Mode d'emploi et manuel de référence d'ePiX

Andrew D Hwang
Department of Math and CS
College of the Holy Cross[1]

Version 1.0.23, Février, 2007

1. Traduction française par le TEXnicien de surface

Table des matières

1 Introduction **3**
1.1 Dépendances logicielles 4
1.2 Installation 6

2 Débuter **9**
2.1 Exécuter `ePiX` 9
2.2 Un fichier d'exemple 11
2.3 Concepts essentiels pour les images 14
2.4 Taille logique et format de l'image 14
2.5 Créer et tracer des objets 15
2.6 La caméra 18
2.7 Attributs de dessin 19
2.8 Typographie 21
2.9 Bases de `C++` 24
2.10 Éléments de dessin de haut niveau 28
2.11 Tracés élémentaires 30

3 Manuel de référence **35**
3.1 Plus loin à propos du `C++` 35
3.2 La caméra 43
3.3 Rognage et cadrage 45
3.4 Attributs 46
3.5 La classe path 48
3.6 Structures des données géométriques 51
3.7 Graphes et domaines 54
3.8 Géométrie non-euclidienne 62
3.9 Animation 64
3.10 Dépannage 65

4 Sujets avancés **68**
4.1 Effacement des objets cachés 68
4.2 Extensions 69

A Liberté des logiciels **73**

B Remerciements **75**

C Glossaire **77**

Chapitre 1

Introduction

ePiX, collection d'utilitaires en ligne de commande pour *nix, trace des figures mathématiquement exactes, des graphes et crée des films avec une syntaxe facile à apprendre. LaTeX et dvips compose le moteur de rendu typographique tandis que ImageMagick est utilisé pour créer les images et les animations *bitmap*. L'interface utilisateur ressemble à celle de LaTeX lui-même : on prépare une courte description d'une scène à l'aide d'un éditeur de texte, on « compile » ensuite le fichier source pour en faire une image. Les formats de sortie par défaut sont eepic (une amélioration en texte simple de l'environnement *picture* de LaTeX), eps, pdf, png et mng.

Les points forts d'ePiX comprennent :

- Qualité de la sortie : ePiX crée des figures mathématiquement exactes et prête à la publication dont l'apparence correspond à ce que produit LaTeX. On peut placer du texte typographié dans une figure aussi facilement que dans un source LaTeX ordinaire.
- Facilité d'utilisation : les objets des images et leurs attribus sont définis par des commandes descriptives simples.
- Flexibilité : dans ePiX, un objet est défini par ses attributs et ses coordonnées cartésiennes ; comme dans LaTeX, l'aspect imprimé est défini à la compilation de la figure. Un figure correctement définie peut être changée radicalement, mais précisément, par quelques altérations du fichier source.
- Puissance et possibilité d'extension : ePiX hérite de la puissance du C++ en tant que langage de programmation ; variables, structures de données, boucles et récursion peuvent être utilisés pour tracer des figures complexes en quelques lignes de source. Du code externe peut être incorporé à une figure ePiX avec une option de la ligne de commande ou avec un Makefile.
- Économie en stockage et transmission : une archive compressée des sources LaTeX et ePiX d'un document contenant de nombreuses figures ne fait généralement que quelques pourcents de la taille du fichier PostScript compressé

correspondant.
- Licence : `ePiX` est un *logiciel libre*. Vous avez le droit d'utiliser le programme pour ce que bon vous semble ; vous avez le droit également d'osculter, de modifier et de redistribuer le code source aussi longtemps que vous ne restreignez pas les droits des autres à faire la même chose. En bref, la licence reprend les conditions dans lesquelles un théorème est publié.

Le rapport entre `ePiX` est un logiciel de dessin est semblable à celui qui exite entre LaTeX et un traitement de texte ; `ePiX` facilite la structuration logique des figures mathématiques. Bien que `ePiX` fasse quelques choix par défaut en matière de style pour rationaliser la création de figures simples, il n'impose aucune restrictions internes au contenu ou à l'aspect d'une figure ; l'esthétique et les décisions pratiques sont laissées à l'utilisateur.

Il vaut mieux lire ce manuel par étapes plutôt qu'en une fois du début à la fin. Si vous êtes :
- un utilisateur potentiel, vous souhaiterez peut-être passer immédiatement à la section « Dépendances logicielles » avant d'investir du temps supplémentaire.
- un nouvel utilisateur, continuez la lecture jusqu'au moment où vous en saurez assez pour faire fonctionner le logiciel ; expérimentez alors à l'aide des fichiers d'exemple en lisant le chapitre 2 ou revenez au manuel en cas de besoin.
- un utilisateur plus avancé, parcourez le manuel à loisir, en commençant peut-être par le chapitre 3.

Dans tous les cas, n'hésitez pas à contacter l'auteur ou à rejoindre la liste de diffusion si vous avez des questions ou des commentaires (bons ou mauvais) à propos du logiciel ou du manuel ou encore si vous avez l'envie et la capacité de vous joindre au développement.

Guidé par l'idée que l'on apprend le plus facilement quand les concepts sont introduits en contexte, ce manuel adopte un style plutôt oral et, à l'occasion, redondant (spécialement entre les parties destinées à des lecteurs se situant à des niveaux différents de familiarité). On a supposé que vous connaissiez LaTeX et les bases de l'algèbre linéaire (la description des points, des vecteurs, des droites et des plans dans un espace à trois dimensions). On introduit le reste, comme par exemple la syntaxe `C++`, en cas de besoin.

1.1 Dépendances logicielles

Si vous utilisez GNU/Linux, un BSD ou Solaris, vous avez certainement (ou pouvez installer) tous les logiciels externes nécessaires à l'utilisation d'`ePiX`. Pour Mac OS X, vous aurez besoin des outils de développement Apple et vous installerez certainement un serveur X et le gestionnaire d'extension `fink` pour bâtir un

environnement *nix complet. Les utilisateurs d'autres systèmes d'exploitations, et notamment Windows, font face à un défi s'ils veulent utiliser `ePiX`, mais pas un défi qu'on ne puisse relever. Le savoir (d'occasion) spécifique à Windows de l'auteur est résumé ci-dessous.

« Sous le capot » un fichier source est converti successivement en `eepic`, `dvi`, PostScript, `pdf` ou `eps` et, si on le désire, `png` ou `mng`. `ePiX` comprend une bibliothèque compilée écrite en `C++`, un fichier d'entête (*header*) `C++` et quatre scripts shell — `epix` , `laps` , `elaps` et `flix` – qui automatisent les conversions dans les différents formats. Chaque script est écrit en GNU `bash`. En conséquence, il y a deux nécessités absolues : un compilateur `C++` (de préférence `g++`) et `bash`. Le script `epix` n'utilise que ces programmes. `ePiX` est d'abord un préprocesseur pour LaTeX mais ne nécessite absolument pas LaTeX en utilisation normale. Toutefois, sans LaTeX et Ghostscript (en particulier `dvips`) vous ne pourrez ni voir ni imprimer les fichiers résultats de `epix` ni lancer `flix` ou `elaps`.

Un éditeur de texte tel que `emacs` ou `vim` qui facilite le formatage du code `C` s'avérera particulièrement utile pour écrire les fichiers sources. Avec `ePiX`, on trouvera un mode `emacs` écrit par Jay Belanger qui permet d'écrire, compiler et voir les figures `ePiX` sans quitter `emacs`.

En présence de LaTeX et Ghostscript, seuls quelques utilitaires standards sont requis pour faire tourner `elaps` le script d'`ePiX` dédiés à la conversion en `eps/pdf`, à savoir `grep`, `sed`, `epstopdf` et `ps2epsi`.

Enfin, `flix` utilise l'utilitaire `convert` d'ImageMagick pour créer les images `png` et pour assembler les `png`s en fichiers `mng` d'images animées. Les programmes `animate` et `display` sont utiles pour visionner les sorties de `flix`.

Outre leur dépendance à des programmes déterminés, les scripts shell d'`ePiX` sont écrits en utilisant des noms de chemins à la Unix. Aussi, le moyen le plus direct d'utiliser `ePiX` est d'installer un environnement de type Unix (*Unix-like*).

Options pour Windows

La version 1.0.4 d'`ePiX` a été codée en Python 2.2 par Andrew Sterian, rendant `ePiX` disponible sur toutes les plateformes disposant de Python sans avoir besoin d'un compilateur `C++` ni de `bash`. Python est un langage de script sous licence GPL et il est disponible avec un installateur Windows et des instructions détaillées. L'option la plus facile pour un utilisateur Windows est probablement d'installer Python 2.2 ou postérieur (si nécessaire) et `Pyepix`. La page d'accueil du projet `Pyepix` est

`http://claymore.engineer.gvsu.edu/~steriana/Python/index.html`

On peut utiliser Cygwin, et en théorie les outils de de Lorie, pour faire tourner `ePiX` sous Windows. Les *packages* Cygwin suivants sont probablement nécessaires

et suffisants :

```
[] bash       [] GhostScript [] sh-utils
[] binutils   [] ImageMagick [] teTeX
[] fileutils  [] make        [] textutils
[] gcc        [] sed
```

Les *packages* `emacs` et `gv` sont hautement désirables mais absolument pas nécessaires. L'auteur a reçu quelques rapports épars de succès avec Cygwin mais n'en connaît pas assez pour fournir une aide substantielle.

1.2 Installation

`ePiX` est distribué par Internet sous forme de source. Les *packages* (stable et développement) sont disponibles à

```
http://math.holycross.edu/~ahwang/current/ePiX.html
```

Dans votre navigateur, `shift`-cliquez sur le lien pour télécharger. La dernière version stable est également présente sur les mirroirs de CTAN, dans le répertoire `graphics`. On y trouve les instructions pour télécharger le répertoire entier ; il n'est pas recommandé de télécharger les fichiers individuellement. (Quelques utilisateurs de Red Hat ont signalé des problèmes de droits sur les fichiers apparus en décompactant les archives de CTAN. Si vous rencontrez une difficulté, essayez de télécharger les sources depuis la page principale du projet.) Décompresser le fichier tar à l'aide des commandes appropriées

```
tar -zxvf epix-x.y.z_complete.tar.gz
tar -jxvf epix-x.y.z_complete.tar.bz2
```

`x.y.z` est le numéro de version ou, si votre `tar` ne fait pas la décompression

```
gunzip -c epix-x.y.z_complete.tar.gz | tar -xvf -
bzcat epix-x.y.z_complete.tar.bz2 | tar -xvf -
```

`cd` dans le répertoire source `epix-x.y.z`. Le fichier `INSTALL` contient les intructions détaillées pour l'installation. Si vous êtes impatient, en voici un bref résumé : `./configure [-options] ; make ; make install`. Lancer `./configure -help` pour voir la liste des options.

Svend Daugård Pedersen a écrit une extension optionnelle qui fournit des améliorations aux systèmes de coordonnées cartésiennes et logarithmiques d'une part et des procédures pour hachurer des polygones ou des régions du plans. **La façon de compiler et installer cette extension a changé avec la version 1.1.** Au

lieu de taper `make contrib ; make ; make install`, on doit passer une option lors de la configuration : `./configure -with-contrib ; make ; make install`. Pour faire appel à l'extension `contrib`, un fichier doit contenir les lignes

```
#include "epix_ext.h"
using namespace ePiX_contrib;
```

Notez qu'il faut désormais inclure explicitement l'entête de `contrib`.

Par défaut, `ePiX` s'installe dans les sous-répertoires de `/usr/local`, si vous désirez l'installer ailleurs, passez à `./configure` l'endroit souhaité à l'aide de l'option `-prefix`. On peut également consulter `POST-INSTALL` pour savoir comment définir la variable `PATH` afin que le shell puisse trouver `ePiX`. Les manuels et document d'exemples, tant les sources que les fichiers ps/pdf compilés, sont installés par défaut dans `$INSTALL/share/doc/epix`.

Je le répète : `ePiX` n'est pas un logiciel autosuffisant mais c'est un ensemble contenant une bibliothèque `C/C++`, un fichier d'entête et des scripts shell. De ce fait son *utilisation normale* dépend d'un compilateur. Le compilateur GNU (`g++`) et les bibliothèques `C++` sont fortement préférables autant parce que je les utilise pour développer `ePiX` que parce qu'on a ainsi accès à des fonctions mathématiques qui ne sont pas spécifiées dans le `ANSI C`. `ePiX` se sert également du shell GNU `bash`. Si vous adaptez `ePiX` pour un autre shell ou un autre environnement, ou si vous réalisez un port pour un autre système d'exploitation, veuillez s'il vous plaît en faire part à l'auteur afin que lien puisse être fait sur votre travail depuis la page du projet et qu'une mention en soit faite dans la documentation.

La commande `make uninstall` supprime les composants d'`ePiX` installés de votre système. Vous devrez être dans le répertoir source et, peut-être, connecté en tant que `root`. (Si vous supprimez les sources et que vous décidiez de désinstaller `ePiX` ensuite il vous faudra décompresser les sources, `./configure` le *package* avec *les mêmes options que lors de l'installation* puis lancer `make uninstall`. Si vous conservez le répertoire source après l'installation, vous n'aurez pas besoin de reconfigurer.)

Autres sources

Un fichier `spec` pour RPM est maintenu par Guido Gonzato. Si vous utilisez une distribution GNU/Linux qui fait appel à RPM 4.x pour gérer les *packages* installés (autrement dit, une version récente de Red Hat ou Mandriva (ex-Mandrake)), vous pouvez compiler et installer `ePiX` avec la commande suivante (exécutée sous `root`)

```
rpm -ta epix_complete.tar.gz
```

Un fichier source `rpm` est disponible sur la page du projet à Holy Cross. Des *packages* existent pour Debian (maintenu par Julian Gilbey), FreeBSD (maintenu par Tsuguru Kato) et Gentoo (maintenu par Olivier Fisette) :

http://www.freshports.org/graphics/epix/
http://packages.gentoo.org/search/?sstring=epix

Le code source le plus récent est disponible par CVS à

http://savannah.nongnu.org/cgi-bin/viewcvs/epix/epix/

L'avenir

Le travail à venir sur ePiX se concentrera sur la version 2, une bibliothèque pleinement 3-D avec de nombreuses capacités complémentaires : modèles plus complexes d'éclairage et d'ombrage, retrait automatique des objets cachés, création modulaire de figure. Faites un tour sur la page du projet pour plus d'information

http://mathcs.holycross.edu/~ahwang/epix/epix2/index.html

Developpement

Il existe deux listes de diffusion pour les deux versions d'ePiX, une pour les questions des utilisateurs, une pour les discussions concernant le développement. Faites un tour sur

http://savannah.nongnu.org/mail/?group=epix

pour vous inscrire.

Chapitre 2

Débuter

Ce chapitre décrit les bases de la création de figure avec `ePiX` et est écrit pour les lecteurs familiers de LaTeX mais complètement novices en `C++`. Aucune connaissance détaillée de `C++` n'est nécessaire pour utiliser `ePiX` si ce n'est un peu de syntaxe (un peu de grammaire du `C` sans pointeurs) que l'on peut facilement absorber par l'exemple. Au fur et à mesure de la lecture, étudiez les fichiers d'exemple distribués avec `ePiX`. Si vous l'avez installé à l'emplacement par défaut, ces exemples sont dans `/usr/local/share/doc/epix`. La façon la plus rapide d'apprendre est de copier les fichiers d'exemple dans un emplacement idoine et d'expérimenter en les modifiant pour voir comment vos modifications agissent sur l'aspect de la figure.

2.1 Exécuter `ePiX`

Comme LaTeX, `ePiX` n'est pas interactif et s'exécute depuis la ligne de commande. Un fichier d'entrée d'`ePiX`, qui a, de préférence, l'extension `.xp` (pour *eXtended Picture*) est un fichier texte de définition de la figure écrit dans un langage de description orienté humain. Un fichier de sortie — `eepic`, `eps`, `pdf`, ou `png` — est inclus directement dans un document LaTeX avec les extensions LaTeXiennes appropriées. Les conversions sont réalisées avec quatre scripts shell — `laps`, `epix`, `elaps` et `flix` — décrits ci-dessous. Chaque script accepte l'option `-help` (ou `-h`) qui le force à décrire les options de la ligne de commande. Les parties entre crochets sont optionnelles.

Sous `X`, on peut simuler un environnement graphique avec `emacs` (et le mode `ePiX` de Jay Belanger), pour éditer et compiler les fichiers, et un visionneur tel que `gv` pour voir les figures compilées. Dans `gv`, sélectionner « Watch file » du menu « State » pour que les images soient automatiquement mises à jour.

Les fragments de code ci-dessous font référence à un hypothétique fichier LaTeX,

`sample.tex`, qui contient une figure compilée depuis un fichier `ePiX` `sample1.xp`. Toutes les extensions mentionnées font partie de toute distribution moderne de LaTeX et des points de suspension prend la place de code LaTeX situé entre les commandes du préambule et le document proprement dit.

laps

Le script `laps` fait la conversion de LaTeX en PostScript et il est indépendant du reste d'`ePiX`. Par défaut, `laps` fait appel à LaTeX et à `dvips`. La commande

```
laps sample[.tex]
```

crée `sample.ps`. L'option `-pdf` crée un fichier PDF en transformant le PostScript à l'aide de `ps2pdf`. On peut utiliser d'autres processeurs de la famille de TeX au lieu de LaTeX : `pslatex`, `pdftex`, etc.

epix

La sortie d'`ePiX` est du `eepic`, l'amélioration due à Conrad Kwok de l'environnement *picture* de LaTeX qui permet les droites de longueur, de pente et d'épaisseur arbitraires. Un fichier `eepic` a généralement une taille de quelques pourcents du fichier `eps` comparable et peut, de plus, être lu (et édité) avec la seule connaissance de LaTeX.

Un fichier `eepic` est inclus directement dans un document LaTeX :

```
\usepackage{epic,eepic,pstricks}
  ...
\begin{figure}[hbt]
  \begin{center}
    \input{sample1.eepic}
    \caption{A compiled \ePiX\ figure.}
    \label{fig:example}
  \end{center}
\end{figure}
```

Un fichier `eepic` pur ne requiert pas `pstricks` mais quelques unes des fonctionalités d'`ePiX` sont implémentées à l'aide des capacités spéciales, `PSTricks` ou de couleur. Pour faire tourner le texte on a besoin de l'extension `rotating`.

Dans un éditeur de texte créez un fichier d'entrée d'`ePiX` `sample1.px` pour la figure puis tapez la commande

```
epix sample1[.xp]
```

On peut compiler plusieurs fichiers à l'aide d'une seule commande p. ex. « `epix sample*.xp` ». Par défaut le fichier de sortie porte le même nom que le fichier d'entrée, ici `sample1.eepic`. Si on ne donne qu'un seul fichier d'entrée, on peut définir le nom du fichier de sortie avec une option de la ligne de commande.

Les extensions `.cc`, `.c`, `.C` et `.cpp` sont reconnues. Toutefois si le fichier de configuration d'`emacs` est adapté convenablement, `emacs` entre directement en mode `ePiX` lorsque l'on ouvre un fichier `.xp` et il insère un préambule lorsque l'on crée un nouveau fichier. Le fichier `FR-POST-INSTALL`[1] contient des instructions détaillées.

Les extensions des fichiers sont optionnelles ; si aucune extension n'est donnée, le script cherchera une correspondance dans le répertoire courant et s'il en trouve plusieurs affichera un avertissement. Le message d'aide donne la liste des extensions reconnues dans l'ordre de préférence.

`elaps`

Les figures LaTeX indépendantes (`eps` ou `pdf` s'opposant à `eepic`) sont utiles dans de nombreuses circonstances : on peut modifier des figures individuelles sans recompiler entièrement le document, la couleur est disponible lors de la visualisation avec `xdvi`, les figures peuvent être utilisées avec PDFLaTeX, etc. L'inconvénient principal est que les fichiers `eps` créés par `dvips` ont tendance à être énormes, des dizaines voire des centaines de fois plus gros que leurs équivalent `eepic` ou `pdf`.

Les commandes

```
elaps sample1.xp
elaps --pdf sample1.xp
```

produisent les fichiers `sample1.eps` et `sample1.pdf` respectivement. Plusieurs fichiers d'entrée ou le nom du fichier de sortie sont donnés sur la ligne de commande comme avec `epix`. On peut donner aussi des extensions LaTeXiennes supplémentaires ou des options `dvips` sur la ligne de commande.

`elaps` peut compiler des fichiers `eepic` (c.-à-d. se comporter comme `eepic2eps` et `eepic2pdf`) même si ces fichiers n'ont pas été produits par `ePiX`.

2.2 Un fichier d'exemple

La figure 2.1 est créée à partir du fichier source commenté `semicirc.xp` présenté ci-dessous dans la figure 2.2.

Les commandes d'`ePiX` sont de trois types : définition (de données et de fonctions), fixation d'attributs et dessin. Comme un document LaTeXien un fichier d'entrée `ePiX` contient un *préambule* qui définit certains aspect de l'apparence de

1. traduction de `POST-INSTALL` par le TdS

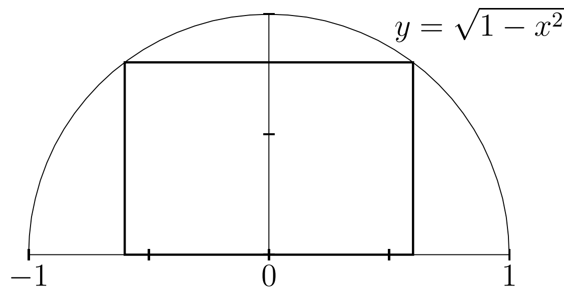


FIGURE 2.1 – Un demi-cercle.

la figure et un *corps* qui contient les commandes proprement dites de création de la figure.

On a utilisé des variables pour donner une structure logique à la figure : la largeur du rectangle et l'emplacement de l'étiquette sont définies dans deux lignes du préambule et la hauteur du rectangle est calculée en fonction de la largeur. La capacité qu'offre `ePiX` de structurer logiquement les figures est un de ses points forts. Dans un fichier aussi court que celui-là on pourrait éventuellement fixer les constantes « en dur » mais l'importance de la structuration s'accroît rapidement avec la taille du fichier d'entrée et il vaut mieux acquérir de bonnes habitudes dès le début. La définition de la fonction est donnée essentiellement comme illustration bien qu'elle permette une organisation légèrement préférable à ce qui suit

```
double width = 0.6, height = sqrt(1-width*width);
double lab_x = 0.5;
P label_here(lab_x, sqrt(1-lab_x*lab_x));
```

Syntaxe des commandes

Les commandes de fixation d'attributs et celles de dessin sont des *fonctions* `C++`, des blocs d'instructions qui peuvent être appelés par leur nom. Comme des fonctions mathématiques, les fonctions `C++` acceptent des *arguments* tels que des entiers (`int`), des nombres réels (`double` pour « nombre décimal à virgule flottante en double précision »), ou `P` (le point d'`ePiX`) et peuvent être codées en dur ou définies symboliquement. `C++` est un langage « typé » ce qui signifie que le compilateur vérifie que les appels de fonction utilisent le bon type et le bon nombre d'arguments et qu'il produit une erreur si aucune fonction correspondante n'est trouvée.

En `C++`, une fonction peut avoir des *arguments avec valeur par défaut* et quand une fonction a des arguments avec valeur par défaut ces arguments peuvent être omis lors de l'appel de la fonction. Les commandes sont décrites ci-dessous en donnant les types et les noms de leurs arguments. Par souci de brièveté, le type

```
#include "epix.h"
using namespace ePiX;                    // semblable à \usepackage

// "double" = réel « décimal » en double precision
double f(double x) { return sqrt(1-x*x); } // c.-à-d. f(x) = \sqrt{1-x^2}

double width = 0.6, height = f(width); // dimensions du rectangle
P label_here(0.5, f(0.5));             // position de l'étiquette du graphique

int main()
{
  unitlength("1in");
  picture(2.5, 1.25);                  // défini la taille imprimée, 2.5 x 1.25in
  bounding_box(P(-1, 0), P(1, 1));     // coins ; représente [-1,1] x [0,1]

  begin();   // ----- le corps de la figure commence ici -----
  h_axis(4);                           // marque 4 intervalles, etc.
  v_axis(2);
  // 2 étiquettes ; descend de 4pt,
  // place sous l'emplacement en coordonnées cartésiennes
  h_axis_labels(2, P(0, -4), b);

  arc(P(0,0), 1, 0, M_PI);             // centre, rayon, angle de début/fin
  bold();                              // seulement les chemins, pas les fontes
  rect(P(-width,0), P(width, height)); // rectangle défini par ses coins

  // déplace l'étiquette à droite 2pt, en haut 4pt,
  // aligne sur point de référence (défaut)
  label(label_here, P(2,4), "$y=\\sqrt{1-x^2}$");
  end();
}
```

FIGURE 2.2 – Le fichier source du demi-cercle

`double` est parfois passé sous silence. Les arguments a valeur par défaut sont donnés entre crochets. Par exemple,

```
line(P pt1, P pt2, [double stretch], [int n]);
```

decrit une commande « `line` » qui accepte deux arguments obligatoires de type `P`, un paramètre réel optionnel `stretch` et un paramètre entier optionnel `n`. Dans un fichier d'entrée, les types des arguments ne sont pas donnés, p. ex.

```
line(P(-2,1), P(1,0), 6.4);
```

Le dernier argument prendrait, dans cet exemple, sa valeur par défaut.

2.3 Concepts essentiels pour les images

Cette section expose les bases typographiques et les données mathématiques servant à déterminer la taille et l'emplacement d'une figure. Le *préambule* d'un fichier d'entrée d'`ePiX` est constitué de tout ce qui se trouve avant la ligne `begin()`, le corps du fichier est la partie située entre les lignes `begin()` et `end()`, ces lignes comprises.

Taille imprimée et emplacement

LaTeX traite le contenu d'un environnement *picture* comme une simple boîte alignée par défaut sur le coin inférieur gauche. Un fichier `ePiX` doit dire à LaTeX quelle sera la largeur de la figure et comment aligner cette « boîte de dessin ». Les commandes

```
picture(2.5, 1.25);
unitlength("1in");
offset(0.25,-0.5);
```

fixe l'unité de longueur LaTeXienne à 1 pouce (2,54 cm), crée un environnement *picture* de 2,5 pouces de largeur sur 1n25 pouces de hauteur puis déplace l'image de 0,25 pouce vers la droite et vers le bas de 0,5 pouce. Les lignes `picture` et `unitlength` sont obligatoires dans le préambule `ePiX`. L'`offset` est optionnel et sa valeur par défaut est $(0, 0)$.

L'argument de la commande `unitlength` est une constante numérique suivie par l'une des unités de longueur valides en LaTeX : `bp` (big point), `cm` (centimètre), `in` (pouce), `mm` (millimètre), `pc` (pica), `pt` (point, unité par défaut), et `sp` (point d'échelle). (Il y a 72 big points par pouce, 12 points par pica, and 65 536 points d'échelle par point.) `ePiX` ne permet pas l'utilisation directe d'unités de longueur différentes pour les horizontales et les verticales ; on est supposé faire les conversions à la main.

Un `offset` (décalage) non nul entraîne l'apparition du contenu de l'image à un endroit où LaTeX ne l'attend pas. Cela peut être utile lorsque le fichier `eepic` est inclus dans un document LaTeX mais demande un ajustement à vue. On prend un risque si l'`offset` n'est pas nul lors de la compilation en EPS ou PDF avec `elaps` puis `dvips` peut rogner la figure selon ses propres règles.

2.4 Taille logique et format de l'image

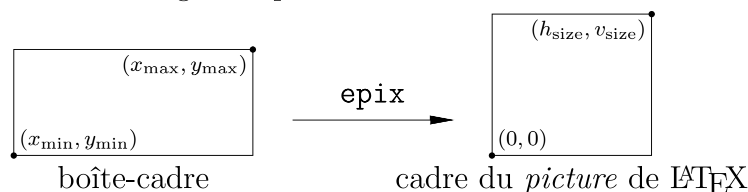
Une figure `ePiX` occupe une boîte-cadre rectangulaire « cartésienne ». Le coin inférieur gauche et le coin supérieur droit de la boîte-cadre sont connus d'`ePiX`

comme (x_min,y_min) et (x_max,y_max) tandis que la largeur et la hauteur sont x_size et y_size. La boîte-cadre est une structure virtuelle, une donnée consultative, ses dimensions ne sont pas directement reliées à la taille de la figure imprimée et les éléments de l'image ne sont pas contraints par défaut à rester dans la boîte-cadre.

La boîte-cadre est déterminée dans le préambule en donnant une paire de coins opposés ; la commande

```
bounding_box(P(-1,0), P(3,2));
```

fixe la boîte-cadre à $[-1, 3] \times [0, 2]$. N'importe laquelle des paires de coins opposés peut être utilisée bien qu'une confusion soit moins possible si l'on donne les coins inférieur gauche et supérieur droit. Une mise à l'échelle affine applique la boîte-cadre au cadre de l'image lorsque le fichier de sortie est écrit :



Le format de la figure est controllé par la taille de la boîte-cadre. Le format est 1:1 si la boîte-cadre et le cadre de l'image sont géométriquement semblable, p. ex. si la boîte et le cadre sont 1,5 fois plus large que haut.

2.5 Créer et tracer des objets

Conceptuellement, ePiX construit un « monde » virtuel en 3 dimensions puis le photographie sur un « écran » à 2 dimensions. L'écran contient la boîte-cadre qui est mise à l'échelle affinement aux dimensions de l'environnement *picture* de LaTeX. Par défaut, l'écran est le plan (x, y) et le monde est représenté en regardant vers le bas le long de l'axe z.

Structures de données géométriques

L'objet le plus simple dans ce monde est le point, représenté par un triplet de nombres réels (décimal en double précision). La fonction P(x1,x2,x3) crée le point de coordonnées (x_1, x_2, x_3). Si l'on ne donne que deux arguments, on a $x_3 = 0$ par défaut. Souvent les points sont nommés afin que l'on puisse les utiliser plusieurs fois dans la figure. Les commandes (essentiellement équivalentes)

```
P pt(x_min, 2*x_min, 4);
P pt = P(x_min, 2*x_min, 4);
```

créent un point nommé `pt` et l'initialise à $(x_{\min}, 2x_{\min}, 4)$ en utilisant la valeur actuelle de $x_{\min}$. Cette valeur est utilisée à chaque fois que `pt` apparait ensuite. En général, on peut utiliser une expression contenant des variables pour allouer une valeur à une structure de donnée (pas uniquement à un point). Toutefois, on ne peut utiliser une variable tant que l'on ne lui a pas alloué une valeur. Par exemple, les commandes précédentes ne fonctionneront pas avant que la boîte-cadre soit déterminée puisque la valeur de $x_{\min}$ est inconnue avant. De plus, changer la valeur d'une variable ne met pas à jour les valeurs des structures de données qui en dépendent.

On peut aussi définir les points en donnant leurs coordonnées polaires, cylindriques ou sphériques ; tous les arguments sont numériques.

```
P pt=polar(r, t);  // (r*Cos(t), r*Sin(t))
P pt=cyl(r, t, z); // (r*Cos(t), r*Sin(t), z)
P pt=sph(r, t, phi);
```

Les arguments de `sph` sont le rayon (distance à l'origine), la longitude (mesurée depuis l'axe (Ox)) et la latitude (mesurée depuis l'équateur). Par défaut, dans `ePiX`, les angles sont mesurés en `radians`. Deux autres « modes angulaires » sont disponibles : `degrees` (degrés) et `revolutions` (tours). Le mode angulaire est déterminé à l'aide d'une commande portant le même nom, p. ex. `degrees()` et toutes les opérations trigonométriques sont affectées par ce choix.

`ePiX` fournit des opérations algébriques sur les triplets, y compris l'addition, la multiplication par un réel, le produit scalaire (*dot product*) et le produit vectoriel (*cross product*) ainsi que quelques autres. Ces opérations sont utilisées pour exprimer les relations entre triplets comme dans

```
P p1(2,1), p2(-3,1);
P q1 = p1-p2, q2 = 2*p1-3*p2, q3=q1*q2;
```

Les points $q_1 = p_1 - p_2$, $q_2 = 2p_1 - 3p_2$ et $q_3 = q_1 \times q_2$ pourraient recevoir des valeurs codées en dur. Toutefois, définir leurs valeurs symboliquement imprègne la figure d'une structure logique, rendant plus facile la lecture du fichier, sa modification, sa maintenance. La base canonique est disponible : `E_1=P(1,0,0)`, etc.

En pratique, les triplets de nombres sont utilisés pour représenter aussi bien des *emplacements* (points) que des *déplacements* (vecteurs). Mathématiquement, les deux concepts sont disctincts, par exemple, en géométrie la somme de deux déplacements a un sens (on obtient un déplacement), la somme d'un déplacement à un emplacement a un sens (on obtient un emplacement) mais on ne peut donner un sens à la somme de deux emplacements. Les opérations algébriques agissent sur les vecteurs et non les points. `ePiX` n'applique pas la distinction entre points et

vecteurs mais il le fera dans la Prochaine Génération [2].

ePiX implémente des structures de données qui représentent les segments, les cercles, les plans et les sphères. Ces structures peuvent être soumises à translation, homothétie et intersection. Un fragment de code illustre les techniques élémentaires :

```
circle C1(P(0,1), P(0,-1), P(0.5,0)); // cercle par 3 points
C1 += P(0.1,0);           // translation du centre
C1 *= 2;                  // double le rayon

sphere S1(P(0,0,0), 1.5); // sphere de rayon 1.5 à l'origine
plane P1(P(0,0,0), E_3);  // plan (x,y)
circle C2 = S1*P1;        // cercle d'intersection
```

L'utilisation de ces structures de données est détaillée dans le chapitre 3.

Dessiner

Les commandes de la section précédente créent des structures de données mais n'écrivent aucune sortie. Chaque type (autre que `P`) est tracé à l'aide d'une syntaxe « orientée objet ». Par exemple, si `C1` est un cercle, la commande `C1.draw()` le dessine. Ce que produit le dessin d'un `plane` (plan) ou d'une `sphere` (sphère) est décrit au chapitre 3. `ePiX` fournit également des commandes de haut niveau pour tracer des polygones, des courbes, des objets composites tels que des flèches et les axes de coordonnées. Les commandes de dessin doivent apparaitre dans le corps du fichier, après `begin()`.

```
line(P p1, P p2);
triangle(P p1, P p2, P p3);   // précise les sommets
rect(P p1, P p2);  // coordonnées de coins opposés du rectangle
quad(P p1, P p2, P p3, P p4); // quadrilatère quelconque

spline(P p1, P p2, P p3); //  splines quadratique/cubique définies
spline(P p1, P p2, P p3, P p4); // par les points de contrôle

arc(P center, radius, t_min, t_max);
ellipse(P ctr, P v1, P v2, [t_min], [t_max], [int n]);

polyline(int n, P p1, ..., P pn);
polygon (int n, P p1, ..., P pn);
```

2. Note du TdS : Ici l'auteur écrit *The Next Generation* et fait ainsi allusion à la seconde série télévisée de Star Treck.

Les arguments de `rect` doivent être dans un plan parallèle à un plan de coordonnées ; les côtés du rectangle sont parallèles aux axes de coordonnées.

Un `arc` a le centre et le rayon comme donnés, il est situé dans un plan parallèle au plan (x, y). Les angles sont mesurés depuis la direction `E1` dans l'unité d'angle courante.

Une commande `ellipse` dessine un arc elliptique avec le centre et les « axes » donnés ; plus précisément, la courbe dessinée est paramètrée par

$$t \mapsto \mathrm{ctr} + (\cos t)v_1 + (\sin t)v_2, \qquad t_{\min} \leqslant t \leqslant t_{\max}.$$

(Remarquez que `ctr` est un emplacement tandis que `v1` et `v2` sont des déplacements.) De même qu'avec `arc` les angles sont mesurés dans l'unité d'angles courante. Si les paramètres fixant les valeurs extrèmes sont omis c'est toute l'ellipse qui est tracée. Le dernier argument (optionnel) précise combien il faut utiliser de point pour tracer l'arc. On peut l'omettre sans danger dans la plupart des situations.

2.6 La caméra

Les étudiants en arts s'entraînent parfois à la perspective en dessinant sur une fenêtre avec un crayon gras, c'est une transformation mathématique appelée *projection conique*. L'application qu'`ePiX` utilise par défaut pour projeter le monde sur l'écran lui ressemble. Imaginez-vous au *point de vue* et doté d'un regard aux rayons X qui rend pour vous tous les objets transparents. Quelque part devant vous se trouve l'écran, un plan. La *cible* est le le point de l'écran pied de la perpendiculaire au plan passant par le point de vue.

Trois vecteurs unité mutuellement perpendiculaires ont leur origine à la cible : mer, ciel et œil. Le vecteur mer pointe vers la droite, le ciel pointe vers le haut et l'œil suit la direction qui relie la cible au point de vue. La boite-cadre de la figure est définie dans le repère (cartésien) mer-ciel de l'écran.

Soit un point p devant le point de vue, nous voulons détermine l'endroit de l'écran où il se projette. Traçons une droite joignant p et le point de vue, cette droite perse le plan de l'écran une fois et une seule, c'est en ce point d'intersection que nous traçons p sur l'écran.

Au début de la figure, la caméra[3] est placée sur l'axe z à une très grande distance de l'origine. La vue qui en résulte, essentiellement une projection parallèle à l'axe z convient aux figures planes. On manipule la caméra avec une syntaxe orientée objet :

```
camera.at(P posn);           // fixe le point de vue à posn
```

3. Note du TdS : Le mot anglais *camera* signifie « appareil photographique » mais j'ai préféré « caméra » qui a pour seul avantage celui de la brièveté.

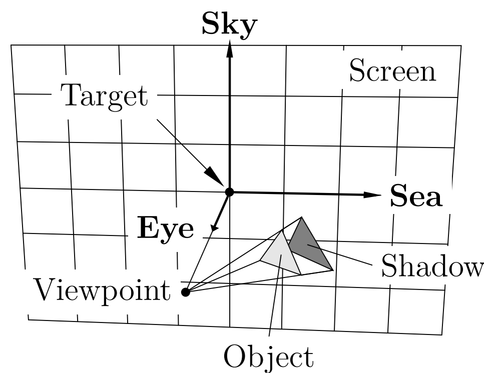


Figure 2.3 – Projection d'un point.

```
camera.look_at(P targ);     // fixe la cible
camera.range(double dist);
                  // fixe la cible, déplace le point de vue
camera.focus(double dist);
                  // fixe le point de vue, déplace la cible
camera.rotate_sea(double angle); // rotation autour d'un axe
```

Ces commandes doivent être placées dans le corps de la figure.

2.7 Attributs de dessin

`ePiX` crée des dessins au trait et non des mappes de pixels. Dans un fichier `eepic` les objets ont soit du type point (glyphes LaTeX, boites de texte) soit du type chemin (tout le reste). L'apparence des objets du type chemin dépend du style de droite, de sa largeur, de son remplissage et de sa couleur. Chaque attribut est une déclaration, dont l'effet se fait sentir jusqu'à ce qu'elle soit supplantée ou jusqu'à la fin du fichier. Au début du fichier, les chemins sont tracés en ligne pleine, noire, non remplie et avec l'épaisseur `thinlines` (environ 0,4 pt). Les commandes de fixation des attributs doivent apparaitre dans le corps de la figure.

Largeur de chemin et style

Les largeurs normales de chemin sont `plain` (`thinlines`) et `bold` (`thicklines`). D'autres épaisseurs sont disponibles à l'aide de la commande `pen()`. L'argument est un nombre, interprété comme une longueur en `pt`, ou un nombre suivi d'une unité de longueur LaTeXienne en deux lettres comme dans la commande `unitlength()`. Les journaux déconseillent les épaisseurs de ligne inférieurs à environ 0,5 point et,

par ailleurs, une multitude d'épaisseurs fixées par l'auteur d'une figure tend à la faire paraître chargée et *ad hoc*. Si possible, il vaut mieux utiliser les épaisseurs normales.

```
  plain();         // thinlines, à peu près comme pen(0.4);
  bold();          // thicklines,                  pen(0.8);
  pen("0.02in");  // fixe l'épaisseur des chemins à 0.02 in
```

Le style de chemin est soit `solid` (*trait plein*) soit `dashed` (*tireté*) soit encore `dotted` (*pointillé*). Le style est fixé par une commande portant le même nom, par exemple `dashed()`. Chaque objet de type chemin est, dans `ePiX`, déssiné en plaçant de petits points à chaque point (emplacement) du chemin. Lorsque le style est `dashed`, des segments de droite sont tracés partiellement d'un sommet à ses voisins.

On peut ajuster quelques paramètres à la main : la distance en page entre les points consécutifs d'un chemin (pour les polygones : triangles, quadrilatères, etc.), la « densité de tirets » (pourcentage d'un chemin tireté rempli par les tirets) et la taille des points :

```
  dash_fill(0.7);
// les tirets emplissent 70% de l'espace entre deux points
  dash_length(6);
// les points du chemin sont séparés de 6pt
  dot_sep(8);      // points séparés de 8pt
  dot_size(2);     // point de 2pt de diamètre
```

Couleur et remplissage

Un fichier de sortie d'`ePiX` rend les couleurs à l'aide du style `pstcol`, un mélange de styles de `colors` et `PSTricks`. On dispose de trois modèles de couleurs : `rgb` – pour *red* (rouge), *green* (vert) et *blue* (bleu) –, `cmyk` – cyan, magenta, *yellow* (jaune) et *black* (noir) – et nommé (mais seulement les couleurs primaires[4]). Une définition de couleur `rgb` se fait avec trois nombres compris entre 0 (pas de couleur) et 1 (saturation complète), chacun d'entre eux représentant la densité d'une des trois couleurs primaires – rouge, vert et bleu respectivement. Une définition de couleur en `cymk` est donnée de même par quatre densités. Enfin, une couleur primaire – *red* (rouge), *green* (vert) ou *blue* (bleu) ; cyan, magenta ou *yellow* (jaune) ; *black* (noir) et *white* (blanc) – peut être donnée par son nom, avec une densité optionnelle.

4. Note du TdS : avec leurs noms anglais, bien entendu

```
rgb(1, 0, 1);  // magenta
cmyk(0,1,0,0); // même chose
magenta();     // troisième méthode
rgb(1, 0.7, 0.7); // rouge clair
rgb(0.4, 0, 0);   // rouge sombre
red(0.4);         // même chose
```

Comme tous les paramètres, les densités de couleur peuvent dépendre de variables. Les valeurs en dehors de l'intervalle [0, 1] sont « raccourcies » ; par exemple, `rgb(1.4,-0.05,2)` est aussi du magenta.

La commande `fill()` fait que les chemins fermés sont remplis en gris, à l'aide de `specials` PostScript. La profondeur de gris va de 0 (blanc) à 1 (noir) et a 0,3 pour valeur par défaut. La commande `gray(0.4)` fixe la profondeur à 0,4. Le remplissage est désactivé par la commande `fill(false)`. L'ordre d'apparition des objets remplis dans le fichier source est significatif parce que `ePiX` écrit sa sortie dans le même ordre et que PostScript ne gère pas la transparence. L'utilisation de couches, l'effacement des objets cachés et les dégradés de couleurs sont traités dans les chapitres 3 et 4.

2.8 Typographie

Dans un fichier `ePiX`, les points sont marqués avec des glyphes LaTeXiens (marques *markers*) ou des boites de texte (étiquettes *labels*). Une marque occupe une boite de taille nulle et est positionné à un emplacement cartésien déterminé. Lorsque la taille ou le format de l'image est ajusté, le corps de la police reste le même. Afin de garder l'alignement correct des étiquettes pour toutes sortes de tailles on attache un point d'alignement invariant par homothétie (agrandissement, réduction) à chaque étiquette et on utilise les coordonnées cartésiennes pour placer ce point d'alignement. Ce dernier est controllé par un déplacement donné en vrai-point et une option facultative d'alignement dans le style LaTeXien.

Marques

Les marques d'`ePiX` sont appelées par leur nom :

```
spot(P pt);  dot(P pt);   ddot(P pt);
             box(P pt);   bbox(P pt);
ring(P pt);  circ(P pt);
```

Les marques `spot` et `(d)dot` sont des points pleins qu'on ne peut pas colorier ; `box` et `bbox` sont des carrés pleins que l'on peut colorier. Une marque `ring` est coloriable

◦ CIRC	● SPOT	○ RING	• DOT	· DDOT
+ PLUS	⊕ OPLUS	× TIMES	⊗ OTIMES	
◇ DIAMOND	△ UP	▽ DOWN	▪ BOX	· BBOX

TABLE 2.1 – Les types de `marques` d'`ePiX`.

et transparent tandis que `circ` est incoloriable et opaque. Les marques situées dans la même colonne sont de même taille et, dans chaque colonne, le diamètre est 1,5 fois le diamètre de la suivante. Le diamètre d'une `dot` – et donc de toutes les marques situées dessous – est fixé à `dot_size(diam)` dont l'argument, qui vaut 3 par défaut, est un nombre de `pt`. Les glyphes présentés dans le tableau 2.1 sont disponibles grâce à la commande

```
marker(P pt, <MARKER TYPE>);
```

On dispose aussi de ces types de marques lorsque l'on représente des données à partir d'un fichier.

Étiquettes

Une *étiquette* est une boite typographique. Comme une boite LaTeXienne occupe un rectangle sur la page imprimée, on n'a pas assez d'information avec une seule position pour placer une étiquette dans une figure ; on a besoin en plus d'un point d'alignement. Par défaut, le point d'alignement d'une boite de texte est le point de référence : le point d'intersection de l'arête gauche avec la ligne de base, celui que LaTeX utilise pour placer la boite sur la page : $y = f(x)$ On peut déplacer un point d'alignement à la main :

```
label(P(3,2), P(2,-1), "$\\rho=\\sin \\theta$");
```

compose l'équation $\rho = \sin\theta$ et place la boite de texte produite au point de coordonnées $(3, 2)$ mais la déplace vers la droite de `2pt` et vers le bas de `1pt`. Remarquez que `C++` traite « \\ » comme un charactère d'échapement et qu'il faut donc doubler la barre oblique inverse dans le source pour en obtenir un seul dans la sortie. Le commande d'étiquettes sont :

```
label(P posn, P offset, <label text>);
label(P posn, <label text>);
label(P posn, P offset, <label text>, align);
```

La première commande imprime l'étiquette avec son point de référence LaTeXien aux coordonnées cartésiennes `posn` en le déplaçant (sur la page) des nombres de vrais points donnés sous formes de coordonnées `P offset`. Des déplacements de 2, 4, 6, 12 ou 18 points conviennent bien à une police de corps 12.

La seconde commande imprime le texte de l'étiquette dans une boite LaTeXienne centrée au point `posn`. Bien que ce soit peut-être la manière la plus évidente de placer une étiquette, ce ne peut être toujours la méthode correcte puisque les étiquettes marquent souvent un objet géométrique qui ne devrait pas être recouvert par son étiquette.

Dans la troisième forme de la commande, l'option `align` peut prendre une valeur – ou une paire convenable de valeurs – parmi `t`, `b`, `r` ou `l` – pour *top* (sommet), *bottom* (bas), *right* (droite), *left* (gauche) – ou le seul `c` (*center*, centre). Comme pour les `offsetts`, ces options d'alignement spécifient une position de l'étiquette *relativement à l'emplacement cartésien de p* et donc fonctionnent *à l'opposé* de la manière dont elles fonctionnent sous LaTeX. Par exemple, l'option d'alignement `br` place l'étiquette dessous et à droite du point *p*.

[l]•[r]   [t] [b]   [tl] [tr] [bl] [br]

Chaque commande d'étiquette possède une version « masque (*mask*) » correspondante qui place un rectangle blanc opaque sous le texte de l'étiquette. Il est utile de masquer lorsque le texte de l'étiquette apparait dans une partie de la figure déjà chargée. Un document contenant un fichier `eepic` faisant appel à la commande `masklabel` doit utiliser l'extension `pstcol`[5].

On peut pivoter une étiquette, l'angle est donné dans l'unité d'angle courant par la commande `label_angle(theta)` . Une rotation de 90° imprime des étiquettes convenant à un axe vertical. Si un fichier `eepic` contient des étiquettes pivotées, le document qui le contient doit charger l'extension `rotating`. (`elaps` inclut automatiquement toutes les extensions LaTeXiennes dont pourrait avoir besoin un fichier `eepic`.)

En construisant et positionnant une étiquette, gardez les points suivants à l'esprit :

– Les déplacements d'alignement sont donnés en `pt` (c.-à-d. en coordonnées de la page imprimée) et non dans les unités cartésiennes parce que le point d'alignement ne devrait pas dépendre de la taille logique ou imprimée de la figure ;
– Le texte de l'étiquette est entouré de *doubles quotes* (le caractère simple `"`) et contient le code LaTeXien pour composer l'étiquette. Les barres obliques inverses sont doublées.

5. Note du TdS : Il n'est plus nécessaire désormais (j'écris le 2007-02-24) de charger `pstcol` si on a chargé `pst-all`.

Polices et corps des caractères

Par défaut, la police dans une figure `ePiX` est celle du document qui la contient. Le corps de la police et son œil sont changés par des commandes de « déclaration de style » comme

```
font_size("Large");
font_face("sc");
```

L'argument de `font_size` est une taille LaTeXienne valide dont la valeur par défaut est `normalsize` ; `font_face` accepte une chaine de deux caractères qui concaténées à la chaine « `text` » donnent une commande LaTeXienne de déclaration de fonte, comme par exemple `textsc` ci-dessus.

Un controle plus fin, sur la police et la taille pour une seule étiquette, est obtenu en codant en dur les commandes LaTeXiennes nécessaires dans le texte de l'étiquette.

Marques pour étiquettes

Les marques élémentaires – `circ`, `ring`, `spot`, `(d)dot` et `(b)box` – accepte des arguments d'étiquette. Par exemple :

```
dot(P(0,0), P(4,2), "The Origin", tr);
```

positionne et étiquette un point aux coordonnées $(0, 0)$.

2.9 Bases de C++

Un fichier source `ePiX` est un programme `C++`. Si vous avec modifié et compilé avec succès l'un des fichiers d'exemple, vous connaissez assez de `C++` pour utiliser `ePiX`. D'après l'expérience de l'auteur, la grammaire `C` suffit pour la plupart des applications. Une excellente introduction à la définition de fonctions et de variables, aux instructions de controle et à la structure générale des programmes est le le livre de Kernighan et Ritchie *Le langage C* [2] (édition anglaise [1]).

Le mode `emacs` pour `ePiX` de Jay Belanger insère un fichier de gabarit (*template*) lorsque l'on ouvre un tampon vide d'extension `xp`. Cette section explique à quoi sert ce gabarit. Quelques remarques supplémentaires devraient vous permettre d'éviter des chausse-trapes de la syntaxe élémentaire.

Format de fichier

Un fichier `C++` est constitué « d'instructions » analogue aux phrases ordinaires. Les types communs comprennent des *déclarations* – qui « enregistrent » une fonction, une variable ou le nom d'un type auprès du compilateur –, des *définitions* – qui assigne un sens à un nom déclaré – et des *appels de fonction* – qui font qu'une fonction nommée est exécutée. La plupart des instructions d'un fichier d'entrée d'`ePiX` sont des appels de fonction – des « commandes `ePiX` ». Les déclarations explicites sont assez rares puisque une définition sert à déclarer tout nom nouveau qu'elle contient.

Toute instruction est terminée par un point-virgule et, par convention, un fichier ne contient qu'une instruction par ligne (si possible). Le compilateur ignore presque tous les blancs – espaces, tabulations et sauts de ligne – qu'on peut donc utiliser à loisir pour rendre le fichier lisible. D'autres marques de ponctuation – points, virgules, deux-points, point-virgules, parenthèses, accolades et *quotes* – dirigent l'analyse lexicale du fichier et doivent respecter très strictement la grammaire.

Un fichier `ePiX` commence toujours avec les lignes

```
#include "epix.h"
// N.B. directive pour le pre-processor, pas de point-virgule
using namespace ePiX;
```

La première ligne est analogue à une commande `\usepackage` pour LaTeX : elle inclut le contenu du fichier, ce faisant importe les noms des commandes fournies par `ePiX`. Pour éviter des conflits de noms, les commandes d'`ePiX` sont enfermées dans un « espace de noms ». Par exemple, la commande `label` est en fait connue par le compilateur comme `ePiX::label`. La deuxième ligne de l'extrait ci-dessus dit au compilateur d'ajouter tacitement ce préfixe.

Variables et fonctions

Les définitions de variables et de fonctions jouent le même rôle dans une figure que les définitions de macro dans un document LaTeXien : grouper et organiser l'information dont la figure dépend. Une variable est définie par la donnée de son type, de son nom et de sa valeur initiale. Les types de données de loin les plus communs dans `ePiX` sont `double` (décimal en double précision), `P` et `int`. Le nom d'une variable ne doit contenir que des lettres (y compris le souligné `_`) et des chiffres et doit commencer avec une lettre :

```
my_var, my_var2,  _MY_var, __, aLongVariableName;  // valide
my-var,    2var, v@riable, $x, ${MY_VARIABLE}; // non valide
```

Les noms de variables sont sensibles à la casse (distinction des minuscules et des majuscules). Il existe de nombreuses conventions quant à la signification de l'usage des majuscules. En général, choisissez des noms descriptifs mais maniables et évitez les noms commençant par un souligné – à moins que vous ne sachiez ce que vous faites.

Une fonction accepte des « arguments » et « retourne une valeur ». Pour définir une fonction en C++ vous devez préciser le type de retour, le nom de la fonction, le type de chaque argument puis l'algorithme par lequel la fonction calcule la valeur de retour à partir des données. Le bloc de code

```
double f(double x)
{
  return sqrt(1-x*x);
}
```

définit la fonction f, renvoyant une valeur `double` et acceptant un argument `double`, telle que $f(x) = \sqrt{1 - x^2}$. De nombreux fichiers d'exemples et fichiers sources (particulièrement `functions.cc`) présentent de nombreux autres exemples. On devrait présenter comme ci-dessus la définition d'une fonction pour des raisons de lisibilité.

Commentaires

C++ connait deux sortes de commentaires. Les commentaires à la C qui peuvent s'étendre sur plusieurs lignes sont délimités par les chaines `/*` et `*/`. Les commentaires d'une seule ligne commencent par `//` qui joue un rôle semblable à celui de `%` en LaTeX. Un commentaire d'une seule ligne peut apparaitre au milieu d'un commentaire multiligne mais pas un commentaire à la C ; le compilateur confondrait le premier `*/` rencontré avec la fin du commentaire multiligne courant.

Exécution du programme

Dans un programme C++, toutes les « actions » ont lieu dans la fonction spéciale `main`. L'exécution d'un programme C++ compilé est vue par le système d'exploitation comme l'appel de la fonction `main` du programme. La valeur de retour (un `int` c.-à-d. un entier) est le statut de sortie du programme. Dans un fichier ePiX, l'action de `main` consiste d'habitude à fixer la taille logique de la figure et sa taille imprimée puis de construire et dessiner la figure en changeant les attributs au moment voulu. La sortie proprement dite d'ePiX commence avec `begin()` et se finit avec `end()`. Les instructions entre ces deux commandes constituent le *corps* de la figure.

En C++, on ne peut pas définir une fonction à l'intérieur de la définition d'une autre. Aussi on peut définir des variables à l'intérieur de `main` mais pas des fonctions.

```
#include "epix.h"
using namespace ePiX;

int main()
{
  using std::cout;      // fonction de sortie de C++
  cout << "\\begin{figure}[hbt]\n";
  unitlength(...);    // picture, bounding_box, etc.
  begin();
  < ... commandes ePiX ... >
  end();
  cout << "\\caption{A \\LaTeX\\ figure.}\n" // N.B. pas de ";"
       << "\\label{fig:example}\n"          // la ligne continue.
       << "\\end{figure}\n%%%%\n";          // formatage LaTeXien
} // End of main()
```

FIGURE 2.4 – Créer une figure autonome avec `ePiX`.

Sortie brute

On peut faire écrire du texte plus ou moins *verbatim* dans le fichier de sortie. Une barre oblique inverse est produite par deux barres obliques inverses dans le fichier d'entrée. Certaines lettres ont une signification spéciales quand elles sont échappées (protégées) par une barre oblique inverse y compris "`\n`" (*newline* saut de ligne) and "`\t`" (TAB : tabulation). Contrairement à LaTeX, `C++` n'exige pas un espace pour séparer une séquence échappée du texte suivant, la chaine « `\textwidth` » est lue « `TABextwidth` » par le compilateur.

En guise d'application, on peut produire un environnement `figure` LaTeXien, avec titre (*caption*) et référence (*label*) avec un fichier `ePiX`. Les sauts de ligne doivent être ajoutés explicitement et toutes les instructions d'impression doivent apparaitre à l'intérieur de l'appel à la fonction `main()`, voir la figure 2.4.

Sauts conditionnels et boucles

Le comportement d'un algorithme dépend d'habitude d'un certain état interne. Une instruction conditionnelle fait que des blocs de code sont exécutés ou non suivant un certain critère. Une *boucle* répète un bloc de code, généralement en changeant les valeurs de quelques variables d'une manière prévisible de sorte que le programme sort de la boucle après un nombre fini de répétition. La figure 2.5 illustre tant les instructions conditionnelles que les boucles avec l'algorithme d'Euclide de détermination du plus grand commun diviseur. Trois notations demandent un éclaircissement : `j%i` signifie $j \pmod{i}$, `||` est le « ou » logique, `==` est le « test d'égalité » – un seul `=` est l'opérateur d'affectation.

```
int gcd (unsigned int i, unsigned int j)
{
    int temp=i;
    if (i==0 || j==0)
        return i+j;    // définit gcd(k,0) = k

    else {
        if (j < i)     // les échange
            {
                temp = j;
                j=i;
                i=temp;
            }
        // le travail se fait ici
        while (0 != (temp = j%i)) // affecte temp, teste la nullité
            {
                j=i;
                i=temp;
            }
        return i;
    }
}
```

FIGURE 2.5 – Algorithme d'Euclide de la division en C++.

Pour des détails sur les intructions conditionnelles et pour des techniques plus avancées de programmation C++ voyez un manuel ou un cours en ligne.

2.10 Éléments de dessin de haut niveau

ePiX définit des capacités de dessin de haut niveau : flèches, axes de coordonnées et étiquettes d'axes, graphiques en coordonnées polaires, représentation graphique de données lues dans des fichiers, champs de vecteurs, résolution d'équations différentielles et géométrie non-euclidienne.

Flèches

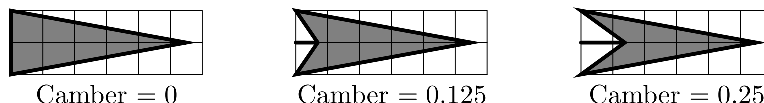
Une flèche (*arrow*) droite est déterminée par sa queue et sa pointe. Des flèches splines quadratiques ou cubiques sont décrites par leurs points de controle. Un argument facultatif final précise l'échelle de la tête de flèche.

```
arrow(P tail, P tip, [double scale]);
arrow(P p1, P p2, P p3, [double scale]);
arrow(P p1, P p2, P p3, P p4, [double scale]);
```

```
  dart (P p1, P p2);  // same as arrow(p1, p2, 0.5);
  aarrow(P p1, P p2); // double-headed arrow <--->
```

En tant que dessin, une flèche consiste en un segment de courbe (le « bois ») surmonté d'une tête de flèche. Vue de profil, la largeur d'une tête de flèche est `3pt` et sa hauteur est 5,5 fois sa largeur. La hauteur effectivement imprimée dépend de l'orientation de la flèche par rapport à la caméra. Par défaut, la tête est un triangle vide. Le remplissage, la forme et la taille peuvent être ajustées avec les déclarations :

```
  arrow_fill(dens); // densité de gris, entre 0 et 1
  arrow_width(w);   // largeur en pt, 3 par défault
  arrow_ratio(r);   // rapport hauteur/largeur, 5,5 par défault
  arrow_camber(c);
// renfoncement de la base comme fraction de la hauteur, 0 par défaut
```



Les têtes de flèche vides sont tracées dans la couleur de chemin courante ; les têtes remplies sont dessinées en noir. On peut utiliser des commandes `PSTricks` pour colorié les flèches.

Axes de coordonnées et étiquettes

Un axe de coordonnées est un segment de droite donné par ses extrémités et un nombre précisé de coches régulièrement espacées :

```
  h_axis(p1, p2, n);    // n intervalles (n+1 coches)
  v_axis(p1, p2, n);
```

Le style des coches est adapté au type d'axe. Si les extrémités ne sont pas précisées elles prennent les valeurs par défaut $p_1 = (x_{\mathrm{min}}, 0)$ et $p_2 = (x_{\mathrm{max}}, 0)$ pour un axe horizontal ou $p_1 = (0, y_{\mathrm{min}})$ et $p_2 = (0, y_{\mathrm{max}})$ pour un axe vertical. Si la boite-cadre a un côté entier alors l'omission du nombre de coches fait qu'une coche est dessinée par unité.

Pour un axe horizontal on crée les étiquettes avec

```
  h_axis_labels(p1, p2, n, P(u,v));
  h_axis_labels(P p1, P p2, int n, P offset, [alignment]);
```

Cela place $n+1$ étiquettes régulièrement espacées sur le segment d'extrémité `p1` et `p2`. Les étiquettes sont crées automatiquement pour correspondre à leur emplacement horizontal. De même que pour des étiquettes ordinaires, le déplacement (`offset()`) est en `pt` et l'option facultative d'alignement LaTeXoïde place les étiquettes en utilisant leurs coins plutôt que leur point de référence. Les étiquettes d'un axe vertical sont crées de manière évidente.

De même que pour les axes de coordonnées, le point initial et final peuvent être omis d'une commande `axis_label` avec les mêmes valeurs par défaut. Toutefois le déplacement (`offset`) et le nombre d'étiquettes doivent toujours être spécifiés.

Quadrillage

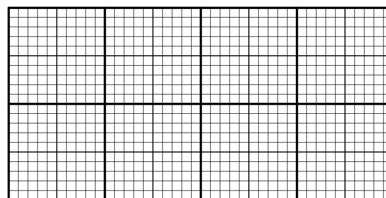
Un quadrillage cartésien emplit un rectangle de coordonnées et a un nombre déterminé de lignes dans chaque direction. Un quadrillage polaire a un rayon, un nombre d'anneaux et un nombre de secteurs angulaires déterminés.

```
grid(n1, n2);          // remplit la boite-cadrex
grid(p1, p2, n1, n2); // remplit la boite de coins p1, p2
grid(p1, p2, mesh(n1, n2), mesh(m1,m2));
polar_grid(r, n1, n2);
```

Chacune des commandes dessine un quadrillage `n1` par `n2`. La troisième utilise un maillage (*mesh*) ce qui n'est utile que si l'objectif de la caméra ne projette pas les droites des objets de l'espace sur des droites du plan.

On peut créer un quadrillage semblable à celui du papier millimétré en superposant des quadrillages :

```
pen(0.25);
grid(10*x_size, 10*y_size);
pen(0.5);
grid(2*x_size, 2*y_size);
pen(1);
grid(x_size, y_size);
```



L'extension `contrib` de S. D. Pedersen fournit un perfectionnement des graphiques cartésiens. Si cette extension est installée, la documentation de `contrib` sera dans `$INSTALL/share/doc/epix`.

2.11 Tracés élémentaires

Comme `eepic.sty` peut tracer des droites de longueurs et de pentes arbitraires, les courbes peuvent être approximées en reliant des points par des segments. `ePiX`

restitue de cette manière les courbes, les polygones et les graphes[6] de fonction.

Pour l'instant, « fonction » signifie « fonction d'une variable » – précisément une fonction à variable `double` et à valeur `double`. Un graphe de fonction dépend du domaine et du nombre de points à utiliser. Chaque commande

```
plot(f, t_min, t_max, n);
polarplot(f, t_min, t_max, n);
shadeplot(f, t_min, t_max, n);
```

trace le graphe de la fonction `f` sur l'intervalle `[t_min, t_max]` en divisant cet intervalle en `n` sous-intervalles d'amplitudes égales. La première donne un graphe dans un repère cartésien, la deuxième dans un repère polaire dont les bornes sont données dans l'unité d'angles courante. Si deux fonction sont passées à `shadeplot` la région qu'elles délimitent est colorée.

Représentation de données

On peut créer des fichiers de données numériques, les manipuler, les analyser, les représenter graphiquement (chemins, nuages de points et histogrammes), les lire et les écrire. Un fichier de données doit contenir un ou plusieurs nombres en point flottant[7] avec le même nombre d'entrées par ligne. Tout ce qui apparait sur une ligne après le caractère `%` de commentaire de LaTeX est considéré comme un commentaire.

`ePiX` fournit deux types de commande `plot` (tracer) pour les fichiers de données. La première facilite la représentation graphique de colonnes choisies, la deuxième simplifie la représentation graphique des deux première colonnes avec une ou deux échelles logarithmique. On peut utiliser l'une ou l'autre forme pour tracé dans une échelle logarithmique des colonnes sélectionnées.

Supposons que `mydata.dat` contienne au moins 5 colonnes. Les commandes

```
plot("mydata.dat", DOWN);
plot("mydata.dat", PLUS, log_lin);
plot("mydata.dat", SPOT, 2, 4, 5, sph);
```

représente (pour la première) graphiquement les deux premières colonnes de `mydata.dat` en plaçant un « $\bigtriangledown$ » à chaque point ; marque (la 2e) les points définis par les deux premières colonnes avec un « + » dans un repère semi-logarithmique, l'axe des abscisses portant l'échelle logarithmique ; extraie (la troisième) les 2e, 4e et 5e colonnes du fichier et les traite comme donnant des coordonnées sphériques de points qui sont représentés par un « • ».

6. Note du TdS : Je sais bien, pour avoir enseigné dans le secondaire, qu'il faudrait dire « représentation graphique » mais, bon…

7. Note du TdS : Il faut utiliser le point décimal et non la virgule.

Les commandes générales, avec leurs arguments optionnels entre crochets, lisent les nombres de deux ou trois colonnes d'un fichier spécifié, les passent comme arguments à une fonction `F` à valeur `P` et tracent les points ainsi déterminés :

```
plot("filename", STYLE, [i_1], [i_2], [i_3], [F]);
plot("filename", STYLE, F, [i_1], [i_2], [i_3]);
```

Le premier argument est le nom du fichier de données. Le `STYLE` peut être `PATH`, qui relie les points dans leur ordre d'apparition, ou n'importe quel type de marques de la table 2.1. Les entiers i_k spécifient les colonnes d'où doivent être extraites les données, par défaut il s'agit de la 1re, de la 2e colonne et de zéro (une colonne de zéros). Si le « système de coordonnées » `F` est omis dans la première commande, il prend la valeur par défaut du constructeur de point cartésien. La fonction `F` est obligatoire dans la deuxième forme ; des choix utiles comprennent `log_log`, `log_lin` et `lin_log` qui représentent les coordonnées correspondantes sur une échelle logarithmique.

data_file

Pour des analyse plus élaborées, la classe `data_file` (fichier_de_données) fournit une interface qui permet de voir un fichier comme une liste de colonnes. Il y a deux manières principales de créer un `data_file` : par lecture dans un fichier externe ou par création de données (jusqu'à trois colonnes) à l'aide de fonctions à valeur `double`. Dans les constructeurs ci-dessous, chaque fonction `fi` est une fonction d'une variable à valeur `double`.

```
data_file DF("my_data"); // lit les données dans un fichier
data_file DF(f1, t_min, t_max, num_pts); // valeurs de f1
data_file DF(f1, f2, t_min, t_max, num_pts);
data_file DF(f1, f2, f3, t_min, t_max, num_pts);
data_file DF(3);  // crée un data_file vide de 3 colonnes
DF.read("file1").read("file2"); // lit deux fichiers
```

Une fois qu'un `data_file` existe, ses colonnes peuvent être transformées par l'application d'une fonction définie par l'utilisateur ; on peut en calculer la moyenne, les corréler, les extraire (pour utilisation par un autre code), en tracer le nuage de points et les écrire sur un fichier à un endroit spécifié. Ci-dessous, la fonction `f` est une fonction d'une variable à valeur `double` et la fonction `F` est une fonction de deux ou trois variables à valeur `P` dont les composants sont réécrits dans les colonnes sélectionnées.

```
DF.transform(f, 3);    // applique f à col3
DF.transform(F, 2, 4); // applique F à col2, col4
```

```
double t(DF.dot(3,2)); // t = produit  scalaire de col2, col3
double av(DF.avg(2));  // av = moyenne de col2
double v(DF.var(1));   // variance de col1
double cor(DF.covar(1, 3)); // covariance de col1, col3
DF.regression(1,3);    // trace la droite de régression
```

Le nuage de point représentant un `data_file` est obtenu avec la syntaxe décrite ci-dessus pour les fichiers de données. La sortie est rognée aux dimensions de la boite-cadre.

```
DF.plot(STYLE, [i1], [i2], [i3], [F]);
DF.plot(STYLE, F, [i1], [i2], [i3]);
```

Un `data_file` peut être écrit sur le disque comme un fichier de données brutes ou dans un format spécifié. Ci-dessous, `fmt` représent une fonction de deux variables à valeur `string` (chaine de caractères) et `myfile` est le nom du fichier à écrire sur le disque.

```
DF.precision(4);    // fixe 4 chiffres significatifs
DF.write("myfile"); // écrit en colonne séparées par des tabulations
DF.write("myfile", fmt, [i1], [i2]); // applique fmt aux colonnes
```

On peut extraire une colonne comme un `vector` (vecteur) de `C++` pour l'utiliser avec une autre fonction :

```
DF.column(i);    // i-ème colonne
DF.column(f, i); // i-ème colonne, transformée par f
```

Histogrammes

Pour permettre la création d'histogrammes adaptables, `ePiX` fournit la classe `data_bins` qui modélise un intervalle défini divisé en intervalles à des emplacements spécifiés, non nécessairement espacés régulièrement. La vie d'un `data_bins` a deux périodes. Dans la première, des « coupures » (extrémités des sous-intervalles) sont ajoutés. Lorsque les données sont lues, les coupures sont « vérouillées » et ne peuvent plus être changées.

```
// [xmin, xmax] divisé en n intervalles égaux, n=1 par défaut
data_bins db(xmin, xmax, [n]);
db.cut(x);  // ajoute une coupure en x (si x est dans l'intervalle)
db.read(vector<double>); // lit les données, vérouille les classes
unsigned int count(db.pop());   // population courante
db.histogram([c]);// rectangles « réduit » d'un facteur c (1 par défaut)
db.plot([c]);     // interpolation par spline
```

La hauteur d'un rectangle de l'histogramme est l'effectif de la classe correspondante ; le comportement était différent dans les versions antérieures.

Cet extrait de code illustre quelques unes des fonctionnalités décrites ci-avant.

```
data_file DF("myfile.dat");
DF.transform(log_log);   // agit sur les 2 premières colonnes
data_bins db(-6, 4, 20); // [-6, 4], 20 classes
db.read(DF.column(1));   // lit col1
db.histogram();

DF.plot(BOX);            // nuage de pont log-log
DF.regression(1, 2);     // trace la droite de régression
DF.write("log_log.dat"); // écrit un fichier sur le disque
```

Chapitre 3

Manuel de référence

Ce chapitre couvre la conception et l'utilisation d'`ePiX` en faisant l'hypothèse que vous avez complétement digéré le matériel du chapitre 2. Les autres fonctionnalités sont expliquées ici et l'implantation en est décrite. Si une fonctionnalité n'est pas expliquée ici, consultez, s'il vous plait, le code source ou contactez l'auteur.

3.1 Plus loin à propos du C++

Un manuel ou un ouvrage de référence détaillé du même type est essentiel pour une étude sérieuse du `C` ou du `C++`. Allez auprès des maitres : *Le langage C* de Brian Kernighan et Dennis Ritchie [2] (version originale [1]) est une excellente ressource très maniable pour les fondements de la programmation procédurale tandis que *Le langage C++* de Bjarne Stroustrup [6] (version originale [5]) est une référence encyclopédique qui fait autorité.

Le `C++` est un langage puissant et complexe dont la syntaxe est semblable à celle du `C` ou à celle des langages de script de Maple et Mathematica. Un fichier d'entrée d'`ePiX` est un fichier de code source pour un programme `C++` qui écrit un fichier `eepic` comme fichier de sortie. On peut regarder `ePiX` comme une extension de `C++` ; de la même façon que LaTeX fournit une interface de haut niveau à TeX, `ePiX` fournit un lien de haut niveau entre la puissance de calcul de `C++` et l'environnement `picture` de LaTeX.

Comme tout langages de programmation de haut niveau, `C++` utilise des variables, des fonctions et des structures de controle. Les variables contiennent des morceaux de données telles que des valeurs numériques, des lieux géométriques, tandis que les fonctions opèrent sur les données. Une structure de controle, comme une boucle ou une instruction conditionnelle, affecte le déroulement du programme en fonction de son état courant. Un fichier source contient essentiellement des « instructions » qui accomplissent des actions allant de la définition d'une variable ou

d'une fonction jusqu'à la fixation d'un attribut d'une figure, la réalisation de calculs et l'écriture d'objets sur le fichier de sortie.

Noms et types

Les noms des variables et des fonctions ne doivent contenir que des lettres, des chiffres et le caractère « souligné (`_`) ». Le premier caractère d'un nom ne doit pas être un chiffre et la norme du langage réserve les noms commençant par un souligné aux auteurs de bibliothèques. Les noms sont sensibles à la casse mais c'est en général une mauvaise idée d'utiliser des noms simples avec majuscule et sans majuscule dans un même fichier. On utilise, de manière informelle, de nombreuses conventions pour l'usage des majuscules ; ce document utilise des mots sans majuscule séparés par des soulignés pour les variables et les fonctions et, à l'occasion, utilise des mots en capitales pour les constantes. De même que pour les noms des macros LaTeXiennes, la première considération est la clarté (de signification), la lisibilité et la cohérence.

Chaque variable, en `C++`, a un « type » tel que entier (relatif) (`int` pour *integer*), nombre décimal en double précision (`double`) ou booléen (`bool` – vrai ou faux –). `ePiX` fournit des types supplémentaires dont le plus courant est `P` pour « point ». Le constructeur `P(x,y,z)` crée (*le point de coordonnées*) (x, y, z) (*dans le repère cartésien courant*), tandis que `P(x,y)` donne $(x, y, 0)$ c.-à-d. en fait la paire (x, y). Une variable est définie par la donnée (*dans l'ordre*) de son type, son nom et une expression d'initialisation.

En `C` et `C++`, un *pointeur* est une variable qui contient l'adresse mémoire d'une autre variable. Bien que plus délicats que des variables ordinaires, les pointeurs sont utiles pour écrire certains algorithmes comme ceux de tri. [1]

Le `C++` fournit également des variables *références* ce qui permet de donner aux variables un pseudonyme. Leur utilisation vient de la manière dont les fonctions `C++` traitent leurs arguments, voir page 39.

Fonctions

Dans un langage de programmation, le terme « fonction » fait référence à un bloc de code qu'on peut appeler par son nom. Une fonction `C++` prend une liste

1. Au Japon, les adresses des batiments sont attribuées chronologiquement plutôt qu'en référence à l'emplacement dans la rue. Un batiment est analogue à une variable tandis que son adresse est celui d'un pointeur. Si le parlement japonais adoptait une loi rendant obligatoire l'attribution de numéros consécutifs le long des rues, il y aurait deux façons de procéder : déplacer les batiments (bouger les variables) ou renuméroter les batiments sans qu'ils bougent – sur place – (trier les pointeurs). Pour de semblables raisons d'efficacité, les algorithmes de tri en `C++` travaillent avec les pointeurs.

« d'arguments » et possède une « valeur de retour ». Il faut fournir ces informations, ainsi que le nom de la fonction, lorsque l'on définit une fonction. On ne peut pas définir une fonction à l'intérieur d'une autre. Toutefois une fonction peut appeler d'autres fonctions (elle-même aussi[2]) lors de son exécution :

```
int factorial(unsigned int n)
{
  if (n == 0) return 1;
  else return n*factorial(n-1);
}
```

Le type spécial `void` représente un « type nul ». Une fonction qui accomplit une action mais ne retourne pas de valeur a comme type de retour le type `void`. Une fonction sans argument peut être vue comme prenant un unique argument de type `void`.

Tout programme doit avoir une fonction spéciale nommée `main()` qui est appelée par le système d'exploitation quand on lance le programme. Les arguments de `main()` sont ceux de la ligne de commande et elle retourne un entier qui signale le succès ou l'échec. Les fonctions de l'utilisateur doivent être définies avant l'appel de `main()` ou dans un fichier compilé séparément.

Les fonctions en `C++` peuvent être aussi simple qu'une formule algébrique ou aussi complexe qu'un algorithme arbitraire. Les algorithmes de calcul de plus grand commun diviseur, de somme finie, de dérivée et intégrale numérique, de tracé de courbes fractales définies récursivement ou de courbe d'ajustement sont quelques unes des applications d'`ePiX`. Les nombreux fichiers d'exemple contiennent des algorithmes, du niveau de l'utilisateur, qui ne nécessitent pas de connaissance des structures de données internes d'`ePiX`. Le fichier source `functions.cc` contient des fonctions simples définies par des algorithmes et `functions.h` illustre l'utilisation des gabarits (*template*) de `C++`. Dans d'autres fichiers sources, tel que `plots.cc`, on trouvera la règle de Simpson, la méthode d'Euler etc.

Une erreur, comme une division par zéro ou la tentative de prendre l'intersection de deux lignes parallèles, peut apparaitre lorsqu'une fonction est exécutée. Dans cette situation, une fonction `C++` peut « lancer une exception » ou retourner un type d'erreur que l'appelant « attrape » et manipule. Si une exeption est lancée mais pas attrapées, le programme s'achève. Les opérateurs d'intersection d'`ePiX` lancent des exceptions lorsque certaines conditions ne sont pas réunies.

Fonctions mathématiques

`C++` connait de plusieurs fonctions mathématiques classiques par leur nom :

2. Note du TdS : On a donc en `C++` la possibilité d'implanter des fonctions récursives.

```
sqrt   exp   log   log10   ceil   floor   fabs
```

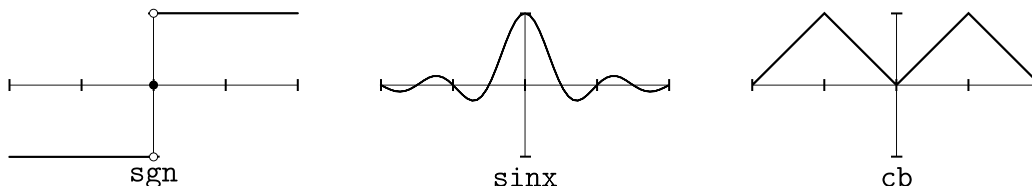
(`fabs` est la valeur absolue d'un argument décimal.) `ePiX` fournit des fonctions trigonométriques et leurs inverses sensibles au mode angulaire :

```
Cos    Sin    Tan
Sec    Csc    Cot
Acos   Asin   Atan
```

Les inverses des fonctions trigonométriques sont les branches principales.

La fonction `pow(x,y)` retourne x^y lorsque $x > 0$ et `atan2(u,x)` (N.B. ordre des arguments) retourne $\mathrm{Arg}(x + iy) \in (-\pi, \pi]$, la branche principale de arg. `C++` connait de nombreuses constantes avec 20 décimales telles que `M_PI`, `M_PI_2` et `M_E` pour π, $\pi/2$, et e respectivement. `ePiX` ajoute quelques fonctions :

```
recip   sgn   zero   sinx   cb   id   proj1   proj2
```



`recip` est la fonction « inverse de » définie comme valant 0 en 0 ; `sgn` est la fonction signum ; `zero` est la fonction constante nulle ; `sinx` est la fonction définie par $x \mapsto \sin(x)/x$ prolongée par continuité ; `cb` (pour « Charlie Brown ») est le prolongement 2-périodique de la valeur absolue réduite à $[-1, 1]$; `id` est l'identité, définie pour un type arbitraire de données ; les fonctions `proj` retournent leur premier et seconde variable quel qu'en soit le type.

La bibliothèque GNU `C++` définit d'autres fonctions, entre autres les fonctions hyperboliques inverses (`acosh` , etc.), `log` et `exp` en base 2, 10 ou b arbitraire (`log2` , etc.), les fonctions d'erreur et gamma (`erf` et `tgamma` [sic], respectivement), les fonctions de Bessel de 1re et 2e espèce : `j0` , `j1` , `y0` , etc. On utilise, par exemple, `jn(5, )` pour obtenir les indices supérieurs. Le manuel de référence de la bibliothèque GNU `C++` [3] décrit ces fonctions, et d'autres, en détail.

On peut utiliser les fonctions dans des définitions postérieures. Les fonctions de deux variables ou plus sont définies de façon analogue à celle d'une seule variable :

```
double f(double t) { return t*t*log(t*t); } // t^2 \ln(t^2)
double g(double s, double t) { return exp(2*s)*Sin(t); }
```

Introduction aux classes

Contrairement au C, le C++ fournit la possibilité d'utiliser la « programmation orientée objet ». En deux mots, une *classe* est la réalisation en code informatique d'un certain concept, tel que le point, la sphère, un graphe que l'on peut tracer ou une caméra. Les classes permettent au programmeur de séparer l'*interface* d'un objet (l'ensemble des opérations ayant un sens) de son *implantation* (les structures de données et les algorithmes qui réalisent cette interface).

L'implantation d'une classe comporte des *membres* (éléments nommés de données) et des *fonctions membres* (fonctions qui appartiennent à la classe et ont un accès libre à ses membres). Les classes C++ appliquent des permissions d'accès à leurs membres, protégeant ainsi les données de manipulations qui ne passeraient pas par l'interface.

Une interface idéale ressemble à une boite noire : elle cache complètement l'implantation. Afin de collaborer deux classes n'ont besoin que de connaitre leurs interfaces. Cette séparation de la forme et de la fonction rend les programmes modulaires, facilite le débogage, la réutilisation du code et la maintenabilité générale, particulièrement dans les gros programmes.

En programmation simple, on peut traiter les classes comme des types intégrés. Chaque objet d'une classe a ses propres fonctions membres dont la syntaxe d'appel diffère d'un appel de fonction « classique » :

```
circle C1(P(1,0), 1.5); // cercle de centre et rayon donnés
C1.draw();              // fonction membre circle::draw();
```

Bien entendu, cet appel dessine le cercle C1. De façon générale, un appel de fonction membre est composé du nom d'un objet de la classe, d'un point et du nom de la fonction membre. Les arguments, s'il y en a, sont placés entre parenthèses après le nom de la fonction membre, comme pour un appel de fonction habituelle.

Quelques courts paragraphes ne peuvent qu'au plus effleurer la surface de la programmation orientée objet et de l'utilisation des classes. Consultez à ce propos un ouvrage comme celui de Stroustrup [6] (original [5]) pour plus de détails.

Références et arguments de fonction

Le C et le C++ sont des langages dans lesquels les arguments sont « passés par valeur ». Les variables ne sont pas passées à la fonction mais des copies en sont faites et la fonction ne travaille que sur les copies. Bien que cette caractéristique entraine parfois quelques inconvénient, elle empêche les modifications intempestives des variable lors d'un appel de fonction dans une autre partie du programme. Le passage par valeur localise donc la logique d'un programme et protège de bogues faciles à écrire mais extrèmement difficiles à trouver.

Il y a deux raisons habituelles pour permettre à une fonction l'accès à une variable plutôt qu'à une copie : la variable est une grosse structure de données dont la copie a un « cout prohibitif » ; la fonction doit changer la valeur de certains de ses arguments (p. ex. une fonction `echange(x,y)` qui échange les valeurs de `x` et de `y`). Dans chacune de ces situations, une valeur peut être passée par référence. En gros, une référence fournit efficacité d'un pointeur (adresse mémoire) en taille mais peut être crée et manipulée dans un fichier comme une variable ordinaire.

Chaque variable référence est « liée » à un objet existant. Les instructions

```
double  x=1;   // définition d'une variable ordinaire
double& rx=x;  // liaison d'une référence, notez le &
```

définissent une variable `x` de valeur 1 puis y lie la variable référence `rx`. Tant que `rx` existe, elle fait référence à `x`. Si la valeur de `x` change, la valeur de `rx` fait de même. Toutefois `rx` a la taille d'un pointeur, quelle que soit la taille de `x`, ce qui fait qu'on peut passer `rx` efficacement à un appel de fonction.

De temps à autres, une fonction a un besoin légitime de changer les valeurs de ses arguments. Dans un tel cas, un appel par référence permet une solution propre. Cette technique de mise à jour de variables est présentée comme une fonctionnalité de `C++`. Toutefois un telle tromperie circonvient à l'encapsulation des données assurée par le passage par valeur et on devrait l'éviter sauf en cas d'absolue nécessité. Si une fonction ne fait que « mettre à jour » la valeur d'une variable, cette variable devrait problablement être de type de classe et sa mise à jour faite par une fonction membre.

Les fonctions qui utilisent des variables références pour des arguments de taille importante peuvent le faire en toute sécurité en déclarant leurs arguments `const`. Ce mot-clé signifie que la fonciton ne change pas la valeur de l'argument. Toute tentative de modification d'un argument `const` sera arrêtée par le compilateur.

La déclaration d'une fonction doit indiquer que ses arguments sont des références. Les déclarations ci-dessous ont les significations idiomatiques indiquées.

```
class matrix;
double det(matrix); // passage par valeur, peut-être inefficace
matrix& transpose(matrix&); // modifie probablement son argument
double trace(const matrix&); // ne modifie pas son argument
```

Contrairement aux pointeurs, les arguments référence n'impose aucun fardeau syntactique à l'utilisateur. Si `A` est une `matrix` (matrice) alors `transpose(A) ;` et `trace(A) ;` compilent. Vous n'avez pas besoin de déclarer explicitement des variables références pour les passer à la fonction.

Surcharge

`C++` fournit la « surcharge » : plusieurs fonctions peuvent avoir le même nom pour autant que le nombre ou le type de leurs arguments soient différents. (Il *ne* suffit *pas* que le type de retour soit différent. Le compilateur doit être capable de sélectionner la fonction à partir de sa syntaxe d'appel.) Pour l'utilisateur il semble que la même fonction manipule intelligemment de multiple listes d'arguments. Naturellement, les noms surchargés devraient faire référence à des fonctions conceptuellement apparentées. `ePiX`, par exemple, fournit de nombreuse fonctions `plot`.

Portée

Une instructions `C++` finit avec un point-virgule. Une collection d'instructions encloses dans une paire d'accolades est un « bloc de code » et peut être vue comme une unique instruction logique. Les accolades déterminent une « portée » à l'intérieur de laquelle on peut réutiliser des noms de variable sans risque d'ambiguïté. Une variable définie entre accolades est dite *locale* (à la portée dans laquelle elle est définie) ; sa valeur ne peut être utilisée en dehors de la portée.

Le corps d'une fonction est un bloc de code, comme les différentes branches associées à une instruction de controle. Le compilateur n'est pas très regardant à propos des espaces, tabulations et sauts de ligne aussi le fichier source devrait être organisé de telle façon qu'il soit facile à lire. Un renfoncement (*indentation*) montre un niveau d'imbrication dans les blocs de code mais les détails spécifiques font l'objet de débats passionnés. De même qu'avec le nommage des variables, la clarté et la cohérence sont les critères importants.

Entêtes et précompilateur

Un source `C++` est compilé en plusieurs étapes qui se suivent à l'insu de l'utilisateur. La première, le prétraitement (ou précompilation), procède au remplacement simple de texte par inclusion de fichier, développement de macros et de d'instruction conditionnelle de compilation. Ensuite, le source est compilé et assemblé : les instructions écrites en langage lisible par l'homme sont analysées puis traduites en assembleur. Enfin les fichiers objets sont liés : les appels de fonction sont transformées en *offsets* de fichiers codés en dur, en impliquant éventuellement des fichiers externes de bibliothèque, et les instructions du programme sont empaquetées dans un exécutable binaire que le système d'exploitation peut utiliser.

Le prétraitement est beaucoup moins utilisé en `C++` qu'en `C` ; le langage luimême permet de remplacer les macros par des fonctionnalités plus sûres et plus riches telles que les variables `const` et les fonctions en-ligne. L'inclusion de fi-

chier et la compilation conditionnelle donnent les occasions principales d'utiliser un précompilateur. Des lignes de la forme

```
#include <cstdlib>
#include "epix.h"
```

fait que le contenu du *fichier d'entête* est lu dans le fichier source. Un fichier d'entête contient des *déclarations* de variables et de fonctions, des instructions qui spécifient des noms et des types mais ne définissent pas vraiment des données. Les déclarations disent au compilateur juste assez pour qu'il puisse résoudre les appels d'expressions et de fonctions sans connaitre les valeurs spécifiques ou les définitions des fonctions.

La compilation conditionnelle ressemble au code conditionnel de LaTeX. Par exemple, un fichier peut produire une sortie couleur ou monochrome comme suit :

```
#ifdef COLOR
  ...  // code pour créer la figure en couleur
#endif // COLOR
#ifndef COLOR
  ...  // code monochrome
#endif // undef COLOR
```

Le « symbole de compilation » `COLOR` est un nom ordinaire de `C++`. Pour controler la compilation on peut soit placer une ligne `#define COLOR` dans le fichier soit (mieux) fournir le drapeau dans la ligne de commande :

```
epix -DCOLOR <file.xp>
```

À chaque `#ifdef` doit correspondre un `#endif`. Placer une commentaire à chaque `#endif` est une bonne habitude ; dans un fichier réel, le début et la fin d'un bloc conditionnel peuvent être séparés de plus d'un écran.

Comparaison avec la syntaxe de LaTeX

En tant que langage de programmation, `C++` fournit certaines fonctionnalités communes à tous les langages (tels que LaTeX, Metapost, Perl, Lisp...) et suit des règles de grammaire. Les différences les plus marquées entre LaTeX et `C++` sont

1. Toute instruction `C++` et tout appel de fonction doit être terminé par un point-virgule. L'oubli d'un point-virgule peut entrainer des messages d'erreur cryptique de la part du compilateur. Les directives du précompilateur, qui commencent avec un `#`, ne sont pas terminées par un point-virgule ;
2. La barre oblique inverse est un caractère d'échappement en `C++` :

```
// Place l'étiquette   $y=\sin x$ en (2,1)
// Notez la barre oblique ^ unique dans le fichier de sortie
label(P(2,1), P(0,0), "$y=\\sin x$");
//    et la double barre  ^^ dans le source
```

3. Les noms de variables et de fonctions ne peuvent contenir que des lettres (y compris le souligné) et des chiffres, sont sensibles à la casse et doivent commencer par une lettre ;
4. En `C++` les variables doivent avoir un *type* déclaré comme `int` (entier) ou `double` (décimal en double précision). Si une variable a une portée globale et que sa valeur n'est pas modifiée, sa définition devrait être placée dans le préambule ou au début de `main`. Les variables locales devraient être définies dans la plus petite portée possible. Contrairement au `C`, `C++` permet de définir les variables à leur première apparition ;
5. `C++` exige l'utilisation explicite de `*` pour noter la multiplication ; la juxtaposition n'est pas suffisante. `C++` ne permet pas l'utilisation de `^` pour l'exponentiation, p. ex. `t^2` est non valide, à la place il faut utiliser `t*t` ou `pow(t,2)`.
6. `C++` permet des commentaires unilignes ou multilignes. Tout ce qui est situé entre une double barre oblique `//` et le saut de ligne est ignoré. Les chaines `/*` et `*/` délimitent des commentaires multilignes. Un commentaire uniligne peut être inclus dans un commentaire multiligne mais le compilateur n'accepte pas les commentaires multilignes imbriqués.

À eux deux, `C` et `C++` réservent environ 100 mots clés que l'on ne peut pas utiliser comme nom de variable ou de fonction.

3.2 La caméra

`ePiX` décrit un monde cartésien en le projetant mathématiquement sur un plan écran puis en faisant une mise à l'échelle affine vers la page imprimée. La caméra, qui réalise la projection du monde sur l'écran, est constitué d'un *corps* (données qui déterminent la position et l'orientation de la caméra) et d'un objectif (la projection effective sur le plan de l'écran).

Le corps

L'orientation spatiale de la caméra est décrite par un triplet de vecteurs unitaires mutuellement orthogonaux. En souvenir de jours heureux à la plage, les vecteurs sont appelés *sea* (mer), *sky* (ciel) et *eye* (œil). Le plan de l'écran est parallèle au plan mer-ciel ; le vecteur mer pointe horizontalement vers la droite, le

ciel pointe verticallement vers le haut. L'œil en est le produit vectoriel [3] qui pointe directement vers le spectateur.

La base mer-ciel-œil est placée sur la *cible* qui est donc l'origine du plan de l'écran. Le *point de vue* est sur la droite passant par la cible et dirigée par le vecteur œil et est le centre de projection de l'objectif par défaut. La distance du point de vue à la cible est la *plage* [4]. L'orientation, le point de vue, la cible et la plage définissent complètement (et de manière redondante) la situation géométrique de la caméra dans le monde.

L'objectif

L'*objectif* est une application du monde sur l'écran. `ePiX` est fournit avec trois objectifs : *ombre* (*shadow*, par défaut), *grand angulaire* (*fisheye*) et *bulle* (*bubble*). Chaque objectif simule l'apparence des objets du monde tels qu'ils seraient vus par un observateur placé au point de vue. L'objectif ombre est une projection conique du point de vue sur l'écran. Les autres objectifs réalisent une projection radiale sur une sphère centrée au point de vue puis projette la sphère sur le plan de l'écran ; le grand angulaire fait une projection orthogonale (de telle sorte que l'image toute entière tient à l'intérieur du disque et que les positions derrière la caméra sont inversées) tandis que l'objectif bulle fait une projection stéréographique depuis le point directement derrière le point de vue.

On peut définir d'autres objectifs ; on pourra consulter le fichier `camera.cc` comme gabarit. Syntactiquement, un objectif est une fonction d'un seul argument `P` à valeur `pair`. Pour assurer le comportement attendu un objectif devrait respecter la signification du corps de la caméra.

Manipulation de la caméra

La caméra est une classe `C++` manipulée par des fonctions membres. La commande `begin()` initialise la caméra aussi toute modification de la caméra doit apparaitre dans le corps de la figure. Par défaut, le point de vue est à une grande distance sur la partie positive de l'axe des z, regardant vers le bas le plan (x, y). Cela fournit le comportement attendu pour les figures en dimension 2.

On peut fixer séparément le point de vue et la cible. En général, une telle opération modifie la direction du vecteur œil ce qui oblige `ePiX` a redéterminer le vecteur ciel. Si possible, l'axe des z se projette vers le ciel, sinon on utilise l'axe des y. Le point de vue et la cible peuvent bouger le long de l'axe de l'œil, ce qui

3. Note du TdS : N'y a-t'il pas quelque poésie à voir un œil comme le produit vectoriel de la mer et du ciel ?

4. Note du TdS : En anglais *range* est l'étendue ou la plage et il m'a semblé plaisant de rester au bord de la mer.

modifie la plage en conservant l'orientation : `range()` fixe la cible et `focus()` fixe le point de vue. Chaque commande redimentionne l'image : notez qu'augmenter la distance focale (*focus*) *augmente* la taille de l'image.

La caméra peut pivoter autour de chacun de ses axes. Les axes de rotation passent par la cible donc les rotations autour des vecteurs mer ou ciel modifie le point de vue. Pour mieux controller la caméra on fixera d'abord la cible puis le point de vue. Si on le veut on finira par les rotation de l'œil.

3.3 Rognage et cadrage

`ePiX` fournit deux opérations de masquage pour gérer les éléments d'une figure situés loin de la cible : le rognage (*clipping*) – dans le monde – et le cadrage (*cropping*) – sur l'écran. La « boite de rognage » peut être vue comme un ensemble de « murs ». Lorsque le rognage est activé, les objets hors les murs sont invisibles. Par défaut, la boite de rognage est un grand cube centré à l'origine.

La « boite de cadrage » est un rectangle du plan de l'écran. Lorsque le cadrage est activé, les objets dont les projections sont situées hors du rectangle de cadrage ne sont plus visibles. Par défaut, la boite de cadrage est la boite-cadre. Comme la figure est dessinée sur la page avec une mise à l'échelle affine de la boite-cadre vers la boite spécifiée LaTeXienne, le cadrage par défaut garantit que la figure est située entièrement dans la région – imprimée – allouée par LaTeX.

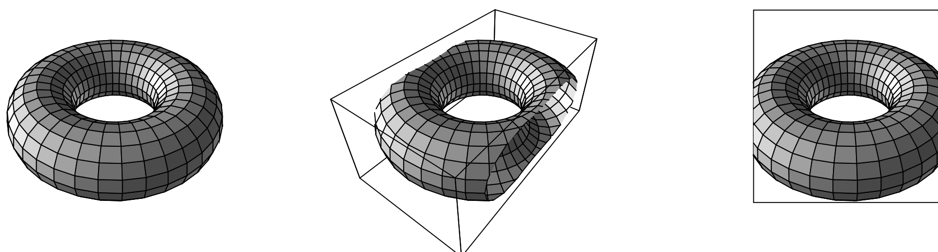


FIGURE 3.1 – Clipping and cropping a torus mesh (boxes added).

Par défaut, cadrage et rognage sont inactifs. La commande `clip(bool)` active et désactive le rognage. L'argument a `true` pour valeur par défaut. On peut fixer la boite de rognage par les commandes

```
clip_box(P pt1, P pt2); // coins opposés
clip_box(P pt);         // coins opposés pt et -pt
clip_to (P pt);         // ...   ...     pt et P(0,0,0)
```

De façon analogue, le cadrage est (dés)activé par `crop(bool)`. La boite de cadrage est donnée par une paire de coins opposés, `cropbox(pt1,pt2)` ; la troisième coor-

données est ignorée. Sans argument, la commande `crop_box()` redéfinit la boite cadre comme égale à la boite-cadre.

3.4 Attributs

Au minimum, on doit dire à `ePiX` la taille finale de la figure et les dimensions du rectangle cartésien à allouer. Le préambule doit contenir assez d'information pour créer un espace de travail. Dans le corps du fichier d'entrée, « l'état de dessin » détermine l'apparence de la figure. Les attributs sont des déclarations, ils sont fixés par des commandes qui acceptent des arguments du type déclaré, éventuellement `void`. Les couleurs et l'épaisseur des chemins sont controllées en écrivant immédiatement des commandes LaTeXiennes dans le fichier de sortie ; les autres attributs sont gérés en interne.

- Mode angulaire : `radians()`, `degrees()` ou `revolutions()` ;
- Épaisseur de chemin : `plain()`, `bold()`, `pen(double)` ;
- Style de chemin :
  - `solid()` ;
  - `dashed(double)`. L'argument facultatif est la densité de pointillé, fraction (de 0,05 à 0,95) du chemin occupé par les pointillés ;
  - `dotted(double)`. L'argument facultatif est le diamètre (en `pt`) du `dot`.

  Les commandes `dash_length(double)` et `dot_sep(double)` fixe la distance (en `pt`) entre les extrémités d'une ligne pointillées ou tiretées.
- Couleur : `rgb(densités)`, `cmyk(densités)`, `primary(densité)`. Une densité est un `double` compris entre 0 (pas de couleur) et 1 (saturation complète).
- Remplissage : `fill(bool)`, l'argument vaut `true` par défaut.
  - Profondeur de gris `gray(double)`, 0 : blanc, 1 : noir ;
  - style de remplissage, couleur de remplissage de PSTricks.
- Rotation de texte : `label_angle(double)`
- Rognage et cadrage ;
- Caméra.

Mode angulaire

`ePiX` a trois modes angulaires : `radians()` (valeur par défaut), `degrees()` et `revolutions()`. Ces modes affectent toutes les opérations trigonométriques, y compris les rotations de la caméra, le tracé des arcs et des ellipses, le tracé des courbes en polaires, les angles des étiquettes et les fonctions trigonométriques elles-mêmes. Les fonctions sensibles au mode angulaire ont un nom à capitale, p. ex. `Cos`, `Tan`.

Couleur et teinte

ePiX fournit une sortie en couleur par l'intermédiaire de l'extension pstcol, en utilisant les modèles rgb et cmyk. Les teintes de gris des régions sont obtenus à l'aide de eepic.sty (sans faire appel à pstcol). On voit mieux les couleurs après conversion du document au format PostScript ou PDF. Si l'on préfère, on pourra regarder les fichiers EPS grâce à xdvi.

Une couleur rgb est donnée par un triplet de décimaux fixant les densités entre 0 (pas de couleur) et 1 (saturation complète). Une couleur cmyk est déterminée de manière analogue par quatre décimaux. Les densités sortant du domaine $[0, 1]$ sont « rognées ». Comme les styles de ligne, les couleurs restent actives jusqu'à ce qu'elles soient supplantées. On peut utiliser sept couleurs primaires et le blanc par leurs noms. Dessiner en blanc (white) peut servir de liquide de correction pour retirer des morceaux de manière précise.

```
red();             // rgb(1,0,0);
magenta(0.6);      // cmyk(0,0.6,0,0);
rgb(0.2,0.7,0.8); // couleur personnelle
```

PSTricks

PSTricks est une puissante collection de macros, crée par Timothy van Zandt [7] et autres, pour incorporer du PostScript dans LaTeX. ePiX utilise PSTricks essentiellement pour le remplissage en couleur mais ne fonctionne pas encore de manière intégrée. On ne devrait utiliser PSTricks dans un fichier qu'en cas d'absolue nécessité. **Ce qui suit suppose qu'on a besoin de PSTricks pour le fichier.** La commande

```
use_pstricks(bool);
```

fixe un drapeau interne qui détermine si un chemin est tracé comme un chemin path de eepic ou une psline de PSTricks. Lorsqu'elle est émise avant le begin(), cette commande synchronise également l'unitlength et l'épaisseur de ligne de PSTricks avec celles de ePiX et fixe le style de remplissage à « plein ». On peut activer et désactiver PSTricks à volonté mais *la première activation doit être faite dans le préambule.*

Quand pstricks est activé, les commandes de la forme

```
fill_color("<color name>");
psset("<pstricks command>");
```

sont utilisées pour fixer des attributs comme le style et la couleur des chemins et du remplissage. La couleur de remplissage par défaut est "white". Cet extrait, pris

dans le fichier d'exemple `contour.xp`, montre comment on définit une nouvelle couleur et l'utilisation de `psset()`

```
use_pstricks();      // synchronise longueur, etc.
begin();
use_pstricks(false); // désactive temporairement
...
std::cout << "\n\\newrgbcolor{orange}{1 0.7 0.2}";
psset("fillcolor=orange, linecolor=green, linewidth=1.5pt");
```

On consultera le manuel de PSTricks pour plus d'information.

Deux incompatibilités majeures impliquent le remplissage et la couleur. À moins que le style de remplissage ne soit explicitement fixé à `none`, PSTricks remplit tous les chemins même ceux qui ne sont pas fermés. Deuxièmement, PSTricks manipule les couleurs par des chaines nommées, aussi il faut utiliser une sortie brute pour exploiter toute la puissance de PSTricks depuis `ePiX`.

3.5 La classe path

Une structure de données `path` est une liste de bas niveau d'`ePiX`, liste ordonnée de points qui peut être rognée, cadrée, projetée, concaténée et dessinée. Les données de chemin brut sont utiles pour les chemins complexes batis par morceaux. Les constructeurs disponibles sont :

```
path(p1, p2);          // droite (extrémités)
path(p1, p2, p3);      // spline quadratique
path(p1, p2, p3, p4);  // spline cubique
path(p1, v1, v2, t_min, t_max); // cf. ellipse arc
path(f, t_min, t_max); // graphe ou chemin paramétré

polyline(n, &p1, ...); // n points, suivis par des pointeurs
polygon(n, &p1, ...);  // idem, mais marqué comme fermé
```

Le premier argument du constructeur de chemin paramétré est une fonction d'une variable à valeur réelle ou vectorielle. Dans chacun des cinq premiers constructeurs, on peut fournir un argument final facultatif pour spécifier le nombre de points utilisés.

Les constructeurs `polyline()` et `polygon()` acceptent un nombre indéterminé d'arguments ; en conséquence leurs arguments doivent être passés en tant que « pointeurs » ou adresses en mémoire :

```
polyline(2, P(0,0), P(1,1));// mauvais : des objets pour arguments
P p1 = P(0,0), p2=P(1,1);
polyline(2, &p1, &p2); // correct : des pointeurs pour arguments
```

Cela rend les polylines et les polygones incommodes pour réaliser une figure vite fait mais n'inflige qu'une peine légère si les sommets sont définis ailleurs comme ils devraient l'être dans un fichier logiquement structuré.

On peut construire un chemin point par point avec la fonction `pt()` qui accepte trois (ou deux) doubles ou un `P`.

```
path bord.pt(0,0).pt(2,0).pt(0,3); // polyline
path octa.close(); // declare un nouveau polygone
for (int i=0; i<8; ++i) octa.pt(polar(2, 45*i)); // on suppose degrees()
```

On peut construire des chemins composites par concaténation. Si `path1` et `path2` sont des chemins alors les commandes

```
path1 += path2;
path1 -= path2;
```

remplace `path1` avec le chemin obtenu en traversant `path1` « vers l'avant » puis en suivant `path2` vers l'avant ou en remontant (respectivement). La notation est sensée faire penser à des chaines d'homologies de dimension 1. Le fichier d'exemple `contour.xp` illustre la création et la manipulation de chemin. Enfin, on notera que `path` est une structure de données et non une commande de dessin. On doit appeler explicitement la fonction `path::draw()` pour créer une sortie visible.

Objets semblables aux chemins

Les objets semblables aux chemins comprennent les polygones à un nombre fixé de sommets – (segments) de droites, triangles, quadrilatères – et des objets construits à partir d'eux – ce qui comprend les axes de coordonnées, les quadrillages et les flèches – ; les courbes à un nombre variable de points – ellipses, arcs, splines – ; les graphes de fonctions d'une variable.

Les polygones à une nombre fixé de sommets sont dessinés avec des commandes de haut-niveau.

```
line(p1, p2);
triangle(p1, p2, p3);
quad(p1, p2, p3, p4);
rect(p1, p2);          // rectangle de coordonnées
spline(p1, p2, p3);    // spline quadratique avec pts de controle
spline(p1, p2, p3, p4);// spline cubique
```

La commande `line()` accepte un argument numérique facultatif interprété comme paramètre d'expansion : `line(p1,p2,t) ;` trace un segment dont le milieu est celui du segment d'extrémités `p1` et `p2` mais dont la longueur est $2^{t/100}$ fois celle du segment $[\mathtt{p1}, \mathtt{p2}]$ – avec $t = 100$ on double la longueur et avec $t = -100$ on la réduit de moitié. Les arguments de `rect()` doivent être des points d'un plan parallèle à l'un des plans de coordonnées.

En interne, `ePiX` marque les chemins comme fermés ou non ; les triangles, les quadrilatères, les polygones, les ellipses et les flèches sont fermés. Si un chemin fermé et plein est tracé alors que le remplissage est activé, le chemin est rempli avec la teinte courante de gris. Si le remplissage est désactivé le chemin est tracé mais non rempli. On ne peut pas remplir des chemins pointillés ou tiretés en une seule manœuvre : il faut d'abord remplir le chemin plein puis tracer la frontière pointillée ou tiretée.

La teinte de gris est un nombre compris entre 0 (blanc) et 1 (noir) et vaut par défaut 0,3. Le grisé est opaque avec PSTricks aussi l'ordre d'apparition dans le fichier source des éléments de la figure est-il significatif lorsque le remplissage est activé.

Le remplissage coloré n'est disponible que *via* PSTricks. Les fonctions de PSTricks relativement bien gérées par `ePiX` sont illustrées dans le fichier d'exemple `contour.xp`. En principe, on peut obtenir n'importe quelle capacité de PSTricks avec une sortie brute. Toutefois cette approche est, en général, déconseillée puisque, p. ex. les déclarations de couleurs de PSTricks ne sont par reconnues par `eepic` et vice-versa. La Prochaine Génération d'`ePiX` travaillera plus harmonieusement avec PSTricks.

Les arcs elliptiques sont définis par leur centre, une paire de vecteurs, une étendue angulaire et un nombre facultatif de points :

```
ellipse(ctr, v1, v2, t_min, t_max, n);
```

utilise $n + 1$ points pour tracer la courbe paramétrée

$$t \mapsto \mathrm{ctr} + (\cos t)v_1 + (\sin t)v_2, \qquad t_{\min} \leqslant t \leqslant t_{\max}$$

Le nombre de points vaut, par défaut, 80 pour un tour complet et proportionnellement moins pour un arc. Lorsque l'étendue angulaire sous-tend au moins un tour complet, l'ellipse est marquée comme fermée et sera rempli si le remplissage est activé.

L'arc du cercle parallèle au plan (x, y), de centre p_1 et de rayon r, sous-tendant l'angle (dans le sens direct [5] dans l'unité d'angle courante) de θ_1 à θ_2 est tracé avec

```
arc(p1, r, theta1, theta2);
arc_arrow(p1, r, theta1, theta2); // idem, avec pointe de flèche
```

5. Note du TdS : Sens direct c.-à-d. le sens contraire des aiguilles d'une montre.

Si θ_2 est plus petit que θ_1, l'arc va dans le sens indirect. La pointe de flèche est en θ_2. Si un `arc_arrow` est trop petit, il n'est pas tracé.

3.6 Structures des données géométriques

Les éléments d'une figure ont une description abstraite et une apparence typographique. Cette section décrit la première propriété. `ePiX` fournit les classes `P`, `segment`, `circle`, `plane` et `sphere` que l'on peut utiliser dans les constructions de géométrie euclidienne. Chaque classe est définie par une petite quantité de données (p. ex. un centre, un rayon et un vecteur unitaire normal pour un `circle`) et fournit des constructeurs, des opérateurs d'intersection, de transformations affines et une fonction `draw`.

`ePiX` fournit également des outils pour les géométries sphérique et hyperbolique, spécialement ce qu'il faut pour tracer des droites dans les modèles dits *du demi-plan* et *du disque de Poincarré* du plan hyperbolique et aussi ce qu'il faut pour tracer les latitudes, longitudes, arcs de grands cercles, triangles sphériques, polyhèdres réguliers et courbes paramétrées sur la sphère.

Triplets et repères

Les points et les déplacements dans l'espace sont représentés par des triplets. Le nom du type est « `P` » bien que on eut pu utiliser « `triple` » pour des raisons de rétrocompatibilité. On fournit des constructeurs sphérique, cylindrique/polaire. On peut effectuer sur ces triplets les opérations canoniques – et d'autres moins canoniques – de l'algèbre linéaire : addition/soustraction, multiplication par un scalaire, produits vectoriel, scalaire, composante par composante et orthogonalisation. Lorsque l'on forme une expression symbolique utilisant des triplets les scalaires doivent être rassemblés à gauche et les triplets à droite. Si nécessaire l'utilisation de parenthèse forcera une association particulière.

```
P pt(x,y,z);         // definit pt = (x,y,z)
double u=pt.x1();    // première coordonée de pt, etc.
P(x,y);              // même chose que P(x, y, 0);
polar(r, theta);
cis(t);              // polaire (1, t), alias Cos(t) + i*Sin(t)
sph(r, theta, phi); // theta=longitude, phi=latitude
P(a,b,c)|P(x,y,z);  // produit scalaire, ax+by+cz
P(a,b,c)&P(x,y,z);  // produit par composante, P(ax,by,cz)
p*q;                 // produit vectoriel, p x q
J(p);                // quart de tour autour de l'axe Oz
p%q;                 // orthogonalisation, p (mod q)
```

De manière explicite, `p%q` est l'unique vecteur `p+l*q` perpendiculaire à `q`.

Un `frame` (repère) est un ensemble de trois vecteurs unitaires orthogonaux deux à deux. Le repère normal est l'ensemble `E_1`, `E_2`, `E_3`. Le constructeur transforme un ensemble de trois vecteurs non-coplanaire en repère :

```
frame(); // le repère normal
frame(P v1, P v2, P v3); // orthonormalise (v1, v2, v3)
```

Le troisième vecteur d'un nouveau repère est positivement proportionnel à `v3`, le deuxième positivement proportionnel à `v2%v3` et le premier est le produit vectoriel de deux derniers. De cette façon, un repère (`frame`) est orienté positivement et ne dépend pas de `v1`.

Intersection

Le concept de *généricité* est au cœur de la compréhension de l'intersection des structures de données géométriques d'`ePiX`. Pour que la définition fonctionne, deux objets disjoints, tangents ou confondus se coupent de manière « non-générique ». (Les géomètres noteront que cela change considérablement de la définition habituelle.) Les opérateurs d'intersection d'`ePiX` lance une exception lorsque les opérandes sont non-génériques. Si `epix` se termine avec un message d'erreur, vérifiez que vous ne cherchez pas à intersecter deux objets malformés ou mal positionnés.

Dans `ePiX`, un `segment` s'allonge en une droite pour les besoins d'une intersection. La table 3.1 présente les types d'intersections (génériques !) dans `ePiX`. L'intersection est commutative aussi on n'a rempli que la partie supérieure de la table. Toutes les intersections ne sont pas définies.

*	segment	circle	plane	sphere
segment	P	segment	P	—
circle		segment	segment	—
plane			line	circle
sphere				circle

TABLE 3.1 – Types d'intersections d'`ePiX`.

Segments

Un `segment` est une paire (*non ordonnée*) de points. Un segment peut être translaté par un `P` et on peut prendre son milieu.

```
segment L1 = segment(P(0,0), P(2,4));
P mid = L1.midpoint();    // le milieu de L1
segment L2(mid, P(-2,3)); // forme un segment
L2 += P(1,0);             // translate L2 de (1,0)
L1.draw(); L2.draw();
dot(L1*L2);               // point d'intersection
```

Cercles

Une structure de données `circle` (cercle) est constituée d'un centre, un rayon et un vecteur unitaire perpendiculaire. On fournit trois constructeurs :

```
circle(P center, double radius, P normal);
circle(P center, P point);
circle(P p1, P p2, P p3);
```

Si les derniers arguments du premier constructeur ne sont pas spécifiés ils prennent, par défaut, les valeurs suivantes : si `normal` n'est pas donné, il prend la valeur `E_3`, si, de plus, le rayon n'est pas spécifié, il vaut 1, enfin, le centre est à l'origine s'il n'est pas donné explicitement. Comme d'habitude en `C++`, seuls les derniers arguments peuvent être implicites ; l'appel `circle(center, normal)` *ne* crée *pas* un cercle de rayon unité avec centre et normale données.

Le second crée un `circle` parallèle au plan (xy), de centre donné, passant par le point donné. Le troisième retourne un `circle` passant par les points donnés. Une exception est lancée si le centre (`center`) et le `point` ne sont pas dans un plan parallèle au plan (xy) ou si les trois points p_i sont colinéaires.

Un `circle` peut être translaté d'un `P` avec l'opérateur + ou agrandit (ou réduit) d'un facteur `double` avec l'opérateur $*$:

```
circle C1=circle();       // cercle unité (« trigo »)
circle C2 = C1+P(1,0.5); // translate vers le haut et la droite
C2 *= 1.5;                // multiplie le rayon par 1.5
C1.draw(); C2.draw();     // montre le travail
(C1*C2).draw();           // trace le segment d'intersection
```

Sphères

Une `sphere` (sphère) est déterminée par un point et un rayon qui sont, par défaut, l'origine et l'unité. On fournit des constructeurs pour une `sphere` de centre donné qui passe par un point donné et pour une `sphere` donnée par ses poles.

```
sphere(center, point);
poles(north, south);
```

De même que pour les `circles`, on fournit les opérateurs de translation et d'homothétie (agrandissement, réduction). Les outils particuliers à la géographie et à la géométrie sphérique sont décrit dans la section 3.8.

La fonction `draw()` d'une `sphere` dessine un horizon visible depuis le point de vue courant. Bien que cet horizon soit un cercle dans l'espace objet, son image sur le plan de l'écran est générallement une ellipse. De plus, des points antipodaux ne sont généralement pas représentés par des points symétriques par rapport au centre de l'ellipse. Ces effets sont d'autant plus prononcés que le point de vue est proche de la `sphere` et que le centre de celle-ci est éloigné de la cible (`target`).

Droites et plans

`ePiX` n'a pas de structure de données spéciale pour les droites mais il fournit la fonction `Line` (droite) qui trace la droite joignant deux points. Naturellement seul un segment est tracé. `ePiX` cadre (`crop`) toujours une `Line` et, de plus, enlève la demi-droite qui se trouve derrière l'observateur. Ainsi, une `Line` apparait imprimée comme un segment dont les deux extrémités sont sur la boite-cadre soit dont une seule extrémité est à l'intérieur de la boite-cadre, extrémité qui représente le point à l'horizon visible. La Prochaine Génération (alias `ePiX3d` ou « Version 2.x ») fournira une structure de données pour les droites.

Un `plane` (plan) est déterminé par un point et un vecteur unitaire normal ou par trois points non-colinéaires. La fonction `draw()` représente les droites d'intersection du plan (`plane`) avec les faces de la boite de rognage. À moins qu'on ait fixé la boite de rognage à la main, les droites d'intersection éviteront presque surement la boite-cadre et resteront donc invisibles.

3.7 Graphes et domaines

On utilisera le mot « application » (*map*) pour désigner une fonction `C++` d'un ou plusieurs arguments `double` retournant un `double` ou un `P`. Mathématiquement, une application peut être représentée de deux façons : avec un graphe (qui retient l'information quant au domaine) ou avec une courbe paramétrée (qui se débarasse du domaine). `ePiX` fait l'hypothèse qu'une application à valeur `double` doit être représentée par un graphe et qu'une application à valeur `P` doit l'être par une courbe paramétrée. Dans les deux cas on parlera de « graphique » (*plot*). Les graphiques d'`ePiX` sont soit « en mailles » (*wire mesh*), obtenus avec une commande `plot`, soit « ombrée » (*shaded*), obtenus par la commande `surface`.

Un `domain` (domaine) d'`ePiX` comprend une boite de coordonnées, définie par une paire de coins opposés, et deux maillages (*meshes*) qui précisent la quantité de données à représenter. Pour des graphiques sélectifs un `domain` peut être *découpé* et *retaillé*. On décrit ces opérations en détail plus loin.

Les arguments d'une commande `plot` sont une application suivie soit d'un domaine soit de quelque chose logiquement équivalent. Par exemple :

```
double f(double t) { return t*t; }
P F(double u, double v) { return P(u, v, exp(u)*Sin(v)); }
P G(double u, double v, double w) { return P(v*w, u*w, u*v); }

// [0,3] x [-1,2]: 12 x 6 rectangles, 60 x 60 intervalles
domain R(P(0,-1), P(3,2), mesh(12,6), mesh(60,60));
// [-1,1] x [-1,1] x [0,1] divisé de manière analogue
domain R3d(P(-1,-1,0), P(1,1,1), mesh(4,8,5), mesh(60,48,120));

plot(f, -2, 2, 60);// f:[-2,2] -> R, en utilisant 60 intervalles
plot(F, R);        // graphe de exp(u)*Sin(v)
plot(G, R3d.slice2(0.5); // G: R^3 -> R^3 restreinte à y=0.5
```

Par convention (que le compilateur fait respecter) des commandes `plot` utilisant des applications à valeur `P` acceptent un argument `domain` comme dans les deuxième et troisième commandes ci-dessus. Pour représenter une fonction à valeur `double`, en revanche, il faut fournir un équivalent logique d'un `domain`, habituellement les extrémités et le nombre d'intervalles, comme dans la première commande `plot` ci-dessus.

Supposons que le `domain R2` est, comme ci-dessus, le rectangle $[0,3] \times [-1,2]$. L'argument `mesh(12,6)` le divise en un réseau de 12×6 sous-rectangles connu sous le nom de *maillage grossier* (*coarse mesh*). Une application de deux variables à valeur `P` est représentée au-dessus du réseau du maillage grossier. Toutefois, plutot que de tracer des quadrilatères, `ePiX` trace les courbes images à la résolution définie par l'argument `mesh(60,60)`, le *maillage fin* (*fine mesh*). Dans cet exemple, les courbes sont dessinées, dans chaque direction, en utilisant 60 segments de droite.

La séparation des roles des maillages grossier et fin permet à un graphe en mailles de suivre de près une surface sans utiliser une grille fine de courbes. Les deux parties de la figure 3.2 sont tracées avec un maillage grossier de 6×20. Dans la première figure, le maillage fin est aussi 6×20 alors que dans la seconde il est 12×60.

Le maillage grossier n'a de sens que pour des domaines de dimension supérieure ou égale à 2. La taille du maillage grossier détermine le nombre de courbes ou de surfaces tracées *perpendiculairement* à la direction d'une coordonnée tandis que le maillage fin détermine le nombre de segments utilisés *le long* de cette direction.

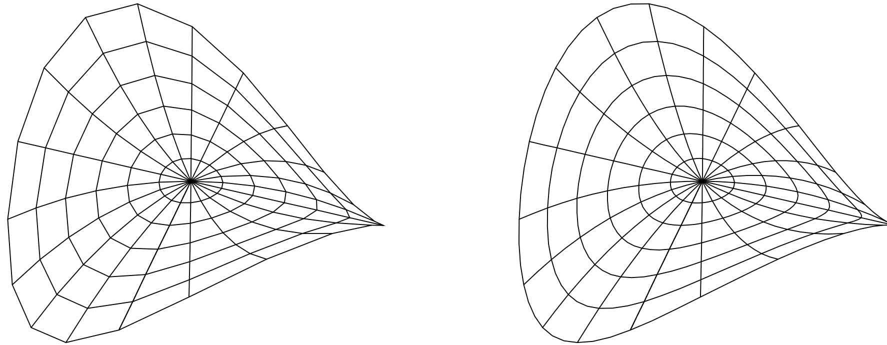


FIGURE 3.2 – Maillage grossier et maillage fin.

Pour obtenir des résultats prévisibles le maillage fin devrait être un petit multiple du maillage grossier.

Les graphiques se comportent de manière analogue pour des `domain`s en dimension 3 et des applications de trois variables : le « squelette de dimension 1 » de l'image du `domain` est tracé. De plus, une application de trois variables à valeur `P` peut être tracée au-dessus d'un `domain` de dimension 1 ou 2. Toutefois, l'effet peut être inattendu à moins que le domaine n'ait été obtenu par découpage (*slicing*). On ne peut pas tracer une application d'une ou deux variables au-dessus d'un `domain` de dimension 3.

Opérateurs sur les domaines

Un domaine peut être *retaillé* suivant chaque axe de coordonnées pour lequel l'épaisseur est positive :

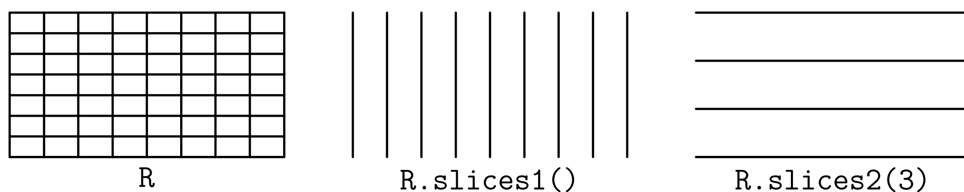
```
domain R(P(0,0), P(2,1), mesh(12,12), mesh(36,36));
// [0,2] x [0.5,0.75], grossier 12 x 3, fin 36 x 9
domain R_new = R.resize2(0.5,0.75);
```

Retailler permet de représenter une application restreinte à un ensemble de parties de son domaine. On peut s'en servir pour insister sur des parties de l'image, couper en couches des éléments de la scène, rapiécer des surfaces etc.

Autant que possible, la retaille conserve les tailles absolues des quadrillages. Pourtant, en général, on tronque. Pour le domaine ci-dessus, `R.resize2(0,0.2)` aurait un maillage grossier de 12×2 (car $12/5 = 2{,}4 \to 2$) et un fin de 36×7. Pour éviter un comportement inattendu, choisissez des tailles de maillage telle qu'une retaille n'entaine pas de troncature des entiers.

Un domaine peut être *tranché* en rendant constante une des variables. Le résultat est un domaine dont le nombre de dimension est inférieur d'une unité de celui originel. Avec le `R` de ci-dessus, `R.slice1(0.3)` est le `domain` de dimension 1 $\{0{,}3 \times [0,1]\}$ avec un maillage grossier de 1×12 et un fin de 1×36.

Quand on représente une famille d'application, un `domain` a des opérateurs `slices` (tranches) qui retourne la liste des `domains` obtenus en tranchant suivant une variable selon des constantes régulièrement espacées. Un argument optionnel fixe le nombre de tranches. Il n'est pas nécessaire que cet argument soit relié au maillage grossier.



On peut utiliser les commandes de taille et de tranchage directement dans une commande `plot` :

```
plot(F, R.resize(0,0.5));
plot(F, R.slices1());
```

Surfaces ombrées

`ePiX` fournit les outils pour représenter des surfaces ombrées, cf. également la section 4.1. Si le remplissage est activé, une surface est ombrée suivant l'angle du vecteur normal avec la direction de la caméra, donnant l'effet d'un éclairage ambient constant. Autrement, on utilise le gris courant – 0,3 par défaut.

La commande `surface` à la même syntaxe que la commande `plot` lorsqu'on ne trace qu'une seule surface. En général, le retrait des objets cachés demande quelques différences. Une commande `ePiX` écrit une strophe dans un fichier de sortie ; les éléments de la scène sont dessinés dans leur ordre dans le fichier d'entrée. Une scène contenant une ou plusieurs surfaces opaque ne peut pas être construire surface par surface. Au contraire, les différentes surfaces doivent être assemblées dans une unique structure de données avant qu'on puisse les dessiner.

Comme dans la représentation en mailles, le maillage fin est utilisé pour tracer les frontières des pièces de surface, cela tend à représenter les surfaces plus souplement pour de faibles tailles du maillage grossier. Si le maillage grossier l'est trop, toutefois, deux effets visuels indésirables peuvent apparaitre. Premièrement, des régions adjacentes de la surface peuvent être ombrées très différemment puisque l'ombrage est constant sur chaque pièce définie par le maillage grossier (voir la 2e image à partir de la gauche de la figure 3.3). Deuxièmement, une pièce presque

tangente à une ligne de visée peut être mal tracée si la pièce se replie vers l'arrière puisque l'on trace la frontière de la *pièce* et pas les bords visibles de la surface mathématique (voir la figure 3.3, les deux images de gauche.)

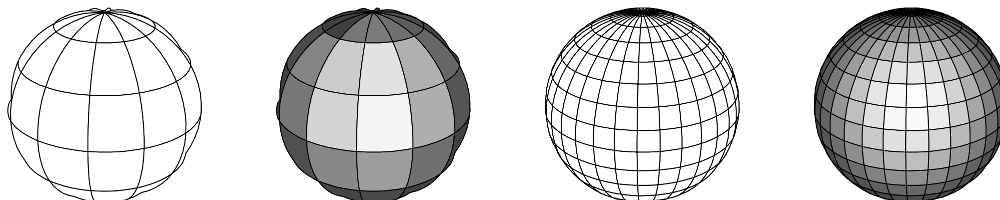


FIGURE 3.3 – Effets de maillage grossier – tous les maillages fins sont identiques.

Surface unique

La commande `surface(F, R)` représente la fonction `F` à valeur `P` sur le domaine `R`, elle est l'équivalent « ombré » de la commande `plot` correspondante. `ePiX` fournit aussi des commandes spéciales pour les surfaces de révolution :

```
void surface_rev(f, t_min, t_max, n_lats, n_longs);
void surface_rev(f, g, t_min, t_max, n_lats, n_longs);
```

La première fait tourner le graphe de f autour de l'axe des x, la seconde utilise la courbe paramétrée $t \mapsto \big(f(t), g(t)\big)$ comme profil. Dans chaque cas, le paramètre intervalle `[t_min,t_max]` est divisé en `n_lats` sous-intervalles de même amplitude, `n_longs` copies du profil (24 par défaut) sont tracés et la surface complète (une révolution complète) est tracée.

Une troisième forme de la commande utilise un argument `domain` pour controler l'étendue des longitudes et dessine la surface de révolution dans un repère cartésien défini par la base orthonormale `coords` – par défaut, la base canonique. Les arguments `f` et `g` définissent une courbe paramétrique dans le plan des deux premiers éléments de `coords` et le premier élément définit l'axe de rotation.

```
void surface_rev(f, g, R, frame coords);
```

Surfaces multiples

L'algorithme de traitement des surfaces cachées d'`ePiX`-1.x découpait la surface en pièces suivant le maillage, les triaient dans l'ordre (approximativement) décroissant de distance à la caméra et les imprimait. Cette technique fonctionnait assez correctement pour des surfaces sans intersection et restait même acceptable

pour manipuler des surfaces sécantes dont les mailles se coupaient suivant leurs frontières.

Les surfaces multiples sont construites à partir d'une ou plusieurs applications et d'un ou plusieurs `domains` de dimension 2. Dans l'extrait de code ci-dessous, `F` et `G` sont des fonctions de 3 variables à valeur `P` et `R` est un `domain` de dimension 3.

Pour représenter les images de plusieurs `domains` par *une seule application*, il faut grouper les `domains` dans une liste puis émettre la commande `surface` :

```
surface(F, R.slices3());

domain_list DL(R.slice1(0)); // construit la liste de domaines
DL.add(R.slice2(0.5));       // ajoute un domaine
DL.add(R.slices3());         // etc.
surface(G, DL);              // dessine
```

Pour traiter des *applications multiples*, `ePiX` fournit une classe `scenery` (paysage). Conceptuellement, une `scenery` est une agglomération de surfaces ombrées, construite l'une après l'autre à partir d'applications et de `domains` de dimension 2. La fonction `add` accepte deux arguments – une application et soit un `domain` soit une liste de `domains` – et offre ses données à la `scenery` au lieu de les tracer immédiatement. La `scenery` complète est tracée par un appel fait à la main :

```
scenery S(F, R.slice3(0.25)); // S contient une surface
S.add(F, R.slice2(0));        // S contient deux surfaces
S.add(G, R.slices1(2));       // S contient cinq surfaces
S.draw();
```

Consultez les fichiers d'exemples `spherical.xp` et `minkowski.xp` dans le répertoire `extras` pour des exemples complets.

En principe, une scène peut contenir un nombre arbitraire de surfaces et l'écriture du fichier `eepic` ne pose aucun problème. Toutefois une figure qui contient beaucoup d'objets tend à peser sur les piles internes de LaTeX. De fréquents changements de couleur exarbent le problème. À moins d'accroitre la mémoire de LaTeX, il est peu vraissemblable qu'une figure contenant plus d'un millier d'éléments maillés compile, même dans une résolution modeste, une surface peut facilement contenir 1 000 pièces de gris d'intensités diverses. Mais cela peut dépendre de votre installation.

Fonctions utilitaires

Dans cette section, `f` et `g` sont deux fonctions d'une variable à valeur `double`. `ePiX` définit des fonctions numériques qui retourne le maximum ou le minimum

sur un intervalle, une approximation des racines et qui travaillent avec les dérivées et les intégrales définies.

```
sup(f, a, b);     // max/min de f sur [a,b]
inf(f, a, b);
newton(f, g, x0); // trouve une approximation du point d'intersection
```

La méthode de Newton retourne le point d'intersection de deux fonctions données, en partant de la graine donnée qui devrait être raisonnablement proche de la solution attendue. Si on tombe sur un point critique ou si on accomplit cinq itérations sans amélioration un avertissement apparait et le résultat courant (probablement incorrect) est retourné [6]. La seconde fonction g est la constante nulle si elle n'est pas spécifiée.

On utilise les classes `Deriv` et `Integral` pour le calcul des dérivées et intégrales numériques et pour représenter graphiquement ces fonctions.

```
Deriv df(f); // objet fonction : df(x) = f'(x)
df.eval(t);  // retourne f'(t)
df.left(t);  // derivée à gauche en t:  (f(t)-f(t-dt))/dt
df.right(t); // derivée à droite en t: (f(t+dt)-f(t))/dt

Integral prim(f,a); // objet fonction: prim(x) = int_a^x f
prim.eval(b);  // intégrale numérique de f sur [a,b]
double val(Integral(f).eval(1));   // val = \int_0^1 f
```

La limite inférieur de l'intégrale vaut 0 par défaut.

Les tangentes et les enveloppes (familles de tangentes) sont dessinées avec

```
tan_line(f, t);                 // f real- or vector-valued
envelope(f, t_min, t_max, n);  // family of tangent lines
tan_field(f1, f2, t_min, t_max, n); // field of tangents
```

Les fichiers d'exemples `conic.xp` et `lissajous.xp` illustrent ces capacités.

Graphiques et analyse

`ePiX` peut représenter la dérivée ou une primitive des fonctions réelles et représenter les sommes de Riemann pour les intégrales définies. Soit f une fonction réelle d'une seule variable.

6. Note du TdS : À ma demande, l'auteur précise ce qui suit : Pour `ePiX` « convergence » signifie obtenir une racine avec une précision d'environ 10^{-5} donc si cinq itérations n'ont pas suffit, *l'utilisateur devrait pouvoir faire mieux.*

```
plot(Deriv(f), a, b, n);  // graphe de f' sur [a,b]
plot(Integral(f, x0), a, b, n);
riemann_sum(f, a, b, n, TYPE);
```

La deuxième commande trace la primitive définie par $x \mapsto \int_{x_0}^{x} f(t)\,dt$ sur $[a, b]$ où, comme avant, x_0 vaut 0 par défaut. La troisième trace les rectangles ou les trapèzes dont la somme des aires approche l'intégrale définie de f sur $[a, b]$. Le `TYPE` peut être `UPPER`, `LOWER`, `LEFT`, `RIGHT`, `MIDPT` ou `TRAP`.

`ePiX` résoud les équations les équations différentielles ordinaires (abr. anglaise *ODE*) en deux ou trois variables et représente les champs de vecteurs (champ des vitesses) et de tangentes (*slope field*) et de tangentes orientées (*dart field*). Soit `F` une fonction de deux ou trois variables à valeur `P`.

```
ode_plot(F, p_0, t_min, t_max, n);
flow(F, p_0, t_max, n);
```

La première commande représente la solution du problème avec condition initiale $\dot{x} = F(x)$, $x(0) = p_0$ sur l'intervalle de temps précisé. Si $t_{\min}$ est omis, sa valeur est 0 et la courbe démarre à p_0. Avec quelques manipulations pour faire pivoter un champ plan d'un quart de tour, on peut utiliser `ode_plot` pour tracer les courbes de niveau d'une fonction de deux variables, cf. le fichier d'exemple `dipole.xp`. La fonction `flow` retourne la valeur (approchée) obtenue en partant de p_0 et en appliquant la méthode d'Euler à n pas de résolution de l'équation $\dot{x} = F(x)$., ce qui est utile pour placer avec précision des marques ou des têtes de flèches le long d'une courbe intégrale.

On peut obtenir la représentation d'un champ de vecteurs (dans un plan ou l'espace) au-dessus d'un domaine `R` de trois façons :

```
vector_field(F, R, [scale]); // vraie longueur
dart_field  (F, R, [scale]); // longueur constante
slope_field (F, R, [scale]); // longueur constante
```

On échantillone le champ aux nœuds du maillage grossier. Si le domaine est de dimension 2, le graphe est une tranche plane du champ, même si le champ dépend de trois variables. Si le domaine est de dimension 3, le champ est dessiné dans des tranches successives $z =$ const en partant de la hauteur du premier coin de `R` et finir à la hauteur du second.

Le dernier argument facultatif, qui vaut 1 par défaut, est un facteur d'homothétie des têtes de flèches d'un champ de vecteurs et de la longueur (constante) des vecteurs dans les champs obtenus par `dart_field` ou `slope_field`. Le fichier d'exemple `extras/vfield.xp` illustre ces capacités, y compris l'utilisation des styles de lignes de PSTricks pour réaliser l'effacement des objets cachés.

Dans chaque commande de représentation de champ, l'argument de type domaine peut être remplacé par deux points – représentant les coins d'un rectangle de coordonnées – et deux entiers – le nombre d'intervalles de quadrillage dans les directions de coordonnées choisies. On ne peut représenter que des tranches planes de champ de vecteurs dans cette syntaxe de rechange.

Courbes fractales récursives

Considérons un chemin fait de segments de même longueur qui peuvent être orienté par n'importe quel angle de la forme $2\pi k/n$ radians pour $0 \leqslant k < n$, comme les dents d'un engrenage. Un chemin est donné par une suite finie d'entiers, pris modulo n. Par exemple, si $n = 6$ alors la suite $0, 1, -1, 0$ correspond au chemin ASCII `_/_`. Une approximation d'une fractale par `ePiX` commence avec une telle « graine » (*seed*) puis remplace récursivement – jusqu'à une profondeur (*depth*) donnée – chaque segment avec une copie réduite et pivotée de la graine. La graine ci-dessus engendre la fractale connue comme « flocon de von Koch ». En voici le code :

```
const int seed[] = {6, 4, 0, 1, -1, 0};
fractal(P(a,b), P(c,d), depth, seed);
```

La première donnée de `seed[]` (6 ici) est le nombre n de « dents », le second (4) est le nombre de termes de la graine et le reste constitue la graine proprement dite. Le chemin final joint (a, b) à (c, d). Le nombre de segments dans le chemin final croît de façon exponentielle avec la profondeur, aussi des profondeurs supérieures à 5 ou 6 risque vraisemblablement de dépasser les capacités de LaTeX ou PostScript.

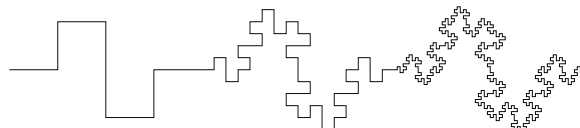


FIGURE 3.4 – Itérations successives de {`4,8,0,1,0,3,3,0,1,0`}

3.8 Géométrie non-euclidienne

On spécifie des segments de droites hyperboliques par leurs extrémités dans les modèles du demi-plan supérieur ou du disque de Poincaré. Dans chaque cas rien n'est produit si l'un des points est situé en dehors du modèle.

```
hyperbolic_line(p, q);
disk_line(p, q);
```

Pour des raisons de compatibilité avec l'espace hyperbolique de dimension 2, le modèle du demi-plan est définie comme $\{(x_1, x_2, x_3) \mid x_2 > 0\}$.

Un repère détermine les coordonnées géographiques sur une `sphere` (sphère) : le premier élément est dirigé vers la longitude 0 sur l'équateur, le troisième vers le pole nord. Une courbe de latitude dépend d'une `sphere`, un `frame` (repère), d'une latitude numérique et d'un intervalle de longitudes. Une courbe de longitude est décrite d'une manière similaire.

```
latitude(lat, long_min,long_max, sphere S, frame coords);
longitude(lngtd, lat_min, lat_max, sphere S, frame coords);
```

Par défaut, `coords` est le repère canonique (`frame`) et `S` est la `sphere` unitaire. Ces commandes ne trace que la partie de la courbe visible depuis le point de vue (`viewpoint`) courant. La fonction `back_latitude` trace la partie invisible de la courbe de latitude.

Les arcs de sphère et les triangles sphériques sont décrits par leurs extrémité. Seule la direction du vecteur, joignant le centre de la sphère à l'extrémité de l'arc ou du triangle, est significative, si la sphère subit une homothétie ou un déplacement, le même appel de fonction tracera l'objet correspondant sur la nouvelle sphère.

Les fonctions qui suivent tracent les parties visibles (*front*) des arcs de grands cercles :

```
front_arc(p1, p2, S);  // petit arc sur S de p1 à p2
front_arc2(p1, p2, S); // arc de p1 à -p1 passant par p2
front_line(p1, p2, S); // grand cercle passant par p1 et p2
```

Par souci de garder un niveau élevé d'abstraction, on fourni triangles et polyèdres réguliers (platoniques). Le fichier d'exemple `extras/polyhedra.xp` en illustre l'utilisation.

```
front_triangle(p1, p2, p3, S); // triangle sphérique
front_tetra(S, coords);  // tétraèdre régulier
front_cube(S, coords);   // hexaèdre
front_octa(S, coords);   // octaèdre
front_dodeca(S, coords); // dodecaèdre
front_icosa(S, coords);  // icosaèdre
```

Chaque fonction a une version `back` (arrière) qui trace la partie cachée. Le tétraèdre, le cube et l'octaèdre sont – avant mise à l'échelle – inscrits dans le cube d'arête 2 centré à l'origine et dont les arêtes sont parallèles aux vecteurs du repère (`frame`). Le point $(1, 1, 1)$ est un sommet du tétraèdre.

Avant mise à l'échelle, les sommets de l'icosaèdre sont situés sur le rectangle doré dont les sommets sont $(\pm\gamma, 0, \pm 1)$ et ses images par permutation circulaire des coordonnées. Le dodécaèdre est le dual de l'icosaèdre.

Les courbes paramétrées sur la sphère unité peuvent être définie soit par projection radiale d'une courbe gauche soit par projection stéréographique d'une courbe plane :

```
plot_R(phi, t_min, t_max, n);    // radial
plot_N(f1, f2, t_min, t_max, n); // depuis le pole nord
plot_S(f1, f2, t_min, t_max, n); // depuis le pole sud
```

Une tentative d'effectuer une projection radiale sur une courbe passant par l'origine engendrera une erreur de division par zéro. La projection stéréographique applique le plan équatorial $z = 0$ à la sphère unité par projection depuis le pole correspondant : $N = (0, 0, 1)$, $S = (0, 0, -1)$.

Chaque commande de graphe sphérique accepte un préfixe `front` ou `back` qui imprime seulement la partie visible ou invisible (respectivement) depuis le point de vue (`viewpoint`) courant.

Compte tenu de la façon dont `ePiX` produit les couches, il vaut mieux en général placer les parties cachées de la sortie avant les parties visibles en fixant des épaisseurs ou des styles de lignes suggérant des lignes cachées.

3.9 Animation

`ePiX` est idéalement adapté à la création d'animation mathématiquement exactes : si une figure dépend convenablement d'un paramètre de « temps » alors on peut utiliser une boucle pour dessiner toute la figure pour des valeurs de temps différentes, produisant ainsi des « instantanés » de la figure au fur et à mesure du temps. Le shell script `flix` automatise le processus de compilation d'un fichier d'entrée convenable en une collection de `png`s et les assemble en une animation `mng`. C'est ImageMagick qui est le moteur de manipulation d'images.

Un fichier `flix` est un fichier `epix` avec deux restrictions :

- La variable `double tix` est utilisé comme « horloge » ;
- `main` accepte deux arguments sur la ligne de commande et s'en sert pour définir `tix`.

Le mode `emacs` de Jay Belanger reconnait l'extension `.flx` et insère un gabarit si on ouvre un tampon vide. La création de fichier `flix` est aussi facile que celle de fichiers `epix`. Le répertoire `samples/extras` contient une poignée de fichier `flix` que l'on peut consulter comme source d'inspiration.

Un fichier `.flx` typique peut prendre de 30 secondes à une paire de minutes pour compiler. `flix` imprime une rassurante barre de progression, comptant le nombre de fichier `eps` créés. Il y a un délai de quelques secondres (ou plus) après que la dernière vue a été produite pendant lequel l'utilitaire `convert` d'ImageMagick crée les fichiers `png` à partir des `eps` puis monte le film.

Pour faciliter le débogage, on peut lancer `elaps` sur un fichier `flix`. `elaps` travaille en une fraction du temps et si `elaps` ne peut pas produire d'image visionable, `flix` échouera certainement.

Par défaut, `flix` crée des films de 24 images dans lesquels `tix` s'écoule de 0 à 1 et anime à 0,08 seconde par image. Des options de la ligne de commande peuvent changer ces paramètres et d'autres, utilisez l'aide intégrée de `flix` pour de plus amples informations.

3.10 Dépannage

`ePiX` est constitué de sept fichiers : un fichier d'entête (`epix.h`), une bibliothèque compilée (`libepix.a`), un fichier de code shell commun (`epix-lib.sh`) et quatre shell-scripts (`epix`, `elaps`, `laps`, et `flix`). Ces fichiers sont créés à la compilation en utilisant les variables du `Makefile`. Outre les parties exécutable du programme, on trouve des fichiers d'exemple, de la documentation, des fichiers de configuration et des notes diverses. Ces composants sont placés dans des emplacements standards.

Installation

L'installation public (dans `/usr/local`) se fait par défaut. Décompacter l'archive `tar` puis déplacez-vous dans le répertoire des sources (`epix-x.y.z_complete`). Lancez

```
./configure [option(s)]
make
make install
```

(la dernière en tant que `root` si besoin). Si vous avez fait l'installation quelque part sous `$HOME` lisez le fichier `FR-POST-INSTALL` pour des instructions complémentaires.

Le répertoire dans lequel vous installez `ePiX` est noté `$INSTALL`. Un `make install` réussi crée les fichiers suivants :

```
$INSTALL/bin/{epix,elaps,laps,flix}
$INSTALL/include/epix.h
$INSTALL/lib/{libepix.a,epix-lib.sh}
$INSTALL/share/man/man1/epix.1
```

Les manuels, documents d'exemple, notes et fichiers README, et les fragments de configuration sont installés dans des sousrépertoires de `$INSTALL/share/doc/epix`.

Problèmes connus

Des versions boguées du script de conversion `ps2epsi` ont été diffusées avec des versions récentes de RedHat et Mandrake. L'auteur a vue deux erreurs *différentes* d'un octet qui font que `elaps` échoue avec une erreur de `sed`. Si `elaps` produit une erreur de `sed` vous avez un `ps2epsi` bogué. Cherchez le message d'erreur précis ou, si vous êtes habile avec les expressions régulières[7] réparer le script vous-même. (Dans les deux cas évoqués ci-dessus le problème était un caractère spécial non-protégé.)

Erreurs d'exécution

Des fautes de frappe créent des erreurs quand on lance `epix` ; le compilateur émet en général des messages utiles, désignant les lignes fautives du fichier d'entrée et le type d'erreur.

Deux sources communes d'erreur dans un fichier d'entrée syntactiquement correct concernent les mauvais placements de la caméra et les intersections non-génériques. Si la caméra est trop près d'objets de la scène, il arrive que l'objectif essaie de diviser par zéro (ou par un petit epsilon) entrainant une erreur `nan` (ou des coordonnées très grandes) dans le fichier de sortie. Dans cette situation, `epix` réussira mais `elaps` échouera. En revanche, un message d'erreur de la forme :

```
/usr/local/bin/epix: line 275: 27333 Aborted ...
```

signale une exception non traitée et donc une intersection non-générique.

Bien que les scripts soient énormément testés, ils sont des sources possibles d'ennuis à l'exécution. Par exemple, `elaps` essaie de passer des options au compilateur tout en s'en gardant d'autres pour son usage propre, tout cela en essayant de traiter intelligemment les commandes malformées. Si un script fait quelque chose d'inattendu, c'est problablement une Fonctionnalité pas un Bogue. Dans tous les cas, faites connaitre à l'auteur les comportements anormaux.

Erreurs de script Si un script échoue (le fichier de sortie est inexistant ou illisible), relancez le script avec l'option `-v` (*verbose* c.-à-d. verbeux, bavard). Cela fera écrire la sortie et les messages d'erreur à l'écran.

Commande introuvable Afin d'utiliser `ePiX`, le répertoire `$INSTALL/bin` doit se trouver dans votre `PATH` (cf. `FR-POST-INSTALL`) ; tapez
`echo $PATH`
pour voir votre `PATH`. Si le répertoire `$INSTALL/bin/` ne s'y trouve pas, lisez le

7. Note du TdS : Ou « rationnelles » dont l'usage se perd.

fichier `FR-POST-INSTALL` ou demandez de l'aide à un « expert ». La procédure peut dépendre du shell que vous utilisez. Dans tous les cas, vous devrez modifier le fichier de configuration du shell.

Si vous ne pouvez toujours pas faire fonctionner `ePiX`, envoyez un courriel à l'auteur. En général, il est utile de préciser la version du système d'exploitation (p. ex. RedHat 8.2 ou Debian Potato sur un G4 Powerbook), la version du compilateur `C` (p. ex. `gcc-3.2`), la version d'`ePiX` et le site depuis lequel vous avez téléchargé les sources (CTAN, le site du projet, etc.) Si vous ne comprenez pas ce que signifie un message d'erreur, n'hésitez pas à le copicoller ou à l'attacher à votre courriel comme texte pur. L'auteur n'a pas accès à un grand nombre de variété de systèmes et parfois les erreurs signalées ne trouvent pas d'explication mais, à ce jour, aucun problème n'est resté sans solution.

Erreurs de LaTeX

Quelques raisons peuvent faire que LaTeX s'arrête avec un message d'erreur pendant la lecture d'un fichier `eepic` écrit par `ePiX`. La plus commune est l'apparition d'un `nan` (*not a number* c.-à-d. « pas un nombre ») là où LaTeX attend un nombre. Cela signale généralement une division par zéro ou une mauvaise exponentiation.

Quand un nombre est très petit, `ePiX` peut l'écrire en notation exponentielle. Si cela se produit, LaTeX s'arrête avec un message d'erreur lorsqu'il tente de lire quelque chose comme `1.4142135e-14`. Ce bogue a été traité ; adressez un rapport de bogue à l'auteur si vous rencontrez ce comportement dans un code `ePiX`. Vous pouvez éditer manuellement le fichier `eepic` et remplacer le soupacement de capacité (*underflow*) par 0. Dans ce cas, il est judicieux de renommer le fichier modifié de peur qu'`ePiX` ne l'écrase à la prochaine exécution.

Des dépassements de capacité sont possibles avec LaTeX si un point a des coordonnées supérieures à 2^{16}, vérifiez que vous ne tentez pas de représentez graphiquement un pole ou quelque chose de ce genre.

Chapitre 4

Sujets avancés

Ce chapitre couvre des trucs *ad hoc* et des techniques ouvertes qui requièrent relativement plus de finesse dans la programmation. Vous aurez presque surement besoin d'une référence `C++` externe si vous ne pratiquez pas ce langage.

4.1 Effacement des objets cachés

`ePiX` écrit le fichier de sortie suivant l'ordre dans lequel les objets apparaissent dans le source. L'ordre est significatif car PostScript construit les figures par couches : les objets dessinés par-dessus les objets tracés précédemment. On peut utiliser les polygones ombrés pour obtenir un effacement des objets cachés d'une surprenante efficacité dans les surfaces à mailles. Cette section décrit les structures de données définies dans les fichiers sources `surface.*`.

L'idée fondamentale est de créer une classe de polygones ombrés qui connaissent leur distance approximative à la caméra. Pour des raisons de simplicité des calculs, une « facette » d'une surface maillée est traitée comme un quadrilatère, placé à la moyenne arithmétique de ses sommets. La frontière d'une facette est crée à partir d'une application et un domaine en traçant un rectangle à maille fine dans le sens direct.

Pour dessiner les surfaces paramétrées, les facettes sont stoquées dans un vecteur `C++`, classées par ordre décroissant de distance à la caméra et écrite dans le fichier de sortie. Si le remplissage est activé, la densité de gris d'une facette dépend du cosinus de l'angle que fait son vecteur normal avec le vecteur joignant la caméra à l'élément.

Cet algorithme simple fonctionne étonnamment bien lorsque des éléments du maillage se coupent suivant une arrête complète. Pour incorporer des éléments semblables à des droites (p. ex. les axes de coordonnées, les graphes maillés) à des surfaces ombrées, la meilleure technique est souvent de ranger manuellement

les éléments d'une scène de haut niveau, en fragmentant les surfaces ombrées (en retaillant le domaine ou en cadrant, par exemple) si nécessaire. Le fichier d'exemple `saddle.xp` illustre les techniques utilisables.

En bidouillant le code on peut décorer les surfaces ombrées. Par exemple, la fonction `facet::draw` du fichier `surface.cc` peut être facilement modifiée pour tracer des éléments de droite, des tangentes ou des vecteurs normaux avec la facette elle-même. Le fichier d'exemple `extras/decorate.xp` contient quelques idées. (Les décorations sont activées par des drapeaux du compilateur, consultez le fichier lui-même pour des informations concernant la compilation.)

4.2 Extensions

Grace à une suggestion de Andrew Sterian, `ePiX` est extensible. La méthode préférée est la création d'un module externe à utiliser à la compilation, analogue d'un fichier de style/macros de LaTeX. On peut aussi modifier le code des sources d'`ePiX` avant de le compiler, c'est rigide (et exige des privilèges d'administrateur) mais peut être utile pour changer les choix par défaut pour tout un système. Les extensions des utilisateurs s'étendent du fichier d'entête ne demandant que quelques connaissances élémentaires de `C++` à une bibliothèque, compilée séparément, qui étend les capacités d'`ePiX` de manière importante.

Fichier d'entête

Un fichier d'entête de `C++` (*header*) a, par convention, le suffixe `.h` comme dans `myheader.h`. Pour utiliser un tel fichier d'entête personnalisé, placez la ligne `#include "myheader.h"` dans le fichier source.

Les définitions de l'utilisateur peuvent être implantées facilement et robustement par des « fonctions en-ligne ». Ces fonctions en-ligne sont semblables à des macros mais sont plus sures et ont plus de capacités (puisqu'elles sont manipulées par le compilateur au lieu du préprocesseur). En voici quelques exemples :

```
inline void Bold(void) { pen(1.5); }
inline void purple(void) { rgb(0.5, 0, 0.7); }
inline void draw_square(double s) { rect(P(-s,-s),P(s,s)); }
inline double cube(double x) { return x*x*x; }
```

Le mot-clé `void` signifie que la fonction ne retourne aucune valeur ou – quand on l'utilise comme paramètre – que la fonction ne prend pas d'argument. Les définitions des fonctions en-ligne sont syntactiquement exactement identiques à celles de fonctions ordinaires mais *doivent* être placées dans un fichier d'entête ou

dans le fichier source dans lequel on les utilise. Les exemples ci-dessus doivent être utilisés dans un fichier d'entrée de la manière suivante :

```
Bold();
draw_square(cube(1.25));
```

Compilation

Les prochaines sections esquissent la création d'une « bibliothèque statique » sur GNU/Linux et expliquent comment incorporer des fonctionnalités personnalisées à la compilation. Les fichiers `extras/affine.cc` et `extras/affine.h` illustrent les techniques décrites ci-dessous et peuvent servir de guide et de base pour des essais.

Une petite bibliothèque est, en général, écrite comme un fichier d'entête (*header*) qui contient des déclarations de fonctions (encore appelées « prototypes ») et un fichier source contenant le code effectif. Par convention – sous *nix – ces fichiers ont des extensions `.h` et `.cc` respectivement. Entête et source peuvent « inclure » (*include*) d'autres fichiers d'entête pour incorporer des capacités supplémentaires.

```
/* my_code.h */
#ifndef MY_CODE
#define MY_CODE
#include <cmath>   // entête de bibliothèque standard de math
#include "epix.h"  // entête d'ePiX
using ePiX::P;

namespace Mine {   // pour éviter les conflits de noms
  // fonctions pour la relativité
  double lorentz_norm(const P&);
  bool   spacelike(const P&);
} // fin de l'espace de noms (namespace)
#endif // MY_CODE
```

Ce fichier montre deux « caractéristiques de sécurité ». Les trois lignes `MY_CODE` empèchent l'inclusion multiple du fichier. Dans un fichier de cette taille, la protection contre l'inclusion est un excès patent de zèle mais au fur et à mesure que votre base de code grossit et que le nombre de fichier d'entête croît, cette protection devient essentielle. Deuxièmement, l'entête introduit « Mine » comme espace de noms. Dans cet espace de noms, deux fonctions sont déclarées comme prototypes en donnant le type retourné, le nom et les types des arguments de la fonction. Un fichier d'entête devrait être commenté largement afin qu'un ou deux ans plus tard vous puissiez en déchiffrer le contenu. Dans un fichier plus long, il est bon d'écrire

des informations quant à la version, l'adresse de contact, un commentaire général décrivant les capacités du fichier et une mention d'une licence.

Ensuite, vient le fichier source correspondant, les définitions y sont placés dans l'espace de noms et doivent correspondre exactement à leur prototypes du fichier d'entête.

```
/* my_code.cc */
#include "my_code.h"
using namespace ePiX;

namespace Mine {
  double lorentz_norm(const P& arg)
  {
    double x(arg.x1()), y(arg.x2()), z(arg.x3());
// extrait les coordonnées
    return (y-x)*(y+x) + z*z; // -x^2 + y^2 + z^2
  }
  bool spacelike(const P& arg)
  {
    return (lorentz_norm(arg) > 0);//vrai s'il y a inégalité
  }
} // end of namespace
```

Des copies de ces fichiers sont inclus avec le code source afin que vous meniez vos expériences avec. Ensuite le source doit être « compilé », « archivé » et « indexé ». Dans les commandes suivantes, le signe % est l'invite du système.

```
% g++ -c my_code.cc
% ar -ru libcustom.a my_code.o
% ranlib libcustom.a
```

Voyez la documentation de votre système pour les détails des options des commandes et ce que fait chaque étape. Pour l'édition des liens (ci-dessous) le nom de la bibliothèque doit commencer par « lib » et avoir l'extension `.a`. Une fois ces étapes franchies avec succès, placez la bibliothèque `libcustom.a` et le fichier d'entête `my_code.h` dans le répertoire de votre projet. Vous êtes prêt à utiliser ce code dans une figure faite avec `ePiX`.

Liaison à l'exécution

Le script `epix` autorise les fichiers d'entrée à être liés à des bibliothèques externes à l'exécution, quand le fichier d'entrée est compilé en un exécutable temporaire.

epix reconnait les options de la ligne de commande et les passe *verbatim* au compilateur. Les options les plus communément utilisées sont celles du type

```
-I<include>     -L<libdir>     -l<lib>
```

Par exemple, pour lier figure.xp à mylibs/libcustom.a, lancez

```
epix -Lmylibs -lcustom figure
```

Les options -I. -L. demande au compilateur de chercher dans le répertoire courant un fichier d'entête et un fichier de bibliothèque. Les options pour le compilateur peuvent apparaitre dans un ordre quelconque mais doivent être placées avant le(s) nom(s) du fichier (des fichiers) d'entrée.

On peut placer les options pour le compilateur dans un fichier de configuration $HOME/.epixrc avec la syntaxe ci-dessus. Une ligne du fichier de configuration contenant un # est un commentaire quelque soit l'endroit ou le # apparait. Si une ligne quelconque qui n'est pas un commentaire ne commence pas avec un tiret le reste du fichier est ignoré silencieusement. Les options de la ligne de commande sont lues avant le fichier de configuration.

Utiliser plusieurs versions

Le script epix lie par défaut à la bibliothèque mathématique du C libm.a et à la bibliothèque d'ePiX libepix.a. L'option de commande -no-defaults efface l'entête et les chemins d'inclusion et retrire libepix.a de la liste des liens. On peut donc utiliser le script pour différentes versions d'ePiX, une caractéristique potentiellement utile si on doit régulièrement compiler d'anciens fichiers source ou si l'on préfère simplement la syntaxe d'une ancienne version.

Pour installer et utiliser la Version 0.8.5 (p. ex.) construisez le package suivant les instructions de son fichier INSTALL mais *n'utilisez pas le makefile pour l'installation.* Au contraire installer manuellement les seuls entête et bibliothèque, en utilisant leur numéro de version :

```
# install -m 644 epix.h /usr/local/include/epix-0.8.5.h
# install -m 644 libepix.a /usr/local/lib/libepix-0.8.5.a
```

On peut utiliser un répertoire non-système au lieu de /usr/local. Pour utiliser l'ancienne version, un fichier source doit inclure (include) le fichier d'entête approprié – qui est identifié par son numéro de version. Pour compiler, entrer une commande telle que :

```
epix --no-defaults -I/usr/local -L/usr/local -lepix-0.8.5 file.c
```

Annexe A

Liberté des logiciels

Les universitaires en général et les mathématiciens en particulier dépendent des logiciels libres dans leur travail. On peut soutenir sans difficulté que les logiciels propriétaires sont contraires à l'éthique universitaire. En mettant de côté la question de l'accès, si on ne sait pas exactement ce qui est dans un programme, on ne peut faire une entière confiance dans ses résultats pas plus qu'on ne peut faire confiance – dans l'optique d'une publication scientifique – aux résultats d'un laboratoire commercial. L'accès au code source n'est pas, toutefois, la seule chose nécessaire. Pour promouvoir la dissémination de l'information, les utilisateurs devraient se voir reconnaitre les quatre grands droits stipulés dans la licence *GNU General Public License* :

`Droit 0` : droit d'exécuter un programme dans un but quelconque ;

`Droit 1` : droit d'étudier le fonctionnement du programme et de l'adapter a ses propres besoins ;

`Droit 2` : droit de redistribuer des copies du programme ;

`Droit 3` : droit d'améliorer le programme et de publier ces améliorations.

De même que l'on ne lie pas les théorèmes à des licences restrictives, je crois que les logiciels que nous utilisons dans notre travail universitaire devraient avoir une licence qui encourage l'ouverture et le partage. Publier un logiciel sous une licence commerciale classique équivaut – pour moi – à publier un théorème en en gardant la démonstration secrète et en faisant payer des droits pour toute citation qu'on en ferait. Publier simplement le code source sans autoriser l'utilisateur à le modifier pour ses propres besoins revient à publier une démonstration mais à interdire aux lecteurs d'en utiliser les idées dans leurs propres travaux.

Le but ultime d'un logiciel est de nous permettre d'être productifs et créatifs. J'espère que ce modeste programme est, conjointement avec les efforts bien plus grands d'autres – spécialement Donald Knuth, Richard Stallman et les nombreuses personnes qui ont contribué à la création de LaTeX et de ses extensions –, utile pour vous dans votre travail mathématique.

Rendez visite à la *Free Software Foundation* sur `http://www.fsf.org` (ou sur `http://fsffrance.org/` pour une version francophone) pour en savoir plus sur le Logiciel Libre et comment vous pouvez contribuer à son développement et son adoption.

Annexe B

Remerciements

`ePiX` est construit sur le travail de nombreuses personnes – dont malheureusement je ne connais pas la plupart. Les personnes qui suivent ont contribué, quelque fois sans le savoir mais toujours généreusement :

Infrastructure Donald Knuth, Conrad Kwok, Leslie Lamport, Tim Morgan, Piet van Oostrum, Sunil Podar, Richard Stallman, Herbert Voss, Timothy van Zandt

Améliorations Jay Belanger, Robin Blume-Kohout, Julian Gilbey, Svend Daugård Pedersen, Andrew Sterian

Portage et *packaging* Steven Bellenot (Fink) ; Julian Gilbey (Debian) ; Tsuguru Kato (FreeBSD) ; Danny van Dyk, Olivier Fisette, Marcus Hanwell, Peter Johanson, and Patrick Kursawe (Gentoo) ; Guido Gonzato (RPM)

Débogage, conseil et autre assistance Jay Belanger, Felipe Paulo Guazzi Bergo, Karl Berry, Robin Blume-Kohout, Patrick Cousot, Stephen Gibson, Julian Gilbey, Dov Grobgeld, Bob Grover, Jim Hefferon, Jacques L'helgoual, Yvon Henel, Hartmut Henkel, Herng-Jeng Jou, Walter Kehowski, Kevin McCormick, Ross Moore, Thorsten Riess, Alan Sill, Neel Smith, Michael Somos, Andrew Sterian, Ryszard Tanas, Kai Trukenmueller, Torbjorn Vik, Wenguang Wang, Gabe Weaver, Mariusz Wodzicki

Bibliographie

[1] B. Kernighan and D. Ritchie, *The `C` Programming Language*, Second Ed., Prentice-Hall Software Series, 1988

[2] B. Kernighan et D. Ritchie, *Le langage `C` : Norme ANSI*, 2e Éd., Dunod, 2004, coll. « Sciences sup », ISBN-10 : `2100487345`, ISBN-13 : `978-2100487349`

[3] S. Loosemore, R. M. Stallman, et. al., *The GNU `C` Library Reference Manual*, GNU Press, 2004.

[4] K. Reckdahl, *Using Imported Graphics in LaTeX2e*, Version 2.0, whitepaper, Dec. 15, 1997

[5] B. Stroustrup, *The `C++` Programming Language*, Special Ed., Addison-Wesley, 1997

[6] B. Stroustrup, *Le langage `C++`, édition revue et corrigée*, Pearson Education ; Éd. revue et corrigée (20 mars 2003), Coll. « Informatique » ISBN-10 : `2744070033`, ISBN-13 : `978-2744070037`

[7] T. van Zandt, *PSTricks : PostScript Macros for Generic TeX*, Version 0.93a, whitepaper, Mar. 12, 1993

Annexe C

Glossaire

On trouvera ici la liste du vocabulaire spécifique de l'original avec les équivalents adoptés par le traducteur.

anglais	*français*
	A — E
assignment	affectation
bounding box	boite-cadre
bubble (lens)	bulle (objectif)
built-in	intégré
calculus	analyse (mathématique)
camera	caméra
Cartesian coordinate system	repère cartésien
clipping	rognage
clip (*vb*)	rogner
conditional statement	instruction conditionnelle
cropping	cadrage
crop (*vb*)	cadrer
dashed	tireté
density	densité
depth	profondeur
domain	domaine
dot	point
dotted	pointillé
eye	œil
	F — O
false	faux
filling	remplissage

suite…

anglais	*français*
fisheye (lens)	grand angulaire (objectif)
floatting point	décimal
frame	repère
function call (`C++`)	appel de fonction
grid	quadrillage
label	étiquette
lens	objectif
loop	boucle
map	application
marker	marque
mesh	maillage
namespace	espace de noms
offset	déplacement
operating system	système d'exploitation
P — Z	
parsing	analyse lexicale
path	chemin
plot (*vb*)	représenter (graphiquement)
plotting	graphe
pre-processing	prétraitement
pre-processor	précompilateur
range (camera)	plage
range	domaine, étendue
scenery	paysage
sea	mer
seed	graine
shaded	ombrée (surface)
shadow	ombre
sky	ciel
solid	plein
statement (`C++`)	instruction
target	cible
template	gabarit
true	vrai
typeset (*vb*)	composer
viewpoint	point de vue
width (path)	épaisseur (chemin)
working state	espace de travail

Index des commandes

(x_max,y_max), 15
(x_min,y_min), 15
*, *52*, *53*
\\, *22*
*=, *53*
+, *53*
+=, *53*
#include, *25*
%, *52*
&, *52*

aarow, 29
Acos(), 38
acosh(), 38
add(), 59, *59*
arc, 18
arc(), *13*, *18*, *51*
arc_arrow(), *51*
arrow, 29
arrow_camber(), 29
arrow_fill(), 29
arrow_ratio(), 29
arrow_width(), 29
Asin(), 38
Atan(), 38
atan2(u,x), 38
avg(), 33

b, 23
back, 63, 64
back_latitude(), 63
bbox(), 21
begin(), *13*, 26
bold, 19
bold(), *13*, *20*, 46
bool, 36
bounding_box(), *13*, *15*
box(), 21

camera.at(), *19*
camera.focus(), *19*
camera.look_at(), *19*
camera.range(), *19*
camera.rotate_sea(), *19*
cb(), 38
circ(), 21
circle, *17*
circle (classe), 51, 53
circle(), 53, *53*
circle::draw(), *39*
cis(), *52*
clip(bool), 45
cmyk, 20, 46
column(), 33, *34*
const, 40
convert, 64
coords, 58
Cos(), 38
Cot(), 38
covar(), 33
crop(bool), 45
crop_box(), 46
cropbox(pt1,pt2), 45
Csc(), 38
cyl(), *16*
cymk(), *21*

dart, 29
dart_field(), *61*
dashed, 20
dashed(double), 46
dash_fill(), *20*
dash_length(), *20*
dash_length(), 46
data_bins (classe), 33
data_file (classe), 32
data_file, *34*
ddot(), 21
degrees(), 16, 46
Deriv (classe), 60, *60*
Deriv(), *61*
disk_line(), *63*
domain, *55, 56*
domain_list, *59*
dot(), 21, 33, *53*
dot_size(diam), 22
dot_sep(), *20*
dot_sep(), 46
dot_size(), *20*
dotted, 20
dotted(), 46
double, 12, 13, *13*, 36
double&, *40*
draw, 51
draw(), 17, 54, *59*

E_1, *16*
eepic, 9, 19
elaps, 11
elaps, 5, 9, 23, 65
ellipse, 18
ellipse(), *18*, *50*
else, *28*
end(), *13*, 26
envelope(), *60*
\ePiX, 10
epix, 5, 9
eps, 9
erf(), 38
eval(), *60*
exp(), 38

factorial() C++ , *37*
fill(), 21
fill(false), 21
flix, 5, 9, 64
flow(), *61*
focus(), 45
font_face(), *24*
font_size(), *24*
fractal(), *62*
frame (classe), 52
frame(), *52*
front, 64
front_cube(), *63*
front_dodeca(), *63*
front_icosa(), *63*
front_octa(), *63*
front_tetra(), *63*
front_triangle(), *63*
front_arc(), *63*
front_arc2(), *63*
front_line(), *63*

gray(), 21
grid(), 30

h_axis(), *13*
h_axis_labels(), *13*, 30
histogram(), *34*
hyperbolic_line(), *63*

id(), 38
if, *28*
inf(), *60*
int, 12, 36
Integral (classe), 60, *60*
Integral(), *60, 61*

J(), *52*

j0(), 38
j1(), 38
jn(), 38
jn(5,)(), 38

l, 23
label(), *13*, *22*, 23
label_angle(), 23
label_angle(), 46
laps, 5, 9, 10
latitude(), *63*
LEFT, 61
left(), *60*
Line, 54
line, 13
line(), *18*, 50, *50*
log(), 38
log2(), 38
longitude(), *63*
LOWER, 61

M_E, 38
M_PI, 38
M_PI_2, 38
magenta(), *21*
main, 26
main(), *13*, 37
marker(), *22*
masklabel, 23
mesh(), *55*, 56
midpoint(), *53*
MIDPT, 61

newton(), *60*
none, 48
normalsize, 24

ode_plot(), *61*
offset, 14
offset(), *14*, 30

P, 12, 36, *52*
P (classe), 51
P(), *16*
P(x,y), 36
P(x,y,z), 36
Path (classe), 48–51
path, 47
path (classe), 49
path(), 48
path::draw(), 49
pdf, 9
pen(), 19, *20*, 46
picture(), *13, 14*
plain, 19
plain(), *20*, 46
plane, *17*, 54
plane (classe), 51
plot, 41, 54, 57
plot(), 31–33, *55*
plot_N(), *64*
plot_R(), *64*
plot_S(), *64*
png, 9
polar(), *16*, *52*
polar_grid(), 30
polarplot(), 31
pole(), 54
polygon(), *18*, 48
polyline(), *18*, 48
pow(), 38
precision(), 33
prim(), *60*
primary, 46
proj(), 38
proj1(), 38
proj2(), 38
ps2epsi, 66
psline, 47
psset(), 48, *48*
pstcol, 23
pstricks, 10

pt(), 49

quad(), *18*, *50*

r, 23
radians(), 46
range(), 45
read(), *34*
recip(), 38
rect, 18
rect(), *13*, *18*, 50, *50*
red(), *21*
regression(), 33, *34*
resize(), *56*, *57*
return, *26*
revolutions(), 46
rgb, 20
rgb(), *21*, 46
riemann_sum(), *61*
RIGHT, 61
right(), *60*
ring(), 21
rotating, 10, 23

scenery, *59*
scenery (classe), 59
Sec(), 38
segment, 52
segment (classe), 51, 52
segment(), *53*
sgn(), 38
shadeplot(), 31
Sin(), 38
sinx(), 38
slice1(), *59*
slice2(), *55*, *59*
slice3(), *59*
slices, *57*
slices1(), *57*
slope_field(), *61*
solid, 20
solid(), 46
sph, 16
sph(), *16*
spher (classe), 53
sphere, *17*
sphere (classe), 51
sphere(), 54
spline(), *18*, *50*
spot(), 21
std::cout, *48*
sup(), *60*
surface, 54, 57
surface(), 58, *59*
surface_rev(), *58*

t, 23
Tan(), 38
tan_field(), *60*
tan_line(), *60*
tgamma(), 38
thinlines, 19
tix, 64
transform(), 33, *34*
TRAP, 61
triangle(), *18*, *50*

unitlength(), *14*, 19
UPPER, 61
use_pstricks(), *48*
using namespace, *25*

v_axis(), *13*
vector_field(), *61*
void, 37, 69

while(), *28*
write(), 33, *34*

x1(), *52*
x_size, 15

y0(), 38
y_size, 15

zero(), 38

Index

Analyse, 59–62
Angle (unité), 16, 18, 23, 46
Angulaire (mode), 16, 38, 46
Animation, 5, 64–65
Arc
  d'ellipse, 18, 50
  de cercle, 18, 50
Argument (avec valeur) par défaut, 12
Axes, 29–30
  coches, 29
  étiquette, 29

Boite-cadre, 14–15, 18, 29, 45

Cadrage, 45–46
  boite de, 45
Caméra, 18–19, *43–45*, 68
  manipulation de la, 18, 44–45
  objectif, 43–44
  point de vue de la, 18
Cercle, 17, 53
  arguments par défaut, 53
  operations sur, 53
Chemin, 15, 21, *48–51*
  en PSTricks, 47
  fractal, 62
  opérations sur, 49
  rempli, 21
  style, 19–20, 32, 46
  style par défaut, 19
  sur la sphère, 64
  épaisseur, 19, 46
Ciel (vecteur), 45
Classes, 39
Commentaires, *26*
Conditionnelle (instruction), 35
Contrib (extension), 6, 30
Couleur, 20–48
  et extensions LaTeX, 47
  remplissage, 47–48, 50
  teinte, 47–48
Courbes de niveau, 61

Déplacement
  étiquette, 23
Domaine, *54–57*
  et graphe, 55
  retaille, 56
  tranchage, 56
Données (fichier de), *31–34*
Droite, 17, 49, 54
Décalage
  image, 14
Déplacement
  étiquette, 30

Écran, 15, 18, 43–45
Effacement d'objet caché, 59, 64, 68–69
Ellipse, 18
  arc de, 18
`emacs`, 5, 6, 9, 11, 24, 64
Enveloppe, 60
Équations différentielles, 61
Erreur (type d') `C++`, 37
Étiquette, *21–24*, 42

alignement, 21, 23
axe, 29–30
déplacement de, 23
doubles quotes, 23
le \ dans les, 23
pivotées, 23
police dans les, 24
rotation de, 46
Euclide (algorithme), 27
Euler (méthode), 37, 61
Exception `C++`, 37

Fichier d'entrée
exemple complet, 12
format de, 25, *42–43*
Figure
environnement autonome, 27
Flèche, 17, *28–29*, 51
forme de la tête, 29
Fonction, 12, 25, 35, *36–41*
algorithme d'Euclide, 27
argument par défaut, 12
définitions imbriquées, 26
exemple de définition, 26
mathématique, *37–38*
membre d'une classe, 39
noms, 25
retournant `void`, 37
récursive, 37
surchargée, 41
syntaxe d'appel, 39
tracé, 31
type d'argument, 12
valeur de retour, 12, 26
Format d'image, 14–15, 21
Free software, 4, 73–74

Graphe, 30–34, *54–59*
analyse, 60–62
champ de vecteurs, 60–62
courbes de niveau, 61
données, 31–34
en maille, 56
et domaine, 55
logarithmique, *31–34*
polaire, 31
sphérique, 63–64
surface, *57–59*, 68–69

Histogrammes, 33

Image
décalage, 14
taille, 14
Installation, 4–7, 65–67
Instructions conditionnelles, 27
Interface graphique, 9
Intersection, 52
générique, 52
table des types, 52

Logiciels libres, 4, 73–74

Mac OS X, 4
Marque (types de), 21–22, 24, 32
Mer (vecteur), 45
Mode angulaire, 16, 38, 46

Newton (méthode), 60

Œil (vecteur), 44

Papier millimétré, 30
Plan, 17, 54
Point, 15–17
comme déplacement, 16
comme emplacement, 16
comme point de vue, 18
coordonnées, 16
d'un fichier de données, 31
nommé, 15
operations algébriques sur les, 16
type de données, *51–52*
Point de vue, 18

Pointeur, 36
    et *polyline*, 48
Police, 24
Programmation orientée objet, 39
Projection
    conique, 18, 44
    orthogonale, 44
    parallèle, 18
    radiale, 44, 64
    stéréographique, 44, 64
Préambule, 11, 14, 46
PSTricks, 20, 29, *47–48*, 50

Quadrillage, 30

Rectangle, 18
    quadrillage, 30
Remplissage, 19, 21, 46, 50
    couleur, 47–48, 50
Rognage, 45–46
    boite de, 45
Récursivité (en `C++`), 37

Simpson (règle), 37
Source (fichier)
    commentaires dans, 43
    compilation conditionnelle, 42
Sphère, 17, 53
    graphe sur la, 63
Système d'exploitation Windows, 5–6

Tangente, 60

Unité, 14
Unité d'angle, 16, 18, 23, 46

Variable, 25
    locale, 41
    noms, 25, *36*
    pointeur, 36
    référence, 39
    type de, 36
Vecteur
    ciel, 18, 45
    mer, 18, 45
    œil, 18, 44