CarScripts

Alain Busser

October 11, 2011

Contents

Le langage Javascript 5

Affectation et égalité 5

Tests 5

Boucles 6

Boucles simples 6

Boucles avec condition 6

Nombres 6

Mathématiques 7

Tableaux 7

Entrées-Sorties 9

Impression d'un texte 9

Impression avec retour à la ligne 10

Texte dans la figure 10

Création 10

Modification 11

Obtention 11

Affichage 11

Entrée de données 12

Choix d'un objet 13
Obtention d'un nombre 14
Obtention d'une abscisse 14
Obtention d'une ordonnée 15
Déplacement d'un objet 15

Propriétés des objets **19**

Visibilité 19
Montrer 19
Cacher 19
Existence 20
Futurs objets 20
Nom 20
Alias 20
Valeur 21
Calque 21
Trace 21
Couleur 22
Couleurs prédéfinies 22
Couleurs d'une palette personnelle 22
Composantes de couleur 24
Style 26
Style des points 26
Style des traits 26
Style des angles 27
Remplissage 28
Dessin partiel 28

Propriétés conditionnelles 29
Restriction de mouvements 30
Portée du champ magnétique 30
Propriété magnétique 30
Point sur objet 31
Fixer un objet 32

Points **33**

Un point c'est tout 33
Milieu 35
À partir d'un vecteur 36
Origine 36
Extrémité 36
Symétrie centrale 36
Symétrie axiale 37
Translation 38
Intersection 39
Premier point 39
Tous les points 39
Tableau de points 40
Homothéties 41

Lignes **43**

Droites 43

Parallèle 44

Perpendiculaire 44

Médiatrice 45

Demi-droites 46

Angles 47

Segments 48

Vecteurs 49

Polygones 50

Cercles 52

Par un point 52

Cercles de rayon donné 52

report d'une distance 53

Cercle circonscrit 54

Arc de cercle 55

Coniques 56

Fonctions **59**

Représentations graphiques 59

Courbes paramétrées 59

Lignes de niveau 60

Nombres 61

Suites 67

Espace **69**

Sphère pointilliste 69

Sphère polyédrale 70

Géométrie hyperbolique **75**
Points hyperboliques 75
Point hyperbolique 75
Milieu hyperbolique 76
Droites hyperboliques 77
Droite hyperbolique 77
Demi-droite hyperbolique 78
Perpendiculaire hyperbolique 78
Médiatrice hyperbolique 78
Segment hyperbolique 79
Symétries hyperboliques 79
Symétrie axiale 79
Bissectrice 79
Perpendiculaire commune 80
Symétrie centrale 81
Cercle hyperbolique 81

Le coin des hackers **83**
Chargement de fichiers 83
Fichier de données 83
Fichier JavaScript 87
Accès à la console de sortie 88
Accès à la construction 88
Équation de droite 89
Accès à la figure 90
Accès aux méthodes 90
Mise à jour 91

Une remarque pour les paresseux: Il est possible de copier les scripts (dans les zones beutées) et de les coller dans l'éditeur de CaRMetal (obtenu en cliquant sur "Javascript>Ouvrir l'éditeur de script")

Le langage Javascript

Développé à l'origine comme langage de script sous le navigateur *Netscape*, le langage ***Javascript*** tire son nom d'une très vague ressemblance avec *Java*. Mais c'est un langage interprété, et non compilé, et les scripts *javascript* sont écrits soit directement dans les pages internet en *html*, soit dans des fichiers texte avec l'extension ***.js***. Ce langage passera de mode lorsque Internet sera désuet, ce qui ne semble pas près d'arriver...

Affectation et égalité

Du point de vue algébrique, le principal défaut de *javascript* est que l'affectation se note "="[1]. Par exemple, l'incrémentation de la variable n se note $n = n+1$, ce qui du point de vue mathématique, est quelque peu étrange... Du coup comme on a parfois besoin de la vraie égalité, on la note "==". Par contre, "supérieur ou égal" se note ">=", etc. et "différent" se note "!=", le point d'exclamation représentant la négation.
Le "et" booléen se note "&&", et le "ou" (inclusif) se note "||" pour des raisons analogues.

Tests

Les instructions conditionnelles (par exemple, celles qu'on ne fait que si un nombre est pair) se notent "if (condition) {ce qu'on fait}" ou "if (condition) {ce qu'on fait} else {ce qu'on fait sinon}", où "condition" est un booléen (comme "n<10")

[1] en même temps, ce n'est de loin pas le seul langage de programmation ayant fait ce choix

et les accolades séparent des suites d'instructions en javascript, terminées par des points-virgules.

Boucles

Boucles simples

Pour exécuter 10 fois une suite d'instructions, on entre "for (n=0;n<10;n=n+1)", syntaxe pas très simple mais puissante parce qu'on peut la remplacer par des variantes plus puissantes comme l'appartenance de n à une liste ou un tableau.
Le "n=n+1" peut être remplacé par "n+=1" ou "n++" qui en sont des abréviations (pas nécessaires). De même la décrémentation se note "n- -".

Boucles avec condition

"while (condition) {suite d'instructions}" répète la suite d'instructions tant que le booléen "condition" est vrai. Cette boucle peut donc très bien ne jamais être exécutée.
La variante "do {suite d'instruction } while (condition)" fait de même mais à ceci près que le booléen n'est évalué qu'à la fin de la boucle, laquelle est donc effectuée au moins une fois.

Nombres

Le plus utile des nombres est "NaN" qui n'est pas un nombre! Il sert de message d'erreur, lorsqu'on attend un nombre et qu'on a autre chose (ou rien). Cette constante appartient à la catégorie "nombres" et se note donc "Number.NaN"
Les plus "analyse non standard" des nombres sont "POSITIVE_ INFINITY" et "NEGATIVE_ INFINITY". Essayer par exemple le script suivant:

```
Println(Number.NEGATIVE_INFINITY+2);
Println(Number.POSITIVE_INFINITY+Number.POSITIVE_INFINITY);
Println(Number.POSITIVE_INFINITY+Number.NEGATIVE_INFINITY);
Println(Number.POSITIVE_INFINITY/Number.POSITIVE_INFINITY);
```

On constate que ces "nombres" étant dans la catégorie "Number", on met le nom de cette catégorie avant celui de la constante elle-même. Les résultats imprimés reviennent à

$$-\infty + 2 = -\infty$$

$$+\infty + \infty = +\infty$$

$$\infty - \infty \ FI$$

et

$$\frac{\infty}{\infty} \ FI$$

(où FI est une abréviation pour "forme indéterminée") qui ne sont pas algébriquement corrects mais mnémotechniquement pratiques...

Mathématiques

Le nombre π est lui, dans la catégorie "Math" et se note donc "Math.PI"; il en est de même pour le nombre e, noté "Math.E".
Les fonctions trigonométriques, valeur absolue, arrondies et surtout puissance, sont aussi dans la catégorie "Math" et par exemple pour calculer le cube de x, la fonction "Math.pow(x,3)" est préférable à "x*x*x" comme on le verra plus bas.

Tableaux

Un tableau de nom t se crée par

```
var t = new Array();
```

Après ça on peut y placer des objets de toute sorte (en général texte ou nombres). Dans ce document on n'utilisera qu'un seul tableau (à la fin) mais pour se rattraper on le fera doublement indexé !

Entrées-Sorties

Impression d'un texte

Print suivi d'un texte entre parenthèses, affiche ce texte.
En javascript, la concaténation de textes se fait par "+", ainsi le texte "CaRMetal" peut aussi s'écrire "Car"+"Metal". L'intérêt apparaît lorsqu'on insère des nombres dans le texte, comme on le fait plus bas.
Par exemple, le code suivant:

```
Print("CaRMetal, c'est génial!")
```

produit l'effet suivant:
Ouverture d'une console, dans laquelle s'écrit le texte "CaRMetal, c'est génial!"
Autre exemple: On tape

```
{for (i=0; i<12; i++){
Print(i)
}
```

On obtient alors:
01234567891011
À méditer...

Impression avec retour à la ligne

Println écrit aussi mais va à la ligne.
Par exemple, le code

```
Println("CaRMetal,")
Println("c'est génial!")
```

écrit le texte:
CaRMetal,
c'est génial!
Aussi, le code

```
for (i=0; i<12; i++){
Println(i)
}
```

écrit les nombres de 0 à 11 en colonne, ce qui est plus lisible que l'exemple précédent!

Texte dans la figure

Création

Text suivi du nom du texte (optionnel), du texte lui-même, et de deux coordonnées, crée un objet *Texte* dans la figure CaRMetal. Tout ce qui est admis dans les objets *texte* de CaRMetal, l'est aussi dans les CaRScripts. Par exemple, le langage LaTeX:

```
t=Text("$\\frac{\\sqrt{2}}{2}$",-4,3);
SetColor(t,"blue");
```

mais dans ce cas, le caractère ”backslash” doit être dédoublé pour éviter que JavaScript ne le considère comme le début d'un caractère d'échappement.

Modification

Pour modifier un texte dans la figure, on peut utiliser **SetText**, suivi du nom de l'objet, et du nouveau texte.

```
t=Text("$\\frac{\\sqrt{2}}{2}$",-4,3);
SetText(t,"%pixel%");
```

Obtention

GetText, suivi du nom d'un objet de type texte (qui est dans la figure), récupère le contenu (le texte) de cet objet. En combinaison avec **SetText**, il permet d'échanger des données entre la figure et le CaRScript.

Affichage

Alert suivi d'un texte entre parenthèses, affiche ce texte dans une boîte de dialogue.
Par exemple, le code suivant

```
Alert("CaRMetal, c'est génial!");
```

fait apparaître au-dessus de la figure, la boîte suivante:

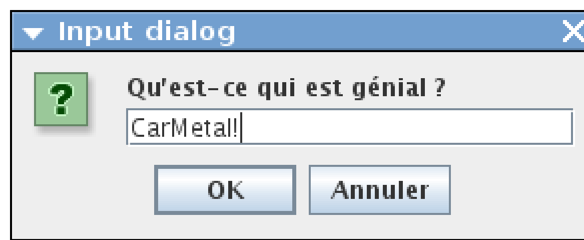
qui disparaît lorsqu'on clique sur "OK".

Entrée de données

Input suivi d'un texte, affiche ce dernier et retourne la réponse une fois qu'elle a été entrée par l'utilisateur dans une boîte de dialogue.
Par exemple, le code

```
rep=Input("Qu'est-ce qui est génial?");
```

affiche la boîte de dialogue suivante:



puis, lorsqu'on clique sur "OK" après avoir répondu, la variable "rep" contient le texte de la réponse (ici, c'est "CaRMetal!"). Cependant, si on entre

```
n=Input("Entrez un nombre entre 1 et 7");
```

la variable "n" contiendra alors un nombre d'un chiffre[2]...

Choix d'un objet

InteractiveInput (obtenu dans l'éditeur en cliquant sur le point d'interrogation) affiche un message en bas de la figure, demandant de cliquer sur un des objets de la figure. Le type de l'objet doit être un des cinq suivants:

- Point: texte "Point"
- Droite: Texte "Line"
- Segment: Texte "Segment"
- Cercle: Texte "Circle"
- Nombre: Texte "Expression"

Le texte figure entre parenthèses après le message.
Exemple: En supposant que dans la figure il y a des droites, le code suivant:

```
d1=InteractiveInput("Choisir une droite","Line");
```

interrompt le déroulement du Carscript, affichant le texte "Choisir une droite" en bas de la figure, et attend que l'utilisateur ait cliqué sur une droite. d1 est alors égal au nom de cette droite.

[2] à condition que l'utilisateur ait respecté la consigne. L'utilisation de tests, avec "if(...)", ou de répétitions, avec "while(...)", est recommandée pour garantir que le script fasse vraiment ce qu'on veut, mais cela allonge les scripts, ce qui est contraire à l'esprit d'un tutoriel. Néanmoins c'est important et les bonnes habitudes se prennent tôt

Obtention d'un nombre

GetExpressionValue, suivie du nom d'un nombre qui est déjà dans la figure, permet au Carscript d'avoir la valeur de ce nombre. Ce nombre peut très bien être un curseur!
Par exemple, si on a créé dans la figure une variable E1 contenant "windoww" (affichage sur la figure: "E1=8" si la fenêtre a 8 unités de largeur), le code suivant:

```
n=GetExpressionValue("E1");
Print("la fenêtre a " + n + " unités de large");
```

crée une fenêtre avec le texte "la fenêtre a 8 unités de large"

Obtention d'une abscisse

X suivie, entre parenthèses, du nom d'un objet, fournit son abscisse.
L'objet n'est pas nécessairement un point:
Si la figure comprend une droite l1, le code suivant:

```
x=X("l1");
y=Y("l1");
Println("l'abscisse est " + x);
Println("et l'ordonnée est " + y);
```

affiche les coordonnées d'un vecteur directeur unitaire de la droite...
Si l'objet n'existe pas, **X** retourne le texte "NaN".

Obtention d'une ordonnée

Y suivie, entre parenthèses, du nom d'un objet, fournit son ordonnée.
L'objet n'est pas nécessairement un point:
Si la figure comprend un cercle c1, le code suivant:

```
x=X("c1");
y=Y("c1");
Println("l'abscisse est " + x);
Println("et l'ordonnée est " + y);
```

affiche les coordonnées du centre du cercle.
Si l'objet n'existe pas, **Y** retourne le texte "NaN"[3].

Déplacement d'un objet

Move déplace un objet, dont le nom est fourni entre parenthèses, suivi par les nouvelles coordonnées. Par exemple, pour simuler un pendule dont la masse (au bout de l'élastique) est soumise à un mouvement brownien, on peut essayer le code suivant:

```
o=Point(0,0); // l'origine
m=Point(0,-2); // point mobile
c=Segment(o,m); // pendule
SetHide(o,true); // on cache l'origine
x=0; // abscisse initiale
y=-2; // ordonnée initiale
for (i=0; i<200; i++){ // on fait le tout 200 fois
    x=x+Math.random()/10-1/20; // on modifie x
    y=y+Math.random()/10-1/20; // on modifie y
    Move(m,x,y); // on bouge le point m
```

[3] abréviation de "not a number"

```
    Pause(50); // pas trop vite!
}
```

C'est un peu long mais les commentaires (complètement inutiles pour CaRMetal mais utiles pour nous !) aident à comprendre de quoi il s'agit: On commence par créer un segment $[om]$ puis cacher l'origine o; puis on bouge légèrement m, deux cents fois de suite, et on attend 50 millisecondes (pour que ça n'aille pas trop vite). Le mouvement de m se fait en ajoutant à chacune de ses coordonnées, un nombre aléatoire uniforme compris entre -0,05 et 0,05.

Note: Pour un carscript aussi long, il peut être intéressant de l'écrire dans un éditeur de texte externe (comme "wordpad" ou "jext") et de le sauvegarder sous un nom tel que "jokari.js"[4]. Dans ce cas, il suffit de cliquer sur "Javascript>Exécuter un programme javascript" pour ouvrir le fichier et CaRMetal l'éxécutera tout seul. Dans ce cas il peut être intéressant pour l'avenir, de rajouter un préambule tel que celui-ci:

```
/*    programme jokari
une simulation de Jokari
Author: Nom de l'auteur
Date: Mai 2009
License: GPL 2.0 or above*/
```

qui permettra de mutualiser des programmes carscript intéressants, sous licence GPL. Les symboles "/*" et "*/" délimitent des longs commentaires (de plusieurs lignes) alors que "//" termine la ligne par un commentaire court.

[4]le nom du fichier est expliqué par exemple en suivant ce lien

Propriétés des objets

Visibilité

Montrer

Show suivi d'une liste de noms d'objets, rend visibles ces objets. Par exemple, si on a créé un point A,

```
Show("A");
```

le rend visible sur la figure.
SetHide suivi d'une liste de noms d'objets et du booléen *false*, a le même effet.

Cacher

Hide suivi d'une liste de noms d'objets, rend invisibles ces objets. Par exemple, si on a créé un point A,

```
Hide("A");
```

le rend invisible. Mais il existe toujours et peut servir à construire d'autres objets.
SetHide, suivi d'une liste de noms d'objets ,et du booléen *true*, a le même effet.

Existence

Exists est une fonction *booléenne* qui répond par *true* ou *false* à la question "le point d'intersection a-t-il des coordonnées réelles?"; il accepte en argument le nom du point d'intersection créé auparavant.

Futurs objets

Montrer

Une fois que le carscript a exécuté **Shownames**, dorénavant les nouveaux objets seront nommés et leurs noms affichés.

Cacher

Une fois que le carscript a exécuté **Hidenames**, dorénavant les nouveaux objets seront nommés mais leurs noms ne seront pas affichés.

Nom

SetShowName suivi du nom d'un objet (ou plusieurs) et d'un booléen ("true" ou "false") rend visible ou non, selon la valeur du booléen, le nom de l'objet (ou des objets). Par exemple, dans un nuage de 200 points, il est sans doute peu utile d'afficher les noms des 200 points...

Alias

SetAlias suivi du nom d'un (ou plusieurs) objet, et du nom qu'on veut leur donner, nomme l'objet (ou les objets). Par exemple, si un point A a été créé, et son nom visible, le script

```
SetAlias("A","%x(A)%");
```

fait que dorénavant, le nom de A sera l'abscisse de A...

Valeur

SetShowValue, suivi du nom d'un objet (ou plusieurs) et d'un booléen ("true" ou "false") rend visible ou non, selon la valeur du booléen, la valeur de l'objet (coordonnées d'un point, longueur d'un segment ou d'un vecteur, rayon d'un cercle ou aire d'un polygone). Par exemple, si un point A a été créé,

```
SetShowValue("A",true);
```

fera afficher à côté du nom de A (ou de son alias), ses coordonnées.

Calque

Le numéro de calque d'un objet détermine quels sont les objets placés au-dessus de lui (ceux dont le numéro de calque est inférieur au sien). Par défaut, le numéro de calque d'un objet est 0. Donc en donnant à un objet un numéro de calque supérieur à 0, on le place en-dessous des autres objets. Le numéro de calque d'un objet est un entier et se détermine par **Layer**. Par exemple, pour mettre le centre d'un disque sous celui-ci, on peut faire ceci:

```
a=Point(-2,1);
c=FixedCircle(a,3);
SetFilled(c,true);
Layer(a,1);
```

Trace

PenDown, suivi du nom d'un objet et de *true* (un booléen), active la trace de l'objet. Dorénavant si l'objet bouge, il laisse une trace à l'écran (mais celle-ci ne s'exporte pas avec la figure produite). Si on veut que l'objet cesse de laisser une trace, on utilise aussi **PenDown**, mais avec *false*.

Couleur

Couleurs prédéfinies

SetColor suivi, entre parenthèses, du nom d'un objet déjà créé et du nom de la couleur, change la couleur de l'objet nommé. La couleur est écrite en anglais, parmi les six suivantes: "green", "blue", "brown", "cyan", "red" et "black". L'effet ne sera pas visible si l'objet est caché. Par exemple, si un point A a été créé, le code suivant le met en vert:

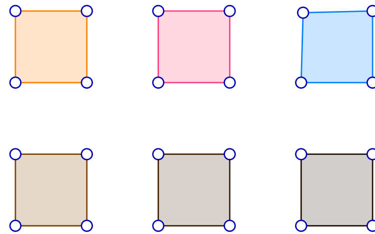
```
SetColor("A","green");
```

Couleurs d'une palette personnelle

SetRGBColor suivi du nom d'un objet puis de trois entiers, colorie cet objet avec une couleur obtenue à partir des nombres entiers, selon la recette suivante: Le premier nombre entre 0 et 255 indique la quantité de rouge: 0 pour noir et 255 pour un maximum de rouge. Le deuxième fait de même pour le vert, et le troisième pour le bleu.
Par exemple, si on a créé des carrés nommés poly2, poly4, poly6, poly8, poly10 et poly12 (de gauche à droite et de haut en bas), le code suivant:

```
SetRGBColor("poly2",255,127,0);
SetRGBColor("poly4",255,63,127);
SetRGBColor("poly6",0,127,255);
SetRGBColor("poly8",127,63,0);
SetRGBColor("poly10",63,32,0);
SetRGBColor("poly12",32,16,0);
```

produit l'image suivante:



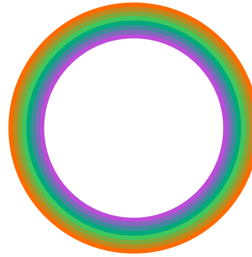
Rappel: Recette pour fabriquer quelques couleurs:

Rouge	***Vert***	***Bleu***	***Résultat***
255	0	0	Rouge
0	255	0	Vert
0	0	255	Bleu
0	255	255	Cyan
255	0	255	Magenta
255	255	0	Jaune
255	255	255	Blanc
0	0	0	Noir

Allez, un autre exemple: Dessiner un arc-en-ciel:

```
o=Point(0,0);SetHide(o,true); // centre des cercles
for (i=8; i>=0; i--){ // sept couleurs et le blanc
    r=FixedCircle(o,2+i/10); // l'arc en ciel d'une couleur
    SetThickness(r,"thick"); // doit cacher les autres
    SetFilled(r,true); // rempli
    SetRGBColor(r,63*Math.abs(i-4),200-31*Math.abs(i-5),31*(8-i));
}
SetRGBColor(r,255,255,255) // le cercle central en blanc
```

La figure obtenue est perfectible:



Par exemple on peut retoucher les formules donnant le rouge, le vert et le bleu (en faisant attention toutefois à obtenir des entiers) ou rajouter un rectangle blanc qui cache le bas de l'arc-en-ciel...

Composantes de couleur

Lire la quantité

GetRed suivi du nom d'un objet, renvoie sous forme d'un entier de 0 à 255, la quantité de rouge que possède cet objet.
GetGreen suivi du nom d'un objet, renvoie sous forme d'un entier de 0 à 255, la quantité de vert que possède cet objet.
GetBlue suivi du nom d'un objet, renvoie sous forme d'un entier de 0 à 255, la quantité de bleu que possède cet objet.

Par exemple, pour savoir quelle est la recette de la couleur marron, on peut créer un objet (ici un point), le colorier en marron, puis lire les composantes de rouge, de vert et de bleu de cet objet:

```
a=Point(-4,3);
SetColor(a,"brown");
Println(GetRed(a));
Println(GetGreen(a));
Println(GetBlue(a));
```

Changer la quantité

SetRed, suivi du nom d'un objet et d'un entier entre 0 et 255, met la quantité de rouge de l'objet à l'entier fourni. Par exemple, pour enlever le rouge d'un objet, on peut faire

```
a=Point(-4,3);
SetRed(a,0);
```

Seulement cela aura pour effet de réinitialiser les composantes verte et bleue aussi à 0.
SetGreen, suivi du nom d'un objet et d'un entier entre 0 et 255, fixe la composante verte de l'objet à l'entier fourni. Par exemple,

```
a=Point(-4,3);
SetRed(a,0);
SetGreen(a,100);
SetRed(a,0);
```

Le premier appel à *SetRed* met toutes les composantes du point à 0. Mais pas la deuxième, parce que la composante verte, qui est déjà de 100, n'a pas besoin d'être réinitialisée.
SetBlue, suivie du nom d'un objet et d'un entier entre 0 et 255, fixe la composante bleue de cet objet à l'entier fourni. Les deux scripts suivants ont donc exactement le même effet:

```
a=Point(-4,3);
SetRed(a,31);
SetGreen(a,67);
SetBlue(a,127);
```

```
a=Point(-4,3);
SetRGBColor(a,31,67,127);
```

Style

Style des points

SetPointType suivi d'une liste de noms de points, et d'un nom de style, donne à ces points la forme donnée par le nom du style. Le style peut être "square" (carré), "circle" (rond), "diamond" (losange), "point" (un pixel), "cross" (+), "dcross" (*).
Par exemple, si des points A, B et C ont été créés, le code

```
SetPointType("A,B,C","cross");
```

va les représenter sous forme de croix (utile en statistique si on veut des nuages de points différents sur un même graphique).

SetIncrement, suivi du nom d'un point (ou d'une liste de noms) et d'un réel, place l'incrément de ce point à la valeur numérique donnée. Par exemple, le script suivant assigne les coordonnées d'un point à être entières:

```
a=Point(4,3);
SetIncrement(a,1);
```

Style des traits

SetThickness suivi du nom d'une ligne (droite, segment, cercle, conique, fonction) et du nom d'un style, règle l'épaisseur de la ligne. Les trois épaisseurs possibles sont, dans l'ordre croissant, "thin", "normal" et "thick". Par exemple, si une conique "quad1" a été créée, le script

```
SetThickness("quad1","thick");
```

la met en gras.

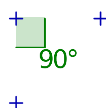
Style des angles

ReflexAngle, suivi du nom d'un angle (ou d'une liste de noms d'angles) et de *true*, permet à l'angle de prendre des valeurs supérieures à 180° (*angles rentrants*). Par exemple, le script suivant permet de construire un angle de 270°:

```
a=Angle(Point(1,0),Point(0,0),Point(0,-1));
e=Expression(a,-3,3);
```

Mais l'angle est affiché (tant sur la figure que dans l'expression créée en rouge) comme un angle droit de droites:

90

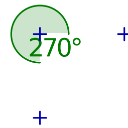


La variante suivante permet d'y remédier:

```
a=Angle(Point(1,0),Point(0,0),Point(0,-1));
ReflexAngle(a,true);
e=Expression(a,-3,3);
```

La figure devient alors

270



On peut changer d'avis en utilisant la même syntaxe avec *false* au lieu de *true*: L'angle redevient alors saillant.

Remplissage

SetFilled suivi du nom d'un objet et d'un booléen ("true" ou "false") remplit ou non l'objet. Un exemple a été donné avec l'arc-en-ciel.

Dessin partiel

SetPartial suivi du nom d'une droite ou d'un cercle et d'un booléen ("true" ou "false"), permet ("true") ou ou non ("false") le dessin partiel de l'objet. Ainsi si une droite l1 a été créée,

```
SetPartial("l1","true");
```

ne dessine qu'un segment dépassant légèrement les deux points qui l'ont définie, et si un cercle c1 a été défini,

```
SetPartial("c1","true");
```

ne montre qu'un arc centré sur le point qui a servi à le définir.

Propriétés conditionnelles

Conditional, suivi du nom d'un objet, du nom d'une propriété et d'un booléen (écrit sous la forme d'une chaîne de caractères dans le langage de CaRMetal), permet de donner à l'objet une propriété conditionnelle, dépendant du booléen. Par exemple la couleur du point P peut dépendre du signe de l'abscisse de P avec

```
Conditional("P","red","x(P)>0");
```

La propriété est l'une des suivantes:

- ***hidden***: Rend l'objet invisible.
- ***superhidden***: Rend l'objet "super-caché" (fonctionnalité décrite dans l'aide en ligne de CarMetal).
- ***solid***: rend l'objet opaque.
- ***thin***: Met les lignes en pointillé.
- ***normal***: Met les lignes en trait plein.
- ***thick***: Met les lignes en trait plein épais.
- ***showname***: Affiche le nom de l'objet.
- ***showvalue***: Affiche la valeur de l'objet (coordonnées pour un point, longueur ou aire etc.).
- ***black***: Met l'objet en noir.
- ***green***: Met l'objet en vert.
- ***blue***: Met l'objet en bleu.
- ***brown***: Met l'objet en marron.
- ***cyan***: Met l'objet en cyan.
- ***red***: Met l'objet en rouge.

Un exemple d'application de cette propriété est donné dans le dernier chapitre, avec la sphère polyédrique dont les arêtes sont en trait pointillés si elles sont cachées.

Restriction de mouvements

Portée du champ magnétique

SetMagneticRay suivi du nom d'un point et d'un nombre, fixe le rayon d'action du point à la valeur donnée. Par exemple, si un point A a été créé,

```
SetMagneticRay("A",30);
```

fera que A sera attiré par tous les objets de sa liste d'amis qui sont à moins de 30 pixels de A.

Propriété magnétique

Création de la liste d'amis

SetMagneticObjects suivi du nom d'un point et d'une liste de noms d'objets, fait de ces objets ceux qui peuvent attirer le point (sa "liste d'amis"). Ainsi, si quatre points A, B, C et M ont été créés,

```
SetMagneticObjects("M","A,B,C");
```

va obliger M à être égal à A, B ou C (du moins tant qu'il n'est pas trop loin de ces points)

Modification de la liste d'amis

AddMagneticObject suivi du nom de deux points, ajoute le deuxième point à la liste d'amis du premier.

Par exemple, on peut lier un point M mobile à un heptadécagone (à 17 côtés) régulier. Pour cela on crée l'un après l'autre les côtés de l'heptadécagone et au fur et à mesure, on y attache M.
Le code est le suivant:

```
m=Point("M",0,0); // le point mobile
SetMagneticRay("M",2000); // attiré de loin
a=Point(2,0); // premier sommet
SetHide(a,true);
for (i=1; i<18; i++){ // heptadécagone...
    x=2*Math.cos(2*Math.PI*i/17); // abscisse
    y=2*Math.sin(2*Math.PI*i/17); // ordonnée
    b=Point(x,y); // le sommet suivant
    SetHide(b,true);
    s=Segment(a,b); // un côté
    AddMagneticObject("M",s); // M peut aller dessus
    a=b // on change de sommet
}
```

Remarque: On peut obtenir le même résultat plus simplement, en créant une courbe paramétrée $x = 2\,cos(t)$ et $y = 2\,sin(t)$ puis en lui mettant un pas de $\frac{360}{17}$. Cette méthode (outre le fait qu'elle évite d'avoir recours à un script) donne un fichier plus léger, et permet de remplacer 17 par un curseur...

Point sur objet

PointOn, suivi du nom d'un point (optionnel) et de celui d'un objet, attache le point à l'objet.

Deux moyens d'attacher un point j à l'axe des abscisses:

```
o=Point(0,0);
i=Point(1,0);
d=Line(o,i);
j=Point(0,1);
PointOn(j,d);
```

ou

```
o=Point(0,0);
i=Point(1,0);
d=Line(o,i);
j=PointOn(d);
```

L'objet peut être une ligne droite, un cercle, une conique, un polygone, une représentation graphique de fonction...

Fixer un objet

SetFixed suivi du nom d'un objet (ou d'une liste d'objets) et d'un booléen ("true" ou "false"), rend cet objet (ou ces objets) fixe ou non selon la valeur du booléen. Cela évite de chambouler la figure avec la souris, et la rend insensible au Monkey.

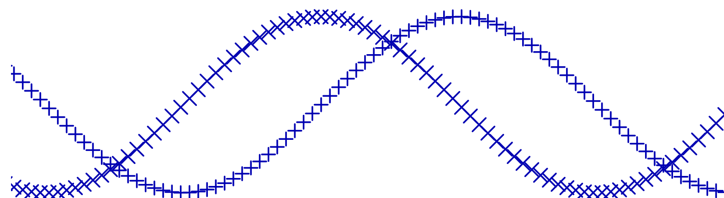
Points

Un point c'est tout

Point, suivi d'un nom et de deux nombres, crée un point de nom celui qui est proposé, d'abscisse le premier nombre, et d'ordonnée le second. Le nom peut très bien être vide comme dans "Point("",2,1)" voire absent: "Point(2,3)". Par exemple, les représentations graphiques en pointillés des fonctions sinus et cosinus s'obtiennent par ce code:

```
for (x=-8; x<8;x=x+0.1){ // pas de 0,1
    a=Point(x,Math.cos(x)); //le cosinus
    SetPointType(a,"dcross"); // en "fois"
    b=Point(x,Math.sin(x)); // et le sinus
    SetPointType(b,"cross"); // en "plus"
}
```

On obtient la figure suivante (où on n'a pas rendu la grille visible):



En réalité, ce n'est pas un point qui est créé, mais un texte javascript, dont le contenu est le nom du point. C'est le cas des objets CaRMetal créés en javascript: En fait les instructions créent des chaînes de caractères sous JavaScript, et ***accessoirement*** des objets graphiques... Comme en LISP, ce sont ces "effets de bord" qui sont recherchés dans les scripts. Mais la chaîne de caractère a son utilité puisqu'on peut s'en servir dans la suite du script.
Quelques exemples: Si on entre

```
a=Point("A",3,2);
```

dans un premier temps, sera créé un point de coordonnées (3,2), auquel le script tentera de donner le nom "A". S'il n'existait pas déjà un objet de ce nom, le point s'appellera effectivement "A". Sinon CaRMetal va lui donner un autre nom comme par exemple "A*". Dans tous les cas, la variable javascript "a" sera égale au nom effectivement donné au point (normalement, ce devrait être "A").
Dorénavant on peut écrire a au lieu de "A"...
Si par contre on entre l'une des deux lignes suivantes (au choix)

```
a=Point("",3,2);
a=Point(3,2);
```

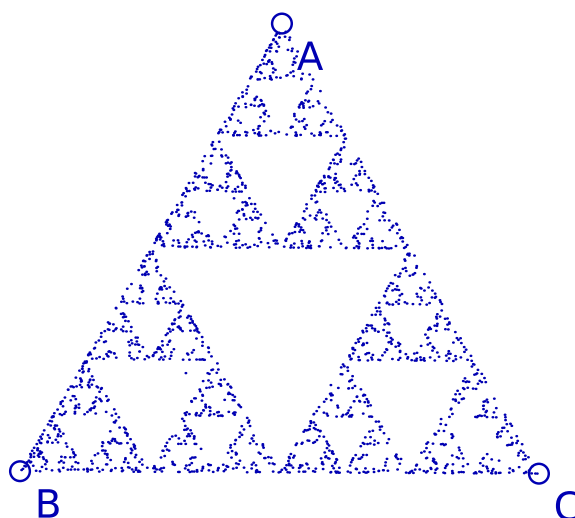
on laisse CaRMetal décider du nom du point créé, par exemple si rien n'a encore été créé, ce sera "$P1$". Dans ce cas, la variable a sera égale au texte "$P1$", et on pourra l'utiliser comme un texte, en le concaténant à autre chose par exemple...
Si au lieu du nom de la variable, on veut la variable elle-même, on précède ledit nom du caractère "underscore" qui sur le clavier se trouve sous le chiffre 8, et ressemble à ceci: "_", comme par exemple si une variable numérique s'appelle m, la suite "$_m$" représente la valeur de m, et non la lettre m. Enfin si a est égal au texte "$P1$", la notation x_a sera interprétée comme le texte "$x(P1)$".
On peut illustrer ceci sur le moyen le plus rapide pour obtenir le centre de gravité d'un triangle ABC déjà construit:

```
a="A";b="B";c="C";
g=Point("G","(x_a+x_b+x_c)/3","(y_a+y_b+y_c)/3");
```

Ici il faut comprendre "x_a" comme "$x(A)$" puisque a représente "A" (d'après la première ligne du script).

Milieu

MidPoint suivi d'une proposition de nom (optionnelle) et de deux noms de points déjà créés, construit l'isobarycentre de ces deux points. Par exemple, on peut dessiner le triangle de Sierpinski en 13 lignes de code! Pour cela, on a préalablement créé trois points A, B et C qu'il restera possible de manipuler à la souris. Ensuite on crée un point m invisible, de coordonnées aléatoires tant qu'on y est. Puis on répète 2000 fois la suite d'opérations suivante: On tire au sort un nombre t dont le premier chiffre est 0, 1 ou 2. Selon le résultat, on remplace m par le milieu de $[mA]$, $[mB]$ ou $[mC]$. Avec 2000 points il faut plusieurs secondes pour fabriquer le fichier, lequel "pèse" 200 kilooctets. Mais la figure obtenue



ne rend pas hommage au résultat, dans lequel on peut déplacer A, B et C, produisant un effet difficile à décrire! Le code est le suivant:

```
m=Point(Math.random(),Math.random());
SetHide(m,true);
for (i=0; i<2000; i++){
    t=3*Math.random();
    if (t<1){
        n=MidPoint(m,"A");
    } else if (t<2){
        n=MidPoint(m,"B");
    } else {
        n=MidPoint(m,"C");
    }
    SetPointType(n,"point");
    m=n;
}
```

À partir d'un vecteur

Origine

Origin, suivi d'une proposition de nom (optionnelle) et du nom d'un vecteur, renvoie l'origine du vecteur (un point).

Extrémité

Extremity, suivi d'une proposition de nom (optionnelle) et du nom d'un vecteur, renvoie l'extrémité du vecteur (un point).

Symétrie centrale

Symmetry suivi d'une proposition de nom (optionnelle) et de deux noms de points, réalise une symétrie centrale de l'un par rapport à l'autre. Le nom du

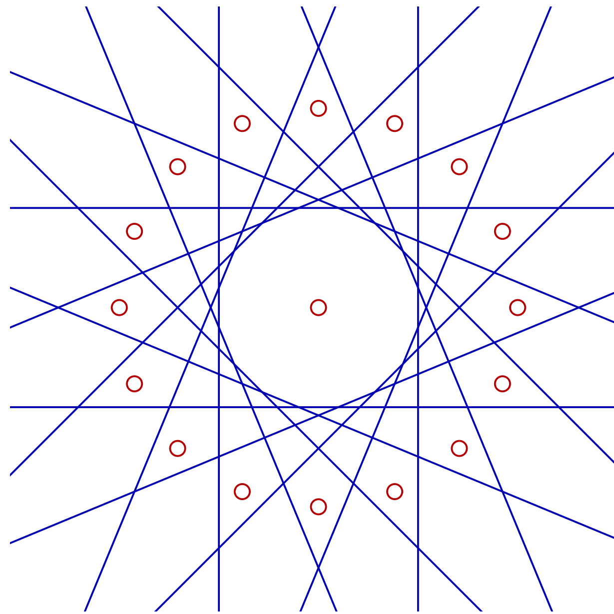
centre est donné d'abord, celui du point à transformer ensuite.
Pour un exemple voir l'hexagone régulier plus bas.

Symétrie axiale

Reflection, suivi du nom d'une droite et de celui d'un point, crée le symétrique du point par rapport à la droite. On peut remplacer la droite par une demi-droite ou un segment.
À titre d'exemple, voici un petit script pour illustrer que le lieu des symétriques du centre d'un cercle par rapport à ses tangentes est un cercle:

```
o=Point(0,0);
for(n=0;n<16;n++){
x=Math.cos(Math.PI*n/8);
y=Math.sin(Math.PI*n/8);
p=Point(x,y);SetHide(p,true);//un point du cercle
s=Segment(o,p);SetHide(s,true);//un rayon du cercle
m=Perpendicular(s,p);//une tangente au cercle
h=Reflection(m,o);
}
```

Voici l'effet obtenu:

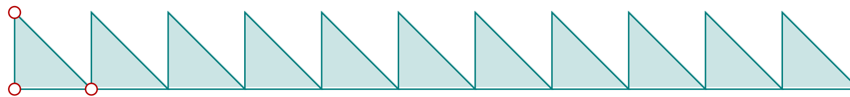


Translation

Translation, suivi du nom de trois points, crée l'image du troisième point par la translation qui envoie le premier point sur le deuxième. On peut donc s'en servir pour faire des frises:

```
a=Point(0,0);
b=Point(1,0);
c=Point(0,1);
p=Polygon(a+","+b+","+c);
d=a;
e=b;
f=c;
for(n=0;n<10;n=n+1){
d=Translation(a,b,d);SetHide(d,true);
e=Translation(a,b,e);SetHide(e,true);
f=Translation(a,b,f);SetHide(f,true);
p=Polygon(d+","+e+","+f);
}
```

La frise n'est pas très originale (d'ailleurs ont aurait pu la créer avec "Point" en choisissant des coordonnées adéquates):



mais elle est dynamique: On peut bouger les trois points rouges et la frise en est modifiée.

Intersection

Premier point

Intersection suivi des noms de deux droites, crée le point d'intersection de ces droites. Avec des segments ou des demi-droites ça marche aussi. Par exemple, l'intersection d'un cercle et d'une demi-droite partant de son centre est un point qui est créé par cette instruction. Mais avec une droite et un cercle, ou deux cercles, on obtient un seul des deux points. Si on cherche l'intersection de deux coniques, on n'a également qu'un seul point.
Pour un exemple d'utilisation, voir la construction de l'orthocentre faite avec des droites perpendiculaires, plus bas.

Tous les points

Intersection2 suivi du nom d'un cercle et d'une droite ou d'un cercle, crée d'un coup les deux points d'intersection. Si on veut les nommer il faut donc écrire "m=Intersection2("I1","I2",c1,c2)". Avec deux coniques **Intersection2** ne fournit qu'un seul point d'intersection. *Intersection2* retourne une liste de deux noms: Ceux des points d'intersection fournis. Ce qui permet de changer leur couleur ou leur aspect d'un coup. Mais pour avoir leurs noms, il est nécessaire de les mettre dans un tableau, ce que permet JavaScript avec la méthode *Split* de l'objet *String*. Celle-ci renvoie un tableau dont l'élément 0 est l'avant-dernier point d'intersection créé, et l'élément 1, le dernier point d'intersection.
Avec les symétriques et les intersections de cercles on peut construire un hexagone régulier:

```
a=Point(0,0); // centre du cercle
b=Point(2,0); // premier sommet
e=Symmetry(a,b);//sommet 4
z=Circle(a,b); // cercle circonscrit
t=Circle(b,a); // autre cercle
c=Intersection2(z,t); // deux nouveaux sommets
Sommets=c.split(",");
d=Symmetry(a,Sommets[0]); // encore un
f=Symmetry(a,Sommets[1]); // et un dernier
l=Polygon("_b,"+Sommets[0]+",_f,_e,_d,"+Sommets[1]);
```

Certes, on eût pu faire autrement, avec des coordonnées polaires par exemple...

Tableau de points

Intersections, suivi des noms de deux objets, renvoie directement la liste de leurs points d'intersection. On peut la convertir en tableau en faisant comme ci-dessus.

Homothéties

Il n'y a pas d'outil homothétie dans CaRMetal, et donc pas de possibilité d'en faire une instruction JavaScript. Mais il y a une homothétie dans les macros, et on peut appeler une macro depuis un CarScript. N'importe quelle macro. Même une qu'on a créée soi-même, ou téléchargée etc. Il suffit de connaître son nom et ses objets initiaux. Par exemple, la macro "homothétie sans dialogue" se situe dans "Transformations" et s'appelle donc "Transformations/Homothétie sans dialogue" et s'exécute depuis un CarScript avec **ExecuteMacro**, suivi de son nom et d'une chaîne de caractères donnant la liste des objets initiaux (rapport, centre, point à transformer).

Comme le triangle de Sierpinski se construit par des homothéties de rapport $\frac{1}{2}$, on peut le construire avec le CarScript suivant, appliqué à une figure comprenant trois points A, B, C et un nombre $E1$ égal à 0,5:

```
m=MidPoint("A","B");
for(i=0;i<1000;i++){
t=Math.floor(Math.random()*3+1);
switch(t){
case(1):{
m=ExecuteMacro("Transformations/Homothétie sans dialogue","E1,A,_m");
break;
}
case(2):{
m=ExecuteMacro("Transformations/Homothétie sans dialogue","E1,B,_m");
break;
}
case(3):{
m=ExecuteMacro("Transformations/Homothétie sans dialogue","E1,C,_m");
}
}
}
```

C'est plus compliqué que le script fait avec les milieux. Mais on peut maintenant mettre $E1$ en mode curseur et voir comment la figure évolue si le rapport des homothéties est modifié (regarder par exemple avec un rapport de l'ordre de 0,6).

L'application de macros dans un CarScript est un sujet bien plus vaste que les homothéties: On peut aussi appliquer une inversion par rapport à un cercle, des rotations, construire des polygones réguliers, des barycentres, des courbes en coordonnées polaires, des nombres dérivés, calculer avec des nombres complexes ou des vecteurs, etc.

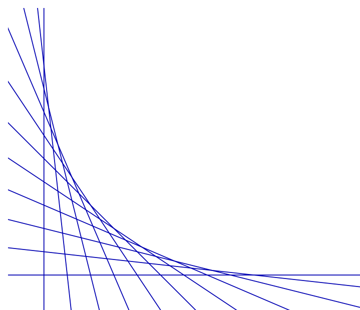
Lignes

Droites

Line, suivi entre parenthèses d'une proposition de nom (optionnelle) et des noms de deux points, crée la droite passant par ces deux points, à condition bien sûr qu'ils ne soient pas confondus! Cet outil permet de dessiner des faisceaux ou enveloppes de droites, comme le montre l'exemple suivant:

```
for (i=0; i<11; i++){ // on dessine 10 droites
    a=Point(0,5-i/2); // un point sur l'axe des y
    b=Point(i/2,0); // et un autre sur l'axe des x
    Hide(a);Hide(b);
    h=Line(a,b);
}
```

On obtient la figure suivante:



Note: CaRMetal possède un outil de dessin d'enveloppe de droites: C'est l'outil "trace automatique" pour lequel on sélectionne la droite au lieu[5] d'un point.

Parallèle

Parallel, suivi d'une proposition de nom (optionnelle) et des noms d'une droite et d'un point, crée la parallèle à la droite passant par le point. Ainsi, si A, B et C ont déjà été créés, le script

```
da=Line("B","C");SetHide(da,true);
db=Line("C","A");SetHide(db,true);
dc=Line("A","B");SetHide(dc,true);
a=Parallel(da,"A");SetHide(a,true);
b=Parallel(db,"B");SetHide(b,true);
c=Parallel(dc,"C");SetHide(c,true);
m=Intersection(b,c);
n=Intersection(c,a);
p=Intersection(a,b);
q=Polygon(m+","+n+","+p);
```

construit le triangle *antimédial* de ABC.

Perpendiculaire

Perpendicular, suivi d'une proposition de nom (optionnelle) et des noms d'une droite et d'un point, crée la perpendiculaire à la droite passant par le point. Par exemple, sur un fichier contenant trois points A, B et C,

```
d1=Line("A","B");SetHide(d1,true);
d2=Line("A","C");SetHide(d2,true);
h1=Perpendicular(d1,"C");SetHide(h1,true);
h2=Perpendicular(d2,"B");SetHide(h2,true);
h=Intersection(h1,h2);
```

[5] c'est le cas de le dire...

crée l'orthocentre de ABC.

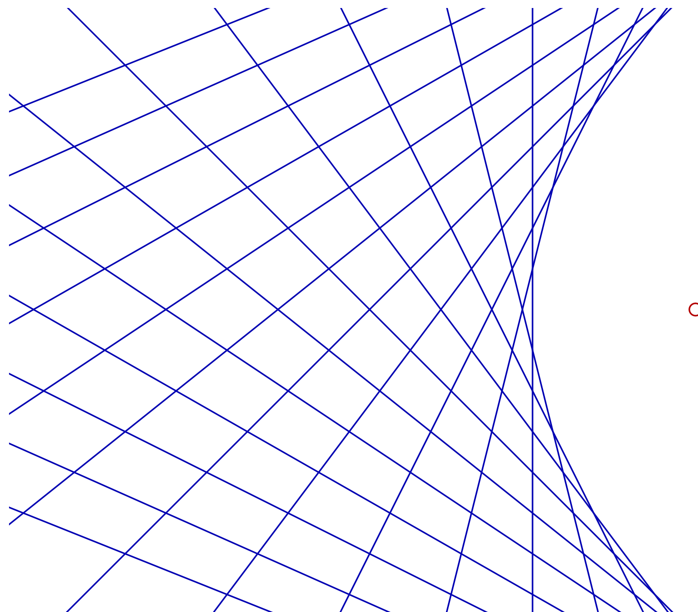
Médiatrice

PerpendicularBisector, suivi de deux noms de points, crée la médiatrice du segment dont les extrémités sont ces points.

Pour construire une parabole par foyer (de coordonnées (4;0)) et directrice (d'équation $x = 0$ sans tracer la directrice, le script suivant

```
a=Point(4,0);
for(i=-10;i<=10;i=i+1){
b=Point(0,i);SetHide(b,true);
d=PerpendicularBisector(a,b);
}
```

dessine la parabole comme enveloppe de droites:

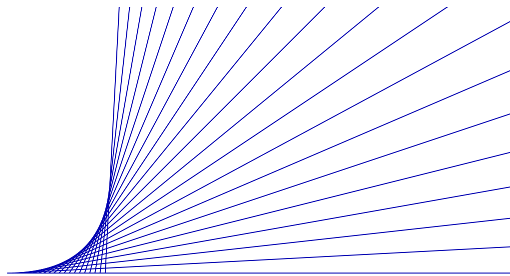


Demi-droites

Ray suivi d'une proposition de nom (optionnelle) et des noms de deux points, crée la demi-droite d'origine le premier point passant par le second point. Par exemple, le script suivant:

```
for (i=0; i<20; i++){
a=Point(i/10-2,0);SetHide(a,true);
b=Point(0,i/10);SetHide(b,true);
o=Ray(a,b);
}
```

donne la figure suivante:



AngleBisector, suivi de trois noms de points, crée la bissectrice intérieure de l'angle défini par ces points.

Par exemple, sur un fichier contenant un triangle ABC, le script suivant crée son point de Lemoine:

```
m1=MidPoint("A","C");SetHide(m1,true);
m2=MidPoint("A","B");SetHide(m2,true);
b1=AngleBisector("C","B","A");SetHide(b1,true);
b2=AngleBisector("B","C","A");SetHide(b2,true);
p1=Reflection(b1,m1);SetHide(p1,true);
p2=Reflection(b2,m2);SetHide(p2,true);
d1=Line("B",p1);SetHide(d1,true);
d2=Line("C",p2);SetHide(d2,true);
l=Intersection(d1,d2);
```

Angles

Angle suivi d'une proposition de nom (optionnelle) et des noms de trois points, crée l'angle défini par les trois points (le sommet étant celui du milieu) et si possible lui donne le nom proposé. Par exemple, le script ci-dessous (modifié à partir de celui sur les demi-droites)

```
o=Point(0,0);SetHide(o,true);
for (i=0; i<20; i++){
a=Point(i/10-2,0);SetHide(a,true);
b=Point(0,i/10);SetHide(b,true);
t=Angle(a,b,o);SetShowValue(t,false);
}
```

crée l'énigmatique figure ci-dessous:

FixedAngle suivi de deux noms de points et d'un nombre crée une demi-droite dont le sommet est le deuxième point entré, et faisant avec la demi-droite de même sommet passant par le premier point (non créée) un angle égal à la valeur numérique fournie (en degrés).
La combinaison de *FixedAngle* et de *Circle* permet facilement de créer un outil rotation. Sur une figure comprenant deux points A et B, l'image de B par la rotation de centre A et d'angle 30° se construit avec le script suivant:

```
c=Circle("A","B");
d=FixedAngle("B","A",30);
m=Intersection(c,d);
```

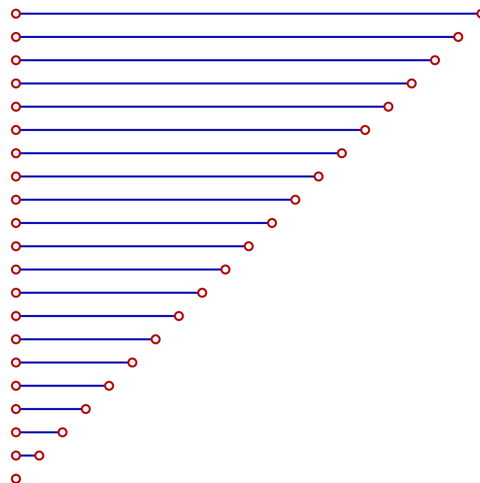
Segments

Segment, suivi d'une proposition de nom (optionnelle) et des noms de deux points, crée le segment entre ces deux points. Des exemples ont été vus à plusieurs reprises.
FixedSegment suivi d'une proposition de nom (optionnelle), du nom d'un point

et d'un nombre (sa longueur) construit un segment horizontal partant du point. Par exemple, le script suivant dessine une lyre:

```
for(n=0;n<=20;n++){
a=Point(0,n/10);
s=FixedSegment(a,n/10);
}
```

Voici l'objet:

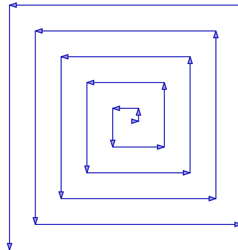


Vecteurs

Vector, suivi d'une proposition de nom (optionnelle) et des noms de deux points, crée un segment fléché entre ces deux points. Par exemple le script suivant

```
x=0;y=0;
a=Point(x,y);SetHide(a,true);
for (n=0; n<20; n++){
x=x+n*Math.cos(n*Math.PI/2);
y=y+n*Math.sin(n*Math.PI/2);
b=Point(x,y);SetHide(b,true);
v=Vector(a,b);
a=b;
}
```

Crée la spirale que voici:



Polygones

Polygon, suivi d'une proposition de nom et d'une liste de nom de sommets, entre guillemets, crée un polygone.
Par exemple, si A, B et C sont déjà créés,

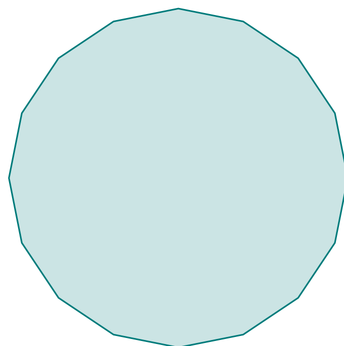
```
Polygon("","A,B,C");
```

Crée le triangle ABC.
Pour changer de l'heptadécagone régulier, on va construire l'***hexadécagone*** à 16 côtés. C'est le même principe que pour l'heptadécagone, mais au lieu de construire l'un après l'autre des segments, on construit ***au fur et à mesure*** la liste des sommets, qui est une chaîne de caractères: Les noms des sommets

séparés par des virgules. Un bon moyen de voir comment ça marche est d'insérer dans le script ci-dessous un "Println(s)" dans la boucle, ce qui permet de voir comment la liste des sommets de l'hexadécagone se construit:

```
a=Point(2,0); // premier sommet
s=a // début du polygone
SetHide(a,true);
for (i=1; i<=15; i++){ // hexadécagone...
x=2*Math.cos(2*Math.PI*i/16); // abscisse
y=2*Math.sin(2*Math.PI*i/16); // ordonnée
b=Point(x,y); // le sommet suivant
SetHide(b,true);
s=s+","+b; // on rajoute b à la liste
a=b // on change de sommet
}
p=Polygon(s);
SetFilled(p,true);
```

On obtient la figure ci-dessous, le polygone ayant été rempli pour montrer la différence avec la collection de segments de l'heptadécagone:



Cercles

passant par un point

Circle est suivi, outre la classique proposition de nom, de deux arguments, qui sont tous les deux des noms de points, et crée le cercle centré sur le premier point passant par le deuxième point. Pour obtenir le cercle circonscrit à un triangle *ABC* déjà créé, on entre le script suivant:

```
m=MidPoint("A","B");
n=MidPoint("A","C");
a=Line("A","B");
b=Line("A","C");
c=Perpendicular(a,m);
d=Perpendicular(b,n);
o=Intersection(c,d);
u=Circle(o,"C");
```

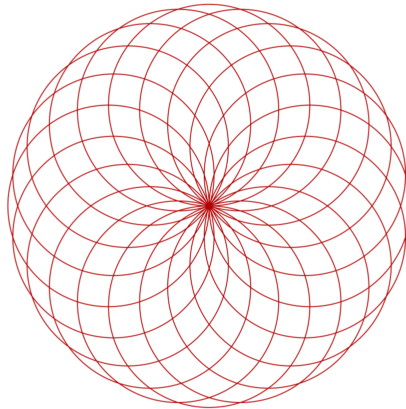
qui construit successivement les milieus, les côtés, les médiatrices, leur intersection et le cercle lui-même.

Cercles de rayon donné

Fixedcircle suivi d'une proposition de nom (optionnelle) et du nom d'un point et d'un nombre, crée le cercle de centre ce point ayant pour rayon ce nombre (à condition qu'il soit positif).
Par exemple, on peut faire un dessin à la LOGO avec ce script:

```
for (i=0; i<20; i++){
    x=2*Math.cos(Math.PI*i/10); // abscisse du centre
    y=2*Math.sin(Math.PI*i/10); // ordonnée du centre
    p=Point(x,y);Hide(p);
    j=FixedCircle(p,2);
}
```

Le résultat est ceci:

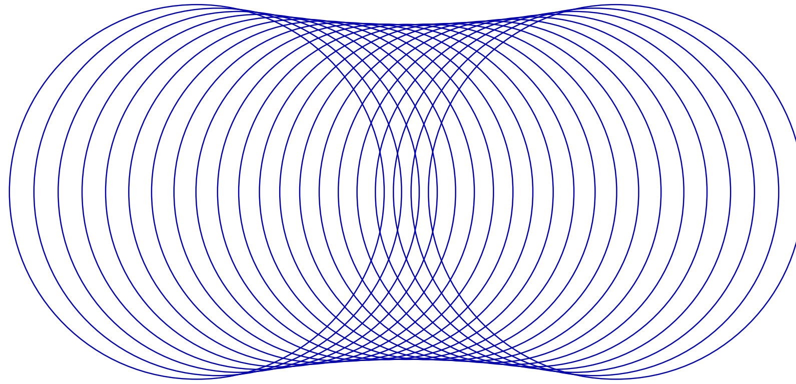


report d'une distance

Circle3, suivi des noms de trois points, crée le cercle centré sur le premier point dont le rayon est égal à la distance entre les deux autres points. Cette instruction permet donc de simuler le fonctionnement d'un compas (on reporte à partir du premier point, la distance entre les deux autres).
Un exemple d'enveloppe de cercles:

```
a=Point(4,0);SetHide(a,true);
for(n=-10;n<=10;n=n+1){
b=Point(n/2,0);SetHide(b,true);
c=Point(0,n/5);SetHide(c,true);
g=Circle3(b,a,c);
}
```

On obtient ceci:

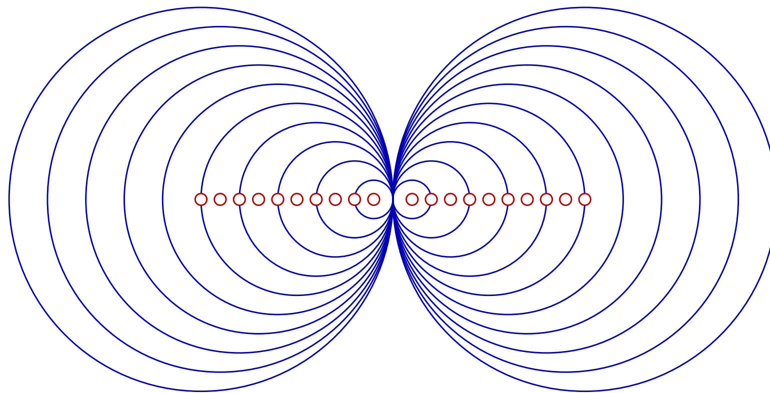


Cercle circonscrit

Circle3pts suivi des noms de trois points, crée le cercle circonscrit au triangle passant par ces points. Le centre du cercle circonscrit est construit avec comme le montre le script suivant, dessinant un faisceau de cercles tangents:

```
for(n=-10;n<=10;n=n+1){
a=Point(n,-n);SetHide(a,true);
b=Point(n,n);SetHide(b,true);
c=Point(2*n,0);SetHide(c,true);
d=Circle3pts(a,b,c);
}
```

Les centres des cercles sont alignés:



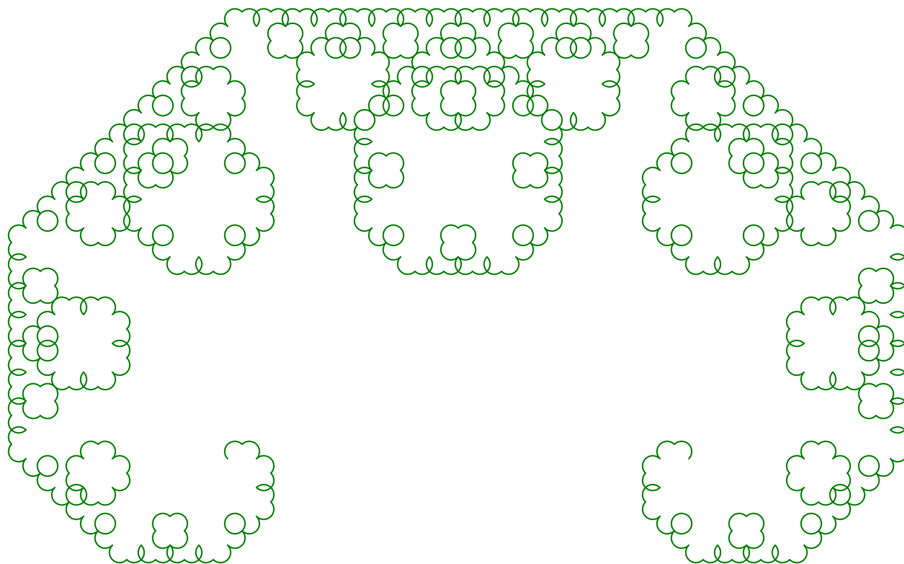
Arc de cercle

Arc3pts, suivi des noms de trois points, crée l'arc de cercle joignant le premier point au troisième et passant par le second. L'application suggérée par l'icône de CaRMetal est un pavage hyperbolique, mais ici on va faire un peu de récursivité, avec une figure à la Von Koch sauf qu'au lieu de segments, on met des demi-cercles:

```
function brocoli(a,b,n){
if(n==10){
var d1=PerpendicularBisector(a,b);SetHide(d1,true);
var d2=FixedAngle(b,a,45);SetHide(d2,true);
var m=Intersection(d1,d2);SetHide(m,true);
l=Arc3pts(a,m,b);
} else {
var d1=PerpendicularBisector(a,b);SetHide(d1,true);
var d2=FixedAngle(b,a,45);SetHide(d2,true);
var m=Intersection(d1,d2);SetHide(m,true);
brocoli(a,m,n+1);
brocoli(m,b,n+1)
}
}
a=Point(-3,-3);SetHide(a,true);
b=Point(3,-3);SetHide(b,true);
```

```
brocoli(a,b,1);
```

La fonction *brocoli* (son nom suggère la figure voulue, ce n'est pas exactement le résultat obtenu) est récursive: À deux points a et b elle associe le troisième sommet d'un triangle rectangle isocèle dont ab est l'hypoténuse. Si n=10 (maximum) la fonction se contente de tracer le demi-cercle défini par ces trois points, sinon elle s'appelle avec le rang suivant, sur chaque côté de l'angle droit du triangle.
Essayer avec des valeurs plus grandes que 1 car avec 1 c'est très long, et ça donne ça:

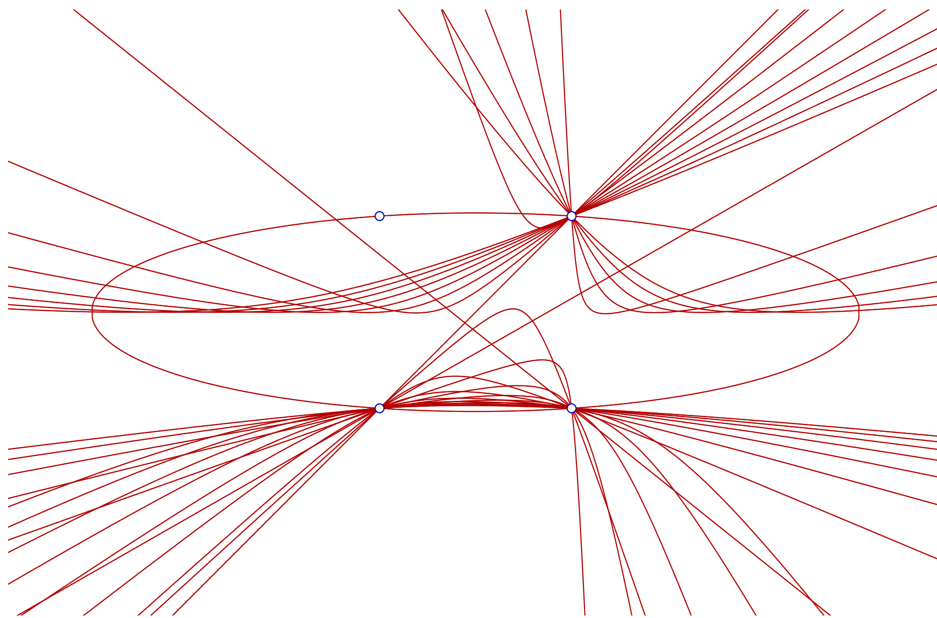


Coniques

Quadric suivi d'une proposition de nom (optionnelle) et de 5 noms de points, crée la conique passant par ces 5 points. Cela permet de tracer des faisceaux de coniques.
Exemple de faisceau:

```
a=Point(-2,-2);b=Point(2,-2);c=Point(-2,2);d=Point(2,2);
for (i=-8; i<8; i=i+0.5){
    e=Point(i,0);Hide(e);
    c=Quadric(a,b,c,d,e);
}
```

On obtient le faisceau de coniques suivant:



Fonctions

Représentations graphiques

CartesianFunction suivi d'une proposition de nom (optionnelle) et de trois paramètres, deux nombres et une expression, représente l'expression sur l'intervalle formé par les deux nombres. Par exemple, la représentation graphique de $x\,\mathrm{e}^{-x}$ sur $[0;8]$ s'obtient en entrant

```
CartesianFunction(0,8,"x*exp(-x)");
```

Courbes paramétrées

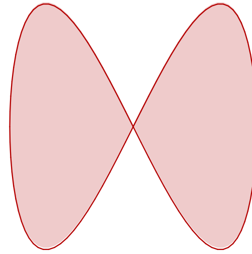
ParametricFunction suivi d'une proposition de nom (optionnelle) et de quatre paramètres, deux nombres et deux expressions en t, représente les deux expressions sur l'intervalle formé par les deux nombres, la première comme abscisses et la deuxième comme ordonnées. Par exemple, la lemniscate de Bernoulli $x = 2cos(t)$, $y = 2sin(2t)$ sur $[0;360°]$ s'obtient en entrant

```
ParametricFunction(0,360,"2*cos(t)","2*sin(2*t)");
```

(rcos et rsin à la place de cos et sin si on est en radians)
En modifiant légèrement le code:

```
a=ParametricFunction(0,360,"2*cos(t)","2*sin(2*t)");
SetFilled(a,true);
```

on obtient la figure suivante:

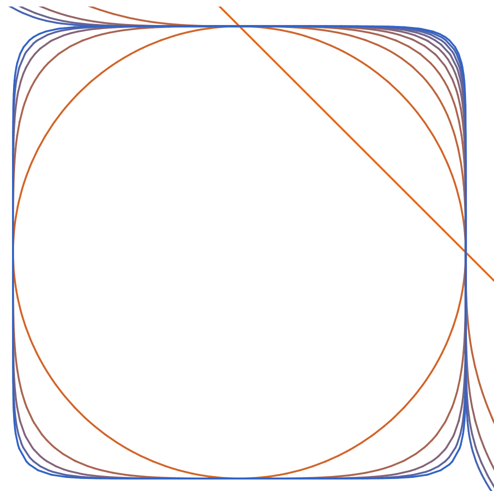


Lignes de niveau

ImplicitPlot, suivi du nom de la courbe (optionnel), et de l'équation cartésienne de la courbe, crée une courbe, ensemble des zéros de la fonction (de x et y) donnée en argument à **ImplicitPlot**. La fonction de x et y doit être fournie entre guillemets puisque c'est une chaîne de caractères interprétée par CaRMetal. Par exemple, pour représenter les lignes de niveau de $x^n + y^n - 1$ pour différentes valeurs de n, on peut faire ceci:

```
for(n=1;n<=10;n++){
c=ImplicitPlot("x^"+n+"+y^"+n+"-1");
SetRGBColor(c,250-20*n,100,20*n);
}
```

on obtient la figure suivante:



Nombres

Expression suivi de l'optionnelle proposition de nom, de l'expression à créer, puis de deux coordonnées (où afficher le nombre), crée un nombre et l'affiche. Par exemple, si A et B sont deux points déjà créés,

```
Expression("d(A,B)",-3,2)
```

affiche la distance entre A et B au point de coordonnées (-3;2).

Pour créer un tableau de valeurs pour la fonction $x \mapsto x - x^3$ sur $[0;1]$, on peut essayer le code suivant:

```
function f(x){
    return(x-x*x*x);
}
p=Point(-4.1,3.3);Hide(p);
q=Point(-1.5,3.3);Hide(q);
s=Segment(p,q);
for (i=0; i<=1; i=i+0.1){
    a=Expression(i,-4,3-i*4);
```

```
    b=Expression(f(i),-3,3-i*4);
    p=Point(-4.1,2.9-i*4);Hide(p);
    q=Point(-1.5,2.9-i*4);Hide(q);
    s=Segment(p,q);
}
p=Point(-4.1,3.3);Hide(p);
q=Point(-4.1,-1.1);Hide(q);
s=Segment(p,q);
p=Point(-3.2,3.3);Hide(p);
q=Point(-3.2,-1.1);Hide(q);
s=Segment(p,q);
p=Point(-1.5,3.3);Hide(p);
q=Point(-1.5,-1.1);Hide(q);
s=Segment(p,q);
```

ce qui donne le tableau que voici:

0	0
0,1	0,099
0,2	0,192
0,3	0,273
0,4	0,336
0,5	0,375
0,6	0,384
0,7	0,357
0,8	0,288
0,9	0,171
1	???

On constate que la fonction n'est pas définie en $x = 1$.
Plus simplement on pouvait créer le tableau dans une console, avec **Println**:

```
function f(x){
    return(x-x*x*x);
}
for (i=0; i<=1; i=i+0.1){
    Println(Math.floor(10*i)/10+"        "+f(i));
}
```

Pas terrible, mais on voit pourquoi le tableau précédent n'affichait pas $f(1)$: Ce nombre n'étant pas rigoureusement nul, son affichage "2.220446049250313e-16" est trop grand...

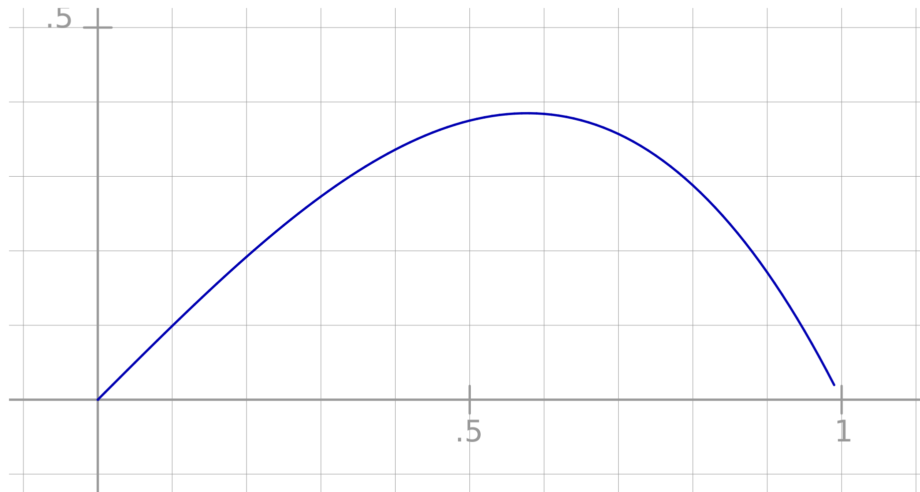
La même technique, au lieu de tableau, peut aussi servir à fabriquer des représentations graphiques:

```
function f(x){
    return(x-x*x*x);
}
for (i=0; i<=1; i=i+0.1){
    a=Point(i,f(i));
}
```

dans un premier temps, pour placer des points (11) sur le graphique, puis

```
function f(x){
    return(x-x*x*x);
}
a=Point(0,0);Hide(a);
for (i=0; i<=1; i=i+0.01){
    b=Point(i,f(i));Hide(b);
    s=Segment(a,b);
    a=b;
}
```

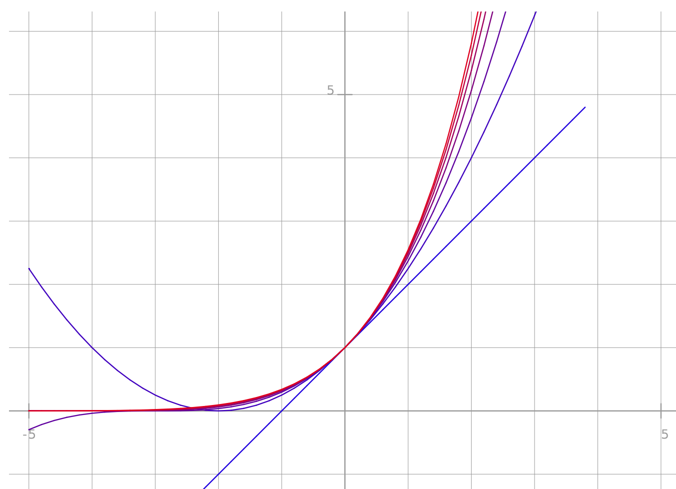
pour faire une "courbe" (ici avec 100 points):



Quel est l'avantage de cette méthode sur **CartesianFunction**? Essentiellement que la fonction définie en javascript peut contenir des variables (puisqu'elle n'est pas entre guillemets). Ce qui permet de représenter des familles de fonctions, comme les célèbres $\left(1+\frac{x}{n}\right)^n$:

```
function f(x,n){
    return(Math.pow(1+x/n,n));
}
for (n=1; n<=7; n++){ // boucle sur l'entier
    a=Point(-5,f(-5,n));Hide(a);
    for (x=-5; x<4; x=x+0.2){ // boucle sur x
        b=Point(x,f(x,n));Hide(b);
        s=Segment(a,b);
        SetRGBColor(s,n*32,0,255-n*32);
        a=b;
    }
}
```

qui donne la figure suivante:

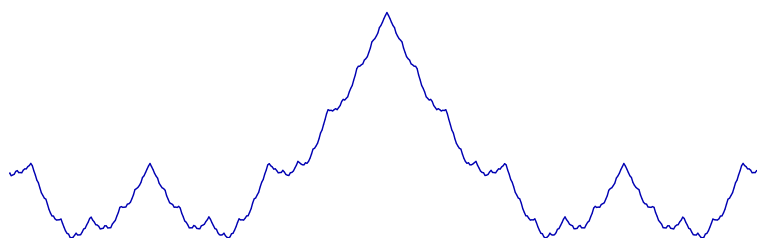


Remarque: Bien que les couleurs permettent de distinguer les différentes valeurs de n, une courbe unique mais dynamique comme on l'aurait obtenue en créant une représentation graphique de fonction, eût sans doute été préférable (et n'eût pas nécéssité de javascript): On crée un curseur n et la fonction sera "(1+x/n)∧n".

Un autre avantage des fonctions définies avec un CarScript est qu'elles peuvent être très spéciales, par exemple définies par des boucles: Par exemple la fonction de Weierstrass[6] définie par la série $f(x) = \sum_{n=0}^{\infty} \frac{\cos 2^n x}{2^n}$ et approchée par $f(x) \simeq \sum_{n=0}^{8} \frac{\cos 2^n x}{2^n}$:

```
function W(x){ // définition de W(x)
    s=0; // somme initialement nulle
    p=1; // facteur initialement un
    for (n=0; n<8; n++){ // pour chaque terme
        s=s+Math.cos(p*x)/p; //on ajoute à la somme le terme courant
        p=2*p; // on double le facteur
    }
    return(s); // la somme est W(x)
}
a=Point(-5,W(-5));Hide(a); // départ
for (x=-5; x<5; x=x+0.02){ // on trace de -5 à 5
    b=Point(x,W(x));Hide(b); // deuxième extrémité
    s=Segment(a,b); // un segment
    a=b; // b devient l'origine du prochain segment
}
```

La représentation graphique est la suivante:



On voit bien qu'elle est continue sans dérivée[7]

[6] à ne pas confondre avec la fonction $\mathcal{P}$ elliptique

[7] mais ce n'est pas la première, des exemples antérieurs ayant été créés par Bolzano et Riemann

Suites

Puisque la représentation graphique d'une suite est un nuage de points, CaRMetal est parfaitement approprié pour étudier une suite. Par exemple, pour le célèbre problème de Collatz, concernant la suite $(u_n)_{n\in\mathbb{N}}$ définie récursivement par

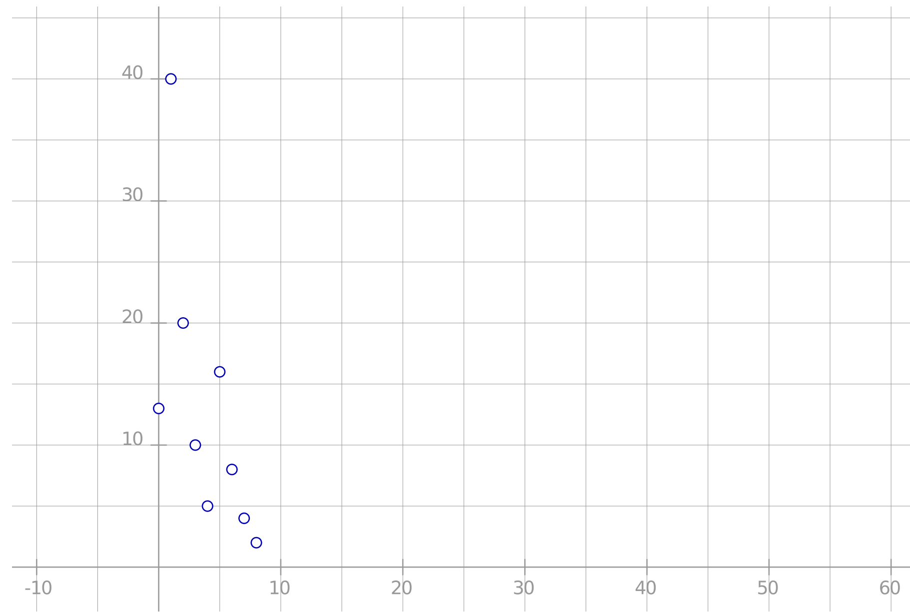
$$\begin{cases} \text{Si } u_n \equiv 0\ [2], & u_{n+1} = \dfrac{u_n}{2} \\ \text{si } u_n \equiv 1\ [2], & u_{n+1} = 3u_n + 1 \end{cases}$$

On calcule et représente les termes successifs de la suite avec ce petit CarScript:

```
u=Math.floor(Math.random()*40); // premier terme
n=0; // premier indice
while (u!=1){ // on s'arrête quand u=1
    a=Point(n,u); // Placer un point
    t=u-Math.floor(u/2)*2; // calcul de u modulo 2
    if (t==0){ //                      si u est pair
        u=Math.round(u/2); // on le divise par 2
    } else { //                            sinon
        u=3*u+1; // 3u+1
    }
    n++    //                    on passe au suivant
}
```

On obtient ce genre de graphique[8]:

[8] La conjecture de ***Collatz*** dit que ce script ne plantera jamais. En théorie ça pourrait arriver puisque personne n'a montré ladite conjecture...


40
30
20
10
-10
10
20
30
40
50
60

Espace

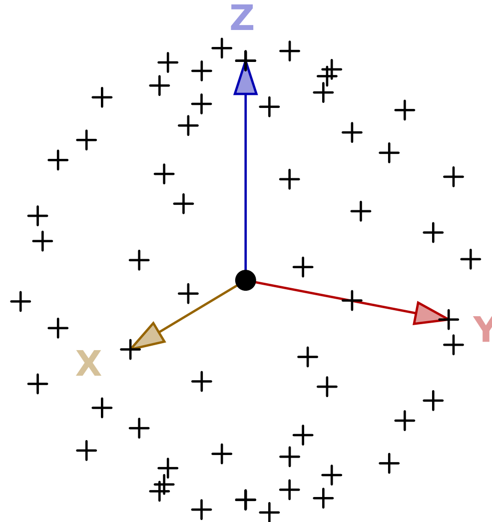
On va voir comment on peut dessiner une sphère sous CaRMetal et l'améliorer un peu. D'autres applications sont envisageables comme le déplacement aléatoire d'un robot dans l'espace, d'autres solides comme des tores, des figures articulables...

Sphère pointilliste

Les CarScripts suivants doivent être lancés dans une figure de l'espace, qu'on obtient en cliquant sur "nouvelle figure 3D". Ce faisant, la figure n'est plus vierge puisqu'elle comprend un repère tridimensionnel dont l'origine s'appelle "O" et les extrémités des vecteurs sont respectivement "X", "Y" et "Z". Ce sont les coordonnées de ces points et les coordonnées x, y et z d'un point M de l'espace qui servent à calculer les coordonnées du projeté de M sur le plan.
Dans le script suivant, on calcule les coordonnées de points de la sphère à partir de leur longitude (indice i) et de leur latitude (indice j). Puis on crée les projetés sur le plan (en traitant une chaîne de caractères) et on a une sphère.

```
var N=8;
for(i=0;i<N;i++){
    for(j=0;j<=N;j++){
        x="cos(_i*360/_N)*sin(_j*180/_N)";
        y="sin(_i*360/_N)*sin(_j*180/_N)";
        z="cos(_j*180/_N)";
        p=Point("x(O)+("+x+")*(x(X)-x(O))+("+y+")*(x(Y)-x(O))+("+z+")*(x(Z)-x(O))",
"y(O)+("+x+")*(y(X)-y(O))+("+y+")*(y(Y)-y(O))+("+z+")*(y(Z)-y(O))");
    }
}
```

Pour voir que les points sont cosphériques il faut faire tourner la figure.



Sphère polyédrale

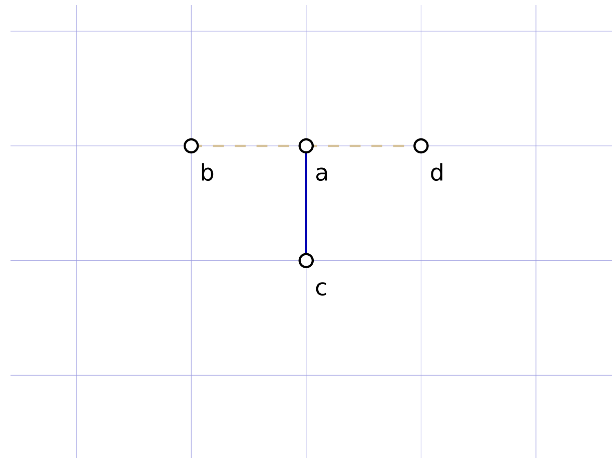
Pour mieux représenter la sphère, au lieu des sommets, il vaut mieux représenter les arêtes (l'approximation de cette sphère est un polyèdre). Ce sont des segments, qui apparaissent en pointillés lorsque certains angles sont rentrants (faces cachées) et le tracé de chacun fait intervenir 4 sommets (les extrémités du segment et deux sommets auxiliaires). Il vaut donc mieux placer les noms de tous les sommets dans un tableau bidimensionnel où ils seront consultés ultérieurement. La première partie du script ressemble donc à la précédente, sauf que les points sont cachés, et stockés dans la variable "Pts" qui est un tableau à deux indices:

```
var N=12;
var Pts=new Array();
for(i=0;i<N;i++){
    Pts[i]=new Array();
        for(j=0;j<=N;j++){
            x="cos(_i*360/_N)*sin(_j*180/_N)";
            y="sin(_i*360/_N)*sin(_j*180/_N)";
```

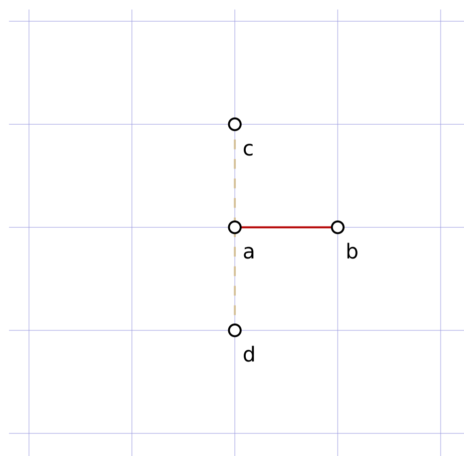
```
            z="cos(_j*180/_N)";
  Pts[i][j]=Point("x(O)+("+x+")*(x(X)-x(O))+("+y+")*(x(Y)-x(O))+("+z+")*(x(Z)-x(O))",
"y(O)+("+x+")*(y(X)-y(O))+("+y+")*(y(Y)-y(O))+("+z+")*(y(Z)-y(O))");
            SetHide(Pts[i][j],true);
    }
}
for(i=0;i<N;i++){
    for(j=1;j<N;j++){
        a=Pts[i][j];
        if(i>0){
            b=Pts[i-1][j];
        } else {
            b=Pts[N-1][j];
        }
        if(i<N-1){
            d=Pts[i+1][j];
        } else {
            d=Pts[0][j];
        }
        c=Pts[i][j+1];
        s=Segment(a,c);
        SetColor(s,"blue");//les méridiens en bleu
        Conditional(s,"thin","(a(_b,_a,_c)>180)&&(a(_c,_a,_d)>180)");
    }
}
for(i=0;i<N;i++){
    for(j=1;j<N;j++){
        a=Pts[i][j];
        if(i<N-1){
            b=Pts[i+1][j];
        } else {
            b=Pts[0][j];
        }
        c=Pts[i][j-1];
        d=Pts[i][j+1];
        s=Segment(a,b);
        SetColor(s,"green");//les parallèles en vert
        Conditional(s,"thin","(a(_d,_a,_b)>180)&&(a(_b,_a,_c)>180)");
    }
```

```
}
```

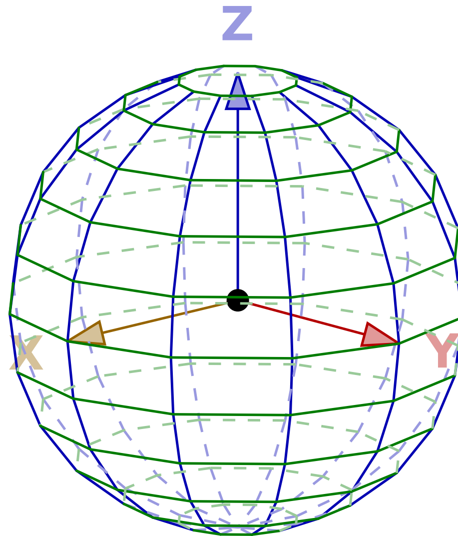
Ensuite on parcourt tout le tableau dans une double boucle, en donnant les noms a, b, c et d aux 4 sommets suivants:



Alors $a = Pts[i][j]$, $b = Pts[i-1][j]$, $c = Pts[i][j+1]$ et $d = Pts[i+1][j]$. L'arête joignant a à c est une partie du méridien, et est en bleu.
Pour peu que l'un des angles $\widehat{bac}$ et $\widehat{cad}$ est saillant, l'arête est "devant", donc en trait plein. Sinon elle est en pointillés ("thin").
Même principe pour les segments rouges qui font partie des parallèles, avec une nouvelle boucle sur i et j et un renommage des points b, c et d:



Le pôle Nord pose un problème car les angles n'étant pas définis, ne peuvent être supérieurs à 180°:



La réalisation de modèles polyédraux de tores, cyclides de Dupin, bouteilles de Klein et toutes autres surfaces paramétrées par des longitude et latitude (par exemple les représentations de fonctions de deux variables) est laissée en exercice aux rares lecteurs qui ont réussi à arriver jusqu'ici !
Idem pour l'extension aux anaglyphes: Il suffit de faire le tout en double exemplaire, en additionnant quelques degrés à la longitude de la version rouge (celle qui dépend de i) et en les soustrayant à la longitude de la partie cyan.

Géométrie hyperbolique

Les instructions JavaScript suivantes sont censées être utilisées dans un plan hyperbolique, qu'on obtient en cliquant sur "Nouveau disque de Poincaré" dans le menu "fichier". En effet la plupart d'entre elles utilisent le disque de Poincaré, qui est un élément de la figure.

Points hyperboliques

Point hyperbolique

DPPoint() crée un point placé au hasard dans le disque de Poincaré. On ne lui donne donc pas de coordonnées puisqu'elles sont déterminées au hasard (d'ailleurs c'est quoi, les coordonnées d'un point, en géométrie hyperbolique?). Par contre on peut ne rien mettre entre parenthèses, auquel cas CaRMetal se chargera de fournir un nom au point hyperbolique créé. De toute manière, le nom est retourné par cette instruction. Ainsi

```
DPPoint("A");
```

Crée un point hyperbolique de nom "A".

```
a=DPPoint("A");
```

en fait de même, mais stocke le nom "A" dans la variable a.

```
a=DPPoint();
```

stocke le nom du point hyperbolique dans a, mais ce nom n'est plus forcément "A". Il peut très bien être égal à "P1" selon le réglage de CaRMetal. Le CarScript ci-dessus peut être utilisé dans une figure euclidienne, la différence étant que le point aléatoire n'est plus lié au disque de Poincaré, celui-ci n'existant pas dans la figure euclidienne. Un nuage de points peut donc facilement être créé par

```
for(n=0;n<100;n++){
    a=DPPoint();
    SetPointType(a,"point");
}
```

Milieu hyperbolique

DPMidPoint, suivi des noms de deux points hyperboliques, renvoie leur milieu. Le triangle de Sierpinski hyperbolique se crée avec ce script:

```
a=DPPoint();
b=DPPoint();
c=DPPoint();
m=DPPoint();
SetHide(m,true);
for(i=0;i<200;i++){
de=Math.ceil(Math.random()*3);
switch (de){
case 1 : {
m=DPMidPoint(m,a);
```

```
break;
}
case 2 : {
m=DPMidPoint(m,b);
break;
}
case 3 : {
m=DPMidPoint(m,c);
break;
}
}
SetPointType(m,"point");
}
```

Droites hyperboliques

Toutes les instructions JavaScript ci-dessous renvoient le nom de l'objet (droite, demi-droite ou segment hyperbolique) créé, et peuvent avoir un paramètre supplémentaire placé avant tous les autres: Le nom proposé pour la droite.

Droite hyperbolique

DPLine, suivi du nom de deux points hyperboliques, crée la droite hyperbolique passant par ces deux points. Si on veut les points idéaux de cette droite (ou points à l'infini), on peut utiliser *Intersection2* ou *Intersections* qui permet d'avoir l'intersection de la droite hyperbolique et de l'horizon, qui s'appelle "Hz":

```
a=DPPoint();
b=DPPoint();
d=DPLine(a,b);
PI=Intersection2(d,"Hz");
```

Demi-droite hyperbolique

DPRay, suivi des noms de deux points hyperboliques, crée la demi-droite hyperbolique de sommet le premier point, passant par le deuxième point. Pour avoir son point idéal, on peut faire comme ci-dessus:

```
a=DPPoint();
b=DPPoint();
d=DPRay(a,b);
i=Intersection(d,"Hz");
```

Perpendiculaire hyperbolique

DPPerpendicular, suivi du nom d'une droite hyperbolique et du nom d'un point hyperbolique, crée la perpendiculaire hyperbolique à la droite hyperbolique passant par le point hyperbolique. Un exemple est visible ci-dessous (cercle inscrit).

Médiatrice hyperbolique

DPPerpendicularBisector, suivi des noms de deux points, crée la médiatrice hyperbolique du segment hyperbolique formé par ces deux points, autrement dit le lieu des points du disque de Poincaré qui sont équidistants (au sens de la distance hyperbolique) des deux points.
Pour construire le cercle circonscrit à un triangle (lorsque celui-ci existe), on peut donc utiliser le CaRSCript suivant:

```
a=DPPoint();
b=DPPoint();
c=DPPoint();
d=DPPerpendicularBisector(a,b);
e=DPPerpendicularBisector(a,c);
```

```
o=Intersection(d,e);
SetHide("_d,_e,_o",true);
c=DPCircle(o,a);
```

Segment hyperbolique

DPSegment, suivi des noms de deux points, crée le segment hyperbolique joignant ces deux points. Un exemple est visible ci-dessous (triangle et cercle inscrit).

Symétries hyperboliques

Symétrie axiale

DPReflexion, suivi du nom d'une droite hyperbolique et de celui d'un point hyperbolique, crée le symétrique du point par rapport à la droite, et renvoie son nom.

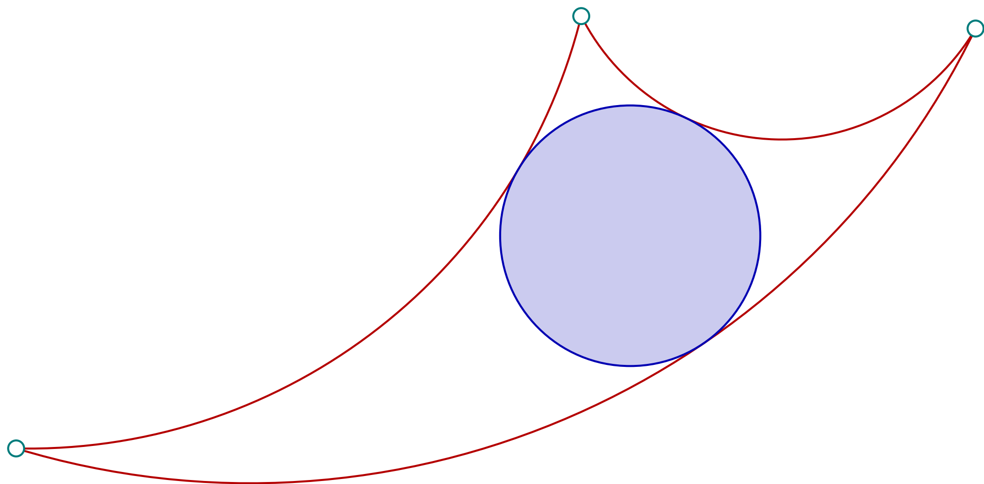
Bissectrice

DPAngleBisector, suivi des noms de trois points, renvoie la bissectrice (une droite hyperbolique) de l'angle hyperbolique formé par ces trois points. On peut s'en servir pour construire le cercle inscrit à un triangle hyperbolique:

```
a=DPPoint();
b=DPPoint();
c=DPPoint();
DPSegment(a,b);
DPSegment(a,c);
DPSegment(b,c);
s=DPAngleBisector(b,a,c);
```

```
SetHide(s,true);
t=DPAngleBisector(a,b,c);
SetHide(t,true);
i=Intersection(s,t);
u=DPLine(a,b);
SetHide(u,true);
v=DPPerpendicular(u,i);
SetHide(v,true);
g=Intersection(u,v);
SetHide(g,true);
e=DPCircle(i,g);
```

On a là la base de Sangakus hyperboliques:



Perpendiculaire commune

Si deux droites hyperboliques ne sont ni parallèles, ni sécantes, il existe une et une seule droite hyperbolique qui les coupe toutes les deux à angle droit. **DP-CommonPerpendicular** crée cette droite. Les deux paramètres qu'il accepte sont les noms des deux droites.

Symétrie centrale

DPSymmetry, suivi des noms de deux points, crée le symétrique hyperbolique du second par rapport au premier.

Cercle hyperbolique

DPCircle, suivi des noms de deux points, crée le cercle hyperbolique de centre le premier point, passant par le second point. Des exemples ont été vus précédemment (cercles circonscrit et inscrit).
Si on veut construire un horocycle (c'est-à-dire un "cercle" dont le centre est à l'infini), il suffit de créer un point idéal (un des deux points idéaux d'une droite, ou le point idéal d'une demi-droite), et d'utiliser *DPCircle* en choisissant ce point idéal comme centre:

```
a=DPPoint();
b=DPPoint();
d=DPRay(a,b);
SetHide(d,true);
i=Intersections(d,"Hz")[1];
c=DPCircle(i,a);
```

Le coin des hackers

Ce chapitre est fortement déconseillé en première lecture. En effet, il ouvre des portes vers des univers vastes et largement inexplorés, mais accessibles depuis les CaRScripts. Enfin pas si accessibles que ça, puisque pour exploiter ces nouvelles possibilités, il faudrait idéalement lire une quantité de documentation largement supérieure au présent pdf. C'est donc essentiellement par souci d'exhaustivité que ce chapitre a été rédigé. Et un peu aussi par plaisir, car, il faut l'avouer, aller regarder sous le capot est aussi jouissif que compliqué!

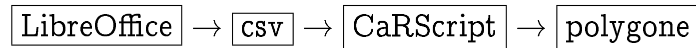
Chargement de fichiers

Load suivi du nom d'un fichier placé dans le dossier des CaRScripts, renvoie le contenu de celui-ci sous forme d'une chaîne de caractères. Ce qui permet de faire entrer des choses de l'extérieur dans le CaRScript. Ces choses peuvent être des données, ou un programme JavaScript... On va montrer ci-dessous un exemple de chaque sorte.

Fichier de données

On constate que CaRMetal n'a (pour l'instant) pas de fonctions de Bessel. Et sait-on jamais, un jour on risque d'avoir besoin de la représentation graphique de la fonction de Bessel J_1 du premier ordre. Qui peut prédire d'avance de quoi on aura besoin après tout? Or il se trouve que les tableurs classiques comme par exemple ***LibreOffice*** peuvent calculer les fonctions de Bessel comme J_1. Et exporter des tableaux. Alors il suffit d'utiliser *Load* pour charger le tableau dans

le CaRScript et construire un polygone approchant la fonction tabulée, selon le schéma suivant:



Sous tableur

Une fois *LibreOffice* démarré, on remplit la colonne A avec des nombres en progression arithmétique (ci-dessous de -5 à 5 par valeurs entières pour ne pas avoir un fichier d'exemple trop gros), puis dans la cellule B1, on entre la formule =BESSELJ(A1;1) qu'on recopie vers le bas:

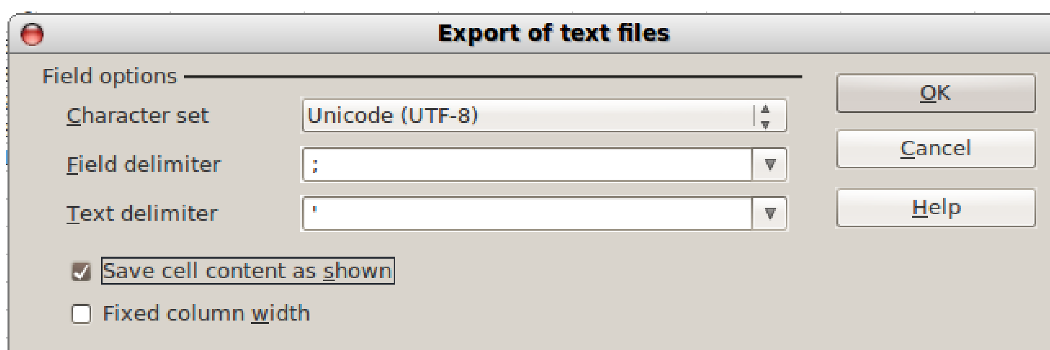
B1 =BESSELJ(A1;1)

	A	B	C	D
1	-5	0,327579138		
2	-4	0,066043328		
3	-3	-0,339058959		
4	-2	-0,576724808		
5	-1	-0,440050586		
6	0	0		
7	1	0,440050586		
8	2	0,576724808		
9	3	0,339058959		
10	4	-0,066043328		
11	5	-0,327579138		

Pour utiliser ces données, on va les exporter au format *csv* (plus ou moins universel) ce qui se fait juste en enregistrant le fichier au format *csv* qu'on choisit dans le menu d'export de *LibreOffice*:

Export

En *enregistrant sous*, on choisit l'extension *csv* et on confirme que c'est bien à ce format qu'on veut l'enregistrer (*LibreOffice* proteste un peu mais cède intelligemment). On a alors la boîte de dialogue modale suivante:



Il est recommandé de choisir le format Unicode UTF-8, parce que celui-ci est universel et que tôt ou tard, si on en choisit un autre, on aura des problèmes à l'ouverture du fichier. Ou quelqu'un d'autre en aura, ce qui somme toute est tout aussi grave! *LibreOffice* propose d'utiliser la virgule comme séparateur des données, et le guillemet pour encadrer les chaînes de caractères. Comme la virgule est déjà utilisée pour les nombres décimaux, on va lui préférer le point-virgule. Et en remplaçant le guillemet par une apostrophe, on aura des nombres en colonne B, et non des chaînes de caractères.
Le fichier exporté contient alors ceci:

```
-5;0,327579137591388
-4;0,0660433280233065
-3;-0,339058958526515
-2;-0,576724807757626
-1;-0,440050585745083
0;0
1;0,440050585745083
2;0,576724807757626
3;0,339058958526515
4;-0,0660433280233065
5;-0,327579137591388
```

On a des nombres entiers et des réels, séparés par des points-virgules. Dans la suite, on suppose que le fichier exporté par le tableur s'appelle *bessel.csv* et qu'il se trouve dans le répertoire des scripts de CaRMetal.

Import

Pour charger le fichier dans une chaîne de caractère qui s'appelle *fichier*, il suffit de taper `fichier=Load("bessel.csv");`
et pour le découper ligne par ligne, il suffit de taper

```
fichier=Load("bessel.csv");
donnees=fichier.split("\n");
```

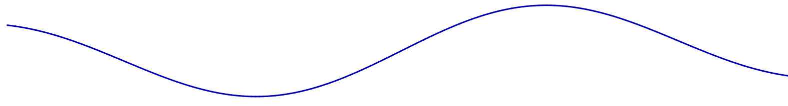
La variable *donnees* contient un tableau dont chaque entrée est une ligne du tableur (une chaîne de caractères avec un point-virgule au milieu).

Représentation graphique

Pour représenter le polygone, il suffit de recommencer à couper les entrées du tableau *donnees*, avec le point-virgule comme séparateur cette fois-ci. Puis de remplacer la virgule par un point, et d'en faire les coordonnées d'un point. Ensuite on fait comme précédemment, en cachant les points et en les joignant par des segments:

```
p=donnees[0].split(";");
p=Point(p[0],p[1].replace(",","."));
SetHide(p,true);
for (i in donnees){
q=donnees[i].split(";");
q=Point(q[0],q[1].replace(",","."));
SetHide(q,true);
s=Segment(p,q);
p=q;
}
```

Avec plus de points (donc plus de nombres dans le tableur) on obtient cette courbe:



Cette technique est surtout intéressante avec des données statistiques issues du monde réel (résultats de mesures par exemple) ou des simulations par le tableur de lois statistiques complexes comme les lois de Fisher ou du χ^2...

Fichier JavaScript

La technique est analogue à celle de l'exemple précédent: Le fichier JavaScript chargé est une chaîne de caractère, à laquelle on applique la méthode *eval* pour l'exécuter. Exemple très court: On crée un fichier appelé *PetitScript.js*, qui contient ceci:

```
a=Point(3,2);
```

Puis dans le CaRScript, on entre

```
programme=Load("PetitScript.js");
eval(programme);
```

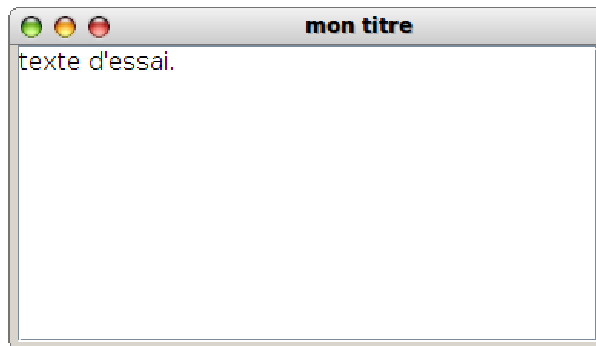
ce qui crée le point de coordonnées (3,2). Pas très interessant, mais on peut ainsi déléguer le gros travail à des fichiers séparés qui peuvent contenir des *function* de JavaScript, et dont le nombre peut être très élevé...

Accès à la console de sortie

Lorsqu'on effectue un *Print*, on modifie une fenêtre Java spécialement réservée à l'affichage de données de sortie: La console de sortie. On peut y accéder par **getCONSOLE()**. Ce qui permet par exemple de lui mettre un titre:

```
sortie=getCONSOLE();
sortie.setTitle("mon titre");
Print("texte d'essai.");
```

La console de sortie a maintenant un titre:



getCONSOLE() permet également de changer la couleur du texte ou sa taille. Ou de modifier la taille de la console elle-même, ou de la mettre par-dessus les autres fenêtres...

Accès à la construction

getC() retourne la construction. on peut affecter une variable avec la construction! Le seul problème: C'est quoi au juste, la construction? En gros c'est la liste des objets géométriques, celle qui est sauvegardée dans le fichier *zir*.

Équation de droite

Par exemple, sur une figure qui comprend juste une droite par deux points, la construction contient deux points et une droite. Et comme le dernier objet construit est la droite, on peut afficher son équation cartésienne avec

```
constr=getC();
droite=constr.last();
Println(droite.equation);
```

La première ligne récupère la construction; la deuxième en extrait le dernier objet: La droite; enfin la dernière ligne imprime son équation. Comme on ne sait pas d'avance si la droite est construite en dernier, on peut aussi la chercher par son nom, en supposant qu'elle s'appelle *d1*:

```
constr=getC();
droite=constr.find("d1");
Println(droite.equation);
```

Le droites ne sont pas les seuls objets à avoir une équation... La question qui pointe maintenant n'aura pas de réponse ici: Comment fait-on pour savoir quelles autres choses on peut faire avec *getC()*? Le script suivant donne la liste des méthodes de l'objet *construction*:

```
m=new Packages.java.lang.Class.forName("rene.zirkel.construction.Construction");
methodes=m.getMethods();
for (i in methodes){
Println(methodes[i]);
}
```

(en fait pour trouver le nom de l'objet, il a suffi d'un `Print(getC());` le nom de l'objet est celui qui a été donné ci-dessus). Pour savoir ce que font ces méthodes et comment on les utilise, pour l'instant, seule la consultation du code source de CaRMetal (autorisée parce que ledit code source est placé sous license GPL) donne des réponses...

Accès à la figure

La figure CaRMetal est aussi accessible, par **getZC()**. Le nom de cette instruction vient de ce que dans le code source, la figure s'appelle *ZirkelCanvas*. Autre question difficile: Quelle est la différence entre la construction et la figure? Dans la construction, il y a des objets géométriques. La figure, c'est ce qui apparaît à l'écran. C'est illustré par le nom de "Canvas": La différence entre la construction et la figure, c'est un peu la différence entre les taches de peinture et le tableau complet, toile de lin comprise. En particulier, il y a des choses qui ne sont pas dans la construction, mais dans la figure, comme par exemple la taille des points.

Accès aux méthodes

getZC() permet donc d'avoir la figure CaRMetal comme variable et de la manipuler par CaRScript. Par exemple, sur une figure comprenant un nuage de points (de forme ronde ou carrée), le script suivant produit un effet psychédélique en modifiant au-delà du raisonnable la taille des points:

```
canevas=getZC();
for(i=1;i<=50;i++){
canevas.PointSize=i;
Pause(100);
}
```

Chaque modification de la figure aura pour effet de redonner aux points leur taille d'origine, mais une fois le script fini, le Monkey baladera bel et bien ces énormes points sur l'écran!

Mise à jour

RefreshZC() met à jour l'affichage de la figure. Son effet est le même que `Pause(0)`.