

# METAPOST

raconté aux piétons

Yves SOULET

Ce petit manuel pour METAPOST a été publié dans les Cahiers GUTenberg,  
n° 52-53, octobre 2009 (épuisé).



# Introduction

Lors de la publication du *Manuel de prise en main pour TikZ* (Cahiers GUTenberg, n° 50, avril 2008), il y a eu une assez longue discussion sur la liste GUT où l'on a pu constater que beaucoup d'utilisateurs de L<sup>A</sup>T<sub>E</sub>X souhaitaient savoir quel était le meilleur outil pour la production de figures de qualité irréprochable (et bien entendu avec des lettrages composés « à la L<sup>A</sup>T<sub>E</sub>X »). D'autres, qui utilisent couramment un ou plusieurs de ces outils ont expliqué les avantages et les inconvénients des uns et des autres. L'un de ces outils se distingue bien des autres : on veut tracer une tangente à un cercle passant par un point extérieur donné ; PSTricks et TikZ tracent cette tangente si on leur donne les coordonnées du point de contact ; METAPOST calcule les coordonnées de ce point de contact si « on sait le lui demander » ; là se trouve *la* différence !

Son histoire commence au début des années 80 où Donald Knuth conçoit METAFONT, un langage destiné à produire des caractères ; c'est ainsi qu'il a pu créer les fontes CM. Ce langage permet de réaliser le dessin des caractères et, dans une deuxième étape, de produire sa rasterisation (c'est-à-dire le choix des pixels à noircir pour obtenir le meilleur rendu à l'impression).

Par la suite apparaît le langage PostScript et les imprimantes professionnelles sont équipées d'un processeur (l'interpréteur PostScript) qui traite parfaitement le problème de la rasterisation en prenant en compte les caractéristiques spécifiques de la machine pour laquelle il a été conçu (résolution et beaucoup d'autres paramètres). En conséquence, au début des années 90, est créé METAPOST qui est un langage dérivé de METAFONT ; quelques primitives ont été ajoutées, d'autres ont été abandonnées. Au lieu de produire des objets graphiques sous forme d'une liste de coordonnées de pixels à noircir, METAPOST produit le code PostScript de ces objets (d'où l'abandon de toutes les primitives concernant la rasterisation).

En 1992, John Hobby publie un document, *A User's Manual for METAPOST*, dont la traduction française *Un manuel de l'utilisateur pour METAPOST* par Jean-Côme Charpentier et Pierre Fournier, est publiée à son tour par les Cahiers GUTenberg, n° 41, novembre 2001. Ce manuel est de lecture assez difficile pour les utilisateurs de L<sup>A</sup>T<sub>E</sub>X qui ne sont pas familiers avec le langage de la discipline informatique. Un manuel du type « pour les piétons<sup>(1)</sup> » devrait aider et encourager les utilisateurs potentiels dans leur début en METAPOST : c'est le but du présent document.

---

<sup>(1)</sup> L'auteur de ces lignes avait été émerveillé par le petit livre *Feynman diagrams for pedestrians* qui lui avait facilité ses débuts en électrodynamique quantique.

Ces quelques pages n'ont pas la rigueur du manuel de John Hobby ; elles n'ont pas la prétention de les remplacer ; en effet, certaines spécificités du langage parmi les plus délicates ne sont pas abordées (ou sont simplement citées avec une petite « illustration »). Elles présentent, dans un langage simple et accompagné d'exercices avec leur code complet, l'essentiel permettant au débutant de réaliser des figures assez complexes. Toutes les primitives et toutes les macros (définies dans le fichier `plain.mp` et dans quelques autres fichiers cités en temps utile) sont illustrées au fur et à mesure par des exemples conçus pour faire découvrir progressivement la structure du langage.

L'originalité de ce document réside dans le fait que le débutant est invité à faire les exercices et en voir les résultats immédiatement avec une visionneuse PostScript (GSVIEW, GHOSTVIEW, etc.).

Les références [xxx] se rapportent aux pages du n° 41 des Cahiers GUTenberg où se trouve, en plus du manuel METAPOST déjà cité et de l'article du même auteur concernant la production de graphiques (extension GRAPH), un petit article de Fabrice Popineau consacré au côté pratique de l'utilisation de METAPOST. Les références du type [M xxx] se rapportent à des pages du livre de référence de METAFONT<sup>(2)</sup>.

Les trois premiers chapitres sont conçus pour permettre aux débutants d'acquérir un savoir faire suffisant pour réaliser des figures relativement complexes. Le quatrième chapitre est consacré à la « machinerie » METAPOST, c'est-à-dire à la structure du langage avec ses variables, ses opérateurs, ses boucles conditionnelles, ses commandes de bifurcations et aux règles de construction des macros utilisateur. Le cinquième chapitre expose la construction des nœuds et des liaisons entre ces nœuds en vue de produire des organigrammes, des algorithmes, etc. Le sixième chapitre est destiné à la construction de graphiques, ce que les physiciens appellent les courbes, à l'aide d'une extension de fichier `graph.mp` ; le point fort de cette extension est une mise à l'échelle automatique des données pour que le graphique ait les dimensions imposées par avance. Quelques macros de l'auteur de ces lignes permettent de « relooker » ces graphiques tout en conservant les avantages de l'extension citée. Le septième et dernier chapitre est très court ; il explique comment ajouter des lettrages « à la L<sup>A</sup>T<sub>E</sub>X » dans les figures et propose une méthode de travail pour les produire.

L'index, abondant, contient toutes les primitives et toutes les macros avec les numéros de page où elles sont introduites puis utilisées ; il ne reprend pas les titres de la table des matières ; au lecteur d'utiliser l'index et la table pour ses recherches.

METAPOST mérite bien plus que ce petit document qui ne veut qu'être une aide pour débiter avec une progression motivante ! L'objectif de ces pages est de permettre à tous ceux qui sont totalement absorbés par leurs tâches quotidiennes d'acquérir une certaine pratique de METAPOST sans y consacrer trop de temps. Après ce début, ils auront la capacité nécessaire pour rechercher et évaluer les extensions METAPOST disponibles qui conviennent à leur objectif.

---

<sup>(2)</sup> Donald Knuth, METAFONTBook, Addison Wesley, 1986.

# Table des matières

Introduction	iii
CHAPITRE 1 Coordonnées et lettrages	1
1.1 Coordonnées cartésiennes	2
1.2 Coordonnées polaires	3
1.3 Lettrage direct	4
CHAPITRE 2 Tracé de lignes brisées	5
2.1 Syntaxe de base du tracé	5
2.2 Options de tracé	6
2.2.1 Epaisseur du trait	7
2.2.2 Terminaisons et jonctions de traits	8
2.2.3 Pointillés et traitillés	8
2.2.4 Flèches	9
2.2.5 Couleurs	10
2.3 Compléments	11
2.3.1 Résolution d'équations linéaires	11
2.3.2 Intersection de droites	11
2.3.3 Précisions de certains points de syntaxe	12
2.3.4 Préambule du fichier des figures	13
CHAPITRE 3 Tracé de lignes courbes	15
3.1 Un tracé élémentaire pour débiter	15
3.2 Courbes de Bézier cubiques	16
3.3 Différentes possibilités de tracés de courbes	17
3.3.1 Tracé avec les seuls points guides	17
3.3.2 Tracé en fixant les tangentes aux points guides	17
3.3.3 Modification du tracé : tension et courbure	18
3.4 Gestion des courbes	19
3.4.1 Découpage des courbes	20
3.4.2 Intersections de courbes	20
3.4.3 Construction directe des zones d'intersection	22
3.4.4 Tangentes et normales aux courbes	23
3.5 Compléments	24
3.5.1 Cercles et arcs de cercles	24
3.5.2 Autres commandes concernant les courbes	25
3.5.3 Explications concernant des points de syntaxe	26

CHAPITRE 4	La machinerie METAPOST	29
4.1	Opérateurs	29
4.2	Variables	30
4.2.1	Type nombre	31
4.2.2	Type paire	31
4.2.3	Type couleur	32
4.2.4	Type transformation	32
4.2.5	Type chemin	34
4.2.6	Type chaîne	34
4.2.7	Type booléen	35
4.2.8	Type dessin	35
4.2.9	Type plume	36
4.3	Variable et équations; variables internes et affectations	36
4.3.1	Variables (ordinaires)	36
4.3.2	Variables internes	37
4.3.3	Remarques concernant les affectations	37
4.4	Commandes et macros	38
4.4.1	Commandes	38
4.4.2	Macros de base	39
4.4.3	Construction de macros	39
4.5	Boucles et tests	40
4.5.1	Boucles	40
4.5.2	Tests	41
CHAPITRE 5	Boîtes et liaisons : organigrammes et algorithmes	43
5.1	Boîtes rectangulaires	43
5.1.1	Création de la boîte	44
5.1.2	Tracé de la boîte	44
5.1.3	Améliorations : fond et contour	45
5.2	Autres formes : arrondies, circulaires, etc.	46
5.3	Macros élémentaires à la demande	47
5.4	Liaisons entre les boîtes	49
5.4.1	Extrémités des liaisons	49
5.4.2	Liaisons à un seul segment	50
5.4.3	Liaisons à deux segments	51
5.4.4	Liaisons à trois segments et plus	52
5.5	Mettre des labels (ou lettrages) sur les liaisons	53
CHAPITRE 6	Tracé de courbes	55
6.1	Tracés par défaut	55
6.2	Types de tracés disponibles	57
6.3	Domaines de variation et types de coordonnées	60
6.4	Cadre (ou axes) et graduations (tiks et labels)	61
6.5	Légende des axes et labels supplémentaires	63
6.6	Difficultés dues aux grands nombres	64
6.7	Tracé de courbes METAPOST « relookées »	66

CHAPITRE 7 METAPOST et T <sub>E</sub> X	69
7.1 Etapes du lettrage des figures	69
7.1.1 Production des fichiers <b>tex</b> et <b>dvi</b> des lettrages	70
7.1.2 Production du fichier <b>mpx</b> des lettrages et des fichiers de figures	70
7.1.3 Automatisation	71
7.2 Lettrage en L <sup>A</sup> T <sub>E</sub> X de quelques figures	72
7.3 Lettrage en L <sup>A</sup> T <sub>E</sub> X de quelques graphiques	73
7.4 METAPOST et T <sub>E</sub> X, tout à la fois!	75
Conclusion	77
Index	79





## Coordonnées et lettrages

Ce premier (et tout petit) chapitre est destiné à introduire les notations de base pour représenter les points du plan avec leurs coordonnées. Dans les premiers exemples, on se donne des points et on les relie entre eux par des segments que l'on trace avec la commande `draw`, commande qui sera reprise en détail au chapitre 2. Une grille, `axespapiermilli`, d'un pas de 1 mm et des axes permettent de visualiser facilement le résultat des commandes. Dès ce premier chapitre, on commence à exploiter toutes les propriétés de METAPOST qui est un langage informatique à part entière avec variables, opérateurs, commandes, fonctions prédéfinies et définies par l'utilisateur, boucles de commandes, etc.

Tout au long de ce petit manuel, le lecteur est invité à refaire les exemples, à les modifier et à en imaginer d'autres. Pour faciliter ce travail, il est parfois important de pouvoir letter les figures des exemples : pour cela, on délaisse provisoirement T<sub>E</sub>X-L<sup>A</sup>T<sub>E</sub>X au profit d'une propriété de METAPOST qui permet de visualiser directement les figures, y compris les lettrages (composés avec une unique fonte, ce qui ne gêne en rien lorsque ces lettrages sont uniquement destinés à repérer des parties des figures). La méthode pour obtenir ces lettrages est exposée à la fin de ce chapitre. On conseille de procéder ainsi (au lecteur de personnaliser la méthode s'il le souhaite!) :

- Faire un répertoire de travail, `mp` par exemple.
- Créer un fichier `figa.mp` dans ce répertoire qui contiendra un préambule (voir sect. 2.3.4) et le code des figures du chapitre 1.

```
% figa.mp
... Preambule : affectations et definitions
... pour toutes les figures de ce fichier
%
beginfig(1);
... code de la figure
endfig;
%
beginfig(2);
... code de la figure
endfig;
%
... etc.
end.
```

- Compiler avec la commande `mp figa`  
Ce traitement crée les fichiers POSTSCRIPT des figures : `figa.1`, `figa.2`, etc. (cf. les arguments des macros `beginfig`). On peut intégrer ces figures dans un document  $\text{T}_{\text{E}}\text{X}-\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$  (ce qu'on ne fera pas pour le moment, comme annoncé ci-dessus). Pour ce faire, il est nécessaire de renommer les fichiers (par exemple : `ren figa.1 figa1.eps`, etc.) ou d'utiliser la déclaration `\DeclaregraphicsRule{*}{eps}{*}{*}` qui force `\includegraphics` à considérer les fichiers d'extension inconnue comme des fichiers `.eps`.
- Visualiser directement les figures `figa.1`, `figa.2`, etc. avec `GSVIEW` : c'est l'idéal pour le débutant qui peut voir immédiatement la figure.
- On peut aussi transformer les fichiers de figure du format PostScript au format PDF avec les commandes :  
`epstopdf --out=figa1.pdf figa.1`, etc.

La création de figures avec lettrages composés en  $\text{T}_{\text{E}}\text{X}-\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$  sera exposée au chapitre 7. Maintenant, tout est prêt pour faire les premiers pas.

## 1.1 Coordonnées cartésiennes

Le point du plan de coordonnées  $x = a$  bp et  $y = b$  bp est représenté par la paire `(a,b)` puisque le bp (point POSTSCRIPT) est l'unité par défaut [19]. On peut aussi utiliser les unités reconnues par  $\text{T}_{\text{E}}\text{X}$  [20] : cm, mm, in, pt (point d'impression), etc. Par exemple, le point de coordonnées  $x = 8$  mm et  $y = 5$  mm est représenté par la paire `(8mm,5mm)`. On convient d'utiliser un facteur d'échelle `u` que l'on peut définir à l'aide des unités reconnues. Pour ce manuel, on prend `u=1mm`. Le point  $x = 8$  mm et  $y = 5$  mm est alors représenté par la paire `(8u,5u)`.

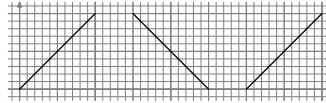
REMARQUE. Lorsqu'une coordonnée est nulle, on note 0 (le bp est donc pris pour unité, ce qui n'a aucune importance). Par contre, une coordonnée de valeur `u` doit être notée `1u` [20] (car `1u` est en réalité une abréviation de `1*u` qui correspond à `1 u mul` en POSTSCRIPT : l'omission du 1 déclencherait une erreur puisqu'il manquerait un élément dans la pile). Ce facteur d'échelle permet de réduire une figure [20] (en le diminuant) sans que cela affecte l'épaisseur des traits, le diamètre des points, etc. que l'on exprime en pt).

Voici un exemple de tracé avec la commande `draw` [19] où l'on utilise les diverses représentations possibles des extrémités des segments. Pour simplifier, on place dans le préambule du fichier `figa.mp` les affectations nécessaires et l'appel du fichier `macutil.mp` (voir sect. 2.3.4) contenant quelques définitions très commodes, en particulier les définitions des commandes de tracé des axes et de la grille d'un pas de 1 mm (l'ancien papier millimétré!). Les caractères `%<` dans le code d'une figure signifient que ce qui suit sur la ligne est redondant, c'est-à-dire déjà défini plus haut (dans le préambule du fichier, dans le fichier `macutil.mp` ou dans le code d'une figure précédente), mais doit être utilisé pour traiter la figure seule.

```

beginfig(1);
<u=1mm;
axespapiermilli(-1.5,-1.5,41.5,11.5)
draw(0,0)--(28.346,28.346);
draw (15u,10u)--(25u,0);
draw (1.1811in,0) -- (4cm,28.453pt)
endfig;

```

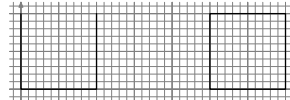


Comme annoncé plus haut, on écrit un autre exemple en introduisant les *variables* du type *paire*  $z_1, z_2$ , etc., les variables du type *chemin*  $p_1, p_2$ , etc., et la *transformation* `shifted(25u,0)` produisant une translation de l'élément précédent de 25 mm suivant les  $x$  et de 0 mm suivant les  $y$ . On remarque que les variables paires  $z[]$  (c'est à dire  $z_1, z_2$ , etc.) ne sont pas déclarées; c'est un cas spécial : elles sont prédéclarées ainsi que les variables du type *nombre*  $x[]$  et  $y[]$ . Les variables chemin  $p[]$  sont ensuite déclarées; les crochets dans ces déclarations ont toujours le même sens : dans le cas de  $p$  par exemple, il s'agit d'une variable tableau à une dimension dont les éléments se nomment  $p_1, p_2$ , etc. On note encore que l'on a ajouté `--cycle` pour fermer le chemin et non pas `--z4` qui n'est pas équivalent (ce choix est fondamental, en particulier pour des traits de forte épaisseur et pour des chemins fermés destinés à être remplis).

```

beginfig(2);
axespapiermilli(-1.5,-1.5,36.5,11.5)
z1=(0,0);z2=(10u,0);z3=(10u,10u);z4=(0,10u);
path p[];p1=z4--z1--z2--z3;p2=p1--cycle;
draw p1;
draw p2 shifted(25u,0);
endfig;

```



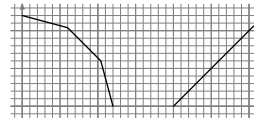
## 1.2 Coordonnées polaires

Il n'y a pas de représentation des points du plan prévue en coordonnées polaires; néanmoins, ce type de représentation est très utile, par exemple, pour définir une droite passant par un point donné et faisant un angle donné avec l'axe des  $x$ . On introduit dans l'exemple qui suit la macro `polar` pour représenter un point de coordonnées polaires  $r, w$  par `polar(r,w)` : les points  $z_5$  et  $z_6$  définissent bien une droite passant par le point  $z_5$  et faisant un angle de  $45^\circ$  avec l'axe des  $x$ .

```

beginfig(3);
axespapiermilli(-1.5,-1.5,31.5,13.5)
<def polar(expr r,w)=(r*cosd w,r*sind w)enddef;
z1=polar(12u,0);z2=polar(12u,30);
z3=polar(12u,60);z4=polar(12u,90);
draw z1--z2--z3--z4;
z5=polar(20u,0);z6=z5+polar(15u,45);
draw z5--z6;
endfig;

```



Dans l'exemple ci-dessus, la macro introduite est une macro à deux paramètres dont le principe est connu des utilisateurs de  $\text{T}_\text{E}_\text{X}$ ; la syntaxe en est différente et plus exigeante. Tout cela sera approfondi plus loin.

### 1.3 Lettrage direct

La commande [44] :  
`label.xxx("aaa",zz);`  
 compose la chaîne `aaa` au point de référence `zz` et la déplace dans la direction `xxx` à la distance `labeloffset` (*variable interne* du type nombre dont la valeur par défaut est 3 bp); les valeurs possibles de la direction sont : `top`, `rt`, `bot`, `lft`, `urt`, `lrt`, `llft` et `ulft`  
 Si aucune direction n'est donnée, la chaîne est centrée au point de référence.

Le choix de la fonte se fait en donnant des valeurs aux variables internes `defaultfont` (de type *chaîne* et de valeur par défaut `cmr10`) et `defaultscale` (de type nombre et de valeur par défaut 1). Dans une première lecture, on peut ignorer la suite de ce paragraphe. Il faut que la fonte choisie ait un fichier métrique (fichier `.tfm` contenant toutes les dimensions des caractères) accessible à `METAPOST`; ensuite, si l'on veut voir directement la figure avec les lettrages, il faut que les fichiers contenant les dessins des lettres soient disponibles pour `GSVIEW` si on veut le résultat à l'écran et disponibles pour l'interpréteur `POSTSCRIPT` de l'imprimante si l'on veut le résultat sur le papier. On choisit par exemple la fonte `phvr8r`, dont le nom `POSTSCRIPT` est `Helvetica`. Cette fonte est une des 35 fontes disponibles sur les imprimantes dites `POSTSCRIPT` et son `.tfm` se trouve sur toutes les installations `TEX-LATEX` à jour. D'abord, `METAPOST` va lire le fichier `phvr8r.tfm` pour positionner les caractères du lettrage, ensuite, si l'on a donné la valeur 2 à la variable interne `prologues`, il va chercher dans l'installation `TEX-LATEX` le fichier `uhv.map` dans lequel il va trouver le nom `POSTSCRIPT` de la fonte choisie; on pourra éditer les fichiers des figures ayant un lettrage et vérifier qu'ils contiennent bien la ligne

```
/phvr8r Helvetica def
```

Ainsi, `GSVIEW` et l'interpréteur `POSTSCRIPT` de l'imprimante disposent du nom `POSTSCRIPT` de la fonte pour montrer les caractères.

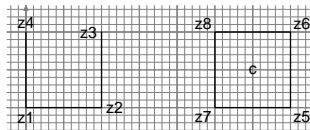
On écrit donc dans le préambule du fichier `figa.mp` (préambule répété pour les fichiers `figb.mp`, `figc.mp` et `figd.mp` et que l'on peut trouver à la section 2.3.4 accompagné de différentes explications) :

```
prologues:=2; labeloffset:=1.7pt; (lettrage plus proche)
```

```
defaultfont="phvr8r"; defaultscale:=0.6; (taille plus petite)
```

On reprend la deuxième figure en y rajoutant les commandes décrites ci-dessus.

```
beginfig(4);
axespapiermilli(-2.5,-3.5,38.5,13.5)
z1=(0,0);z2=(10u,0);z3=(10u,10u);z4=(0,10u);
z5=(5u,5u);path p[];p1=z4--z1--z2--z3;
p2=p1--cycle;draw p1;
label.rt("z2",z2);label.lft("z3",z3);
label.bot("z1",z1);label.top("z4",z4);
draw p2 shifted(25u,0);
label.lrt("z2",z2 shifted (25u,0));
label.urt("z3",z3 shifted (25u,0));
label.llft("z1",z1 shifted (25u,0));
label.ulft("z4",z4 shifted (25u,0));
label("c",z5 shifted (25u,0));
endfig;
```



## Tracé de lignes brisées

Dans ce chapitre, on trace des lignes brisées, ouvertes ou fermées, avec toutes les options possibles de tracé : épaisseur du trait, couleur du trait, forme de la jonction des segments élémentaires, etc. Ce choix est motivé par le désir de simplifier l'exposé et de permettre au lecteur d'utiliser le plus tôt possible la richesse contenue dans METAPOST. La plupart de ces options se retrouveront pour les lignes courbes. Bien sûr, on va continuer à exploiter toutes les propriétés du langage en écrivant quelques macros.

### 2.1 Syntaxe de base du tracé

La syntaxe pour la définition d'une ligne brisée passant par la suite de points  $z_1, z_2, \dots, z_n$ , est :

`p=z1--z2--etc--zn;` (ligne ouverte) [19],  
`p=z1--z2--etc--zn--cycle;` (ligne fermée) [70].

Il y a ensuite les commandes utilisant ce chemin :

`draw p options;` pour tracer le chemin [51],  
`undraw p options;` pour effacer le chemin [70],

et, si le chemin est fermé :

`fill p options;` pour remplir le chemin [51],  
`unfill p options;` pour effacer l'intérieur du chemin [54],

`clip currentfile to p;` pour découper et ne conserver que la partie de la figure courante intérieure au chemin `p` [75].

`options` représente d'éventuelles options qui sont détaillées à la section suivante.

On évitera, en débutant, l'utilisation de `filldraw` et `unfilldraw` plus délicates à utiliser. Le format `metafun` (distribution de `CONTEXT`) permet d'ombrer l'intérieur d'un chemin [172].

Sur l'exemple suivant (fichier de figures `figb.mp`) on trouve dans l'ordre :

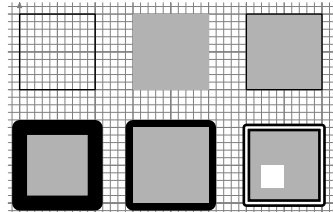
- un carré tracé, un carré rempli, un carré tracé et rempli ;
- un carré d'abord rempli puis tracé, ensuite le même carré d'abord tracé puis rempli afin de montrer la conséquence de l'inversion des opérations (nettement visible avec une forte épaisseur de trait) ;

- une utilisation astucieuse de la commande `undraw` pour tracer un trait double et une utilisation de la commande `unfill` pour gommer un petit carré déjà coloré afin, par exemple, de placer un lettrage.

Les commandes `undraw` et `unfill` correspondent à un effacement et donc *ne sont pas équivalentes* aux commandes `fill` et `draw` avec la couleur blanche.

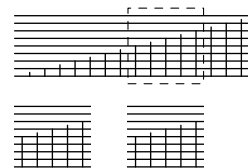
On introduit dans l'exercice suivant la macro `cl`, reportée dans `macutil.mp` (cf. sect. 2.2.5 pour la définition) ; son argument est l'intensité du gris (de 0 à 1).

```
beginfig(1);axespapiermilli(-1.5,-1.5,42.5,26.5)
%<def cl(expr s)=withcolor (s,s,s) enddef;
z1=(0,0);z2=(10u,0);z3=(10u,10u);z4=(0,10u);path p;p=z1--z2--z3--z4--cycle;
draw p shifted(0,15u);
fill p shifted(15u,15u) cl(0.7);
fill p shifted(30u,15u) cl(0.7);
draw p shifted(30u,15u);
fill p shifted(0,0) cl(0.7)
draw p shifted(0,0)withpen pencircle scaled2mm;
draw p shifted(15u,0)
      withpen pencircle scaled2mm;
fill p shifted(15u,0) cl(0.7);
fill p shifted(30u,0) cl(0.7);
draw p shifted(30u,0)withpen pencircle scaled3pt;
undraw p shifted(30u,0)withpen pencircle scaled1pt;
unfill p scaled 0.3 shifted (32u,2u);
endfig;
```



Enfin un exemple de la commande `clip` qui détoure ou découpe la partie de la figure faite avant cette commande et intérieure à un contour donné. La figure déjà faite est, à tout instant, la valeur de la variable interne `currentpicture`. Pour conserver cette figure initiale avant de la réduire par découpe, on la sauvegarde dans la variable du type *dessin* nommée `dessininitial` préalablement déclarée ; on fait la découpe en bas à droite ; on sauvegarde la découpe dans une autre variable ; on montre, en les déplaçant, la figure initiale (avec le chemin de découpe en traitillé) en haut et la copie de la découpe en bas à gauche.

```
beginfig(2);
z1=(0,0);z2=(10u,0);z3=(10u,10u);z4=(0,10u);
%<p=z1--z2--z3--z4--cycle;
picture dessininitial,dessindecoupe;
for i=1 upto 16:
  draw(i*2u,1u)--(i*2u,i*0.5u+1u); endfor
for i=1 upto 9:draw(0,i*u)--(32u,i*u); endfor
dessininitial=currentpicture;
clip currentpicture to p shifted(15u,0);
dessindecoupe=currentpicture;
draw dessininitial shifted(0,12u);
draw dessindecoupe shifted(-15u,0);
draw p shifted(15u,12u) lw(0.4) dashed evenly;
endfig;
```



## 2.2 Options de tracé

On va passer en revue les options de tracé en donnant des exemples. La plupart de ces options restent valables pour les lignes courbes.

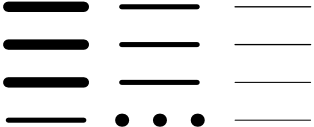
### 2.2.1 Épaisseur du trait

Cette épaisseur est choisie par l'option [70] :  
`withpen pencircle scaled n pt` ( $n$  est un nombre)  
 placée à la fin de la commande `draw` ou bien par la déclaration [20] :  
`pickup pencircle scaled n pt;`  
 placée en avant de la commande `draw` concernée ou bien encore par la déclaration générale [70] :  
`drawoptions(withpen pencircle scaled n pt,option de couleur,...);`  
 toujours placée avant la commande `draw` et qui peut inclure d'autres options que l'option d'épaisseur. Par défaut, cette épaisseur vaut 0.5 bp (voir [19] pour la différence entre le pt et le bp; elle peut être bien souvent négligée).

```

beginfig(3);
z1=(0,0);z2=(10u,0);path p; p=z1--z2;          % ligne 1
draw p shifted(0,15u) withpen pencircle scaled 4pt; %ligne 1
pickup pencircle scaled 2pt;draw p shifted(15u,15u);
pickup defaultpen;draw p shifted(30u,15u);
drawoptions(withpen pencircle scaled 4pt);
draw p shifted(0,10u);                          %ligne 2
draw p shifted(15u,10u)
      withpen pencircle scaled 2pt;
drawoptions();draw p shifted(30u,10u);
def setlinewidth(expr s)=
  pickup pencircle scaled s enddef; %ligne 3
setlinewidth(4pt);draw p shifted(0,5u);
draw p shifted(15u,5u)
      withpen pencircle scaled 2pt;
setlinewidth(0.5pt);draw p shifted(30u,5u);
%<def lw(expr s)=withpen pencircle scaled s enddef;
draw p lw(2pt);                                %ligne 4
%<def drawpt(expr z,s)=draw z withpen pencircle scaled s enddef;
drawpt((15u,0),5pt); drawpt((20u,0),5pt); drawpt((25u,0),5pt);
def drawthin(expr p)=draw p withpen pencircle scaled 0.3 pt enddef;
drawthin((30u,0)--(40u,0));
endfig;

```



Dans l'exemple ci-dessus, on a testé d'abord l'option `withpen`. On a aussi testé les deux déclarations, `pickup ...` et `drawoptions`, et on a vérifié que l'on peut désactiver leur effet en appelant la plume par défaut, `defaultpen`, ou en donnant une liste d'options vide à `drawoptions` (ligne 1). On a aussi vérifié que, lorsque l'une de ces déclarations est active, on peut encore tracer un trait d'une épaisseur différente de celle fournie par cette déclaration grâce à l'option `withpen` placée en fin de la commande `draw` (ligne 2).

Ensuite, pour simplifier la saisie, on a défini la macro `setlinewidth` qui abrège la première déclaration, on l'a testée et on a aussi testé la macro `lw` (reportée dans le fichier `macutil.mp`) abrégeant l'option `withpen` (ligne 3).

Enfin, puisqu'en traçant un segment de longueur nulle (commande `draw` avec une seule paire) on obtient un cercle plein (un point) dont le diamètre est l'épaisseur de trait choisie [21], on a introduit et testé la macro `drawpt` (reportée dans `macutil.mp`) (ligne 4). Enfin, puisque dans une figure ou une suite de figures d'un même ouvrage, on n'aura qu'un très petit nombre de sortes de traits (pour des raisons de qualité pédagogique), on a tout intérêt à définir les

quelques macros correspondantes pour alléger la saisie : par exemple la macro `drawthin` qui pourrait aussi contenir une option de couleur.

REMARQUE. Pour terminer cette sous-section, on explique le choix du vocabulaire de l'option `withpen` et de la déclaration `pickup`. METAFONT a été conçu pour travailler comme un humain qui trace des traits avec des feutres qu'il fait glisser sur le papier. Pour augmenter l'épaisseur du trait, il faut augmenter le diamètre de la pointe du feutre d'où `scaled n pt`. Mais le concepteur a aussi prévu l'utilisation de feutres dont la section de la pointe n'est pas forcément circulaire : ainsi, avec une plume elliptique de grand axe parallèle à l'axe des  $x$ , on conçoit aisément qu'un déplacement suivant l'axe des  $y$  donne un trait plus épais qu'un déplacement suivant l'axe des  $x$  ; cela permet le tracé de caractères calligraphiques ... et ceci se retrouve dans METAPOST [73].

### 2.2.2 Terminaisons et jonctions de traits

Les terminaisons et jonctions des traits sont choisies par l'intermédiaire de deux variables internes.

Pour la terminaison, la variable et ses trois valeurs possibles sont [67] :  
`linecap:=butt`; ou `squared`; ou `rounded`; (défaut)

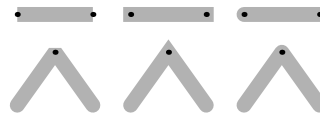
On constate que les deux dernières possibilités conduisent à un dépassement aux extrémités (représentées ici par un point).

Pour la jonction des traits, on dispose d'une autre variable interne ayant aussi trois valeurs possible [68] :

`linejoin:=beveled`; ou `mitered`; ou `rounded`; (défaut)

Après un changement provisoire, il ne faut pas oublier de reprendre la valeur par défaut. Il y a une autre possibilité pour un tel changement [78] ; elle n'abrège pas la saisie et donc on n'en dira pas davantage. Voici un exemple des six possibilités (l'ordre de présentation assure le retour aux valeurs par défaut).

```
beginfig(4);
path p; p=(0,0)--(10u,0);
%<def cl(expr s)=withcolor (s,s,s) enddef;
drawoptions(lw(2mm)cl(0.7));
linecap:=butt;draw p shifted(0u,12u);
linecap:=squared;draw p shifted(15u,12u);
linecap:=rounded;draw p shifted(30u,12u);
path p; p=(0,0)--(5u,7u)--(10u,0);
linejoin:=beveled;draw p;
linejoin:=mitered;draw p shifted(15u,0);
linejoin:=rounded;draw p shifted(30u,0);
drawoptions(lw(2pt));
drawpt((0,12u),2pt);drawpt((15u,12u),2pt);drawpt((30u,12u),2pt);
drawpt((10u,12u),2pt);drawpt((25u,12u),2pt);drawpt((40u,12u),2pt);
drawpt((5u,7u),2pt);drawpt((20u,7u),2pt);drawpt((35u,7u),2pt);
endfig;
```



### 2.2.3 Pointillés et traitillés

Les pointillés sont obtenus avec l'option [64] :

`dashed withdots scaled n`

placée en fin de la commande `draw` ( $n$  est un nombre).



Les traitillés s’obtiennent avec l’option [64] :

`dashed evenly scaled n`  
aussi placée en fin de la commande `draw` (`n` est encore un nombre dont on va tester l’action sur l’exemple suivant).

L’exemple donne trois lignes de pointillés suivies de trois lignes de traitillés avec différents espacements. Dans certains cas, on peut être amené à rechercher la perfection : on voit sur l’exemple que les traitillés commencent par un tiret ; à l’autre extrémité, il peut y avoir un blanc ou un tiret coupé ; pour assurer une certaine symétrie, on peut déplacer le début des pointillés ou des traitillés en ajoutant à l’option `shifted(d,0)` où `d` est une distance choisie par tâtonnement. La septième ligne reprend la sixième avec un décalage de 2,4 pt assurant la symétrie.

```
beginfig(5);
path p; p=(0,0)--(30u,0); drawoptions(lw(0.8));
draw p shifted(0,32u) % ligne 1
      dashed withdots scaled 0.5; .....
draw p shifted(0,28u) dashed withdots; .....
draw p shifted(0,24u)
      dashed withdots scaled 1.5; .....
draw p shifted(0,20u) dashed evenly; % ligne 4 -----
draw p shifted(0,16u) dashed evenly scaled 1.5; -----
draw p shifted(0,12u) dashed evenly scaled 3; -----
draw p shifted(0,8u) % ligne 7
      dashed evenly scaled 3 shifted(2.4pt,0); -----
draw p shifted(0,4u) dashed % ligne 8 -----
      dashpattern(on 6pt off 3pt on 3pt off 3pt); -----
def drawrule(expr q,s)=draw q dashed evenly
      scaled s enddef; drawrule(p,1.5); % ligne 9
endfig;
```

L’exemple montre aussi comment on peut définir un traitillé de style personnel avec la définition d’un motif spécifique, par exemple [65] :

```
dashpattern(on 6pt off 3pt on 3pt off 3pt)
```

qui donne, en se répétant indéfiniment, un traitillé dont le tiret de base a une longueur alternativement de 3 pt et de 6 pt (huitième ligne). Enfin, on propose et teste une macro donnant un traitillé afin d’abrégier la saisie (neuvième ligne).

## 2.2.4 Flèches

La construction des flèches par défaut est assez pauvre : on dispose d’un seul type de flèche et de trois commandes [69] :

```
drawarrow chemin options;
```

qui place une flèche à l’extrémité du chemin,

```
drawarrow reverse chemin options;
```

qui place une flèche inversée à l’origine du chemin et

```
drawdblarrow chemin options;
```

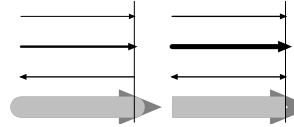
qui place une flèche à chaque extrémité du chemin.

Les flèches sont caractérisées par deux variables internes prédéfinies [69] : `ahlength` et `ahangle`. La première a du être fortement diminuée. Sur l’exemple qui vient, on trouve un test des trois commandes de tracé de flèches avec `linejoin:=mitered`; pour avoir des flèches bien pointues.

```

beginfig(6);
linejoin:=mitered; path p; p=(0,0)--(15u,0);
drawarrow p shifted(0,12u) lw(0.2pt); % ligne 1
drawarrow p shifted(20u,12u);
drawarrow p shifted(0,8u) lw(1pt); % ligne 2
drawarrow p shifted(20u,8u) lw(2pt);
drawarrow reverse p shifted(0,4u); % ligne 3
drawdblarrow p shifted(20u,4u);
drawarrow p lw(8pt) cl(0.5); % ligne 4
draw p lw(8pt) cl(0.5);
linecap:=butt;
drawarrow p shifted(20u,0)lw(8pt) cl(0.5);
draw p shifted(20u,0) lw(8pt) cl(0.5); linecap:=rounded;linejoin:=rounded;
draw (15u,-2u)--(15u,14u) lw(0.2pt); draw (35u,-2u)--(35u,14u) lw(0.2pt);
endfig;

```



Par construction même [69], la flèche dépasse l'extrémité du chemin; cette propriété est bien visible pour les chemins de forte épaisseur et est clairement montrée par les deux dernière flèches de l'exemple ci-dessus où la flèche entière est tracée en gris foncé puis surchargée par le chemin (un segment dans ce cas) en gris clair. Cette construction a aussi pour conséquence que la valeur `butt` de la variable `linecap` entraîne le même effet que la valeur `beveled` de la variable `linejoin` comme cela se découvre sur la flèche en bas à droite.

## 2.2.5 Couleurs

La couleur est gérée dans METAPost suivant le codage RGB. Une couleur est représentée par un triplet de trois nombres inférieurs à 1. Les couleurs prédéfinies sont `red`, `green`, `blue`, `white` et `black` (défaut) [34]; on peut définir toutes les couleurs que l'on veut, par exemple `yellow=(1,1,0)` définit le jaune. Les tracés et les remplissages sont colorés en utilisant l'option [51] :

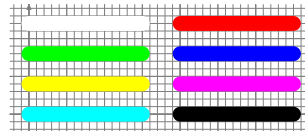
`withcolor couleur`

où `couleur` est le nom d'une couleur ou tout simplement un triplet définissant une couleur. L'exemple qui suit montre l'utilisation de cette option pour les tracés mais elle s'utilise de la même manière pour les remplissages.

```

beginfig(7);axespapiermilli(-2.5,-2.5,37.5,14.5);
path p; p=(0,0)--(15u,0);
color yellow,magenta,cyan;
yellow=(1,1,0);magenta=(1,0,1);cyan=(0,1,1);
drawoptions(withpen pencircle scaled 2mm);
draw p shifted(0u,12u) withcolor white;
draw p shifted(20u,12u) withcolor red;
draw p shifted(0,8u) withcolor green;
draw p shifted(20u,8u) withcolor blue;
draw p shifted(0,4u) withcolor (1,1,0);
draw p shifted(20u,4u) withcolor magenta;
draw p withcolor cyan;
draw p shifted(20u,0) withcolor black;
endfig;

```



On peut additionner, soustraire et multiplier par un nombre ces triplets : le résultat est automatiquement traité par METAPost pour que les trois nombres soient inférieurs ou égaux à 1.

## 2.3 Compléments

On trouvera dans certains chapitres une section constituée par des aspects de METAPOST en relation plus ou moins directe avec l'objet du chapitre; on y trouvera parfois quelques précisions concernant la syntaxe.

### 2.3.1 Résolution d'équations linéaires

Puisque que ce chapitre est consacré aux lignes brisées constituées par des segments de droite, il est naturel de présenter la résolution des systèmes d'équations linéaires par METAPOST, résolution qui permet en particulier de déterminer, dans la sous-section suivante, le point d'intersection de deux droites (quand il existe, bien entendu!).

On a vu que l'on représente un point du plan par une variable du type paire; si l'on a une paire  $z1$ , alors on a les relations [40] :

$x1=xpart\ z1$ ; et  $y1=ypart\ z1$ ;

qui donnent  $x1$  et  $y1$  ( $z1=(x1,y1)$ ). Cela est particulièrement utile lorsque  $z1$  est le résultat d'un calcul, ces deux opérateurs  $xpart$  et  $ypart$  permettent alors de déterminer l'un ou l'autre des éléments de la paire  $z1$ . Voici un exemple où l'on veut utiliser uniquement l'abscisse  $x1$  de  $z1$ .

```
beginfig(8);
axespapiermilli(-1.5,-1.5,36.5,16.5)
z1+z2=(5u,2u);z1-z2=(2u,5u);a=xpart z1;
draw (35u,0)--(a,15u);drawpt(z1,3pt);
label.urt("(a,15u)",(a,15u));
endfig;
```



### 2.3.2 Intersection de droites

Une droite est définie par deux points,  $z1$  et  $z2$  par exemple. Alors tout point de cette droite peut s'écrire sous la forme  $z5 = z1 + a \cdot (z2 - z1)$ , ce que l'on note [30], [M 9] :

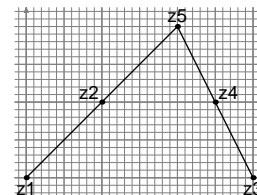
$z5=a[z1,z2]$ ;

( $a = 0.5$  correspond au milieu du segment d'origine  $z1$  et d'extrémité  $z2$ ). Si l'on a une autre droite définie par  $z3$  et  $z4$ , tout point de cette droite est encore de la forme :

$z5=b[z3,z4]$ ;

Ces deux equations sont en fait quatre équations à quatre inconnues  $x5$ ,  $y5$ ,  $a$  et  $b$  que METAPOST résoud; en particulier  $z5$  est disponible pour un tracé, voir l'exemple suivant.

```
beginfig(9);
axespapiermilli(-1.5,-1.5,31.5,21.5);
z1=(0,0);z2=(10u,10u);z3=(30u,0);z4=(25u,10u);
numeric a;b;z5=a[z1,z2]=b[z3,z4];
draw z1--z5;draw z3--z5;drawpt(z1,2pt);
drawpt(z2,2pt);drawpt(z3,2pt);drawpt(z4,2pt);
drawpt(z5,3pt);label.bot("z1",z1);
label.ulft("z2",z2);label.bot("z3",z3);
label.urt("z4",z4);label.top("z5",z5);
endfig;
```



Ici on est dans un cas où il faut déclarer les variables du type nombre (elles ne sont pas définies en les égalant à un nombre : ce sont des inconnues!). Pour ces variables annexes dont la valeur est sans intérêt, on dispose d'un unique mot `whatever` qui, de façon transparente, définit une nouvelle variable du type nombre chaque fois qu'il est utilisé : les équations ci-dessus s'écrivent ainsi :  
`z5=whatever[z1,z2]=whatever[z3,z4];`

### 2.3.3 Précisions de certains points de syntaxe

On a utilisé cinq types de variables qui doivent être (presque toujours) déclarées :

`a,b,x[],y[]` dans `b=1;b=3;x1=5;y2=7;` etc., type *nombre* sans déclaration;  
`numeric c,s[]` dans `z3=c[z2,z1];s1=9;s2=8;` etc., type *nombre* ;  
`z[]` dans : `z1=(7,9);` etc., type *paire* sans déclaration ;  
`pair qa,q[]` ; dans `qa=(7,8);q1=(3,4);q2(5,6);` etc., type *paire* ;  
`color yellow;` dans `yellow=(1,1,0);` type *couleur* ;  
`path pa,p[]` ; dans `pa=z1--z4;p1=z1--z2;p2=z3--z4;` etc., type *chemin* ;  
`picture dessin;` dans `dessin=currentpicture;` type *dessin*.

Toutes les variables tableaux de tous les types doivent être déclarées sauf : `x[], y[]` et `z[]` qui sont prédéclarées (nombres et paire); en outre `xi` et `yi` sont, par le biais d'une macro, respectivement les éléments de la paire `zi` (par exemple, `x1=1;` et `z1=(2,3);` sont incompatibles); les valeurs de ces variables (`x[], y[]` et `z[]`) sont effacées par les macros `beginfig`.

Quand une nouvelle variable est introduite, elle est considérée du type numérique; si on écrit `a=1;` sans déclaration, il n'y a pas de message d'erreur car 1 est bien un nombre. Par contre, si l'on écrit `q=(3,4);` sans déclarer `q` comme variable du type paire, alors on a un message du type :

« *inconsistent equation (numeric=pair)* »

pour rappeler qu'il manque la déclaration. En fait, il n'y a que les variables nombre qui ne nécessitent pas de déclaration si elles sont introduites par une équation non ambiguë du point de vue syntaxique (`numeric=numeric`), ce qui n'est pas le cas de l'équation `z3=c[z2,z1]` qui exige une déclaration préalable de `c` (voir la deuxième ligne de la liste ci-dessus). On a rencontré aussi les types de variables *transformation* (exemple : `shifted`) et *plume* (exemple : `pencircle`); on y reviendra plus loin car il en manque encore deux et il y a aussi les *variables internes* (exemple : `ahlength`).

On a aussi défini des macros; la syntaxe des ces définitions est [76] :

```
def nom-macro = texte de remplacement enddef;
pour une macro sans paramètre :
def nom-macro(expr s,t,etc.) =
    texte de remplacement incluant les paramètres s,t,etc. enddef;
pour une macro avec paramètres; expr signifie que les paramètres qui suivent
sont des variables des neuf types qui peuvent être définis par METAPOST (dont
trois n'ont pas encore été rencontrés). Une définition plus générale est [83] :
def nom-macro(expr s,t,etc.)(text tt) =
    texte de rempl. incl. les param. s,t,etc. et tt enddf;
```

où `text` signifie que le paramètre qui suit est du texte, par exemple une suite d'instructions (cas de la macro `axespapiermilli` dans `macutil.mp`); l'utilité de ce type de paramètre apparaîtra dans des cas bien différents de cette macro citée (macro `arcd` définie à la section 3.5.1). Il faut noter qu'il faut introduire autant de groupes (`text ...`) qu'il y a de paramètres du type `text` car la virgule ne joue plus le rôle de séparateur (elle peut faire partie des paramètres).

### 2.3.4 Préambule du fichier des figures

On écrit les affectations et les définitions utilisées par les figures dans les préambules des fichiers des figures.

Si une affectation est répétée; elle ne donne pas lieu à une erreur mais simplement à un avertissement de redondance. Le lecteur pourra vérifier en enlevant les caractères `%<` devant la ligne `u=1mm`; de la figure 1 (chap. 1) et devant la ligne `p=z1--z2--z3--z4--cycle`; de la figure 2 (chap 2).

Les déclarations de variables et les définitions de macros peuvent ne pas être répétées, elles restent valables pour les figures suivantes. Si elles sont répétées, elles ne donnent pas lieu à message. Il vaut mieux répéter systématiquement les déclarations pour éviter des erreurs : si une figure contient `a=1`; et la suivante `a=2`; alors un message signale l'incompatibilité : il faut rajouter `numeric a`; dans la deuxième figure, la déclaration effaçant la précédente égalité.

Voici le préambule des fichiers `figa.mp`, ..., `figd.mp` des figures des chapitres 1 à 4. Comme signalé à la section 2.1, pour plus de commodité, les définitions sont rassemblées dans le fichier `macutil.mp` donné en suivant.

```

%% figb.mp -- Preambule
u=1mm;ahlength:=2pt;
prologues:=2;labeloffset:=1.7pt;
defaultfont:="phvr8r";defaultscale:=0.7;
input macutil.mp;

%% macutil.mp -- Macros utiles
def lw(expr s)=withpen pencircle scaled s enddef;
def cl(expr s)=withcolor (s,s,s) enddef;
def drawpt(expr z,s)=draw z withpen pencircle scaled s enddef;
def polar(expr r,w)=(r*cosd w,r*sind w)enddef;
def axespapiermillii(expr r,s,v,w,k)(text t)= % grille millimetre
  for i=0 step k until w : draw (r*u,i*u)--(v*u,i*u) t; endfor
  for i=0 step k until v : draw (i*u,s*u)--(i*u,w*u) t; endfor
  for i=-k step -k until s : draw (r*u,i*u)--(v*u,i*u) t; endfor
  for i=-k step -k until r : draw (i*u,s*u)--(i*u,w*u) t; endfor
enddef;
def axespapiermilli(expr r,s,v,w)=
  axespapiermillii(r,s,v,w,1)(lw(0.1pt) cl(0.5));
  axespapiermillii(r,s,v,w,5)(lw(0.2pt) cl(0.5));
  axespapiermillii(r,s,v,w,10)(lw(0.3pt) cl(0.5));
  drawarrow (r*u,0)--(v*u,0) lw(0.4pt) lw(0.4pt) cl(0.5);
  drawarrow (0,s*u)--(0,w*u) lw(0.4pt) lw(0.4pt) cl(0.5);
enddef;
def arcd(expr z,r,a,b)(text parg)=begingroup % arcs de cercle
  save c,p,pc; path p,pc; c=b-a; p=(0,0)--(1.2,0)rotated c;
  pc=fullcircle cutafter p; parg=pc scaled r rotated a shifted z;
endgroup enddef;

```



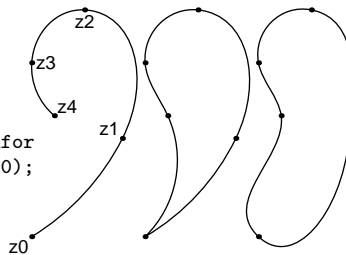
## Tracé de lignes courbes

Dans ce chapitre, on va découvrir la puissance de METAPOST (héritée bien entendu de la puissance de METAFONT comme cela est expliqué dans l'introduction). On va pouvoir tracer des courbes passant par des points donnés, appelés points guides, avec continuité de la tangente en ces points. Ensuite, on va pouvoir intervenir sur la forme de ces courbes par le biais d'options faciles à utiliser (par exemple, on va pouvoir imposer la direction de la tangente en chaque point guide, modifier la « rondeur » des éléments des courbes situés entre deux points guides, etc.). On donne d'abord un aperçu très partiel de ce que l'on peut faire en traçant des variantes d'une courbe passant par cinq points guides. Ensuite, on expose la méthode de tracé utilisée par METAPOST (courbes de Bézier cubiques). Enfin, on fait le tour des possibilités offertes pour modifier la forme de ces courbes afin de leur donner les propriétés voulues. On termine le chapitre en exposant les puissantes commandes disponibles (par exemple, pour découper des morceaux de courbe, pour déterminer les points d'intersection des courbes, pour tracer des tangentes aux courbes, etc.). Dans ce chapitre, le code des lettrages n'est pas reporté dans le listing des exercices.

### 3.1 Un tracé élémentaire pour débiter

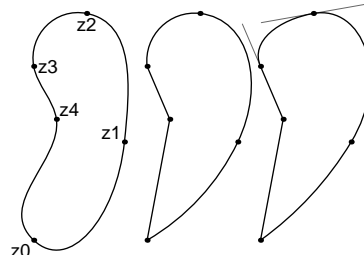
On considère cinq points représentés par  $z_0$ ,  $z_1$ ,  $z_2$ ,  $z_3$  et  $z_4$ . On trace plusieurs courbes passant par ces cinq points appelés *points guides* (fichier de figures `figc.mp`). La syntaxe ne diffère de la syntaxe de tracé des lignes brisées que par le remplacement de `--` par `..`; sur l'exemple, il y a d'abord la courbe ouverte allant de  $z_0$  à  $z_4$ , ensuite on ferme cette courbe en revenant au point  $z_0$  et enfin on ferme la courbe avec `..cycle` au lieu de revenir en  $z_0$  : cette option ferme la courbe et assure la continuité de la tangente [22].

```
beginfig(1);
path p[];z0=(2u,0);z1=(14u,13u);z2=(9u,30u);
z3=(2u,23u);z4=(5u,16u);draw z0..z1..z2..z3..z4;
for i=0 upto 4:drawpt(z[i],2pt);endfor
p1=z0..z1..z2..z3..z4..z0;draw p1 shifted(15u,0);
for i=0 upto 4:drawpt(z[i],2pt)shifted(15u,0);endfor
p2=z0..z1..z2..z3..z4..cycle;draw p2 shifted(30u,0);
for i=0 upto 4:
  drawpt(z[i],2pt) shifted(30u,0);endfor
endfig;
```



L'exemple ci-dessus montre bien l'importance de l'utilisation de `cycle` pour fermer les lignes courbes (on reviendra sur ce point par la suite). Pour l'exemple suivant, on reprend la courbe fermée correctement puis on remplace les deux derniers éléments par deux segments (`..` remplacé deux fois par `--`) : on constate qu'il n'y a pas continuité de la tangente en `z0` et `z3`, ainsi qu'en `z4` où se joignent les deux segments ; on montre ensuite que l'on peut imposer la continuité de la tangente au point `z3` (tangente dans le prolongement du premier segment) et que l'on peut imposer une direction à la tangente au point `z2` (tangente orientée faisant un angle de  $90^\circ$  avec l'axe des  $x$ ) [24], [M 16].

```
beginfig(2);
path p[];
z0=(2u,0);z1=(14u,13u);z2=(9u,30u); z3=(2u,23u);z4=(5u,16u);
%<def lwcl=lw(0.4pt) cl(0.4) enddef;
p1=z0..z1..z2..z3..z4..cycle; draw p1;
for i=0 upto 4 : drawpt(z[i],2pt); endfor
p2=z0..z1..z2..z3--z4--cycle;
draw p2 shifted(15u,0);
for i=0 upto 4 :
  drawpt(z[i],2pt) shifted(15u,0); endfor
p[4]=z3--1.8[z4,z3];draw p4 shifted(30u,0)lwcl;
p5=z2+polar(7u,190)--z2-polar(7u,190);
draw p5 shifted(30u,0)lwcl;
p3=z0..z1..{dir190}z2..{z4-z3}z3--z4--cycle;
draw p3 shifted(30u,0);
for i=0 upto 4 : drawpt(z[i],2pt) shifted(30u,0); endfor
endfig;
```



Tout cela paraît un peu mystérieux ; pour comprendre, il faut examiner la méthode de tracé : il faut se familiariser avec les courbes de Bézier cubiques.

### 3.2 Courbes de Bézier cubiques

Considérons quatre points `z1`, `zc1`, `zc2` et `z2` ; ces quatre points définissent de façon unique une courbe dite de Bézier dont l'origine est `z1`, l'extrémité est `z2` (elle est donc orientée de `z1` vers `z2`, cela est fondamental pour la suite) et les *points de contrôle* sont `zc1` et `zc2` : ce sont seulement ces deux points qui caractérisent la forme de la courbe entre son origine et son extrémité. Du point de vue mathématique, cette courbe est définie paramétriquement par deux polynômes du troisième degré :  $X(t)$  et  $Y(t)$  pour  $t \in [0, 1]$  [23]. La syntaxe pour tracer la courbe est [24] :

```
draw z1..controls zc1 and zc2..z2;
```

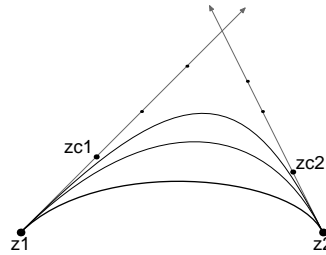
Ce tracé fait l'objet de l'exemple suivant. La caractéristique première de cette courbe est que la tangente orientée au point `z1` est dans la direction de `zc1` et que la tangente orientée en `z2` est dans la direction opposée à la direction de `zc2`. La deuxième caractéristique est que les points de contrôle donnent l'impression de « tirer » la courbe vers eux comme le montre l'exemple où les point de contrôle sont éloignés (suivant les flèches) de manière à conserver les directions des tangentes aux deux extrémités. Ces courbes de Bézier peuvent être construites point par point par un méthode géométrique [M 13].



```

beginfig(3);
z1=(0,0);z2=(40u,0);drawpt(z1,3pt);drawpt(z2,3pt);
drawarrow z1--polar(42u,45)lwcl;
drawarrow z2--(25u,30u)lwcl;
pair zc[];zc1=(10u,10u);zc2=(36u,8u);
drawpt(zc1,2pt);drawpt(zc2,2pt);
draw z1..controls zc1 and zc2..z2;
zc3=(16u,16u);zc4=(32u,16u);
drawpt(zc3,1.2pt);drawpt(zc4,1.2pt);
draw z1..controls zc3 and zc4..z2lw(0.35pt);
zc5=(22u,22u);zc6=(30u,20u);
drawpt(zc5,1.2pt);drawpt(zc6,1.2pt);
draw z1..controls zc5 and zc6..z2 lw(0.35pt);
endfig;

```

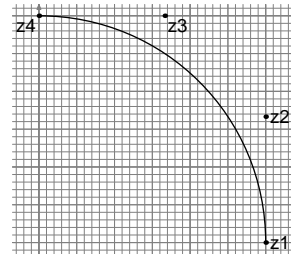


On peut aussi obtenir un quart de cercle avec une très grande précision par la courbe de Bézier de l'exemple suivant :

```

beginfig(4);
axespapiermilli(-1.5,-1.5,31.5,31.5);
z1=(30u,0);z2=(30u,16.65u);
z3=(16.65u,30u);z4=(0,30u);
draw z1..controls z2 and z3..(0,30u);
drawpt(z1,1.2pt);drawpt(z2,1.2pt);
drawpt(z3,1.2pt);drawpt(z4,1.2pt);
endfig;

```



Pour les courbes de la section précédente, on n'a pas disposé de points de contrôle pour les différents éléments constitutifs et pourtant, ces courbes sont bien « sympathiques ». On va maintenant voir les différentes possibilités de METAPOST pour construire des courbes, possibilités basées sur les courbes de Bézier cubiques.

### 3.3 Différentes possibilités de tracés de courbes

#### 3.3.1 Tracé avec les seuls points guides

A la section 2.1, on a construit des courbes en ne donnant que les points par lesquels elles doivent passer. METAPOST trace ces courbes en construisant  $n-1$  (resp.  $n$ ) courbes de Bézier pour les courbes ouvertes (resp. fermées) définies par  $n$  points guides sans la donnée de points de contrôle. Le programme détermine, par un algorithme complexe qui sort du cadre de ce manuel et des références citées, les points de contrôle nécessaires pour obtenir la courbe la « meilleure » possible. Evidemment ce qualificatif est subjectif!

#### 3.3.2 Tracé en fixant les tangentes aux points guides

On peut tout de même donner un certain sens à ce qualificatif : on peut imaginer que l'application pour laquelle on dessine la courbe exige que les tangentes orientées à la courbe aux points guides aient des directions données. On

a déjà vu que cela est possible à la fin de la section 2.1. On précise maintenant les deux formes de syntaxe de la commande qui permet de satisfaire à cette exigence [24], [M 16] :

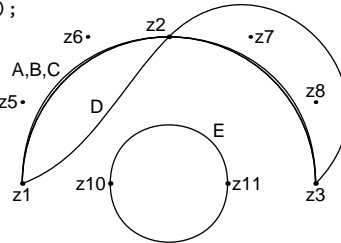
```
draw z1{dir nombre}..z2{zi-zj}..z3{...}.. ..{...}z4;
```

où `nombre` est la valeur souhaitée de l'angle en degrés que la tangente orientée doit faire avec l'axe des  $x$  croissants,

et où `zi-zj` désigne un vecteur d'origine  $zj$  et d'extrémité  $zi$  dont on veut que la tangente orientée ait la direction.

On montre sur l'exemple suivant des tracés qui appellent quelques commentaires. On donne les trois points  $z1=(0,0)$ ,  $z2=(20u,20u)$  et  $z3=(40u,0)$  ; en donnant les bonnes directions aux tangentes en ces points, on obtient, ligne A, presque le demi cercle obtenu avec la courbe de Bézier et les points de contrôle particuliers déjà utilisé à la section précédente, ligne B, pour tracer le quart de cercle ; plus suprenant, la considération des points extrêmes seulement et des directions des tangentes en ces points donne encore, ligne C, presque le demi cercle ; plus surprenant encore, la seule donnée de deux points points et de la commande de fermeture du chemin (`..cycle`) produit pratiquement un cercle, ligne E. On peut dire que l'algorithme de détermination des points de contrôle tend à faire un chemin le plus « arrondi » possible.

```
beginfig(5);u:=0.97mm;
z1=(0,0);z2=(20u,20u);z3=(40u,0);
drawpt(z1,1.8pt);drawpt(z2,1.8pt);drawpt(z3,1.8pt);
z5=(0,11.1u);z6=(8.9u,20u);z7=(31.1u,20u);
z8=(40u,11.1u);drawpt(z5,1.5pt);drawpt(z6,1.5pt);
drawpt(z7,1.5pt);drawpt(z8,1.5pt);
draw z1{dir90}..z2{dir0}..z3{dir-90};% ligne A
draw z1..controls z5 and z6..z2..
      controls z7 and z8..z3; % ligne B
draw z1{dir90}..{dir-90}z3; % ligne C
draw z1{dir20}..z2{dir45}..{dir-135}z3;% ligne D
z10=(12u,0);z11=(28u,0);drawpt(z10,1.8pt);
drawpt(z11,1.8pt);draw z10..z11..cycle;% ligne E
endfig;
```



### 3.3.3 Modification du tracé : tension et courbure

On expose maintenant les options disponibles pour imposer des contraintes aux courbes telles qu'elles sont tracées ci-dessus, soit à partir des points guides seulement, soit à partir des points guides et des directions des tangentes en ces points. Ces possibilités sont au nombre de trois. Sur l'exemple suivant où l'on trouvera le tracé normal et les tracés modifiés en trait plus fin (pour les deux premières possibilités : un tracé pour une valeur du paramètre plus faible que la valeur par défaut et un tracé pour la situation contraire ; pour la troisième possibilité : un seul tracé avec modification).

- Dans le cas où on impose les directions des tangentes aux extrémités d'un segment, on peut, en modifiant le paramètre `tension` (défaut : 1, valeur dans  $[0.75, \infty]$ ), modifier le « gonflement » de la courbe, partie (A). La syntaxe à utiliser [26], [M 15] :  

```
draw z0{dir35}.. tension 1.5 ..z1{dir-45};
```

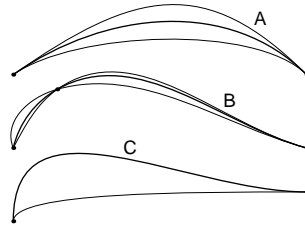
- Dans la cas d'une extrémité d'une courbe ouverte où la direction de la tangente orientée n'est pas imposée, on peut modifier le paramètre `curl` (défaut : 1, valeur dans  $[0, \infty]$ ) pour modifier la courbe, partie (B). Dans ce cas-ci, la dénomination du paramètre n'est pas très explicite ; il caractérise la courbure au voisinage de l'extrémité concernée. Pour la valeur 1, le tronçon de courbe au voisinage de `z1` est presque un arc de cercle d'un certain rayon, pour les valeurs plus grandes que 1, ce tronçon est voisin d'un arc de cercle d'un rayon plus petit et c'est le contraire pour les valeurs plus petites que 1. La syntaxe correspondante est alors [27], [M 17] :

```
draw z5{curl15}..z6..z7{dir-15};
```

- De nouveau pour un tronçon de courbe avec les directions des tangentes aux extrémités imposées, on peut vouloir éviter un éventuel point d'inflexion ; cela se fait en remplaçant les `..` par `...` dans la commande de tracé, partie (C) ; la syntaxe à utiliser est [26] :

```
draw z11{dir90}...z12{dir0};
```

```
beginfig(6);
path p[];z0=(0,20u);z1=(40u,20u);drawpt(z0,1.6pt);drawpt(z1,1.6pt);
% Partie A
draw z0{dir35}.. tension 0.76 ..z1{dir-45}lw(0.35pt);
draw z0{dir35}..z1{dir-45};
draw z0{dir35}.. tension 1.5 ..z1{dir-45}lw(0.35pt);
% Partie B
z5=(0,10u);z6=(6u,18u);z7=(40u,10u);
drawpt(z5,1.6pt);drawpt(z6,1.6pt);
drawpt(z7,1.6pt);
draw z5{curl0.01}..z6..z7{dir-15}lw(0.35pt);
%% Partie C
draw z5..z6..z7{dir-15};
draw z5{curl15}..z6..z7{dir-15}lw(0.35pt);
z11=(0,0);z12=(40u,4u);
drawpt(z11,1.6pt);drawpt(z12,1.6pt);
draw z11{dir90}..z12{dir0};draw z11{dir90}...z12{dir0}lw(0.35pt);
endfig;
```



### 3.4 Gestion des courbes

Cette section traite du découpage des courbes en éléments, des intersections de courbes, de la construction de chemins fermés en vue d'un remplissage et des tangentes et des normales aux courbes. Avant d'entrer dans le vif du sujet il faut préciser la représentation paramétrique des courbes telle que l'utilise METAPOST. Une courbe de Bézier est représentée par les deux pôlynomes  $X(t)$  et  $Y(t)$  où  $t \in [0, 1]$ . Une courbe ouverte ou fermée formée de  $n$  éléments a encore une représentation paramétrique semblable avec  $t \in [0, n + 1]$ . Le paramètre  $t$  est parfois appelé temps car il est identifié au temps auquel un voyageur imaginaire passerait au point défini par  $X(t)$  et  $Y(t)$ ; cette identification explique les dénominations utilisées pour les commandes relatives à la gestion des courbes.

### 3.4.1 Découpage des courbes

A partir d'une valeur de  $t$ , on peut obtenir les coordonnées  $X(t)$  et  $Y(t)$  du point correspondant de la courbe par la commande [57] ;

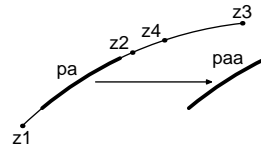
```
z=point t of p;
```

où  $p$  est le nom de la courbe. On utilise aussi la forme :  $(a,b)=\text{point } t \text{ of } p$ ; qui donne directement les éléments de la paire représentant ce point. La partie de la courbe nommée  $pa$  et décrite par les valeurs de  $t \in [t_1, t_2]$  est donnée par la commande [57] :

```
pa=subpath (t1,t2) of p;
```

Ces propriétés sont montrées sur l'exemple suivant.

```
beginfig(7);
numeric t[];path p,pa,paa;
z1=(0,0);z2=(15u,10u);z3=(30u,14u);
drawpt(z1,1.8pt);drawpt(z2,1.8pt);drawpt(z3,1.8pt);
p=z1..z2..z3;draw p;
z4=point 1.3 of p;drawpt(z4,1.3pt);
pa=subpath(0.2,0.9) of p;
paa=pa shifted(20u,0);draw paa lw(1.5pt);
drawarrow(10u,6u)--(25.8u,6u);
endfig;
```



### 3.4.2 Intersections de courbes

Le point d'intersection  $z$  des courbes  $pa$  et  $pb$  est donné par [55], [M 137] :

```
z=pa intersectionpoint pb;
```

s'il n'y a qu'un seul point d'intersection. On peut aussi utiliser [56] :

```
(ta,tb)=pa intersectiontimes pb;
```

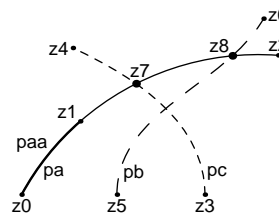
qui donne les temps  $ta$  et  $tb$  correspondants au point d'intersection sur chacun des deux chemins. Ensuite, le point d'intersection  $z$  est obtenu avec :

```
z=point ta of pa; ou tb of pb;
```

C'est cette deuxième méthode va être utilisée lorsqu'il y a plusieurs points d'intersection ; l'exemple suivant montre les deux possibilités.

S'il n'y a aucun point d'intersection, la paire  $(ta,tb)$  vaut  $(-1,-1)$  ; on peut voir que c'est le cas des chemins  $paa=\text{subpath}(0,1)$  of  $pa$  et  $pb$  en affichant à la console ces valeurs avec la commande : `show ta, tb;`

```
beginfig(8);
z0=(0,0);z1=(8u,10u);z2=(35u,19u);z3=(25u,0);
z4=(7u,20u);z5=(13u,0);z6=(33u,24u);
drawpt(z0,1.8pt);drawpt(z1,1.8pt);drawpt(z2,1.8pt);
drawpt(z3,1.8pt);drawpt(z4,1.8pt);
drawpt(z5,1.8pt);drawpt(z6,1.8pt);
path pa,paa,pb,pc;pa=z0..z1..z2;draw pa;
pb=z3{dir90}..z4{dir155};draw pb dashed evenly;
z7=pa intersectionpoint pb;drawpt(z7,3pt);
pc=z5{dir90}..z6{dir60};draw pc
dashed evenly scaled 2;
(ta,tb)= pa intersectiontimes pc;
z8=point ta of pa;drawpt(z8,3pt);
paa=subpath(0,1) of pa;
(tc,td)=paa intersectiontimes pb;show tc,td;% test de non intersection
endfig;
```



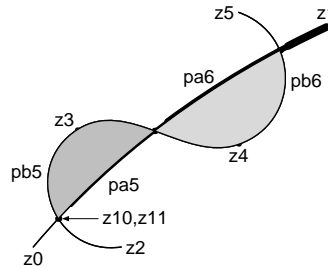
Il y a enfin le cas où l'on ne sait rien sur les positions des points d'intersection. Le programme METAPOST est conçu pour donner, de préférence, la paire  $(ta, tb)$  qui correspond au point d'intersection de  $pa$  et  $pb$  ayant la plus petite valeur de  $ta$  [56]. Cela permet de procéder comme suit :

- on commence par écrire l'équation :  
 $(ta1, tb1) = pa \text{ intersectiontimes } pb;$   
ce qui détermine le premier point d'intersection avec :  
 $zint1 = \text{point } ta1 \text{ of } pa;$  ou  $zint1 = \text{point } tb1 \text{ of } pb;$
- on recommence l'opération avec :  
 $pa1 = \text{subpath}(ta1 + 0.01, \text{length of } pa);$   
 $(ua2, tb2) = pa1 \text{ intersectiontimes } pb;$   
(sur la figure, on a pris 0.05 au lieu de 0.01 pour que l'effet soit visible :  $pa1$  est plus épais que  $pa$ ). Mais cette fois-ci,  $ua2$  est une valeur du paramètre du sous-chemin  $pa1$  et non une valeur du paramètre du chemin  $pa$ , donc le deuxième point d'intersection sera donné par :  
 $zint2 = \text{point } ua2 \text{ of } pa1;$  (et non  $zint2 = \text{point } ta2 \text{ of } pa;$  comme on le souhaiterait) ou  $zint2 = \text{point } tb2 \text{ of } pb;$

```

beginfig(9);
z0=(0,0);z1=(40u,30u);z2=(12u,0);z3=(6u,16u);z4=(28u,14u);z5=(28u,32u);
path pa,pa[],pb,pb[];pa=z0{dir50}..z1{dir25};pb=z2..z3..z4..z5;draw pa;draw pb;
numeric ta[],ua[],tb[],ub[];
%% Partie 1 : calcul de ta1, tb1, tb2 et tb3
(ta1,tb1)=pa intersectiontimes pb;show ta1,tb1;pa1=subpath(ta1+0.01,1)of pa;
(ua2,tb2)=pa1 intersectiontimes pb;show ua2,tb2;pa2=subpath(ua2+0.01,1)of pa1;
(ua3,tb3)=pa2 intersectiontimes pb;show ua3,tb3;pa3=subpath(ua3+0.01,1)of pa2;
(ua4,tb4)=pa3 intersectiontimes pb;show ua4,tb4;
drawpt(point tb1 of pb,2.5pt);
drawpt(point tb2 of pb,2.5pt);
drawpt(point tb3 of pb,2.5pt);
%% Partie 2 : calcul de ta2 et ta3
pb1=subpath(tb1+0.01,3)of pb;draw pb1 lwcl;
(ub2,ta2)=pb1 intersectiontimes pa;show ta2,ub2;
pb2=subpath(ub2+0.01,3)of pb1;
(ub3,ta3)=pb2 intersectiontimes pa;show ta3,ub3;
%% Partie 3 : creation des zones grisees
pa5=subpath(ta1,ta2)of pa;
pb5=subpath(tb1,tb2)of pb;
pa6=subpath(ta2,ta3)of pa;
pb6=subpath(tb2,tb3)of pb;
%<def c1(expr s)=withcolor (s,s,s) enddef;
pb7=pa5..reverse pb5..cycle;fill pb7 c1(0.75);pb8=pa6..reverse pb6..cycle;
fill pb8 c1(0.85);draw pa; draw pb;
%% Partie 4 : verification de z10 different de z11
z10=point ta1 of pa; show z10;z11=point tb1 of pb; show z11;
endfig;

```



- on arrête lorsque  $(uaj, tbj) = (-1, -1)$  (d'où la nécessité des commandes  $\text{show}(ta, tb)$  à chaque pas); on a alors déterminé les  $j - 1$  points d'intersection des chemins  $pa$  et  $pb$  (partie 1); on peut suivre sur la figure car les sous-chemins  $pa1$ ,  $pa2$  et  $pa3$  sont de plus en plus épais.
- ainsi, on a bien les  $j - 1$  valeurs de  $tb$  avec lesquelles on peut découper des morceaux du chemin  $pb$  limités par des points d'intersection avec le chemin  $pa$ , mais on ne peut pas découper le chemin  $pa$  car on n'a que la première

valeur `ta1` de `ta`; les autres valeurs `ta2` et `ta3` sont nécessaires si l'on veut découper des morceaux de `pa` limités par des points d'intersection avec `pb`; il suffit pour cela de recommencer le processus (partie 2) en interchangeant le rôle des chemins `pa` et `pb`.

On peut enfin nommer les morceaux des chemins `pa` et `pb` (`pa5`, `pb5`, etc.), puis former avec ces morceaux des chemins fermés (partie 3) avec, par exemple : `pb7=pa5.reverse pb5.cycle; fill pb7;`  
La nécessité de `.cycle` (au lieu de l'opérateur de concaténation `&`, voir sect. 4.2.6 et 3.5.3) est le résultat des calculs approchés : le début `z10` du chemin `pa5` et la fin `z11` du chemin `reverse pb5` ne coïncident pas exactement.

On vérifie cette non coïncidence (partie 4) en calculant le même point d'intersection par `z10 point ta1 of pa;` et `z11 point tb1 of pb;` qui donneraient le même résultat si les calculs étaient infiniment précis (la différence est de l'ordre de un millième de pt).

Pour s'éloigner au delà du point d'intersection, on a arbitrairement choisi le temps `ta1+0.01`; METAPOST dispose pour cela d'un nombre très petit nommé `epsilon` [110], [M 62] valant  $1/65536$ .

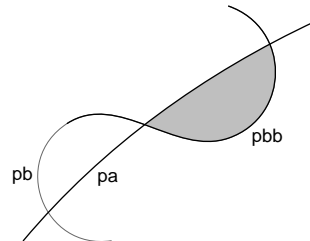
### 3.4.3 Construction directe des zones d'intersection

Dans la sous-section précédente, on a construit puis coloré des zones que l'on peut appeler zones d'intersection; on a calculé les points d'intersection, puis formé des chemins partiels avec ces points et enfin construit les chemins fermés limitant ces zones. METAPOST dispose d'une puissante macro (qui n'existe pas dans METAFONT) permettant de définir ces zones d'intersection directement dans des cas particuliers auxquels on peut très souvent se ramener.

Le premier cas correspond à la situation où deux chemins se coupent en deux points seulement. On va reprendre l'exemple précédent où il y a trois points d'intersection mais où on peut limiter le chemin `pb` pour retomber dans la situation de validité de la macro; on écrit simplement [54] :

```
zoneint=buildcycle(chemin1,chemin2);
Il faut s'assurer de l'orientation compatible des chemins concernés. On rappelle qu'il faut remplir puis tracer les zones concernées (cf. section 1.2); si l'on est amené, lors de la construction d'une figure, à tracer des chemins pour suivre les étapes de la construction, il faudra peut-être en effacer une partie à la fin.

beginfig(10);
z0=(0,0);z1=(40u,30u);z2=(12u,0);z3=(6u,16u);
z4=(28u,14u);z5=(28u,32u);
path pa,paa,pb,pbb,zoneint;pa=z0{dir50}..z1{dir25};
pb=z2..z3..z4..z5;draw pb lwcl;
pbb=subpath(1,3) of pb;paa=reverse pa;
zoneint=buildcycle(paa,pbb);
fill zoneint c1(0.75);draw pa;draw pbb;
endfig;
```

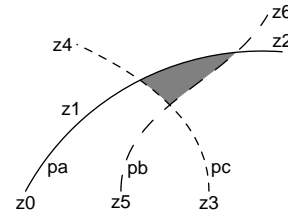


Le deuxième cas correspond à plus de deux chemins. Si l'on écrit par exemple la commande [54] :

```
zoneint=buildcycle(p1,p2,p3);
```

METAPOST choisit le point d'intersection du chemin  $p_i$  avec le chemin  $p_{i+1}$  qui a la plus petite valeur du paramètre  $t$  sur le chemin  $p_i$  et la plus grande valeur sur le chemin  $p_{i+1}$ . On peut bien entendu obtenir des résultats surprenants sauf si les chemins  $p_i$  et  $p_{i+1}$  ( $\forall i, i = 1, \dots, n$ , avec  $p_{n+1} = p_1$ ) n'ont qu'un seul point d'intersection ou si l'on peut définir des sous-chemins ayant cette propriété. On reprend l'exemple de la figure 8 qui satisfait à cette condition.

```
beginfig(11);
z0=(0,0);z1=(8u,10u);z2=(35u,19u);
z3=(25u,0);z4=(7u,20u);z5=(13u,0);z6=(33u,24u);
path pa,paa,pb,pbb,pc,zoneint;pa=z0..z1..z2;
pb=z3{dir90}..z4{dir155};pc=z5{dir90}..z6{dir60};
paa=reverse pa;pbb=reverse pb;
zoneint=buildcycle(paa,pbb,pc);
fill zoneint c1(0.5);
draw pa;draw pb dashed evenly;
draw pc dashed evenly scaled 2;
endfig;
```



### 3.4.4 Tangentes et normales aux courbes

On peut tracer la tangente en un point de paramètre  $t$  d'une courbe  $p$  de la manière suivante, partie A de l'exemple [58] :

```
(a,b)=direction t of p;
```

donne les composantes  $a$  et  $b$  du vecteur direction de la tangente orientée à la courbe en ce point. Le point de contact est alors donné par :

```
zc=point t of p;
```

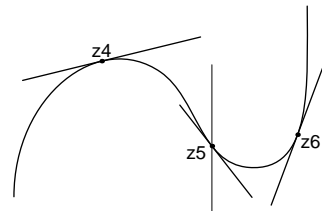
la tangente est alors définie avec  $zc$  et le point  $zct=zc+(a,b)$  et la normale avec les points  $zc$  et  $zcn=zc+(-b,a)$ .

On peut aussi tracer la tangente à la courbe  $p$  au point d'abscisse  $xc$ , partie B ; on définit alors le chemin rectiligne  $pc=(xc,0)--(xc,30u)$  et alors la valeur du paramètre  $tc$  correspondant au point de contact  $zc$  est donnée par [56] :

```
(tc,whatever)=p intersectiontimes qc;
```

ce qui permet de tracer la tangente comme on l'a fait plus haut.

```
beginfig(12);
z0=(0,2u);z1=(20u,19u);z2=(32u,6u);z3=(40u,28u);numeric tc;
path p,pp,droite,tanga,tangb,tangc;p=z0{dir90}..z1..z2..z3{dir90};draw p;
% Donnée t=0.8 ; partie A à gauche
(a,b)=direction 0.7 of p;z4=point 0.7 of p;
tanga=1.2[z4,z4-(a,b)]--1.5[z4,z4+(a,b)];
drawpt(z4,1.8pt);draw tanga;
% Donnée xc=27u ; partie B au milieu
droite=(27u,0)--(27u,30u);draw droite lw(0.3pt);
(tc,whatever)=p intersectiontimes droite;
(c,d)=direction tc of p;z5=point tc of p;
tangb=1.2[z5,z5-(c,d)]--1.5[z5,z5+(c,d)];
drawpt(z5,1.8pt);draw tangb;
% Donnée (3u,8u) ; partie C à droite
pp=subpath(1,4) of p;tcc=directiontime(3u,8u) of pp;z6=point tcc of pp;
tangc=1.2[z6,z6-(3u,8u)]--1.5[z6,z6+(3u,8u)];drawpt(z6,1.8pt);draw tangc;
endfig;
```



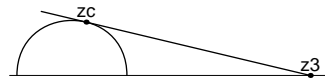
La dernière possibilité consiste à tracer la tangente de direction donnée par un vecteur  $(a,b)$  à un chemin  $p$ , partie C de l'exemple. La commande [58] :

`tc=directiontime(a,b) of p;`

donne la valeur `tc` (la plus petite valeur s'il y en a plusieurs possibles) du paramètre déterminant le point de tangence; le tracé de la tangente se fait alors comme ci-dessus. Il faut noter qu'il y a dans l'exemple ci-dessus deux possibilités pour tracer cette tangente de direction donnée : c'est pour cela que l'on a considéré le sous-chemin `pp` de `p`; ce type de méthode est conseillé car, très souvent, un simple examen visuel des chemins permet de choisir des sous chemins conduisant à une unique solution pour la recherche des points d'intersection ou, comme ici, pour un tracé de tangente.

Il reste un tracé non prévu par METAPOST : mener, à partir d'un point `z3`, une tangente à un chemin `p`. On propose dans l'exemple suivant une méthode par approximations successives qui nous donne l'occasion de faire une boucle avec sortie conditionnelle. Pour ce petit exercice, on choisit d'abord un intervalle de variation de la valeur du paramètre `w` correspondant au point de contact; avec la valeur de `w`, on construit la tangente dont on calcule le point d'intersection `zinter` avec l'axe des `x`. On balaie l'intervalle de variation dans le sens croissant : la valeur de `w` pour laquelle `xint-x3` change de signe donne le point de contact `zc` avec une précision que l'on peut apprécier en affichant, à chaque pas, ses coordonnées (de l'ordre de deux millièmes de pt).

```
beginfig(13);
z0=(0,0);z1=(15u,0);z3=(40u,0);path p;p=z0{dir90}..z1{dir-90};draw p;
draw(-1u,0)--(43u,0)lw(0.3pt);drawpt(z3,2pt);
for w=0.53 step 0.0001 until 0.59:
numeric a,b,xinter;pair zc, zcc, zinter;
(a,b)=direction w of p;
zc=point w of p;zcc=zc+(a,b);
%show zc %% Pour tester la precision
zinter=whatever[zcc,zc]=whatever[z0,z1];
xinter=xpart zinter;exitif xinter<x3;endfor
draw 1.3[z3,zc]--z3;drawpt(zc,2pt);
endfig;
```



Sur cet exemple, il faut remarquer la présence à l'intérieur de la boucle des déclarations des variables `a`, `b`, `xinter`, `zinter` etc.; cette présence est nécessaire pour annuler les affectations faites au pas précédent de la boucle, sans quoi on aurait des messages d'erreur du type « *inconsistent equations* ». Une autre solution serait de déclarer au début `a[]`, `b[]`, etc. et de travailler avec ces variables tableaux dans la boucle; mais toutes ces valeurs (des milliers si le pas est petit) sont inutiles, seule la dernière sert à construire la tangente. C'est une méthode à ne pas oublier.

## 3.5 Compléments

### 3.5.1 Cercles et arcs de cercles

On dispose des commandes [110–111] :

`fullcircle`, `halfcircle` et `quatercircle`

qui définissent respectivement (avec un rayon de 1 bp et l'origine en  $(0,0)$ ) un



cercle complet, un demi cercle et un quart de cercle. On écrit par exemple :

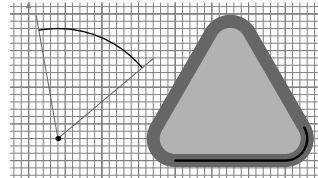
```
p=fullcircle scaled r shifted z;
```

pour avoir un cercle centré en  $z$  et de rayon  $r$ . On construit une macro :

```
arcd(z,r,a,b)(q)
```

qui définit (mais ne trace pas) un arc de cercle nommé  $q$  centré en  $z$ , de rayon  $r$ , d'angle origine  $a$  et d'angle extrémité  $b$ . L'exemple utilise cette macro pour tracer un arc de cercle centré en  $(4u, 3u)$ , de rayon  $6u$ , d'angle origine  $40$  degrés et d'angle extrémité  $100$  degrés. Ensuite on dessine le « tableau » d'un panneau de signalisation routière (l'élément de base nommé **canne** apparaît en trait noir en bas du panneau). Ce code sera expliqué et commenté la section 3.5.3.

```
beginfig(14);
axespapiermilli(-2.5,-2.5,40.5,21.5);
pair za,zb;path p,pcc,canne,tournant,triangle;
%<def arcd(expr z,r,a,b)(text parg)=begingroup
%<save c,p,pc;path p,pc;c=b-a;p=(0,0)--(1.2,0)rotated c;
%<pc=fullcircle cutafter p;
%<parg=pc scaled r rotated a shifted z;
%<endgroup enddef;
arcd((4u,3u),30u,45,105)(pcc);draw pcc;
p=(0,0)--(17u,0);
draw p rotated 40 shifted(4u,3u)lw1;
draw p rotated 100 shifted(4u,3u)lw1;
drawpt((4u,3u),2pt);
arcd((15u,3u),6u,-90,30)(tournant);
canne=(0,0)--(15u,0)&tournant;
%draw canne; % test
za=(35u,0)+(0,3u)+polar(3u,30);
zb=(5u,0)-(0,3u)-polar(3u,30);
triangle=canne shifted(20u,0)..
  canne shifted za rotatedaround(za,120)..
  canne shifted zb rotatedaround((20u,0),-120)..cycle;
fill triangle cl(0.7);
draw triangle lw(5pt) cl(0.4);
draw canne shifted(20u,0)lw(1pt);
endfig;
```



### 3.5.2 Autres commandes concernant les courbes

On signale d'abord que l'on a utilisé sans le dire le fait que tout ce qui concerne les intersections de chemins est encore valable si les chemins sont des droites (les polynômes  $X(t)$  et  $Y(t)$  sont alors du premier degré).

On a vu que l'on peut imposer la direction de la tangente orientée en chaque point guide des courbes mais on peut aussi imposer les directions des deux tangentes orientées de part et d'autre des points guides; pour cela on donne les valeurs des angles de part et d'autre du point-guide [25] :

```
draw etc..{dir170}z4{dir-140}..etc;
```

ce qui produit un point anguleux en  $z4$  (angle de  $50$  degrés).

La longueur paramétrique  $lp$  et la longueur d'arc  $l$  d'un chemin  $p$  sont respectivement données par [57], [60] :

```
lp=length p; et l=arclength p;
```

Par exemple, la longueur paramétrique de `subpath(t1,t2)` est  $t2-t1$ , celle d'une courbe ouverte passant par  $n$  points guides est  $n - 1$ .

La valeur `tz` du paramètre qui définit le point `z` du chemin `p` est donnée par l'opérateur [60] :

```
tz=arctime z of p;
```

On dispose également de deux commandes qui permettent d'effacer la partie du chemin `p` se trouvant après (resp. avant) le point d'intersection avec le chemin `q`; la partie restante `ppart` est [58] :

```
ppart=p cutafter q; (resp. cutbefore q;
```

S'il y a plusieurs points d'intersection entre les chemins, la partie supprimée est celle située après le point d'intersection ayant la plus grande valeur du paramètre dans le cas de `cutafter` et inversement dans le cas de `cutbefore`. On utilisera cette possibilité dans l'exemple suivant.

### 3.5.3 Explications concernant des points de syntaxe

Le dernier exemple, figure 14, ainsi que la macro `arcd` qui y est introduite, appellent les remarques importantes qui suivent.

On a déjà vu dans ces deux premiers chapitres des macros qui, en plus des paramètres dits « expressions » représentant des variables des 9 types possibles (dont seulement 6 ont été rencontrés à ce stade), contiennent des paramètres *texte*; dans le cas de la macro `axespapiermillii`, ce paramètre *texte* consiste en deux macros avec paramètres fixés. Dans le dernier exemple, la macro `arcd` a un paramètre *texte* qui est une liste de caractères : en effet, au moment où l'on définit cette macro `arcd`, `parg` n'est pas encore le nom d'un chemin mais une simple liste de caractères : cette liste ne peut donc pas faire partie du groupe de paramètres *expressions*.

On a défini plusieurs macros qui tracent des chemins ou des points (`drawpt`, `drawrule`, `axespapiermillii`, etc.); la macro `arcd` définit, sans le tracer, un chemin (un arc de cercle très précisément) dont le nom est la liste de caractères donnée pour l'argument du type *texte* : dès que l'on a écrit la macro avec tous les arguments nécessaires, l'argument du type *texte* donné est devenu le nom de l'arc de cercle construit (la simple liste est devenue une variable *chemin*). Bien entendu on peut tracer ce chemin mais on peut aussi l'utiliser pour construire d'autres chemins comme cela est le cas dans l'exemple.

Dans cette macro `arcd`, on remarque les nouvelles commandes `begingroup` et `endgroup` qui limitent un groupement et la commande `save` dont le rôle va être expliqué sur un exemple. Dans le texte dit de remplacement de la macro `arcd`, on a besoin de variables intermédiaires; on prend pour exemple la variable `c=a-b`; : elle ne doit jouer aucun rôle en dehors de l'intérieur de la macro; pour qu'elle ne perturbe pas le code environnant de la macro, la commande `save c`; sauvegarde la valeur de la variable `c` à l'entrée du groupement et la restitue à la sortie du groupement; il est évident que cela n'a d'utilité que si on utilise une variable nommée `c` dans le code environnant, par contre, cela permet d'utiliser la macro sans avoir à connaître les noms des variables utilisées en interne. Après la commande `save` on trouve les déclarations faites suivant les règles habituelles.

La dernière remarque porte sur la construction du chemin fermé avec trois

exemplaires du chemin nommé `canne` adhéqatement déplacés et tournés avec la translation `shifted` et la rotation `rotatedaround`. Les éléments du chemin composé sont concaténés avec `..` au lieu de `&` et la fermeture du chemin est assurée par l'option `..cycle`; en effet, lors de la construction d'un chemin composé avec d'autres chemins, il se peut que l'extrémité d'un élément ne coïncide pas exactement avec l'origine de l'élément suivant et il faut alors remplacer `&` par `..` sinon on a le message « `& will be changed to ..` » qui précise que `META-FONT` a fait lui-même le remplacement. On a aussi un tel exemple sur la figure 9 où les deux dernières commandes, partie 4, permettent de voir que les points `z10` et `z11` ne coïncident pas rigoureusement. Pour voir le message, il suffit de remplacer `..` par `&` dans la ligne `pa7=pa5..reverse etc`; et de regarder le message à la console.



# La machinerie METAPOST

Dans les deux premiers chapitres, on a appris à faire des figures simples assez facilement ; cependant, au chapitre 3, l'écriture des commandes est devenue de plus en plus complexe et même un peu mystérieuse par certains aspects. METAPOST est un vrai langage informatique et, après les débuts destinés à motiver le lecteur, on doit aborder de façon logique et rigoureuse les éléments et la structure de ce langage. Nous allons examiner successivement les différents éléments en proposant toujours des exercices :

- les opérateurs,
- les variables,
- les variables internes,
- les équations et les affectations,
- les commandes,
- les comparaisons et les boucles,
- les macros.

## 4.1 Opérateurs

Les opérateurs produisent, à partir de un ou deux éléments, un troisième élément ; ces éléments sont les variables qui sont l'objet de la section suivante. Voilà des exemples d'opérateurs rencontrés dans les deux premiers chapitres :

- 1) nombre + nombre = nombre
- 2) nombre \* nombre = nombre
- 3) nombre \* paire = paire
- 4) `round` nombre = nombre
- 5) `xpart` paire = nombre
- 6) `cosd` nombre = nombre
- 7) nombre < nombre = booléenne
- 8) chemin `intersectionpoint` chemin = paire
- 9) `point` nombre of chemin = paire
- 10) `subpath` paire of chemin = chemin
- 11) chemin `shifted` paire = chemin
- 12) paire `shifted` paire = paire

On remarque d'abord qu'il y a des opérateurs qui prennent une variable et d'autres qui en prennent deux; tous donnent une variable en résultat. On reconnaît l'addition et la multiplication habituelle des nombres réels aux lignes 1) et 2). Dans ces deux lignes et dans la dernière, les trois variables sont du même type; à l'opposé il y a des cas où les variables concernées (deux ou trois suivant les opérateurs) sont de type différents; c'est le cas des lignes 5) et 9); il y a enfin les cas intermédiaires : lignes 7), 8), 10) et 11) où deux des trois variables sont d'un même type.

La liste des opérateurs avec les arguments qu'ils peuvent prendre, les résultats qu'ils peuvent donner et l'explication de leur action se trouve dans un long tableau du manuel cité [112–119]; le lecteur s'y reportera chaque fois qu'il voudra une précision concernant un opérateur.

## 4.2 Variables

Il faut noter que, dans les exercices suivants (fichier de figures `figd.mp`), on doit parfois déclarer les variables nombre dont la déclaration est facultative (cas où ces variables sont utilisées plusieurs fois). Cette déclaration, à faire dès la deuxième utilisation, efface la valeur de la variable correspondante si elle a déjà été utilisée auparavant, en particulier dans une figure précédente. Cette remarque est sans objet pour les variables `z[]`, `x[]` et `y[]` dont les valeurs sont effacées par la commande `beginfig`. Pour les autres variables, leur redéclaration assure de la même manière l'effacement des valeurs prises antérieurement.

On va examiner les neuf types de variables en donnant leurs propriétés (si ce n'est déjà fait) et les principaux opérateurs qui peuvent leur être associés.

On commence par une commande et deux opérateurs qui concernent toutes les variables et qui sont utiles dans le débogage :

`show a`; [28] montre à la console la valeur de la variable `a` quel que soit son type ainsi que les chaînes présentées entre "... " (début de l'exercice 1);  
`known a`; [39] vaut `true` si la variable `a` a une valeur particulière et `false` dans le cas contraire : autrement dit `show known a` affiche respectivement `true` ou `false` (milieu de l'exercice);  
`numeric a`; [39] vaut `true` si la variable `a` est du type nombre et `false` dans le cas contraire; on dispose des huit opérateurs similaires pour les huit autres types de variables : `pair`, `color`, etc. (fin de l'exercice).

Dans ce chapitre, on va trouver des exercices « à la console » dont le résultat va s'afficher à l'écran; s'il y a trop de lignes, il faut éditer le fichier `figd.log` pour voir l'ensemble des résultats.

```
beginfig(1);show "Exercice 1";
a=1.25; show a; pair zz; zz=(5,2.5); show zz; show red;
path p; p=(0,0)--(1,0); show p;
transform T; T=identity shifted(10,0); show T;
show known b; b=5; show known b;
show known q; path q; show known q; q=(0,0)--(0,1); show known q;
show pair a;show numeric a;show numeric zz;show pair zz;show zz;
endfig;
```

### 4.2.1 Type nombre

Principaux opérateurs du type  $a \text{ o}_p b \rightarrow c$  et  $\text{o}_p a \rightarrow c$  où  $a$ ,  $b$  et  $c$  sont des variables du type nombre [33] (début de l'exercice 2) :

- addition, soustraction, multiplication, division (sauf par 0), division entière, addition pythagoricienne, exponentiation, (+, -, \*, /, div, ++, \*\*),
- racine carrée, valeur absolue, plus petit entier supérieur ou égal, plus grand entier inférieur ou égal, cosinus et sinus (angle en degrés), plus proche entier (sqrt, abs, ceiling, floor, cosd, sind, round).

Principaux opérateurs de comparaison du type  $a \text{ o}_p b \rightarrow \mathbf{b}$  ou  $\text{o}_p b \rightarrow \mathbf{b}$  où  $a$  et  $b$  sont des variables nombre et  $\mathbf{b}$  est une variable du type booléen (fin de l'exercice) :

- égalité, plus petit, plus grand, différent, plus petit ou égal, plus grand ou égal, imparité (=, <, >, <>, <=, >=, odd).

D'autres opérateurs prenant ou donnant des variables du type nombre et aussi des variables d'autres types seront présentés avec ces autres types de variables.

```
beginfig(2);show "Exercice 2";
numeric a,b; a=1; b=2; c=a+b; show c; d=b**b; show d; e=cosd 60;show e;
f=sqrt d;show f; g=abs -25; show g; h=round 5.43; show h;
if b=c : show "oui b=c" else : show "non b <> c" fi;
if b<c : show "vrai" else : show "faux" fi;
if odd b : show "b est impair" else : show "b est pair" fi;
endfig;
```

### 4.2.2 Type paire

Opérateurs du type  $z_a \text{ o}_p z_b \rightarrow z_c$  et  $a \text{ o}_p z_a \rightarrow z_c$  où  $z_a$ ,  $z_b$  et  $z_c$  sont des variables du type paire [34] et  $a$  est une variable nombre (début de l'exercice 3) :

- addition, soustraction, multiplication par variable nombre et division par une variable nombre différente de 0 (+, -, \*, /).%

Principaux opérateurs de comparaison du type  $z_a \text{ o}_p z_b \rightarrow \mathbf{b}$  où  $\mathbf{b}$  est une variable booléenne (milieu de l'exercice) :

- égalité, différent (=, <>).

Principaux autres opérateurs (fin de l'exercice) :

- `xpart z`, `ypart z`, `abs z`, `round z` donnent respectivement le premier nombre de la paire  $z$ , le deuxième nombre de  $z$ , le module de  $z$ , la paire des deux plus proches entiers des deux nombres de  $z$ .
- `angle z`, `unit vector z`, `dir u` donnent respectivement l'angle que le vecteur de composantes `xpart z` et `ypart z` fait avec l'axe des  $x$ , la paire des composantes du vecteur unitaire suivant le vecteur de composantes `xpart z` et `ypart z`, la paire des composantes du vecteur unitaire faisant l'angle  $u$  avec l'axe des  $x$ .

```
beginfig(3);show "Exercice 3";
pair za,zb,zc,zd,zo,zu; za=(3,4); zb=(5.1,5.9); zc=za+zb; show zc;
numeric a,b,c; a=5; zd=a*za; show zd;
if za<>zc : show "vrai"; else : show "faux"; fi
b=xpart za; c=ypart za; show b,c; show abs za; zo=round zb; show zo;
zu=(4,4); show angle zu; show unitvector za; show dir 45;
endfig;
```

### 4.2.3 Type couleur

Les couleurs sont codées à l'aide du système RGB : une variable du type couleur [34] est un triplet  $(\mathbf{a}, \mathbf{b}, \mathbf{c})$  de trois nombres inférieurs ou égaux à 1.

Opérateurs du type du type  $c_a \text{ o}_p c_b \rightarrow c_c$  et  $a \text{ o}_p c_a \rightarrow c_c$  où  $c_a$ ,  $c_b$  et  $c_c$  sont des variables du type couleur et  $a$  est une variable nombre :

- addition, soustraction, multiplication par une variable nombre et division par une variable nombre différente de 0 ( $+$ ,  $-$ ,  $*$  et  $/$ ).

Principaux opérateurs de comparaison du type  $c_a \text{ o}_p c_b \rightarrow \mathbf{b}$  où  $\mathbf{b}$  est une variable booléenne :

- égalité, différent ( $=$ ,  $<>$ ).

Autres opérateurs éventuellement utiles :

- `redpart c`, `bluepart c` et `greenpart c` donnent respectivement les composantes rouge, bleue et verte de la couleur  $c$ .

Il est possible que les différentes opérations faites sur les couleurs conduisent à des triplets dont un ou plusieurs éléments dépassent 1 ; dans ce cas, METAPOST multiplie le triplet par le nombre qui ramène à 1 le plus grand élément du triplet ; par exemple :  $(0.8, 0.6, 0.2) + (0.7, 0.6, 0.4) = (1.5, 1.2, 0.6)$  sera multiplié par  $1/1.5$  ce qui donne  $(1, 0.8, 0.4)$ <sup>(1)</sup>.

### 4.2.4 Type transformation

D'une façon complètement générale, on peut représenter une transformation affine du plan par un sextuplet  $(t_x, t_y, t_{xx}, t_{xy}, t_{yx}, t_{yy})$  tel que :

$$(x, y) \rightarrow (t_x + t_{xx}x + t_{xy}y, t_y + t_{yx}x + t_{yy}y)$$

Heureusement, les transformations utiles peuvent s'obtenir très simplement à partir des transformations élémentaires suivantes [60-63], [M 141-145] (début de l'exemple 4) :

- `shifted(a,b)` : translation de  $a$  suivant les  $x$  et de  $b$  suivant les  $y$  (translation de vecteur  $(a, b)$  très souvent utilisée, exemple : fig. 1.4, sect. 1.3),
- `rotated u` : rotation autour de l'origine d'un angle  $u$  (sens positif habituel),
- `slanted a` :  $(x, y) \rightarrow (x + ay, y)$  (utilisée initialement pour pencher les caractères),
- `scaled a` :  $(x, y) \rightarrow (ax, ay)$ , agrandissement ou réduction suivant les valeurs du facteur d'échelle  $a$ ,
- `xscaled a` :  $(x, y) \rightarrow (ax, y)$ , changement d'échelle suivant les  $x$  seulement,
- `yscaled a` : changement d'échelle suivant les  $y$  seulement,
- `zscaled(a,b)` : rotation et homothétie qui transforme  $(0, 1)$  en  $(a, b)$  (rotation d'un angle égal à l'angle que le vecteur  $(a, b)$  fait avec l'axe des  $x$  suivie d'une homothétie de facteur  $\sqrt{a^2 + b^2}$ ),
- `rotatedaround(z,u)` : rotation d'angle  $u$  autour du point  $z$ ,

<sup>(1)</sup> Autre exemple :  $(3, 3, 3)$  sera remplacé par  $(1, 1, 1)$  car il ne peut y avoir de blanc plus blanc que blanc, excepté dans le monde des lessives comme l'avait, en son temps, remarqué Coluche !



- `reflectedabout(z1,z2)` : réflexion par rapport à la droite définie par les points  $z_1$  et  $z_2$ .

On retrouve un résultat bien connu (milieu de l'exemple) :

`rotatedaround(z,u)`

donne le même résultat que

`shifted -z` suivie de `rotated u` suivie de `shifted z`

Le chemin transformé `pp` s'écrit, si `p` est le chemin initial :

- `p shifted -z rotated u shifted z`

L'application d'une succession de transformations prédéfinies ci-dessus, s'écrit donc sans notation spécifique.

Les transformations peuvent être appliquées à une paire représentant un point du plan, à un chemin ou à un dessin ; on obtient alors respectivement une autre paire représentant un autre point du plan, un autre chemin ou un autre dessin : on constate alors que les transformations jouent le rôle d'un opérateur du type qui intervient dans une équation telle que  $p \circ_p \rightarrow p$  où  $p$  est un point, un chemin ou un dessin.

Mais les transformations sont aussi des variables [34] que l'on déclare, définit et utilise comme suit :

- `transform Ta,Tb,Tc,T[]` ;
- `T=identity shifted za rotated u etc reflectedabout(zb,zc)` ;
- `zb=za transformed Ta; pb=pa transformed Ta`; (idem avec un dessin),
- `Tc=Ta transformed Tb`;
- `inverse Ta` est la transformation inverse de `Ta`.

L'avant dernière ligne (fin de l'exemple) montre la loi de composition (non commutative<sup>(2)</sup>) des transformations avec l'opérateur `transformed`. La dernière ligne définit la transformation inverse avec l'opérateur `inverse` (fin de l'exemple); dans l'exemple choisi, on ne retrouve pas exactement la transformation identifiée à cause des approximations de calcul (le lecteur doit commencer à être familier avec ces effets).

```
beginfig(4);show "Exercice 4";
show "debut (exemples de transformations)";
show identity;
show identity shifted(20u,10u);
show identity rotated 45;
show identity rotatedaround((10u,30u),60);
show "milieu (remarque)";
show identity rotatedaround((20u,10u),45);
show identity shifted-(20u,10u) rotated 45 shifted(20u,10u);
show "fin (composition sur l'exemple de la remarque)";
pair zr;transform Ta,Tb,Tc,Td,Te;zr=(20u,10u);
Ta=identity shifted-zr;Tb=identity rotated 45;Tc=identity shifted zr;
Td=Ta transformed Tb transformed Tc; show Td;
Te=Td transformed inverse Td;show Te;
endfig;
```

Du point de vue mnémotechnique, il suffit de se souvenir que les transformations

---

<sup>(2)</sup> Le lecteur se souvient probablement de l'heureux temps où un professeur lui a fait découvrir qu'une loi de composition n'est pas nécessairement commutative avec l'exemple d'une rotation et d'une translation !

autres que les transformations prédéfinies (ci-dessus) sont utilisées précédées de `transformed` et que toute définition d'une transformation avec les transformations prédéfinies débute par la transformation `identity`.

#### 4.2.5 Type chemin

Les variables du type chemin [34] peuvent :

- être transformés par les transformations définies à la sous-section précédente, par exemple : `p shifted(a,b)` donne le chemin `p` déplacé de `a` suivant les  $x$  et de `b` suivant les  $y$ ,
- être concaténés par l'opérateur `&` : si `pa` et `pb` sont des chemins, alors `pa & pb` est un chemin si et seulement si l'extrémité de `pa` coïncide avec l'origine de `pb` ; il peut arriver que cette coïncidence ne soit pas exacte (arrondis de calcul) ; dans ce cas, si l'on veut former un chemin fermé (par exemple pour être rempli), alors METAPOST remplace `&` par `..`, ce qui est invisible (le chemin rajouté entre les points qui ne coïncident pas n'est long que de quelques dix millièmes de pt, c'est le cas dans la figure 3.9 du chapitre 3, chemins `pb7` et `pb8`),
- définir un chemin fermé grâce à la macro `buildcycle(pa,pc, etc)` (sect. 3.4.3) sans que soit nécessaire le calcul des points d'intersection des chemins. On dispose aussi d'un opérateur `cycle` [39] pour tester si un chemin est ou n'est pas fermé ; de toute manière, si l'on tente de remplir (avec `fill`) un chemin non fermé, METAPOST le signale.

Autres opérateurs déjà rencontrés :

- `point t of p` (sect. 3.4.1) donne le point de `p` au temps `t`,
- `subpath(t1,t2) of p` (sect. 3.4.1) donne la partie de `p` comprise entre les points aux temps `t1` et `t2`,
- `pa intersectionpoint pb` et `pa intersectiontimes pb` (sect. 3.4.2) permettent de trouver les points d'intersection des chemins `pa` et `pb`,
- `pa cutafter pb` et `pa cutbefore pb` (sect. 3.5.2) permettent de gommer la partie de `pa` située après et avant le point d'intersection avec `pb`,
- `direction t of p` et `directiontime t of p` (sect. 3.4.4) permettent les tracés de tangentes au chemin `p` ;
- `length p`, `arclength p`, `arctime p` (sect. 3.5.1) servent à gérer le chemin `p`.

#### 4.2.6 Type chaîne

Les valeurs sont attribuées aux variables du type chaîne [35] en les limitant par des *double-quote* : après déclaration, on écrit par exemple : `ch="abcdef"` ;

Principaux opérateurs [112–119] :

- opérateur de concaténation `&`,
- opérateurs de comparaison : égalité , différent, plus petit, plus grand, plus petit ou égal, plus grand ou égal (`=`, `<>`, `<`, `>`, `<=` et `>=`) ; dans les quatre derniers cas, c'est le code ASCII du premier caractère qui est seul pris en compte.

Autres opérateurs utiles présentés dans l'exemple suivant :

- `ASCII ch` ; donne le code ASCII du premier caractère de `ch`,
- `char n` ; donne le caractère de code `n`,

- `substring(a,b) of ch`; extrait la sous-chaîne dont le premier caractère est à la position  $a$  et le dernier à la position  $b - 1$  (attention : dans les chaînes, la première position est la position 0, `substring(2,4)"abcde"` donne `cd`),
- `label(ch,z)`; permet d'écrire la chaîne `ch` à la position `z`.
- `str aaa`; (où `aaa` est une simple liste de caractères) donne la chaîne `"aaa"`.

```
beginfig(5);show "Exercice 5"
string s[];s1="ab";s2="cde";s3=s1&s2;show s3;
s5="ab"; if s1=s5 : show "vrai" else : show "faux" fi;
s6="ac"; if s1=s6 : show "vrai" else : show "faux" fi;
s7="bb"; if s1<s7 : show "vrai" else : show "faux" fi;
show ASCII "ABC";show char65;s8=substring(2,4) of "abcde";show s8;
s9=str uuu;show s9;
endfig;
```

### 4.2.7 Type booléen

Les variables booléennes [35] apparaissent la plupart des fois comme le résultat d'un opérateur de comparaison (exemple : fig. 3.13) ou d'un opérateur de test (`known`, `numeric`, `path`, etc., sect. 4.2). On les compose avec les opérateurs [112–119] `and` et `or`; la négation s'obtient avec l'opérateur `not` (début exercice suivant). On leur attribuer des valeurs en utilisant les parenthèses comme délimiteurs; par exemple, après déclaration, on écrit (fin exercice) :

- `ba=(a<b)`; `bb=(b<c)`; puis `if ba and bb : etc ;`

```
beginfig(6);show "Exercice 6";
k=1;l=3;m=5;n=2;
if (k<l) and (l<m) : show "k,l,m en ordre" else : show "k,l,m en desorde" fi;
if (k<l) and (l<n) : show "k,l,n en ordre" else : show "k,l,n en desorde" fi;
if (k<l) and not(l<m) : show "k,l,m en ordre" else : show "k,l,m en desorde" fi;
boolean ba,bb,bc;ba=(k<l);bb=(l<m);bc=ba and bb;show bc;
endfig;
```

### 4.2.8 Type dessin

Un ensemble de chemins tracés et/ou remplis constitue un dessin. Après déclaration, on peut attribuer à une variable de type dessin [35] une « valeur » constituée par un dessin donné.

On dispose d'une variable dessin prédéfinie nommée `currentpicture` pour cela [74], [M 114]; elle contient tout ce qui a été tracé et/ou rempli lorsque la variable est invoquée (elle a été utilisée pour découper une partie du dessin préalablement fait, figure 2.2 du chapitre 2). On dispose aussi d'une constante prédéfinie `nullpicture` qui contient un dessin vide.

On donne la « valeur » constituée par les chemins qui ont été précédemment tracés et/ou remplis à la variable dessin préalablement déclarée en écrivant :

- `picture D;D=currentpicture;currentpicture=nullpicture;`

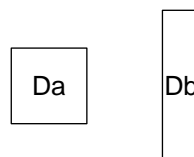
Après cette ligne, la variable `D` contient le dessin préalablement fait et la variable `currentpicture` ne contient plus de dessin : en d'autres termes si `currentpicture` représente le contenu du tableau, la première partie de la ligne fait une copie nommée `D` de ce qu'il y a sur le tableau; la seconde partie efface le tableau où l'on peut commencer un autre dessin !

Dans l'exemple suivant, on montre comment on peut préparer des éléments d'une figure très simplement, puis les assembler pour former la figure complète : le principe de base consiste à faire un élément, le sauvegarder dans une variable dessin, égaliser `currentpicture` à `nullpicture` « pour effacer le tableau » et construire l'élément suivant, etc. ; quand tous les éléments sont terminés, on les assemble par concaténation avec la syntaxe suivante (étant entendu que la variable `currentpicture` a été complètement « vidée ») [73], [M 144] :

- `addto currentpicture also Pa;`  
`addto currentpicture also Pb shifted za;`  
`addto currentpicture also etc ;`

et ainside suite jusqu'à la construction complète de la figure.

```
beginfig(7);show "Exercice 7";
picture Da,Db,Dc;
draw (0,0)--(10u,0)--(10u,10u)--(0,10u)--cycle;
Da=currentpicture;currentpicture:=nullpicture;
draw (0,-5u)--(5u,-5u)--
      (5u,15u)--(0,15u)--cycle;
Db=currentpicture;currentpicture:=nullpicture;
addto currentpicture also Da;
addto currentpicture also Db shifted(20u,0);
endfig;
```



#### 4.2.9 Type plume

Ce type de variable n'est pas abordé dans ce manuel puisque l'on s'est fixé l'objectif de faire des figures avec simplement des traits d'épaisseur variable, ce qui s'obtient directement avec la plume par défaut (`pencircle`) dont on fait varier le diamètre avec une option du type (sec. 2.2.1) :

- `withpen pencircle scaled 0.8pt`

### 4.3 Variable et équations ; variables internes et affectations

Il est grand temps de faire le point !

#### 4.3.1 Variables (ordinaires)

Ce sont les variables déjà examinées dans la section 4.2 : elles sont d'abord déclarées (sauf dans quelques cas bien précisés, sect. 2.3.3) puis déterminées par METAPOST avec les équations que l'on écrit (exercice suivant avec des paires) :

- `z1=(3,5)` ; METAPOST résoud l'équation et attribue la valeur (3,5) à la variable `z1` (semble tout à fait trivial) ;
- `z1+z2=(5,9)` ; `z1-z2=(3,3)` ; METAPOST attribue la valeur (4,6) à `z1` et la valeur (1,3) à `z2` (c'est moins trivial) ;
- `z1+z2=(5,9)` ; `z1-z2=(3,3)` ; `z1=(4,3)` ; va donner un message d'incompatibilité car les deux premières équations déterminent une valeur de `z1` différente de celle donnée par la dernière équation ; par contre, si la dernière équation est `z1=(4,6)`, il n'y aura pas de message d'erreur ;

- `pair za,zb;za+zb=(5,9);za-zb=(3,3);pair za;za=(4,3)` ne va pas donner de message d'erreur car la déclaration `pair za;` fait que METAPOST oublie la valeur qu'il a attribuée à `za`; cette propriété permet, dans une boucle où les valeurs intermédiaires des variables n'ont aucune utilité après la boucle, d'utiliser les mêmes variables à chaque pas de la boucle à condition de les redéclarer dans la boucle avant leur nouvelle utilisation (voir figure 3.13, sect. 3.4.4)

```
beginfig(8);show "Exercice 8";
z1+z2=(5,9);z1-z2=(3,3);show z1,z2;
%z2=(4,3); % poucenter apres essai
z1=(4,6);
pair za,zb;za+zb=(5,9);za-zb=(3,3);show za;pair za;za=(4,3);
endfig;
```

### 4.3.2 Variables internes

Certaines variables internes sont prédéfinies et d'autres sont introduites par l'utilisateur à l'aide de la déclaration `newinternal`; leur valeur, nulle à leur déclaration, est donnée, on dit affectée, par l'utilisateur; voici quatre exemples :

- `ahlength` est une variable interne (valeur de la longueur de la pointe des flèches, sect. 2.2.4) dont la valeur par défaut est 4bp; on l'a changée en écrivant `ahlength=2pt` dans le préambule du fichier des figures du chapitre 2; on peut revenir à la valeur par défaut en écrivant `ahlength:=4bp`; une autre possibilité consiste à écrire `ahlength:=ahlength-2bp` et `ahlength:=ahlength+2bp` pour retrouver la valeur initiale;
- `linecap` est une variable interne (forme de la terminaison des traits, sect. 2.2.2) dont la valeur par défaut est `rounded` mais on peut aussi choisir une autre valeur en écrivant `linecap:=squared`; ici `butt`, `rounded` et `squared` sont des constantes prédéfinies que peut prendre la variable interne `linecap` (l'intérêt de la définition de ces constantes, qui valent respectivement 0, 1, et 2, est simplement d'ordre mnémotechnique);
- `currentpicture` est une variable interne prédéfinie qui contient l'ensemble des chemins tracés et/ou remplis précédemment (sect. 2.1 et 4.2.8);
- `newinternal dist;dist:=3pt;` définit une variable interne utilisateur représentant une distance pour une construction spécifique (par exemple, distance entre les deux filets d'un double cadre).

Les variables internes prédéfinies et les constantes prédéfinies sont respectivement données par les tableaux du manuel cité [107–109] et [110–111].

### 4.3.3 Remarques concernant les affectations

Les variables internes, prédéfinies ou définies par l'utilisateur, sont toujours affectées au niveau METAPOST pour les premières et par l'utilisateur pour les secondes. On change à la demande leur valeur par une nouvelle affectation (`:=`).

On peut aussi affecter une valeur à une variable (ordinaire), voici quelques cas possibles proposés surtout dans un but pédagogique (exercice suivant) :

- si `x1` est déterminée par des équations et vaut 4 et si on écrit `x1:=5`; ou `x1:=x1+1`; après les équations, alors `x1` prend la valeur 5;
- si on écrit `x11:=5`; avant les équations qui donnent pour solution `x11=4`;

on a un message d'incompatibilité mais, si on écrit `x21:=4;` avant les équations qui donnent pour solution `x21=4;` alors on a un message de redondance.

- si on écrit `x31:=x31+1;` avant les équations qui déterminent `x31` et `x32`, ces équations donnent encore la même solution ; en effet, comme `x31` n'a pas encore de valeur, l'équation `x31:=x31+1;` ne peut lui en donner une ; c'est ce qui est confirmé par la réponse "false" à la commande `show known x31;`.

```
beginfig(9);show "Exercice 9";
x1+x2=7;x1-x2=1;show x1,x2;x1:=5;show x1;
%x11:=5;x11+x12=7;x11-x12=1;show x11, x12 % pourcenter apres essai
x21:=4;x21+x22=7;x21-x22=1;show x21,x22;
x31:=x31+1;show known x31;x31+x32=7;x31-x32=1;show x31,x32;
endfig;
```

## 4.4 Commandes et macros

### 4.4.1 Commandes

On va donner la liste des commandes déjà utilisées ou qui vont être utilisées dans ce document. Pour débiter en METAPOST, on peut se limiter à une partie de la liste complète de ces commandes [120]. En effet, cette liste contient des primitives, par exemple `addto`, que l'on utilise rarement dans la mesure où le fichier `plain.mp` (chargé par défaut au lancement de METAPOST) comprend certaines commandes, `draw` et `fill` par exemple, qui sont en fait des macros définies à partir de cette primitives.

Le tableau suivant contient des primitives et des macros définies dans le fichier `plain.mp`, macros jouant le rôle de commandes (d'où leur présence dans cette liste).

Commande	Action
<code>addto...also...</code>	primitive pour « assembler » les éléments d'un dessin
<code>btex...etex</code>	entourent le matériel que T <sub>E</sub> X doit composer préalablement
<code>clip</code>	découpe la partie de la figure courante intérieure à un contour
<code>draw</code>	trace un chemin ou un dessin
<code>drawarrow</code>	trace un chemin terminé par une flèche
<code>fill</code>	rempli un contour fermé
<code>filldraw</code>	trace et remplit un contour fermé (usage délicat, macro associée : <code>unfilldraw</code> )
<code>newinternal</code>	définit une nouvelle variable interne
<code>save</code>	sauvegarde la valeur d'une variable au début de l'exécution d'un macro et redonne cette valeur à la fin de l'exécution si la macro est limitée par une paire <code>begingroup...endgroup</code>
<code>show</code>	montre à la console les expressions (valeur ou code correspondant, cf. section 4.2)
<code>str</code>	transforme un suffixe en une chaîne de caractères contenant les mêmes caractères
<code>undraw</code>	efface un chemin ou un dessin
<code>unfill</code>	efface l'intérieur d'un chemin fermé
<code>verbatimtex</code>	remplace <code>etex</code> pour passer des commandes à T <sub>E</sub> X

Pour le débogage, il a beaucoup d'autres commandes et variables internes disponibles mais, au niveau débutant, on peut se limiter à la commandes `show` éventuellement accompagnée de l'opérateur `known` (sect. 4.2); voici trois exemples typiques d'utilisation :

- `show a, za;` montre à la console les valeurs du nombre `a` et de la paire `pa`, ce qui permet de réaliser que l'on a mal expliqué à METAPOST ce que l'on voulait faire;
- `show ca intersectiontimes cb;` donne `(-1,-1)` si les chemins `ca` et `cb` ne se coupent pas;
- `show "oui";` et `show "non";` insérés respectivement dans chacune des suites d'instructions d'une boucle permettent de découvrir parfois que tout se déroule bien différemment de ce que l'on espérait !

#### 4.4.2 Macros de base

Le tableau suivant donne les macros de base [121–122] les plus utilisées par le débutant et définies dans le fichier `plain.mp`. Il faut noter que le tableau cité contient en plus les macros de base destinées à la construction de boîtes et définies dans le fichier `boxes.mp`.

Commande	Action
<code>bbox</code>	donne le contour entourant un dessin
<code>beginfig</code>	marque le début d'une figure
<code>buildcycle</code>	construit un contour fermé avec des chemins concourants
<code>dashpattern</code>	crée un motif de traitillé
<code>dotlabel</code>	trace un point et place le label correspondant
<code>endfig</code>	marque la fin d'une figure
<code>incr</code>	incrémente une variable nombre ( <code>decr</code> fait l'inverse)
<code>label</code>	place les labels
<code>max</code>	donne le maximum d'une suite de nombres ( <code>min</code> donne le minimum)
<code>thelabel</code>	désigne la boîte fantôme (au sens de $\text{T}_{\text{E}}\text{X}$ ) de mêmes dimensions que celle créée par la commande <code>label</code> de même argument

#### 4.4.3 Construction de macros

On a déjà vu les cas simples de macros sans paramètre et de macros avec paramètres du type `expr` qui signifie que ces paramètres sont des variables des différents types (sect. 2.3.3).

On revient sur les macros avec paramètre du type `text`; chaque paramètre de ce type doit être limité par un couple de parenthèses car il peut contenir des virgules (qui ne peuvent plus alors jouer le rôle de séparateur). Ces arguments peuvent être une suite d'instructions incluant des macros avec paramètres fixés (macro `axespapiermilli`, sect. 2.3.4); ils peuvent être aussi une liste de caractères qui devient un nom de variable lors de l'exécution (macro `arcd`, sect. 3.5.1); ils peuvent être une liste de *suffixes* séparés par des virgules utilisée pour l'incrémentation dans une boucle, macro `beginbox` dans le fichier `boxes.mp`.

Dans le cas de la macro `arcd`, l'argument du type `text` est d'un type particulier car il devient le nom d'un chemin ; par exemple, il ne peut contenir ni virgule ni point-virgule : c'est pour cela qu'a été introduit le type de paramètre `suffix` [84] moins général que le type `text`. La macro `arcd` peut fonctionner en prenant un argument du type `suffix` au lieu du type `text`. Ce type d'argument est utilisé par les macros `vardef` examinées au paragraphe suivant.

Les macros du type `vardef` [85] sont ainsi nommées parce que, dans leur définition, `def` est remplacé par `vardef`. On peut dire que ces macros sont des « macros-tableau » où l'indice du tableau est un suffixe ou bien des « macros-variables » dont la variable est un suffixe ; c'est le cas de `boxit`, `circleit`, `diamit`, etc. (sect. 5.1) et `donnees`, `tracebf`, `tracebe`, etc. (sect.5.3).

Il y a enfin les macros primaires, secondaires et tertiaires [88-89] dont le but est essentiellement de permettre l'utilisation de paramètres sans délimiteurs<sup>(3)</sup> : les paramètres des types `expr` et `suffix` sont placés entre parenthèses et séparés par des virgules ; ceux du type `text` sont individuellement placés entre parenthèses ; mais on peut aussi imaginer d'autres définitions ayant d'autres syntaxes d'utilisation. Ce type de macro sort largement du cadre de ce petit manuel (ainsi que certains aspects des macros `vardef`) : on va simplement « illustrer » la motivation avec un exemple. Imaginons que l'on définisse le produit spécial  $a * b - (a + b)$  où  $a$  et  $b$  sont des nombres et pour lequel on veut que la syntaxe d'utilisation soit du type de celle de la multiplication usuelle, c'est-à-dire `a prodspe b` ; au lieu de `prodspe(a,b)` ; METAFONT prévoit pour ce type de définition `secondarydef a prodspe b = a*b-(a+b) enddef` ; en remplacement de `def prodspe(expr a,b)= a*b-(a+b) enddef` ; Le résultat est le même excepté que, dans l'ordre des opérations, les définitions primaires, secondaires et tertiaires sont exécutées dans le même ordre que exponentiation, multiplication et addition.

## 4.5 Boucles et tests

Ces instructions existent dans la plupart des langages ; on en donne simplement la syntaxe METAPOST.

### 4.5.1 Boucles

Les intructions de boucles sont [91-92] [M 172] :

- si l'index est un nombre :

```
for i=a step s until b : (intructions) endfor ;
```

Dans le cas où le pas d'incrémentation est 1 ou  $-1$ , le début de l'instruction

---

<sup>(3)</sup> Le lecteur intéressé est invité à consulter l'index du METAFONTBook pour retrouver tous les passages concernant ces types de macros.



peut être abrégée en :

`for i=a upto b` : ou `for i=a downto b` :

- si l'index est un nombre dont les valeurs ne peuvent pas être engendrées par une incrémentation d'un pas donné, on écrit le début de l'instruction :

`for i=1.05,1.58,3.07 ... 28.23` :

c'est-à-dire l'on donne la liste des valeurs de l'indice séparées par des virgules ;

- si l'index est un est un suffixe, le début de l'instruction s'écrit alors :

`forsuffixes i=sufa,sufb,sufc ... sufn` :

- pour une boucle qui se répète sans fin, la syntaxe est :

`forever` : (instructions) `endfor` ;

Pour les sorties conditionnelles de boucle, on a les instructions [93] :

`exitif` (condition) ; ou `exitunless` (condition) ;

Dans le premier cas, la boucle s'arrête lorsque la condition devient vérifiée (autrement dit lorsque la variable booléenne exprimant la condition prend la valeur `true`) ; dans le deuxième cas, la boucle s'arrête lorsque la condition cesse d'être vérifiée (donc lorsque la booléenne prend la valeur `false`) ; il est sage de toujours s'assurer que toute boucle `forever` contient une sortie conditionnelle convenablement écrite ; on peut tester le fonctionnement en écrivant :

`a:=1; forever: a:=a+1; endfor`; (arrêter avec Ctrl C),

puis tester la commande d'arrêt de la boucle en écrivant :

`a:=1; forever: a:=a+1; exitif a>20; endfor`;

#### 4.5.2 Tests

La syntaxe de l'instruction de test va rappeler quelque chose aux utilisateurs de  $\text{\TeX}$  [80], [M 169] :

- `if` (booléenne) : (instructions A) `else:` (instructions B) `fi`

où les instructions A (resp. B) sont exécutées si la booléenne a la valeur `true` (resp. `false`). On a donné des exemples à la section 4.2.7.

Les tests du type :

`if (a) : (A) else: if (b) : (B) else: (C) fi fi`

peuvent être abrégés en :

`if (a) : (A) elseif (b) : (B) else: (C) fi`



# Boîtes et liaisons : organigrammes et algorithmes

Dans ce chapitre, on va construire des boîtes de différentes présentations (formes, contours, fonds, et positionnement). Toujours pour simplifier, on ne va considérer que des boîtes dont le contenu est un texte de quelques lettres composé avec la fonte `phvr8r` utilisée pour les exemples de labels (cf. sect. 1.3) ; cela permet encore de voir le résultat des exercices tout de suite à l'écran. Au lecteur d'imaginer que le contenu de la boîte pourrait être un `\parbox` composé à l'aide de  $\text{\LaTeX}$  ou une figure déjà produite d'une toute autre manière ! Pour disposer des commandes nécessaires, il faut charger le fichier `rboxes.mp` qui appelle le fichier `boxes.mp` et ajoute une commande pour construire des boîtes rectangulaires avec les angles arrondis. On reprend le préambule des quatre premiers fichiers de figures auquel on ajoute les commandes

```
input rboxes.mp
input boitesup.mp
linejoin:=mitered; ahlength:=2pt; (flèches pointues, cf. sect. 2.2.4)
rbox_radius:=3pt; bboxmargin:=0.5pt; (pour les boîtes, sections suivantes)
pour servir de préambule au nouveau fichier de figures fige.mp. Quant au
fichier boitesup.mp, il permet de faire des boîtes en forme d'ellipse et de lo-
sange; il contient aussi quelques macros élémentaires permettant le tracé de
boîtes « à la chaîne », macros aisément modifiables et adaptables par le lecteur
de ce document ; enfin, il propose quelques autres macros destinées à faire des
liaisons entre les boîtes. Bien entendu, ce petit nombre de macros élémentaires
écrites par l'auteur de ce petit document ne peuvent rivaliser avec celles des
nombreuses extensions spécialisées telles que metaobj par exemple.
```

## 5.1 Boîtes rectangulaires

On va exposer en détail la construction des boîtes de forme rectangulaire car tout ce qui va être explicité est encore valable pour les boîtes de forme différente : les principes de base restent les mêmes, certaines macros ont des noms différents pour les différentes formes mais il y en a d'autres qui gardent le même nom. La construction d'une telle boîte se fait en réalité en deux étapes qui vont être détaillées.

### 5.1.1 Création de la boîte

La *création* d'une boîte rectangulaire se fait à l'aide de la commande [95] `boxit.bb(mat)` ; où `bb` désigne le nom de la boîte, `mat` désigne le contenu de la boîte (ce qui est nommé « matériel » dans le langage du  $\text{T}_{\text{E}}\text{X}$ Book) : un ou plusieurs mots, une boîte  $\text{T}_{\text{E}}\text{X}$  ou  $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$  ou encore une figure déjà faite. Cette création comprend :

- les déclarations des variables du type paire :
  - `bb.c` : position du centre de la boîte,
  - `bb.n` : position du milieu de l'arrête supérieure, puis les positions des milieux des trois autres arrêtes avec les suffixes respectifs `e`, `s` et `w`,
  - `bb.ne` : position de l'angle supérieur droit de la boîte, puis les positions des trois autres angles avec les suffixes respectifs `se`, `sw` et `nw`,
  - `bb.off` : déplacement de la boîte par rapport à sa position d'origine ;
- les déclarations des variables du type nombre :
  - `bb.dx` : distance entre le matériel et les arrêtes gauche et droite,
  - `bb.dy` : distance entre le matériel et les arrêtes supérieure et inférieure.

### 5.1.2 Tracé de la boîte

Le *tracé* est assuré par la commande [95] `drawboxed(bb)` ; D'abord elle lance la résolution des équations reliant les variables déclarées par la commande `boxit.bb` (voir ci-dessus) en prenant par défaut [95] : `bb.off=(0,0)` ; `bb.dx=defaultdx` ; `bb.dy=defaultdy` ; Ensuite, cette commande trace la boîte en tenant compte des valeurs calculées des variables. Elle peut tracer plusieurs boîtes dont la liste des noms séparés par des virgules est donnée en argument.

On peut imposer des valeurs à toutes ces variables mais pas à toutes à la fois : il faut que le système d'équations linéaires reliant ces variables soit résoluble : on ne peut pas, par exemple, donner à la fois n'importe quelle valeur à `bb.off`, `bb.n` et `bb.dy` car on a :

$$\text{ypart } bb.off + (\text{hauteur du contenu}) + bb.dy = \text{ypart } bb.n;$$

comme on peut le vérifier sur la figure suivante. Si l'on fait une telle erreur, alors  $\text{METAPOST}$  affiche à la console le message d'erreur :

« *inconsistent equations* ».

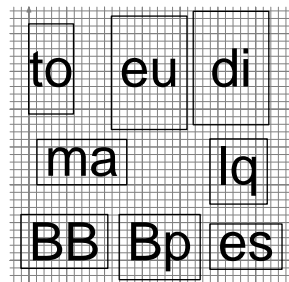
Par contre on peut faire tout ce qui est raisonnable au sens de la résolution des systèmes linéaires : on va voir quelques unes des possibilités sur la figure 1.

- **(BB)** est une boîte tracée avec les valeurs par défaut données ci-dessus : la boîte de matériel est placée avec l'angle bas gauche du **B** situé à l'origine. Le montant du **B** ne touche pas l'axe des  $y$ , il en est séparé par une distance appelée *approche gauche* qui est une caractéristique spécifique de chaque lettre ; on voit que le **e** de la boîte (**es**) a une approche gauche beaucoup plus petite (presque nulle, ce qui est logique pour une minuscule galbée à gauche) ; on voit également que le **e** descend légèrement en dessous de la ligne de base, ce qui constitue une autre caractéristique des lettres galbées

à la partie inférieure; ces remarques étant faites, on ne reviendra pas sur les propriétés des caractères).

- (Bp) est déplacée avec `bbb.off=(13u,0)`; (la vérification est aisée grâce à la grille millimétrique); il est également important de constater que la boîte de matériel était initialement placée à l'origine avec la ligne de base portée l'axe ce  $x$  : le déplacement suivant les  $y$  étant nul, la ligne de base est toujours portée pas l'axe des  $x$ .
- (es) est déplacée avec `bbc.off=(25u,0)`;
- (ma) est déplacée en écrivant `bbd.c=(7u,13u)`; afin qu'elle soit centrée au point  $(7u,13u)$ .
- (lq) est déplacée avec `ypart bbd.off=10u`; qui élève de  $10u$  la ligne de base et `bbd.w=bbc.w`; qui aligne verticalement par les côtés gauches cette boîte avec la boîte (es).
- (to) est une boîte où l'on a fortement réduit la distance séparant le matériel des arrêtes droite et gauche et fortement augmenté la distance entre le matériel et les arrêtes supérieure et inférieure; elle est ensuite déplacée vers le haut pour ne pas recouvrir les autres.
- (eu) et (di) sont des boîtes dont on a imposé la hauteur et la largeur en écrivant des égalités du type `ypart bb.w-ypart bb.e=15u`; Elles ont même ligne de base mais sont décalées l'une par rapport à l'autre car, au sens de  $\text{\TeX}$ , elles ont une même profondeur (nulle) mais une hauteur différente; ce problème est automatiquement résolu lorsque les boîtes initiales sont des boîtes  $\text{\TeX-L\TeX}$  où l'on rajoute une commande `\strut` au début et une autre à la fin du texte.

```
\beginfig(1);axespapiermilli(-2.5,-3.5,35.5,33.5);
boxit.bba("BB");drawboxed(bba);
boxit.bbb("Bp");bbb.off=(13u,0);drawboxed(bbb);
boxit.bbc("es");bbc.off=(25u,0);drawboxed(bbc);
boxit.bbd("ma");bbd.c=(7u,13u);drawboxed(bbd);
boxit.bbe("lq");ypart bbe.off=10u;
xpart bbe.w=xpart bbc.w;drawboxed(bbe);
boxit.bbf("to");bbf.dx=0.1pt;bbf.dy=10pt;
bbf.off=(0,23u);drawboxed(bbf);
boxit.bbg("eu");
ypart bbg.n-ypart bbg.s=15mm;
xpart bbg.e-xpart bbg.w=10mm;
bbg.off=(12u,23u);drawboxed(bbg);
boxit.bbh("dp");
ypart bbh.n-ypart bbh.s=15mm;
xpart bbh.e-xpart bbh.w=10mm;
bbh.off=(24u,23u);drawboxed(bbh);
endfig;
```

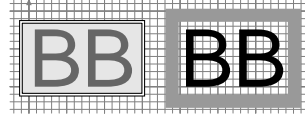


### 5.1.3 Améliorations : fond et contour

Une fois la macro `boxit.bb` utilisée on dispose de plusieurs macros qui permettent de diversifier la présentation de ces boîtes : `bpath.bb` et `pic.bb` permettent de tracer séparément le fond, le contour et le contenu avec des options différentes. Sur la figure suivante on a : un fond gris léger, un contour

en trait double et le contenu en gris foncé pour la boîte (BB) et pas de fond, un contour gris très épais et le contenu en noir pour la boîte (eu).

```
beginfig(2);
axespapiermilli(-2.5,-2.5,35.5,10.5);
defaultscale:=3;boxit.bbi("BB");
fill bpath.bbi c1(0.8);draw bpath.bbi lw(1.2pt);
undraw bpath.bbi lw(0.6);
draw pic.bbi c1(0.3);
boxit.bbj("BB");bbj.off=(20u,1u);
bbj.dx=defaultdx+2.5pt;bbj.dy=defaulddy+2.5pt;
draw bpath.bbj lw(5pt) c1(0.6);drawunboxed(bbj);
endfig;
```



Ces macros disponibles sont les suivantes ;  
**bpath.bb** [95] calcule les valeurs des variables comme le fait **drawboxed** puis devient le nom du contour entourant la boîte,  
**pic.bb** [97] calcule les valeurs des variables comme le fait **drawboxed** si ce n'est déjà fait puis devient le nom du contenu de la boîte,  
**drawunboxed(bb)** [97] calcule les valeurs des variables comme le fait **drawboxed** si ce n'est déjà fait puis trace le contenu de la boîte en noir (couleur par défaut).

La figure appelle deux remarques importantes.

- La frontière extérieure du contour de la deuxième boîte a les angles vifs : c'est le résultat de la valeur **mitered** donnée à la variable interne **linejoin** dans le préambule du fichier **fige.mp** afin d'obtenir des flèches pointues. Avec la valeur par défaut **rounded**, la frontière extérieure du contour aurait des angles arrondis (sect. 2.1 et 2.2.2). Cet effet serait d'autant plus visible que l'épaisseur du trait du contour est forte ; il faut bien noter que cet effet n'a rien à voir avec l'arrondissement des angles du contour des boîtes que l'on va voir à la prochaine section (macro **rboxit**) ; dans ce cas, ce sera la ligne centrale du contour qui aura les angles arrondis.
- L'utilisation de fortes épaisseurs de trait du contour exige une augmentation des variables **bb.dx** et **bb.dy** pour garder toujours la même distance entre le matériel contenu dans la boîte et le contour de la boîte (augmentation d'une demi épaisseur exactement).

## 5.2 Autres formes : arrondies, circulaires, etc.

On dispose de quatre autres formes de boîte prêtes à l'emploi ; les quatre commandes **drawboxed(bb)**, **bpath.bb**, **pic.bb** et **drawunboxed(bb)** ont toujours le même rôle (**bb** étant toujours le nom de la boîte). Les quatre autres commandes de création de boîte sont les suivantes.

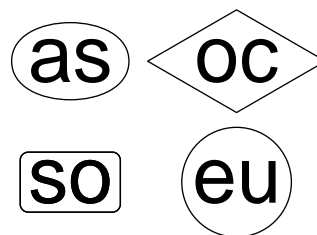
**rboxit.bb** crée une boîte avec les angles arrondis (boîte(**so**)) par un quart de cercle de rayon **rbox\_radius** (cette variable interne est réduite dans le préambule du fichier **fige.mp** de 8 bp à 3 pt pour des raisons d'ordre esthétique). Les huit paires **bb.n**, **bb.s**, etc. représentent toujours les coordonnées des mêmes points (milieu de l'arrête supérieure, milieu de l'arrête inférieure, etc.

`circleit.bb`, abrégée en `circit.bb` [99] crée un boîte de forme circulaire; `bb.dx` prend la valeur par défaut mais, dans ce cas, `bb.dy` prend une valeur définie pour que le contour soit un cercle (boîte `(eu)`); on peut cependant fixer `bb.dy` comme l'on veut et l'on obtient alors une ellipse; c'est en introduisant l'équation `bb.dx=bb.dy`; dans cette macro (et en y faisant quelques aménagements supplémentaires) qu'a été créée la macro suivante donnant un contour elliptique; pour cette forme et pour les autres deux formes possibles, d'une part, les paires correspondant aux directions intermédiaires `bb.ne`, `bb.se`, etc. ne sont plus définies et d'autre part, la paire `bb.n` représente les coordonnées du point le plus haut, la paire `bb.s` représente les coordonnées du point le plus bas, etc.

`ellit(bb)` crée un boîte elliptique (boîte `(as)`); il faut, pour cette forme de boîte et pour la suivante charger le fichier complémentaire `boitesup.mp`.

`diamit(bb)` crée une boîte en forme de losange (boîte `(oc)`); cette macro a été créée à partir de la macro `boxit`.

```
beginfig(3);
rboxit.bbk("so");bbk.c=(6u,2u);
drawboxed(bbk);
circleit.bbl("eu");bbl.c=(28u,2u);
drawboxed(bbl);
ellit.bbm("as");bbm.c=(6u,18u);
drawboxed(bbm);
diamit.bbn("oc");bbn.c=(28u,18u);
drawboxed(bbn);
endfig;
```



### 5.3 Macros élémentaires à la demande

On commence par faire le point sur les étapes nécessaires pour la construction des boîtes.

- On dispose de cinq formes prédéfinies pour lesquelles les macros de création correspondantes ont toujours la même syntaxe : `boxit.bb`, `rboxit.bb`, `circit.bb` (équivalent de `circleit.bb`), `ellit.bb` et `diamit.bb` qui créent les boîtes en prenant le contenu pour argument.
- Après la création, on donne diverses spécifications :
  - positionnement; ex : `bb.c=(3u,25u)`;
  - dimensionnement; ex : `ypart(bb.n-bb.s)=16u`;
  - marges internes; ex : `bb.dx=6pt`;
- Ensuite, on dispose de la macro `bpath.bb` qui est le nom du contour de la boîte; cela permet d'obtenir :
  - le tracé d'un fond coloré; ex : `fill bpath.bb c1(0.95)`;
  - le tracé du contour; ex : `draw bpath.bb lw(0.4pt) c1(0.4)`;
  - la rotation du contour autour de son centre; pour cela il faut introduire, dans les *deux* commandes ci-dessus et *immédiatement* après `bpath`, l'instruction `rotatedaround(bb.c,90)`; pour obtenir par exemple la position « lecture de bas en haut ».

- Enfin, avec la macro `pic.bb` qui désigne le contenu de la boîte, on peut obtenir :
  - le tracé du contenu en couleur ; ex : `draw pic.bb cl(0.4)` ;
  - la rotation du contenu autour de son centre ; après avoir tourné contour et fond, on place encore la même instruction après `pic.bb`.

Cela paraît un peu fastidieux à utiliser : il n'en est rien car, dans un document, on ne va pas faire des boîtes de toutes les manières possibles pour montrer son savoir, on va choisir un petit nombre de types de boîte bien adaptés aux types de contenu pour faciliter le travail d'assimilation du lecteur.

Pour cela, le fichier `boitesup.mp` contient une macro de base qui permet à l'utilisateur de construire très facilement des macros donnant des boîtes de différents styles ; sept exemples de ces macros sont fournies et seront très certainement suffisantes dans bien des cas. Bien évidemment, ces macros ne prétendent pas rivaliser avec celles des extensions spécialisées.

Cette macro de base est ;

```
btrace(pos,ang)(nom)(fond)(contour)(contenu) ;
```

`pos` est la paire correspondant au centre de la boîte,

`ang` est l'angle de rotation de la boîte autour du centre,

`nom` est le nom de la boîte,

`fond` est une liste de caractères caractérisant le fond de la boîte,

`contour` est une liste de caractères caractérisant le contour,

`contenu` est une liste de caractères caractérisant le contenu.

Par exemple, en composant des algorithmes, on veut représenter les données par des boîtes de forme elliptique avec un fond gris léger, un contour en trait double et le texte en noir ; pour cela, on définit la macro :

```
vardef donnees@#(pos,ang) =
```

```
  btrace(pos,ang)(@#)(ffgl)(ccd)(ddn) enddef ;
```

où `@#` est le nom de la boîte, `pos` et `ang` sont définis ci-dessus et où les autres trois arguments sont déterminés après avoir consulté le tableau suivant que tout débutant peut compléter tant les macros définissant ces spécifications sont élémentaires (voir le fichier `boitesup.mp`).

<code>ffo</code>	pas de fond	<code>cceg</code>	contour épais gris
<code>ffgl</code>	fond gris léger	<code>ccd</code>	contour trait double
<code>ffgc</code>	fond gris clair	<code>ddo</code>	pas de contenu
<code>ffgf</code>	fond gris foncé	<code>ddn</code>	contenu noir
<code>cco</code>	pas de contour	<code>cdgc</code>	contenu gris clair
<code>ccfn</code>	contour fin noir	<code>ddg</code>	contenu gris
<code>ccen</code>	contour épais noir	<code>ddgf</code>	contenu gris foncé

Ensuite, on écrit pour chaque boîte de ce type :

```
ellit.bb(Donnees) ;
```

```
donnees.bb((12u,42.5u),0) ;
```

Voici les deux macros passe-partout caractérisées par : pas de fond, un contour noir fin pour la première (d'où le `f` final) et épais pour la deuxième (d'où le `e` final) et le contenu en noir :

```
vardef tracebf@#(pos,ang) =
```

```
  btrace(pos,ang)(@#)(ffo)(ccfn)(ddn) enddef ;
```



```
vardef tracebe@#(pos,ang) =
  btrace(pos,ang)(@#)(ffo)(ccen)(ddn) enddef;
```

On trouvera sur le listing de la figure suivante quatre autres macros définies dans le fichier `boitesup.mp`; pour faciliter la lecture, le nom des macros de ce type est, excepté pour les deux macros passe-partout, constitué par la chaîne constituant le contenu de la boîte.

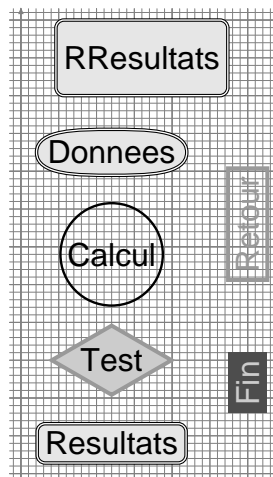
Les spécifications supplémentaires de dimensions (marges intérieures et dimensions imposées des boîtes) doivent être données, dans le cas de l'exemple ci-dessus, entre la macro `ellit.bb` et la macro `donnees.bb` et ce pour chaque boîte. C'est pour faciliter cette situation qu'une deuxième version de la macro de base a été créée (voir encore le fichier `boitesup.mp`) :

```
bbtrace(pos,ang)(nom)(spec)(fond)(contour)(contenu);
```

Elle fonctionne de la même manière avec en plus un argument complémentaire `spec` constitué par des équations imposant des valeurs aux marges intérieures ou aux dimensions des boîtes (sect. 5.1.2). On donne un exemple de l'utilisation de `bbtrace` pour créer une macro donnant des boîtes de hauteur donnée :

```
vardef rresultats@#(pos,ang) =
  bbtrace(pos,ang)(@#)(ypart(@#.N-@#.s)=9u)(ffgl)(ccd)(ddn)enddef;
```

```
beginfig(4);
defaultscale:=1.2;
axespapiermilli(-1.5,-1.5,34.5,47.5);
ellit.ba("Donnees");
donnees.ba((12u,42.5u),0);
cirit.bb("Calcul");
tracebe.bb((12u,29u),0);
diamit.bc("Test");
test.bc((12u,15u),0);
rboxit.bd("Resultats");
resultats.bd((12u,4u),0);
boxit.be("Retour");
retour.be((30u,33u),90);
boxit.bf("Fin");
fin.bf((30u,12u),90);
rboxit.bg("RResultats");
rresultats.bg((12u,4u),0);
endfig;
```



## 5.4 Liaisons entre les boîtes

### 5.4.1 Extrémités des liaisons

Il y a deux possibilités pour les extrémités des liaisons ; on peut choisir :

- le centre des boîtes,
- un des quatre points principaux définis sur le contour des boîtes : ce sont ces points qui vont être utilisés par la suite pour les raisons qui sont mises en évidence sur l'exemple suivant.

Une liaison (ou un lien ou encore une flèche) partant du centre de la boîte `bh` et allant au centre de la boîte `bi` s'obtient avec :

```
drawarrow bh.c -- bi.c
cutbefore bpath.bh cutafter bpath.bi cl(1pt);
```

Cette méthode (flèche 1) a un premier inconvénient : les opérateurs `cutbefore` et `cutafter` suppriment bien les parties de la liaison internes aux boîtes mais on ne peut empêcher la flèche de traverser le contour de la boîte cible. La commande :

```
drawarrow bi.e -- bj.w cl(1pt);
```

a le même défaut (flèche 2) mais il est facile d'apporter d'apporter une correction (flèche 3) en écrivant :

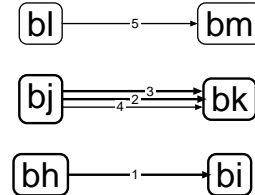
```
drawarrow bk.e -- bl.w+1.5*left cl(1pt);
```

On a ici utilisé une des très utiles définitions :

```
up=(0,1), right=(1,0), down=(0,-1) et left=(-1,0)
```

L'exemple montre bien que la correction de positionnement des flèches est satisfaisante : coefficient de 1.5 pour les boîtes avec contour épais (1 pt) avec liaisons de même épaisseur (flèche 3), coefficient 0.75 pour les boîtes à contour fin (0.6 pt) avec liaisons de même épaisseur (flèche 5) et coefficient 1 pour les boîtes à contour épais avec liaisons fines (flèche 4). En outre, on constate que la méthode se prête fort bien pour traiter les cas où plusieurs liaisons partent de (ou arrivent sur) la même boîte (flèches 3, 4 et 5). Pour faciliter la lecture de la fin de ce chapitre, le nom des boîtes est rappelé dans leur contenu.

```
beginfig(5);
defaultscale:=1.3;
rboxit.bh("bh");tracebe.bh((5u,0),0);
rboxit.bi("bi");tracebe.bi((30u,0),0);
rboxit.bj("bj");tracebe.bj((5u,10u),0);
rboxit.bk("bk");tracebe.bk((30u,10u),0);
rboxit.bl("bl");tracebf.bl((5u,20u),0);
rboxit.bm("bm");tracebf.bm((30u,20u),0);
drawarrow bh.c--bi.c
cutbefore bpath.bh cutafter bpath.bi lw(1pt);
drawarrow bj.e--bk.w lw(1pt);
drawarrow bj.e+3*up--bk.w+3*up+1.5*left lw(1pt);
drawarrow bj.e-3*up--bk.w-3*up+1*left lw(0.6pt);
drawarrow bl.e--bm.w+0.7*left lw(0.6pt);
endfig;
```



#### 5.4.2 Liaisons à un seul segment

La simplicité des commandes ci-dessus permet de définir la macro suivante très simple, facile à utiliser et reportée dans le fichier `boitesup.mp` :

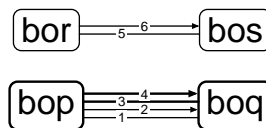
```
lien(dep,arr)(type);
```

où `dep` est la paire représentant le point de départ, `arr` est la paire représentant le point d'arrivée et `type` représente le type de lien voulu choisi sur le tableau à double entrée suivant :

épaisseur des contours et des liens	trait	flèche
contours et lien épais	<b>te</b>	<b>fe</b>
contours et liens fins	<b>tf</b>	<b>ff</b>
contours épais et liens fins	<b>tm</b>	<b>fm</b>

La figure suivante présente un exemple des six possibilités.

```
beginfig(6);
defaultscale:=1.3;
rboxit.bop("bop");tracebe.bop((5u,0),0);
rboxit.boq("boq");tracebe.boq((30u,0),0);
rboxit.bor("bor");tracebf.bor((5u,10u),0);
rboxit.bos("bos");tracebf.bos((30u,10u),0);
lien(bop.e-4.5*up,boq.w-4.5*up)(tm);
lien(bop.e-1.5*up,boq.w-1.5*up)(fm);
lien(bop.e+1.5*up,boq.w+1.5*up)(te);
lien(bop.e+4.5*up,boq.w+4.5*up)(fe);
lien(bor.e-1.5*up,bos.w-1.5*up)(tf);
lien(bor.e+1.5*up,bos.w+1.5*up)(ff);
endfig;
```



### 5.4.3 Liasons à deux segments

En utilisant le calcul de l'intersection de deux droites vu à la section 2.3.2, il est facile d'écrire par exemple la commande qui trace la flèche 3 de l'exemple suivant (sans correction de positionnement mais avec correction de flèche) :

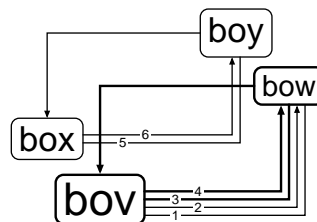
- la première droite est définie par les paires  $za=bov.e$  et  $zb=bov.e+polar(1,0)$ ,
- la deuxième droite est définie par les paires  $zc=bow.s$  et  $zd=bow.s+polar(1,90)$ ,
- le point d'intersection est défini par  $zint=whatever[za,zb]=whatever[zc,zd]$ ,
- le tracé de la flèche se fait alors avec  $drawarrow za -- zc+1.5*down lw(1pt)$

Il est encore facile d'écrire une macro relativement simple, facile à utiliser et également reportée dans le fichier `boitesup.mp` :

```
llien(dep,angdep,arr,angarr)(type) :
```

où `angdep` et `angarr` sont respectivement les directions orientées au départ et à l'arrivée ; les autres notations sont définies ci-dessus. La figure suivante présente un exemple des six possibilités.

```
beginfig(7);
defaultscale:=1.7;rboxit.bov("bov");tracebe.bov((5u,0),0);
defaultscale:=1.2;rboxit.bow("bow");tracebe.bow((30u,15u),0);
defaultscale:=1.3;
rboxit.box("box");tracebf.box((-2u,8u),0);
rboxit.boy("boy");tracebf.boy((23u,22u),0);
llien(bov.e-6*up,0,bow.s+6*right,90)(tm);
llien(bov.e-3*up,0,bow.s+3*right,90)(fm);
llien(bov.e,0,bow.s,90)(te);
llien(bov.e+3*up,0,bow.s-3*right,90)(fe);
llien(bov.w,0,bov.n,-90)(fe);
llien(box.e-1.5*up,0,boy.s-1.5*left,90)(tf);
llien(box.e+1.5*up,0,boy.s+1.5*left,90)(ff);
llien(boy.w,-180,box.n,-90)(ff);
endfig;
```



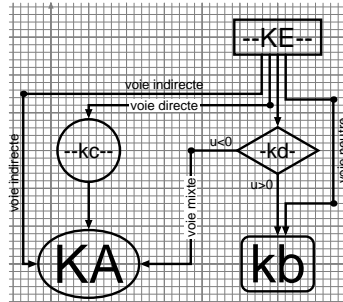
#### 5.4.4 Liaisons à trois segments et plus

Au delà de deux segments, on est obligé de fournir des données supplémentaires ; certaines extensions demandent une ou deux longueurs de bras ; on choisit la méthode la plus simple : on va se donner un point pour une liaison à trois ou quatre segments et deux points pour une liaison à cinq segments ; la liaison complète est ensuite obtenue à l'aide de deux ou trois éléments de base proposés dans les deux sections précédentes. La figure suivante montre un exemple de liaisons à trois, quatre et cinq segments ; les points dont les coordonnées ont été données sont repérés par un petit cercle noir et signalés sur le listing par P1, P2, etc. ; ils permettent d'identifier facilement les deux ou trois éléments de base constituant la liaison complète ; la grille millimétrique aide à bien reconnaître l'effet des lignes du listing.

```

beginfig(8);
axespapiermilli(-5.5,-5.5,39.5,34.5);
%%%% Fabrication des boites
defaultscale:=2;ellit.ka("KA");tracebe.ka((5u,0),0);
rboxit.kb("kb");tracebe.kb((30u,0),0);
defaultscale:=0.8;cirit.kc("--kc--");tracebe.kc((5u,15u),0);
diamit.kd("-kd-");tracebe.kd((30u,15u),0);
defaultscale:=1.2;boxit.ke("--KE--");tracebe.ke((30u,30u),0);
%%%% Pose des liens, --P1 : lien vers P1, P1-- : lien partant de P1...
lien(ke.s,kd.n)(fe);
lien(ke.s+3*left,(xpart(ke.s+3*left),21u))(te); % --P1
llien((xpart(ke.s+3*left),21u),-180,kc.n,-90)(fe); % P1--
llien(ke.s+6*left,-90,(xpart(ka.w-2u,22.5u),-180)(te); % --P2
llien((xpart(ka.w-2u,22.5u),-90,ka.w,0)(fe); % P2--
lien(kd.s,kb.n)(fe);
llien(ke.s+3*right,-90,
(xpart(kd.e+2u,21.75u),0)(te); % --P3
lien((xpart(kd.e+2u,21.75u),
(xpart(kd.e+2u,8u))(te); % P3P4
llien((xpart(kd.e+2u,8u),
-180,kb.n+3*right,-90)(fe); % P4--
lien(kc.s,ka.n)(fe);
lien(kd.w,(18.5u,ypart(kd.w))(te); % --P5
llien((18.5u,ypart(kd.w),-90,ka.e,-180)(fe);%P5-
drawpt((xpart(ke.s+3*left),21u))(2.5pt); % P1
drawpt((xpart(ka.w-2u,22.5u))(2.5pt); % P2
drawpt((xpart(kd.e+2u,21.75u))(2.5pt); % P3
drawpt((xpart(kd.e+2u,8u))(2.5pt); % P4
drawpt((18.5u,ypart(kd.w))(2.5pt); % P5
%%%% Pose des labels : voir section suivante
defaultscale:=0.5;picture ffa,ffb,ffc,ffd,ffe;
label.top("voie indirecte",(15u,22.5u));
ffe=thelabel("voie directe",(15u,21u));
unfill bbox ffe;draw ffe;
ffa=thelabel.top("voie indirecte",(xpart(ka.w-2u,12u));
draw ffa rotatedabout((xpart(ka.w-2u,12u),90);
label.top("u<0",(22.5u,15u));label.left("u>0",(30u,10u));
ffc=thelabel.bot("voie neutre",(xpart(kd.e+2u,15u));
draw ffc rotatedaround((xpart(kd.e+2u,15u),90);
ffd=thelabel("voie mixte",(18.5u,7.5u));
unfill bbox ffd rotatedaround((18.5u,7.5u),90);
draw ffd rotatedaround((18.5u,7.5u),90);
endfig;

```



Pour les boîtes en forme de cercle, d'ellipse ou de losange, on remarque que l'on ne peut, avec les deux simples macros (`lien` et `llien`), que faire partir (ou arriver) des liaisons aux quatre points principaux `bb.n`, `bb.s`, etc. (sinon il faudrait faire des liaisons partant du (ou arrivant au) centre et utiliser les opérateurs `cutbefore` et `cutafter`, ce qui est en dehors de ce petit manuel simplement destiné à débiter en METAPOST).

## 5.5 Mettre des labels (ou lettrages) sur les liaisons

On dispose pour cela des macros suivantes :

- `label.xx("aaa",zz)` ;  
qui compose la chaîne "aaa" au point de référence `zz` et la déplace d'une distance `labeloffset` dans la direction `xx` [44] (sect. 1.3) : il faut être attentif à ne pas confondre les quatre directions principales que peut prendre `xx` : `top`, `rt`, `bot` et `lft`,  
et les quatre vecteurs unitaires :  
`up=(0,1)`, `right=(1,0)`, `down=(0,-1)` et `left=(-1,0)`.

- `thelabel.xx("aaa",zz)` : qui crée le dessin contenant la composition de la chaîne "aaa" comme ci-dessus mais ne l'imprime pas [45] : en d'autres termes, si `des` est une variable dessin déclarée, `des=thelabelxx("aaa",zz);draw des;` équivaut à `label.xx("aaa",zz)` ;

On verra ci-après tout l'intérêt de cette macro qui, à première vue, semble inutile. On dispose enfin de la macro :

- `bbox des` qui donne le chemin rectangulaire entourant le dessin `des` avec une marge de largeur `bboxmargin` dans les quatre directions (c'est une *BoundingBox* agrandie) [50].

Avec ces deux dernières macros, on va pouvoir

- faire tourner le label d'un angle `ang` autour de son centre en écrivant `draw des rotatedaround(zz,ang);`
- faire un fond blanc pour poser le label et le rendre plus lisible en effaçant l'intérieur du chemin rectangulaire entourant le label : on écrit pour cela `unfill bbox des rotatedaround(zz,ang);`  
où la rotation n'est bien évidemment nécessaire que si le label est tourné lui-même.

Il faut noter que si, par exemple, on veut un label tourné de 90° et situé à gauche d'une liaison verticale, il faut choisir la valeur `top` pour `xx` !

La figure précédente comprend tous les cas possibles :

- ligne (1) : label horizontal en dessus de la liaison (cas le plus simple) ;
- ligne (2) : label horizontal sur la liaison elle-même ;
- lignes (3) et (4) : label vertical à gauche d'une liaison verticale ;
- ligne (6) : label horizontal à gauche d'une liaison verticale ;
- lignes (9), (10) et (11) : label vertical sur la liaison elle-même ;

On reviendra (chap. 7) sur ces macros dans le cas des lettrages composés avec T<sub>E</sub>X-L<sup>A</sup>T<sub>E</sub>X où l'on rencontrera une petite différence qui facilitera la tâche.



## Tracé de courbes

Dans le cadre d'un petit manuel d'initiation à METAPOST tel que le présent document il est raisonnable, qu'après un chapitre consacré aux boîtes et à leurs liaisons, il y ait un chapitre concernant le tracé de graphes ou le « tracé de courbes » comme disent les physiciens quand ils reportent sur un document la variation d'une grandeur physique en fonction d'un paramètre ; ce langage sera utilisé par la suite.

On va créer un nouveau fichier de figures, `figf.mp`, en prenant le préambule du fichier `fige.mp` (avec `labeloffset` agrandi d'un demi pt) et en y rajoutant :

```
input graph.mp
```

qui charge un fichier contenant un ensemble de macros destinées à la présente tâche (ce fichier en charge d'autres, notamment `sarith.mp` et `format.mp` qui vont jouer un rôle dans le traitement des grand nombres ;

```
input graphsup.mp
```

qui appelle un fichier où sont rassemblées quelques macros élémentaires simplifiant l'écriture ou permettant quelques possibilités supplémentaires sans prétention par rapport à des extensions spécialisées ; ce fichier commence en donnant la longueur 2 bp (au lieu de 7 bp) aux repères des graduations des axes, les *ticks* ; ce mot sera utilisé par la suite (cette modification a pour unique but d'avoir une présentation esthétiquement acceptable). On ajoute aussi à ce préambule [161] :

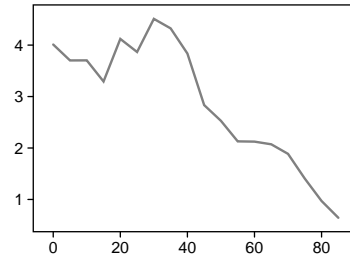
```
init_numbers(thelabel("-", (0,0)), thelabel("1", (0,0)),
             thelabel("e", (1pt, 1.5pt)), thelabel("-", (0,0)),
             thelabel("2", (1pt, 2pt)));
```

qui est une commande de formatage des valeurs affectées aux graduations pour lesquelles on gardera le mot *label* ; cette commande est nécessaire pour pouvoir voir directement à l'écran le résultat des exercices sans utiliser  $\text{\TeX-L\TeX}$  ; bien entendu, elle devra être reformulée différemment pour l'intégration des figures dans un document  $\text{\TeX-L\TeX}$  où l'on voudra alors avoir tous les lettrages composée avec  $\text{\LaTeX}$  (on reviendra donc sur cette commande au chapitre 7).

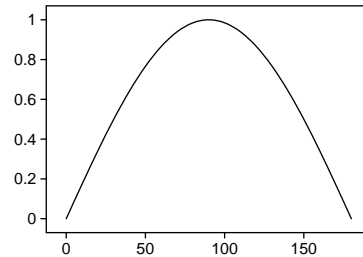
### 6.1 Tracés par défaut

Les macros puissantes contenues dans le fichier `graph.mp` permettent de tracer une courbe avec axes et graduations en trois lignes de code [141] :

```
beginfig(1);
draw begingraph(43u,30u);
gdraw "data1" lw(1pt) cl(0.75);
endgraph;
endfig;
```



```
beginfig(2);
path chemin;
draw begingraph(43u,30u);
chemin=(0,0) for x=0 step 10 until 180: %Ex 1
  .. (x,sind x) endfor; %Ex 1
%chemin=(0,0) for x=0 step 30 until 180: %Ex 2
% -- (x,sind x) endfor; %Ex 2
gdraw chemin ;
endgraph;
endfig;
```



Dans les codes des figures précédentes :

```
draw begingraph(larg,haut);
```

est la commande qui débute tous les tracés de courbes : **larg** et **haut** sont les dimensions voulues du tracé y compris les graduations (tiks et labels);

```
endgraph;
```

est la commande qui termine tous les tracés;

```
gdraw "data1" ou chemin (options de tracé);
```

est le commande qui :

- trace la courbe après avoir lu les données dans le fichier **data1** ou lu la définition du chemin **chemin**; ceci après avoir fait les mises à l'échelle nécessaires pour que la figure rentre dans le rectangle de dimensions données et après avoir pris en compte les éventuelles options de tracé;
- construit les axes appropriés avec les tiks et les labels (on note que c'est METAPOST qui choisit les emplacements des graduations).

Les options de tracé [142], situées immédiatement après la chaîne constituée par le nom du fichier (figure 1) ou par le nom du chemin (figure 2), peuvent être : **withpen ...**, **withcolor ...** et/ou **dashed ...** ou bien encore toutes les macros correspondantes **cl**, **lw**, etc.

La structure du fichier de données est la suivante : coordonnée  $x$ , un espace, coordonnée  $y$ , éventuellement un ou plusieurs espaces et fin de ligne; l'exemple montre comment y insérer des commentaires :

```
00 4.011 data1 : nombre d'habitants
05 3.702         divisé par 1000000
10 3.703         en fonction de l'âge
... ..
85 0.644
```

Bien entendu, le lecteur va vouloir une tout autre présentation du tracé : c'est l'objet des toutes les sections suivantes.

Dans ce chapitre, pour diminuer le nombre de figures, on utilisera la méthode

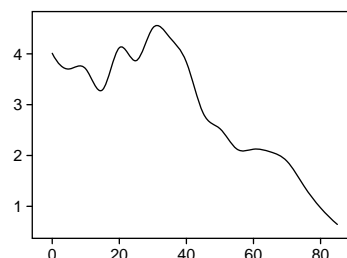


présentée sur la figure 2 ; des parties de code, (Ex 1), (Ex 2), etc., affichent des tracés identiques ou différents suivant les cas. La partie qui n'est pas « cachée » par des % est celle correspondant à la figure affichée ; le lecteur peut ensuite traiter successivement les autres parties par une simple manipulation de %. Dans la figure qui va suivre, on va introduire encore une autre convention : des parties de code, (Tt 1), (Tt 2), etc., peuvent afficher un tracé supplémentaire permettant de faire des comparaisons ; pour cela, le lecteur doit simplement enlever les % « cachant » la (les) partie(s) souhaitée(s).

## 6.2 Types de tracés disponibles

Dans la grande majorité des cas, on dispose d'un fichier de données. Le tracé le plus élémentaire est le tracé en « zig-zag » de la figure 1. Une très minime modification d'une macro du fichier `graph.mp` permet, moyennant l'utilisation d'une variable interne supplémentaire, d'obtenir un tracé « lissé » de la courbe. La macro modifiée et la déclaration de la variable interne nommée `lisse` sont dans le fichier `graphsupsup.mf`. La partie de code (Tt 1) trace la version zig-zag de la courbe (figure 1), il permet d'avoir un aperçu de la « valeur » du lissage.

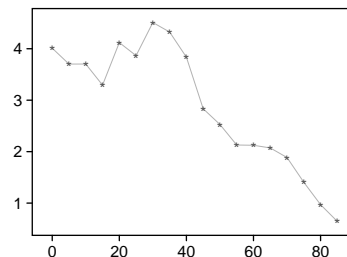
```
beginfig(3);
draw begingraph(43u,30u);
%gdraw "data1" lw(0.3pt);           %Tt 1
lisse:=1;gdraw "data1";lisse:=0;
endgraph;
endfig;
```



On remarque que le tracé produit à partir du deuxième (resp. premier) chemin de la figure 2 est du même type que le tracé produit avec le fichier de données sur la figure 1 (resp. avec l'option `lisse:=1` sur la figure 3) ; il suffit de consulter la macro modifiée `Gscan_` dans le fichier `graphsupsup.mp` pour s'en rendre compte. Par la suite, on ne parlera le plus souvent que de tracés à partir d'un fichier de données.

On peut aussi vouloir marquer les points guides de la courbe par un caractère, une astérisque par exemple, ou tout simplement ne tracer que ces points guides ; cela se fait avec l'option de tracé `plot` (spécifique à `gdraw`) [143] :

```
beginfig(4);
draw begingraph(43u,30u);
gdraw "data1" lw(0.3pt) cl(0.7);           %Tt 1
%lisse:=1; gdraw "data1" lw(0.3pt);       %Tt 2
% lisse:=0;                                %Tt 2
%gdraw "data1" plot thelabel("*", (0,0)); %Ex 1
gdata("data1",s,glabel("*",s1,s2)        %Ex 2
      cl(0.3);)                             %Ex 2
endgraph;
endfig;
```



Le rôle de `plot` est le suivant : la macro `thelabel` a été vue à la en section 5.5 ;  
`draw thelabel("*", (0,0))` ; affiche une astérisque à l'origine,  
`draw thelabel("*", (0,0)) shifted(a,b)` ; affiche une astérisque au point de coordonnées  $a$  et  $b$ .

Ici, dans la figure 4, partie (Ex 1) :

`plot thelabel("*", (0,0))` ; affiche une astérisque en tous les points dont les coordonnées sont données dans le fichier lu.

Toujours dans la même figure, la partie (Ex 2), on utilise la macro `gdata` qui donne encore le même résultat avec en plus la possibilité de modifier la couleur de l'astérisque, ce qui n'est pas possible avec la partie (Ex 1). On va expliquer en détail l'utilisation de cette macro dont la syntaxe générale est [152] :

```
gdata("nom-fichier",s,liste-commandes;)
```

avec les définitions suivantes :

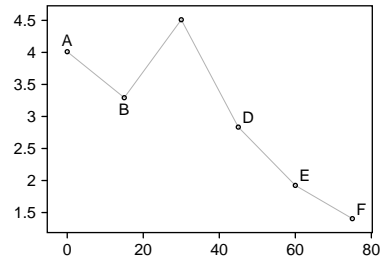
- `nom-fichier` est le nom d'un fichier de données comprenant  $N$  données par ligne (au lieu de 2) séparées par un espace ;
- `v` est une variable tableau dont les  $N$  composantes `v1`, `v2` ... prennent, à la lecture de la ligne  $i$  du fichier, les  $N$  valeurs qui s'y trouvent ;
- `liste-commandes` est une liste de commandes exécutées avec l'indice  $i$ , numéro de la ligne, et les valeurs de `v1`, `v2` ... prises à la lecture de la ligne.

On dispose en outre des macros `glabel`, `gdotlabel`, `gdraw` et `gfill` qui sont respectivement équivalentes aux macros `label`, `dotlabel`, `drawet fill` excepté que les valeurs données des positions subissent la mise à l'échelle nécessaire pour que la figure ait les dimensions voulues ; on a donc écrit :

```
gdata("data1",s,glabel("*",s1,s2) cl(0.5));
```

On va voir comment ces macros permettent de faire de nombreux types de tracés dont on ne va donner que quelques exemples. La partie (Ex 1) de la figure suivante utilise une bille noire définie préalablement pour montrer que l'on peut utiliser un marqueur fait-maison ; ce type de définition, très courant, utilise la commande `image` qui transforme une liste de commandes de tracé en une variable dessin (cette variable dessin peut être tracée par la suite avec toutes les options souhaitées).

```
%<def bille=image(draw fullcircle scaled 1.5)endef;
beginfig(5);
draw bevingraph(43u,30u);
string rien; rien="";
gdraw "data2" lw(0.3pt) cl(0.7);           %Tx 1
%lisse:=1; gdraw "data2" lw(0.3pt);       %Tt 2
%lisse:=0;                                 %Tt 2
gdata("data2",s,glabel(bille,s1,s2);)     %Ex 1
gdata("data2",s,if s3=rien: else:         %Ex 2
  forsuffices suff = top,bot,urt :        %Ex 2
  if s4=(str suff): glabel.suff(s3,s1,s2) %Ex 2
  else: fi; endfor fi;);                  %Ex 2
endgraph;
endfig;
```



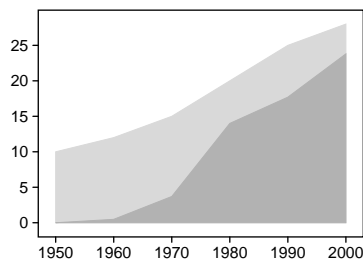
La partie (Ex 2) de cette même figure utilise une possibilité de la macro `gdata` : elle place un label à chaque point-guide dans la direction appropriée. Les trois premières lignes du fichier de données sont :

```
00 4.011 A top
15 3.293 B bot
30 4.51
```

Le test sur la chaîne `s3` permet de ne pas mettre de label à tous les points (bien entendu, il faut faire un pré-traitement pour déterminer les positions des labels et les reporter dans le fichier de données).

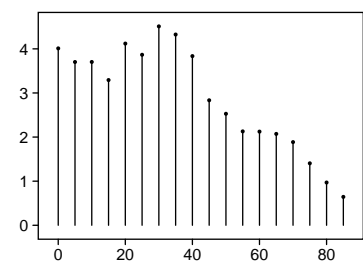
Les deux figures suivantes utilisent la macro `augment.p(z)` [154] où `p` est un chemin préalablement déclaré et où `z` est une paire représentant un point ; la première fois que `augment` est invoquée avec le chemin `p` et une paire `(a,b)`, elle définit `p` comme le chemin ponctuel `(a,b)`, la deuxième fois qu'elle est invoquée avec le chemin `p` et une autre paire `(c,d)`, elle définit `p` comme étant le chemin `(a,b)--(c,d)` ; à chaque lecture de ligne, la macro génère un nouveau segment de la courbe en style zig-zag. Il suffit alors de rajouter la projection de la courbe sur l'axe des  $x$  pour obtenir un chemin fermé que l'on colore pour avoir le résultat de la figure. Pour fixer les idées, cette figure représente en gris clair l'énergie consommée par un pays (en unités arbitraires) et en gris foncé la partie de cette énergie d'origine nucléaire<sup>(1)</sup>. Les colonnes du fichier de donnée sont les années, la consommation totale (`s3`) et la partie d'origine nucléaire (`s2`).

```
beginfig(6);
draw begingraph(43u,30u);
path p,pp;
gdata("data3",s,augment.p(s1,s3);)
  gfill p--(2000,0)--(1950,0)--cycle
  cl(0.85);
gdata("data3",s,augment.pp(s1,s2);)
  gfill pp--(2000,0)--(1950,0)--cycle
  cl(0.70);
endgraph;
endfig;
```



Dans la figure suivante, la macro `augment` est utilisée, à chaque lecture de ligne, pour contruire la « tige de l'épingle » correspondante (ce type de tracé est particulièrement apprécié dans certaines disciplines, peut-être avec une tête, faite avec la macro `bille`, un peu plus grosse).

```
beginfig(7);
draw begingraph(43u,30u);
faitlabille;
gdata("data1",s,glabel(bille,s1,s2);)
gdata("data1",s,path p;augment.p(s1,s2);)
  augment.p(s1,0);gdraw p;)
endgraph;
endfig;
```



Pour ne pas embrouiller le lecteur, on a gardé, et on gardera autant que possible, les caractéristiques par défaut défaut excepté celles que l'on est en train d'examiner.

<sup>(1)</sup> Le lecteur n'aura pas de difficulté pour deviner quel pays à suggéré les donnés de cette figure.

### 6.3 Domaines de variation et types de coordonnées

Par défaut, le cadre entourant le tracé est déterminé par les coordonnées de ses sommets inférieur gauche et supérieur droit :

$x_{\min} - a$ ,  $y_{\min} - b$ ,  $x_{\max} + a$  et  $y_{\max} + b$ ,

où les coordonnées  $x_{\min}$ ,  $y_{\min}$ ,  $x_{\max}$  et  $y_{\max}$  sont lues dans le fichier de données et où  $a$  et  $b$  ont des valeurs appropriées pour que ce cadre entoure le tracé à une certaine distance en donnant un résultat visuellement agréable. Si on le veut, on peut modifier cet aspect avec la macro [145] :

`setrange(ux,uy,vx,vy)` ;

où `ux`, `uy`, `vx` et `vy` sont les coordonnées voulues pour les deux sommets cités exprimées avec les mêmes unités que les coordonnées contenues dans le fichier de données (pour supprimer des points, il faut effacer les lignes correspondantes : le fichier ne doit pas avoir de ligne blanche ni de lignes cachées par des %). Si l'on ne veut modifier que certaines de ces quatre bornes, les autres doivent être affectées avec la chaîne `whatever` pour prendre leur valeur par défaut déterminée comme expliqué ci-dessus ; on donne un exemple assez couramment utilisé :

`setrange(origin,whatever,whatever)` ;

où `origin` est une abréviation de `0,0`.

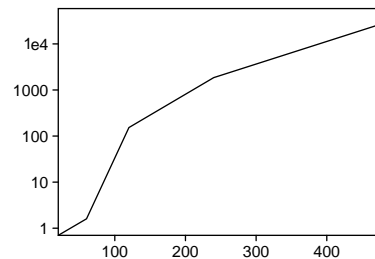
On est de nouveau confronté au fait que METAPOST n'utilise en interne que des nombres entiers bornés ; pour le tracé de courbe, les nombres doivent être inférieurs à 1000. Pour passer outre cette limitation, il faut écrire les coordonnées données dans la macro `setrange` sous la forme de chaînes de caractères du type "`m`" où `m` est un nombre « acceptable », `e` est le caractère « e » et `n` est la puissance de 10 par laquelle il faut multiplier le nombre précédent pour retrouver le nombre initial trop « grand » [146].

La figure suivante utilise successivement les fichiers `data4` et `data5` dont les premières et dernières lignes sont les suivantes :

```
data4 : 20 0.007      data5 : 20 0.7
        480 279.3      480 27930
```

La partie (Ex 1) utilise le fichier `data4` dont les nombres sont acceptables et la macro `setrange` prend les coordonnées des premier et dernier points. La partie (Ex 2) utilise le fichier `data5` où les ordonnées ont été multipliées par 100 ; mais, le fait d'avoir écrit pour dernier argument de `setrange` la plus grande ordonnée sous la forme spécifique détaillée ci-dessus, suffit pour que METAPOST produise le tracé correct qui, bien entendu, coïncide avec le précédent (les labels de l'axe des ordonnées ne sont évidemment pas les mêmes) :

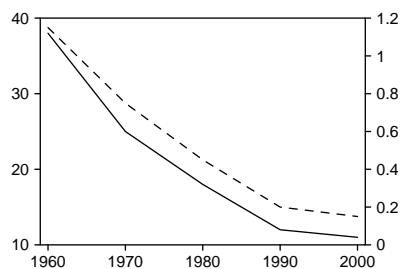
```
beginfig(8);
draw begingraph(43u,30u);
setcoords(linear,log);
%setrange(20,0.007,480,279.3); %Ex 1
%gdraw "data4"; %Ex 1
setrange(20,0.7,480,"2.793e4"); %Ex 2
gdraw "data5"; %Ex 2
endgraph;
endfig;
```



On arrive aux types de coordonnées que l'on choisit avec la macro [147] : `setcoords(typex, typey)` ; où `typex` et `typey` peuvent prendre les valeurs `linear`, `log` ainsi que `-linear` et `-log` ; le défaut correspond `linear` pour les abscisses et les ordonnées (comme on peut le constater, la figure 8 correspond au choix `linear` pour les  $x$  et `log` pour les  $y$ ). Par défaut, c'est le type `linear` qui est utilisé pour les abscisses et pour les ordonnées.

On va voir maintenant comment on peut tracer, sur une même figure, deux courbes différentes avec des unités différentes suivant les  $y$ . Sur la figure suivante, on trace, en fonction des années entre 1960 et 2000, l'évolution de la pollution en éléments irritants (exprimée en unités arbitraires, courbe continue) et le pourcentage de personnes ayant nécessité une courte hospitalisation (courbe en traitillé). Ce tracé utilise deux fichiers `data6` et `data7`, mais on pourrait mettre les données en un seul fichier où les données des deux courbes seraient séparées par un ligne blanche : en arrivant à une ligne blanche, la lecture s'arrête ; elle reprend lorsqu'une nouvelle lecture est demandée. Cette propriété fait que tout se bloque si le fichier de données se termine malencontreusement par une ligne blanche, car METAPOST attend une nouvelle commande de lecture du même fichier !

```
beginfig(9);
draw begingraph(43u,30u);
setrange("1.959e3",10,"2.001e3",40);
gdraw "data6";
autogrid(,otick.lft);
setcoords(linear,linear);
setrange("1.959e3",0,"2.002e3",1.21);
gdraw "data7" dashed evenly;
autogrid(otick.bot,otick.rt);
endgraph;
endfig;
```



Dans le code de cette figure, on remarque attentivement les positions des commandes `setcoords(...)` ; `setrange(...)` ; et `autogrid(...)` ; qui va être décrite à la sous-section suivante. Bien que `setcoords(linear,linear)` ; donne le choix de coordonnées par défaut, et donc soit en général inutile, sa présence est ici nécessaire pour signaler le changement d'ordonnées pour la deuxième courbe. On constate aussi que, pour faire apparaître les labels 1960, 2000 et 1.2, on a dû prendre les bornes 1.959e3, 2.001e3 et 1.21. On verra lus loin comment remédier systématiquement à ces problèmes (liés aux arrondis des calculs internes de METAPOST).

## 6.4 Cadre (ou axes) et graduations (ticks et labels)

Les ticks et les grilles sont choisis avec la macro [149] : `autogrid(xxx,yyy)`(options de tracé) ; où `xxx` pour les abscisses et `yyy` pour les ordonnées sont soit vides soit prennent les valeurs suivantes :

`otick.suff`

pour des ticks vers l'extérieur du cadre ; `suff` est un suffixe donnant le côté du cadre où l'on veut placer les ticks : `top`, `rt`, `bot` et `lft` respectivement pour côté haut, droit, bas et gauche ;

`itick.suff`

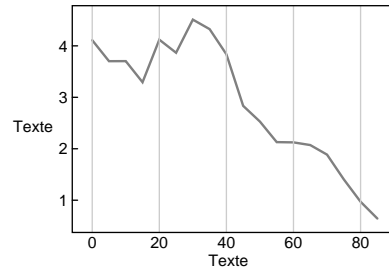
pour des ticks vers l'intérieur du cadre ; `suff` joue le même que ci-dessus ;

`grid.suff`

trace des traits parallèles à partir du côté désigné par le suffixe `suff` déjà défini ci-dessus ; il est recommandé d'ajouter une option de tracé (gris très clair ou couleur claire) pour laisser la ou les courbes bien en évidence ;

Enfin si un argument reste vide, alors il n'y a aucune graduation correspondante, c'est le cas de la partie (Ex 1) sur la figure suivante. La partie (Ex 2) montre comment en utilisant deux fois la macro `autofig`, on peut avoir des lignes de grille gris clair pour les abscisses et des ticks noirs pour les ordonnées.

```
beginfig(10);
draw bevingraph(43u,30u);
gdraw "data1" lw(1pt) cl(0.5);
%autogrid(itick.lft); %Ex 1
autogrid(grid.bot,) cl(0.8); %Ex 2
autogrid(itick.lft); %Ex 2
glabel.bot("Texte",OUT);
glabel.lft("Texte",OUT);
endgraph;
endfig;
```

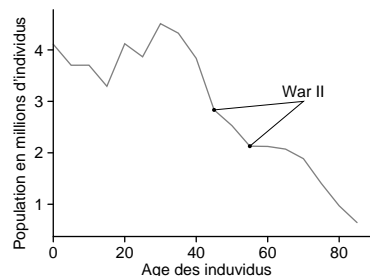


Pour modifier le cadre on dispose de la macro [151] :

`frame.suff`(options de tracé);

où `suff` prend les valeurs `top`, `rt`, `bot`, `lft` et leurs combinaisons sous une forme spécifique déjà rencontrée : `llft` donne les axes traditionnels, partie (Ex 1) sur la figure 11 : en ajoutant `frame.rt`, partie (Ex 2), on aura un nouvel axe d'ordonnées à droite que l'on peut graduer indépendamment des graduations de l'axe vertical de gauche, comme fait sur la figure 9. Evidemment, on a le cadre complet par défaut.

```
beginfig(11);
draw bevingraph(43u,30u);
setrange(0,whatever,whatever,whatever);
lisse:=1;gdraw "data1" cl(0.5);lisse:=0;
frame.llft; %Ex 1
%frame.rt; %Ex 2
glabel.bot("Age des individus",OUT);
picture legendey;legendey=thelabel
("Population en millions d'individus", (0.0))
ratated 90; glabel(legendey,OUT);
%% label supplémentaire
gdraw(55,2.129)--(70,3)lw(0.2pt);
gdraw(45,2.834)--(70,3)lw(0.2pt);
faitlabille;
glabel(bille,55,2.129);
glabel(bille,45,2.834);
glabel.top("War II",70,3);
endgraph;
endfig;
```



## 6.5 Légende des axes et labels supplémentaires

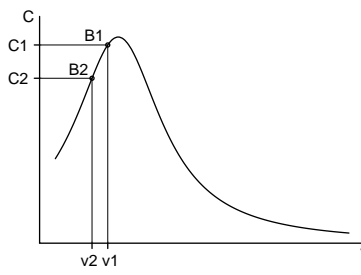
Il est important pour le lecteur du document de rappeler sur les figures les grandeurs représentées sur les axes. Pour cela, on dispose de la macro [144] : `glabel.suff("Texte",OUT)` ; où `suff` est un suffixe qui prend les valeurs déjà rencontrées : `bot` pour l'axe des abscisses, `lft` pour l'axe des ordonnées et éventuellement `rt` pour le deuxième axe de coordonnées ; le résultat se trouve sur la la figure 10.

Pour être parfait, il faut encore faire tourner le texte de la légende de l'axe des ordonnées ; comme cela a été fait à la section 5.5, on utilise la macro `thelabel` pour créer une variable dessin nommée `legendey` située à l'origine et que l'on tourne de 90 degrés ; ensuite, la commande `glabel(legendey,OUT)` ; trace cette légende ; le résultat se trouve sur la figure 11.

Il y a des situations où la raison pédagogique nécessite quelques tracés supplémentaires absolument indispensables. On peut mettre en évidence certains points de la courbe par l'ajout d'une astérisque (ou d'un tout autre caractère : une bille sur l'exemple suivant). Cela se fait avec la macro `glabel` qui permet aussi de placer aussi un petit commentaire. On peut en outre ajouter des traits de « renvoi » afin de relier un commentaire au(x) point(s) concernés. Un exemple de ces possibilités est présenté sur la figure 11 pour signaler un « manque » d'individus ayant entre 45 et 55 ans qui correspond aux naissances pendant les années de guerre et les années d'incertitude précédentes (les données ont été recueillies en 1991).

On aborde enfin, figure 12, les adaptations possibles concernant les ticks.

```
%<def trait=image(draw(0,0.5bp)--(0,-2bp);undraw(0,0.5bp)--(0,2bp)) enddef;
%<def pxtick(expr x)=glabel(trait,x,0) enddef;
%<def pytick(expr y)=glabel(trait rotated-90,0,y) enddef;
beginfig(12)
def fonction(expr x)=1/(1+(x-1.5)**2) enddef;
draw begingraph(43u,30u);setrange(0,0,6.2,1.1);
path courbe;courbe=(0.3,fonction(0.3))for i=0.4
step 0.1 until 6:..(i,fonction(i)) endfor;
gdraw courbe;frame.llft;autogrid(,);
a=fonction(1);b=fonction(1.3);
glabel(bille,1,a);glabel(bille,1.3,b);
pxtick(1);pxtick(1.3);pytick(a);pytick(b);
glabel(bille,1,a);glabel(bille,1.3,b);
gdraw(1,0)--(1,a)--(0,a) lw(0.2pt);
gdraw(1.3,0)--(1.3,b)--(0,b) lw(0.2pt);
glabel.lft("C",0,1.1);glabel.bot("v",6.2,0);
labeloffset:=4.2bp;glabel.bot("v2 v1",1.15,0);
glabel.lft("C1",0,b);glabel.lft("C2",0,a);labeloffset:=2.2bp;
endgraph;
endfig;
```



On expose comment on peut, d'une manière très simple :

- placer des ticks supplémentaires (macros `pxtick` et `pytick` et même remplacer éventuellement les ticks par défaut :
- tracer des traits de renvoi entre les points de la courbe mis en évidence et leurs projections sur les axes. Ces derniers tracés ne peuvent être faits

que si les bornes inférieures données en premier et deuxième argument de `setrange` sont des nombres « acceptables », c'est-à-dire non écrits sous la forme appelée `men` à la section 6.3.

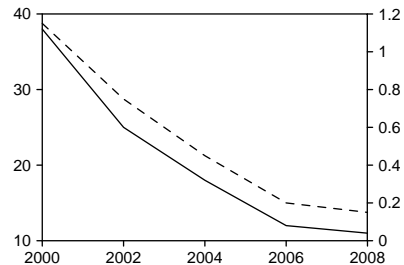
Il faut remarquer que les définitions `pxtick` et `pytick` contiennent respectivement les valeurs particulières à la figure courante  $x_{min}$  et  $y_{min}$  (0 et 0 dans le cas de la figure 12). Les labels sont placés tout simplement avec la macro `glabel` après avoir pris la valeur 4.2 bp pour la distance `labeloffset` afin que la distance tick-label pour les éléments ajoutés soit identique à cette même distance pour les éléments par défaut. Pour que  $v_1$  et  $v_2$  ne se chevauchent pas, on les a rassemblés en un seul label (séparés par un espace) placé entre les ticks correspondants.

Les macros du fichier `graph.mp` ont des possibilités limitées en contre partie de leurs avantages dont le principal est bien entendu la mise à l'échelle automatique des données. Cet avantage à un inconvénient : on ne peut pas affiner la position d'un lettrage autrement que par la modification de la distance, comme fait ci-dessus ; pour cela il faudrait savoir quel accroissement des  $x$  et des  $y$  correspond par exemple à un déplacement de 1 pt (METAPOST fait ce calcul en interne).

## 6.6 Difficultés dues aux grands nombres

Des difficultés ont été cachées : c'est le moment d'en parler ! On commence avec la figure 9 (où l'on avait dû faire une « réparation » pas très rigoureuse) pour laquelle les années concernées sont en vérité les années 2000 à 2008. Si l'on refait cette figure avec la modification proposée, alors on constatera, en reprenant la partie (Ex 1) et en cachant la partie (Ex 2), un problème d'affichage des graduations de l'axe des abscisses :

```
beginfig(13);
draw bevingraph(43u,30u);
setrange("2e3",10,"2.008e3",40);
gdraw "data8";
autogrid(,otick.lft);
setcoords(linear,linear);
setrange("2e3",0,"2.008e3",1.2001);
gdraw "data9" dashed evenly;
%autogrid(otick.bot,otick.rt);           %Ex 1
for x=2000,2002,2004,2006,2008:         %Ex 2
  otick.bot(format("%4g",x),x);endfor    %Ex 2
autogrid(,otick.rt);                     %Ex 2
endgraph;
endfig;
```



Le problème est résolu en annulant l'affichage des graduations de l'axe des abscisses avec `autogrid` et en le réalisant directement « à la main » et en dehors de `autogrid` avec la commande `format` que l'on va détailler plus loin ; il suffit de recacher la partie (Ex 1) et de reprendre la partie (Ex 2) pour avoir le résultat [152]. Cette méthode est utilisable chaque fois que les graduations par défaut sont incorrectes où ne sont pas celles que l'on voudrait.



Pour comprendre, il faut approfondir le problème des formats ; en réalité, on peut dire qu'il y a deux sortes de formats.

Les formats de composition (ainsi nommés car ils sont utilisés pour l'affichage des labels) sont choisis avec la commande `init.numbers` [152,161] donnée en tout début de ce chapitre afin visualiser les figures sans passer par  $\text{T}_{\text{E}}\text{X}-\text{L}_{\text{A}}\text{T}_{\text{E}}\text{X}$  mais en utilisant la fonte PostScript Helvetica ; pour comprendre le rôle des paramètres de cette commande, il suffit d'en considérer un : si on avait écrit `thelabel("2", (0,0))` au lieu de `thelabel("2", (1pt,2pt))`, les puissances de dix seraient trop près du caractère « e » et pas assez hautes, voir la figure 8. C'est ce format qui va changer lorsque l'on intégrera les figures dans un document  $\text{L}_{\text{A}}\text{T}_{\text{E}}\text{X}$ , chapitre 7.

Les formats de production (ainsi appelés car ils sont utilisés dans les calculs) sont des chaînes de caractères du type [158] :

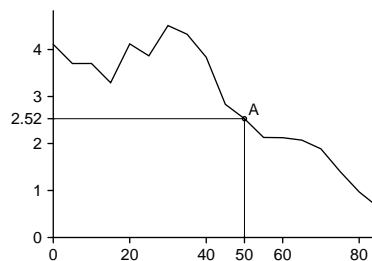
`"%px"`

- `p` est un nombre optionnel représentant la précision ; par défaut,  $p = 3$  ;
- `x` est une lettre `e`, `f`, `g` ou `G` ; pour les choix `e` et `f`, `p` est le nombre de chiffres significatifs après arrondi ; pour les choix `f` et `G`, le nombre est arrondi au multiple de  $10^p$  le plus près ;

On revient à notre exemple : le format par défaut `"%g"` arrondit 2000, 2001, . . . 2004 à 2000 et 2005, 2006, . . . à 2010, d'où la nécessité d'imposer le format `"%4g"` ( $p = 4$ ) pour avoir les labels de l'axe des abscisses correct.

Il est difficile de prévoir tous les cas qui mériteraient un exemple. On va se borner à une difficulté cachée : pour la figure 1, le nombre d'habitants avait été divisé par un million, tâche fastidieuse dans les cas de gros fichiers. En cachant la partie (Ex 2) et en prenant la partie (Ex 1), les graduations ont la forme `1e6` ; cependant il est nettement préférable de revenir à une graduation en millions d'individus. Ce résultat peut s'obtenir directement en reprenant la partie (Ex 2) et en cachant la partie (Ex 1) ; la macro `Mreadpath` lit le fichier de données et permet de diviser les ordonnées par  $10^6$  à condition de modifier la variable interne `Gpaths` :

```
beginfig(14);
draw begingraph(43u,30u);
setrange(0,0,85,4.8e6);
%gdraw "data10";
Gpaths:=log;
gdraw Mreadpath("data10")shifted(0,-6*Mten);
Gpaths:=linear;
glabel(bille,50,2.528);glabel("A",50,2.528);
glabel(trait,50,0);
glabel(trait rotated -90,0,2.528);
labeloffset:=4.2bp;glabel.lft("2.52",0,2.528);
glabel.bot("50",50,0);labeloffset:=2.2bp;
gdraw(-5,2.528)--(50,2.528)lw(0.3pt);
frame.llft;
endgraph;
endfig;
```



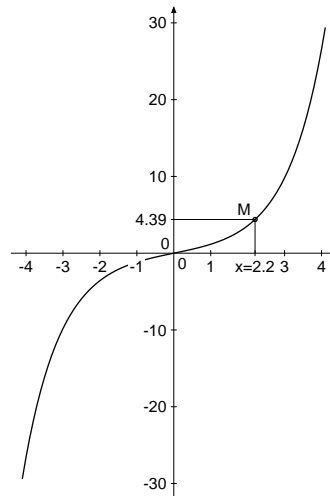
La macro `Mreadpath` [156] lit un fichier de coordonnées dont la structure est celle du fichier utilisé pour la figure 1 et donne le chemin correspondant ; il

faut ensuite préciser comment doivent être traitées les données lues : pour le cas présent, la variable interne `Gpaths` (valeur par défaut `linear`) doit avoir la valeur `log`. Ce chemin est ensuite tracé avec `gdraw` après avoir subi une translation, mais ici une translation de  $-6$  sur les puissances de 10 (la variable `Gpaths` a pour valeur `log`) est bien une division par un million ! On a mis en évidence un point de la courbe et on a vérifié que l'on pouvait tracer les traits de rappel, mais ceci après avoir modifié la valeur de l'abscisse minimum (cf. le réserve faite à la fin de la section 6.5).

## 6.7 Tracé de courbes METAPOST « relookées »

Il est dommage de se priver de la puissance des macros fournies dans le fichier `graph.mp`, notamment de la machinerie qui fait la mise à l'échelle : on peut imposer des dimensions au tracé sans modifier les données. Non seulement METAPOST fait cette mise à l'échelle mais calcule les valeurs des graduations. On peut exploiter cette puissance et obtenir une présentation plus discrète avec quelques macros (fournies dans le fichier `graphsupsup.mp`) exploitant les résultats de calculs faits en interne. Voilà la méthode à suivre :

```
beginfig(15);
% Partie commune : 5 lignes
def sinh(expr x)=(2.7**x-2.7**-x)/2 enddef;
draw begingraph(43u,65u);setrange(-4.4,-32,4.4,32);
path psinh;
psinh=(-4.1,sinh(-4.1))
for i=-4 step 0.1 until 4.2:--(i,sinh(i)) endfor;
gdraw psinh lw(0.6pt);
% partie Ex 1 : 7 lignes
pmaxes(-4.4,-32,4.4,32);glabellrt("0",0,0);
glabelluft("0";0,0);pmxticks(1,-4,1);
pmxticks(1,3,4);pmyticks(10,-30,30);
numeric N;N=sinh(2.2);glabell("bille",2.2,N);
pmyticlab(format("%3g",N),N);
glabelluft("M",2.2,N);
gdraw(2.2,0)--(2.2,N)--(0,N)lw(0.4pt);
% Partie Ex 2 : 8 lignes
%pmaaxes(-4.4,-32,4.4,32,-2,10);
%pmxxticlab("0",0,10);pmxxticlab("2",2,10);
%pmxxticlab("3.5",3.5,10);
%pmyyticlab("20",20,-2);pmyyticlab("30",30,-2);
%numeric N;N=sinh(3.5);glabell("bille",3.5,N);
%pmyyticlab(format("%4g",N),N,-2);
%glabelluft("10",-2,10);glabellrt("-2",-2,10);
%gdraw(3.5,10)--(3.5,N)--(-2,N)lw(0.4pt);
% Partie Ex 3 : 1 ligne
Gneedfr_:=false;Gneedgr_:=false;
endgraph;
endfig;
```



**Etape 1 :** On fait le tracé de la courbe normalement, c'est-à-dire en cachant les petites parties de code (Ex 1), (Ex 2) et (Ex 3); on peut utiliser des améliorations à condition de garder le cadre et les graduations par défaut. Dans le cas de la

figure 15 (représentation de la courbe sinus hyperbolique), un calcul élémentaire nous suggère de prendre  $|x_{\min}|$  et  $|x_{\max}|$  ne dépassant pas 4 ou 5 pour que  $e^x$  soit inférieur à 1000) : on choisit 4. On compile *uniquement* la partie commune du code et on regarde le résultat à l'écran : compte tenu des graduations affichées, on choisit 32 pour  $|y_{\min}|$  et  $|y_{\max}|$ .

**Étape 2 :** On peut alors ajouter la commande `setrange` avec les 4 valeurs choisies ; cet ajout est facultatif, il a pour but de diminuer l'espace entre la courbe et le cadre ; en effet, sans cette commande, la valeur par défaut de cet espace sera trop grande et il y aura trop d'espace entre la figure et sa `BoundingBox`. Après cette petite amélioration recommandée, on passe à l'opération principale : avec la partie (Ex 3), on supprime le cadre et les graduations par défaut et, avec la partie (Ex 1), on trace de nouveaux axes gradués en reproduisant les valeurs des graduations lues sur le résultat de la première étape. Pour cela on utilise les macros suivantes.

- `pmaxes(xmin,ymin,xmax,ymax)` ;

trace les axes.

- `pmxticks(xstep,xxmin,xxmax)` ;

trace les graduations de l'axe des abscisses ; pour `xxmin` et `xxmax` on choisit (toujours d'après la visualisation) un pas de 1 et les deux couples de bornes  $-4, 1$  et  $3, 4$ , pour éviter de tracer la graduation en  $x = 2$  ; la raison en est que l'on veut éviter un recouvrement de lettrages par la suite. Le tick et le label en  $x = 0$  ne sont pas tracés ; si nécessaire, on rajoute le label « à la main » avec `glabel`, en général dans le cadran bas droit.

- `pmyticks(ystep,ymin,ymax)` ;

trace les graduations de l'axe des ordonnées avec les mêmes notations et les mêmes possibilités. On choisit un pas de 10 et le couples de bornes  $-30, 30$  ; il faut veiller à ce que la boucle `for ... endfor` de la macro « tombe » sur la valeur 0 ; on pourrait prendre le pas 0.5 et les bornes  $-25, 25$  (bien entendu, la symétrie par rapport à 0 n'est pas nécessaire). rien d'obli.

- `Gneedfr_:=false;Gneedgr_:=false;`

font respectivement disparaître le cadre et les graduations par défaut.

On a aussi ajouté quelques commandes pour mettre en évidence un point de la courbe avec les lignes de rappel vers les axes. Cela conduit à quelques remarques.

- Les labels sont posés sur un petit rectangle « effacé » pour favoriser leur lecture : dans certains cas, on peut recouvrir parfaitement un label avec un autre label mais la vraie solution consiste à éviter de tracer le label qu'il faudra effacer.
- Pour tracer des graduations supplémentaires ou remplaçant les graduations automatiques ne convenant pas, on utilise les macros `pmxticlab(lab,x)` ; et `pmyticlab(lab,y)` ; et, pour tracer de labels supplémentaires, notamment les 0 à l'intersection des axes, on utilise `glabel.suff(lab,x,y)` ; où `x` et `y` sont les abscisses et les ordonnées correspondantes. L'argument

`lab` peut être une *chaîne* et on écrit alors "`ab`" ou "`3.4`"; il peut aussi être un nombre `n` préalablement fixé ou calculé par METAPOST auquel cas on écrit alors `decimal n`, où `decimal` est une commande qui prend un nombre et donne la chaîne correspondante, ou bien encore [158]

```
format("%3g",n)
```

pour se limiter à 3 chiffres significatifs (ce qui suffit largement pour une figure).

La procédure de remplacement des axes et des graduations a des limites que l'on peut contourner assez facilement.

- Les macros `pmxticks` et `pmxticks` ne peuvent pas être utilisées dans le cas de graduations non linéaires (cas des arguments `log` de la commande `setcoords`). Il faut alors placer les graduations une par une avec les macros `pmxticlab(lab,x)`; et `pmyticlab(lab,y)`;
- Ces quelques macros élémentaires précédentes ne peuvent pas traiter le cas où les axes ne se coupent pas au point  $(0,0)$ . Si ce point d'intersection est  $(a,b)$ , on tracera les axes avec la macro `pmaaxes(xmin,ymin,xmax,ymax,a,b)`; et on placera les graduations « à la main » avec les macros `pmxxticlab(lab,x,b)`; et `pmyyticlab(lab,y,a)`; en remplacement des macros précédentes, partie (Ex 2) de la figure 15. Les commandes `glabel` et `gdraw` s'utilisent toujours de la même manière. On peut faire un test en cachant la partie (Ex 1) et en découvrant la partie (Ex 2).

En guise de conclusion de cette section, on peut dire que, si on ne peut pas tout faire, on peut tout de même faire pas mal de choses pédagogiquement valables; sont évidemment exclues toutes les innombrables fioritures qui remplissent certains ouvrages du secondaire et dont on peut discuter l'utilité quand on est face à certains groupes de TD de première année d'université. L'auteur de ce petit document pense que l'idéal serait de reprendre `graph.mp` en y intégrant au moins les fonctionnalités fournies par `graphsupsup.mp`; en effet, il faudrait des macros pour ajouter des ticks sans label et quelques autres petites choses. Ce n'est peut-être pas un tâche énorme : en effet, on peut remarquer que `gdraw` permet de déplacer les axes « d'origine » mais les graduations au point d'intersection vont se superposer...

## METAPOST et T<sub>E</sub>X

Ce chapitre est consacré à l'utilisation conjointe de T<sub>E</sub>X et METAPOST, plus précisément à l'intégration dans des documents T<sub>E</sub>X ou L<sup>A</sup>T<sub>E</sub>X de figures faites avec METAPOST et portant des lettrages<sup>(1)</sup> composés avec T<sub>E</sub>X ou L<sup>A</sup>T<sub>E</sub>X. Il y a deux possibilités :

- on fait les figures, on y ajoute les lettrages puis on les intègre dans le document ;
- on inclue le code des figures dans le document lui-même et un traitement du document conduit au résultat final de façon transparente.

La deuxième possibilité, initialement utilisable seulement dans le cadre de CONTEX<sub>T</sub>, est maintenant disponible pour les utilisateurs de L<sup>A</sup>T<sub>E</sub>X. Bien entendu, cette solution exige l'utilisation d'extensions appropriées. Le lecteur se doute bien qu'un premier traitement va extraire le code des figures qui devra être traité par METAPOST ... et qu'un autre traitement va placer la figure à la place demandée (mais grâce à la primitive T<sub>E</sub>X `\write18`, on a l'impression qu'il n'y a qu'un seul traitement).

Pour celui qui écrit ces quelques lignes, il semble que, dans le cas d'un auteur assurant lui-même la composition de son ouvrage, la première solution est de loin la plus pratique : qui n'a jamais modifié une figure en cours de réalisation et qui n'a jamais adapté une figure au texte explicatif (ou la réciproque quand la figure est terminée) ? Laissons de côté cette discussion et passons au vif du sujet.

### 7.1 Etapes du lettrage des figures

On va créer un document qui est en fait le présent chapitre de ce petit manuel. On commence par créer un fichier de figures `figz.mp` en reprenant le préambule des fichiers de figure précédents, en y enlevant les trois instructions qui concernent l'utilisation de la fonte Helvetica (sect. 1.3) et en y remplaçant

---

<sup>(1)</sup> Lettrage est utilisé ici pour tout élément de figure exigeant d'être composé avec T<sub>E</sub>X ou L<sup>A</sup>T<sub>E</sub>X : caractères mathématiques, formules, éléments de texte, etc.

la commande de formatage de composition `init_numbers(...)` (cf. introduction du chap. 6) par :

```
init_numbers(btex$- $\etex$ ,btex$1 $\etex$ ,btex$\times 10 $\etex$ ) $\etex$ ,
             btex$\{ $\}^{\sim}$ - $\etex$ ,btex$\{ $\}^{\sim 2}$  $\etex$ )
```

pour avoir une composition correcte de la notation scientifique des grands nombres.

La première figure que l'on fait comprend :

- les axes et un point de coordonnées  $x = 13$  mm et  $y = 19$  mm,
- le lettrage « cas 1) :  $x \in \mathbb{Q}$  » affecté à ce point.

La commande pour ce lettrage est [46] :

```
label.top(btex cas 1):  $\$x\in\mathbb{Q}\$$   $\etex$ , (13u,9u));
```

La macro `label.suff` est toujours utilisée : son premier argument est le « matériel » qui doit être composé par  $\TeX$  entouré par le couple d'instructions `btex ... etex`.

Pour que cette composition soit possible, il faut que  $\TeX$  ait un certain nombre de directives pour faire cette composition. Dans l'exemple choisi, pour avoir le caractère  $\mathbb{Q}$  ajouré, il faut utiliser l'extension `amsmath`, etc. On rajoutera donc, avant le code des figures les instructions :

```
verbatim $\etex$ 
\documentclass{book} ou {article}
\usepackage{amsmath,amssymb}
 $\etex$ 
```

D'un façon générale, il faut ajouter, entourées par les instructions [48]

```
verbatim $\etex$  ...  $\etex$ ;
```

toutes les commandes que  $\TeX$  doit utiliser pour composer le « matériel » du lettrage, en particulier, tout ce qui concerne les fontes et leurs codages.

Lorsque l'on fait une figure complexe, il peut être avantageux de faire dans un premier temps la figure sans lettrage et de la tester ; à chacun de voir ce qui lui convient le mieux !

### 7.1.1 Production des fichiers `tex` et `dvi` des lettrages

Le fichier `figz.tex` est extrait du fichier `figz.mp` grâce à un utilitaire nommé `mpto` [168] :

```
mpto figz.mp > figz.tex
```

Il contient uniquement la partie  $\TeX$ - $\LaTeX$  des commandes de lettrages ; il est instructif d'éditer ce fichier `figz.tex` pour voir le résultat de l'extraction faite par `mpto` et constater que son traitement par  $\LaTeX$  va donner des « petites boîtes » contenant ces lettrages. Ce traitement se fait par :

```
latex figz.tex
```

pour donner le fichier `figz.dvi`.

### 7.1.2 Production du fichier `mpx` des lettrages et des fichiers de figures

Le fichier de lettrage `figz.dvi` est alors transformé en un fichier de commandes de bas niveau `METAPOST` [168] :

```
dvitompx figz.dvi figz.mpx;
```

Ce fichier `figz.mpx` peut être édité et, malgré son aspect peu engageant, on pourra y reconnaître des positionnements, des caractères et les noms L<sup>A</sup>T<sub>E</sub>X des fontes utilisées.

On termine par le traitement normal du fichier de figure :

```
mp figz.mp
```

qui produit les fichiers POSTSCRIPT `figz.1`, `figz.2`, etc. Dans cette opération, lorsque METAPOST rencontre le groupe `btex ... etex`, il cherche le fichier `figz.mpx` de même nom et y prend le code METAPOST de bas niveau correspondant aux lettrages pour écrire le code POSTSCRIPT complet des figures.

### 7.1.3 Automatisation

Une fois que l'on a compris ce que l'on doit faire, on peut le rendre transparent en définissant par exemple le fichier de commandes `mmp.bat`

```
mpto %1.mp > %1.tex
```

```
latex %1.tex
```

```
dvitompx %1.dvi %1.mpx
```

```
mp %1.mp
```

qui prend en charge toutes les opérations (utilisation : `mmp figz`).

```
%%% Fichier de figures figz.mp
```

```
%%% 1) Preamble
```

```
% Affectations generales
```

```
u=1mm;linejoin:=mitered;ahlength:=2pt;prologues:=1;labeloffset:=2pt;
```

```
% Macros utiles
```

```
input macutil.mp;
```

```
% Pour les boites et liaisons
```

```
input rboxes.mp;input boitesup.mp;
```

```
rbox_radius:=3pt;bboxmargin:=1pt;
```

```
% Directives pour composition
```

```
verbatimtex
```

```
\documentclass{book}
```

```
\usepackage[cp850]{inputenc} % Pour editeur DOS
```

```
\usepackage[T1]{fontenc}
```

```
\usepackage{amsmath,amssymb}
```

```
\usepackage{aeguill} % Pour ce manuel seulement
```

```
\footnotesize
```

```
\def\fnsz{tiny}
```

```
etex
```

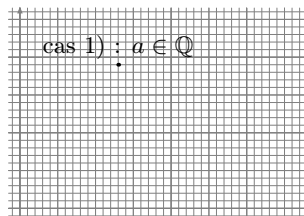
```
%%% 2) Figures
```

```
beginfig(1);
```

```
axespapiermilli(-1.5,-1.5,38.5,26.5);drawpt((13u,19u),1.5pt);
```

```
label.top(btex \footnotesize cas 1): $ a\in\mathbb{Q} $ etex,(13u,19u));
```

```
endfig;
```



On constate sur le code précédent que, partout où l'on avait une chaîne de caractères à composer sous la forme "abcde", il faut la remplacer par l'ensemble : `btex` (composition en L<sup>A</sup>T<sub>E</sub>X) `etex`.

Reste à savoir si l'on peut voir directement la figure `figz.1` correspondant au fichier `figz.mp` dont le début est ci-dessus. Bien sûr qu'on ne peut pas la voir car ce fichier POSTSCRIPT ne contient pas les dessins des caractères des fontes utilisées (dessins contenus dans les fichiers `.pfa` ou `.pfb`)! Si l'on introduit

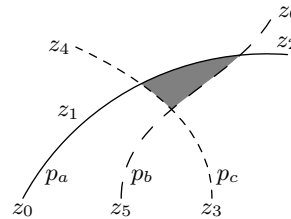
dans le fichier `figz.mp` l'instruction `prologues:=1`; ou `:=2`; au lieu de `:=0`; alors on peut, en général, voir la figure où chaque caractère est représenté par le caractère de la fonte par défaut de même code s'il existe.

Pour voir la figure et ses « vrais » lettrages, il faut l'intégrer dans le document. Le fichier POSTSCRIPT du document complet créé par DVIPS contient les dessins des caractères des fontes utilisées (dessins pris dans les fichiers `.pfa` ou `.pfb`), ce qui permet de voir les « vrais » lettrages.

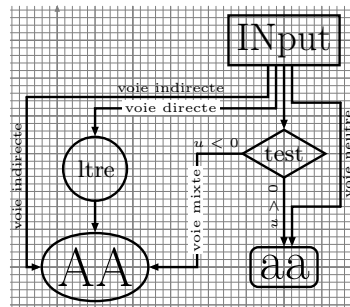
## 7.2 Lettrage en L<sup>A</sup>T<sub>E</sub>X de quelques figures

On reprend d'abord les figures 3.11 et 5.8 en y introduisant les lettrages « à la L<sup>A</sup>T<sub>E</sub>X »; on ne porte sur les listings que les lignes changées.

```
beginfig(2);
%% Rreprise de la figure 3.11
...
label.bot(btex $ z_{0} $ etex,z0);
label.ulft(btex $ z_{1} $ etex,z1);
label.top(btex $ z_{2} $ etex,z2);
label.bot(btex $ z_{3} $ etex,z3);
label.lft(btex $ z_{4} $ etex,z4);
label.bot(btex $ z_{5} $ etex,z5);
label.rt(btex $p_{a}$ etex,(6.5pt,8.1pt));
label.rt(btex $p_{c}$ etex,(71pt,8.1pt));
label.rt(btex $p_{b}$ etex,(38.7pt,8.1pt));
label.rt(btex $z_{6}$ etex,z6+(0,1.5pt));
endfig;
```



```
beginfig(3);
%% Rreprise de la figure 5.8
axespapiermilli(-6.2,-5.5,39.8,34.5);
ellit.ka(btex \huge AA etex);tracebe.ka((5u,0),0);
rboxit.kb(btex \huge aa etex);tracebe.kb((30u,0),0);
cirit.kc(btex filtre etex);tracebe.kc((5u,13u),0);
diamit.kd(btex test etex);
tracebe.kd((30u,15u),0);
boxit.ke(btex \Large INput etex);
tracebe.ke((30u,30u),0);
...
picture laba,labb;
label.top(btex \fnsz voie indirecte etex,
(15u,22.5u));
laba=thelabel(btex \fnsz voie directe etex,
(15u,21u)+0.5*up);
unfill bbox laba; draw laba;
label.lft(btex \fnsz voie indirecte etex
rotated 90,(xpart ka.w-2u,12u));
label.top(btex \fnsz $u<0$ etex,(21u,15u));
label.lft(btex \fnsz $u>0$ etex rotated 90,(30u,8u));
label.rt(btex \fnsz voie neutre etex rotated 90,(xpart kd.e+2u,15u)+left);
labb=thelabel(btex \fnsz voie mixte etex rotated 90,(18.5u,7.5u)+0.5*left);
unfill bbox labb; draw labb;
endfig;
```





Pour la figure 3.11, on constate que l'on a du corriger la position des noms de deux des chemins car le pied de la lettre italique *p* avait tendance à recouvrir le chemin lui-même.

Pour la figure 5.8, on a aussi fait des corrections car, dans le cas particulier de cette figure, on a des textes sans lettre à pied et avec très peu de lettres à hampe ; le résultat est que le texte vertical « voie indirecte » semble plus près de la liaison que le texte « voie neutre ».

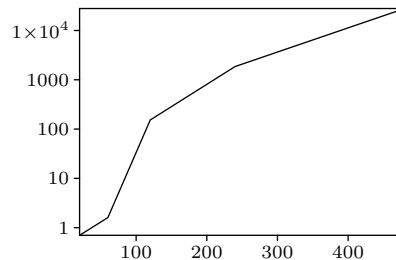
Ce type de correction est la plupart du temps superflu mais on peut rencontrer des cas où une intervention est souhaitable (cas de caractères mathématiques avec exposants ou indices, par exemple). La correction apparaît d'autant plus nécessaire que l'on diminue la valeur de `labeloffset` ; cependant, une faible valeur améliore, aux yeux de l'auteur de ces lignes, la qualité des figures ; à chacun de choisir !

L'intégration de ces figures dans un document L<sup>A</sup>T<sub>E</sub>X se fait normalement avec l'extension `graphicx` (voir début du chapitre 1). Il est signalé que, si le document est composé avec `pdflatex` [F 171], on peut obtenir directement le fichier `.pdf` ; mais cela nécessite une variante de l'exécutable PDF<sub>T</sub>E<sub>X</sub> [F 171].

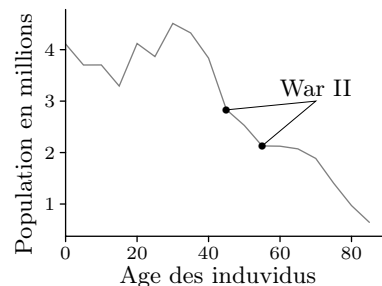
### 7.3 Lettrage en L<sup>A</sup>T<sub>E</sub>X de quelques graphiques

On reprend directement les figures 6.8 (pas de modification nécessaire) et 6.11 (modification du codage des trois chaînes de caractères). On constate que la taille obtenue par l'introduction de `\scriptsize` dans la macro `init_numbers` est suffisante.

```
beginfig(4);
draw begingraph(43u,30u);
setcoords(linear,log);
%setrange(20,0.007,480,279.3);
%gdraw "data4";
setrange(20,0.7,480,"2.793e4");
gdraw "data5";
endgraph;
endfig;
```



```
beginfig(5);
draw begingraph(43u,30u);
setrange(0,whatever,whatever,whatever);
gdraw "data1" cl(0.5);
frame.llft;
glabel.bot(btex Age des individus etex,OUT);
picture legendey; legendey=thelabel%
(btex Population en millions
d'individus etex,(0,0)) rotated 90;
glabel.lft(legendey,OUT);
gdraw(55,2.13)--(70,3)--(45,2.83)lw(0.2pt);
glabel(bille,55,2.13);glabel(bille,45,2.83);
glabel.top(btex War II etex,70,3);
endgraph;
endfig;
```



On termine en reprenant la figure 15 qui présente des difficultés résultant des principes mêmes de la méthode de production des lettrages. On rappelle que l'on doit remplacer toutes les occurrences de chaînes du type "abcd" par des ensembles du type `btex` (composition en L<sup>A</sup>T<sub>E</sub>X) `etex`. Mais les macros qui tracent les graduations, `pmxticlab` par exemple, contiennent des boucles du type suivant :

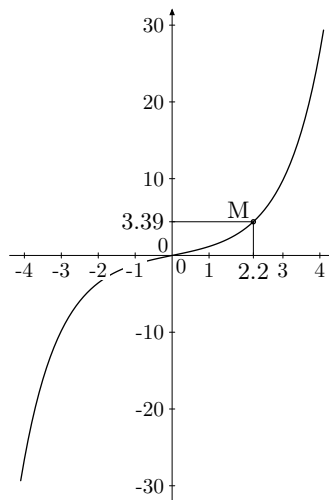
```
for i=xmin.....glabel(decimal i,i,0)...endfor;
```

Pour extraire les lettrages à composer, `mpto` repère les couples `btex...etex`, il ne peut pas voir ceux qui sont dans la macro : il ne voit que le nom de la macro et passe à la suite : les ensembles `btex...etex` ne peuvent pas être dans un macro ... et c'est pour cela que, si cette prescription n'est pas respectée, la compilation donne un message d'erreur du type *commande interdite dans la définition etc.* Pour les graduations des axes on garde les définitions contenant des boucles dans lesquelles on ne touche rien ; par contre on va ajouter dans le prologue les commandes :

```
defaultfont:="CMR10"; et defaultscale:=0.5;
```

Cette facilité ne perturbe pas la qualité de la figure : le seul reproche que l'on peut faire concerne le point décimal (au lieu de la virgule). Ici, les tailles sont données à titre indicatif ; sur une figure plus grande, on pourrait choisir le facteur d'échelle 0.9, ce qui correspond à la taille par défaut `small` du préambule. Par contre les chaînes de caractères sont traitées comme cela a été expliqué.

```
beginfig(6);
%% Partie commune
def sinh(expr x)=(2.7**x-2.7**-x)/2 enddef;
draw begingraph(43u,65u);setrange(-4.4,-32,4.4,32);
path psinh,pcosh;
psinh=(-4.1,sinh(-4.1))
  for i=-4 step 0.1 until 4.2:--(i,sinh(i)) endfor;
gdraw psinh lw(0.6pt);
% partie Ex 1 : 10 lignes
pmaxes(-4.4,-32,4.4,32);glabel.lrt("0",0,0);
defaultfont:="CMR7";defaultscale:=0.9;
pmxticks(1,-4,1);pmxticks(1,3,4);
pmyticks(10,-30,30);
glabel.lrt("0",0,0);glabel.ulft("0",0,0);
pmxticlab(btex\kern-6pt$x=2.2$etex,2,2);
numeric N;N=sinh(2.2);glabel(bille,2,2,N);
pmyticlab(format("%3g",N),N);
glabel.ulft(btex$\mathrm{M}$ etex,2,2,N);
gdraw(2.2,0)--(2.2,N)--(0,N)lw(0.4pt);
% Partie Ex 2 : 9 lignes
%defaultfont:="CMR7";defaultscale:=0.8;
%pmaaxes(-4.4,-32,4.4,32,-2,10);
%pmxticlab("0",0,10);pmxticlab("2",2,10);
%pmyticlab("20",20,-2);pmyticlab("30",30,-2);
%numeric N;N=sinh(3.5);glabel(bille,3,5,N);
%glabel.ulft("10",-2,10);glabel.lrt("-2",-2,10);
%defaultscale:=0.9;pmxticlab("3.5",3,5,10);
%pmyticlab("16.5",N,-2);%(format("%3g",N),N,-2);
%gdraw(3.5,10)--(3.5,N)--(-2,N)lw(0.4pt);
% Partie Ex 3 : 1 ligne
Gneedfr_:=false;Gneedgr_:=false;
endgraph;
endfig;
```



## 7.4 METAPOST et T<sub>E</sub>X, tout à la fois !

Comme l'auteur de ces lignes ne pense pas qu'un débutant puisse se lancer dans la composition d'un long document avec de nombreuses figures en suivant cette méthode [172–175], il est simplement donné un petit exemple que l'on peut facilement tester dans un autre répertoire que celui des exercices précédents, `mpb` par exemple. Le traitement se fait en donnant l'option autorisant l'utilisation de la primitive T<sub>E</sub>X `\write18` :

```
latex -shell -escape testmp

%%% fichier testmp.tex
%\documentclass[pdftex]{article}
\documentclass[dvips]{article}
\usepackage[cp850]{inputenc}
%\usepackage[latin1]{inputenc}
\usepackage[T1]{fontenc}
\usepackage{amsmath,amssymb}
\usepackage{graphicx,emp,ifpdf}
%\DeclareGraphicsRule{*}{mps}{*}{*}
\usepackage{aeguill}
\usepackage[frenchb]{babel}
\begin{document}
%% Debut du code figure
\empaddtoTeX{
  \usepackage[cp850]{inputenc}
  \usepackage[T1]{fontenc}
  \usepackage{amsmath,amssymb}
  \usepackage{aeguill}
  \usepackage[frenchb]{babel}
  \begin{empfile}
  \begin{empcmds}
    u=1mm;linejoin:=mitered;ahlength:=2pt;
    prologues:=1;labeloffset:=2pt;
    input c:/mp/macutil.mp;input c:/mp/rboxes.mp
    input c:/mp/boitesup.mp;rbox_radius:=3pt;
  \end{empcmds}
  \begin{empdef}[fig1](5cm,5cm)
    axespapiermilli(-1.5,-1.5,38.5,26.5);
    drawpt((13u,19u),1.5pt);
    label.top(btex \footnotesize cas 1):
      $ a\in\mathbb{Q} $ etex,(13u,19u));
  \end{empdef}
  \end{empfile}
  \immediate\write18{mp -tex=latex \jobname}
  % Debut texte avec integration
  Ceci est un petit exemple pour tester le \og tout en une fois
  \fg{} et se faire une opinion sur ce type de methode. On
  verra bien comment ça marche !
  $$ \empuse{fig1} $$
  Encore le même texte : ceci est un petit exemple pour tester
  le \og tout en une fois \fg{} et se faire une opinion sur ce
  type de methode. Oui, ça à marché !
\end{document}
```

Le fichier ci-dessus correspond au traitement par L<sup>A</sup>T<sub>E</sub>X et DVIPS. La même chose est également possible pour un traitement par P<sub>D</sub>F<sub>T</sub>E<sub>X</sub> [F 172-174].



# Conclusion

Avant la conclusion proprement dite, on signale qu'il y a de très nombreuses extensions disponibles pour METAPOST ; elles sont en grande majorité très spécialisées. C'est à chacun de savoir si, pour une tâche limitée (et exceptionnelle), il est plus rentable d'écrire quelques macros (peut-être pas très bien optimisées) que de se plonger dans le manuel d'une extension avec le risque de découvrir après des essais qu'elle ne convient pas du tout. Un chercheur de théorie des particules peut valablement se plonger dans l'extension FEYNMAN, mais pour illustrer un paragraphe d'un article avec un diagramme de Feynman, il est bien plus avantageux de le créer directement (en s'aidant d'un modèle pour respecter les conventions traditionnelles). Voici une liste non exhaustive d'extensions suivies d'une très succincte description :

`textpath` : écriture le long d'une courbe,  
`venn` : diagramme de Venn,  
`threed` : dessin en 3 D,  
`splines` : tracé de splines,  
`slideshow` : transparents de présentation,  
`piechartmp` : camemberts pour statisticiens,  
`misc` : objets de base en 3 D,  
`mfpic` : macros générales ; liste de couleurs (en CMYK),  
`metaobj` : « `pstricks` écrit en METAPOST »,  
`latexmp` : macros de gestion (type `etex...btex`),  
`hatching` : fonds hachurés,  
`feynmf` : diagrammes de Feynman,  
`featpost` : figures complexes (contributions multiples),  
`exteps` : pour compléter en surcharge des `.eps`,  
`metaULM` : nœuds et liaisons inter-nœuds,  
`cmarrows` : flèches, accolades, etc.,  
`mppattern` : motifs de remplissage,  
`metaplot` : tracé de graphiques,  
`makecirc` : dessin de circuits électroniques,  
`m3d` : animation en 3 D,  
`expressg` : organigrammes élémentaires,  
`graph` : tracé de graphiques (cf. chap. 6),  
`graphsup` : quelques macros complémentaires à `graph` (cf. chap. 6),  
`config` : fichier d'initialisation de METAPOST.

En outre, CONTEXT contient des modules utilisables dans l'environnement L<sup>A</sup>T<sub>E</sub>X ; par exemple, le module METAFUN permet d'obtenir des remplissages dégradés allant d'une couleur à une autre en passant par une troisième.

Comme on peut le constater, il est parfois difficile de faire un choix ; par contre, on peut signaler que `mfpic` contient de nombreuses macros très utiles, fonctions trigonométriques inverses par exemple, (fichier `graphbase.mp`), ainsi que les définitions des 64 couleurs de la table de Hafner (fichier `dvipsnam.mp`).

Il semble évident que l'utilisation de METAPOST demande plus d'investissement que l'utilisation de TikZ ; cela parce que METAPOST est un vrai langage avec sa propre syntaxe alors que TikZ est un outil de dessin d'objets graphiques intégré dans le « monde » L<sup>A</sup>T<sub>E</sub>X (bien qu'ayant une syntaxe spécifique en ce qui concerne les paramètres et les options des commandes) ; la dernière phrase du premier paragraphe de l'introduction précise le sens de ce commentaire : TikZ dessine, METAPOST construit.

On ne peut en dire plus car la personnalité de l'utilisateur potentiel joue un rôle important. C'est pour cela que l'auteur de ces quelques lignes a voulu présenter METAPOST d'une manière telle que, assez rapidement (autrement dit avec un langage assez simple et sans trop perdre de temps), on puisse découvrir METAPOST avec ses avantages et ses inconvénients ; il espère y avoir réussi !

Pour terminer, le lecteur pourra reproduire l'illustration du phénomène de l'arc-en-ciel figurant sur la couverture du n° 41 de la revue dans laquelle ce manuel a été publié ; tout le parcours des rayons lumineux (une réfraction à l'entrée, une réflexion interne et une autre réfraction à la sortie de la goutte d'eau) a été calculé entièrement par METAPOST (fichiers `couv-gut.mp` et `couv-gut.ps`). Encore un exemple où METAPOST fait les calculs complètement : l'effet de rotation du périhélie de Mercure (fichier `figy2.pdf` ; l'effet est fortement amplifié car sur 5 ans, il ne serait pas visible).

# Index

## Symbols

`%3g`, 68  
`%4g` (format `-`), 65  
`%<`, 2  
`%g` (format `-`), 65  
`%pg %pe` (formats `-`), 65  
`&`, 27, 34  
`*`, 31, 32  
`**`, 31  
`+`, 31, 32  
`++`, 31  
`-`, 31, 32  
`--`, 5  
`..`, 15  
`...`, 19  
`/`, 31, 32  
`<`, 31, 34  
`<=`, 31, 34  
`<>`, 31, 32, 34  
`=`, 31, 32, 34  
`>`, 31, 34  
`>=`, 31, 34

## A

`abs`, 31  
`addtocurrentpicture` also, 36  
`ahangle`, 9  
`ahlength`, 9, 37  
`and`, 35  
`angle`, 31  
`arcd`, 13  
`arclength`, 25, 34  
`arctime`, 34  
`arctime of`, 26  
ASCII, 34  
`augment.p`, 59  
`autogrid`, 64  
`autogrid.lft -bot -rt`, 61  
`axespapiermilli`, 1

## B

`bb.c`, 44

`bb.dx`, 44  
`bb.dy`, 44  
`bb.n -e -s -w`, 44  
`bb.ne -se -sw -nw`, 44  
`bb.off`, 44  
`bbox`, 53  
`bboxmargin`, 43, 53  
`bbtrace`, 49  
`beginfig`, 1, 30  
`begingroup`, 26  
`beveled`, 8  
`black`, 10  
`blue`, 10  
`bluepart`, 32  
`boitessup.mp`, 43  
`boitesup.mp`, 47  
`boolean`, 30  
`boxes.mp`, 39  
`boxit.bb`, 44  
`bpath.bb`, 46  
`btex ... etex`, 70, 74  
`btrace`, 48  
`buildcycle`, 22, 34  
`butt`, 8

## C

`ceiling`, 31  
`char`, 34  
`chemin`, 3, 12  
`circuit ou circleit`, 47  
`clip`, 5, 6  
`color`, 30  
`commandes (primitives et macros)`  
    les plus utilisées, 38  
`concaténation de chemins`, 27  
`controls and`, 16  
`cosd`, 31  
`couleur`, 12  
`courbe de Bézier`, 16  
`courbure près d'un point guide`, 19  
`création des figures`, 2

curl, 19  
 currentpicture, 6, 35, 37  
 cutafter, 26, 34, 50  
 cutbefore, 26, 34, 50  
 cyan, 10  
 cycle, 3, 15

## D

dashed evenly scaled, 9  
 dashed withdots scaled, 8  
 dashpattern, 9  
 débogage, 30, 39  
 decimal, 68  
 déclaration des variables, 12, 30  
 découper une partie d'une figure, 5  
 def ... enddef, 12  
 defaultdx, 44  
 defaultdy, 44  
 defaultfont, 4, 74  
 defaultpen, 7  
 defaultscale, 4, 74  
 dessin, 6, 12  
 diamit.bb, 47  
 dir, 31  
 direction de la tangente (donner la -),  
     18  
 direction of, 23, 34  
 directiontime of, 24, 34  
 div, 31  
 donnees.bb, 48  
 dotlabel, 58  
 down, 50  
 draw, 1, 2, 5  
 draw begingraph, 56  
 drawarrow, 9, 50  
 drawboxed, 44  
 drawbarrow, 9  
 drawoptions, 7  
 drawpt, 7  
 drawunboxed, 46  
 droite (représentation d'une) -, 11  
 dvitompx, 71

## E

effacement, 6  
 ellit.bb, 47  
 end, 1  
 endfig, 1  
 endgraph, 56  
 endgroup, 26  
 epsilon, 22

exitif, 41  
 exitunless, 41  
 expr, 12, 40

## F

facteur d'échelle, 2  
 fermeture d'un chemin, 15, 22, 27  
 figa.mp, 1, 2, 4  
 figb.mp, 4, 5  
 figc.mp, 4  
 figd.mp, 4  
 fige.mp, 43, 46  
 figf.mp, 55  
 figz.dvi, 70  
 figz.mp, 69, 70  
 figz.mpx, 71  
 figz.tex, 70  
 fill, 5  
 fin.bb, 49  
 floor, 31  
 for : endfor, 41  
 for downto : endfor, 41  
 for step until : endfor, 40  
 for upto : endfor, 41  
 forever: endfor, 41  
 format  
     de composition,  
         voir aussi `init_numbers`, 65  
     de production,  
         voir aussi `%pg` et `%pe`, 65  
 format, 64, 68  
 forsuffixes : endfor, 41  
 fullcircle, 24

## G

gdata, 58  
 gdraw, 56  
 gfill, 58  
 glabel, 58, 67  
 Gneedfr\_, 67  
 Gneedgr\_, 67  
 Gpaths, 66  
 graph.mp, 55  
 graphsup.mp, 55, 57  
 graw, 58  
 green, 10  
 greenpart, 32  
 Gscan\_, 57

## H

halfcircle, 24



- I**
- identity, 33
  - if : else: fi, 41
  - if : elseif : else : fi, 41
  - image, 58
  - init\_numbers, 55, 65, 70
  - intersectionpoint, 20, 34
  - intersectiontimes, 20
  - inverse, 33
  - inversion des commandes
    - draw et fill, 5
- K**
- known, 30, 39
- L**
- label, 35
  - label.top -rt ... -urt ..., 4
  - label.top -rt ... urt ..., 53
  - labeloffset, 4
  - left, 50
  - length, 25, 34
  - lettrage direct, 4
  - lettrage avec LaTeX, 70
  - lien, 50
  - linear, 61, 66
  - linecap, 8, 37
  - linejoin, 8
  - lisse, 57
  - llien, 51
  - log, 61, 66
  - lw, 7
- M**
- macros les plus utilisées, 39
  - macutil, 13
  - macutil.mp, 2
  - magenta, 10
  - men (format -), 60
  - méthode de travail, 1
  - metafun, 5
  - mitered, 8
  - mmp.bat, 71
  - mpto, 70
  - Mreadpath, 65
- N**
- newinternal, 37
  - nombre, 12
  - normale (définition de la -), 23
  - not, 35
- O**
- nullpicture, 35
  - numeric, 30
- P**
- pair, 30
  - paire, 2, 3, 12
  - path, 30
  - pen, 30
  - pic.bb, 46
  - pickup pencircle scaled, 7
  - picture, 30
  - plain.mp, 39
  - plot, 58
  - plume, 12
  - pmaaxes, 68
  - pmaxes, 67
  - pmxticks, 67
  - pmxticlab, 67
  - pmxxticlab, 68
  - pmyticks, 67
  - pmyticlab, 67
  - pmyyticlab, 68
  - point d'inflexion, 19
  - point de contrôle, 16
  - point guide, 15
  - point of, 20, 34
  - pointe de flèche, 9
  - polar, 3
  - préambule, 1, 13
  - primaire, 40
  - prologues, 4
  - pxtick, 64
  - pxytick, 63
  - pytick, 63, 64
- Q**
- quart de cercle, 17
  - quatercircle, 24
- R**
- rbox\_radius, 43, 46
  - rboxes.mp, 43
  - rboxit.bb, 46
  - red, 10
  - redpart, 32
  - reflectedabout, 33

représentation polynomiale  
 des courbes de Bézier, 19  
 resultats.bb, 49  
 retour.bb, 49  
 reverse, 9, 22  
 right, 50  
 rotated, 32  
 rotatedaround, 27, 32  
 round, 31  
 rounded, 8  
 rresultats, 49

**S**

save, 26  
 secondaire, 40  
 secondarydef ... enddef, 40  
 setcoords, 61  
 setlinewidth, 7  
 setrange, 60  
 shifted, 3, 27, 32, 34  
 show, 30, 39  
 sind, 31  
 slanted, 32  
 sqrt, 31  
 squared, 8  
 str, 35  
 string, 30  
 structure d'un fichier de données, 56  
 subpath, 20  
 subpath of, 34  
 substring, 35  
 suffix, 40  
 suffixe, 39

**T**

tangente  
 (définition de la -), 23  
 de direction donnée, 23  
 en un point d'abscisse donnée, 23  
 menée d'un point donné, 24  
 tangente au point guide, 17  
 tangente en un point, 23  
 tangente orientée, 16  
 tension, 18  
 tension, 18  
 tertiaire, 40  
 test.bb, 49

text, 12, 13, 39, 40  
 thelabel, 58  
 thelabel.top -rt ... urt ..., 53  
 tracé de lignes brisée, 5  
 tracé de lignes courbes, 15  
 tracebe, 49  
 tracebf, 48  
 transform, 30  
 transformation, 3, 12  
 transformation EPS-PDF, 2  
 transformed, 33

**U**

u, 2  
 undraw, 5, 6  
 unfill, 5, 6  
 unit vector, 31  
 unités, 2  
 up, 50

**V**

vardef ... enddef, 40  
 variable interne, 6, 12  
 variable tableau, 3  
 verbatimex ... etex, 70  
 visualisation des figures, 2

**W**

whatever, 12, 60  
 white, 10  
 withcolor, 10  
 withpen pencircle scaled, 7, 36  
 \write18, 69, 75

**X**

x[], 3, 30  
 xpart, 11, 31  
 xscaled, 32

**Y**

y[], 3, 30  
 yellow, 10  
 ypart, 11, 31  
 yscaled, 32

**Z**

z[], 3, 30  
 zscaled, 32