Algorithmique en seconde avec Xcas

Jean-Pierre Branchard Renée De Graeve Bernard Parisse

Octobre 2009

Table des matières

1 Introduction **2**
1.1 Installation de Xcas . . . 2
1.2 Algorithme et programme . . . 2

2 Premiers algorithmes **2**
2.1 Première construction . . . 2
2.2 Premier script . . . 3
2.3 Première fonction . . . 4
2.4 Utilisation d'Xcas en ligne . . . 6
2.5 Exercices . . . 6

3 Variables, affectation, fonctions. **6**

4 L'alternative (si) **7**

5 Les boucles (pour, tantque, repeter) **10**

6 Exécution en mode pas à pas **12**

7 Synthèse : L'automate à billet **14**
7.1 L'énoncé . . . 14
7.2 La correction . . . 14

8 Suggestion : algorithmique et code génétique **16**

9 Glossaire **18**
9.1 Pour écrire une fonction ou un programme . . . 18
9.2 Le menu Add d'un niveau éditeur de programme . . . 18
9.3 Les instructions en français . . . 19
9.4 Les instructions comme en C . . . 19
9.5 Les instructions en mode Maple . . . 20
9.6 Signification des signes de ponctuation . . . 20
9.7 Les opérateurs . . . 20
9.8 Séquences, listes et chaines de caractères . . . 21
9.9 Les fonctions utilisées . . . 21

1 Introduction

1.1 Installation de Xcas

Le programme Xcas est un logiciel libre multiplateformes que l'on récupère sur :
http : //www − fourier.ujf − grenoble.fr/~parisse/giac_fr.html
Une fois installé, pour lancer Xcas :

- Windows : cliquez sur l'icone xcasfr.bat
- Linux : dans le menu des applications, chercher Xcas dans la catégorie Education, s'il n'y est pas, ouvrez un Terminal dans Accessoires et tapez la commande xcas &
- Mac : cliquez sur Xcas dans le menu Applications du Finder.

1.2 Algorithme et programme

Ce document commence par introduire la notion d'algorithme en partant d'une construction géométrique qu'on transforme progressivement en une fonction. Puis on présente les structures classiques de controle (tests, boucles), et un exemple de synthèse (l'automate à billets). La dernière section est une ouverture vers l'algorithmique pour la génétique. On trouvera en appendice une synthèse des diverses commandes et mots clefs sous forme de tableaux pouvant servir de carte de référence.

Nous avons pris le parti d'illustrer les principes de base de l'algorithmique avec plusieurs exemples issus de la géométrie, pour montrer l'intérêt d'utiliser Xcas par rapport à un langage généraliste dans le cadre d'un enseignement d'algorithmique intégré dans un enseignement de mathématiques. Le lecteur pourra se reporter aux nombreux autres documents de la documentation en ligne de Xcas pour des exemples plus classiques (Euclide, primalité, cryptographie, ...).

La traduction d'un algorithme avec Xcas peut se faire de plusieurs manières, soit avec des mots clef en français et une syntaxe très proche du langage algorithmique, soit avec des mots clef en anglais et au choix une syntaxe similaire à Maple ou au langage C++. Dans les exemples qui suivent, on utilisera la version française et on se reportera au glossaire à la fin de l'article (section 9) pour les autres syntaxes. Lorsque la description d'un algorithme et sa traduction en Xcas sont très proches, ce qui est souvent le cas, nous donnons directement le programme pour éviter les redondances.

2 Premiers algorithmes

2.1 Première construction

Nos élèves ne feraient-ils pas déjà de l'algorithmique sans le savoir ? Le mode opératoire pour construire le centre du cercle circonscit à un triangle, par exemple, n'est-il pas un algorithme ?

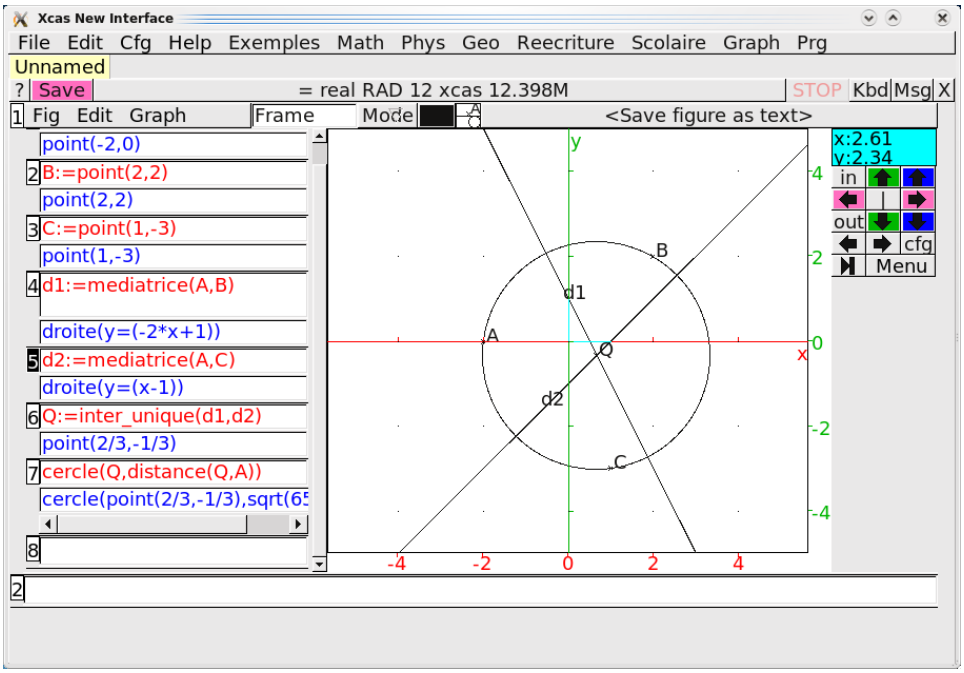
Voyons ce que cela donne dans Xcas : on commence par ouvrir une fenêtre graphique avec la combinaison de touches Alt-g. Puis on tape successivement dans la zone de saisie :

1. `A:=point(-2,0)`

2. `B:=point(2,2)`
3. `C:=point(1,-3)`
4. `d1:=mediatrice(A,B)`
5. `d2:=mediatrice(A,C)`
6. `Q:=inter_unique(d1,d2)`
7. `cercle(Q,distance(Q,A))`

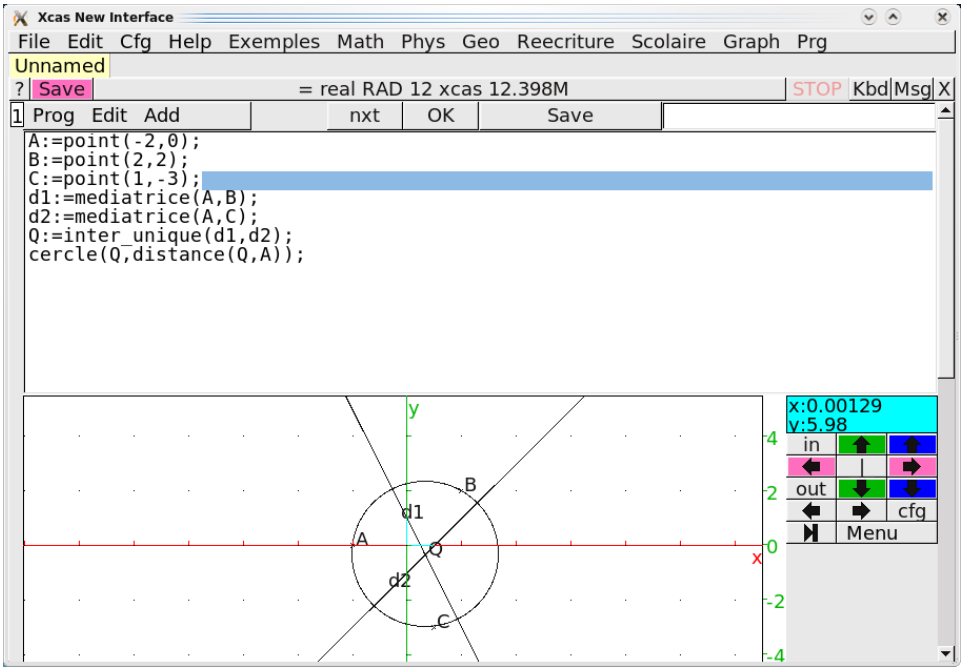
Ces premières commandes permettent déjà d'apréhender les notions de **variable** et d'**affectation**. Par exemple, la commande B :=point(2,2) nous permet de nommer B le point de coordonnées (2 ;2). Autrement dit, on crée une nouvelle variable nommée B et on lui affecte le point (2 ;2).

Dans la fenêtre graphique ; la construction est faite :



2.2 Premier script

Pour éviter d'avoir à taper toutes ces commandes chaque fois qu'on veut construire un cercle circonscrit, on peut imaginer d'enregistrer celles-ci dans un **script**. Pour cela, on ouvre une fenêtre de programmation dans `Xcas` avec `Alt-p` et on y tape les commandes, terminées par un point-virgule et on enlève le `:;` déjà écrit à la fin. Il suffit ensuite de cliquer sur le bouton **OK(F9)** ou d'appuyer sur la touche **F9** pour que le script s'exécute :



2.3 Première fonction

Bien-sûr, ce script a un inconvénient majeur : il sait construire seulement le cercle circonscrit du triangle ABC que nous avions défini. Nous allons donc le faire évoluer afin de le transformer en une **fonction** capable de construire le cercle circonscrit à n'importe quel triangle.

Une fonction est un algorithme qui réalise une action avec les éléments qu'on lui donne (les paramètres) puis retourne un certain résultat. Celle-ci construira le cercle éventuel passant par les trois points qu'on lui donnera. Pour cela, on ouvre une fenêtre de programmation dans `Xcas` avec `Alt-p` et on tape :

```
Cercle_circ(A,B,C):={
 local d1,d2,Q;
 d1:=mediatrice(A,B);
 d2:=mediatrice(A,C);
 Q:=inter_unique(d1,d2);
 retourne Q,cercle(Q,distance(Q,A));
}
```

Il suffit ensuite de cliquer sur le bouton **OK(F9)** (ou touche **F9**) pour compiler cette fonction. `Xcas` renvoie les erreurs éventuelles de syntaxe ou lorsque c'est correct :

```
// Parsing Cercle_circ
// Success compiling Cercle_circ
```

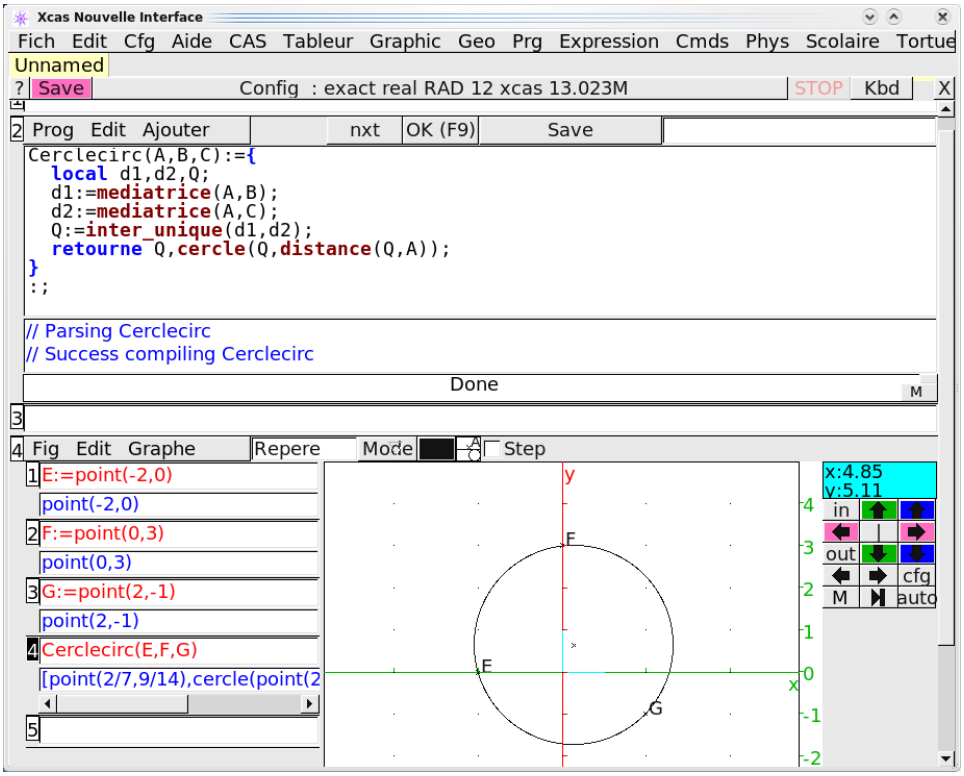
mais `Xcas` n'exécute rien car il attend que cette fonction soit utilsée, par exemple, dans un niveau d'entrée avec des valeurs pour les paramètres A, B et C.

Quelques remarques :

– Nous avons nommé la fonction `Cercle_circ`. les noms de fonctions définies dans ce document commenceront toutes par une lettre majuscule, pour éviter de les confondre avec les fonctions prédéfinies de `Xcas`.

– Les points A, B et C sont les **paramètres** en fonction desquels sera construit le cercle (si les 3 points sont alignés, la fonction ne retourne rien).
– Les variables d1, d2 et Q sont **locales**, ce qui veut dire qu'elles sont employées seulement à l'intérieur de la fonction `Cercle_circ` et qu'elles cessent d'exister dés qu'`Xcas` a fini d'exécuter la fonction.
– La dernière instruction dit que la fonction doit retourner à l'utilisateur le centre Q et le cercle.

Pour utiliser cette fonction, tapons successivement des commandes pour créer trois nouveaux points, nommés par exemple `E, F` et `G`, puis tapons la commande `Cercle_circ(E,F,G)`. `Xcas` exécutera la fonction en faisant jouer à `E, F` et `G` les rôles respectifs de `A, B` et `C`. On remarque que le centre `Q` est dessiné avec une croix mais n'a pas de nom sur la figure (il faudrait utiliser `legende(Q,"Q")` à la place de `Q` pour voir le nom[1]). On obtient l'illustration suivante :

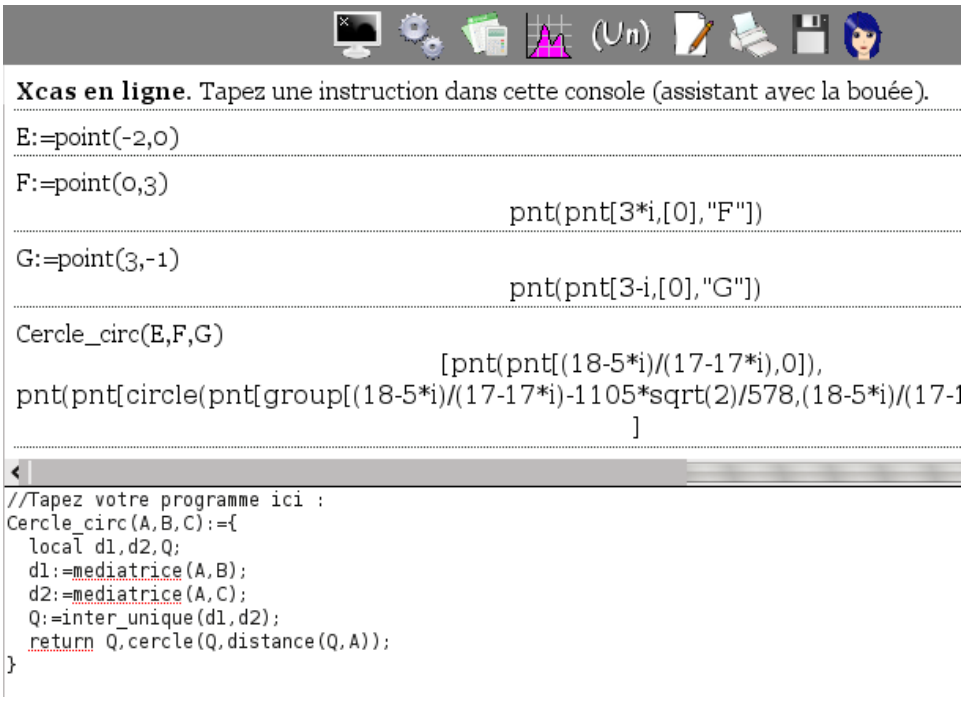


Nous croyons que la conception et l'écriture d'algorithmes sous forme de fonctions peut éclairer cette dernière notion dans le cours de mathématiques. L'expérience montre en effet que les lycéens ne donnent pas de sens au mot «fonction», souvent confondu avec «formule» ou bien «courbe». Par contre, une fois qu'on a programmé des fonctions qui réalisent des figures géométriques à l'aide de points passés en paramètre, on peut comprendre qu'une fonction pourra aussi calculer un nombre réel à partir d'un autre nombre réel.

[1] On peut aussi lui donner un nom (par exemple `H`) en tapant `H:=Cercle_circ(E,F,G)[0]`

2.4 Utilisation d'Xcas en ligne

Le site http ://xcasenligne.fr permet d'utiliser partiellement Xcas dans son navigateur si on ne l'a pas sur son ordinateur. Le travail précédemment exposé peut être mené sous Xcas en ligne. Il suffit de cliquer sur l'engrenage pour passer en mode programmation. On tape alors le programme dans le cadre du bas et les instructions dans le cadre du haut, comme le montre cette capture d'écran :



Toutefois, le débogueur (voir plus loin) n'est pas disponible sous Xcas en ligne.

2.5 Exercices

1. Écrire une fonction pour construire le cercle inscrit d'un triangle et une autre pour construire le centre de gravité.
2. La consommation d'un véhicule est proportionnelle au carré de sa vitesse. Sachant qu'un certain véhicule consomme 5 litres aux 100 km s'il roule à 90 km/h, écrire une fonction qui calcule sa consommation à partir de sa vitesse.

3 Variables, affectation, fonctions.

Le langage Xcas n'est pas typé mais le contenu d'une variable peut être de différents types :

- entier `j:=3`, fraction `j:=3/2`,
- réel `j:=1.5`,
- expression `j:=sin(x)+1`,
- liste `l:=[1,2,3]`, séquence `l:=1,2,3`
- chaine de caractère `s:="bonjour"`
- objet graphique `A :=point(1,2);`, `C :=cercle(0,1);`,

– ...

et les variables peuvent être définies globalement ou seulement à l'intérieur d'une fonction (variables locales).

Une fonction regroupe plusieurs instructions pour effectuer un traitement des données et renvoyer un résultat appelé valeur de retour. Les données sont soit les paramètres passés en argument à la fonction soit des données saisies en cours d'exécution (par la commande `saisir`). En cours d'exécution, on peut afficher des résultats intermédiaires (commande `afficher`) mais il ne faut pas confondre ces affichages avec le renvoi de la valeur de retour.

La syntaxe d'une fonction est la suivante :

```
f(x,y):={
  local z,a,b; // declaration des variables locales
  instruction1;
  .....
  instructionk;
}
```

Lors de l'exécution d'une fonction, si il n'y a pas d'instruction `retourne`, `Xcas` renvoie la valeur de l'évaluation de la dernière instruction. Lorsque `Xcas` exécute une instruction `retourne`, `Xcas` renvoie la valeur qui suit `retourne` ce qui termine immédiatement l'exécution de la fonction.

En pratique, il est conseillé d'écrire les définitions des fonctions dans un niveau éditeur de programmes (menu `Prg->Nouveau programme`). Dans un éditeur de programmes, les mots clef apparaissent en bleu et les commandes de `Xcas` en brun. Pour compiler une fonction, on clique sur le bouton `OK(F9)` (ou la touche `F9`). S'il y a des erreurs, la ligne où est l'erreur est surlignée (mais l'erreur peut provenir de la ligne juste avant). S'il n'y a pas d'erreurs, la valeur `Done` apparait (si on a fait suivre la fin de la fonction par `:;`, sinon on voit le listing de la fonction compilée). Si vous tapez deux fonctions dans la même fenêtre de programmation, il est conseillé de les terminer par des `:;`.

4 L'alternative (si)

Les instructions `si...fsi` et `si...sinon...fsi` permettent d'exécuter une ou plusieurs instructions selon une condition (`fsi` marque la fin du `si`).

– Pour le `si`, la syntaxe admise est :
  `si` *condition* `alors` *instructions*`; fsi;`
  On teste la condition : si elle est vraie, on exécute les instructions et si elle est fausse on passe aux instructions qui suivent `fsi`.
– Pour le `si...sinon`, la syntaxe admise est :
  `si` *condition* `alors` *instructions1*`; sinon` *instructions2*`;`
  `fsi;`
  On teste la condition : si elle est vraie, on exécute les instructions1 et si elle est fausse on exécute les instructions2.

Exemples : Intersection de 2 cercles

1. On veut écrire une fonction qui permet de dessiner deux cercles et leur intersection quand elle existe !

Soient deux cercles c_1 et c_2 de centres respectifs O_1 et O_2 et de rayons respectifs r_1 et r_2. On montre que si $|r_1 - r_2| \leq O_1O_2 \leq r_1 + r_2$ l'intersection $c_1 \cap c_2$ est constituée d'1 ou de 2 points et sinon cette intersection est vide.
On utilisera les fonctions de `Xcas` :
- `distance(A,B)` qui renvoie la longueur du segment AB
- `rayon(c1)` qui renvoie la valeur du rayon de c_1
- `centre(c1)` qui renvoie le point qui est le centre de c_1
- `inter(c1,c2)` qui renvoie selon les cas une liste de 0, 1 ou 2 points.

On utilisera les opérateurs de `Xcas` :
- `==` opérateur booléen entre deux expressions qui renvoie `vrai` si ces deux expressions ont égales et `faux` sinon,
- `et` opérateur booléen entre deux conditions qui renvoie `vrai` si les 2 conditions sont toutes les 2 vraies et `faux` sinon,
- `ou` opérateur booléen entre deux conditions qui renvoie `faux` si les 2 conditions sont toutes les 2 fausses et `vrai` sinon,

On tape :

```
Inter2(c1,c2):={
 local O1,O2,r1,r2,d;
 r1:=rayon(c1);
 r2:=rayon(c2);
 O1:=centre(c1);
 O2:=centre(c2);
 d:=distance(O1,O2);
 si d>=abs(r1-r2) et d<=r1+r2  alors
   retourne c1,c2,inter(c1,c2,affichage=epaisseur_point_5);
 sinon
   afficher("intersection vide");
   retourne c1,c2;
 fsi;
}:;
```

On remarquera que :
- l'opérateur booléen `et` traduit la double inégalité,
- l'instruction `afficher` permet de faire des affichages en bleu, dans une zone intermédiaire.
- le 3ième paramètre `affichage=epaisseur_point_5` de l'instruction `inter` permet de dessiner le ou les points avec une croix d'epaisseur 5.
- puisque l'instruction `retourne` fait sortir du test, on peut remplacer :

```
 si d<abs(r1-r2) et d<r1+r2 alors
   retourne c1,c2,inter(c1,c2,affichage=epaisseur_point_5);
 sinon
   afficher("intersection vide");
   retourne c1,c2;
 fsi;
```

par un code plus lisible :

```
 si d<abs(r1-r2) et d<r1+r2 alors
   retourne c1,c2,inter(c1,c2,affichage=epaisseur_point_5);
 fsi;
```

```
  afficher("intersection vide");
  retourne c1,c2;
```

2. On veut écrire une fonction qui donne le nombre de points d'intersection de deux cercles.
   Ce deuxième exemple comporte 2 tests imbriqués.
   On écrit :

```
Ninter2(c1,c2):={
 local O1,O2,r1,r2,d;
 si c1==c2 et r1==r2 alors retourne inf; fsi;
 r1:=rayon(c1);
 r2:=rayon(c2);
 O1:=centre(c1);
 O2:=centre(c2);
 d:=distance(O1,O2);
 si abs(r1-r2)<d et d<r1+r2 alors
   retourne 2;
 sinon
   si d==r1+r2 ou d==abs(r1-r2) alors
     retourne 1;
   sinon
     retourne 0;
   fsi;
 fsi;
}:;
```

   On remarquera que :
   – l'opérateur booléen `ou` traduit l'une ou l'autre des égalités,
   – le test d'égalité est ==,
   – puisque l'instruction `retourne` fait sortir du test, on peut remplacer :

```
si abs(r1-r2)<d et d<r1+r2 alors
   retourne 2;
sinon
   si d==r1+r2 ou d==abs(r1-r2) alors
        retourne 1;
   sinon
        retourne 0;
   fsi;
fsi;
```

   par un code plus lisible :

```
si abs(r1-r2)<d et d<r1+r2 alors retourne 2;fsi;
si d==r1+r2 ou d==abs(r1-r2) alors retourne 1;fsi;
retourne 0;
```

Exercices

1. Écrire une fonction qui nous dit si un point est dans le disque, sur sa frontière ou à l'extérieur du disque.
2. Écrire une fonction qui nous dit si 3 points forment un triangle équilatéral, isocèle, rectangle en utilisant

- uniquement la fonction `distance` qui renvoie la distance entre 2 objets géométriques,
- la fonction `distance` et la fonction `angle`. `angle(A,B,C)` renvoie la valeur (en radians ou en degrés) de l'angle `A` du triangle ABC (pour être en degré, il faut cliquer sur la barre `Config` et décocher `radian`)

3. écrire un nombre de secondes en heure, minutes, secondes en utilisant les fonctions
   - `iquo` qui renvoie le quotient euclidien de deux entiers, par exemple `iquo(19,3)` renvoie 6
   - `irem` qui renvoie le reste euclidien de deux entiers, par exemple `irem(19,3)` renvoie 1

5 Les boucles (pour, tantque, repeter)

Le langage `Xcas` propose plusieurs sortes de boucles :

1. La boucle `pour`
   La boucle `pour` permet de faire des instructions un nombre connu de fois. La boucle `pour` utilise une variable pour compter le nombre d'itérations que l'on fait. **Attention** le nom de cette variable sera `j` ou `k`...mais pas `i` qui désigne un nombre complexe ! ! !
   Les syntaxes admises sont :
   `pour j de n1 jusque n2 faire` *`instructions`*`; fpour;`
   `pour j de n1 jusque n2 pas p faire` *`instructions`*`; fpour;`
   On initialise `j` à `n1` puis on teste la condition `j<=n2` :
   - si elle est vraie, on fait les instructions puis on incrémente `j` (soit de 1, soit de `p`), puis, on teste condition `j<=n2` : si elle est vraie, on fait les instructions puis on incrémente `j` etc...
   - si elle est fausse on passe aux instructions qui suivent `fpour`.
2. La boucle `repeter`
   La boucle `repeat` ou `repeter` permet de faire plusieurs fois des instructions avec une condition d'arrêt à la fin de la boucle.
   La syntaxe de cette boucle est :
   `repeter` *`instructions`*`; jusqua` *`condition`*`;`
   On fait les instructions, puis on teste la condition :
   - si elle est vraie, on fait à nouveau les instructions puis on teste la condition etc...
   - si elle est fausse, on passe aux instructions qui suivent l'instruction `repeter`.
3. La boucle `tantque`
   La boucle `tantque` permet de faire plusieurs fois des instructions avec une condition d'arrêt au début de la boucle.
   La syntaxe de cette boucle est :
   `tantque` *`condition`* `faire` *`instructions`*`; ftantque;`
   On teste la condition :
   - si elle est vraie, on fait les instructions puis, on teste la condition : si elle est vraie, on fait les instructions etc...
   - si elle est fausse on passe aux instructions qui suivent `ftantque`.

4. L'instruction `break`
   L'instruction `break` permet de sortir immédiatement d'une boucle, elle s'utilise à l'intérieur du corps de la boucle après un test.
   On peut par exemple utiliser l'instruction `break` dans une boucle `pour` lorsque le nombre d'itérations dépend d'une condition et est au plus un nombre connu. La syntaxe est la suivante :
   `si` *`condition`* `alors break; fsi;`

Exemples

– La suite des points de la parabole $y = x^2$ dont les abscisses sont les entiers compris entre -5 et 5.
```
L:=NULL;
pour j de -5 jusque 5 faire
  L:=L,point(j,j^2);
fpour;
L;
```
On pourra se reporter à la section 9.8 pour les instructions de manipulation des séquences (`L` ici). La dernière ligne permet d'afficher la séquence des points de `L`. En effet, l'instruction `pour j de -5 jusque 5 faire point(j,j^2);fpour;` ne dessine que le dernier point car `Xcas` renvoie la valeur de l'évaluation de la dernière instruction.

– les points de la parabole $y = x^2$ d'abscisse entière et positive situés en-dessous de la droite $y = 30$
```
L:=point(0,0);
j:=1;
tantque j*j<=30 faire
  L:=L,point(j,j^2);
  j:=j+1;
ftantque;
L;
```

– Saisir un nombre compris entre 1 et 10 et afficher ce nombre multiplié par 10
```
repeter
  saisir("Nombre entre 1 et 10?",a);
jusqua a>=1 et a<=10;
afficher(a*10);
```

– Déterminer si un nombre est premier en testant la divisibilité (`irem(a,b)` renvoie le reste de la division euclidienne de `a` par `b`) :
```
Est_premier(n):={
  local j;
  pour j de 2 jusque n-1 faire
    si irem(n,j)==0 alors break; fsi;
  fpour;
  si j==n alors retourne 1; sinon retourne 0; fsi;
}
```
On peut aussi éviter le `break` en quittant immédiatement la fonction par `retourne 0` ce qui simplifie l'écriture mais oblige à traiter à part le cas $n = 1$:
```
Est_premier(n):={
```

```
  local j;
  si n==1 alors retourne 0 fsi;
  pour j de 2 jusque n-1 faire
    si irem(n,j)==0 alors retourne 0; fsi;
  fpour;
  retourne 1;
}
```

On peut bien sûr améliorer, par exemple en testant jusqu'à $\sqrt{n}$ (inclus) ou en testant séparément pour $j = 2$ puis pour j de 3 jusqu'à $\sqrt{n}$ avec un pas de 2.

On peut aussi utiliser la fonction `Sommediviseurs(n)` définie page 12 et qui renvoie la somme des diviseurs de n (1 compris mais n non compris)

`Est_prem(n) :=Sommediviseurs(n)==1;`

Exercices

Écrire une fonction ou un programme

1. renvoyant un polygone régulier à n cotés (on pourra utiliser l'instruction `rotation` de `Xcas`, attention aux unités d'angles),
2. qui détermine le nombre de pliages d'une feuille papier d'épaisseur 0.1mm jusqu'à atteindre la hauteur de la tour Eiffel,
3. qui effectue le calcul de la somme $1+\frac{1}{2}+\frac{1}{3}+..+\frac{1}{n}$ (c'est la somme de la série harmonique jusqu'au rang n)
4. qui effectue la saisie d'un mot de passe (3 essais maximum).

6 Exécution en mode pas à pas

Le **débogueur** d'`Xcas` exécute un programme pas à pas. Il permettra donc à l'élève d'observer le déroulement de son programme. Voyons son fonctionnement sur l'exemple d'un algorithme de recherche des «nombres parfaits» (entier naturel égal à la somme de ses diviseurs sauf lui même). Commençons par une fonction qui étant donné un entier naturel n, calcule la somme de ses diviseurs (1 compris mais n non compris). On applique l'algorithme suivant :

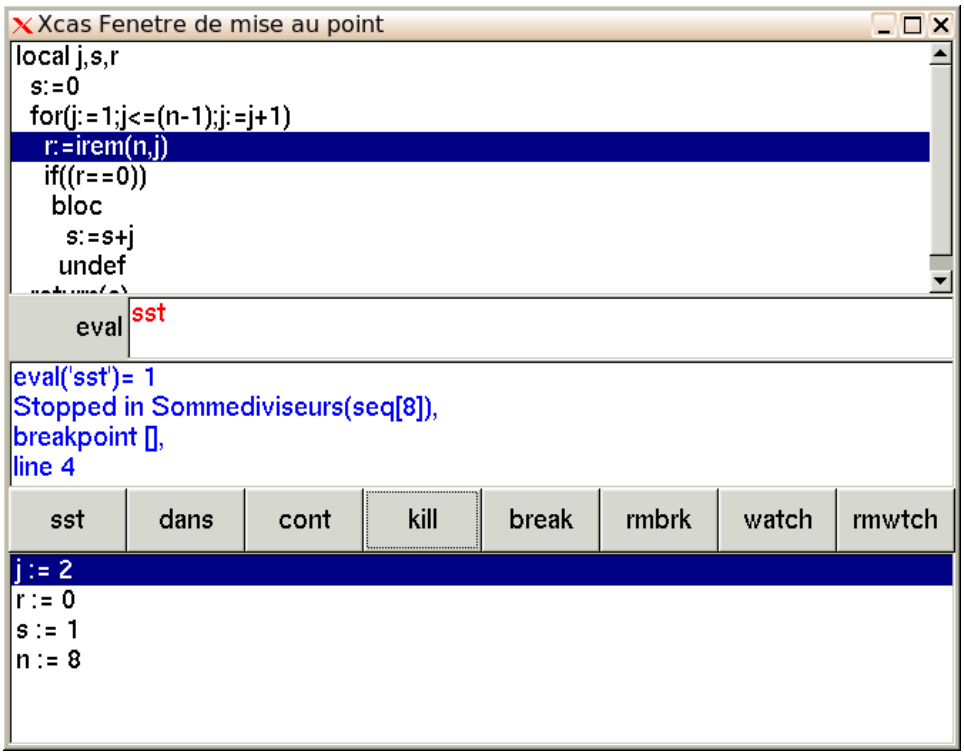
1. Créer une variable s destinée à recevoir la somme des diviseurs, lui donner la valeur 0.
2. Pour tous les entiers j compris entre 1 et $n-1$, faire les actions (a) et (b)
   (a) calculer le reste r de la division de n par j
   (b) si r est nul, ajouter j à s.
3. Retourner la valeur finale de s

Voici la traduction en `Xcas` :

```
Sommediviseurs(n):={
 local j,s,r;
 s:=0;
 pour j de 1 jusque n-1 faire
  r:=irem(n,j);
```

```
  si r==0
  alors
   s:=s+j;
  fsi;
 fpour;
 retourne s;
}
```

Il nous suffira de taper la commande `Sommediviseurs(8)` pour obtenir la somme des diviseurs de 8. Mais nous pouvons faire plus : en tapant la commande `debug(Sommediviseurs(8))`, nous ouvrons la fenêtre du débogueur dont la partie haute reprend le programme tandis que la partie basse donne les valeurs courantes des variables :



On clique sur le bouton **sst** pour exécuter le programme pas à pas. Dans la capture d'écran ci-dessus, vous voyez que le programme en était à l'instruction r :=irem(n,j). La partie basse montre que j était alors égal à 2, ce qui veut dire qu'il s'agissait du deuxième tour de la boucle.

On peut ainsi suivre l'exécution du programme et notamment voir se réitérer les instructions de la boucle. Quand on en a assez, on ferme le débogueur en cliquant sur **cont** (continue en mode normal) **kill** (arrêt du programme) puis en validant. L'intérêt pédagogique d'un tel dispositif saute aux yeux !

Il ne reste plus qu'à conclure notre recherche avec une fonction qui teste si un nombre est parfait ou non. Ouvrir un nouvelle fenêtre de programmation avec `Alt-p` puis taper :

```
Parfait(n):={
```

```
 si Sommediviseurs(n)==n
 alors
  retourne vrai;
 sinon
  retourne faux;
 fsi
};;
```

ou encore puisque `Sommediviseurs(n)==n` est soit vrai soit faux :

```
Parfait(n):=Sommediviseurs(n)==n;
```

Constatez au passage qu'une fonction peut appeler une autre fonction.
Exercices

1. Améliorer la fonction Sommediviseurs (est-il vraiment nécessaire de tester tous les entiers de 1 à $n-1$?)
2. Afficher les nombres parfaits inférieurs à 500 et les compter.

7 Synthèse : L'automate à billet

7.1 L'énoncé

Un distributeur de billets doit donner la somme S avec des billets de 10, 20 ou 50 euros et avec le moins de billets possibles. La somme S doit être un multiple de 10 et $S \leq 500$ euros.

1. Écrire un programme qui demande à l'utilisateur la somme S et renvoie les nombres de billets de 10, 20 et 50 euros qui seront distribués.
2. Écrire une fonction de paramètre la somme S qui renvoie le nombre total n de billets rendus lorsque S est un multiple de 10.
3. Tracer à l'aide d'un programme les points de coordonnées $(S; n)$.
4. Écrire une fonction `Sol_automate(n)` qui renvoie toutes les sommes que l'on peut obtenir avec n billets.

7.2 La correction

1. On fait rentrer la somme S par l'utilisateur jusqu'à ce que ce nombre soit un multiple de 10 inférieur à 500.
   On cherche ensuite le nombre maximum c de billets de 50 euros pour avoir $c*50 \leq S < (c+1)*50$: c'est donc `c :=iquo(S,50)`. Puis, il reste a fournir la somme $S - 50*c$ (`S :=S-50c`) et on cherche le nombre maximum de billets de 20 euros pour avoir etc...
   On tape :

```
Automate():={
  local a,b,c,S;
  repeter
    saisir("S<=500 multiple de 10",S);
  jusqua S<=500 et irem(S,10)==0;
```

```
  c:=iquo(S,50);
  S:=S-50*c;
  b:=iquo(S,20);
  S:=S-20*b;
  a:=iquo(S,10);
  print(a*10+20*b+50*c);
  retourne a,b,c;
}:;
```

On fait afficher `print(a*10+20*b+50*c) ;` pour vérifier que le compte est bon ! ! ! !

2. Ici la somme S est le paramètre de la fonction, il faut donc renvoyer une erreur si S n'est pas un multiple de 10.
   On fait ensuite les mêmes instructions que précédemment en remarquant que `S-50c=irem(S,50)` et on renvoie $a+b+c$

```
N_automate(S):={
  local a,b,c;
  si irem(S,10)!=0 alors
    retourne "S n'est pas un multiple de 10";
  fsi;
  c:=iquo(S,50);
  S:=irem(S,50);
  b:=iquo(S,20);
  S:=irem(S,20);
  a:=iquo(S,10);
  retourne a+b+c;
}:;
```

3. Ici on utilise une séquence P qui contiendra les 50 points du graphe. Au début on initialise par `P:=NULL;`, la séquence de points est vide. Puis on rajoute à cette séquence `P`, les points au fur et à mesure à l'aide de la fonction précédente, en mettant `P:=P,point(s,N_automate(s))`.

```
Sn_automate():={
  local s,n,P;
  P:=NULL;
  pour s de 10 jusque 500 pas 10 faire
    P:=P,point(s,N_automate(s));
  fpour;
  retourne P;
}:;
```

4. Ici, le graphique précédent peut aider à faire le raisonnement suivant :
   lorsque $n = 1$ la somme peut être de 50, 20 ou 10 euros et
   lorsque $n > 1$, avec n billets, on peut avoir :
   – n billets de 50,
   – $n - 1$ billets de 50 et un billet de 20,
   – $n - 1$ billets de 50 et un billet de 10,
   – $n - 2$ billets de 50 et 2 billets de 20 ou
   – $n - 2$ billets de 50 et un billet de 20 et un billet de 10

On tape donc :

```
Sol_automate(n):={
 si n==1 alors retourne 50,20,10 fsi;
 retourne 50*n,50*(n-1)+20,50*(n-1)+10,50*(n-2)+40,50*(n-2)+30;
}
```

8 Suggestion : algorithmique et code génétique

Nous proposons ici quelques exercices d'algorithmique sur le thème de la synthèse des protéines à partir du code génétique contenu dans les chaine d'ADN. Nous donnons ici quelques explications pour motiver les exercices qui suivent, que le lecteur uniquement intéressé par les questions algorithmiques peut naturellement sauter.

Le code génétique se trouve dans les chromosomes sous forme de la fameuse hélice d'ADN (l'hélice est repliée sur elle-même à l'intérieur du chromosome), celle-ci est composée de deux brins, chaque brin d'ADN comportant une suite de bases parmi 4 bases notées A, C, G et T. Les 2 brins d'ADN se faisant face voient leurs bases se compléter selon la règle : à A correspond T, à C correspond G et réciproquement. Ceci permet de dupliquer facilement le code génétique lorsqu'une cellule se multiplie.

Lors de la synthèse d'une protéine, un des brins d'ADN est traduit en une chaine d'ARN. L'ARN peut aussi se voir comme une succession de bases parmi les 4 bases : A, C, G et U qui remplace T avec les même règles de correspondance (A donne U, T donne A, C donne G et G donne C). Contrairement à l'ADN qui se replie en hélice et n'est donc pas contenu dans un plan, l'ARN reste essentiellement contenu dans un plan. C'est l'ARN qui permet de synthétiser les protéines. Une protéine peut être vue comme une suite d'acides aminés, il y a 20 acides aminés différents (codés avec 20 lettres de l'alphabet). la synthèse est initiée au niveau d'un triplet de bases, toujours le même, appelé codon de start (AUG). Elle se poursuit ensuite par triplets de bases d'ARN jusqu'à ce qu'on recontre un des trois triplets de bases terminal, appelé codons de stop (UAA, UAG, UGA). Il y a $4^3 = 64$ codons possibles pour 20 acides aminés, donc plusieurs codons peuvent donner le même acide aminé, selon le tableau suivant :

	U			C			A			G			
U	UUU	Phe	[F]	UCU	Ser	[S]	UAU	Tyr	[Y]	UGU	Cys	[C]	U
	UUC	Phe	[F]	UCC	Ser	[S]	UAC	Tyr	[Y]	UGC	Cys	[C]	C
	UUA	Leu	[L]	UCA	Ser	[S]	UAA	*STOP*		UGA	*STOP*		A
	UUG	Leu	[L]	UCG	Ser	[S]	UAG	*STOP*		UGG	Trp	[W]	G
C	CUU	Leu	[L]	CCU	Pro	[P]	CAU	His	[H]	CGU	Arg	[R]	U
	CUC	Leu	[L]	CCC	Pro	[P]	CAC	His	[H]	CGC	Arg	[R]	C
	CUA	Leu	[L]	CCA	Pro	[P]	CAA	Gln	[Q]	CGA	Arg	[R]	A
	CUG	Leu	[L]	CCG	Pro	[P]	CAG	Gln	[Q]	CGG	Arg	[R]	G
A	AUU	Ile	[I]	ACU	Thr	[T]	AAU	Asn	[N]	AGU	Ser	[S]	U
	AUC	Ile	[I]	ACC	Thr	[T]	AAC	Asn	[N]	AGC	Ser	[S]	C
	AUA	Ile	[I]	ACA	Thr	[T]	AAA	Lys	[K]	AGA	Arg	[R]	A
	AUG	Met	[M]	ACG	Thr	[T]	AAG	Lys	[K]	AGG	Arg	[R]	G
G	GUU	Val	[V]	GCU	Ala	[A]	GAU	Asp	[D]	GGU	Gly	[G]	U
	GUC	Val	[V]	GCC	Ala	[A]	GAC	Asp	[D]	GGC	Gly	[G]	C
	GUA	Val	[V]	GCA	Ala	[A]	GAA	Glu	[E]	GGA	Gly	[G]	A
	GUG	Val	[V]	GCG	Ala	[A]	GAG	Glu	[E]	GGG	Gly	[G]	G

La succession des acides aminés d'une protéïne détermine ensuite la façon dont elle se replie dans l'espace, et donc sa forme géométrique, et c'est celle-ci qui est responsable des propriétés de la protéïne.

Certaines étapes de la traduction ont été ici volontairement simplifiées, par exemple la possibilité de couper des morceaux d'ARN et de les recoller entre eux,

ou l'existence de sites "promoteurs" situés quelques caractères avant le codon de start pour démarrer la synthèse d'une protéïne...

Pour les chaines de caractère, on utilise en `Xcas` le délimiteur `"` au début et à la fin de la chaine, par exemple `s:="ACGGTCC"`. Les caractères sont numérotés en commençant à 0, jusqu'à la taille de la chaine moins un (`dim(s)-1`), ainsi `s[0]` désigne le caractère `"A"` ci-dessus. Pour ajouter un caractère à une chaine, on utilise l'opérateur +.

Exemple : traduction d'une chaine d'ADN en chaine d'ARN

```
adn2arn(adn):={
  local arn,j,s;
  s:=dim(adn);
  arn:=""; // chaine d'ARN vide au debut
  pour j de 0 jusque s-1 faire
    si adn[j]=="A" alors arn := arn + "U"; fsi;
    si adn[j]=="C" alors arn := arn + "G"; fsi;
    si adn[j]=="G" alors arn := arn + "C"; fsi;
    si adn[j]=="T" alors arn := arn + "A"; fsi;
  fpour;
  retourne arn;
}:;
```

Exercices

1. Ecrire une fonction testant si un caractère est parmi A, C, G, T. On renvoie 1 si c'est le cas et 0 sinon.
2. Ecrire une fonction qui teste si une chaine de caractère n'a que des caractères A, C, G, T.
3. Ecrire une fonction qui compte le nombre de A, C, G et T dans une chaine.
4. Ecrire une fonction qui teste si deux caractères sont complémentaires (A correspond à T et C à G)
5. Ecrire une fonction qui teste si deux chaines sont complémentaires.
6. Ecrire une fonction qui renvoie la chaine ARN complémentaire d'une chaine ADN.
7. Écrire une fonction renvoyant la position d'un codon de start (triplet de lettres AUG) dans une chaine d'ARN. Faites de même pour déterminer la position d'un codon de stop (un des triplets UAA, UAG ou UGA) à partir d'une position donnée (attention il faut se déplacer de 3 en 3 dans la chaine à partir du codon de start).
8. Écrire un programme qui détermine le début et la fin d'une partie codante d'une séquence d'ARN (en recherchant le premier triplet de démarrage AUG et l'un des triplets de fin correspondants UAA, UGA, UAG). On renverra la longueur de la séquence codante et le rapport du nombre de bases AU sur le nombre de bases CG.
9. On cherche souvent à comparer une chaîne ADN à une autre chaîne connue. Pour le faire, construire une fonction `poids(lettre1, lettre2)` qui retournera 2 si lettre1 et lettre2 sont égales et -1 sinon. Ensuite, écrire une

fonction calculant le poids total obtenu en sommant tous ces poids pour les caractères de deux chaines de même longueur.
Modifier la fonction poids pour renvoyer 0 si les caractères sont complémentaires (ceci afin de tenir compte des échanges entre bases complémentaires au sein de la double hélice d'ADN). Ceci permet de voir si les deux chaines sont proches ou non quantitativement parlant, et peut ensuite servir à comparer une chaine à des chaines connues en trouvant la plus "proche" dans une base de données.

Pour aller plus loin (tenir compte de chaine de longueur différente ou de décalage possible), on peut programmer un algorithme d'alignements de chaines (d'ADN, de protéines,...), cf. par exemple
`www-fourier.ujf-grenoble.fr/~parisse/info/dynamic/dynamic.html`

9 Glossaire

9.1 Pour écrire une fonction ou un programme

Vérifier au préalable soit avec le menu `Cfg->Configuration du CAS`, soit en cliquant sur la bouton `Config` en haut de la session, que :

- vous avez choisi la syntaxe en mode `Xcas`,
- l'unité d'angle est la bonne pour des programmes de géométrie, en cochant ou décochant `radian` dans la fenêtre de configuration qui s'ouvre en appuyant sur la barre `Config`,

Puis :

1. ouvrir un niveau éditeur de programme soit en tapant `Alt-p`, soit avec le menu `Prg->Nouveau programme`. Il contient déjà le `:;` qui doit terminer le programme.
2. taper la fonction en terminant chaque instruction par `;`. Les noms de cette fonction, de ses arguments, de ses variables locales ne doivent pas déjà être utilisés par `Xcas`. On peut commencer le nom des fonctions par une Majuscule pour diminuer le risques de conflits avec une fonction qui existe déjà dans `Xcas`. Notez que dans un niveau éditeur de programmes, les mots clés apparaissent en bleu et les nomms de commandes `Xcas` apparaissent en brun.
3. appuyer sur `OK(F9)` (touche F9), pour compiler le programme.
4. pour exécuter le programme, on se place dans une ligne de commande vide, on tape le nom de la fonction suivi entre parenthèses par les valeurs des paramètres séparées par des virgules. Pour l'exécuter en mode pas à pas, on précède le nom de la fonction de `debug(` et on clot la parenthèse à la fin.

9.2 Le menu `Add` d'un niveau éditeur de programme

Ce menu vous permet d'avoir facilement la syntaxe d'une fonction, d'un test et des différentes boucles. Par exemple une fonction s'écrit avec la syntaxe suivante :

```
f(x,y):={
  local z,a,b,...,val;
```

```
  instruction1;
  .....
  instructionk;
};;
```

On termine par `:;` pour que la réponse à une compilation reussie soit `Done` ou par `;` pour avoir en réponse, la traduction du programme après la compilation.

9.3 Les instructions en français

Instructions en fançais	
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;`
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;`
boucle faire	`faire <inst1> si <condition> break;<inst2> ffaire;`

9.4 Les instructions comme en C

Instructions comme en C++	
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;`
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>};`
boucle faire	`do <inst1> if (<condition>) break;<inst2> od;`

9.5 Les instructions en mode Maple

Instructions en mode Maple	
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 by p do <inst> od;`
boucle tantque	`while <condition> do <inst> od;`

(Attention, l'instruction `while ... do ... od;` nécessite d'avoir choisi la syntaxe en mode compatible Maple).

9.6 Signification des signes de ponctuation

Signification des signes de ponctuation	
`.`	sépare la partie entière de la partie décimale
`,`	sépare les éléments d'une liste ou d'une séquence
`;`	termine chaque instruction d'un programme
`:;`	termine les instructions dont on ne veut pas l'affichage de la réponse
`!`	factorielle

9.7 Les opérateurs

Opérateurs	
`+`	addition
`-`	soustraction
`*`	mutiplication
`/`	division
`^`	puissance
`==`	teste l'égalité
`!=`	teste la différence
`<`	teste la stricte infériorité
`<=`	teste l'infériorité ou l'égalité
`>`	teste la stricte supériorité
`>=`	teste la supériorité ou l'égalité
`ou`	opérateur booléen infixé
`et`	opérateur booléen infixé
`non`	renvoie l'inverse logique de l'argument
`vrai`	est le booléen vrai ou true ou 1
`faux`	est le booléen faux ou false ou 0

9.8 Séquences, listes et chaines de caractères

Séquences et listes	
`S:=a,b,c`	S est une séquence de 3 élements
`L:=[a,b,c]`	L est une liste de 3 élements
`S:=NULL`	S est une séquence de 0 élement
`L:=[]`	L est une liste de 0 élement
`dim(S)`	renvoie le nombre d'éléments de S
`S[0]`	renvoie le premier élément de S
`S[n]`	renvoie le $n+1$ unième élément de S
`S[dim(S)-1]`	renvoie le dernier élément de S
`S:=S,d`	ajoute l'élément d à la fin de la séquence S
`L:=append(L,d)`	ajoute l'élément d à la fin de la séquence L

Chaines de caractères	
`S:="abc"`	S est une chaine de 3 caractères
`S:=""`	S est une chaine de 0 caractère
`dim(S)`	renvoie le nombre de caractères de S
`S[0]`	renvoie le premier caractère de S
`S[n]`	renvoie le $n+1$ unième caractère de S
`S[dim(S)-1]`	renvoie le dernier caractère de S
`S+d`	ajoute le caractère d à la fin de la chaine S

9.9 Les fonctions utilisées

Fonctions mathématiques	
`floor(t)`	renvoie la partie entière t
`round(t)`	renvoie l'entier le plus proche de t
`irem(a,b)`	renvoie le reste de la division de a par b
`iquo(a,b)`	renvoie le quotient de la division de a par b
`abs(t)`	renvoie la valeur absolue de t
`sqr(t)`	renvoie la racine carrée de t

Fonctions de géométrie	
`A:=point(a,b)`	point A de coordonnées (a, b)
`affichage=` `epaisseur_point_5`	3ième argument de `point` pour le dessiner avec une croix d'épaisseur 5
`droite(A,B)`	droite AB
`triangle(A,B,C)`	triangle ABC
`bissectrice(A,B,C)`	bissectrice de l'angle A du triangle ABC
`angle(A,B,C)`	valeur de la mesure (radians ou degrés) de l'angle A du triangle ABC
`mediane(A,B,C)`	médiane issue de A du triangle ABC
`mediatrice(A,B)`	médiatrice de AB
`cercle(A,r)`	cercle de centre A et de rayon r
`cercle(A,B)`	cercle de diamètre AB
`rayon(c)`	rayon du cercle c
`centre(c)`	centre du cercle c
`distance(A,B)`	longueur de AB
`distance(A,d)`	distance de A à la droite d
`inter(G1,G2)`	liste des points de $G1 \cap G2$
`inter_unique(G1,G2)`	un des points de $G1 \cap G2$
`rotation(A,t,B)`	point transformé de B par la rotation de centre A et d'angle t