

Manuel viewer 3D

La science c'est le plaisir de discuter pour comprendre

Gianni Mocellin

Straco
www.straco.ch
07.10.2025, 05h00

Introduction	4
Session, séance, réunion, assemblée	4
4 Le système	4
4.1 Virgule, point-virgule et espace	5
4.2 L'idée variable <i>ans</i>	5
4.3 Abductions, déductions, affectations et précédences	6
Abductions	6
Déductions	7
Tableau des déductions par ordre de précedence	8
4.4 Idées variables, types et moulage (casting)	9
4.5 Idées constantes internes	12
4.6 Ajout de constantes utilisateur	13
4.7 Tableaux (arrays)	14
4.8 Appel de fonctions	14
4.9 Fonctions internes	15
4.9.1 Déductions de base	16
Tableau des appels explicites aux productions	16
Tableau de deux fonctions de bas niveau	16
4.9.2 Productions de base en multilogique	18
Abductions	18
Tableau des ordres.....	18
Déductions non valoriques.....	18
Tableau des déductions de base.....	19
Tableau de déductions pratiques	19
4.9.3 Binaires	19
Tableau des abductions binaires	20
Tableau des déductions binaires	20
Tableau des abductions bitaires	20
Tableau des déductions bitaires	20
4.9.4 Dessins	21
Couleurs et alpha	22
Tableau des couleurs prédéfinies.....	22
Tableau des couleurs variables	22
Tableau de options graphiques	23
Tableau des propriétés de dessin.....	24
Etiquettes	24
Versatrices	26
4.9.5 Sliders (controls)	27
4.9.6 Productions mathématiques	27
Fonctions goniométriques.....	27
Fonctions numériques.....	28
Fonctions logarithmiques	28
Fonctions puissance	28
4.9.7 Créations orologiques	28
4.9.8 Créations infologiques	28
4.9.9 Créations cerveau	29
4.9.10 Réseau	30
4.10 Créations dynamiques	31
4.10.1 Créations dynamic nommées	33
4.10.2 Animations	34

4.11 Controles	36
4.11.1 <i>if...else</i>	36
4.11.2 <i>for</i>	36
4.11.12 <i>while</i>	37
4.11.13 <i>switch</i>	37
4.12 Ecriture de fonctions comme des modules.....	37
4.12.1 <i>Modules (batches)</i>	40
4.13 Couleurs.....	41
4.13.1 <i>Fonction autocolor spéciale</i>	42
5 Etiquettes.....	43
5.1 Modes <i>text</i> et <i>eqn</i>.....	44
5.1.1 <i>Détails du mode txt</i>	45
5.1.2 <i>Le mode eqn en détail</i>	46
5.2 Polices	46
5.3 Modulation des polices	47
5.4 Blancs forcés, nouvelles lignes forcées.....	47
5.5 Alignements	48
5.6 Subscripts et superscripts.....	48
5.7 Parenthèses	48
5.8 Tableaux.....	49
5.9 Racines carrées.....	49
5.10 Fractions.....	49
5.11 Chapeaux	50
5.12 Couleurs.....	50
Couleurs particulières	50
5.13 Commandes personnelles	51
5.14 Symboles spéciaux	52
Tableau des symboles de commandes multilogiques.....	53

Introduction

Le présent texte a pour but de présenter ce que le cerveau peut faire lors de

une session scientifique

Session, séance, réunion, assemblée

- A sitting together of a court, council, legislature or the like for conference or the transaction of business
 - A single continuous sitting or period of sitting of persons so assembled
 - The period or term during which such series is held
- A single continuous course or period of study, lessons etc. in the work of a day at school
 - A period of the year into which instruction is organized at college, university or other educational institution
- A period of time during which a group of persons meet to pursue a particular activity

Session, conference, discussion, hearing, period, term, affair, assembly, concourse, huddle, meet, showdown, sitting, get-together, jam-session, workshop.

4 Le système

Le système gère un ensemble des trois de types d'idées scientifique

les versidées

les oridées

et

les infidées

Ces types peuvent être automatiquement fixés par le système

Les abductions et déductions peuvent être surchargées par le cerveau

Le système possède un support limité pour les vecteurs et les matrices

Les instructions

dynamic

permettent une grande flexibilité

Les instructions *dynamic* dépendent en effet des idées variables utilisées pour les évaluer

Chaque fois que l'une des idée variable change dans une instruction, l'instruction *dynamic* est réévaluée

Les fichier .g contiennent

des fonctions

et

des batch

4.1 Virgule, point-virgule et espace

,

termine une instruction et détermine si le résultat doit être écrit sur la console et dessiné sur la feuille

```
>> a=eI,
```

va dessiner une flèche a sur la feuille et écrire les coordonnées de a sur la console

L'instruction

```
>> a=eI;
```

de dessine pas la flèche sur la feuille et n'écrit pas les coordonnées sur la console

L'idée a existe bien mais elle n'est simplement montrée nulle part

Ne pas mettre de signe à la fin d'une instruction sur la console est équivalent à une virgule

```
>> a=eI
```

Cette instruction va dessiner une flèche sur la feuille et écrire les coordonnées sur la console

Dans les fichier .g il faut toujours terminer une instruction soit par une virgule soit par un point-virgule

4.2 L'idée variable *ans*

Quand le cerveau entre *e1* sur la console

```
>>e1
ans=1.00*e1
```

on voit que l'état de l'idée *e1* est affecté à *ans*

Quand le cerveau donne une instruction au système, qui est implicitement terminé par une virgule, toute idée variable à laquelle a été assignée un état est écrit sur la console et dessiné sur la feuille

Si aucune assignation n'a été faite par le cerveau, le résultat de l'instruction est affecté à l'idée variable

ans

Quand le cerveau donne une instruction terminée par un point-virgule, l'idée variable *ans* est effacée

4.3 Abductions, déductions, affectations et précédences

Abductions

Les abduction ont la plus grande précédence dans le système

Elles sont donc exécutées avant toute autre déduction

Comme chacune des abductions s'annule d'elle-même, le système détermine lui même si l'abduction est à faire ou pas

```
>>- ~ ~ ! ! idée
```

Aucune abduction ne sera appliquée à l'idée

idée

car toutes les abductions ci-dessus s'annulent les unes des autres

Symbole	Abduction	Précédence
-	adverser	10
~	renverser	10

!	inverser	10
---	----------	----

Tableau des abductions

Déductions

Symbole	Déduction	Précédence
^	enjecter	9
	réunir	8
&	croiser	8
.	cojecter et injecter	7
*	imposer	6
/	opposer	6
+	adjoindre	5
-	subjoindre	5
<	plus petit	4
>	plus grand	4
<=	plus petit ou égal	4
>=	plus grand ou égal	4
==	égal	3
!=	différent	3
&&	et	2
	ou	1

=	affectation	0
---	-------------	---

Tableau des déductions par ordre de précedence

Toutes les déductions sont

associatives par la gauche

excepté l'affectation (qui n'est pas réellement une déduction)

Toutes les instructions (productions, constructions) sont traduites en appels de fonction par le système (4.9.1 et 4.9.3)

L'instruction abrégée

.

est traduite par le système dans la fonction

hip

c'est-à-dire

injection de Hesteness

par défaut

Mais elle peut être fixée autrement par la fonction

inner_product()

Le défaut est *hip* mais

mhip()

rcont()

et

lcont()

sont aussi possibles

4.4 Idées variables, types et moulage (casting)

Les idées variables comme x, a et b ont toujours un type

Le type d'idées peut être l'un des trois types suivants

- *e3ga*

versologique

versidée

- *p3ga*

orologique

oridée

- *c3ga*

infologique

infidée

Les idées variables héritent leur type des idées variables utilisées pour les produire

Ainsi si les idées variables a et b sont toutes deux de type p3ga, alors l'idée résultante d'une production sera aussi de type p3ga

Le type d'une idée variable détermine comment l'idée variable est interprétée, écrite et dessinée par le système

Le système peut analyser

les enjectages et les versations

des trois logiques que sont la versologique, l'orologique et l'infologique et les dessiner comme telles

Les idées qui ne peuvent pas être interprétées ne sont pas dessinées par le système

Les types des idées variables peuvent être fixés

$a=(c3ga)e1$

$\gg a=e1$

$$b=(e3ga)ni$$

$$>> b=0$$

Dans le second exemple aucune interprétation n'est possible durant l'affectation (casting)

Comme il n'y a pas l'idée

ni

dans la versologie elle est simplement ignorée par le système

Comme autre exemple on peut dire que

une flèche libre de l'infologique

ne sera pas interprétée en

flèche de la versologique

en la re-qualifiant simplement

Si le nom d'une idée variable n'existe pas encore il est supposée valoir

0

>>a=LongNomD'uneIdéeVariableQuiN'existePasEncore

a=0

Le nom des productions est très important car il permet de les distinguer des noms des idées

$$a=duel(x)$$

\approx

$$a=duel^*(x)$$

$$>> a=0$$

Ici le cerveau a entré *duel* au lieu de *dual*

Le système suppose que *duel* est une idée variable et non la production *dual*

Il évalue l'imposition de duel, qui est supposée valoir 0 puisque non définie, et de l'idée variable x et le résultat sera donc encore 0

Le blanc entre les deux idées est important

Dans la deuxième ligne on a mis une étoile explicite pour bien voir qu'on a une imposition

Si un nom comme

alpha

est une production on peut le forcer à devenir le nom d'une idée variable en utilisant la fonction

variable

Le retour à l'état précédent se fait en redéclarant la fonction

```
>> a=alpha(e1, 0.5)
```

```
a=1.00*e1
```

```
>>variable alpha
```

```
>>alpha=1
```

```
alpha=1.00
```

```
>>
```

```
>>function alpha(x,y);
```

L'instruction suivante n'est plus permise

```
>>alpha=1
```

```
line 1:7 : expecting (, found' '='
```

```
ans=1.00
```

```
>> a = alpha(e1,0.5)
```

```
a=1.00*e1
```

4.5 Idées constantes internes

Les idées constantes internes suivantes sont disponibles

- tous les nombres sont des constantes

Les nombres peuvent avoir les formes

*1
1.2
1.2e3
1.2-e3*

Les nombres sont de type e3ga, donc versologiques

- les idées e1, e2, e3 sont les trois unités de l'unologie

Elles sont de type e3ga

-l'idée e0 est l'origine de l'onologie

Elle est de type p3ga

- les idées no et ni sont l'origine et l'infini de l'inologie

Elles sont de type c3ga

einf est synonyme de ni en c3ga

- l'idée pi = 3.14

est de type e3ga

- l'idée e_ = 2.71

est de type e3ga

On note e_ et non pas e pour ne pas faire de confusion avec les e_i

Par défaut les idées constantes ont toujours le type de la plus petite logique qui les contienne

Ainsi les nombres et les uno-flèches sont tous de type e3ga par défaut

Mais ce comportement peut être changé en appelant la fonction

default_model

Par exemple

```
>>default_model(c3ga);
```

Après cette instruction toutes les idées constantes (sauf *e0*) seront de type *c3ga*

Si on veut retourner au comportement normal il suffit de taper

```
>>default_model();
```

sans aucun argument

La spécification de

la multilogique utilisée

pour les idées constantes car elle provoque des différences de comportement de certaines fonctions comme la complémentation par rapport à tout l'univers

Elle provoque aussi une différence d'interprétation

Par exemple *e3* est dessiné comme une flèche en *e3ga* mais est dessiné comme la surface complément en *c3ga*

On peut renommer les idées constantes internes en utilisant la fonction

```
rename_builtin_const()
```

Cette instruction affecte l'affichage textuel des coordonnées d'une idée

Par exemple, on peut renommer le point origine inflologique (*no* par défaut) et le point infini (*ni* par défaut)

```
>> rename_builtin_const(e0,e4);
```

```
>> rename_builtin_const(no, e0)
```

```
>>rename_builtin_const(ni, einf)
```

```
>>a=e0^einf
```

```
a=1.00*e0^einf
```

4.6 Ajout de constantes utilisateur

Le cerveau peut ajouter ses propres idées constantes utilisateur avec

add_const()

Cela peut être utile car les idées variables ordinaires sont effacées quand on fait

clf()

mais les constantes ne le sont pas

Les idées constantes utilisateur sont sujettes à des règles de moulage/*default_model_rules* décrites ci-dessus

Un exemple

```
>>add_const(I3=e1^e2^e3);
>>I3
ans=1.00*e1^e2^e3

>>remove_const(I3);
```

4.7 Tableaux (arrays)

Les indices des tableaux peuvent être utilisés pour gérer des nouveaux noms (identifiants) d'idées variables

Par exemple, on ne peut pas

passer des arrays à des fonctions

ou

les retourner depuis une fonction

La syntaxe du système pour accéder à un élément est

A[idx1][idx2] ... [idxn]

4.8 Appel de fonctions

L'appel d'une fonction se fait de la manière suivante

```
>>result_value = function_name(argument_name_1, ... , argument_name_n);
```

Par exemple si on veut projeter un enjctage a sur un enjctage b et mémoriser le résultat dans l'enjctage c

$c = \text{project}(a, b);$

Le système cherche toujours la fonction la plus proche pour faire le job

La plus proche signifie

- la fonction doit avoir

. le bon nom

et

. le bon nombre d'arguments

- de préférence tous les arguments doivent avoir le même type à savoir

$e3ga, p3ga$ ou $c3ga$

conviendrait parfaitement

- si un appariement parfait ne peut être trouvé, le meilleur appariement suivant est cherché

. toutes les fonctions avec le bon nom et le bon nombre d'arguments sont scrutées

. la fonction la plus semblable est celle pour laquelle

la distance de coercion

est la plus petite

Cette distance de coercion est définie comme ci-dessous

une coercion vers une logique plus complexe est préférée à une coercion vers une logique moins complexe puisque dans ce cas aucune information n'est perdue

4.9 Fonctions internes

Une description de ces fonctions est accessible par

$\text{help}();$

et

$\text{help}(\text{sujet})$

4.9.1 Dédutions de base

De nombreuses déductions sont disponibles par des appels abrégés

Néanmoins elles peuvent être spécifiées explicitement comme dans le tableau ci-dessous

<i>production(arguments)</i>	<i>résultat</i>
<i>gp(a,b)</i>	<i>imposition(a,b)</i>
<i>igp(a,b)</i>	<i>opposition(a,b)</i>
<i>op(a,b)</i>	<i>enjection(a,b)</i>
<i>hip(a,b)</i>	<i>injection d'hestenes(a,b)</i>
<i>mhip(a,b)</i>	<i>injection d'hestenes modifiée(a,b)</i>
<i>lcont(a,b)</i>	<i>injection depuis la gauche(a,b)</i>
<i>rcont(a,b)</i>	<i>injection depuis la droite(a,b)</i>
<i>scp(a,b)</i>	<i>produit scalaire(a,b)</i>
<i>meet(a,b)</i>	<i>réunion(a,b)</i>
<i>join(a,b)</i>	<i>croisement(a,b)</i>

Tableau des appels explicites aux productions

Deux autres déductions sont également disponibles en parfum

valorique inologique

et peuvent être utiles pour un travail de bas niveau

Par exemple, le croisement et la réunion utilisent ces déductions de manière interne

<i>production(arguments)</i>	<i>résultat</i>
<i>gpem(a,b)</i>	<i>imposition en valorique inologique(a,b)</i>
<i>lcem(a,b)</i>	<i>injection de gauche en valorique inologique(a,b)</i>

Tableau de deux fonctions de bas niveau

4.9.2 Productions de base en multilogique

Abductions

Les abductions sont des productions ayant un seul argument

<i>abduction(argument)</i>	<i>résultat</i>
<i>scalar(a)</i>	<i>la partie numérique de a</i>
<i>dual(a)</i>	<i>le complément de a dans tout l'univers</i>
<i>reverse(a)</i>	<i>la renversée de a</i>
<i>clifford_conjugate(a)</i>	<i>la conjuguée de clifford de a</i>
<i>grade_involution(a)</i>	<i>l'involution de l'enjectance de a</i>
<i>inverse(a)</i>	<i>la versatrice inverse de a</i>
<i>general_inverse(a)</i>	<i>l'inverse de a même si a n'est pas une versatrice (0 si l'inverse n'existe pas)</i>
<i>negate(a)</i>	<i>l'adversion de a</i>
<i>norm2(a)</i>	<i>la somme des carrés de tous les côtés de a</i>
<i>norm_r(a)</i>	<i>la partie d'enjectance 1 de a</i>
<i>norm(a)</i>	<i>le carré de abs(norm_r(a)) multiplié par le signe de a</i>
<i>normalize(a)</i>	<i>a / abs(norm_r(a))</i>
<i>grade(a)</i>	<i>si a est un enjactage, retourne l'enjectance du premier argument et -1 sinon</i>
<i>grade(a,b)</i>	<i>la partie d'enjectance b de l'enjactage a</i>
<i>versor_parity(a)</i>	<i>0 si a est une versatrice paire 1 si a est une versatrice impaire -1 si a n'est pas une versatrice</i>

Tableau des ordres

Déductions non valoriques

Les déductions sont des productions ayant plusieurs arguments

<i>déduction(arguments)</i>	<i>résultat</i>
$add(a,b)$	$adjonction(a,b)$
$sub(a,b)$	$subjonction(a,b)$
$dual(a,b)$	<i>le complément de a dans b</i>

Tableau des déductions de base

Il y a quelques déductions de base pratiques pour faire des versatrices, des projections, des réjections et des décompositions (factorisations) d'énjectages

<i>déduction(arguments)</i>	<i>résultat</i>
$versor_product(a,b)$	<i>retourne $a*b*inverse(a)$</i>
$vp(a,b)$	<i>synonyme de $vp(a,b)$</i>
$inverse_versor_product(a,b)$	<i>$inverse(a)*b*a$</i>
$ivp(a,b)$	<i>synonyme de $inverse_versor_product$</i>
$project(a,b)$	<i>projection de a dans b</i>
$reject(a,b)$	<i>dijection de a depuis b</i>
$factor(a,b)$	<i>le facteur b d'un éjectage a b doit être un nombre entier entre 1 et grade a</i>

Tableau de déductions pratiques

4.9.3 Binaires

Un certain nombre de déductions binaires sont disponibles pour faire de la binologie

la bine 0.0 est fausse

Toute bine qui n'est pas fausse est considérée comme vraie

<i>déduction(arguments)</i>	<i>résultat</i>
-----------------------------	-----------------

$not(a)$	vrai si a est faux
----------	----------------------

Tableau des abductions binaires

<i>déduction(arguments)</i>	<i>résultat</i>
$equal(a,b)$	vraie si $a-b = 0$
$ne(a,b)$	vraie si $a-b \neq 0$
$less(a,b)$	vraie si $scalar(a) < scalar(b)$
$greater(a,b)$	vraie si $scalar(a) > scalar(b)$
$le(a,b)$	vraie si $scalar(a) \leq scalar(b)$
$ge(a,b)$	vraie si $scalar(a) \geq scalar(b)$
$and(a,b)$	vrai si a est vrai et b est vrai
$or(a,b)$	vrai si a est vrai ou b est vrai

Tableau des déductions binaires

La logique binaire bit à bit peut être faite selon les tableaux suivants

$bit_not(a)$	le bitwise not de $scalar(a)$
---------------	-------------------------------

Tableau des abductions bitaires

<i>déduction(arguments)</i>	<i>résultat</i>
$bit_and(a,b)$	le bitwise and de $scalar(a)$ et $scalar(b)$
$bit_or(a,b)$	le bitwise or de $scalar(a)$ et $scalar(b)$
$bit_xor(a,b)$	le bitwise xor de $scalar(a)$ et $scalar(b)$
$bit_shift(a,b)$	le bitwise shift left de $scalar(a)$ par $scalar(b)$

Tableau des déductions bitaires

Le second argument de bit_shift peut être négatif pour un right shift

Le seul usage de ces productions bitwise est de permettre une modification de la fonction

autocolor()

où quelques tests sur les bits sont nécessaires

Les productions bitwise sont de peu d'usage pour la multilogique

4.9.4 Dessins

Plusieurs propriétés graphiques des multienjettages peuvent être fixées par certaines instructions

```
>>a=cyan(a)
a=1.00*e1
>>
```

dessine la fêche a en cyan

```
>>green(a)
ans=1.00*e1
>>
```

On pourrait s'attendre à ce que

green(a)

dessine le multienjettage *a* en vert

Mais ce qui se passe effectivement est que la valeur de *a* est assignée à *ans*

Ensuite *ans* est dessinée

Comme *ans* est égale à *a* elle peut ou pas être dessinée au dessus de *a*

Quand l'instruction suivante est donnée, la valeur *ans* va être écrasée (overwritten) ou enlevée (removed) et la couleur de *a* n'a pas changé

Ainsi les fonctions qui modifient

les propriétés graphiques

ne mettent que certains flags et valeurs sur les variables intermédiaires et n'ont aucun effet à moins que de telles variables intermédiaires ne soient assignées à quelque-chose

Les fonctions graphique peuvent être emboîtées

```
>>a=cyan(stipple(e1))
```

$$a=1.00*e1$$

$$>>$$

Couleurs et alpha

<i>production(arguments)</i>	<i>effet</i>
<i>red(a)</i>	<i>met a en rouge</i>
<i>green(a)</i>	<i>met a en vert</i>
<i>blue(a)</i>	<i>met a en bleu</i>
<i>white(a)</i>	<i>met a en blanc</i>
<i>magenta(a)</i>	<i>met a en magenta</i>
<i>yellow(a)</i>	<i>met a en jaune</i>
<i>cyan(a)</i>	<i>met a en cyan</i>
<i>black(a)</i>	<i>met a en noir</i>
<i>grey(a)</i>	<i>met a en gris</i>
<i>gray(a)</i>	<i>met a en gris</i>

Tableau des couleurs prédéfinies

<i>color(a, r, g, b)</i>	<i>met a en r,g,b</i>
<i>color(a, r, g, b, a)</i>	<i>met a en r,g,b, alpha</i>
<i>color(a, value)</i>	<i>l'opacité de a devient alpha</i>

Tableau des couleurs variables

Les couleurs et alpha doivent être dans la gamme [0.0,1.0]

Un alpha 0.00 est totalement transparent et un alpha 1.00 est totalement opaque

Les informations peuvent être dessinés hachurés, grillagés, avec ou sans pertinence ou altériorité

Certaines interprétations des informations permettent le dessin d'une silhouette (outline)

<i>instruction(argument)</i>	<i>résultat</i>
<i>stipple(a)</i>	<i>hachuré</i>
<i>no_stipple(a)</i>	<i>non hachuré</i>
<i>wireframe(a)</i>	<i>grillagé</i>

<i>no_wireframe</i>	<i>sans grillagé</i>
<i>outline(a)</i>	<i>silhouette</i>
<i>no_outline(a)</i>	<i>pas de silhouette</i>
<i>weight(a)</i>	<i>dessine la pertinence de a</i>
<i>no_weight(a)</i>	<i>sans pertinence</i>
<i>ori(a)</i>	<i>altériorité de a</i>
<i>no_ori(a)</i>	<i>pas d'alteriorité</i>

Tableau de options graphiques

Pour cacher ou montrer une information utiliser les ordres *show(a)* ou *hide(a)*

Mais ne pas oublier d'assigner

show()

```
>>a=e1
a=1.00*e1
dessine a
```

```
>>a=hide(a),
a=1.00*e1
cache a malgré la virgule
```

```
>>a=show(a);
a=1.00*e1
montre a malgré le point virgule
```

hide(a)

ne cache pas a mais assigne la valeur de a à ans et cache ans

Quelques interprétations d'informations peuvent être dessinées de différentes manières

```
>>line=ori(no^e1^ni)
```

propose un pop-up menu

Draw method

dans les contrôles à droite

Sélectionne plusieurs manières de dessiner l'intériorité d'une direction

On peut aussi fixer la méthode de dessin depuis la console

```
>>line=dm2(ori(no^e1^ni))
```

L'indice 2 de cet exemple peut aller de 1 à 7

S'il est en dehors de cette gamme pour l'interprétation de l'information en question, la méthode par défaut est utilisée

On peut retrouver les propriétés de dessin des informations en utilisant les commandes suivantes

commande(arguments)	information en retour
<i>get_color(a)</i>	<i>un vecteur avec le rgb de a</i>
<i>get_alpha(a)</i>	<i>un nombre avec l'alpha de a</i>
<i>get_stipple(a)</i>	<i>une bine avec le signal stipple de a</i>
<i>get_wireframe(a)</i>	<i>une bine avec le signal wireframe de a</i>
<i>get_outline(a)</i>	<i>une bine avec le signal outline de a</i>
<i>get_weight(a)</i>	<i>une bine avec le signal weight de a</i>
<i>get_ori(a)</i>	<i>une bine avec le signal ori de a</i>
<i>get_hide(a)</i>	<i>une bine avec le signal hide de a</i>

Tableau des propriétés de dessin

Étiquettes

Une étiquette peut être affiché à "la position" d'une idée en utilisant la fonction

label()

Les idées n'ont pas toutes

un aspect positionnel

auquel cas l'étiquette est dessiné à l'origine

Le premier argument de *label* est la variable qu'on veut étiqueter

Le second argument optionnel est le texte de l'étiquette

Par défaut le nom de la variable est utilisé dans le texte de l'étiquette

```

>>a=e1
1.00*e1

label(a);

>>label(b=2*a)
b=2.00*e1
racourci pour b=2*a, label(b);

>>label(c=e2, "cette idée s'appelle c")
c=1.00*e2

```

Les deux fonctions ci-dessous n'affectent pas vraiment comment la variable est dessinée mais plus comment la variable est interprétée

```

versor(a)
force une interprétation versatrice de a

blade(a)
force une interprétation enjactage de a

```

La fonction *versor()* peut être utile quand une versatrice devient par coïncidence d'enjactence unique

```

>>a=versor(e1^e2)
a=1.00*e1^e2

```

Ceci dessine une rotatrice alors que simplement

```

>>a=e1^e2
2.00*e1^e2

```

dessine un 2-enjactage

La production

```

blade()

```

est utile quand l'incertitude floating point sur une certaine enjactence devient si grande que le système confond un enjactage avec une versatrice

Si on a un bi-enjactage

```

a=e1^e2

```

et qu'à causes de quelques manipulations le bruit floating point met la partie numérique à 0.01 au lieu de 0

```
>> a=e1^e2+0.01
a=0.01 + 1.00*e1^e2
```

a est interprétée et dessinée comme une rotatrice dans ce cas

On peut forcer a à être interprétée comme un enjctage

```
>>a=blade(e1^e2+0.01)
a=0.01 + 1.00*e1^e2
```

Versatrices

Les deux instructions suivantes n'affectent pas réellement la manière dont une réponse est dessinée

```
versor(a)
force une interprétation versatrice de a
```

```
blade(a)
force une interprétation enjctage de a
```

- *versor()* peut être utile quand une versatrice devient incidentellement d'une enjctence unique

```
>>a=versor(e1^e2)
a=1.00*e1^e2
```

Cette instruction dessine une rotatrice alors que

```
>>a=e1^e2
a=1.00*e1^e2
```

dessine un 2-enjctage

- *blade()* est utile quand le bruit sur une certaine partie devient si grand que le système confond un enjctage avec une rotatrice

Supposons qu'on ait

```
>>a=e1^e2
a=1.00*e1^e2
```

mais que du à certaines manipulations le bruit de floating point cause la partie numérique à être *0.01* au lieu de *0.00*

```
>>a=e1^e2 + 0.01
```

$$a=0.01+1.00*e1^e2$$

a est interprété et dessiné comme un rotatrice dans ce cas

Ainsi, pour focer a à être interprétée comme un enjctage il faut faire

```
>>a=blade(e1^e2 + 0.01)
```

$$a=0.01+1.00*e1*e2$$

4.9.5 Sliders (controls)

Des sliders peuvent être créés

```
>>ctrl_range(a=2.0,0.5,10.0)
```

```
>>dynamic{v=a*e1,}
v=2.00*e1
```

On a les sliders suivants à disposition

- *ctrl_bool(name=value)*

crée le slider binaire avec name fixé à value

-*ctrl_range(name=value, min_value, max_value)*

crée un slider de nom name, fixé à la valeur value, limité par min_value et max_value

-*ctrl_range(name=value, min_value, max_value, step)*

ne peut être changé qu'un pas à la fois

- *ctrl_select(value, option1=value1, ... , option-n=value-n)*

crée un menu de sélection de nom name, fixé à value

7 options maximales peuvent être spécifiées et value doit être l'une des options

- *ctrl_remove(name)*

enlève le slider name

4.9.6 Productions mathématiques

Fonctions goniométriques

<i>fonctions</i>	<i>résultat</i>
<i>sin(a)</i>	<i>sinus(a)</i>
<i>cos(a)</i>	<i>cosinus(a)</i>
<i>atan(a)</i>	<i>tangente(a)</i>
<i>asin(a)</i>	<i>arc sinus(a)</i>
<i>acos(a)</i>	<i>arc cosinus(a)</i>

$atan2(a)$	<i>arc tangente(a)</i>
$sinh(a)$	<i>sinus hyperbolique de a</i>
$cosh(a)$	<i>cosinus hyperbolique a</i>
$sinc(a)$	$\sin(a)/a$

Fonctions numériques

$sqrt(a)$	<i>racine carrée(a)</i>
$abs(a)$	<i>taille(a)</i>

Fonctions logarithmiques

$log(a)$	<i>logarithme naturel de scalar(a)</i>
$exp(a)$	<i>exponentiation de a</i>

Fonctions puissance

$pow(a,b)$	<i>a multiplié par b fois lui-même b = entier ≥ 0</i>
$scalar_pow(a,b)$	<i>a élevé à la puissance de b</i>

4.9.7 Créations orologiques

- $p3ga_point(a,b,c)$

*retourne le opoint construit à partir de la flèche $a*e1 + b*e1+c*e3$*

- $p3ga_point(e3ga a)$

retourne le opoint construit à partir de la flèche a

4.9.8 Créations infologiques

- $c3ga_point(a,b,c)$

*retourne le ipoint construit à partir de la flèche localisation $a*e1 + b*e1+c*e3$*

- $c3ga_point(e3ga a)$

retourne le ipoint construit à partir de la flèche localisation a

- $c3ga_point(p3ga b)$

retourne le ipoint construit à partir du opoint b

- *translation_versor(a)*

retourne une translatrice infologique qui translate de la flèche a

- *tv(a)* synonyme de *translation_versor(a)*

- *translation_versor(a,b,c)*

*retourne une translatrice infologique qui translate de la flèche $a*e1+b*e2+c*e3$*

- *tv(a,b,c)* synonyme de *translation_versor(a,b,c)*

4.9.9 Créations cerveau

assign(a,b)

assigne la valeur b à a

cprint("a string")

écrit "a string" dans la console

print(a)

écrit les coordonnées de a dans la console

print(a,prec)

écrit les coordonnées de a avec une certaine précision

prompt()

fixe l'invite par défaut dans la console

prompt("prompt text")

fixe l'invite à "prompt text"

select(a)

sélectionne a comme l'idée variable courante comme un ctrl-left-click

remove(a)

enlève l'idée variable a comme si on cliquait remove

clc()

efface la console, la mémoire de travail, et toutes les slider

clf()

efface toutes les idées variables et idées

reset()

redémare la session du cerveau

4.9.10 Réseau

Le système communique par TCP

add_net_port(6860)

fixe comme prise (port, socket) le port 6860

Après cette commande le cerveau (système) écoute sur le port 6860 les client qui veulent se connecter

On peut le tester en faisant depuis l'application telnet

telnet localhost 6860

pour se connecter

Le cerveau (système) va immédiatement envoyer les valeurs de toutes les idées variables connues

Dans l'application telnet le cerveau peut passer des commandes comme s'il les passait à la session mais elles doivent être suivies par un signe \$

a= e2+e2,\$

On a les commandes suivantes

add_net_port(port)

commence à écouter le port port

remove_net_port(port)

arrête l'écoute du port et disconnecte tous les clients courants

net_status()

affiche le statut du réseau

net_close()

arrête toutes les connexions réseau et ports

On peut lancer le système (cerveau, conscience) en faisant

gaviewer -net

Ceci engage immédiatement le port 6860

Le réseau est désactivé par défaut car n'importe qui sur internet pourrait se connecter au système de cette manière

4.10 Créations dynamiques

*>>a=e1,
a=1.00*e1*

*>>dynamic{b=a^e2,}
b=1.00*e2*

Les créations

dynamic

dépendent des idées variables utilisées pour les évaluer

L'instruction

b=a^e2

dépend des variables *a* et *b*

Chaque fois que l'une de ces deux idées variables change, l'instruction *dynamic* est réexécutée

Puisque des idées constantes comme *e2* ne changent jamais, elles n'ont aucun effet quand une instruction *dynamic* est exécutée

Si après avoir mis l'instruction ci-dessus dans la session le cerveau change une valeur

*>>a=e1,
a=2.00*e1*

le cerveau peut constater que l'idée variable *b* a été mise à jour automatiquement

Si le cerveau fixe la valeur de *b* à volontairement

```
>>b=0,
b=0
```

le cerveau peut constater que

l'idée variable b

ne change pas car la session détecte que l'idée variable b a été changée et elle réévalue l'instruction dynamique impliquant l'idées variable b

Si le cerveau crée de multiples instructions

dynamic

qui assignent des valeurs aux mêmes idées variables les choses peuvent devenir compliquées

```
>>a=e1,
a=1.00*e1
```

```
>>dynamic{b=a^e2,}
b=1.00*e1^e2
```

```
>>dynamic{b=c^e2,}
b=0
```

Directement après que la seconde instruction *dynamic* ait été donnée, l'idée variable b est fixée à 0

Mais immédiatement après cela la session détecte que la première instruction

dynamic{b=a^e2,}

doit être réévaluée parceque la valeur de l'idée variable b a été changée

Ainsi la session va maintenant réévaluer

$b=a^e2,$

Si aucune protection contre une boucle infinie est en place, la session réévalue une nouvelle fois la seconde instruction *dynamic*, suivie par la première et ainsi de suite

Cependant la session protège contre les boucles infinies dans l'évaluation des instructions *dynamic*

Une instruction *dynamic* n'est jamais réévaluée deux fois grace à une sorte de dépendance de boucle dans les instructions *dynamic*

En général il vaut mieux pour le cerveau d'éviter de telles boucles car elles peuvent être déroutantes, déconcertantes, confuses

Quelques remarques pertinentes

- *dynamic* ne peut pas être appelée dans le domaine global

et

ne peut pas être appelé l'intérieur de fonctions ou de blocs appelés par des fonctions

- fixer l'injection en utilisant

inner_product() (voir 4.3)

impose aux instructions *dynamic* utilisant la création (l'opérateur)

.

de ne pas être réévaluées

- la fonction

cld()

efface toutes les instructions *dynamic*

- le fait de terminer des instructions par une virgule ou un point-virgule n'affecte que la visibilité des variables la première fois qu'une instruction *dynamic* est évaluée

hide()

et

show()

affectent quant-à eux la visibilité à chaque réévaluation

4.10.1 Créations *dynamic* nommées

Le problème avec les instructions

dynamic

est que le cerveau (l'utilisateur) a peu de contrôle sur elles une fois qu'elles ont été données

L'utilisateur (le cerveau) peut les enlever en utilisant

cld()

et les afficher

et c'est tout

Les idées variables *dynamic* nommées offrent plus de flexibilité

En ajoutant une étiquette avec un nom à chaque idée variable *dynamic*, l'utilisateur (le cerveau, la conscience) peut modifier les idées plus tard

```
>>a=e1
a=1.00*e1
```

```
>>dynamic{mon_idée_variable_dynamique: b=a^e2,}
      b*e1^e2
```

```
>>dynamic{mon_idée_variable_dynamique: b=a^e3,}
      b*e1^e3
```

4.10.2 Animations

Une manière de faire des animations consiste à écrire des équations *dynamic* qui dépendent de l'idée variable particulière

```
atime
```

```
>>dynamic{print(atime);}
```

```
atime=0
```

Si le cerveau fait

Dynamic -> Start/Resume Animation

```
>>atime=0
>>atime=0.01
>>atime=0.07
>>atime=0.13
>>atime=0.19
```

Comme on le voit, la variable

```
atime
```

est fixé à l'instant correspondant à la durée depuis laquelle l'animation a commencé

```
atime
```

est fixée au maximum à 30 fois par seconde mais elle peut être plus lente, dépendant de la durée nécessaire pour redessiner la vue et réévaluer les variables *dynamic*

Pour arrêter l'animation

Dynamic -> Stop Animation

Pour faire une pause

Dynamic -> Stop Animation

L'écriture d'instructions plus compliquées impliquant *atime* permet des animations plus intéressantes

Le cerveau peut aussi commencer des animations depuis la console (depuis la conscience)

start_animation()

commence l'animation des dynamics en fonction de *atime*

Il est garanti que quand l'animation commence, *atime* est fixée à 0

On peut vérifier par le test

atime==0

et faire quelque initialisations nécessaires

stop_animation()

arrête l'animation

pause_animation()

pause l'animation

resume_animation()

recommence l'animation (synonyme de *start_animation*)

Exemple

>>*cld()*;

efface toutes les idées variables dynamic

>>*dynamic{a=sin(atime)*e1, if (atime>10) stop_animation();}*
*a=-0.86*e1*

>>*start_animation()*;

Cette animation va durer 10 secondes car elle s'arrête elle-même quand *atime* est plus grande que 10

4.11 Contrôles

Plusieurs types de contrôles sont disponibles

4.11.1 if...else...

if(condition) statement
[else statement]

>>*if(a==1) {b=1;} else {b=2;}*

On peut négliger les crochets pour les if simples

Ainsi l'exemple ci-dessus peut être réécrit sans changement sémantique comme

if(a==1) b=1; else b=2;

4.11.2 for

for([init_statement]; [condition_statement]; [update_statement]) effective_statement

Le

init_statement

est exécuté en premier

Puis il est vérifié que

condition_statement

soit vraie avant que toute exécution de *effective_statement*

Si tel est le cas

effective_statement

est exécutée

Sinon la boucle se termine

Le cerveau peut interrompre la boucle prématurément en utilisant l'instruction

break

et la forcer à terminer l'exécution de

effective_statement

par

continue

4.11.12 while

Permet l'exécution d'une boucle mais de manière un peu différente

while (condition) statement

Elle boucle tant que la condition est vraie

4.11.13 switch

Permet de comparer la valeur d'une expression à un groupe d'autres expressions

```
switch(expression){
  case expression1: statements1
  case expression2: statements2
}
```

Tous les statements sont optionnels

Ils peuvent inclure un *break* pour quitter le *switch*

4.12 Ecriture de fonctions comme des modules

Le cerveau peut créer ses propres fonctions et modules (batch) et les charger dans la session

Un module (batch) est une fonction excepté qu'elle exécute dans le même domaine que la fonction appelante

On peut raisonner généralement en terme de fonctions et examiner le concept de module après

Le cerveau peut enregistrer des fonctions dans des fichiers

fichier.g

ou les entrer à la console

Par exemple, une petite fonction qui retourne la plus grande de deux variables peut être entrée directement à la console

```
>>function max(a,b) {if (norm(a) > norm(b) {return a;} else {return b;}};
```

puis le cerveau peut utiliser

max

```
>>max(1,3)
ans=3.00
```

La forme générale d'une fonction est la suivante

```
function fuction_name(
[e3ga | p3ga | c3ga]argument_1_name, argument_2_name, ...
[e3ga | p3ga | c3ga]argument_n_name, argument2_name)
{function_statements}
}
```

Le premier mot d'une définition de fonction est toujours *function* ou *batch*

Il est suivi par le nom de la fonction

function_name

Entre une parenthèse ouvrante et une parenthèse fermante sont spécifiés les arguments de la fonction (les arguments de la déduction)

Une fonction peut avoir de 0 à 8 arguments

La valeur 8 est codée fixe

Plus de 8 argument provoque un arrêt de la session

La spécification de type de logique

e3ga, *p3ga* ou *e3ga*

est optionnelle

Si aucun type n'est spécifié, les arguments effectifs correspondent (match) toujours parfaitement

Une spécification de type de retour ne peut pas être spécifiée

Un multienjettage est toujours retourné sans aucune restriction de logique ou de type

Dans une fonction le cerveau peut mettre des instructions arbitraires

Une instruction

return expression_value

provoque la fin d'exécution de la fonction et retourne

expression_value

Le cerveau peut aussi définir des fonctions de fonctions, des fonctions à l'intérieur de fonctions, des fonctions emboîtées

function une_fonction_f_de(a)

{

function neg(b){

return(-b);

}

*return 2 * neg(a);*

}

Le cerveau peut déclarer l'existence d'une fonction sans la définir vraiment

function une_fonction_g(a);

Ainsi la déclaration de fonction est comme une définition de fonction si ce n'est que le corps de la fonction est remplacé par un point-virgule

Toutes les variables à l'intérieur d'une fonction sont locales par défaut

Cela signifie que le cerveau ne peut pas voir les idées variables depuis le domaine global

Ni le cerveau ne peut fixer des idées variables dans le domaine global depuis l'intérieur d'une fonction

```

function fixer_une_variable_globale(a)
    {
        :: variable_globale=a;
    }

```

4.12.1 Modules (batches)

Les modules sont des fonctions qui s'exécutent dans le même domaine que l'appelant

Cela signifie que le cerveau doit faire attention dans l'appel de modules et être sûr que les noms d'idées variables utilisés dans le module ne sont pas utilisés pour un autre usage dans le domaine appelant

Par exemple, si des noms d'arguments formels d'un module sont déjà présents dans le domaine de l'appelant ils seront écrasés (overwrite)

Les modules sont très utiles pour écrire des démonstrations interactives et des présentations

Souvent le cerveau veut exécuter un groupe d'instructions qui sont trop fatigantes à entrer à la main

Le cerveau peut collecter ces instructions dans un module, les enregistrer dans un fichier .g et le faire charger

Il y a une instruction spéciale

suspend

qui permet une interaction à l'intérieur des modules

batch demonstration_1()

{

a=show(e1);

cprint ("a' vaut maintenant e1.");

cprint("Balade le bouton gauche pour tourner la vue.");

cprint("Tape une commande arbitraire sur la console");

cprint("Appuye sur enter pour continuer");

ou encore fixer une invite (prompt, incitation) spéciale

```
prompt("démonstration_1 suspendue... >>");
```

Remettre l'incitation, l'invite, le souffleur (prompt, prompter) en mode normal

```
prompt();
```

```
cprint("Bienvenue de retour à la démonstration_1");
```

```
a=show(e2);
```

```
cprint("'a' est maintenant égal à e2");
```

```
cprint("Ceci est la fin de la démonstration démonstration_1");
```

L'usage de

```
suspend
```

n'est pas permis dans le domaine global

4.13 Couleurs

Les couleurs des idées sont fixées par une fonction interne appelée

```
autocolorfunc()
```

Toutes les fois qu'une idée variable dans le domaine global à laquelle est assignée une valeur, elle est passée à travers

```
autocolorfunc()
```

pour lui donner un aspect (look, apparence)

La fonction *autocolorfunc()* interne par défaut change la couleur d'une variable selon son éjectence (mais seulement si la couleur n'a pas été explicitement fixée par les fonctions d'affichage (voir 4.9.4)

La fonction active aussi le hachurage pour les idées imaginaires de l'infologique comme les cercles d'intersection de deux sphères qui ne se croisent pas

On peut modifier les caractéristiques (features) de la fonction en fixant la variable

```
autocolor
```

à

false

Si la variable *autocolor* est à *false* toutes les variables prennent la même couleur de fond

4.13.1 Fonction autocolor spéciale

Le cerveau peut écrire sa propre fonction

autocolorfunc()

s'il n'est pas satisfait de celle par défaut

Pour cela il faut regarder à quoi elle ressemble

A noter que

autocolorfunc()

s'exécute plus rapidement que les fichiers *.g* car elle est écrite en C++ et est compilée en code machine, alors que les fichiers *.g* sont interprétés

Deux choses importantes

- Le cerveau peut trouver la manière dont une idée est interprétée en appelant la fonction

get_interpretation(a)

Elle retourne un bitmap qui contient de l'information sur l'interprétation de la variable *a*

On peut utiliser les fonctions bitmap binaires pour extraire de l'information

Voir le fichier

autocolor.g

- Le cerveau peut savoir s'il a fixé au préalable une propriété d'affichage d'une idée variable

Bien sûr, le cerveau ne veut pas outrepasser, bafouer, contourner, abroger, dominer, neutraliser (override) ce qu'il a explicitement fixé

Le cerveau peut appeler la fonction

get_draw_flags(a)

Elle va retourner un bitfield qui contient l'information sur les propriétés d'affichage qui ont été fixées par le cerveau (l'utilisateur)

5 Etiquettes

L'entrée est passée dans un analyseur (parser) qui vérifie les erreurs de syntaxe et la convertit en instructions d'affichage

Les principales caractéristiques de l'éditeur sont

- modes *text* et *equation*
 - *sub* et *sup*
- quatre fonts: *regular*, *italic*, *bold*, *greek*
- quatre symboles graphiques spéciaux
 - *tabulars*
- alignements *left*, *right*, *center* et *justifiable*
 - *chapeaux (hats)*
 - *scalable parenthesis*
 - *square roots*
 - *fractions*
- commandes propres (*macros*)
- couleurs propres

L'arbre d'analyse (parse tree) obtenu, les instructions graphiques sont données

Parceque l'analyse, l'interprétation et l'affichage ne sont pas faits en un passage, une grande place mémoire est nécessaire si de longs textes sont fournis en entrée

lex

et

yacc

sont utilisés pour analyser l'entrée

La polic (font) utilisée pour l'écriture est la texture 1024x1024 pixels de 'GL_ALPHA'

Elle est directement incluse dans le code source dans

fontdata.cpp

La police est initialisée et contient tous les symboles ASCII des quatre polices *regular*, *italic*, *bold* et *greek*

Elle contient aussi quelques symboles spéciaux au système

5.1 Modes *text* et *eqn*

Il y a deux modes d'analyse avec des différences subtiles

Le mode

txt

est le mode par défaut et est utilisé pour taper du texte ordinaire, comme une phrase

Le mode

eqn

sert à taper des équations

Les blancs, les nombres et les opérateurs sont traités différemment dans les deux modes et quelques commandes comme *sqrt* et *frac* ne sont valables qu'en mode *eqn*

Voici un exemple de passage, commutation, coupure, changement (switching) entre les modes

La barre oblique inverse (backslash) suivie par des lettres est interprétée comme une commande

$\backslash\textit{txt}\{Ceci est un texte ordinaire\}$

$\backslash\textit{eqn}\{Ceci est une équation\}$

On peut combiner les modes *txt* et *eqn*

$\backslash\textit{txt}\{Une sphère:\}$

$\backslash\textit{eqn}\{\boldsymbol{s}\}$

=

$\backslash\textit{par}\{()\}$

$$\{\boldsymbol{q}\} + \frac{1}{2}\rho^2\boldsymbol{e}_{\infty}^{\ast}$$

La partie *eqn* est emboîtée dans la partie *txt*

Comme la méthode d'analyse (parsing) par défaut est *txt* l'instruction suivante est équivalente à la précédente

`\txt{Une sphère:`

`\eqn{\bold{s}}`

=

`\par{}`

$$\{\boldsymbol{q}\} + \frac{1}{2}\rho^2\boldsymbol{e}_{\infty}^{\ast}$$

5.1.1 Détails du mode *txt*

En mode *txt* plusieurs blancs, nouvelle ligne et tab sont entrés comme un seul blanc

Les blancs en fin de ligne sont ignorés

Pour forcer l'insertion de blancs il faut utiliser la commande

`ws`

Les séquences de lettre, de nombres, de sous_tirets, de signes + et - sont interprétés comme des mots

Les autres caractères ASCII sont interprétés comme des mots d'un seul caractère

La barre oblique inverse (backslash) suivie par des lettres est interprétée comme une commande

Une barre oblique inverse suivie par des nombres est interprétée comme un argument de commande personnelle (voir 5.13)

Deux barres inverses suivie par n'importe quel autre caractère est interprété comme une abbréviation pour la commande de nouvelle ligne

`newline`

Une barre inverse suivie de n'importe quel autre caractère est interprétée comme une séquence d'échappement (escape sequence) pour ce caractère

Pour produit une barre inverse réelle dans un texte il faut utiliser la commande

backslash

Les commandes

frac, sqrt et par

ne sont pas valides en mode *txt*

5.1.2 Le mode *eqn* en détail

En mode *eqn* tous les blancs sont ignorés sauf s'ils ont été forcés avec la commande

ws

Les caractères et symboles qui sont reconnus comme des opérateurs sont automatiquement entourés par un petit morceau de blanc

Une séquence de lettres est interprétée comme un mot

Ces mots sont affichés en italique par défaut

Une séquence de nombres, possiblement suivis par un exposant, sont interprétés comme des nombres

\eqn{123e-3}

Les commandes *frac, sqrt, har* et *par* peuvent être utilisées pour afficher des racines (carrées), des fraction, pour mettre des chapeaux sur des expressions et pour entourer une expression entre parenthèses

5.2 Polices

Quatre polices sont disponibles

regular, bold, italic et greek

\regular{Police en régular.}

\bold{Police en bold.}

\italic{Police en italic regular.}

\greek{Police en greek regular.}

Les quatre polices ont des abréviations

fr, fb, fi et fg

Les polices peuvent être emboîtées les une dans les autres tout comme dans les modes *txt* et *eqn*

```
\regular{Regular font.}\newline
\bold{Bold font.}\newline
```

La police par défaut en math est le mode *italic* pour les lettres et *regular* pour les autres symboles

L'alphabet grec est projeté sur l'alphabet ASCII par des commandes

Il y a deux manières de produire des caractères grecs

- la première manière est de passer à la police *greek*

```
\greek{a}
```

- la seconde est d'utiliser une commande pour produire un symbol grec

```
\alpha
```

5.3 Modulation des polices

Le système commence toujours à afficher avec une taille par défaut

La taille des caractères peut changer implicitement par l'usage de certaines commandes et constructions

Par exemple, la taille de la police *sub* ou *sup* est de 0.6 fois la taille du texte parent

On peut explicitement moduler la taille de la police en utilisant la commande

scale

```
\scale{1}{Toto}
```

```
\scale{0.5}{Toto}
```

```
\scale{2}{Toto}
```

5.4 Blancs forcés, nouvelles lignes forcées

Pour forcer des blancs entre mots ou à la fin d'une ligne il faut utiliser la commande

ws

```
\italic{Toto:}\ws{0.5}présent.
```

Les nouvelles lignes sont automatiquement insérées quand le texte est trop long pour tenir dans la ligne courante

Pour forcer une nouvelle ligne on doit utiliser la commande

newline

et

`\`

est synonyme de *newline*

5.5 Alignements

Quatre modes d'alignement

left, center, right et *justify*

`\left{Left mode active}\newline`

`\right{Right mode active}`

`\newline\center{Center mode active}`

5.6 Subscripts et superscripts

En mode *eqn* les sub et sup peuvent être ajoutés au mot précédent dans la phrase

`\eqn{x^y} + Vecteur_{indice}`

5.7 Parenthèses

Trois types de parenthèses peuvent être placées autour des parties de texte

`\eqn{\par{()}{entre parenthèses}\newline`

`\eqn{\par{[]}{entre crochets}\newline`

`\eqn{\par{}}{entre accolades}\newline`

Le premier argument de la commande *par* est les parenthèses voulues

Elles peuvent être mélangées

`()`

$$\} [$$

Le second argument spécifie le contenu de la parenthèse

A noter qu'il faut toujours être dans le mode *eqn* mais on peut tricher en incluant des commandes *txt* dans les commandes de parenthèses

5.8 Tableaux

La commande

tabular

sert à faire des tableaux

Le premier argument est le nombre de colonnes, l'alignement du contenu dans les colonnes, l'espacement vertical et les lignes verticales optionnelles entre colonnes

Tous les arguments suivants spécifient les lignes

5.9 Racines carrées

La commande

sqrt

prend un argument au minimum, le contenu de la racine

Optionnellement un second argument est la puissance de la racine

$$\backslash eqn\{sqrt\{x+y\}$$

$$\backslash eqn\{sqrt\{\pi\}\{3\}$$

5.10 Fractions

Pour construire des fractions où le haut et le bas sont séparés par une ligne horizontale il faut utiliser la fonction

frac

Elle prend deux arguments, le contenu du haut et le contenu du bas

La taille de la police est implicitement de 0.9 dans une fraction mais elle peut être adaptée en utilisant la commande

scale

$\backslash\text{eqn}\{\frac{a}{c+d}\}$

5.11 Chapeaux

On peut mettre des chapeaux, barres, tildes, sur du texte avec les commandes

hat, widehat, bar, widebar, et widetitle

Les variantes de

wide

se modulent avec la taille du contenu

5.12 Couleurs

Les textes peuvent être de couleur arbitraire

Un nombre fixe de couleurs est connu par le système

red, green, blue, magenta, yellow, cyan, black, white, grey, gray (deux synonymes)

Des couleurs particulières peuvent être définies par des commandes

Le défaut est

black

Black text, $\backslash\text{red}\{\text{Red text}\}$, $\backslash\text{green}\{\text{Green text}\}$

Couleurs particulières

Des couleurs particulières peuvent être ajoutées en utilisant la commande

newcolor

Elle prend un argument qui est une phrase contenant le nom de la couleur, le rouge, le vert et le bleu, séparés par un blanc

Une fois qu'une couleur est définie elle peut être utilisée comme n'importe quelle couleur

5.13 Commandes personnelles

Si le cerveau veut faire la même chose un grand nombre de fois, il peut être utile pour lui de créer une commande personnelle ou macro

Une commande personnelle définit un mot fixe d'entrée et des arguments variables

Un exemple serait si le cerveau veut toujours afficher les multienectages en bold

Le cerveau pourrait définir une commande

men

(*multienjectage*)

et l'utiliser chaque fois qu'on veut afficher quelque-chose comme un multiéjectage

Les commandes personnelles peuvent grandement simplifier le travail d'entrée, rendre l'entrée plus lisible, et changer facilement ce qu'elles font par la suite

Voici un exemple de définition et usage d'une commande personnelle

`\newcommand{\men}`

`{\bold{I}}`

`\eqn{\men{C}\op\men{a}\op\men{b}}`

Le premier argument de *newcommand* est le nom de la nouvelle commande à savoir

men

Le second argument est ce que la commande doit faire

Dans le second argument une barre inversée suivie d'un nombre positif *i* sera remplacé par le *i*-ème argument de la nouvelle commande

A noter que la commande *eqn* est incluse dans le commandes personnelle pour forcer le mode *eqn* chaque fois que la commande est utilisée

Ceci est similaire à

insuremath

en Latex

Les commandes sans argument sont aussi possibles en n'utilisant aucun nombre derrière la barre inversée dans le texte de commande personnelle

Si une commande est déjà définie, une *newcommand* pour elle écrasera sa valeur courante

La redéfinition d'une commande prédéfinie comme *txt* ou *par* est possible mais non recommandée

5.14 Symboles spéciaux

Quelques symboles spéciaux de la multilogique sont disponibles et d'autres peuvent être facilement ajoutés

Actuellement on a les symboles suivants sont accessibles par des commandes

Commande	Symbole
<i>gp</i>	<i>demi espace (*)</i>
<i>op</i>	^
<i>ip</i>	.
<i>lc</i>	_/
<i>rc</i>	_
<i>cp</i>	<i>x</i>
<i>infty</i> ou <i>inf</i>	∞

Tableau des symboles de commandes multilogiques

L'accès à des symboles spéciaux grècs a été discuté dans 5.2

La plupart des caractères ASCII qui ne sont pas des nombres ou des lettres peuvent aussi être atteints en tapant une barre inversée suivie par le caractère

Pour certains caractères, c'est la seule manière de les produire puisqu'ils sont réservés, par exemple la barre inversée