/x)/2`, factoriser sa dérivée (`'` et `factor`).
– Simuler 100 lancers d'un dé à 6 faces (`rand` et `seq` ou `randvector`), stocker le résultat dans une liste `l` et en faire la moyenne (`mean`).

1.2 Instructions

1.2.1 Syntaxe

Une instruction valide dans Xcas doit suivre une syntaxe précise qui ressemble à l'écriture d'une expression en mathématique. Les différents éléments d'une instruction simple peuvent être des nombres (entiers, réels, flottants), des noms de variables, des opérateurs (par exemple +), des appels à des fonctions (l'argument ou les arguments se trouvant entre les parenthèses séparés par une virgule s'il y a plusieurs arguments) : par exemple

`sqrt(3)`

$$\sqrt{3}$$

pour désigner la racine carrée de 3 ou

```
irem(26,3)
```

2

pour désigner le reste de la division euclidienne de 26 par 3).

Il faut prendre garde à la priorité entre les différentes opérations. Par exemple dans 1+2∗3 l'opération ⋆ est effectué avant + comme en mathématiques. On peut utiliser des parenthèses pour forcer l'ordre des opérations, par exemple

```
(1+2)*3
```

9

.

Exercice : Calculer (vous pouvez utiliser la console en bas de page si vous consultez la version HTML de cette page)
`5/2/3`, `5/(2/3)`, `5/2*3`, `5/(2*3)`, `5^2*3`, `5^(2*3)`,`5*2^3`, `(5*2)^3`
Conclure sur les priorités relatives de la multiplication, de la division et de la puissance.

Attention, la multiplication doit (presque) toujours être indiquée explicitement contrairement aux mathématiques. Ainsi l'écriture `ab` ne désigne pas le produit de 2 variables `a` et `b` mais une variable dont le nom est `ab`.

Lorsqu'on veut exécuter plusieurs instructions à la suite, on termine chaque instruction par le caractère `;` (ou par les caractères `:;` si on ne souhaite pas afficher le résultat de l'instruction). En Python il suffit de passer à la ligne, en Scilab, Maxima, on utilise `;` (en syntaxe TI on utilise `:`).

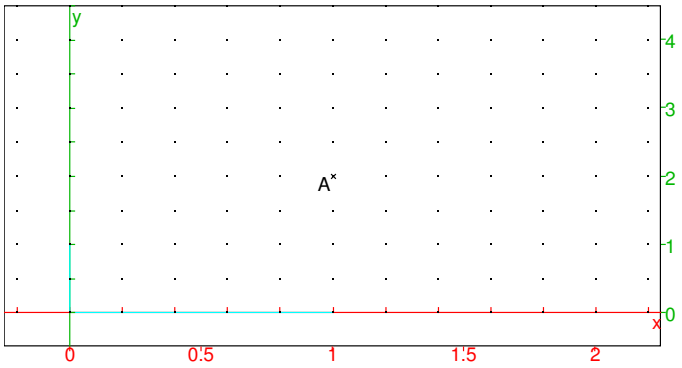
1.2.2 Instructions graphiques

Pour réaliser des graphes de fonction, on dispose d'assistants. dans le Menu Graphiques de Xcas PC.

La suite de cette section est spécifique à Xcas PC (non disponible dans Xcas pour Firefox), les autres systèmes n'ayant en général pas d'instruction géométrique interagissant comme du calcul numérique. Les habitués de Carmetal pourront noter une certaine similarité dans la programmation géométrique des deux systèmes (les instructions ont en général le même nom, avec une majuscule dans Carmetal et la syntaxe est très proche du C dans les 2 languages).

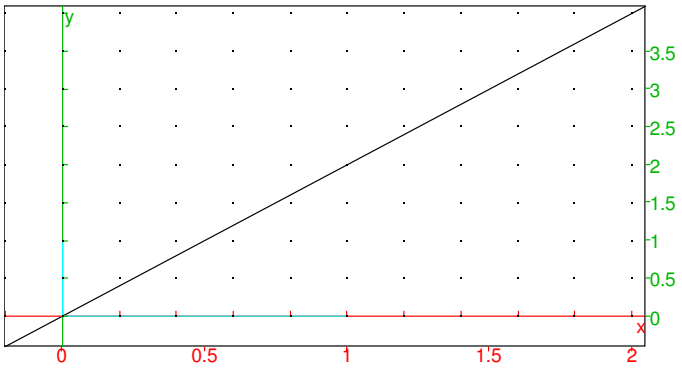
Toutes les instructions du menu `Geo` ont un résultat graphique (à l'exception des instructions du menu Mesure), par exemple :

- `A:=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, `droite(A,pente=2)`



la droite passant par A de pente 2.

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 $\perp$. Si une ligne de commande contient des instructions graphiques et non graphiques, c'est la nature de la dernière instruction qui décide du type d'affichage. On peut donner des attributs graphiques aux objets graphiques en ajoutant à la fin de l'instruction graphique l'argument `affichage=...`, argument dont la saisie est facilitée par le menu `Graphic->Attributs`.

L'instruction `A:=point(click())` permet de définir une variable contenant un point du plan que l'on clique avec la souris (`click()` renvoie le nombre complexe correspondant). C'est un peu l'analogue géométrique de l'instruction `saisir`.

Lorsqu'on veut voir sur un même graphique le résultat de plusieurs lignes de commandes, on crée une figure avec le menu `Geo->Nouvelle figure->graph geo2d`, la figure correspond alors aux lignes de commande situées à gauche. Dans ce mode, on peut créer les objets graphiques en ligne de commande ou pour certains directement à la souris en sélectionnant un `Mode` et en cliquant. On peut aussi déplacer un point en mode `Pointeur` et observer les modifications des objets géométriques qui en dépendent.

Exemples :
– calcul du milieu de 2 points sans utiliser l'instruction `milieu`.
```
A:=point(click()); B:=point(click());
xm:=(abscisse(A)+abscisse(B))/2;
ym:=(ordonnee(A)+ordonnee(B))/2;
C:=point(xm,ym)
```
– droite verticale passant par un point
```
A:=point(click()); D:=droite(x=abscisse(A))
```
Exercices (cf. le menu Geo pour les commandes de géométrie de Xcas) :
– faire cliquer un point puis afficher son symétrique par rapport à l'axe des x.
– Faire cliquer deux points A et B à la souris, déterminer un 3ème point C tel que ABC soit rectangle en A avec AC de longueur 1.
– Créer une figure (menu Geo, nouvelle figure, graph geo2d). Créer 3 points (soit avec l'instruction `point`, soit en passant en mode `point` et en cliquant 3 points à la souris). Afficher le centre du cercle circonscrit du triangle formé par ces 3 points en utilisant uniquement les instruction `mediatrice` et `inter_unique`.

1.2.3 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. En Python et Scilab, les tests sont identiques, avec Maxima on utilise = pour l'égalité.

Un test renvoie 1 ou true s'il est vrai, 0 ou false s'il est faux. On peut combiner le résultat de deux tests au moyen des opérateurs logiques `&&` (ou `et` ou `and`), `||` (ou `ou` ou `or`), et on peut calculer la négation logique d'un résultat de test avec `!` (ou `not`). En Python, on utilise les symboles `&&`, `||` ou `!`. Avec Maxima, Scilab, `and`, `or` et `not`.

On utilise ensuite souvent la valeur du test pour exécuter une instruction conditionnelle comme l'instruction `si` :
```
si condition alors bloc_vrai sinon bloc_faux fsi;
```
Un test peut être exécuté directement en ligne de commande, avec la syntaxe indiquée ci-dessous ou avec l'instruction `when` (Xcas syntaxe TI). La plupart du temps, on utilise un test dans une fonction ou un programme. Un test peut aussi servir pour arrêter une boucle en cours d'exécution ou renvoyer la valeur calculée par une fonction.
– En Python (ou Xcas compatible Python), l'alternative s'écrit `if(test):` passer à la ligne puis mettre le bloc vrai indenté, et `else:` passer à la ligne puis mettre le bloc faux indenté.
– En Xcas ou en Scilab
```
if condition then bloc_vrai else bloc_faux end.
```
– En Maple (aussi accepté en Xcas)
```
if condition then bloc_vrai else bloc_faux fi;.
```
– Avec Maxima
```
if condition then (bloc_vrai) else (bloc_faux)
```
où un bloc est une suite d'instructions séparées par des virgules et délimitée par des parenthèses.

– Xcas syntaxe TI, choisir le test dans les menus
`:If condition Then : action1 : Else : action2: EndIf.`
– Enfin 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;
```

Ifte : UnabletochecktestError : BadArgumentValue

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

Exercices :

– Saisir deux nombres réels et tester s'ils sont de même signe.
– (Xcas) Créer une figure (menu Geo->Nouvelle figure->graph, geo 2d) puis la `droite` d'équation $y = 2x + 1$, puis le `point` O origine, puis un point A quelconque, faire afficher si A est du même côté de la droite que O. De même pour O un point quelconque. Faire bouger le point A à la souris (en passant en `Mode` pointeur) pour vérifier les différentes possibilités.
– Tester si un triangle dont on fait cliquer les 3 sommets à l'utilisateur est rectangle (on pourra tester le théorème de Pythagore en chacun des trois sommets, en utilisant l'instruction `distance` pour avoir la distance entre 2 sommets).

1.2.4 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
`pour ... de ... jusque ... faire ... fpour`
Exemple, calcul de 10 !

```
f:=1; pour j de 1 jusque 10 faire f:=f*j; fpour;
```

$$1,3628800$$

On peut ajouter un pas s'il est différent de 1, par exemple le produit des nombres pairs de 2 à 10

```
f:=1; pour j de 2 jusque 10 pas 2 faire f:=f*j;
 fpour;
```

$$1,3840$$

– Boucle indéfinie
– `repeter ... jusqu_a ...`
Exemple, saisir un nombre entre 1 et 10

```
repeter saisir("Entrer un nombre entre 1 et 10",a);
 jusqu_a a>=1 && a<=10;
```

– `tantque ... faire ... ftantque`
  Exemple, algorithme d'Euclide

```
a:=25; b:=15; tantque b!=0 faire r:=irem
(a,b); a:=b; b:=r; ftantque
```

$25, 15, 0$

Remarques :

– Xcas accepte aussi la syntaxe de type C
  `for(init;condition;incrementation){ instructions }`
  l'arrêt de boucle en cours d'exécution (`if (...) break;`) dont l'usage peut éviter l'utilisation de variables de contrôle compliquées, et le passage immédiat à l'itération suivante (mot-clef `continue`) qui évite des forêts d'`if`.
– Lorsqu'on crée des objets graphiques dans une boucle, seul le dernier est affiché (car c'est le résultat de l'évaluation de la boucle). Pour afficher tous les objets graphiques d'une boucle, il faut les conserver dans une variable au cours de la boucle, en pratique on initialise une séquence `res:=NULL` avant la boucle, et on écrit dans la boucle `res:=res,objet_graphique`. Notez que les objets graphiques intermédiaires sont de toutes facons affichés dans la fenêtre `DispG` (menu `Cfg->Montrer`).
– En Python (et en Xcas en mode compatible Python), la boucle définie se fait sur une liste, souvent définie par `range`,
  `for var in range(start,stop):`
  on passe à la ligne et on saisit le bloc de la boucle indenté. La boucle indéfinie utilise `while condition:` et le bloc indenté.
– En Xcas, la syntaxe de la boucle définie de type Maple est
  `for var from debut to fin do bloc; od;`
  et la boucle indéfinie
  `while condition do instructions od;`.
– Avec Maxima, il y a de nombreuses facons d'écrire une boucle, (voir l'aide en ligne) par exemple pour une boucle définie
  `for var:debut thru fin do (bloc)`
– En Scilab, la syntaxe de la boucle définie est
  `for var=debut:fin; bloc; end;`
  et de la boucle indéfinie
  `while condition do instructions end;`.
– Xcas Syntaxe TI
  `:For I,A,B : action : EndFor`

Exercices :

– calculer la somme des cubes des nombres de 1 à 10.
– afficher la liste des diviseurs de 100 en testant si les entiers de 1 à 100 divisent 100
– afficher quelques points d'un graphe de fonction (par exemple en calculant les images des nombres de -2 à 2 avec un pas de 0.1, pour la fonction $x \to x^2$)
– Construire un polygone régulier à 10 côtés dont le centre est donné, ainsi qu'un

sommet (on pourra utiliser l'instruction `rotation`).

1.3 Fonctions

Pour réaliser des taches un peu plus complexes, il devient intéressant de les subdiviser en plusieurs entités indépendantes appelées fonctions, qui permettent d'étendre les possibilités des instructions du logiciel. Une fonction peut avoir 0, 1 ou plusieurs arguments (comme la fonction racine carrée `sqrt()` en a un). Elle calcule une valeur, appelée valeur de retour. Cette valeur de retour peut être utilisée dans une autre fonction ou expression algébrique, exactement comme le résultat de la fonction racine carrée.

La définition de fonctions peut parfois se faire directement avec une formule (fonction définie algébriquement) mais nécessite parfois un algorithme plus complexe. Elle peut calculer des résultats intermédiaires et les stocker dans des variables locales (afin de ne pas interférer avec les variables que l'utilisateur pourrait avoir défini en-dehors de la fonction). Par exemple pour calculer les racines d'un polynôme du second degré, on aura intérêt à calculer le discriminant et le stocker dans une variable locale.

Remarque :
De nombreuses calculatrices (TI non formelles, Casio, HP38/40) ne permettent pas de définir des fonctions, ainsi de nombreux algorithmes des programmes du secondaire ne sont pas rédigés comme des fonctions, mais comme des programmes ne prenant pas d'arguments et saisissant leurs arguments par des instructions `saisir` ou équivalent et **affichent** le résultat au lieu de le **renvoyer**. Il est alors difficile d'utiliser la valeur calculée (il faut la stocker dans une variable globale convenue á l'avance...) et l'exécution plusieurs fois d'un même programme pour le mettre au point devient vite pénible (il faut saisir encore et encore les mêmes valeurs). **Au niveau du concours de l'agrégation interne, on préférera l'écriture d'algorithmes sous forme de fonctions, conformément à l'esprit des nouveaux programmes du lycée.**

1.3.1 Fonctions simples

Il s'agit de fonctions définies par une formule algébrique.
Exemples :

- pour définir $f(x) = x^2 + 1$, on tape
  `f(x):=x^2+1`

$$x \to x^2 + 1$$

  (Maxima même syntaxe, Xcas accepte aussi `f:=x->x^2+1`)
- pour définir la valeur absolue sans utiliser `abs`, on tape :

```
absolu(x):= si x>0 alors x sinon -x fsi;
```

$$x \to \text{si}\,(x > 0, x, -x)$$

Exercices :

- définir une fonction par morceaux.

– définir une fonction `max2` calculant le maximum de 2 entiers (sans utiliser `max`) puis de 3 entiers en appelant la fonction précédente ou sans appeler la fonction précédente. Observer l'intérêt d'utiliser la fonction `max2`.

Attention

Il faut faire la différence entre fonction et expression. Par exemple `a:=x^2-1`

$$x^2-1$$

définit une expression alors que

`b(x):=x^2-1`

$$x \to x^2-1$$

définit une fonction. On peut donc écrire `b(2)`

$$3$$

qui renverra 3 mais l'analogue avec a serait `subst(a,x=2)`

$$3$$

. On peut composer des fonctions, par exemple la fonction $c = b \circ b$ est

`c:=b@b; normal(c(x))`

$$(x \to x^2-1)@(x \to x^2-1), x^4-2\cdot x^2$$

, composer une fonction n fois avec elle-même (`c:=b@@n`), et construire une fonction à partir d'une expression avec l'instruction `unapply` par exemple la fonction `b` dépendant de la variable `x` et définie par l'expression `a` est

`b:=unapply(a,x)`

$$x \to x^2-1$$

1.3.2 Fonctions plus complexes

La plupart des fonctions ne peuvent pas être définies simplement par une formule algébrique. On doit souvent calculer des données intermédiaires, faire des tests et des boucles. On peut observer que c'est déjà ce qui se passe pour des fonctions système comme `sum()` ou `seq` qui effectuent une boucle implicite (avec une variable d'index intermédiaire "muette"). Il faut alors définir la fonction par une suite d'instructions, délimitées en Xcas par `{ ... }` ou compris entre `fonction nom(parametres)` et `ffonction`.

La valeur calculée par la fonction est alors implicitement la valeur calculée par la dernière instruction exécutée (qui n'est pas forcément la dernière instruction du programme) ou peut être explicitée en utilisant le mot-clef `return` suivi de la valeur à renvoyer. Attention, dès que le mot-clef `return` est rencontré la valeur qui suit `return` est évaluée et la fonction se termine. Si on veut renvoyer 2 valeurs a et b, il ne sert à rien d'écrire `return a; return b;`, on les renverra plutot dans une liste ou une séquence (`return [a,b];`).

- En Python ou Xcas compatible Python, on écrit `def f(parametres):`, on saisit le bloc définissant la fonction indenté où on renvoie la valeur de retour par `return`.
- En Xcas syntaxe de type Maple, on utilise
  `f:=proc(parametres) ...; end;`

- Avec Maxima `f(parametres):=(bloc)`
- En Scilab, on utilise la syntaxe
  `function nom_retour=f(parametres); ...; endfunction`
  et on donne au cours de l'exécution une valeur à `nom_retour`.

Notez que l'exécution de `return` met un terme à la fonction, même à l'intérieur d'un test ou d'une boucle, ce qui permet souvent d'améliorer la lisibilité :

- au lieu d'écrire `si ... alors return a; sinon bloc fsi;` on peut écrire `si ... alors return a; fsi; bloc` . Ceci évite d'avoir des forêts d'ifs.
- au lieu d'ajouter des variables booléennes dans le test d'une boucle `tantque`, on peut souvent introduire un test avec `return`

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 (attention, en Python, toute variable intermédiaire d'une fonction est considérée comme locale, sauf indication contraire avec `global` ceci peut être source d'erreur si vous faites une faute de frappe dans un nom de variable).

Comme le texte définissant une telle fonction ne tient en général pas sur une ou deux lignes, il est commode d'utiliser un éditeur de programmes pour définir une fonction, soit l'éditeur fourni par le logiciel, soit un éditeur généraliste (comme ultraedit, atom, emacs, vi, ...). Dans Xcas, il est conseillé d'utiliser le menu `Prg->Nouveau programme`. Les assistants `Fonction`, `Boucle` et `Test` ou le menu `Prg->Ajouter` facilitent la saisie des principales structures et commandes de programmation. Les commandes de Xcas sont affichées en brun, les mots clef du langage en bleu.

Exemple : encore le PGCD mais sous forme de fonction

```
fonction pgcd(a,b)
  local r;
  tantque b!=0 faire
    r:=irem(a,b);
    a:=b;
    b:=r;
  ftantque;
  return a;
ffonction:;

pgcd(25,15)
```

On peut exécuter en mode pas à pas un programme et visualiser l'évolution de la valeur des variables avec l'instruction `debug()`, ce qui peut servir à comprendre le déroulement d'un algorithme, mais aussi à corriger un programme erroné (un analogue

existe en Scilab et Python, selon l'environnement de programmation). Par exemple, enlevez le `//` de commentaire et exécutez

```
// debug(pgcd(25,15))
```

undef

Exercice :

– Reprendre les énoncés des exercices précédents en créant des fonctions et en déclarant en variables locales les variables qui servent à effectuer des calculs intermédiaires.

1.3.3 Récursivité

On peut au cours du déroulement d'une fonction faire appel à cette même fonction avec un ou des arguments "plus simples", de sorte qu'après un nombre fini d'appels récursifs, l'argument ou les arguments sont devenus triviaux, et on peut alors renvoyer le résultat. Par exemple, l'algorithme d'Euclide peut s'énoncer sous la forme

Le PGCD de a et b est a si b est nul et est le PGCD de b et du reste de la division euclidienne de a par b sinon.

On peut le traduire en Xcas par

```
pgcdr(a,b):=si b==0 alors a sinon pgcdr(b,irem
(a,b)); fsi;
```

$$a,b \to \text{si}\,(b==0, a, \text{pgcdr}(b, \text{irem}\,(a,b)))$$

Dans certains cas, la récursivité permet de simplifier grandement la conception d'un algorithme, par exemple pour le calcul du reste de a^n par un entier fixé m, en distinguant n pair et n impair et en utilisant pour $n > 0$ pair :

$$a^n \pmod m = (a^{n/2} \pmod m)^2 \pmod m$$

et pour $n > 1$ impair :

$$a^n \pmod m = (a \times (a^{(n-1)/2} \pmod m)^2) \pmod m$$

Exercice : implémenter le calcul de $a^n \pmod m$ de cette manière.

Attention à bien s'assurer que le programme récursif se termine. En particulier quand on teste un programme récursif, il est conseillé de commencer par tester le cas où la récursion doit se terminer. Notez aussi que Xcas limite le nombre de récursions (réglage modifiable dans la configuration du cas).

Attention aussi, un algorithme récursif peut être très inefficace, par exemple si on calcule les termes de la suite de Fibonacci par la formule

```
F(n):=si n>1 alors F(n-1)+F(n-2); sinon 1; fsi;
```

$$n \to \text{si}\,(n > 1, F(n-1) + F(n-2), 1)$$

alors le calcul de F_5

```
F(5)
```

8

nécessite le calcul de F_4 et F_3 mais le calcul de F_4 demande lui-même le calcul de F_3 et F_2, on calcule donc deux fois F_3, trois fois F_2, cinq fois F_1, comme on peut le voir en affichant la valeur de n pour laquelle F est calculée :

```
 fonction F(n) print(n); si n>1 alors F(n-1
)+F(n-2); sinon 1; fsi;ffonction:;
```

Done

(exercice : le nombre total de calcul est lui-même une suite de Fibonacci !). Il est plus efficace dans ce cas d'écrire une boucle (exercice : le faire !).

2 Arithmétique

Pour cette section, on pourra se référer au manuel de programmation de Xcas. Pour les instructions prédéfinies, utiliser le menu CAS, Arithmétique ou le menu Cmds, Entier et Cmds, Polynômes.

Exposés : 103, 104, 106, 143, **159**, 168
Exercices : 302, 304, 305, **306**, 309, (pour **349** voir la section codage), 357

2.1 Arithmétique des entiers

2.1.1 Division euclidienne

Application :écriture d'un entier en base b. Instructions : `iquorem`, `iquo`, `irem`.

2.1.2 Euclide, Bézout

Commandes de Xcas : `gcd`, `iegcd` et `iabcuv`.

Exercice : Écrire un algorithme de PGCD itératif ou/et récursif. Même question pour Bézout.

Solution : Algorithme itératif pour a et b :

- Initialiser l_1 et l_2 à $[1, 0, a]$ et $[0, 1, b]$.
- Tant que $l_2[2] \neq 0$ (comme les indices commencent à 0, $l_2[2]$ est le dernier nombre de la liste l_2), faire
  - $q =$ quotient euclidien de $l_1[2]$ par $l_2[2]$,
  - $l_3 := l_1 - ql_2$, $l_1 := l2$, $l2 := l3$ (décalage)
- Renvoyer l_1

Exemple : exécutez la commande ci-dessous qui calcule l'identité de Bézout pour 125 et 45 en affichant les étapes de l'algorithme dans la partie basse de la page HTML

```
step_infolevel:=1:;iegcd(125,45);step_infolevel:=0:;
```

$$\text{Done}, [4, -11, 5], \text{Done}$$

Explication : A chaque itération, l'élément `l[2]` d'indice 2 de la liste l_1 (ou l_2) prend comme valeur l'un des restes successifs de l'algorithme d'Euclide, et on a un invariant de boucle `a*l[0]+b*l[1]=l[2]` qui donne l'identité de Bézout lorsque `l[2]` est le dernier reste non nul. Cet invariant se prouve facilement par récurrence.

On peut aussi voir cet algorithme comme une version arithmétique de l'algorithme du pivot de Gauss pour créer un zéro dans la dernière colonne de la matrice $\begin{pmatrix} L_1 & 1 & 0 & a \\ L_2 & 0 & 1 & b \end{pmatrix}$. Mais seules les manipulations de lignes du type $L_{n+2} = L_n - qL_{n+1}$ avec q **entier** sont admises, il faut donc plusieurs manipulations de lignes avant d'avoir un 0, la ligne précédente donnant le PGCD (dernier reste non nul) et les coefficients de Bézout.

$$\begin{pmatrix} L_1 & 1 & 0 & 125 \\ L_2 & 0 & 1 & 45 \\ L_3 = L_1 - 2L_2 & 1 & -2 & 35 \\ L_4 = L_2 - L_3 & -1 & 3 & 10 \\ L_5 = L_3 - 4L_4 & 4 & -11 & 5 \\ L_6 = L_4 - 2L_5 & -9 & 25 & 0 \end{pmatrix}$$

Algorithme de Bézout itératif en syntaxe Xcas (disponible depuis Doc/Exemples lycée dans Xcas pour Firefox le jour de l'oral ou depuis Xcas dans la session du sous-menu Exemples/arit/bezout.xws)

```
fonction bezout(a,b)
  local l1,l2,q;
  l1:=[1,0,a];
  l2:=[0,1,b];
  tantque l2[2]!=0 faire
    q:=iquo(l1[2],l2[2]);
    l1,l2:=l2,l1-q*l2;
  ftantque;
  return l1;
ffonction:;
```

On a utilisé l'affectation simultanée pour simplifier le décalage des indices.

```
bezout(25,15)
```

$$[-1, 2, 5]$$

Le même algorithme en Python. Attention, la combinaison linéaire de ligne doit se faire élément par élément.

```
def bezoutpy(a,b):
    # local l1,l2,q
    l1=[1,0,a]
```

```
    l2=[0,1,b]
    while l2[2]!=0:
        q=l1[2]//l2[2]
        l1,l2=l2,[l1[0]-q*l2[0],l1[1]-q*l2[1],l1[2]-q*l2[2]]
    return l1

bezoutpy(125,45)
```

$$[4, -11, 5]$$

On peut adapter pour calculer l'inverse d'un entier modulo un autre entier en diminuer la longueur des listes de 1 (on n'a besoin que d'un seul des coefficients de Bézout).

Applications/exercices possibles :

- PGCD : simplification, On peut aussi discuter le nombre d'étapes (maximisé par la suite de Fibonacci).
- PPCM : réduction au même dénominateur, recherche de racines rationnelles dans $\mathbb{Q}[X]$. Les cofacteurs du PPCM apparaissent dans l'algorithme itératif de Bézout à la ligne du reste nul.
- Bézout (N.B. : peut servir dans la 306) inverse dans $\mathbb{Z}/p\mathbb{Z}$, restes chinois, décomposition en éléments simples (peu d'intérêt pratique dans $\mathbb{Z}$)

2.1.3 Nombres premiers

Test de primalité par division. Recherche des nombres premiers inférieurs à un entier donné (crible).

```
fonction crible(n)
  local tab,prem,p;
  tab:=seq(j,j,0,n); // liste des entiers <= n
  tab[0]:=0; tab[1]:=0; // on remplace par 0 un entier non premier
  p:=2;
  tantque p*p<=n faire
    // afficher(tab," on barre les multiples de "+p);
    // p est premier, on barre les multiples de p
    pour j de p*p jusque n pas p faire
      tab[j]:=0;
    fpour;
    // on cherche le prochain nombre non barre qui est premier
    p:=p+1;
    tantque p*p<=n et tab[p]==0 faire
      p:=p+1;
    ftantque;
  ftantque;
  // on cree la liste des nombres non barres
  prem:=[];
  pour j de 2 jusque n faire
    si tab[j]!=0 alors prem.append(j); fsi;
```

```
  fpour;
  return prem;
ffonction:;
```

```
crible(100)
```

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

Pour obtenir la traduction en Python de ce programme, on peut utiliser la commande `python(crible)` dans Xcas, après nettoyage :

```
def criblepy(n):
    # local tab,prem,p
    tab=list(range(n+1))
    tab[0]=0
    tab[1]=0
    p=2
    while p*p<=n :
        j=p*p
        while j<=n :
            tab[j]=0
            j=j+p
        p=p+1
        while p*p<=n and tab[p]==0 :
            p=p+1
    prem=[]
    for j in range(2,n+1):
        if tab[j]!=0 :
            prem.append(j)
    return(prem)
```

```
criblepy(100)
```

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

Test de primalité probabiliste de Miller-Rabin :
Ce test utilise le fait que si p est premier, $\mathbb{Z}/p\mathbb{Z}$ est un corps et pour $a \neq 0$, $a^{p-1} = 1 \pmod p$. On écrit p premier différent de 2 sous la forme $2^t s$ avec s impair. Comme l'équation $x^2 = 1$ n'admet que 1 et -1 comme racines modulo p, pour a fixé on peut avoir soit $a^t = 1 \pmod p$, sinon en prenant t fois le carré modulo p on doit prendre la valeur -1. Si le test échoue, p n'est pas premier. Si le test réussit, p n'est pas forcément premier, mais on peut montrer qu'il y a au plus 1 entier sur 4 pour lequel il réussit. On reprend donc le test pour quelques autres valeurs de a jusqu'à être raisonnablement sur que p est premier (il existe aussi des certifications de primalité).

```
fonction millerrabin(p,a)
  local t,s,j;
  si irem(p,2)==0 alors return "p doit etre impair"; fsi;
```

```
  // on ecrit p-1 sous la forme 2^t*s
  s:=p-1;
  t:=0;
  tantque irem(s,2)=0 faire
    s:=s/2;
    t++;
  ftantque;
  // on calcule a^s, si c'est 1 le test passe,
  // sinon on doit avoir un -1 dans la chaine des carres
  a:=powmod(a,s,p);
  si a==1 alors return "p est peut-etre premier"; fsi;
  pour j de 1 jusque s faire
    si a==p-1 alors return "p est peut-etre premier"; fsi;
    a:=irem(a*a,p);
  fpour;
  return "p n'est pas premier";
ffonction:;
```

```
millerrabin(101,2)
```

$$\text{pestpeut} - \text{etrepremier}$$

En Python def millerrabinpy(p,a) : if p return "p doit etre impair" s=p-1 t=0 while s s=s//2 t+=1 a=pow(a,s,p) if a==1 : return "p est peut-etre premier" for j in range(s) : if a==p-1 : return "p est peut-etre premier" a=a*a return "p n'est pas premier"

On peut aussi aborder l'algorithme de Pollard-rho basé sur le théorème des anniversaires, voir le manuel Algorithmes [4] dans l'aide de Xcas.

2.1.4 Restes chinois

Recherche de $a \pmod{\prod p_i}$ connaissant $a \pmod{p_i}$. On peut commencer par 2 nombres premiers (et utiliser Bézout). En effet si $a = a_1 \pmod{p)_1}$ et $a = a_2 \pmod{p)_2}$ alors il existe des entiers u et v tels que $a = a_1 + up_1 = a_2 + vp_2$, on peut calculer u et v en appliquant Bézout à $a_1 - a_2 = -up_1 + vp_2$.

La commande de Xcas correspondante est `ichinrem`.

2.1.5 RSA and co

On choisit $n = pq$ produit de deux nombres premiers assez grands gardés secrets, de sorte que la factorisation de n soit trop longue pour pouvoir déterminer p et q connaissant n. L'application de codage dans $\mathbb{Z}/n\mathbb{Z}$ consiste à calculer

$$x \to x^d \pmod{n}$$

Elle est inversible si d admet un inverse e modulo $\phi(n)$ où $\phi(n) = (p-1)(q-1)$ est l'indicatrice d'Euler de n, son inverse est identique à l'application de codage en remplacant d par e.

4. https://www-fourier.ujf-grenoble.fr/~parisse/algo.html

Exercice : écrire des fonctions de chiffrement/déchiffrement par RSA. Instruction Xcas utiles : `powmod`, `inv(d % n)`. Pour convertir une chaine de caractères en liste d'entiers (codes ASCII) on peut utiliser `asc` et `char`. Pour grouper plusieurs entiers, on peut utiliser `convert(.,base,.)`

Voir la section 3.2.2 ci-dessous.

2.2 Arithmétique des polynômes

2.2.1 Division euclidienne

Calcul du quotient et du reste de la division de 2 polynômes dont les coefficients sont donnés dans une liste. Instructions : `quorem`, `quo`, `rem`. Exemple d'application : élimination de racines (pour trouver toutes les racines d'un polynôme)

2.2.2 Évaluation d'un polynôme en un point

Application : inverse de l'écriture en base b. Programmation de la méthode de Horner (fonction `horner` de Xcas)
Il s'agit d'évaluer efficacement un polynôme

$$P(X) = a_n X^n + ... + a_0$$

en un point. On pose $b_0 = P(\alpha)$ et on écrit :

$$P(X) - b_0 = (X - \alpha)Q(X)$$

où :

$$Q(X) = b_n X^{n-1} + ... + b_2 X + b_1$$

On calcule alors par ordre décroissant b_n, b_{n-1}, ..., b_0.

1. Donner b_n en fonction de a_n puis pour $i \leq n-1$, b_i en fonction de a_i et b_{i+1}. Indiquez le détail des calculs pour $P(X) = X^3 - 2X + 5$ et une valeur de α entière non nulle.
2. Écrire un fonction `horn` effectuant ce calcul : on donnera en arguments le polynôme sous forme de la liste de ces coefficients (dans l'exemple `[1,2,0,-1,5]`) et la valeur de α et le programme renverra $P(\alpha)$. (On pourra aussi renvoyer les coefficients de Q).
3. Quel est le nombre d'opérations effectuées (comparer à ce que donne le calcul en appliquant la forme développée du polynôme) ?
4. En utilisant cette fonction, écrire une fonction qui calcule le développement de Taylor complet d'un polynôme en un point.

Solution : menu Aide de Xcas, Exemples, poly, horner.

```
def Horner(p,a):
  # p liste des coefficients par ordre décroissant
  # renvoie p évalué en a
  # local c,r
```

```
r=0
for c in p:
  r=r*a+c
return r
```

Test et vérification avec la commande interne `horner` de Xcas

```
Horner([1,2,3,4],5); horner([1,2,3,4],5)
```

$$194, 194$$

2.2.3 Euclide, Bézout

Commandes de Xcas : `gcd`, `egcd` et `abcuv`. Se programme comme pour les entiers en utilisant la division euclidienne des polynômes (`quo`, `rem`). Applications :

- Racines multiples d'un polynôme (PGCD de P et P'), factorisation `squarefree` (i.e. comme produit de polynômes sans racines multiples premiers entre eux à une puissance).
- (variante d'Euclide) Suites de Sturm pour localiser les racines réelles d'un polynome (cf. le manuel Algorithmes du menu Aide de Xcas).
- Bézout : décomposition en éléments simples de $\frac{C}{AB}$, si A et B sont premiers entre eux, il existe U et V tels que $C = AU + BV$ et on a

$$\frac{C}{AB} = \frac{AU + BV}{AB} = \frac{U}{B} + \frac{V}{A}$$

- élimination d'une variable dans un système polynomial de 2 équations à 2 inconnues (en utilisant Bézout, le résultant n'est pas au programme). On considère le système

$$A(x, y) = 0, \quad B(x, y) = 0 \quad (S)$$

où A et B sont deux polynômes en x et y. On voit A et B comme deux polynômes en x à coefficients polynomiaux en y. On applique Bézout, si A et B sont premiers entre eux, on a $AU + BV = 1$ avec U et V des polynômes en x à coefficients dans le corps des fractions en y. On calcule $R(y)$, le PPCM des dénominateurs de U et V, on a alors

$$A(UR) + B(VR) = R(y)$$

Les solutions (x, y) de (S) vérifient alors $R(y) = 0$ car UR et VR sont des polynômes en x, y.
- (Bézout) : Approximant de Padé. Voir le manuel Algorithmes de Xcas.

2.2.4 Racines rationnelles

Écrire une fonction qui détermine les racines rationnelles d'un polynôme P à coefficients entiers, en observant qu'elles sont de la forme p/q où q divise le coefficient dominant de P et $\pm p$ divise son coefficient de plus bas degré. Tester avec le polynôme $P = 12x^5 + 10x^4 - 6x^3 + 11x^2 - x - 6$.

N.B. : ce n'est pas un algorithme efficace, cf. le manuel Algorithmes de calcul formel. Les commandes Xcas correspondantes sont `rationalroot, crationalroot`.

Solution :

```
fonction racrat(P)
  local Lp, Lq, rac, p, q, r;
  // determiner les fractions possibles p/q
  // p est dans la liste des diviseurs du coefficient constant de P
  Lp:=idivis(abs(tcoeff(P)));
  // q est dans la liste des diviseurs du coefficient dominant de P
  Lq:=idivis(abs(lcoeff(P)));
  rac:=[]; // liste des racines rationnelles, initialement vide
  pour p in Lp faire
    pour q in Lq faire
      si gcd(p,q)==1 alors
        r:=p/q;
        // test si P(r)=0 ou P(-r)=0, si oui ajout a la liste des racines
        si P(x=r)==0 alors rac:=append(rac, r); fsi;
        si P(x=-r)==0 alors rac:=append(rac, -r); fsi;
      fsi;
    fpour;
  fpour;
  return rac;
ffonction:;
```

```
racrat(4x^2-1)
```

$$[\frac{1}{2}, \frac{-1}{2}]$$

2.2.5 Racines réelles et complexes

`proot` permet d'avoir les racines approchées d'un polynôme à 1 variable. `solve, csolve` calculent les racines racines exactes si le calcul a un intérêt, au sens où le résultat peut être raisonnablement manipulé par le calcul formel. `pcoeff` donne les coefficients du polynôme unitaire dont on passe en arguments la liste des racines.

2.2.6 Autres

Interpolation de Lagrange, fonction `lagrange` de Xcas. Référence : documentation de Xcas et le livre de Demailly.

Transformée de Fourier rapide `fft, ifft`, application que l'on peut citer pour la lecon 167.

3 Codage et cryptographie

Ces thèmes entrent au programme de l'agrégation interne en 2012 (exercices 349), probablement pour faire suite aux changements dans le programme de spécialité de Terminale S.

3.1 Codes

On désigne sous ce nom la numérisation de données, mais aussi diverses applications de l'algèbre et de l'arithmétique pour tester qu'une donnée a été transmise correctement en ajoutant une information supplémentaire. Pour certains codes (codes correcteurs), cette information supplémentaire permet de corriger un petit nombre d'erreurs de transmission.

3.1.1 Numérisation de texte

On peut citer le code ASCII, qui permet d'associer à tous les caractères américains un entier compris entre 0 et 127 (7 bits). Ainsi A correspond à 65, B à 66, etc. On peut tester la transmission en ajoutant un 8ème bit de parité (bit de poids fort mis à 0 ou à 1 pour que l'entier transmis soit pair). La prise en compte des accents et alphabets non latins à conduit à utiliser d'autres codes, les isolatin par exemple ou les pages de code windows. Ces codes n'étaient pas compatibles entre eux, récemment, ils ont été remplacés par l'unicode, dont deux variantes sont très populaires : UTF8 (compatible avec ASCII) et UTF16.

Xcas utilise l'UTF8 pour coder des chaines de caractères. Les commandes `asc` et `char` permettent de convertir une chaine de caractères en une liste et réciproquement. Par exemple `l:=asc("Bonjour tout le monde"); char(l);`

$$[66, 111, 110, 106, 111, 117, 114, 32, 116, 111, 117, 116, 32, 108, 101, 32, 109, 111, 110, 100, 101], \text{Bonjourtoutlemond}$$

On peut travailler sur la liste d'entiers obtenus, par exemple en considérant qu'il s'agit d'un seul entier écrit en base 256 (instruction `convert`), par exemple

```
m:=convert(l,base,256);
```

$$148185027236072995390230131666040330257055330758466$$

si cet entier est trop grand, on peut le décomposer à nouveau, par exemple selon une base une puissance de 256 (ce qui revient à regrouper les caractères du message initial par bloc)

```
L:=convert(m,base,256^8)
```

$$[2338060277946150722, 2334390868510076788, 435476655981]$$

3.1.2 Représentation des nombres approchés

Pour représenter un nombre à virgule flottante, les microprocesseurs utilisent le format "double" composé d'un bit de signe, 11 bit pour l'exposant et 52 bits pour la mantisse. Voir le manuel Algorithmes de Xcas.

3.1.3 Vérification

Les codes de vérification les plus simples appelés clés, consistent à calculer le reste de la division de l'entier codant la donnée par un entier donné et à ajouter cette information supplémentaire (la clé). Le bit de parité est l'exemple le plus simple (parité du nombre de 1 dans l'écriture en base 2 d'un nombre). Autre exemple la clé RIB est le complément à 97 du reste de la division par 97 de l'information. Comme pour la preuve par 9, si le test de divisibilité est incorrect, on est sur que l'information est incorrecte, mais si le test est correct on n'est pas sur que l'information est bonne.

3.1.4 Codes polynomiaux

Définition : On travaille sur un corps fini (par exemple $\mathbb{Z}/p\mathbb{Z}$ pour p premier). On représente le message de longueur k à coder par un polynôme P de degré $k-1$ dont les coefficients contiennent l'information (par exemple message). On se donne un polynôme $g(x)$ de degré $m = n-k$ (avec $n > k$). On multiplie P par x^{n-k}, on calcule le reste R de la division euclidienne de Px^{n-k} par g. On émet alors $Px^{n-k} - R$ qui est un polynôme de degré $\leq n-1$ divisible par g.

Un des intérêts des codes polynomiaux est que la vérification de bonne transmission de l'information est très simple : il suffit de tester la divisibilité par g (et on extrait l'information pertinente en ne gardant que les k premiers coefficients).

En Xcas, on peut travailler avec des polynômes à coefficients dans $\mathbb{Z}/p\mathbb{Z}$ comme avec les mêmes instructions que pour des polynômes à coefficients dans $\mathbb{Q}$ en ajoutant `mod p`. Par exemple `rem((x^10+3x^3+5) mod 7,(3x^2-2x+3) mod 7)`

$$-1\%7 \cdot x + 2\%7$$

. On peut aussi travailler avec un corps fini de cardinal non premier (instruction `GF`, par exemple

```
GF(2,8)
```

$$GF(2, k^8 + k^4 + k^3 + k^2 + 1, [k, K, g], \text{ undef })$$

permet de travailler sur le corps fini à 256 éléments, cf. le manuel Algorithmes de Xcas) mais ceci est un peu au-dessus du niveau de l'agrégation interne.

Exercice : écrire de cette facon le codage du bit de parité (divisibilité par $x+1$ modulo 2 ou évaluation nulle en 1). Puis une procédure Xcas de codage utilisant $g = X^7 + X^3 + 1$ sur le corps $\mathbb{Z}/2\mathbb{Z}$ (ce polynôme était utilisé par le Minitel).

Remarque : les codes polynomiaux sont un cas particulier de codes linéaires mais les espaces vectoriels à coefficients dans un corps finis ne sont pas au programme de l'agrégation interne, on n'en parlera donc pas.

3.1.5 Codes polynomiaux correcteurs

Avertissement : les notions abordées ici vont parfois bien au-delà de ce qu'on peut attendre d'un candidat à l'agrégation interne. Il s'agit plutot de montrer que des notions assez abstraites comme les polynômes à coefficients dans un corps fini ont des applications.

Si le polynôme g est de degré pas trop petit, on peut espérer que l'information complémentaire contenue dans le reste peut permettre de corriger une ou 2 erreurs de transmission.

Définition : On appelle distance (de Hamming) entre 2 polynômes le nombre de coefficients distincts de ces 2 polynômes (On vérifie qu'il s'agit bien d'une distance).

Définition On appelle distance du code polynomial défini par l'entier k et le polynôme g de degré $m = n - k$ le minimum de la distance de Hamming entre 2 multiples distincts de g de degré $< n$ (ou ce qui revient au même d'un multiple non nul de g de degré $< n$).

Si une information reçue est codée par un polynôme P qui n'est pas multiple de g, il y a erreur de transmission, on peut chercher à corriger les coefficients mal transmis en remplaçant P par un multiple de g de degré $< n$ proche de P au sens de la distance de Hamming. Cela fait sens si on peut assurer l'unicité du multiple de g le plus proche (ceci n'assure pas l'existence d'un tel multiple de g, même si c'est vrai pour certains codes). Si le nombre de coefficients à corriger est au plus t et si $2t + 1$ est inférieur ou égal à la distance du code, alors il y aura unicité (sinon il y aurait deux multiples distincts de g à distance $\leq t$ de P donc dont la distance mutuelle serait $\leq 2t$, impossible car cette distance serait strictement inférieure à la distance du code).

On a donc tout intérêt à ce que la distance du code soit la plus grande possible. On observe que la distance du code est inférieure ou égale au nombre de coefficients non nuls de g puisque g est un multiple non nul de lui-même, donc à $m + 1$. Dans les cas favorables (choix judicieux de g), la distance du code vaut exactement $m + 1$ et on pourra alors espérer corriger au plus $m/2$ erreurs.

Proposition Soit K corps fini de générateur a, on considère un code polynomial tel que $n <$cardinal(K) et

$$g(x) = \prod_{j=1}^{m} (x - a^j)$$

Alors la distance du code correspondant est $m + 1$.

Preuve : soit P un multiple de g de degré $< n$ ayant au plus m coefficients non nuls, alors on peut écrire

$$P(x) = \sum_{j=1}^{m} p_{k_j} x^{k_j}, \quad k_j < n$$

Comme P est un multiple de g, il s'annule en $a, a^2, .., a^l, .., a^k$, on a donc le système

$$\sum_{j=1}^{m} p_{k_j} (a^{k_j})^l = 0, \quad l = 1..m$$

Mais ce système en les p_{k_j} est un système de Cramer m, m, car son déterminant est un déterminant de Vandermonde engendré par des éléments du corps fini, les $a^{k_j}, j = 1..m$, qui sont distincts 2 à 2 puisque a est un générateur du corps.

Exercice : générer un code polynomial sur $\mathbb{Z}/97Z$ pouvant corriger jusqu'à 2 erreurs. Programmer la correction au plus proche (on commencera par tester la divisibilité en changeant un coefficient, puis 2).

Exercice : lien avec la loi binomiale. On suppose que la probabilité d'erreur de transmission d'un coefficient est ε (par exemple $\varepsilon = 0.001 = 1e-3$, et que les erreurs de transmission coefficient par coefficient sont indépendantes. On transmet n caractères, quelle est la probabilité d'avoir au plus t erreurs de transmission (et donc de pouvoir corriger avec le code précédent) ?

3.2 Cryptographie

3.2.1 Jules César, Vigenère, affine, Hill.

Le principe consiste à utiliser une bijection de $\mathbb{Z}/p\mathbb{Z}$ dans lui-même pour crypter le message. Le codage de Jules César peut être vu comme l'addition d'une constante dans $\mathbb{Z}/26\mathbb{Z}$, Vigenère est l'addition des lettres du message à crypter avec les lettres d'un texte fixé. Le codage affine utilise une application affine

$$x \to ax + b$$

avec a inversible modulo p (le calcul de l'inverse fait intervenir l'identité de Bézout). Aucune de ces méthodes n'est résistante à une analyse statistique du message crypté (dans le cas de Vigenére, l'attaque est plus complexe à mettre en oeuvre).

Le chiffrement de Hill groupe les lettres du message à crypter par paquets de n (n fixé) pour éviter l'attaque par analyse statistique, on a donc un vecteur $v \in (\mathbb{Z}/p\mathbb{Z})^n$ dont on calcule l'image par un chiffrement affine

$$v \to Av + b$$

où A est une matrice inversible sur $\mathbb{Z}/p\mathbb{Z}$ (donc est à la limite du programme sauf dans des cas comme $n = 2$ où on peut exprimer l'inverse explicitement de manière simple, malheureusement dans ce cas n n'est pas suffisamment grand pour que ce code soit résistant à une analyse statistique).

De plus toutes ces méthodes supposent que les clés de chiffrement et de déchiffrement sont secretes (en effet si on connait l'une des clefs on en déduit l'autre), alors que RSA par exemple permet de publier une des deux clefs.

Exercices : écrire des fonctions de chiffrement/déchiffrement par ces méthodes. Instruction en Xcas : `asc`, `char`, `%`, `inv`.

3.2.2 Cryptographie RSA

Rappel du principe de codage RSA :

- Chaque personne souhaitant coder ou signer un message dispose d'une clef privée, un entier s connu de lui seul, et d'une clef publique, une paire d'entiers (c, n).
- n est le produit de 2 nombres premiers p et q, et s et c sont inverses modulo $\varphi(n)$, où $\varphi(n) = (p-1)(q-1)$ (le nombre d'entiers de l'intervalle $[1, n[$ premiers avec n), on a alors (cf. devoir 1)

$$(a^c \pmod n)^s \pmod n = (a^s \pmod n)^c \pmod n = a \pmod n \tag{1}$$

- Pour coder un message à destination d'une personne dont la clef publique est (c, n), on commence par le transformer en une suite d'entiers, On peut par exemple remplacer chaque caractère par un entier compris entre 0 et 255, son code ASCII.
- Puis on envoie la liste des nombres $b = a^c \pmod n$. En principe, seule la personne destinataire connait s et peut donc retrouver a à partir de b en calculant $b^s \pmod n$.
- On peut aussi authentifier qu'on est l'auteur d'un message en le codant avec sa clef privée, tout le monde pouvant le décoder avec la clef publique.

Exercice 1 : Générer une paire de clefs

Génèrez deux nombres premiers p et $q > 256$ au hasard, en utilisant par exemple les fonctions `nextprime` et `rand`. Calculez $n = p \times q$ puis $\varphi(n) = (p-1)(q-1)$ puis choisissez une clef secrète s inversible modulo $\varphi(n)$ et calculez son inverse c. Vérifiez sur quelques entiers la propriété (1), on utilisera la fonction `powmod(a,c,n)` pour calculer $a^c \pmod n$.

Exercice 2 : Codage et décodage d'un message

On transforme une chaine de caractère en une liste d'entiers et réciproquement avec `asc` et `char`. Pour l'appliquer à une liste `l`, on peut utiliser `map(l,powmod,c,n)`. En utilisant la paire de clefs de l'exercice 1, codez un message puis décodez ce message pour vérifier. Décodez le message authentifié situé sur ce lien [5]

Exercice 3 : Puissance modulaire rapide

Pour pouvoir crypter des messages de cette manière, il est nécessaire d'avoir une fonction calculant $a^c \pmod n$ rapidement. Comparer le temps de calcul de `a^c mod n` et `powmod(a,c,n)` pour quelques valeurs de a, c, n (on pourra utiliser `time(instruction)` pour connaitre le temps d'exécution d'une instruction. Pour comprendre cela, programmez une fonction `puimod` calculant $a^c \pmod n$ en utilisant et en justifiant un des algorithmes

- récursif : si $c == 0$ on renvoie 1, si $c == 1$ on renvoie a, sinon, si c est pair, on pose $b = a^{c/2} \pmod n$ (calculé par appel récursif) et on renvoie $b \times b \pmod n$, et enfin si c est impair, on pose $b = a^{(c-1)/2} \pmod n$ (calculé par appel récursif) et on renvoie $b \times b \times a \pmod n$

5. `https://www-fourier.ujf-grenoble.fr/~parisse/mat249/rsa1`

– itératif : on initialise $A \leftarrow a$, $b \leftarrow 1$, puis tant que $c \neq 0$
  – si c est impair, $b \leftarrow A \times b \pmod n$,
  – $c \leftarrow$ quotient euclidien de c par 2,
  – $A \leftarrow A \times A \pmod n$
  et on renvoie b.

Exercice 4 : Attaque simple
L'utilisation de 256 valeurs possibles pour a se prête à une attaque très simple : la personne souhaitant décoder un message codé avec une clef publique sans en connaitre la clef secrète calcule simplement la liste des $a^c \pmod n$ pour les 256 valeurs possibles de a et compare au message. Décodez de cette manière le message situé ici [6].

Exercice 5 : Groupement de lettres
Pour parer à cette attaque, on va augmenter le nombre de valeurs possibles de a pour que le calcul de la liste de toutes les puissances des a possibles soit trop long. Pour cela, on groupe par paquets de x caractères et on associe à un groupe de caractères l'entier correspondant en base 256. Par exemple, si on prend des groupes de $x = 3$ caractères, "ABC" devient `65*256^2+66*256+67` car le code ASCII de A, B, C est respectivement 65, 66, 67. Donner une condition reliant n et x pour que le décodage redonne le message original. Choisissez une paire de clefs vérifiant cette condition pour $x = 8$. Ecrire un programme de codage et de décodage avec groupement (on commencera par compléter le message original par des espaces pour qu'il soit un multiple de 8 caractères, l'instruction `size` permet de connaitre la taille d'une chaine de caractères, on pourra utiliser la fonction d'écriture en base de la feuille d'exercices 2).

Exercice 6 : Sécurité du codage 1
Montrer que la connaissance de $\varphi(n)$ et de n permet de calculer p et q par résolution d'une équation de degré 2. La sécurité du codage repose donc sur la difficulté de factoriser n. Tester sur des entiers de taille croissante le temps nécessaire au logiciel pour factoriser p et q. Une valeur de n de taille 128 bits, 512 bits, 1024 bits parait-elle suffisante ?

Exercice 7 : Sécurité du codage 2
Le choix de c et de s est aussi important. Pour le comprendre, prenons $p = 11$ et $q = 13$. Représentez pour différentes valeurs de c les points $(a, a^c \pmod n)$, plus le dessin obtenu est aléatoire, plus il sera difficile à une personne mal intentionnée de déchiffrer un message sans connaitre la clef. On pourra utiliser les instructions `seq` pour générer une suite de terme général exprimé en fonction d'une variable formelle, et `scatterplot(l)` qui représente le nuage de points donné par une liste `l` de couples de coordonnées. Observez en particulier les cas où c n'est pas premier avec $\varphi(n)$ et $c = 3$. Conclusions ?

Exercice 8 : Primalité

6. `https://www-fourier.ujf-grenoble.fr/ parisse/mat249/rsa2`

Pour créer une paire de clefs, il faut générer des nombres premiers et donc être capable de déterminer si un nombre est premier ou non. Un test simple consiste à appliquer la contrapposée du petit théorème de Fermat : si $a^p \neq a \pmod p$, alors p n'est pas premier. Ecrire une fonction prenant en argument a et p et renvoyant 0 si p n'est pas premier et 1 si $a^p = a \pmod p$, puis une fonction prenant en argument p et effectuant le test pour toutes les valeurs de $a \in [2, p-1]$ jusqu'à ce que le test échoue. Si tous le tests réussissent, p est peut-être premier mais ce n'est pas certain : existe-t-il un nombre non premier pour lequel tous les tests réussissent ?

3.2.3 Partage de secret

On représente un secret par un entier s. On souhaite transmettre ce secret à n personnes. Une première méthode consiste à découper s en plusieurs parties, par exemple en écrivant s dans une base convenable, ou bien en donnant la valeur de s modulo plusieurs nombres premiers (reconstruction par le lemme chinois). Cette méthode est peu résistante, d'une part parce que la connaissance d'une partie des morceaux peut permettre la reconstruction de s en testant toutes les valeurs possibles des morceaux manquants, d'autre part parce qu'on ne peut pas se prémunir contre des erreurs (ou un ou deux morceaux manquants).

Une deuxième méthode, plus résistante, consiste si $s \in K = \mathbb{Z}/p\mathbb{Z}$ avec p premier (ou K un corps fini non premier) à choisir aléatoirement $n-1$ éléments de K, pour construire un polynôme P de degré $n-1$ ayant s comme coefficient constant. On calcule ensuite pour chacun des n morceaux la valeur de $t_k = P(x_k)$ en n abscisses distinctes 2 à 2 fixées x_k, le morceau de secret est alors t_k. On peut alors reconstruire P avec les t_k par interpolation de Lagrange et retrouver $s = P(0)$. Si le polynôme P est de degré plus petit, on peut reconstruire P et donc s avec un nombre plus faible de morceaux du secret (voire éliminer des morceaux de secrets invalides).

Instruction en Xcas : `convert(.,base,.)`, `ichinrem`, `horner`, `lagrange`.

Pour en savoir plus sur toute cette section, cf. par exemple Demazure, Gilles Zémor, le manuel de programmation de Xcas.

4 Analyse

Référence : manuel algorithmes de Xcas, le livre de Demailly. Exposés : 201, 208, 217, 220, 225, 235, **251**, **254**, **256**, 262
Exercices : 403 (voire 401 à 404), 406, 417, **421**, 428, 429, 430, **432**, 440, 441, **443**, **444**, **455**

4.1 Dichotomie

Recherche de solutions de $f(x) = 0$ sur $[a, b]$ sachant que $f(a)f(b) < 0$.

Solution : cf. la section 7.5 ou dans Xcas pour Firefox dans Doc, Exemple lycée ou dans le manuel Algoseconde [7]

7. `https://www-fourier.ujf-grenoble.fr/~parisse/algoseconde.html`

4.2 Méthode du point fixe

Recherche de solutions de $f(x) = x$ par étude de la suite $u_{n+1} = f(u_n)$. Cf. la section 7.5 Exemple, résolution de l'équation du temps en mécanique céleste $x - e\sin(x) = y,\ e \in [0, 1[$.

Exercice 1

Écrire un programme `iter` prenant en argument la fonction f, la valeur de u_0, de N et de ε, et qui s'arrête dès que l'une des conditions suivantes est satisfaite :

- $|u_{n+1} - u_n| < \varepsilon$
- le nombre d'itérations dépasse N.

Dans le premier cas le programme renverra la valeur de u_{n+1}, dans le second cas une séquence composée de u_N et de N.

Tester votre programme avec $f(x) = \sqrt{2 + x}$ et $f(x) = x^2$.

On suppose que la fonction f satisfait aux hypothèses du théorème du point fixe. On notera $k < 1$ la constante de contractance.

On peut alors trouver un encadrement de la limite l de la suite (u_n) en fonction de u_n, u_{n-1} et k.

Exercice 2

Écrire un programme `iter_k` prenant en argument la fonction f, la valeur de u_0, la constante k et l'écart toléré ε, et qui s'arrête dès que $|u_n - l| \leq \varepsilon$.

Vérifier les hypothèses du théorème du point fixe pour $f(x) = 3\cos(x/4)$ sur $[0, 3]$ et expliciter une constante de contractance k. Déterminer une valeur approchée de la limite de (u_n) à 10^{-3} près en utilisant la fonction `iter_k`.

La convergence de ces suites est en général linéaire*, le nombre de décimales exactes augmente de la même valeur à chaque itération. Par contre lorsqu'on est prêt d'une racine, la méthode de Newton permet en gros de multiplier par deux le nombre de décimales à chaque itération.*

Exercice 3

1. Donner une suite itérative obtenue par la méthode de Newton convergeant vers $\sqrt{5}$.
2. Montrer que la fonction $f : \mathbb{R}_+ \to \mathbb{R}_+$ définie par

$$f(x) = \frac{5x + 5}{x + 5}$$

   admet $\sqrt{5}$ pour point fixe. Trouver un intervalle I contenant $\sqrt{5}$ sur lequel les hypothèses du théorème du point fixe sont satisfaites et expliciter une constante de contractance k.
3. Comparer au tableur la vitesse de convergence des 10 premiers termes des deux suites (calculez les avec par exemple 100 chiffres significatifs et faites les différence entre 2 termes successifs).

4. En utilisant la fonction `iter`, trouver un encadrement de $\sqrt{5}$ à 10^{-6} près par les deux méthodes (on pourra prendre une valeur initiale approchée puis entière exacte pour avoir une valeur numérique approchée puis une fraction). Combien d'itérations sont nécessaires ?

Dans certains cas, la fonction f n'est pas contractante, mais on peut réécrire l'équation à résoudre sous une autre forme avec une fonction contractante, par exemple en utilisant une fonction réciproque.

Exercice 4

Donner un encadrement à 10^{-6} près d'une racine de l'équation $\tan(x) = x$ sur l'intervalle $]5\pi/2, 7\pi/2[$ en utilisant une méthode de point fixe.

Solution (point fixe) :

```
fonction iter(f,u0,N,eps)
  local u,v,j;
  u:=evalf(u0); // pas de calcul exact
  pour j de 1 jusque N faire
    v:=f(u);
    si abs(u-v)<eps alors return v; fsi;
    u:=v;
  fpour;
  return "pas de convergence";
ffonction:;
```

```
f(x):=cos(x):; iter(f,0.0,100,1e-8);
```

$$\text{Done}, 0.739085136647$$

En Python :

```
def fixe(f,u0,N,eps):
  # local j,u1
  u0=u0*1.0
  for j in range(N):
    u1=f(u0)
    if abs(u1-u0)<eps*abs(u0):
      return u1,j
    u0=u1
  return "Pas de convergence"

#from math import *f=lambda x:cos(x)fixe
(f,0.0,100,1e-8)
```

$$\#\text{frommathimport} * \text{f} = \text{lambdax} : \cos(\text{x})\text{fixe}(\text{f}, 0.0, 100, 1\text{e} - 8)$$

4.3 Méthode de Newton

$$u_{n+1} = u_n - \frac{f(u_n)}{f'(u_n)}$$

Programmation de la méthode de Newton (voir Xcas en ligne, Exemples). Utilisation de la convexité pour prouver la convergence (voir le manuel Algorithmes de Xcas).

Exemple : racine carrée, recherche des racines d'un polynôme (illustration avec l'utilisation du calcul formel pour prouver la convexité), bassins d'attraction des racines.

Solution (Newton) :

```
fonction newt(f,u0,N,eps)
  local u,v,g,j;
  u:=evalf(u0); // pas de calcul exact
  g:=id-f/f'; // fonction a iterer
  pour j de 1 jusque N faire
    v:=g(u);
    si abs(u-v)<eps alors return v; fsi;
    u:=v;
  fpour;
  return "pas de convergence";
ffonction:;
```

```
f(x):=x^2-2:; newt(f,2,100,1e-8);
```

Done, 1.41421356237

Python n'étant pas un langage formel on ne peut pas faire tourner l'équivalent (en tout cas sans addition de modules externe). On peut bien sur saisir un programme en syntaxe Python dans Xcas (par exemple en traduisant le programme ci-dessus avec la commande `python(newt)`).

4.4 Intégration numérique

Rectangles, trapèzes, Simpson. Programmation ou/et illustration. Majoration de l'erreur (illustration possible avec calcul formel pour calcul exact de l'intégrale). Exemple : la méthode des trapèzes

```
fonction trap(f,a,b,N)
  // trapezes pour la fonction f sur [a,b] N subdivisions
  local h,j,r;
  h:=evalf((b-a)/N);
  r:=1/2*(f(a)+f(b));
  pour j de 1 jusque N-1 faire
    r += f(a+j*h);
  fpour;
  return h*r;
ffonction:;
```

On teste avec $f(x) = \exp -x^2$ sur $[0,1]$ pour différentes valeurs de N, par exemple 100 et 1000, observez que l'erreur est grosso-modo proportionnelle à $1/N^2$:

```
f(x):=exp(-x^2):; N:=100; T:=trap(f,0,1,N
); T-int(f(x),x,0,1);
```

$$\text{Done}, 100, 0.746818001468, -6.13134456273e-06$$

En Python :

```
def trap(f,a,b,N):
    # local h,j,r
    h=(b-a)/(1.0*N) # 1.0 permet la compatibilite Python 2 et 3
    r=1/2*(f(a)+f(b))
    for j in range(1,N):
        r+=f(a+j*h)
    return h*r

#from math import *f=lambda x:exp(-x*x)N=100T=trap
(f,0,1,N)
```

$$\#\text{frommathimport} * \text{f} = \text{lambdax} : \exp(-\text{x} * \text{x})\text{N} = 100\text{T} = \text{trap}(\text{f}, 0, 1, \text{N})$$

Exercice 1 : Calculer une valeur approchée de

$$\int_0^1 \frac{dx}{1+x}$$

par la méthode des rectangles, du point milieu et des trapèzes en utilisant un pas de $1/10$ et de $1/100$ (vous pouvez la fonction `plotarea` ou utiliser le tableur ou écrire un programme effectuant ce calcul avec comme arguments la fonction, la borne inférieure, la borne supérieure et le nombre de subdivision). Observez numériquement la différence entre les valeurs obtenues et la valeur exacte de l'intégrale.

Exercice 2

Calculer le polynôme interpolateur P de Lagrange de $f(x) = \frac{1}{1+x^2}$ aux points d'abscisse $\frac{j}{4}$ pour j variant de 0 à 4. Donner un majorant de la différence entre P et f en un point $x \in [0,1]$. Représenter graphiquement ce majorant. Calculer une majoration de l'erreur entre l'intégrale de f et l'intégrale de P sur $[0,1]$. En déduire un encadrement de $\pi/4$.

Exercice 3

On reprend le calcul de $\int_0^1 \frac{dx}{1+x}$ mais en utilisant un polynôme interpolateur de degré 4 sur N subdivisions de $[0,1]$ (de pas $h = 1/N$). Déterminer une valeur de N telle que la valeur approchée de l'intégrale ainsi calculée soit proche à 10^{-8} près de $\ln(2)$. En déduire une valeur approchée à 10^{-8} de $\ln(2)$.

Même question pour $\int_0^1 \frac{dx}{1+x^2}$ et $\pi/4$ (pour majorer la dérivée n-ième de $\frac{1}{1+x^2}$, on pourra utiliser une décomposition en éléments simples sur $\mathbb{C}$).

4.5 Équations différentielles

Résolution numérique : méthode d'Euler explicite (ou implicite) illustration avec `interactive_odeplot` ou/et programmation.

```
fonction Euler(f,t0,t1,N,y0)
  // y'=f(t,y) avec f(t0)=y0, calcul de f(t1)
  local h,y,j;
  h:=(t1-t0)/N; // pas
  y:=evalf(y0);
  pour j de 0 jusque N-1 faire
    y += h*f(t0+j*h,y); // y(t+h)=y(t)+h*y'
  fpour;
  return y;
ffonction:;
```

```
f(t,y):=y; Euler(f,0,1.0,100,1);exp(1.0);
```

$$t, y \to y, 2.70481382942, 2.71828182846$$

En Python

```
def Eulerpy(f,t0,t1,N,y0):
  # local j,h,t,y
  h=(t1-t0)/(N*1.0)
  y=y0
  for j in range(N):
    t = t0+j*h
    y += f(t,y)*h
  return y
```

```
#from math import *f=lambda t,y:yEuler(f,0,1,100,1
)exp(1.0)
```

$$\#\text{frommathimport} * \text{f} = \text{lambdat, y} : \text{yEuler(f, 0, 1, 100, 1)exp(1.0)}$$

Voir la section 14 du manuel Algorithmes [8] qui contient aussi des méthodes de résolution exacte d'équations différentielles, et des exemples venant de la physique.

Exemple : allure des solutions d'un système différentiel linéaire à coefficients constants :

```
A:=[[-1,1],[1,2]];p,d:=jordan(A)
```

$$\begin{pmatrix} -1 & 1 \\ 1 & 2 \end{pmatrix}, \begin{pmatrix} \sqrt{13}-3 & -\sqrt{13}-3 \\ 2 & 2 \end{pmatrix}, \begin{pmatrix} \frac{(\sqrt{13}+1)}{2} & 0 \\ 0 & \frac{(-\sqrt{13}+1)}{2} \end{pmatrix}$$

```
A:=[[1,-1],[2,4]];p,d:=jordan(A)
```

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

8. https://www-fourier.ujf-grenoble.fr/~parisse/algo.html

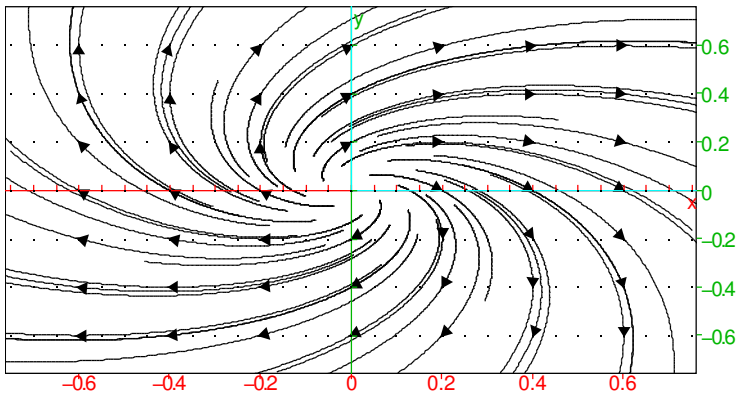
```
A:=[[0,1],[-1,0]];p,d:=jordan(A)
```

$$\begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}, \begin{pmatrix} -i & i \\ -1 & -1 \end{pmatrix}, \begin{pmatrix} -i & 0 \\ 0 & i \end{pmatrix}$$

```
A:=[[1,1],[-1,1]];p,d:=jordan(A)
```

$$\begin{pmatrix} 1 & 1 \\ -1 & 1 \end{pmatrix}, \begin{pmatrix} -i & i \\ -1 & -1 \end{pmatrix}, \begin{pmatrix} 1-i & 0 \\ 0 & 1+i \end{pmatrix}$$

```
seq(seq(plotparam(exp(A*t)*[a/5,b/5],t=-1..1,affichage=arrow_line
),a,-3,3),b,-3,3)
```



L'étude qualitative près d'un point d'équilibre d'un système autonome se fait en linéarisant en ce point.

4.6 Propriétés métriques des courbes

Voir la section 10 du manuel Algorithmes de Xcas.

4.7 Séries entières

Exercice 1.

1. Rappeler le développement de Taylor $T_{2n}(x)$ au voisinage de $x = 0$ de $f(x) = \cos(x)$ à l'ordre $2n$.
2. Tracer sur un même graphique les graphes des fonctions f et T_2, T_4, T_6, T_8
3. Graphiquement on voit que $T_8(x)$ approche $\cos(x)$: sur quel intervalle cette approximation vous paraît-elle acceptable ?
4. Donner une majoration du reste $R_{2n}(x)$ du développement de Taylor de f à l'ordre $2n$, où $f(x) = T_{2n}(x) + R_{2n}(x)$.
5. On prend $T_8(x)$ comme valeur approchée de $\cos(x)$ pour $x \in [-1, 1]$.
   Donner une majoration indépendante de x de l'erreur commise.
   (A titre d'illustration, tracer la différence $T_8(x) - \cos(x)$.)

6. En déduire un encadrement de $\cos(1)$.

Exercice 2. On veut approcher $\sin(x)$ à `1e-6` près en utilisant des développements en séries entières.

1. Déterminer le plus petit k tel que :

$$T_{2k+1}(x) = \sum_{j=0}^{k} (-1)^j \frac{x^{2j+1}}{(2j+1)!}$$

réalise cette approximation sur $[0, \pi/4]$.

2. Écrire une fonction qui calcule une valeur approchée à `1e-6` de $\sin(x)$ sur $[-100, 100]$ en justifiant et en effectuant les étapes suivantes :
   - on retire un multiple entier de π (obtenu par arrondi de x/π) pour se ramener à l'intervalle $[-\pi/2, \pi/2]$ (on discutera sur l'erreur commise)
   - si x est négatif, on remplace x par $-x$ (que devient $\sin(x)$?)
   - lorsque $x \in [0, \pi/4]$, on utilise le développement en séries ci-dessus.
   - lorsque $x \in [\pi/4, \pi/2]$, on se ramène au développement de l'exercice précédent en appliquant la formule $\sin(x) = \cos(\pi/2 - x)$.
3. Afin de tester votre fonction f et d'attraper d'éventuelles erreurs grossières, faites afficher le graphe de f, disons sur l'intervalle $[-10, 10]$, puis le graphe de la différence $f - \sin$ où $\sin$ est la fonction déjà implémentée dans Xcas.

Exercice 3

1. Pour $x > 0$ exprimer $\arctan(-x)$ en fonction de $\arctan(x)$. Calculer la dérivée de $\arctan(x) + \arctan(1/x)$, en déduire $\arctan(1/x)$ en fonction de $\arctan(x)$ pour $x > 0$. Montrer que le calcul de $\arctan(x)$ sur $\mathbb{R}$ peut se ramener au calcul de $\arctan(x)$ sur $[0, 1]$.
2. Rappeler le développement en séries entières de $\arctan(x)$ en $x = 0$, et son rayon de convergence. Soit $\alpha \in [0, 1]$, montrer que

$$\alpha - \frac{\alpha^3}{3} + \frac{\alpha^5}{5} - \frac{\alpha^7}{7} \leq \arctan(\alpha) \leq \alpha - \frac{\alpha^3}{3} + \frac{\alpha^5}{5}$$

en déduire que la méthode de Newton appliquée à l'équation $\tan(x) - \alpha = 0$ avec comme valeur initiale $\alpha - \frac{\alpha^3}{3} + \frac{\alpha^5}{5}$ est une suite décroissante qui converge vers $\arctan(\alpha)$.
3. Déterminez par cette méthode une valeur approchée à `1e-8` près de $\pi/4 = \arctan(1)$.
4. On pourrait calculer $\pi/4$ avec la même précision en utilisant le développement en séries de la formule de Machin

$$\frac{\pi}{4} = 4\arctan(\frac{1}{5}) - \arctan(\frac{1}{239})$$

Combien de termes faudrait-il calculer dans le développement des deux arctangentes ?

Exercice 4

1. Écrire le développement en séries entières au voisinage de $x = 0$ de :

$$g(x) = \frac{e^{-x} - 1}{x}$$

2. On veut calculer une valeur approchée de

$$I = \int_0^1 g(x)\, dx$$

   En utilisant le développement de g, écrire I sous la forme d'une série $\sum_{j=0}^{\infty} v_j$.

3. Soit $R_n = \sum_{j=n+1}^{\infty} v_j$ le reste de cette série. Donner une majoration de $|R_n|$.

4. En déduire un encadrement de I faisant intervenir $\sum_{j=0}^{n} v_j$. Calculer explicitement cet encadrement lorsque $n = 10$.

4.8 Accélération de convergence

Méthode de Richardson lorsqu'on connait le développement asymptotique d'une suite convergeant vers la limite. Par exemple pour la constante d'Euler, cf. le manuel de programmation de Xcas.

Cas particulier : convergence de la méthode de trapèzes : méthode de Romberg pour approcher numériquement une intégrale.

Autres : méthode du Δ^2 d'Aitken, séries alternées (théorie voir le texte 585 [9] de l'option C de l'agrégation externe, cf. la session du menu Exemples, analyse de Xcas).

5 Algèbre linéaire

Référence : dans Xcas, menu Aide, puis manuel, puis Programmation (pour le pivot de Gauss) ou Algorithmes de calcul formel section 20.

Exposés : 110, **114**, 150, 151, 155, 156
Exercices : 310, 313, **319**, 348, **350**, 357

5.1 Pivot de Gauss et applications

5.2 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

9. `http://agreg.dnsalias.org/Textes/585.pdf`

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)

Voir le manuel de programmation de Xcas pour une solution.

5.3 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}. On peut aussi utiliser la factorisation LU de la matrice A et résoudre les n systèmes avec cette factorisation.
Écrire un programme de calcul d'inverse de matrice par cet algorithme.

5.4 Noyau d'une application linéaire

On présente ici deux méthodes, la première se généralise au cas des systèmes à coefficients entiers, la deuxième utilise un peu moins de mémoire (elle travaille sur une matrice 2 fois plus petite).

Première méthode Soir M la matrice dont on cherche le noyau. On ajoute à droite de la matrice transposée de M une matrice identité ayant le même nombre de lignes que M^t. On effectue une réduction sous-diagonale qui nous amène à une matrice composée de deux blocs

$$(M^t I_n) \ \rightarrow \ (U \tilde{L})$$

Attention, $\tilde{L}$ n'est pas la matrice L de la décomposition LU de M^t, on a en fait

$$\tilde{L} M^t = U$$

donc

$$M \tilde{L}^t = U^t$$

Les colonnes de $\tilde{L}^t$ correspondant aux colonnes nulles de U^t (ou si on préfère les lignes de $\tilde{L}$ correspondant aux lignes nulles de U) sont donc dans le noyau de M et réciproquement si $Mv = 0$ alors

$$U^t(\tilde{L}^t)^{-1}v = 0$$

donc, comme U est réduite, $(\tilde{L}^t)^{-1}v$ est une combinaison linéaire des vecteurs de base d'indice les lignes nulles de U. Finalement, les lignes de $\tilde{L}$ correspondant aux lignes nulles de U forment une base du noyau de M.

On peut faire le raisonnement ci-dessus à l'identique si M est une matrice à coefficients entiers, en effectuant des manipulations élémentaires réversibles dans $\mathbb{Z}$, grâce à l'idendité de Bézout. Si a est le pivot en ligne i, b le coefficient en ligne j à annuler, et u, v, d les coefficients de l'identité de Bézout $au + bv = d$ on fait les changements :

$$L_i \leftarrow uL_i + vL_j, \quad L_j \leftarrow -\frac{b}{d}L_i + \frac{a}{d}L_j$$

qui est réversible dans $\mathbb{Z}$ car le déterminant de la sous-matrice élémentaire correspondante est

$$\begin{vmatrix} u & v \\ -\frac{b}{d} & \frac{a}{d} \end{vmatrix} = 1$$

Cette réduction (dite de Hermite) permet de trouver une base du noyau à coefficients entiers et telle que tout élément du noyau à coefficient entier s'écrit comme combinaison linéaire à coefficients entiers des éléments de la base.

Deuxième méthode On commence bien sûr par réduire la matrice (réduction complète en-dehors de la diagonale), et on divise chaque ligne par son premier coefficient non nul (appelé pivot). On insère alors des lignes de 0 pour que les pivots (non nuls) se trouvent sur la diagonale. Puis en fin de matrice, on ajoute ou on supprime des lignes de 0 pour avoir une matrice carrée de dimension le nombre de colonnes de la matrice de départ. On parcourt alors la matrice en diagonale. Si le i-ième coefficient est non nul, on passe au suivant. S'il est nul, alors tous les coefficients d'indice supérieur ou égal à i du i-ième vecteur colonne v_i sont nuls (mais pas forcément pour les indices inférieurs à i). Si on remplace le i-ième coefficient de v_i par -1, il est facile de se convaincre que c'est un vecteur du noyau, on le rajoute donc à la base du noyau. On voit facilement que tous les vecteurs de ce type forment une famille libre de la bonne taille, c'est donc bien une base du noyau.

5.5 Factorisation $PA = LU$

Voir l'explication de la factorisation LU section 20.4 du manuel Algorithmes de Xcas.
La commande `lu` de Xcas renvoie `P,L,U` où `P` code la permutation P (on donne la liste des images des entiers de 0 à $n-1$). La commande `linsolve(P,L,U,b)` permet de résoudre le système linéaire $Ax = b$ en utilisant la décomposition LU de A donc en effectuant $O(n^2)$ opérations (résolution de deux systèmes triangulaires $Ly = Pb$ et $Ux = y$).

5.6 Algorithme de Gauss-Bareiss (hors programme)

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)

5.7 Exercices

Exercice 1

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 2

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 $E_1 \cap E_2$. 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 3

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.

5.8 Déterminants

Calcul par réduction. Autres algorithmes (à la limite du programme) : développement intelligent (2^n opérations), modulaire (si les coefficients sont dans $\mathbb{Z}$).

5.9 Polynome minimal, 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.

5.9.1 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.

5.9.2 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).

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.

Algorithme de Fadeev (hors programme, référence, manuel algorithmes de calcul formel).

5.10 Méthode de la puissance

Si la matrice M possède une seule valeur propre de module maximal, alors la suite $v_{n+1} = Mv_n$ se rapproche de l'espace propre correspondant. En pratique, on normalise v_n pour éviter les débordements numériques. Écrire un programme mettant en oeuvre la méthode de la puissance dans le cas réel (penser à normaliser v_n pour éviter les overflow). Utilisez ce programme pour trouver une valeur approchée de la valeur propre de norme maximale par la méthode de la puissance de $B\,{}^tB$ où B est une matrice aléatoire. Trouver la plus grande racine en module d'un polynôme en appliquant à la matrice companion.

Solution :

```
fonction pui(A,eps,N)
  // A la matrice, eps la precision, N nombre maximum d'iterations
  local j,n,v,w,l;
  n:=size(A); // taille de la matrice
  v:=ranv(n,uniformd,-1,1); // vecteur aleatoire
  v:=normalize(v); // de norme 1
  pour j de 1 jusque N faire
    w:=A*v;
    l:=dot(w,v); // valeur propre si v vecteur propre normalise
    si l2norm(w-l*v)<eps alors return l,v,j; fsi;
    v:=normalize(w);
  fpour;
  return "pas de convergence";
ffonction:;

A:=ranm(5,5); B:=A+trn(A); l,v,j:=pui(B,1e-4,10000
);
```

$$\begin{pmatrix} -50 & -9 & -44 & -66 & -44 \\ 45 & 11 & 62 & -72 & 71 \\ 44 & 79 & 11 & 77 & 94 \\ -19 & 35 & -92 & -15 & 89 \\ 67 & -46 & -22 & -63 & 14 \end{pmatrix}, \begin{pmatrix} -100 & 36 & 0 & -85 & 23 \\ 36 & 22 & 141 & -37 & 25 \\ 0 & 141 & 22 & -15 & 72 \\ -85 & -37 & -15 & -30 & 26 \\ 23 & 25 & 72 & 26 & 28 \end{pmatrix}, 200.694582552, [0.147257282996, 0.62$$

5.11 Factorisation QR

La factorisation QR d'une matrice (écriture comme produit d'une matrice orthogonale/unitaire et d'une matrice triangulaire supérieure) sert à calculer simultanément toutes les valeurs propres (approchées) d'une matrice, commande Xcas `qr`. L'idée est de poser $A = QR$ puis $A_1 = RQ$ et de recommencer, on peut montrer qu'il y a convergence vers une matrice (presque) diagonale. Pour optimiser les itérations, on utilise la méthode QR implicite ou algorithme de Francis sur la forme de Hessenberg de la matrice, c'est la commande `schur` qui fait ce calcul dans Xcas, `P,H:=schur(A)` renvoie dans `P` une matrice orthogonale (ou unitaire) et dans `H` une matrice (presque) diagonale telles que $A = PHP^{-1} = PHP^*$. Cet algorithme peut servir pour la lecon 150, et aussi pour la lecon 168 pour la matrice companion d'un polynome.

6 Autres idées d'algorithmes.

6.1 Permutations

Représentation par la liste des images ou par une liste de cycles, chaque cycle étant la liste des éléments du cycle. Composition, inverse, décomposition en cycles.

6.2 Méthode de Monte-Carlo

Il s'agit d'un algorithme probabiliste pour déterminer la valeur approchée d'une intégrale $\int_a^b f(t)\,dt$. Le principe : on tire au hasard selon la loi uniforme sur $[a,b]$ N réels $x_1, ..., x_N$, et on fait la moyenne des $f(x_i)$. On a convergence pour $N \to +\infty$ vers $1/(b-a)\int_a^b f(t)\,dt$. La convergence est lente en $1/\sqrt{N}$ donc cette méthode est peu utile en dimension 1, par contre en dimension grande, elle est beaucoup plus efficace que des quadratures classiques. Illustration avec $\int_0^2 e^{-t^2}\,dt$

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

$$x \to e^{-x^2}$$

```
mu:=1/2*int(f(x),x,0,2.0);
```

$$0.441040695381$$

On fait une expérience avec $N = 400$

```
N:=400; v:=ranv(N,uniformd,0,2):; v[0..3]
```

$$400, \text{Done}, [1.9582536174, 1.06113011762, 0.0715601900592, 1.97895547841]$$

```
w:=map(v,f):; w[0..3]
```

Done, $[0.0216065832626, 0.324328909111, 0.994892228454, 0.019915344071]$

```
mean(w); mean(w)-mu;
```

$$0.442084539724, 0.00104384434312$$

On fait une série d'expériences (500 ici) et on trace l'histogramme des valeurs approchées, que l'on compare avec le graphe de la loi normale de moyenne μ et d'écart-type

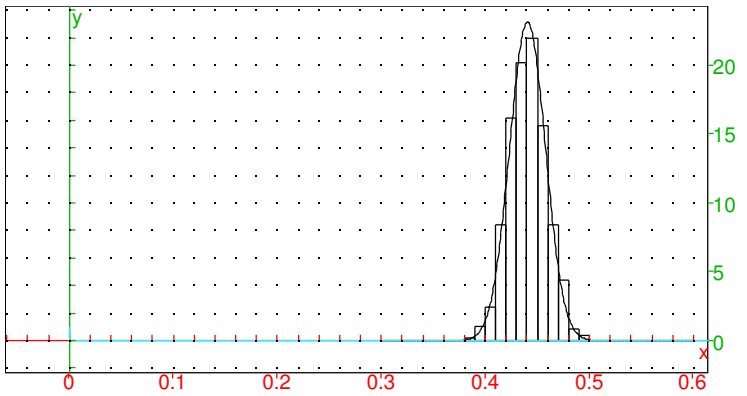
```
s:=sqrt(1/2*int(f(x)^2,x,0,2.0)-mu^2)/sqrt(N)
```

$$0.0172330926083$$

```
l:=seq(mean(map(ranv(N,uniformd,0,2),f)),j,1,500
):;
```

Done

```
histogramme(l,0.3,0.01);plot(loi_normale
(mean(l),s,x),x=0.3..0.6)
```



En pratique, on estime s par `stddevp(w)/sqrt(N)`

6.3 Calcul de π par polygones

Encadrement de π par un polygone régulier inscrit et par un polygone extérieur.

6.4 Calcul probabiliste de π

En prenant un couple de nombres aléatoires entre -1 et 1, on regarde si le couple est dans le disque de rayon 1, et on calcule la proportion.

6.5 Fluctuations

6.5.1 Loi discrète

Expérience : on tire n fois au hasard une boule dans une urne avec remise, la proportion de boules noires dans l'urne étant p (proche de 0.5), on compte le nombre de boules noires. On effectue N fois l'expérience. On compte le nombre de fois où la proportion de boules noires observée est dans un intervalle centré en p de taille proportionnelle à $1/\sqrt{n}$ (par exemple $[p - 1/\sqrt{n}, p + 1/\sqrt{n}]$). On peut aussi représenter graphiquement par des segments horizontaux les intervalles proportion de boules observées $\pm 1/\sqrt{n}$ et tracer verticalement la droite d'abscisse p.

6.5.2 Loi continue

Expérience : on ajoute n réels tirés au hasard pris entre 0 et 1 (loi uniforme, ou autre loi), par exemple pour $n = 30$. On effectue N fois l'expérience (par exemple $N = 1000$), et on trace un histogramme des résultats, et sur le même graphique on représente la loi normale de moyenne l'espérance de la loi et d'écart-type l'écart-type de la loi divisé par $\sqrt{n}$.

6.6 Tracé de courbes

Par discrétisation.

7 Exemples d'exercices pour l'oral 2.

Remarques :

- Les exercices de cette section présentent chacun un algorithme, ces algorithmes ont été choisis en raison de leur importance, ils sont donc par nature génériques et plus théoriques que des exercices standards, puisqu'ils implémentent une partie du cours ou un complément de cours, de ce fait on a aussi indiqué les leçons qui correspondent aux algorithmes présentés.
- On a fait le choix d'algorithmes effectivement implémentables par un candidat pendant le temps de préparation pour un exercice qu'il choisirait de développer. Un candidat moins à l'aise sur machine peut très bien décider de présenter un algorithme sans l'implémenter effectivement en écrivant l'algorithme au tableau et en l'illustrant sur machine par l'instruction intégrée du logiciel. Il est aussi

possible de présenter une implémentation non mise au point, en insistant sur le fait que les conditions de préparation (temps, stress) ne sont pas des conditions favorables à la mise au point (et les candidats doivent être particulièrement attentifs à ne pas perdre plus de quelques minutes au débogage, au risque de paniquer et de ne pas préparer correctement les autres exercices).
– Les implémentations sont proposées en syntaxe en français pour Xcas (donc en language algorithmique mais qui de plus fonctionne) et/ou en syntaxe Python.

7.1 Écriture en base 2 et puissance rapide modulaire.

Numéros de leçons correspondants : 302, éventuellement 303. Exposés : 103.

Énoncé : : le but de l'exercice est de calculer rapidement a^n modulo p où a, n, p sont 3 entiers avec $0 \leq a < p$.

1. Soit n un entier et
$$n = \sum_{j=0}^{k} n_j 2^j, \quad n_j \in \{0, 1\}$$
sa décomposition en base 2. Écrire un algorithme qui renvoie la liste des n_j.

2. On calcule $a_j = a^{2^j}$ modulo p successivement pour $j = 2..k$ par
$$a_j = a^{2^j} \pmod p = (a^{2^{j-1}} \pmod p)^2 \pmod p$$
Calculer $a^n \pmod p$ en fonction des a_j et n_j. Quel est le nombre d'opérations à effectuer ? Comparer avec le nombre d'opérations à effectuer en faisant $a \times a \times ... \times a \pmod p$.

3. Écrire un algorithme permettant de calculer a^n modulo p par cet algorithme.

Solution :

1. On observe que n_0 est le reste de la division euclidienne de n par 2, n_1 s'obtient à partir du quotient euclidien de n par 2 de la même façon que n_0 à partir de n et donc si on remplace n par le quotient euclidien de n par 2, n_1 est alors le reste de la division euclidienne de n par 2 ... D'où le programme Xcas/Python :

```
def base2(n):
    # local l
    l=[]
    while n>0 :
        l.append(n
        n = n // 2
    return l[::-1] # liste l a l'envers
```

L'instruction intégrée Xcas permettant de faire la conversion est `convert(n,base,2)`, elle renvoie la liste des bits à l'envers.

2. On a

$$\begin{aligned} a^n \pmod p &= a^{\sum_{j=0}^{k} n_j 2^j} \pmod p \\ &= \prod_{j=0}^{k} a_j^{n_j} \pmod p \\ &= \prod_{0\le j\le k, n_j=1} a_j \pmod p \end{aligned}$$

Il faut effectuer $k-1$ opérations d'élévation au carré suivi de réduction modulo p pour calculer les a_j, puis effectuer le produit de a_j pour les $n_j = 1$. Au maximum on fait $2k-1$ produits suivi de réduction modulo p, où k est la partie entière du log en base 2 de n. Ce qui nécessite beaucoup moins d'opérations que de calculer $n-1$ produits suivi de réduction modulo p.
Il est essentiel d'effectuer les réductions modulo p après chaque produit et non pas seulement à la fin, sinon la taille des entiers à multiplier augmenterait considérablement, et donc le temps nécessaire pour effectuer une multiplication. Ici, toutes les multiplications se font avec des nombres plus petits que p, en temps constant donc.
3. On pourrait utiliser la fonction `base2` précédente, on peut aussi calculer simultanément les n_j et $a^n \pmod p$. En syntaxe Xcas/Python :

```
def puimod(a,n,p):
    # local aj,an_modp
    aj=a
    an_modp=1
    while n>0 :
        if n
            an_modp=an_modp*aj
        n=n // 2
        aj=aj*aj
    return an_modp
```

Commentaires

- L'exercice peut se prolonger par les applications comme le système de cryptographie RSA, voir aussi l'exercice suivant sur les tests de primalité.
- La décomposition en base 2 peut se faire au lycée, la suite également mais il est difficile d'en justifier l'intérêt (sauf en TS spécialité maths).

7.2 Primalité et petit théorème de Fermat

Numéros de leçons correspondants : 305, 302. Exposés : 104, 103.
Énoncé :

1. Soit p un nombre premier. En utilisant la formule du binôme de Newton, montrer par récurrence que a^p est congru à a modulo p.
2. Écrire un algorithme qui teste pour quelques valeurs de a si a^p est congru à a modulo p, on renverra a si p n'est pas premier, et 0 sinon. Que signifie la valeur 0 ?

3. Écrire un algorithme qui détermine le premier entier p non premier tel que pour tout a premier avec p, a^p est congru à a modulo p (nombre de Carmichael). On pourra utiliser la fonction `isprime` pour tester si un entier est premier.

Solution :

1. On fait une démonstration par récurrence. On a $0^p = 0$ est bien congru à 0 modulo p. La formule du binôme donne :

$$(a+1)^P = \sum_{j=0}^{p} \binom{p}{j} a^j = 1 + a^p + \sum_{j=1}^{p-1} \binom{p}{j}$$

Dans la somme, les coefficients binomiaux valent

$$\frac{\prod_{k=0}^{j-1}(p-k)}{j!}$$

et sont donc divisibles par p puisque le numérateur l'est, le dénominateur ne l'est pas et p est premier. Donc $(a+1)^P$ est congru à $a^p + 1$ modulo p donc à $a+1$ modulo p par hypothèse de récurrence. 2. Le programme utilisable avec Xcas ou Python

```
def test(p,nmax):
    # local a
    if nmax>=p-1:
        nmax=p-1
    for a in range(2,nmax+1):
        if pow(a,p,p)!=a :
            return(a)
    return 0
```

On utilise ici la fonction `pow` avec 3 arguments (puissance modulaire rapide, synonyme de `powmod` ici). Le paramètre `nmax` sert à limiter le nombre d'essais, ici on essaie les entiers de 2 à `nmax`, mais on pourrait aussi tirer `nmax` entiers au hasard. La boucle sur a se poursuit jusqu'à ce que le test de Fermat échoue, `return` provoque en effet l'arrêt prématuré de la fonction (donc de la boucle) ou va à son terme (test réussi, valeur de retour 0). Exemple

```
test(12345,7);test(101,3)
```

$$2, 0$$

Lorsque la réponse du programme est non nulle, on est assuré que le nombre n'est pas premier, la valeur renvoyée est un certificat de non-primalité (un entier tel que $a^p \neq a \pmod p$). Par contre, si la valeur renvoyée est 0, on ne peut pas en déduire que p est premier (il y a seulement des présomptions).

3. Pour trouver un nombre de Carmichael, on doit tester que $a^p = a \pmod p$ pour tous les entiers compris entre 2 et $p-1$ premiers avec p et on doit de plus tester que p n'est pas premier. Avec Xcas :

```
def carmi(nmax):
    # local a,p
    for p in range(3,nmax+1):
        for a in range(2,p):
            if gcd(a,p)==1 and pow(a,p,p)!=a :
                break # si le test de Fermat echoue on arrete la boucle sur a
        if a==p and not(isprime(p)) :
            return(a)
    return "non trouve"
```

On exécute par exemple

```
carmi(100)
```

nontrouve

qui renvoie "non trouvé" (il n'y a pas de nombre de Carmichael inférieur à 100) alors que `carmi(1000)` renvoie après quelques instants 561, le premier nombre de Carmichael. Pour exécuter ce programme en Python, il faut définir au préalable une commande gcd de calcul de PGCD de 2 entiers et une commande isprime qui teste la primalité d'un entier, et il faut faire varier `a in range(2,p+1)` (en effet si la boucle s'exécute sans interruption, la valeur de sortie de a varie de 1 entre Xcas et Python).

Commentaires :

- Cet exercice de niveau 1ère ou 2ème année universitaire est à la limite accessible en terminale S spécialité maths.
- On peut le prolonger par le test de pseudo-primalité de Miller-Rabin (cf. le manuel de programmation de Xcas pour l'algorithme, les justifications nécessitent du candidat plus de bagages mathématiques dans les corps finis) qui donne une bonne probabilité d'être premier s'il réussit pour plusieurs entiers contrairement au test de Fermat pour lesquels cet exercice construit des contre-exemples.
- L'intérêt de ces méthodes est de tester rapidement la primalité de grands entiers. On peut en effet montrer que le nombre d'opérations élémentaires à effectuer pour le test ci-dessus ou le test de Miller-Rabin est en $O(\ln(p)^3)$, alors que le test de divisibilité peut atteindre $O(\sqrt{p}\ln(p)^2)$.
- On peut aussi parler de certificats de primalité, comme l'exemple de la documentation de Xcas
  `pari("isprime",9856989898997789789,1).`

7.3 PGCD, PPCM

Numéros de leçons correspondants : 306, Exposés : 159, 157

Énoncé : Soient a et b 2 entiers. On pose $r_0 = a$,$r_1 = b$, et pour $n \geq 0$ tel que $r_{n+1} \neq 0$, on note q_{n+2} et r_{n+2} le quotient et le reste de la division euclidienne de r_n par r_{n+1}. On rappelle qu'il existe un (premier) entier N tel que $r_N = 0$, et que $\text{pgcd}(a, b) = r_{N-1}$ (dernier reste non nul).

1. On définit pour $n \in [0, N]$, les suites u_n et v_n par récurrence

   $$u_0 = 1, u_1 = 0, u_{n+2} = u_n - q_{n+2}u_{n+1}, \quad v_0 = 1, v_1 = 0, v_{n+2} = v_n - q_{n+2}v_{n+1}$$

   Montrer que
   $$au_n + bv_n = r_n$$

2. En déduire un algorithme permettant de déterminer des coefficients de l'identité de Bézout
   $$au + bv = au_{N-1} + bv_{N-1} = \text{pgcd}(a, b)$$

3. Adapter l'algorithme ci-dessus pour calculer l'inverse de a modulo b lorsque a et b sont premiers entre eux.

4. Déterminer en fonction de n la valeur de
   $$u_n r_{n+1} - u_{n+1} r_n$$
   En déduire que $|au_N| = |bv_N| = \text{ppcm}(a, b)$.

Solution

1. Par récurrence, au rang $n = 0$ et $n = 1$ c'est une conséquence de la définition de $r_0, r_1, u_0, u_1, v_0, v_1$. Pour $n \geq 2$,
   $$\begin{aligned} au_n + bv_n &= a(u_{n-2} - q_n u_{n-1}) + b(v_{n-2} - q_n v_{n-1}) \\ &= au_{n-2} + bv_{n-2} - q_n(au_{n-1} + bv_{n-1}) \\ &= r_{n-2} - q_n r_{n-1} \\ &= r_n \end{aligned}$$

2. On remarque que la relation de récurrence définissant r_n, u_n et v_n est identique, on utilisera une liste pour contenir ces 3 valeurs. On exécute une boucle (avec test d'arrêt sur la valeur de r_{n+1}), au cours de la boucle La liste `l0` contiendra les valeurs de r_n, u_n, v_n, `l1` les valeurs de $r_{n+1}, u_{n+1}, v_{n+1}$ et `l2` les valeurs de $r_{n+2}, u_{n+2}, v_{n+2}$, en fin de boucle l'incrémentation de n se traduit par la recopie de `l1` dans `l0` et de `l2` dans `l1`. Voir les programmes à la section 2.1.2.

3. L'inverse de a modulo b est u, le coefficient de Bézout de a dans
   $$au + bv = 1$$
   On remarque que la valeur des u_n se calcule indépendamment de la valeur des v_n, on peut donc écrire un algorithme de calcul d'inverse modulaire un peu plus rapide que l'algorithme de Bézout en ne calculant pas les v_n. (N.B. : c'est un des avantages de l'algorithme itératif présenté ici par rapport à l'algorithme récursif présenté par exemple dans le manuel de programmation de Xcas).

4. On a
   $$\begin{aligned} u_{n+1}r_{n+2} - u_{n+2}r_{n+1} &= u_{n+1}(r_n - q_n r_{n+1}) - (u_n - q_n u_{n+1})r_{n+1} \\ &= u_{n+1}r_n - u_n r_{n+1} \\ &= -(u_n r_{n+1} - u_{n+1} r_n) \end{aligned}$$

Donc

$$u_n r_{n+1} - u_{n+1} r_n = (-1)^n (u_0 r_1 - u_1 r_0) = (-1)^n b$$

Au rang $n = N - 1$, on a donc $u_N \text{pgcd}(a, b) = (-1)^N b$. En multipliant par a, on a alors

$$u_N a = (-1)^N \frac{ab}{\text{pgcd}(a, b)} = (-1)^N \text{ppcm}(a, b)$$

On peut faire de même avec v_N ou tout simplement utiliser

$$au_N + bv_N = 0$$

Bien entendu, on ne calculera les cofacteurs du PPCM de cette manière que si on doit calculer les coefficients de Bézout.

Commentaires

- Cet exercice peut être traité dès la Terminale S (spécialité maths) (raisonnement par récurrence, thème abordé) et dans le supérieur.
- On peut parler de la méthode de cryptage RSA (le calcul de la clef de décodage en fonction de la clef de codage fait intervenir les coefficients de Bézout de l'indicatrice d'Euler de l'entier n modulo lequel on travaille, n étant le produit de 2 grands nombres premiers).
- On peut prolonger l'exercice avec le théorème des restes chinois et l'algorithme correspondant.
- On peut aussi s'intéresser à la reconstruction d'un rationnel à partir de son image modulo un entier. L'algorithme de reconstruction est en effet un algorithme de Bézout avec arrêt prématuré. Dans le cas des polynômes, ce même algorithme avec arrêt prématuré permet de calculer des approximants de Padé de fonctions. Voir par exemple le manuel Algorithmes de calcul formel de Xcas. La combinaison des restes chinois et de la reconstruction rationnelle est une brique de base de nombreux algorithmes efficaces en calcul formel.

7.4 Calcul efficace du polynôme caractéristique

Numéros de leçons correspondants : 310, 319. Exposés : 110.

Énoncé : Soit A une matrice carrée d'ordre n à coefficients complexes. On se propose de déterminer le polynôme caractéristique de A en recherchant des relations linéaires entre un vecteur v_0, et ses images successives par A : $v_{k+1} = Av_k$

1. Montrer que $\{v_0, ..., v_n\}$ est une famille liée (on pourra donner une combinaison linéaire faisant intervenir les coefficients du polynôme minimal de A). Déterminer une matrice dont le noyau est le sous-espace vectoriel des coefficients réalisant une combinaison linéaire nulle entre ces vecteurs.
2. On suppose qu'il existe, à multiplication près, une unique relation linéaire entre ces vecteurs $\{v_0, ..., v_n\}$, en déduire le polynôme minimal et caractéristique de A. Programmer cet algorithme avec un choix au hasard de v_0.
3. On suppose que la matrice A a un polynôme minimal de degré n. Expliquer quelles stratégies permettraient d'améliorer le programme pour tenir compte de choix malchanceux de v_0.

Solution

1. Comme la famille $\{v_0, ..., v_n\}$ contient $n+1$ vecteurs dans $\mathbb{C}^n$ elle est forcément liée. On peut d'ailleurs donner explicitement une combinaison linéaire nulle, si $P(x) = \sum_{j=0}^{n} p_j x^j$ est le polynôme caractéristique de A, le théorème de Cayley-Hamilton donne $P(A) = 0$, d'où l'on déduit $P(A)v = 0$ soit

$$\sum_{j=0}^{n} p_j v_j = 0$$

Plus généralement, avoir une combinaison linéaire nulle de coefficients $\Lambda = (\lambda_n, ..., \lambda_0)$ revient à dire que Λ est dans le noyau de la matrice B dont les colonnes sont $v_n, ..., v_0$.

2. On a le même type de relation avec le polynôme minimal de A, donc s'il existe une seule combinaison linéaire à multiplication près, elle est multiple des coefficients du polynôme minimal et de tout polynôme annulateur de A non nul. Il s'en suit que le degré du polynôme minimal est n, qu'il est donc égal au polynôme caractéristique et que l'on peut déduire les 2 du calcul du noyau de B (en multipliant de sorte que le coefficient de v_n soit 1 ou $(-1)^n$ selon la définition adoptée pour le polynôme caractéristique).

L'algorithme va donc choisir un vecteur aléatoire v_0, calculer les v_k (attention pour l'efficacité, il faut utiliser la récurrence et surtout ne pas calculer les A^k), les mettre en ligne dans une matrice que l'on transpose pour obtenir B, puis appeler l'instruction `ker` de calcul du noyau.

```
fonction polmin(A)
  local k,n,vk,B,noyau;
  n:=size(A); // nombre de lignes
  vk:=ranm(n); // initialise v0 aleatoire (reel)
  B:=[vk]; // initialise B (tranposee)
  pour k de 1 jusque n faire
    vk:=A*vk;
    B[k]:=vk;
  fpour;
  B:=tran(revlist(B)); // vn ... v0
  noyau:=ker(B);
  si size(noyau)>1 alors
    return "echec de l'algorithme";
  fsi;
  return noyau[0];
ffonction:;
```

On peut le tester avec une matrice aléatoire, par exemple

```
A:=ranm(3,3);polmin(A);pmin(A);
```

$$\begin{pmatrix} 19 & -38 & -80 \\ -87 & -23 & -62 \\ 39 & -45 & -82 \end{pmatrix}, [\frac{1}{-39160}, \frac{43}{-19580}, \frac{-617}{-7832}, -1], poly1[1, 86, -3085, 39160]$$

Le même algorithme nécessite la maitrise de librairies d'algèbre linéaire en Python. Mais on peut bien entendu saisir le programme en syntaxe Python dans Xcas

```
def polmin(A):
    # local k,n,vk,B,noyau
    n=size(A)
    vk=ranm(n)
    B=[vk]
    for k in range(1,n+1):
        vk=A*vk
        B[k]=vk
    B=tran(revlist(B))
    noyau=ker(B)
    if size(noyau)>1 :
        return "echec de l'algorithme"
    return noyau[0]
```

3. L'algorithme peut échouer parce que v_0 se trouve dans un sous-espace strict de $\mathbb{C}^n$ formé par une somme de sous-espaces caractéristiques, le polynôme minimal relatif à v_0 (obtenu comme premier élément du noyau de B) est alors un diviseur strict du polynôme minimal. On peut y remédier en ajoutant une boucle extérieure qui effectue un nombre fini d'essais de choix de v_0 (par exemple 3). On peut même améliorer en prenant le PPCM des polynômes minimaux relatifs aux vecteurs v_0 testés.
Néanmoins l'algorithme échouera si le polynôme minimal n'est pas égal au polynôme caractéristique. L'algorithme de Danilevsky ne présente pas cet inconvénient (il est aussi en $O(n^3)$, voir le manuel algorithme de Xcas).

Commentaires

- Cet exercice peut être proposé à partir de la 2ème année universitaire.
- Cet algorithme nécessite $O(n^3)$ opérations sur le corps des coefficients. Il est donc plus efficace que l'interpolation de Lagrange ou que le calcul du déterminant de $A - xI$ (ce dernier nécessite $O(n^3)$ opérations mais sur l'anneau des polynômes à une variable sur ce corps), mais peut échouer (ce sera le cas avec une probabilité très faible pour une matrice à coefficients entiers, nulle pour des coefficients réels, mais pas si on se réfère aux bases d'exercices qui font la part belle aux matrices dont le polynôme minimal n'est pas de degré n).
- Sur le thème des algorithmes en algèbre linéaire, on peut être tenté de proposer l'algorithme du pivot de Gauss, mais il faut avoir bien conscience qu'il risque d'être difficile à mettre au point pendant le temps de préparation. On peut par contre implémenter par exemple l'algorithme de la puissance ou le calcul du polynôme caractéristique par interpolation de Lagrange.

7.5 Exemples de méthodes et d'algorithmes de résolution approchée d'équations $F(X) = 0$.

Numéros de leçons correspondants : 443. Exposés : 250, 201, 208

Énoncé :

Soit $e \in [0,1[$ et $t \in [-\pi, \pi]$ On se propose de résoudre l'équation

$$x = t + e\sin(x)$$

par dichotomie et par la méthode du point fixe et comparer ces deux méthodes. (N.B. e ne désigne pas la base des logarithmes ici, mais l'excentricité d'une ellipse dont on cherche à résoudre l'équation du temps).

1. Par dichotomie. Soit $f(x) = x - t - e\sin(x)$. Déterminer le signe de $f(-\pi - e)$ et de $f(\pi + e)$, ainsi que le sens de variations de f, en déduire qu'il existe une unique solution de l'équation. Proposer un algorithme permettant de trouver une valeur approchée de la solution à `1e-8` près. Combien d'étapes sont-elles nécessaires ?
2. Par le point fixe. Soit $g(x) = t + e\sin(x)$. Montrer que g est contractante sur $[-\pi - e, \pi + e]$. En déduire une suite récurrente convergeant vers l'unique solution de l'équation sur $[-\pi - e, \pi + e]$. Proposer un algorithme permettant de trouver une valeur approchée de la solution à `1e-8` près. Combien d'étapes sont-elles nécessaires ?
3. Comparer la vitesse de ces deux méthodes en fonction de e.

Solution

1. f est une fois continument dérivable sur $\mathbb{R}$ et sa dérivée est $f'(x) = 1 - e\cos(x) > 1 - e > 0$ donc f est strictement croissante. De plus $f(-\pi - e) \leq -\pi - e - t + e \leq 0$, $f(\pi + e) \geq \pi + e - t - e \geq 0$ donc f s'annule une fois et une seule sur l'intervalle $[-\pi - e, \pi + e]$.

```
def dicho(f,a,b,eps):
    # local c,niter;
    if f(a)*f(b)>=0:
        return "erreur: f(a)*f(b)>=0"
    while b-a>eps:
        c=(a+b)/2.0
        if f(a)*f(c)>0:
            a=c
        else:
            b=c
    return [a,b]
```

Puis on essaie par exemple pour $t = 1$ et $e = 0.1$.

```
f(x):=x-1-0.1*sin(x); dicho(f,-pi-0.1,pi+0.1,1e-8)
```

$$x \to x - 1 - 0.1\sin(x), [1.08859775129, 1.08859775733]$$

Le nombre d'étapes est le plus petit entier n tel que

$$\frac{2(\pi + e)}{2^n} \leq \varepsilon$$

soit :

$$n \geq \frac{\ln(2(\pi+e)/\varepsilon)}{\ln(2)}$$

2. On a clairement $g([-\pi-e,\pi+e])$ inclus dans $[-\pi-e,\pi+e]$. De plus la dérivée de g est de valeur absolue au plus e, indépendant de x et strictement inférieur à 1. Donc g est contractante de rapport e. Le théorème du point fixe assure que la suite récurrente définie par $u_0 \in [-\pi-e,\pi+e]$ et $u_{n+1} = g(u_n)$ converge vers l'unique solution de $g(x) = x$ dans $[-\pi-e,\pi+e]$. Si l désigne la solution de l'équation, on a de plus l'estimation à priori :

$$|u_n - l| \leq 2(\pi+e)e^n$$

Plus généralement, si k est la constante de contractance, on a l'estimation à postériori :

$$|u_n - l| \leq \frac{|u_{n+1}-u_n|}{1-k}$$

qui permet de fournir un test dans l'algorithme pourvu que k soit passé en paramètre

```
def fixe(f,u0,k,eps):
    # local u1
    u0=u0*1.0
    u1=f(u0)
    eps=eps*(1-k)
    while (abs(u1-u0))>eps :
        u0=u1
        u1=f(u0)
    return(u0)
```

On teste ensuite avec par exemple $u_0 = 0.0$:

```
g(x):=1+0.1*sin(x); fixe(g,0.0,0.1,1e-8)
```

$$x \to 1 + 0.1\sin(x), 1.08859775144$$

Le nombre d'étapes se majore avec l'estimation à priori

$$n \geq \frac{\ln(2(\pi+e)/\varepsilon)}{-\ln(e)}$$

En réalité, le nombre d'étapes peut être (très) inférieur, car la majoration de la constante de contractance peut être beaucoup trop large lorsqu'on se rapproche du point fixe (par exemple si la dérivée de g est nulle au point fixe, ce qui serait le cas si on appliquait une méthode de Newton), d'où l'intérêt de l'estimation à postériori utilisée dans l'algorithme.

3. Le nombre d'étapes aboutit à la même formule que pour la dichotomie, à l'exception de $\ln(2)$ remplacé par $-\ln(e)$. On a donc intérêt à utiliser la dichotomie si la constante $e > 0.5$ et le point fixe sinon (avec les mêmes réserves que précédemment si la majoration de $|g'|$ par e au point fixe est beaucoup trop large).

Commentaires

– La première partie de l'exercice peut être traitée au lycée (dès la seconde en adaptant la fonction, par exemple avec un polynôme de degré 2 sans paramètre). La deuxième partie est plutôt du niveau du supérieur. On peut évidemment faire deux énoncés distincts avec cet exercice.
– Ces deux méthodes ont une vitesse de convergence linéaire. On pense bien sur à la méthode de Newton pour avoir une vitesse de convergence quadratique, mais il est souvent plus difficile de justifier la convergence en pratique (sauf à une variable réelle sous des hypothèses de convexité, par exemple ici f est convexe ou concave par intervalles).
– La méthode du point fixe (et la méthode de Newton) se généralisent en dimension plus grande que 1, ce qui n'est pas le cas en général de la dichotomie. Dans certains cas particuliers, on peut généraliser la dichotomie, par exemple dans $\mathbb{C}$, en cherchant une solution dans un rectangle que l'on coupe en 4.
– Pour les équations polynomiales, il existe une variété d'algorithmes de résolution : des exactes pour les degrés 2 à 4 (Cardan, Ferrari) aux approchées (Newton à une variable complexe, Bairstow qui est une méthode de Newton à 2 variables réelles, les suites de Sturm réelles pour isoler les racines réelles en basculant sur la dichotomie lorsqu'il ne reste qu'une seule racine dans un intervalle, ou la diagonalisation numérique de la matrice companion du polynôme).

8 Références

Dans la documentation de Xcas (donc disponible le jour de l'oral 2), aller dans le menu Aide, sous-menu Manuels, puis Programmation. Vous y trouverez une grande partie des algorithmes ci-dessus. Le manuel Exercices et le manuel Amusement peuvent aussi contenir des exercices à vocation algorithmique ou illustrable par Xcas. Enfin, le manuel tableur, statistiques peut servir pour une illustration d'exercices de probastats. Vous pouvez aussi lancer une recherche avec un mot clef (menu Aide, Rechercher un mot, par exemple Bezout).

En livres :

– Guide du calcul avec les logiciels libres XCAS, Scilab, Bc, Gp, GnuPlot, Maxima, MuPAD..., G. Connan et S. Grognet
– Analyse numérique et équations différentielles, Demailly J.-P., Presses Universitaires de Grenoble, 1996 (pour l'analyse)
– The Art of Computer Programming, Vol. 2 : Seminumerical algorithms, Knuth D., Addison-Wesley, 1998 (pour certains aspects en arithmétique)
– A course in computational number theory, Henri Cohen
– Mathématiques concrètes, illustrées par la TI-92 et la TI-89. Lemberg H, et Ferrard J.-M., Springer, 1998
– Maths et Maple, J.M. Ferrard, Dunod, 1998

Sur le web :

– `http://www-fourier.ujf-grenoble.fr/~parisse/giac_fr.html`, la page de Xcas, avec en particulier la page algorithmique `http://www-fourier.ujf-grenoble.fr/~parisse/algoxcas.html`

- http://tehessin.tuxfamily.org/, site de Guillaume Connan, contient un livre téléchargeable librement couvrant pratiquement tout le programme
- http://www.math.jussieu.fr/~han/M1E/, F. Han, algorithmique M1E.
- http://www-fourier.ujf-grenoble.fr/~parisse/#mat249, mon cours Math Assistés par ordinateur de licence 2ième année

9 Aide-mémoire

Instructions Xcas en français	
puissance	`^ ** powmod`
quotient, reste euclidien	`iquo irem`
affectation	`a:=2;`
entrée expression	`saisir("a=",a);`
entrée chaine	`saisir_chaine("a=",a);`
sortie	`afficher("a=",a);`
valeur retournée	`retourne(a);`
arrêt dans boucle	`break;`
déclaration fonction	`f(parametres):={ ... }`
alternative	`si <condition> alors <inst> fsi;` `si <condition> alors <inst1> sinon <inst2> fsi;`
boucle pour	`pour j de a jusque b faire <inst> fpour;` `pour j de a jusque b pas p faire <inst> fpour;`
boucle répéter	`repeter <inst> jusqua <condition>;`
boucle tantque	`tantque <condition> faire <inst> ftantque;`

Les indices de tableau commencent à 0 avec la notation `[]` ou à 1 avec la notation `[[]]`, par exemple `l:=[1,2]; l[0]; l[[1]]`

Instructions Python (utilisables dans Xcas)	
puissance	`** pow(a,n,m)`
quotient, reste euclidien	`// %`
affectation	`a=2`
entrée expression	`a=input("a=")`
sortie	`print "a=",a`
déclaration fonction	`def f(parametres):` `  instructions`
valeur retournée	`return a;`
arrêt dans boucle	`break;`

alternative	`if <condition>:`
	`  instructions1`
	`else:`
	`  instructions2`

boucle pour	`for j in range(debut,fin,pas):` `  instructions`
boucle tant que	`while condition:` `  instructions`

Les indices de tableau commencent à 0, par exemple `l=[1,2]; l[0];`. Attention, le test d'égalité se fait avec == et non = (affectation).

Instructions Xcas "C-like"	
puissance	`^ ** powmod`
quotient, reste euclidien	`iquo irem`
affectation	`a:=2;`
entrée expression	`input("a=",a);`
entrée chaine	`textinput("a=",a);`
sortie	`print("a=",a);`
valeur retournée	`return(a);`
arrêt dans boucle	`break;`
déclaration fonction	`f(parametres):={ ... }`
alternative	`if (<condition>) {<inst>};`
	`if (<condition>) {<inst1>} else {<inst2>};`
boucle pour	`for (j:= a;j<=b;j++) {<inst>}`
	`for (j:= a;j<=b;j:=j+p) {<inst>}`
boucle répéter	`repeat <inst> until <condition>;`
boucle tantque	`while (<condition>) {<inst>};`

Les indices de tableau commencent à 0 avec la notation `[]` ou à 1 avec la notation `[[]]`, par exemple `l:=[1,2]; l[0]; l[[1]]`

Instructions Maxima	
puissance	`^ ** power_mod`
division euclidienne, reste	`divide mod`
affectation	`a:2;`
sortie	`print("a=",a)`
valeur retournée	`return(a)`
déclaration fonction	`f(parametres):=( ... )`
alternative	`if (<condition>) then <inst>`
	`if (<condition>) then <inst1> else <inst2>`
boucle pour	`for j:1 thru 26 do (inst1,inst2)`
	`for j:2 thru 20 step 2 do (inst,...,inst)`
boucle tantque	`for i:1 while <condition> do (inst,...,inst)`

Les indices de tableau commencent à 1, par exemple `l:[1,2]; l[1];`

Instructions Xcas en mode Maple	
puissance	`^ ** a&^n mod m`
quotient, reste euclidien	`iquo irem`
affectation	`a:=2;`
sortie	`print("a=",a);`
valeur retournée	`RETURN(a);`
arrêt dans boucle	`break;`
déclaration fonction	`f:=proc(parametre) ... end;`
alternative	`if <condition> then <inst> fi;` `if <condition> then <inst1> else <inst2> fi;`
boucle pour	`for j from a to b do <inst> od;` `for j from a to b step p do <inst> od;`
boucle tantque	`while <condition> do <inst> od;`

Les indices de tableau commencent à 1 (Xcas syntaxe mode Maple), par exemple `l:=[1,2]; l[1];`

Instructions Scilab	
affectation	`a=2`
entrée expression	`a=input("a=")`
sortie	`disp("a=",a)`
déclaration fonction	`function nom_var=f(parametres) ... endfunction`
valeur retournée	`nom_var=...;`
arrêt dans boucle	`break;`
alternative	`if <condition> then <inst> end;` `if <condition> then <inst1> else <inst2> end;`
boucle pour	`for j=debut:fin do <inst> end;`
boucle tantque	`while <condition> do <inst> end;`

Les indices de tableau commencent à 1, par exemple `l=[1,2]; l(1);`.