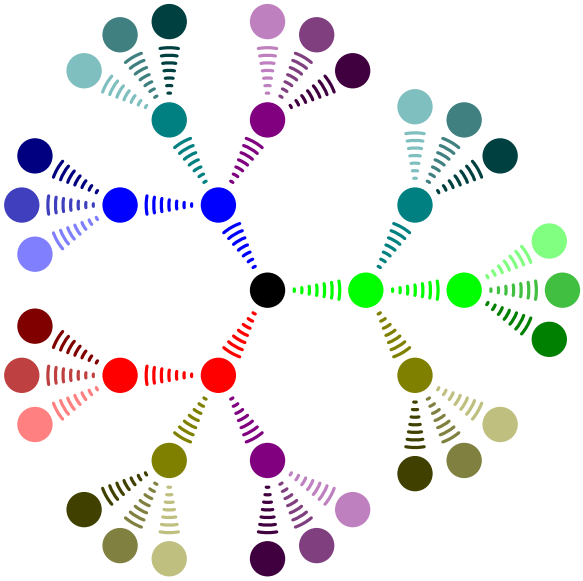
Ti*k*Z et PGF

Manuel pour la version 1.01



```
\tikzstyle{level 1}=[sibling angle=120]
\tikzstyle{level 2}=[sibling angle=60]
\tikzstyle{level 3}=[sibling angle=30]
\tikzstyle{every node}=[fill]
\tikzstyle{edge from parent}=[snake=expanding waves,segment length=1mm,segment angle=10,draw]

\tikz [grow cyclic,shape=circle,very thick,level distance=13mm,cap=round]
  \node {} child [color=\A] foreach \A in {red,green,blue}
    { node {} child [color=\A!50!\B] foreach \B in {red,green,blue}
      { node {} child [color=\A!50!\B!50!\C] foreach \C in {black,gray,white}
        { node {} }
      }
    };
```

Für meinen Vater, damit er noch viele schöne TEX-Graphiken erschaffen kann.

Les extensions Ti*k*Z et PGF
Manuel pour la version 1.01*

http ://sourceforge.net/projects/pgf

Till Tantau
tantau@users.sourceforge.net

30 avril 2006

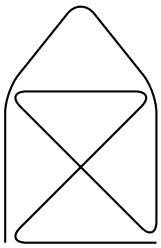
Table des matières

*Traduction française par le TeXnicien de surface. Je remercie les relecteurs pour leur aide précieuse : François Giron, Arnaud Schmittbuhl, Denis Vergès.

Première partie

Pour démarrer

Cette partie est conçue pour vous aider à démarrer avec l'extension PGF. D'abord, on explique le processus d'installation ; toutefois, ce système est, normalement, déjà installé sur votre machine et cette partie peut-être souvent sautée. Ensuite, on donne un court tutoriel pour expliquer les commandes les plus souvent utilisées et les concepts de Ti*k*Z, sans aller jusqu'aux détails splendides. À la fin de la section vous trouverez quelques indications, qu'on espère utiles, pour créer de « bons » graphiques en général. Les informations de cette section ne sont pas spécifiques à PGF.



```
\tikz \draw[thick,rounded corners=8pt]
  (0,0) -- (0,2) -- (1,3.25) -- (2,2) -- (2,0) -- (0,2) -- (2,2) -- (0,0) -- (2,0);
```

1 Introduction

L'extension PGF, où « PGF » est sensé signifier « format graphique portable » (*portable graphics format*), est une extension pour la création de graphiques « en ligne ». L'extension définit un certain nombre de commandes TEX qui dessinent des graphiques. Par exemple, le code `\tikz \draw (0pt,0pt) -- (20pt,6pt);` produit le segment  et le code `\tikz \fill[orange](1ex,1ex) circle (1ex);` produit  .

Dans un sens, lorsque vous utilisez PGF vous « programmez » votre graphique comme vous « programmez » votre document en utilisant TEX. Cela signifie que vous obtenez les avantages de « l'approche TEXienne de la typographie » pour vos graphiques aussi : création rapide de graphiques simples, positionnement précis, utilisation de macros, typographie souvent meilleure. Vous héritez également de tous les inconvénients : apprentissage à pente rude, pas de WYSIWYG, nécessité d'un temps long de recompilation pour de petits changements et le fait que le code ne « montre » pas vraiment l'aspect qu'auront les choses.

1.1 Structure du système

Le système de PGF comporte plusieurs couches :

Couche système : Cette couche fournit une abstraction complète de ce qui se passe « dans le pilote ». Le pilote est un programme comme `dvips` ou `dvipdfm` qui prend un fichier `.dvi` en entrée et crée un fichier `.ps` ou `.pdf`. (On considère aussi le programme `pdftex` comme un pilote même s'il ne prend pas de fichier `.dvi` en entrée. Ce n'est pas grave.) Chaque pilote a sa propre syntaxe pour la création de graphiques, source de maux de tête pour quiconque veut créer des graphiques de manière portable. La couche système de PGF permet de s'abstraire de ces différences. Par exemple, la commande système `\pgfsys@lineto{10pt}{10pt}` étend le chemin courant aux coordonnées $(10\text{pt}, 10\text{pt})$ de la `{pgfpicture}` courante. Suivant qu'on utilisera `dvips`, `dvipdfm` ou `pdftex` pour compiler le document, cette commande système sera convertie en différentes commandes `\special`.

La couche système est aussi minimaliste que possible puisque chaque commande additionnelle crée plus de travail pour le port de PGF sur un nouveau pilote. Actuellement, seuls les pilotes produisant du PostScript ou du PDF sont gérés et seulement quelques uns d'entre eux (ce qui fait que le nom de format graphique *portable* est actuellement un peu une vantardise). Toutefois, en principe, la couche système pourrait être portée facilement vers de nombreux autres pilotes. Il devrait même être possible de produire, par exemple, du SVG en collaboration avec TEX4HT.

En temps qu'utilisateur vous n'utiliserez pas directement la couche système.

Couche de base : La couche de base fournit un jeu de commandes de base qui vous permettent de produire des graphiques complexes plus simplement qu'en utilisant la couche système directement. Par exemple, la couche système ne fournit aucune commande pour créer des cercles puisque les cercles peuvent être composés (enfin, presque) à partir des courbes de Bézier, plus fondamentales. Toutefois, en tant qu'utilisateur vous aimeriez certainement avoir une commande simple de création de cercles (moi, en tout cas, j'aimerais) au lieu d'avoir à écrire une demi-page de coordonnées de points de définition de courbes de Bézier. Aussi, la couche de base fournit la commande `\pgfpathcircle` qui engendre les coordonnées nécessaires pour vous.

La couche de base est composée d'un noyau (*core*) comportant plusieurs extensions interdépendantes qu'on ne peut charger qu'*en bloc*[1], et d'extensions additionnelles qui étendent le noyau avec des commandes plus spécialisées comme celles concernant la gestion des nœuds ou l'interface de tracé de courbe. Par exemple, l'extension BEAMER utilise le noyau mais pas toutes les extensions additionnelles de la couche de base.

Couche d'interface : Une interface (il peut en exister plusieurs) est un ensemble de commandes ou une syntaxe spéciale qui rend plus facile l'utilisation de la couche de base. L'utilisation directe de la couche de base conduirait à un code souvent trop « verbeux ». Par exemple, pour dessiner un simple triangle, vous pourriez avoir besoin d'au moins cinq commandes en utilisant la couche de base : une pour commencer un chemin au premier sommet du triangle, une pour étendre ce chemin jusqu'au deuxième sommet, une pour atteindre le troisième, une pour fermer le chemin et une pour peindre le triangle (et non pas le remplir). Avec l'interface `tikz` tout cela se réduit à l'unique commande « à la METAFONT » :

```
\draw (0,0) -- (1,0) -- (1,1) -- cycle;
```

Il y a plusieurs interfaces :

1. En français dans le texte

– Ti*k*Z est l'interface « naturelle » de PGF. Elle vous donne accès à toutes les caractéristiques de PGF mais elle est sensée être facile à utiliser. La syntaxe est un mélange de celle de METAFONT et de PSTRICKS et de quelques unes de mes idées. Cette interface n'est *ni* une couche complète de compatibilité avec METAFONT *ni* une couche de compatibilité avec PSTRICKS et n'est pas destinée à le devenir.
– L'interface `pgfpict2e` réimplante l'environnement `{picture}` et les commandes telles que `\line` ou `\vector` de LaTeX en utilisant la couche de base de PGF. Cette couche n'est pas vraiment « nécessaire » puisque l'extension `pict2e.sty` réalise un travail au moins aussi bon pour une réimplantation de l'environnement `{picture}`. En fait, l'idée qui est derrière la création de cette extension est de faire une démonstration simple de la manière d'implanter une interface.

Il aurait été possible de créer une interface `pgftricks` qui aurait lié les commandes de PSTRICKS aux commandes de PGF. Toutefois, je ne l'ai pas fait ; et même si c'était totalement implanté, de nombreuses choses qui fonctionnent dans PSTRICKS ne marcheraient pas ; de fait certaines commandes de PSTRICKS s'appuient trop sur des trucs de PostScript. Néanmoins une telle extension pourrait présenter un intérêt dans certaines circonstances.

En tant qu'utilisateur de PGF vous utiliserez les commandes de l'interface plus, peut-être, quelques commandes de la couche de base. Pour cette raison, ce manuel présente d'abord les interfaces puis la couche de base et, enfin, la couche système.

1.2 Comparaison avec d'autres extensions graphiques

Il y avait deux raisons principales pour créer PGF :

1. L'environnement standard de LaTeX `{picture}` n'est pas assez puissant pour créer autre chose que des graphiques très simples. Cela n'est certainement pas dû à un manque de connaissances ou d'imagination de la part du (des) créateur(s) de LaTeX. Mais c'est le prix à payer pour la portabilité de l'environnement `{picture}` : il fonctionne avec tous les pilotes travaillant en arrière-plan.
2. L'extension `{pstricks}` est sans aucun doute assez puissante pour créer tout graphique imaginable mais elle n'est pas du tout portable. Plus important encore elle ne fonctionne ni avec `pdftex` ni avec tout autre pilote produisant autre chose que du code PostScript.

L'extension PGF est un compromis entre portabilité et expressivité. Elle n'est pas aussi portable que `{picture}` et peut-être pas aussi puissante que `{pspicture}`. Toutefois, elle est plus puissante que `{picture}` et plus portable que `{pspicture}`.

1.3 Utilitaires : Gestion de pages

L'extension PGF comprend une sous-extension spéciale appelée `pgfpages` qu'on utilise pour assembler plusieurs pages sur une seule. Cette extension ne concerne pas en fait la création de graphique mais c'est néanmoins une partie de PGF, principalement parce que son implantation utilise beaucoup PGF.

La sous-extension `pgfpages` fournit des commandes pour assembler plusieurs « pages virtuelles » sur une seule « page physique ». L'idée est que, à chaque fois que TeX a une page prête à être sortie (*shipout*), `pgfpages` interrompt la sortie et place la page à sortir dans une boite spéciale. Lorsque assez de « pages virtuelles » ont été accumulées de cette façon, elles sont réduites et positionnées sur la « page physique » qui est alors « vraiment » sortie. Ce mécanisme vous permet de créer une version « deux pages sur une » d'un document directement depuis LaTeX sans utiliser de programmes extérieurs.

Toutefois, `pgfpages` peut faire bien plus que cela. Vous pouvez l'utiliser pour placer des logos ou des fonds de page, imprimer 16 pages sur une page, ajouter des bordures, et plus encore.

1.4 Comment lire ce manuel

Ce manuel décrit à la fois la conception du système de PGF et son utilisation. Cette organisation est plus ou moins conviviale. Les commandes et sous-extensions les plus facilement et les plus souvent utilisées sont décrites d'abord, les caractéristiques de plus bas niveau ou ésotériques sont discutées ensuite.

Si vous n'avez pas encore installé PGF, veuillez lire d'abord la section « installation ». Ensuite ce pourrait être une bonne idée de lire le mode d'emploi. Enfin, vous souhaiterez peut-être parcourir la description de Ti*k*Z. En général vous n'aurez pas besoin de lire les sections concernant la couche de base. Vous n'aurez besoin de lire la partie sur la couche système que si vous avez l'intention d'écrire votre propre interface ou de porter PGF vers un nouveau pilote.

Les commandes et environnements « publiques » fournis par l'extension `pgf` sont décrits dans tout le texte. Dans chacune de ces descriptions, la commande, l'environnement, ou l'option décrit est imprimé en rouge. Le texte en vert est optionnel et peut être passé.

1.5 Obtenir de l'aide

Quand vous avez besoin d'aide sur PGF et Ti*k*Z, veuillez appliquer ce qui suit :

1. Lisez le manuel, à tout le moins la partie qui a affaire avec votre problème.
2. Si cela ne résout pas votre problème, essayez de jeter un œil à la page de développement de PGF et Ti*k*Z sur sourceforge (voyez le titre de ce document). Peut-être quelqu'un a-t-il déjà signalé un problème semblable et que quelqu'un y a trouvé une solution.
3. Sur le site web vous trouverez de nombreux groupes de discussion où vous pourrez trouver de l'aide. Là, vous pouvez écrire à des groupes d'aide, envoyer des rapports de bogue, rejoindre des listes de diffusion, etc.
4. Avant d'envoyer un rapport de bogue, spécialement un rapport de bogue concernant l'installation, assurez-vous qu'il s'agit bien d'un bogue. En particulier, jetez un œil au fichier `.log` produit quand vous TEXez vos fichiers. Ce fichier `.log` devrait montrer que tous les fichiers convenables ont été chargés depuis les bons répertoires. Presque tous les problèmes d'installation peuvent être résolus en regardant le fichier `.log`.
5. *En dernier ressort* vous pouvez essayer de m'envoyer (à moi, l'auteur) un courriel. Recevoir des courriels ne me dérange pas mais j'en reçois simplement bien trop. De ce fait, je ne peux pas garantir que je répondrai en temps et en heure, voire même que je répondrai simplement, à votre courriel. Vos chances de voir votre problème résolu sont un peu plus grandes si vous écrivez à la liste de diffusion de PGF (naturellement, je lis cette liste et je réponds aux questions quand j'en ai le temps).
6. Veuillez ne pas me téléphoner au bureau. Si vous avez besoin d'une aide en ligne, achetez un produit commercial.

2 Installation

Il y a plusieurs manières d'installer PGF qui dépendent de votre système et de vos besoins. Vous pourriez de plus avoir besoin d'installer d'autres extensions également (voir ci-dessous). Avant d'installer, vous pourriez souhaiter revoir la licence GPL sous laquelle cette extension est distribuée, voyez la section **??**, p. **??**.

En général, cette extension est déjà installée sur votre système. Naturellement, dans ce cas vous n'avez pas besoin du tout de vous en faire à propos du processus d'installation et vous pouvez passer le reste de cette section.

2.1 Versions de l'extension et des pilotes

Cette documentation fait partie de la version 1.01 de l'extension PGF. Afin de faire fonctionner PGF vous aurez besoin d'une installation raisonnablement récente de TEX. Si vous utilisez LATEX, vous aurez besoin que les extensions suivantes soient installées (des versions plus récentes devraient marcher aussi) :

– `xcolor` version 2.00.
– `xkeyval` version 1.8, si vous désirez utiliser Ti*k*Z.

Avec « plain TEX », on n'a pas besoin de `xcolor` mais bien évidemment on n'a pas ces fonctionnalités (complètes).

À ce jour, PGF est compatible avec les pilotes suivants :

– `pdftex` version 0.14 ou supérieure. Les versions antérieures ne marchent pas.
– `dvips` version 5.94a ou supérieure. Il se peut que les versions antérieures marchent.
– `dvipdfm` version 0.13.2c ou supérieure. Il se peut que les versions antérieures marchent.
– `tex4ht` version 2003-05-05 ou supérieure. Il se peut que les versions antérieures marchent.
– `vtex` version 8.46a ou supérieure. Il se peut que les versions antérieures marchent.
– `textures` version 2.1 ou supérieure. Il se peut que les versions antérieures marchent.

À ce jour, PGF est compatible avec les formats suivant :

– `latex` avec toutes ses fonctionnalités ;
– `plain` avec toutes ses fonctionnalités sauf l'inclusion de graphiques qui ne marche que pour pdfTEX.
– `context` devrait marcher comme `plain` mais je ne l'ai pas essayé.

Pour plus de détails, voyez la section 5, p. 39.

2.2 Installer des archives

Je ne crée ni ne gère d'archives de PGF mais heureusement d'autres gens gentils le font. Je ne peux pas donner d'instructions détaillées sur l'installation de ces paquets, puisque je ne les gère pas, mais je *peux* vous dire où les trouver. Si vous avez un problème avec l'installation, vous devriez d'abord jeter un œil sur la page Debian ou la page MikTEX.

2.2.1 Debian

La commande « `aptitude install pgf` » devrait suffire. Asseyez-vous et détendez-vous. Dans le détail, les paquets suivants sont installés :

`http://packages.debian.org/pgf`
`http://packages.debian.org/latex-xcolor`

2.2.2 MiKTeX

Avec MikTEX, utilisez l'assistant de mise-à-jour (*update wizard*) pour installer les (dernières versions des) extensions appelées `pgf`, `xcolor` et `xkeyval`.

2.3 Installation dans une arborescence texmf

Pour une installation permanente, on place les fichiers de l'extension PGF dans une arborescence `texmf` adéquate.

Lorsque l'on demande à TEX d'utiliser une certaine classe ou extension, il cherche normalement les fichiers nécessaires dans ce que l'on appelle une arborescence `texmf`. Par défaut, TEX cherche dans trois arborescences `texmf` différentes :

– L'arborescence `texmf` principale (ou *racine*) qui est placée sous `/usr/share/texmf/` ou `c:\texmf\` ou quelque chose de semblable.

– L'arborescence locale placée d'habitude sous `/usr/local/share/texmf/` ou `c:\localtexmf\` ou quelque chose de semblable.
– Votre arborescence `texmf` personnelle placée dans votre répertoire personnel (*home*) sous `~/texmf/` ou `~/Library/texmf/`.

Vous devriez installer les extensions soit dans l'arborescence locale ou dans votre arborescence personnelle suivant que vous avez ou non les droits d'écriture sur l'arborescence locale. L'installation dans l'arborescence principale peut créer des problèmes puisque une mise-à-jour de toute l'installation TEX remplacera tout cette arborescence.

2.3.1 Installer tout ensemble

Une fois l'arborescence adéquate choisie, on doit décider si on veut installer PGF de telle façon que « tous les fichiers soient au même endroit » ou si on veut être conforme à la TDS, où TDS signifie « structure des répertoires de TEX » (*TEX directory structure*).

Si vous voulez conserver « tout ensemble » dans l'arborescence `texmf` que vous avez choisie vous devez créer un sous-sous-répertoire nommé `texmf/tex/generic/pgf` ou `texmf/tex/generic/pgf-1.01` si vous préférez. Puis vous placerez tous les fichiers de l'extension `pgf` dans ce répertoire. Enfin vous reconstruirez les bases de noms de fichiers de TEX. Cela se fait en lançant la commande `texhash` ou `mktexlsr` (ce sont les mêmes). Dans MikTEX il y a une option du menu pour ce faire [2].

2.3.2 Installation conforme à la TDS

Alors que l'installation décrite ci-dessus est la plus « naturelle » et bien que je voudrais la recommander puisqu'elle facilite la mise-à-jour et la gestion de l'extension PGF elle n'est pas conforme à la TDS. Si vous voulez être conforme à la TDS, faites ce qui suit (si vous ne savez pas ce que signifie « conforme à la TDS », vous ne voulez probablement pas être conforme à la TDS).

L'archive `.tar` de l'extension `pgf` contient les fichiers et répertoires suivants à sa racine : `README`, `doc`, `generic`, `plain` et `latex`. Vous devriez « fusionner » chacun de ces quatre répertoires avec les répertoires suivants `texmf/doc`, `texmf/tex/generic`, `texmf/tex/plain` et `texmf/tex/latex`. Par exemple, dans l'archive `.tar`, le répertoire `doc` ne contient que le répertoire `pgf` et ce répertoire doit être copié comme `texmf/doc/pgf`. Le fichier `README` placé à la racine de l'archive peut être laissé de côté puisqu'il est reproduit en `doc/pgf/README`.

Vous pourriez envisager également de tout placer au même endroit et d'utiliser des liens symboliques pour pointer des répertoires conformes à la TDS sur celui de l'installation centrale.

Pour des explications plus détaillées sur le processus normal d'installation d'extensions, vous pouvez consulter : `http ://www.ctan.org/installationadvice/`. Toutefois, notez que l'extension PGF n'est pas livrée avec un fichier `.ins` (passez simplement cette partie).

2.4 Mise à jour de l'installation

Pour mettre à jour votre installation depuis une version précédente, vous devez simplement remplacer tout ce qui se trouve dans `texmf/tex/generic/pgf` par les fichiers de la nouvelle version (ou bien dans tous les répertoires où `pgf` a été installé si vous avez choisi une installation conforme à la TDS). Le plus simple est de commencer par effacer l'ancienne version et de procéder comme décrit ci-dessus. Parfois, il y a quelques changements dans la syntaxe de certaines commandes d'une version à l'autre. Si les choses qui marchaient ne marchent plus vous devriez jeter un œil à la note de parution (*release note*) et au fichier de changements (*change log*).

2. NdTdS : Lancer MikTeX Options, voir l'onglet « General », utiliser le bouton « Refresh now ».

3 Mode d'emploi : un petit dessin pour les étudiants de Karl

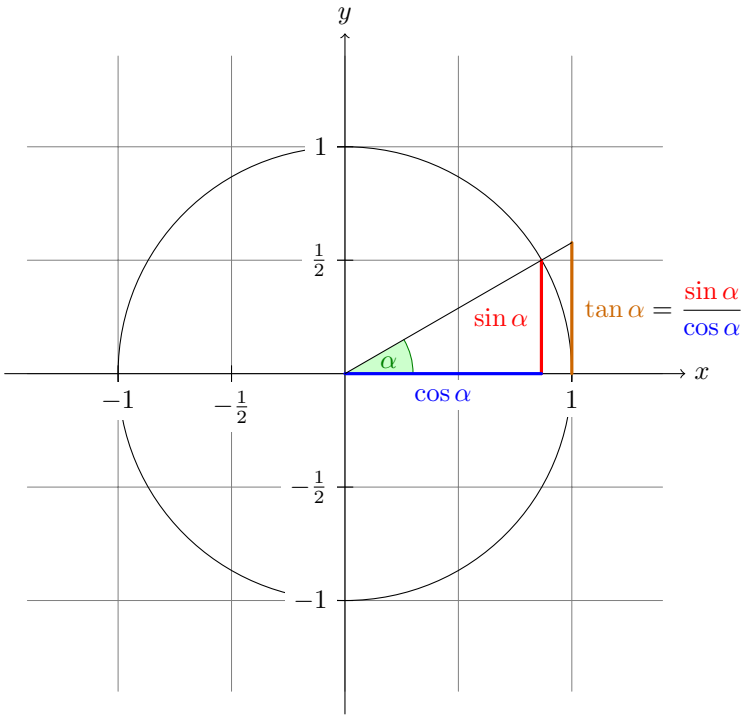
Ce mode d'emploi est conçu pour les nouveaux utilisateurs de PGF et Ti*k*Z. Il ne donne pas une description exhaustive de toutes les fonctionnalités de Ti*k*Z ou de PGF mais seulement de celles dont on peut faire usage directement.

Karl est professeur de mathématiques et chimie en lycée. Il avait l'habitude de créer des graphiques dans ses feuilles d'exercices et d'examens avec l'environnement `{picture}` de LaTeX. Alors que les résultats semblaient acceptables, la création de graphiques s'avérait souvent un processus long. De plus, il y avait une tendance à avoir des problèmes avec les droites qui montraient des angles légèrement incorrects et les cercles qui semblaient être difficiles à obtenir correctement. Naturellement, ses étudiants se contrefichaient de savoir si les droites formaient les angles corrects et ils trouvaient que les examens de Karl étaient trop difficiles même si les graphiques étaient tracés avec soin. Mais Karl n'était jamais entièrement satisfait du résultat.

Le fils de Karl, qui était encore moins satisfait du résultat (il n'avait pas à passer les examens après tout), dit à Karl qu'il pouvait peut-être essayer une nouvelle extension de création de graphique. Portant un peu à confusion, cette extension semblait avoir deux noms : d'abord Karl devait télécharger et installer une extension appelée PGF. Puis il apparaissait qu'il y en avait une autre à l'intérieur, appelée Ti*k*Z, ce qui est sensé vouloir dire « Ti*k*Z ist *kein* Zeichenprogramm[3] ». Karl trouvait tout cela un peu étrange et Ti*k*Z semblait indiquer que cette extension n'était pas ce dont il avait besoin. Toutefois, ayant utilisé les logiciels GNU depuis un bon moment et « GNU n'étant pas Unix », il semblait encore y avoir de l'espoir. Son fils l'assurait que le nom de Ti*k*Z était fait pour prévenir les gens que Ti*k*Z n'est pas un logiciel que l'on peut utiliser pour fait des graphiques avec une souris ou une tablette. C'est plutôt quelque chose comme un « langage pour graphique ».

3.1 Énoncé du problème

Karl veut placer un graphique dans le prochain poly de ses étudiants. En ce moment, il enseigne le sinus et le cosinus. Ce qu'il voudrait faire est quelque chose qui ressemblerait (idéalement) à ceci :



L'angle α vaut 30° dans l'exemple ($\pi/6$ en radians). Le sinus de α, qui est la longueur du segment rouge est

$$\sin\alpha = 1/2.$$

Par le théorème de Pythagore, il vient $\cos^2\alpha + \sin^2\alpha = 1$. Donc la longueur du segment bleu, qui est le cosinus de α, vaut

$$\cos\alpha = \sqrt{1 - 1/4} = \tfrac{1}{2}\sqrt{3}.$$

Cela démontre que $\tan\alpha$, qui est la longueur du segment orange, vaut

$$\tan\alpha = \frac{\sin\alpha}{\cos\alpha} = 1/\sqrt{3}.$$

3.2 Préparer l'environnement

Pour dessiner une figure avec Ti*k*Z on doit au début de celle-ci dire à TeX ou LaTeX que l'on veut commencer une figure. En LaTeX on utilisera l'environnement `{tikzpicture}`, en « plain TeX » on commencera la figure par `\tikzpicture` et on la finira par `\endtikzpicture`.

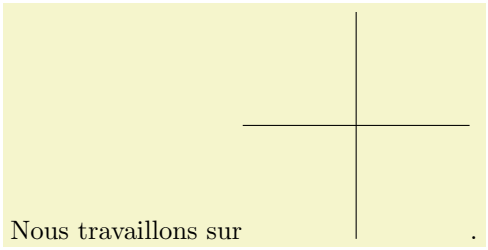
3. NdTdS : Ce qui signifie Ti*k*Z n'est *pas* un programme de dessin.

3.2.1 Préparation de l'environnement en LaTeX

Karl, utilisant LaTeX, créera donc son fichier comme suit :

```
\documentclass{article} % say
\usepackage{tikz}
\begin{document}
Nous travaillons sur
\begin{tikzpicture}
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
\end{tikzpicture}.
\end{document}
```

Une fois le fichier exécuté, c.-à-d. compilé par `pdflatex` ou par `latex` suivi de `dvips`, le résultat contiendra quelque chose qui ressemblera à ceci :




```
Nous travaillons sur
\begin{tikzpicture}
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
\end{tikzpicture}.
```

D'accord, pas encore toute la figure pour l'instant, mais nous avons déjà placé les axes. Enfin, pas tout à fait, mais nous avons des droites qui deviendront des axes. Soudain, Karl a le sentiment poignant que la figure est encore loin.

Regardons ce code de plus prêt. D'abord, on charge l'extension `tikz`. Cette extension est appelée « interface » au système de base de PGF. La couche de base, décrite également dans ce manuel, est quelque peu, disons, basique et, de ce fait, plus difficile à utiliser. L'interface rend les choses plus faciles en fournissant une syntaxe plus simple.

Dans l'environnement il y a deux commandes `\draw`. Elles signifient : « Le chemin, qui est spécifié après la commande et jusqu'au point-virgule, doit être tracé. » Le premier chemin est déterminé par `(-1.5,0) -- (0,1.5)` ce qui signifie « un segment de droite depuis le point à la position $(-1.5, 0)$ jusqu'au point à la position $(0, 1.5)$ ». Ici, les positions sont déterminées dans un repère spécial pour lequel une unité, au départ, vaut 1 cm.

Karl est assez content de voir que l'environnement réserve automatiquement assez d'espace pour contenir toute la figure.

3.2.2 Préparation de l'environnement en Plain TeX

Il se trouve que Gerda, la femme de Karl, qui est aussi professeur de mathématiques, n'utilise pas LaTeX mais plain TeX car elle préfère faire les choses à « l'ancienne ». Elle aussi peut utiliser Ti*k*Z. Au lieu de `\usepackage{tikz}` elle doit écrire `\input tikz.tex`, au lieu de `\begin{tikzpicture}` elle écrit `\tikzpicture` et au lieu de `\end{tikzpicture}` elle écrit `\endtikzpicture`.

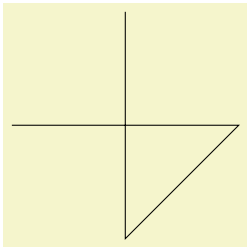
Elle utiliserait donc :

```
%% fichier Plain TeX
\input tikz.tex
\baselineskip=12pt
\hsize=6.3truein
\vsize=8.7truein
Nous travaillons sur
\tikzpicture
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
\endtikzpicture.
\bye
```

Gerda peut compiler ce fichier avec `pdftex` ou bien `tex` suivi de `dvips`. Ti*k*Z détectera automatiquement le pilote utilisé. Si elle désire utiliser `dvipdfm` avec `tex`, elle devra soit modifier le fichier `pdf.cfg` ou écrire `\def\pgfsysdriver{pgfsys-dvipdfm.def}` quelque part *avant* qu'elle ne charge `tikz.tex` ou `pgf.tex`.

3.3 Construction d'un chemin droit

Le bloc de base de toute figure dans Ti*k*Z est le chemin (*path*). Un chemin est une suite de segments de droites ou de courbes qui sont connectés (ce n'est pas le tout de la chose mais laissons les complications pour l'instant). On commence un chemin en déterminant les coordonnées de sa position de départ comme un point, entre parenthèses, comme dans `(0,0)`. On continue par une suite « d'opérations d'extension de chemin ». La plus simple est `--` que l'on a déjà utilisée. Elle doit être suivie d'autres coordonnées et étend le chemin en ligne droite jusqu'à la nouvelle position. Par exemple, si nous transformions les deux chemins des axes en un seul, ceci s'en suivrait :



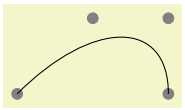
```
\tikz \draw (-1.5,0) -- (1.5,0) -- (0,-1.5) -- (0,1.5);
```

Karl est un peu troublé par le fait qu'il n'y a pas d'environnement `{tikzpicture}` ici. À la place, on utilise la commande `\tikz`. Cette commande prend soit un seul argument (commençant par une accolade ouvrante comme dans `\tikz{\draw (0,0) -- (1.5,0)}` dont résulte _____), soit absorbe tout ce qui se trouve avant le point-virgule suivant et le place dans un environnement `{tikzpicture}`. En gros, toutes les commandes de dessin de Ti*k*Z doivent apparaître comme argument de `\tikz` ou dans un environnement `{tikzpicture}`. Heureusement, la commande `\draw` n'est définie que dans cet environnement, il y a donc peu de risques pour que l'on fasse, accidentellement, quelque chose de travers.

3.4 Construction de chemin courbe

Ce que veut faire Karl ensuite c'est tracer un cercle. Pour ça, évidemment, on ne s'en sortira pas avec des droites. Nous avons besoin, au contraire, de tracer des courbes. Pour cela, Ti*k*Z fournit une syntaxe spéciale. On a besoin de un ou deux « points de contrôle ». Les maths cachées derrière ne sont pas tout à fait triviale mais voici l'idée de base : supposez que l'on soit au point x et que le premier point de contrôle soit y. Alors la courbe va démarrer « dans la direction de y en partant de x », c'est-à-dire que la tangente à la courbe en x va pointer vers y. Ensuite, supposons que la courbe doive finir en z avec w pour deuxième point de contrôle. Alors la courbe finira bien en z et la tangente à la courbe au point z passera par w.

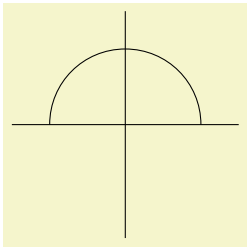
Voici un exemple (on a ajouté les points de contrôle pour plus de clarté) :



```
\begin{tikzpicture}
  \filldraw [gray] (0,0) circle (2pt)
                   (1,1) circle (2pt)
                   (2,1) circle (2pt)
                   (2,0) circle (2pt);
  \draw (0,0) .. controls (1,1) and (2,1) .. (2,0);
\end{tikzpicture}
```

La syntaxe générale pour étendre un chemin d'une manière « courbe » est `.. controls` ⟨*premier point de contrôle*⟩ `and` ⟨*second point de contrôle*⟩`..` ⟨*point final*⟩. On peut ne pas utiliser le `and` ⟨*second point de contrôle*⟩ ce qui fera que le premier sera utilisé deux fois.

Désormais, Karl peut ajouter le premier demi-cercle à la figure :



```
\begin{tikzpicture}
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (-1,0) .. controls (-1,0.555) and (-0.555,1) .. (0,1)
               .. controls (0.555,1) and (1,0.555) .. (1,0);
\end{tikzpicture}
```

Karl est très content du résultat mais trouve que décrire des cercles de cette manière est extrêmement maladroit. Heureusement, il y a une manière bien plus simple.

3.5 Construction d'un chemin circulaire

Afin de tracer un cercle, on peut utiliser l'opération de construction de chemin `circle` (*cercle*). Cette opération est suivie du rayon entre parenthèses comme dans l'exemple suivant (Notez que la position précédente est utilisée comme *centre* du cercle.) :

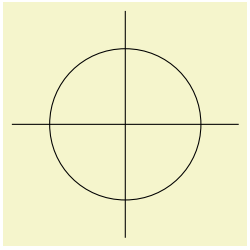
```
\tikz \draw (0,0) circle (10pt);
```

On peut également ajouter une ellipse au chemin avec l'opération `ellipse`. Au lieu d'un seul rayon on en donne deux, l'un pour la direction des x et l'autre pour celle des y, séparés par `and` :

```
\tikz \draw (0,0) ellipse (20pt and 10pt);
```

Pour tracer une ellipse dont les axes ne sont pas horizontaux et verticaux mais pointent dans une direction arbitraire (une « ellipse tournée » comme ⬭) on peut utiliser des transformations, ce que l'on expliquera plus loin. En passant, le code de la petite ellipse est `\tikz \draw[rotate=30] (0,0) ellipse (6pt and 3pt);`.

Donc, pour retourner au problème de Karl, il peut écrire `\draw (0,0) circle (1cm);` pour tracer un cercle.

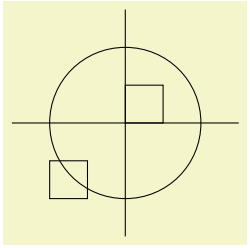


```
\begin{tikzpicture}
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle (1cm);
\end{tikzpicture}
```

À ce moment-là, Karl est un peu alarmé de voir ce cercle si petit quand il voudrait que la figure finale soit bien plus grande. Il est content d'apprendre que Ti*k*Z possède de puissantes options de transformation et qu'agrandir tout d'un facteur trois est très facile. Mais gardons la taille comme elle est pour le moment afin de gagner un peu de place.

3.6 Construction d'un chemin rectangulaire

La prochaine chose que nous voudrions avoir est un quadrillage de fond. Il y a plusieurs manières de l'obtenir. Par exemple, on peut tracer un grand nombre de rectangles. Comme les rectangles sont si courants, il y a une syntaxe spéciale pour eux : pour ajouter un rectangle au chemin courant, utiliser l'opération de construction de chemin `rectangle`. Cette opération doit être suivie d'autres coordonnées et ajoutera un rectangle au chemin de telle sorte que les coordonnées précédentes et les suivantes déterminent les coins du rectangle. Ajoutons donc deux rectangles à la figure :



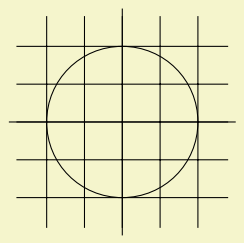
```
\begin{tikzpicture}
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle (1cm);
  \draw (0,0) rectangle (0.5,0.5);
  \draw (-0.5,-0.5) rectangle (-1,-1);
\end{tikzpicture}
```

Bien que ça puisse être sympa dans d'autres circonstances, ça ne nous aide pas vraiment pour le problème de Karl : d'abord il nous faudrait bien trop de ces rectangles et puis la bordure n'est pas « fermée ».

Aussi, Karl est prêt à recourir simplement au tracé de quatre verticales et quatre horizontales en utilisant l'aimable commande `\draw` lorsqu'il apprend qu'il y a aussi une opération de construction de chemin `grid` (*quadrillage*).

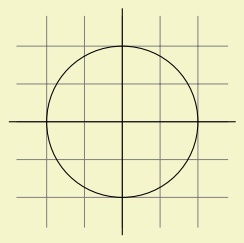
3.7 Construction d'un quadrillage

Karl pourrait utiliser le code suivant :



```
\begin{tikzpicture}
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle (1cm);
  \draw[step=.5cm] (-1.4,-1.4) grid (1.4,1.4);
\end{tikzpicture}
```

Après un autre coup d'œil sur la figure attendue, Karl remarque que ça serait mieux si le quadrillage était moins visible. (Son fils lui a dit que les quadrillages ont tendance à troubler le lecteur s'ils sont trop visibles.) Pour cela, Karl ajoute deux nouvelles options à la commande `\draw` qui dessine le quadrillage. D'abord il utilise la couleur `gray` pour les lignes du quadrillage. Ensuite il réduit la largeur des lignes avec `very thin`. Enfin il échange les places des commandes afin que le quadrillage soit tracé d'abord et que tout le reste vienne par dessus.



```
\begin{tikzpicture}
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle (1cm);
\end{tikzpicture}
```

3.8 Ajouter une touche de style

Au lieu d'utiliser les options `gray, very thin`, Karl aurait pu tout aussi bien écrire `style=help lines`. Les *styles* sont des ensembles prédéfinis d'options que l'on peut utiliser pour organiser la manière dont le graphique sera tracé. En écrivant `style=help lines` on dit : « utilise le style que moi (ou un autre) j'ai défini pour tracer les *lignes d'aide* ». Si Karl décide, plus tard, que les quadrillages devraient être tracés avec, par exemple, la couleur `blue!50` au lieu de `gray`, il lui suffirait d'écrire ce qui suit :

```
\tikzstyle help lines=[color=blue!50,very thin]
```

Ou il pourrait écrire :

```
\tikzstyle help lines+=[color=blue!50]
```

Cela ajouterait l'option `color=blue!50`. Le style `help lines` contiendrait maintenant *deux* options de couleur mais la deuxième remplacerait la première.

En utilisant les styles on rend le code des graphiques plus flexible. On peut changer l'aspect des choses d'une manière plus cohérente.

On peut construire une structure hiérarchique de styles en définissant des styles à l'aide d'autres. Ainsi, afin de définir un style `Karl's grid` basé sur le style `grid`, Karl pourrait écrire

```
\tikzstyle Karl's grid=[style=help lines,color=blue!50]
...
\draw[style=Karl's grid] (0,0) grid (5,5);
```

On peut ne pas écrire le `style=`. En fait, à chaque fois que Ti*k*Z rencontre une option qu'il ne connaît pas, il vérifie si cette option n'est pas un nom de style. Si oui, le style est utilisé. Ainsi, Karl pourrait avoir écrit :

```
\tikzstyle Karl's grid=[help lines,color=blue!50]
...
\draw[Karl's grid] (0,0) grid (5,5);
```

Pour certains styles, comme `very thin`, on voit clairement ce que le style fait et il n'est pas nécessaire d'écrire `style=very thin`. Pour d'autres, comme `help lines`, il me semble qu'écrire `style=help lines` est plus naturel. Mais c'est essentiellement une question de goût.

3.9 Option de dessin

Karl se demande quelles sont les autres options qui influent sur la façon dont un chemin est tracé. Il a déjà vu que l'on peut utiliser l'option `color=`⟨*color*⟩ pour fixer la couleur d'une ligne. L'option `draw=`⟨*color*⟩ fait presque la même chose, seulement elle fixe uniquement la couleur des traits et on peut utiliser une autre couleur pour le remplissage (Karl en aura besoin pour colorier le secteur angulaire).

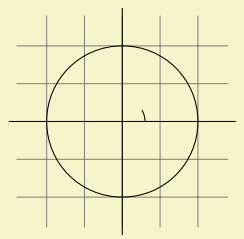
Il remarque que le style `very thin` produit des traits très fins. Karl n'en est pas vraiment surpris ni par le fait que `thin` produise des traits fins, `thick` des traits épais, `very thick` des traits très épais, `ultra thick` des traits vraiment très, très épais et que `ultra thin` produise des traits si fins que certaines imprimantes basse définition ou certains écrans ont du mal à les montrer. Il se demande quelle option permet d'obtenir des traits d'épaisseur « normale ». Il se trouve que c'est `thin` qui le fait. Karl trouve cela étrange mais son fils lui explique que LaTeX a deux commandes nommées `\thinlines` et `\thicklines` et que `\thinlines` donne les traits d'épaisseur « normale », plus précisément, de l'épaisseur, par exemple, du fut d'une lettre comme « T » ou « i ». Néanmoins, Karl voudrait savoir s'il existe quelque chose entre `thin` et `thick`. Il y a quelque chose, c'est `semithick`.

Une autre chose que l'on peut faire c'est tracer les lignes en pointillés ou avec des tirets. Pour cela on peut utiliser les styles `dashed` et `dotted` ce qui produit _ _ et ..... Ces deux options existent dans une version relâchée (*loose*) et une version resserrée (*dense*) appelée `loosely dashed`, `densely dashed`, `loosely dotted` et `densely dotted`. S'il y tient vraiment beaucoup Karl peut aussi définir des motifs plus complexes de pointillés avec l'option `dash pattern` mais son fils insiste sur le fait que les pointillés doivent être utilisés avec la plus grande prudence et que, en général, ils perturbent le lecteur. Le fils de Karl maintient que les motifs compliqués de pointillés sont mauvais. Les étudiants de Karl se contrefichent des motifs de pointillés.

3.10 Construction d'arc

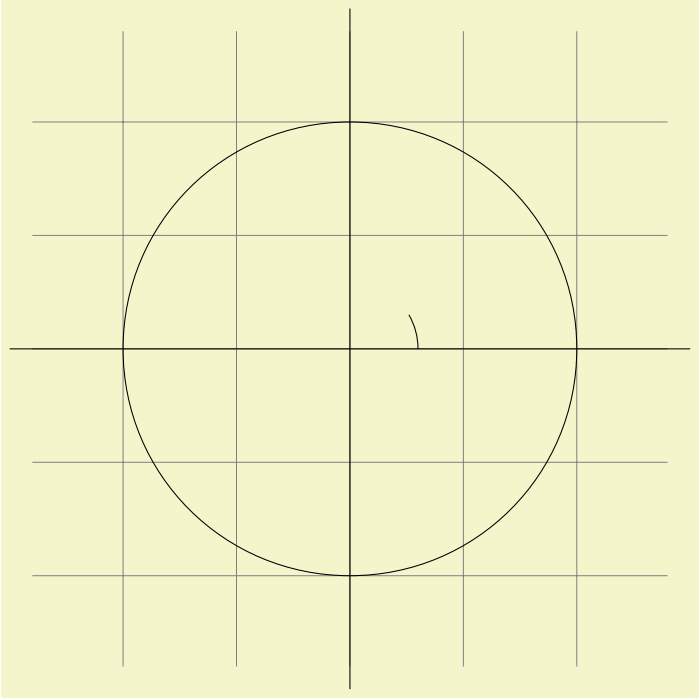
Notre prochain obstacle est de dessiner un arc pour marquer l'angle. Pour cela, l'opération de construction de chemin `arc` se révèle utile. Elle trace une partie de cercle ou d'ellipse. Cette opération `arc` doit être suivie par un triplet entre parenthèses. Les deux premières composantes sont des angles, la dernière est un rayon. Par exemple, `arc(10:80:10pt)` signifie « un arc allant de 10 à 80 degrés sur un cercle de rayon 10pt ». Karl a évidemment besoin d'un arc de 0° à 30°. Le rayon devrait être assez petit, peut-être environ un tiers du rayon du cercle. Cela donne `(0:30:3mm)`.

Lorsque l'on utilise l'opération de construction de chemin `arc`, l'arc déterminé est ajouté avec pour point de départ la position courante. Nous avons donc besoin d'abord d'y aller.



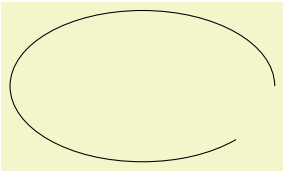
```
\begin{tikzpicture}
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle (1cm);
  \draw (3mm,0mm) arc (0:30:3mm);
\end{tikzpicture}
```

Karl pense que tout cela est un peu trop petit et qu'il ne peut pas continuer sans apprendre comment mettre à l'échelle. Pour cela, il peut ajouter l'option `[scale=3]`. Il peut ajouter cette option à chaque commande `\draw` mais ce serait maladroit. Au lieu de cela il l'ajoute à tout l'environnement, ce qui conduit cette option à s'appliquer partout à l'intérieur.



```
\begin{tikzpicture}[scale=3]
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle (1cm);
  \draw (3mm,0mm) arc (0:30:3mm);
\end{tikzpicture}
```

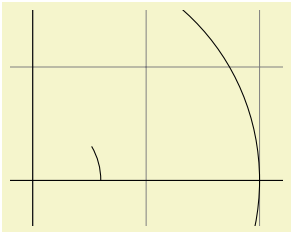
Comme pour les cercles, on peut spécifier « deux » rayons pour obtenir un arc d'ellipse.



```
  \tikz \draw (0,0) arc (0:315:1.75cm and 1cm);
```

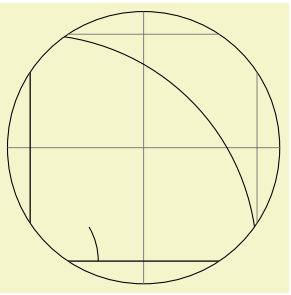
3.11 Découpage d'un chemin

Pour gagner un peu de place dans ce manuel il serait bien de rogner un peu le graphique de Karl afin de pouvoir nous concentrer sur les parties « intéressantes ». Découper est vraiment simple avec Ti*k*Z. On utilise la commende `\clip` qui découpe tout le dessin qui suit. Cela marche comme `\draw` mais au lieu de tracer quelque chose ça utilise le chemin donné pour découper tout ce qui suit.



```
\begin{tikzpicture}[scale=3]
  \clip (-0.1,-0.2) rectangle (1.1,0.75);
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle (1cm);
  \draw (3mm,0mm) arc (0:30:3mm);
\end{tikzpicture}
```

On peut faire les deux choses en même temps : dessiner *et* découper un chemin. Pour cela, on utilise la commande `\draw` en ajoutant l'option `clip`. (Ce n'est pas tout : on peut également utiliser la commande `\clip` avec l'option `draw`. Bon, ce n'est pas tout non plus : en fait, `\draw` est juste une abréviation pour `\path[draw]` et `\clip` une abréviation pour `\path[clip]` et on peut aussi écrire `\path[draw,clip]`.) Voici un exemple :



```
\begin{tikzpicture}[scale=3]
  \clip[draw] (0.5,0.5) circle (.6cm);
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle (1cm);
  \draw (3mm,0mm) arc (0:30:3mm);
\end{tikzpicture}
```

3.12 Construction de chemin parabolique ou sinusoïdal

Bien que Karl n'en ait pas besoin pour son graphique, il est content d'apprendre qu'il y a des opérations de chemin `parabola` (*parabole*), `sin` et `cos` pour ajouter des courbes paraboliques ou sinusoïdales au chemin courant. Pour l'opération `parabola` le point courant sera sur la parabole ainsi que le point donné après l'opération. Considérons l'exemple suivant :

```
\tikz \draw (0,0) rectangle (1,1)  (0,0) parabola (1,1);
```

On peut aussi placer le sommet ailleurs :

```
\tikz \draw[x=1pt,y=1pt] (0,0) parabola bend (4,16) (6,12);
```

Les opérations `sin` et `cos` ajoutent une courbe d'équation $y = \sin x$ ou $y = \cos x$ pour x dans l'intervalle $[0, \pi/2]$ de telle sorte que le point courant soit le point de départ de la courbe et que celle-ci finisse au point donné. Voici deux exemples :

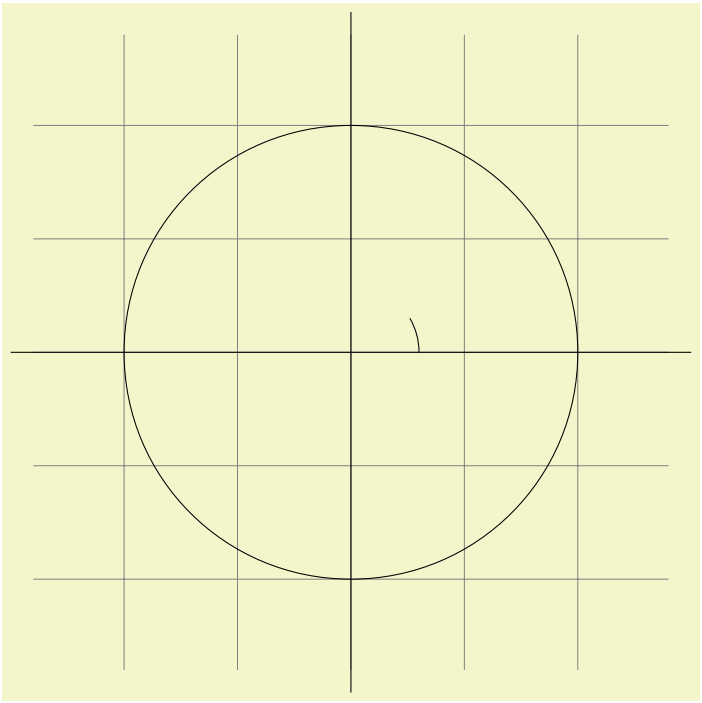
Une sinusoïde .

```
Une sinusoïde \tikz \draw[x=1ex,y=1ex] (0,0) sin (1.57,1);.
```

```
\tikz \draw[x=1.57ex,y=1ex] (0,0) sin (1,1) cos (2,0) sin (3,-1) cos (4,0)
                            (0,1) cos (1,0) sin (2,-1) cos (3,0) sin (4,1);
```

3.13 Tracer et colorier

Revenant à la figure, Karl veut maintenant colorier l'angle avec un vert très pâle. Pour cela il utilise `\fill` au lieu de `\draw`. Voici ce que fait Karl :



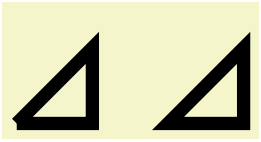
```
\begin{tikzpicture}[scale=3]
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle (1cm);
  \draw (3mm,0mm) arc (0:30:3mm);
\end{tikzpicture}
```

La couleur `green!20!white` est composée de 20% de vert et de 80% de blanc. Une telle expression de couleur est possible car PGF utilise l'extension `xcolor` de Uwe Kern. Voyez la documentation de cette extension pour des détails sur les expressions de couleur.

Que se serait-il passé si Karl n'avait pas « fermé » le chemin avec `--(0,0)` ? Dans ce cas le chemin aurait été fermé automatiquement et on aurait pu omettre le dernier point. En fait, il aurait mieux valu coder à la place quelque chose comme ce qui suit :

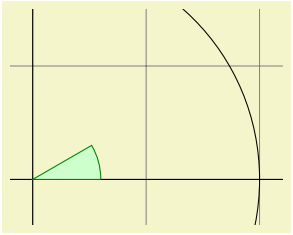
```
  \fill[green!20!white] (0,0) -- (3mm,0mm) arc (0:30:3mm) -- cycle;
```

Le `--cycle` fait que le chemin courant est fermé (en fait la partie courante du chemin courant) en joignant souplement le premier et le dernier point. Pour apprécier la différence regardez l'exemple suivant :



```
\begin{tikzpicture}[line width=5pt]
  \draw (0,0) -- (1,0) -- (1,1) -- (0,0);
  \draw (2,0) -- (3,0) -- (3,1) -- cycle;
  \useasboundingbox (0,1.5); % agrandit la boîte-cadre
\end{tikzpicture}
```

On peut tracer et remplir un chemin en même temps avec la commande `\filldraw`. Cela tracera d'abord le chemin puis le remplira. Cela peut ne pas sembler très utile mais on peut définir des couleurs différentes pour tracer et remplir. On spécifie cela avec des arguments optionnels comme ici :



```
\begin{tikzpicture}[scale=3]
  \clip (-0.1,-0.2) rectangle (1.1,0.75);
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle (1cm);
  \filldraw[fill=green!20!white, draw=green!50!black]
    (0,0) -- (3mm,0mm) arc (0:30:3mm) -- cycle;
\end{tikzpicture}
```

3.14 Estomper

Karl considère brièvement la possibilité de rendre l'angle « plus fantaisiste » en l'*estompant*[4]. Au lieu de le remplir avec une couleur uniforme, on utilise une transition douce entre différentes couleurs. Pour cela on peut utiliser `\shade` ou `\shadedraw`, pour tracer et estomper en même temps :

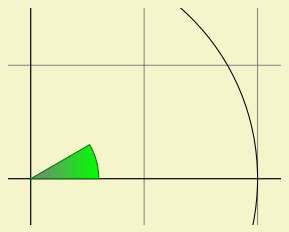
```
  \tikz \shade (0,0) rectangle (2,1)  (3,0.5) circle (.5cm);
```

L'ombrage par défaut est une transition douce entre le gris et le blanc. Pour spécifier d'autres couleurs, on peut utiliser les options :

```
\begin{tikzpicture}[rounded corners,ultra thick]
  \shade[top color=yellow,bottom color=black] (0,0) rectangle +(2,1);
  \shade[left color=yellow,right color=black] (3,0) rectangle +(2,1);
  \shadedraw[inner color=yellow,outer color=black,draw=yellow] (6,0) rectangle +(2,1);
  \shade[ball color=green] (9,.5) circle (.5cm);
\end{tikzpicture}
```

4. NdTdS : Il faudrait, en toute rigueur, utiliser soit le verbe « dégrader » qui, d'après le Littré, s'applique à une couleur mais dont le sens courant est un peu trop prégnant et négatif pour qu'il ne soit pas source de confusion, soit la locution « appliquer un dégradé ». Je vais, une fois de plus, au plus court, revendiquant pour moi ce qu'on trouve bon pour la nature : le principe de moindre action.

Pour Karl, ceci pourrait convenir :



```
\begin{tikzpicture}[scale=3]
  \clip (-0.1,-0.2) rectangle (1.1,0.75);
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle (1cm);
  \shadedraw[left color=gray,right color=green, draw=green!50!black]
    (0,0) -- (3mm,0mm) arc (0:30:3mm) -- cycle;
\end{tikzpicture}
```

Toutefois, il décide avec sagesse que l'estompage ne fait d'habitude que perturber le lecteur sans ajouter quoique ce soit à la figure.

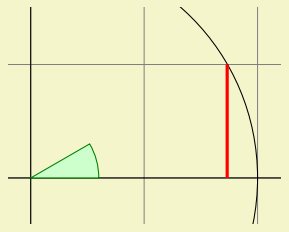
3.15 Définir des coordonnées

Karl veut maintenant ajouter les courbes du sinus et du cosinus. Il sait déjà qu'il peut définir la couleur de ces courbes avec l'option `color=`. Mais quelle est la meilleure manière de définir les coordonnées ?

Il y a deux manières de définir des coordonnées. La plus simple est d'écrire quelque chose comme `(10pt,2cm)`. Cela signifie 10pt suivant l'axe des x et 2 cm suivant l'axe des y. Autrement on peut aussi omettre les unités comme dans `(1,2)` qui signifie « une fois le vecteur courant suivant x plus deux fois le vecteur courant suivant y [5] ». Ces vecteurs ont pour longueur par défaut 1 cm dans chaque direction.

Afin de définir des points en coordonnées polaires on utilise la notation `(30:1cm)`, qui signifie 1 cm dans la direction 30 degrés. C'est évidemment bien utile pour « obtenir le point $(\cos 30°, \sin 30°)$ sur le cercle ».

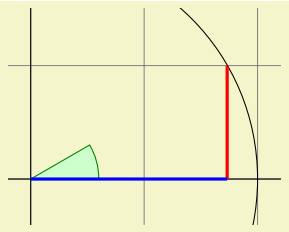
On peut ajouter, devant les coordonnées, un signe `+` simple ou un double comme dans `+(1cm,0cm)` ou `++(0cm,2cm)`. De telles coordonnées sont interprétées différemment : la première signifie « 1 cm vers le haut depuis la position définie précédemment » et la deuxième « 2 cm vers la droite de la position précédemment définie en en faisant la nouvelle position définie ». Par exemple, on peut tracer la courbe du sinus comme suit :



```
\begin{tikzpicture}[scale=3]
  \clip (-0.1,-0.2) rectangle (1.1,0.75);
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle (1cm);
  \filldraw[fill=green!20,draw=green!50!black]
    (0,0) -- (3mm,0mm) arc (0:30:3mm) -- cycle;
  \draw[red,very thick] (30:1cm) -- +(0,-0.5);
\end{tikzpicture}
```

Karl utilise le fait que $\sin 30° = 1/2$. Toutefois, il doute très fortement que ses étudiants le sachent, aussi il pense que ce serait bien s'il y avait un moyen de déterminer un point comme « le point, juste sous le point `(30:1cm)`, placé sur l'axe des x ». C'est en fait possible avec une syntaxe spéciale : Karl peut écrire `(30:1cm |-0,0)`. De manière générale, le sens de `(`$\langle p \rangle$ `|-` $\langle q \rangle$`)` est « le point d'intersection de la verticale passant par p et de l'horizontale passant par q ».

Ensuite, traçons la courbe du cosinus. Une façon serait d'écrire `(30:1cm |- 0,0) -- (0,0)`. Une autre façon est la suivante : nous continuons depuis le point où s'arrête le sinus :



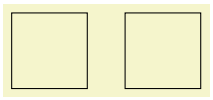
```
\begin{tikzpicture}[scale=3]
  \clip (-0.1,-0.2) rectangle (1.1,0.75);
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle (1cm);
  \filldraw[fill=green!20,draw=green!50!black] (0,0) -- (3mm,0mm) arc
  (0:30:3mm) -- cycle;
  \draw[red,very thick]  (30:1cm) -- +(0,-0.5);
  \draw[blue,very thick] (30:1cm) ++(0,-0.5) -- (0,0);
\end{tikzpicture}
```

Notez qu'il n'y a pas de `--` entre `(30:1cm)` et `+(0,-0.5)`. En détail, le chemin est interprété comme suit : « d'abord, le `(30:1cm)` me dit de déplacer mon crayon jusque $(\cos 30°, 1/2)$. Ensuite vient une autre définition de coordonnées, aussi je déplace mon crayon sans rien tracer. Ce nouveau point est une unité sous

5. NdTdS : pour un français frotté de maths du secondaire on dira « vecteur courant colinéaire au vecteur directeur de l'axe des abscisses » etc.

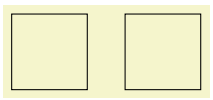
la dernière position c.-à-d. à $(\cos 30°, 0)$. Enfin, je déplace le crayon à l'origine mais cette fois je trace quelque chose (à cause du `--`). »

Pour voir la différence entre `+` et `++` regardez ce qui suit :



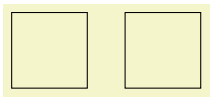
```
\begin{tikzpicture}
  \def\rectanglepath{-- ++(1cm,0cm)  -- ++(0cm,1cm)  -- ++(-1cm,0cm) -- cycle}
  \draw (0,0) \rectanglepath;
  \draw (1.5,0) \rectanglepath;
\end{tikzpicture}
```

Par comparaison, lorsque l'on utilise un simple `+`, les coordonnées sont différentes :



```
\begin{tikzpicture}
  \def\rectanglepath{-- +(1cm,0cm)  -- +(1cm,1cm)  -- +(0cm,1cm) -- cycle}
  \draw (0,0) \rectanglepath;
  \draw (1.5,0) \rectanglepath;
\end{tikzpicture}
```

Naturellement, on aurait pu écrire tout cela plus clairement et à moindre frais comme ceci (avec un simple ou un double `+`) :

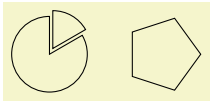


```
\tikz \draw (0,0) rectangle +(1,1)  (1.5,0) rectangle +(1,1);
```

Il reste à Karl le segment pour $\tan\alpha$ qu'il semble difficile de définir en utilisant les transformations et les coordonnées polaires. Pour cela il a besoin d'une autre façon de définir des coordonnées : Karl peut définir des coordonnées à l'aide d'intersections de droites. Le segment pour $\tan\alpha$ commence à $(1, 0)$ et monte verticalement jusqu'au point d'intersection d'une droite verticale et d'une droite passant par l'origine et `(30:1cm)`. La syntaxe pour obtenir ce point est la suivante :

```
\draw[very thick,orange] (1,0) -- (intersection of 1,0--1,1 and 0,0--30:1cm);
```

Dans ce qui suit, deux derniers exemples présentent la façon d'utiliser les positions relatives. Notez que les options de transformations qui sont expliquées plus loin sont souvent plus utiles pour déplacer des objets que des positions relatives.

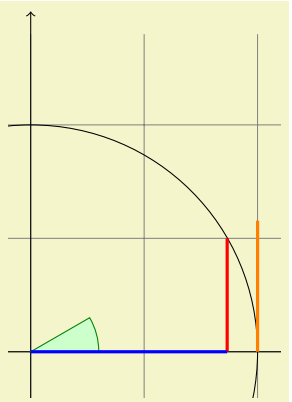


```
\begin{tikzpicture}[scale=0.5]
  \draw (0,0) -- (90:1cm) arc (90:360:1cm) arc (0:30:1cm) -- cycle;
  \draw (60:5pt) -- +(30:1cm) arc (30:90:1cm) -- cycle;

  \draw (3,0)  +(0:1cm) -- +(72:1cm) -- +(144:1cm) -- +(216:1cm) --
               +(288:1cm) -- cycle;
\end{tikzpicture}
```

3.16 Ajouter des pointes de flèches

Karl veut maintenant ajouter les petites pointes de flèche au bout des axes. Il a remarqué que dans de nombreuses figures, même dans des journaux scientifiques, ces pointes semblent manquer, peut-être parce que les programmes qui ont créé les graphiques ne peuvent pas les produire. Karl pense qu'il doit y avoir des pointes au bout des axes. Son fils est d'accord. Ses étudiants ne s'occupent pas des flèches.

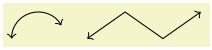
Il se trouve qu'ajouter des flèches est plutôt simple : Karl ajoute l'option `->` aux commandes de tracé des axes :



```
\begin{tikzpicture}[scale=3]
  \clip (-0.1,-0.2) rectangle (1.1,1.51);
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \draw[->] (-1.5,0) -- (1.5,0);
  \draw[->] (0,-1.5) -- (0,1.5);
  \draw (0,0) circle (1cm);
  \filldraw[fill=green!20,draw=green!50!black] (0,0) -- (3mm,0mm) arc
  (0:30:3mm) -- cycle;
  \draw[red,very thick]    (30:1cm) -- +(0,-0.5);
  \draw[blue,very thick]   (30:1cm) ++(0,-0.5) -- (0,0);
  \draw[orange,very thick] (1,0) -- (intersection of 1,0--1,1 and 0,0--30:1cm);
\end{tikzpicture}
```

Si Karl avait utilisé l'option `<-` à la place de `->` les pointes auraient été placées au début du chemin. L'option `<->` place des flèches à chaque extrémité du chemin.

Il y a certaines restrictions au type de chemins auxquels on peut ajouter des flèches. En gros, on ne peut ajouter des flèches qu'aux « lignes » simples ouvertes. Par exemple, on ne devrait pas essayer d'ajouter des flèches à, disons, un rectangle ou un cercle. (On peut essayer mais il n'y a aucune garantie sur ce qui arrivera ni aujourd'hui ni dans les prochaines versions.) Toutefois, on peut ajouter des flèches à des chemins courbes et à des chemins composés de plusieurs segments, comme dans les exemples qui suivent :



```
\begin{tikzpicture}
  \draw [<->] (0,0) arc (180:30:10pt);
  \draw [<->] (1,0) -- (1.5cm,10pt) -- (2cm,0pt) -- (2.5cm,10pt);
\end{tikzpicture}
```

Karl regarde plus en détail la flèche que Ti*k*Z place à la fin. Elle apparaît comme ceci quand il l'agrandit : ⟶. La forme semble vaguement familière et, de fait, c'est exactement celle du bout de la flèche normal que TEX utilise dans quelque chose comme $f\colon A \to B$.

Karl aime cette flèche d'autant plus qu'elle n'est pas aussi épaisse que celles offertes par tant d'extensions. Toutefois, il s'attend à ce que, parfois, il ait besoin d'utiliser d'autres genre de flèches. Pour cela, Karl peut écrire, par exemple `>=`⟨*right arrow tip kind*⟩ où ⟨*right arrow tip kind*⟩ est une spécification spéciale de pointe de flèche. Par exemple, si Karl écrit `>=stealth` alors il dirait à Ti*k*Z qu'il voudrait une pointe de flèche « à la combattant furtif » :

```
\begin{tikzpicture}[>=stealth]
  \draw [->] (0,0) arc (180:30:10pt);
  \draw [<<-,very thick] (1,0) -- (1.5cm,10pt) -- (2cm,0pt) -- (2.5cm,10pt);
\end{tikzpicture}
```

Karl se demande si un nom aussi guerrier[6] pour un type de flèche est vraiment nécessaire. Il n'est pas vraiment radouci lorsque son fils lui dit que PowerPoint de Microsoft utilise le même nom. Il décide d'en faire discuter ses étudiants un jour.

En plus de `stealth` il y a de nombreuses autres pointes prédéfinies parmi lesquelles Karl peut choisir, voyez la section **??**, p. **??**. De plus il peut définir des types de flèche lui-même s'il en a besoin.

3.17 Portée

Karl a déjà vu que de nombreuses options de graphique peuvent influer sur le rendu des chemins. Souvent, il voudrait appliquer certaines options à tout un ensemble de commandes graphiques. Par exemple, Karl pourrait désirer tracer trois chemins en utilisant le crayon `thick` (*épais*) mais vouloir que le reste soit tracé « normalement ».

Si Karl souhaite qu'un certain ensemble d'options de graphique s'appliquent à toute la figure, il peut simplement passer ces options à la commande `\tikz` ou à l'environnement `{tikzpicture}` (Gerda passerait ces options à `\tikzpicture`). Toutefois, si Karl veut les appliquer à un groupe local, il place ces commandes dans un environnement `{scope}` (Gerda utilise `\scope` et `\endscope`). Cet environnement prend des options de graphique comme argument optionnel et ces options s'applique à tout ce que contient l'environnement mais à rien d'extérieur.

Voici un exemple :

```
\begin{tikzpicture}[ultra thick]
  \draw (0,0) -- (0,1);
  \begin{scope}[thin]
    \draw (1,0) -- (1,1);
    \draw (2,0) -- (2,1);
  \end{scope}
  \draw (3,0) -- (3,1);
\end{tikzpicture}
```

Délimiter la portée à l'aide de `{scope}` a un autre effet intéressant : tout changement affectant la surface découpée est local à la portée. Ainsi si on écrit `\clip` quelque part dans la portée, l'effet de cette commande `\clip` s'achève à la fin de la portée. C'est utile car il n'y a pas d'autre moyen d'agrandir la partie découpée.

Karl a déjà vu également que les options données à une commande ne s'applique qu'à cette commande. Il se trouve que la situation est un peu plus compliquée. D'abord, les options passées à une commande comme `\draw` ne sont pas vraiment des options de la commande mais des « options de chemin » et peuvent être données n'importe où dans le chemin. Ainsi, au lieu de `\draw[thin] (0,0) -- (1,0);` on peut écrire aussi

6. NdTdS : L'adjectif « stealth » qui signifie furtif est employé pour un certain type d'avion militaire.

`\draw (0,0) [thin] -- (1,0);` ou `\draw (0,0) -- (1,0) [thin];`. Tous ces codes auront le même effet. Cela peut paraître étrange puisque dans le dernier cas il pourrait sembler que le `thin` ne prend effet qu'après que la droite de $(0,0)$ à $(1,0)$ a été tracée. Toutefois la plupart des options de graphique ne s'applique qu'à un chemin complet. En fait si l'on écrit et `thin` et `thick` dans un même chemin, c'est la dernière des options données qui « gagne ».

En lisant ce qui précède, Karl remarque que seulement « la plupart » des options de graphique s'applique au chemin complet. En fait, toutes les options de transformations *ne* s'appliquent *pas* au chemin entier mais seulement à « tout ce qui les suit sur le chemin ». Nous regarderons cela plus en détail d'ici peu. Cependant toutes les options données pendant la construction d'un chemin ne s'appliquent qu'à ce chemin.

3.18 Transformations

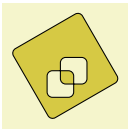
Quand on définit des coordonnées comme `(1cm,1cm)`, où le point sera-t-il placé sur la page ? Pour déterminer cette position, Ti*k*Z, TEX et PDF ou PostScript appliquent tous certaines transformations aux coordonnées données pour déterminer la position finale sur la page.

Ti*k*Z fournit de nombreuses options qui permettent de transformer les coordonnées dans le système privé de coordonnées de PGF. Par exemple, l'option `xshift` permet de pousser tous les points suivants d'une certaine quantité :

```
\tikz \draw (0,0) -- (0,0.5) [xshift=2pt] (0,0) -- (0,0.5);
```

Il est important de noter que l'on peut changer de transformation « au milieu du chemin », une caractéristique que n'a ni PDF ni PostScript. La raison en est que PGF garde la trace de sa propre matrice de transformation.

Voici un exemple plus complexe :



```
\begin{tikzpicture}[even odd rule,rounded corners=2pt,x=10pt,y=10pt]
  \filldraw[fill=examplefill] (0,0)   rectangle (1,1)
    [xshift=5pt,yshift=5pt]   (0,0)   rectangle (1,1)
                  [rotate=30]   (-1,-1) rectangle (2,2);
\end{tikzpicture}
```

Les transformations les plus utiles sont `xshift` et `yshift` pour déplacer, `shift` pour déplacer jusqu'à un point donné comme dans `shift={(1,0)}` ou `shift={+(0,0)}` (les accolades sont nécessaires pour que TEX ne prenne pas les virgules comme séparateurs d'options), `rotate` pour faire tourner d'un certain angle (il y a également un `rotate around` pour faire tourner autour d'un point donné), `scale` pour agrandir ou rétrécir d'un certain facteur, `xscale` et `yscale` pour agrandir uniquement parallèlement à l'axe des x ou l'axe des y, (`xsacle=-1` est une symétrie axiale) et `xslant` et `yslant` pour incliner. Si ces transformations et celles que je n'ai pas mentionnées ne suffisent pas, l'option `cm` permet d'appliquer une matrice de transformation arbitraire. Les étudiants de Karl, au passage, ne savent pas ce qu'est une matrice de transformation.

3.19 Répéter : boucles pour

Le but suivant de Karl est d'ajouter des graduations sur les axes aux emplacements -1, $-1/2$, $1/2$ et 1. Pour ce faire il lui serait agréable de pouvoir utiliser une sorte de « boucle » d'autant plus qu'il souhaite faire la même chose à chacune de ces positions. Il existe plusieurs extensions pour faire ça. LATEX a ses propres commandes internes, `pstricks` est livré avec la puissante commande `multido`. Toutes peuvent être utilisées avec PGF et Ti*k*Z, aussi si vous en êtes familier, n'hésitez pas à les utiliser. PGF apporte encore une autre commande, nommée `\foreach` (*pour chaque*), que j'ai ajoutée parce que je n'arrivais jamais à me souvenir de la syntaxe des autres extensions. `\foreach` est définie dans l'extension `pgffor` et peut être utilisée indépendamment de PGF. Ti*k*Z la charge automatiquement.

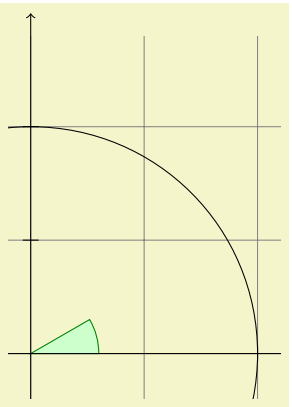
Dans sa forme de base, la commande `\foreach` est d'une utilisation aisée :

$x = 1$, $x = 2$, $x = 3$,

```
\foreach \x in {1,2,3} {$x =\x$, }
```

La syntaxe générale est `\foreach` ⟨*variable*⟩ `in {`⟨*liste de valeurs*⟩`}`⟨*commandes*⟩. À l'intérieur de ⟨*commandes*⟩ la ⟨*variable*⟩ prendra les diverses valeurs. Si ⟨*commandes*⟩ ne commence pas par une accolade, tout ce qui précède le prochain point-virgule est utilisé comme ⟨*commandes*⟩.

Pour Karl et les graduations sur les axes, il pourrait utiliser le code suivant :

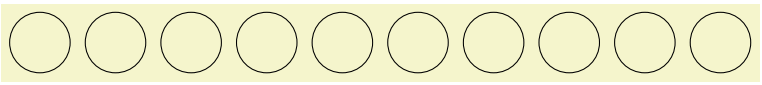


```
\begin{tikzpicture}[scale=3]
  \clip (-0.1,-0.2) rectangle (1.1,1.51);
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \filldraw[fill=green!20,draw=green!50!black] (0,0) -- (3mm,0mm) arc
  (0:30:3mm) -- cycle;
  \draw[->] (-1.5,0) -- (1.5,0);
  \draw[->] (0,-1.5) -- (0,1.5);
  \draw (0,0) circle (1cm);

  \foreach \x in {-1cm,-0.5cm,1cm}
    \draw (\x,-1pt) -- (\x,1pt);
  \foreach \y in {-1cm,-0.5cm,0.5cm,1cm}
    \draw (-1pt,\y) -- (1pt,\y);
\end{tikzpicture}
```

En fait, il y a plusieurs façon de créer les graduations. Par exemple, Karl pourrait avoir placé le `\draw ...;` entre accolades. Il pourrait aussi avoir utilisé, par exemple,

```
\foreach \x in {-1,-0.5,1}
  \draw[xshift=\x cm] (0pt,-1pt) -- (0pt,1pt);
```

Karl est curieux de savoir ce qui se passerait dans une situation plus complexe lorsqu'il y a, disons, 20 graduations. Cela semble ennuyeux d'avoir à mentionner explicitement tous les nombres dans l'ensemble de `\foreach`. En effet, on peut utiliser `...` dans l'expression du `\foreach` pour itérer sur un grand nombre de valeurs (qui doivent être cependant des nombres réels sans dimension) comme dans l'exemple suivant :



```
\tikz \foreach \x in {1,...,10}
        \draw (\x,0) circle (0.4cm);
```

Si on fournit *deux* nombres avant le `...`, l'expression du `\foreach` utilisera leur différence comme pas :


```
\tikz \foreach \x in {-1,-0.5,...,1}
       \draw (\x cm,-1pt) -- (\x cm,1pt);
```

On peut aussi emboîter les boucles pour créer des effets intéressants :

1,5	2,5	3,5	4,5	5,5
1,4	2,4	3,4	4,4	5,4
1,3	2,3	3,3	4,3	5,3
1,2	2,2	3,2	4,2	5,2
1,1	2,1	3,1	4,1	5,1

7,5	8,5	9,5	10,5	11,5	12,5
7,4	8,4	9,4	10,4	11,4	12,4
7,3	8,3	9,3	10,3	11,3	12,3
7,2	8,2	9,2	10,2	11,2	12,2
7,1	8,1	9,1	10,1	11,1	12,1

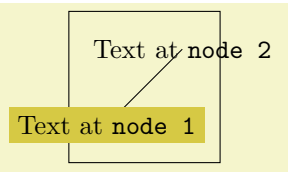
```
\begin{tikzpicture}
  \foreach \x in {1,2,...,5,7,8,...,12}
    \foreach \y in {1,...,5}
    {
      \draw (\x,\y) +(-.5,-.5) rectangle ++(.5,.5);
      \draw (\x,\y) node{\x,\y};
    }
\end{tikzpicture}
```

L'expression `\foreach` peut faire des choses encore plus sournoises mais ce qui précède donne l'idée principale.

3.20 Ajouter du texte

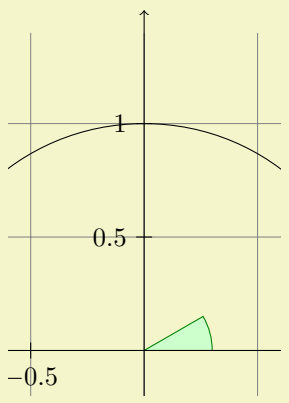
Karl est, maintenant, assez satisfait de la figure. Toutefois, la partie la plus importante, à savoir les étiquettes, manque encore !

Ti*k*Z offre un système puissante et facile à utiliser pour ajouter à une figure du texte et, plus généralement, des formes complexes à une position donnée. L'idée de base est la suivante : lorsque Ti*k*Z construit un chemin et rencontre le mot-clef `node` (*nœud*) au milieu du chemin, il lit la *spécification de nœud*. Le mot-clef `node` est suivi de quelques options et de texte placé entre accolades. Ce texte est placé dans une boite normale de TEX (si la spécification de nœud suit directement des coordonnées, ce qui le cas généralement, Ti*k*Z est capable de faire un peu de magie afin qu'il soit même possible d'utiliser du texte *verbatim* dans les boites) et puis la place à la position courante, c.-à-d. à la dernière position déterminée (éventuellement déplacée un peu, suivant les options données). Toutefois, tous les nœuds ne sont tracés seulement qu'après que le chemin a été complètement tracé/rempli/ombré/découpé/etc.




```
\begin{tikzpicture}
  \draw (0,0) rectangle (2,2);
  \draw (0.5,0.5) node [fill=examplefill]
                      {Text at \verb!node 1!}
    -- (1.5,1.5) node {Text at \verb!node 2!};
\end{tikzpicture}
```

Visiblement, Karl voudrait placer des nœuds non seulement à la dernière position définie mais aussi à gauche ou à droite de cette position. Pour cela, chaque nœud qu'on place dans une figure est équipé de plusieurs *ancres* (*anchor*). Par exemple, l'ancre `north` (*nord*) est au milieu à la partie supérieure de la forme, l'ancre `south` (*sud*) est au bas et l'ancre `north east` (*nord-est*) est dans le coin supérieur droit. Lorsque l'on passe l'option `anchor=north`, le texte est placé de telle sorte que cette ancre nordique est située à la position courante et que le texte est, de ce fait, sous la position courante. Karl utilise cela pour tracer les graduations comme suit :



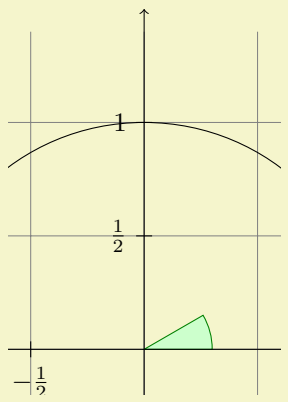

```
\begin{tikzpicture}[scale=3]
  \clip (-0.6,-0.2) rectangle (0.6,1.51);
  \draw[step=.5cm,style=help lines] (-1.4,-1.4) grid (1.4,1.4);
  \filldraw[fill=green!20,draw=green!50!black]
    (0,0) -- (3mm,0mm) arc (0:30:3mm) -- cycle;
  \draw[->] (-1.5,0) -- (1.5,0);   \draw[->] (0,-1.5) -- (0,1.5);
  \draw (0,0) circle (1cm);

  \foreach \x in {-1,-0.5,1}
    \draw (\x cm,1pt) -- (\x cm,-1pt) node[anchor=north] {$\x$};
  \foreach \y in {-1,-0.5,0.5,1}
    \draw (1pt,\y cm) -- (-1pt,\y cm) node[anchor=east] {$\y$};
\end{tikzpicture}
```

C'est déjà pas mal. En utilisant ces ancres, Karl peut maintenant ajouter la plupart des autres éléments textuels. Toutefois Karl pense que, bien que « correcte », c'est plutôt contrintuitif que, pour placer quelque chose *sous* un point donné, il faille utiliser l'ancre *north*. Pour cette raison, il y a une option `below` (*dessous*) qui produit la même chose que `anchor=north`. De même, `above right` (*dessus à droite*) fait comme `anchor=south east`. De plus `below` prend un argument optionnel de dimension. Si on le donne, la forme sera, en plus, poussée vers le bas de la valeur donnée. Ainsi, on peut utiliser `below=1pt` pour placer une étiquette textuelle sous un point et, en plus, la déplacer de 1pt vers le bas.

Karl n'est pas tout à fait satisfait des graduations. Il voudrait avoir 1/2 ou $\frac{1}{2}$ plutôt que 0.5, en partie pour faire montre des exquises capacités de TEX et de Ti*k*Z, en partie parce que pour des positions comme 1/3 ou π il est de beaucoup préférable d'avoir la graduation « mathématique » plutôt que juste « numérique ». Ces étudiants, par ailleurs, préfèrent 0.5 à 1/2 puisqu'ils ne sont pas de manière générale très friands de fractions.

Karl maintenant fait face à un problème : dans l'expression `\foreach` la position `\x` devrait toujours être données comme `0.5` puisque Ti*k*Z ne comprendrait pas ce que `\frac{1}{2}` est sensé être. Par ailleurs, le texte écrit devrait vraiment être `\frac{1}{2}`. Pour résoudre ce problème, `\foreach` offre une syntaxe spéciale : au lieu d'une seule variable `\x`, Karl peut en définir deux (ou même plus) séparées par une barre oblique (*slash*) comme dans `\x / \xtext`. Alors, les éléments de l'ensemble sur lequel `\foreach` itère doivent être aussi de la forme ⟨*premier*⟩`/`⟨*second*⟩. À chaque itération, `\x` prendra la valeur de ⟨*premier*⟩ et `\xtext`

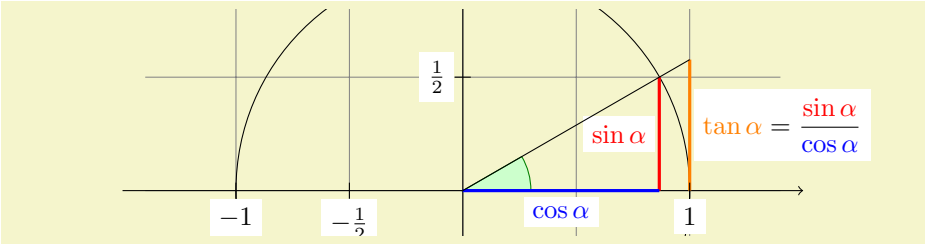
celle de ⟨*second*⟩. Si on ne donne pas de ⟨*second*⟩, le ⟨*premier*⟩ est utilisé à nouveau. Aussi, voici le nouveau code pour les graduations :




```
\begin{tikzpicture}[scale=3]
  \clip (-0.6,-0.2) rectangle (0.6,1.51);
  \draw[step=.5cm,style=help lines] (-1.4,-1.4) grid (1.4,1.4);
  \filldraw[fill=green!20,draw=green!50!black]
    (0,0) -- (3mm,0mm) arc (0:30:3mm) -- cycle;
  \draw[->] (-1.5,0) -- (1.5,0); \draw[->] (0,-1.5) -- (0,1.5);
  \draw (0,0) circle (1cm);

  \foreach \x/\xtext in {-1, -0.5/-\frac{1}{2}, 1}
    \draw (\x cm,1pt) -- (\x cm,-1pt) node[anchor=north] {$\xtext$};
  \foreach \y/\ytext in {-1, -0.5/-\frac{1}{2}, 0.5/\frac{1}{2}, 1}
    \draw (1pt,\y cm) -- (-1pt,\y cm) node[anchor=east] {$\ytext$};
\end{tikzpicture}
```

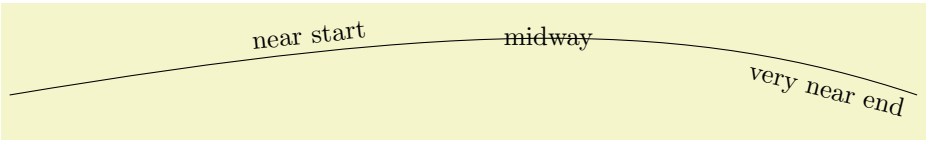
Le résultat fait plaisir à Karl mais son fils montre que ce n'est pas parfaitement satisfaisant : le quadrillage et le cercle interfèrent avec les nombres et amoindrissent leur lisibilité. Karl ne se sent pas très concerné par ça (ses étudiants ne s'en rendent même pas compte) mais son fils insiste sur le fait qu'il y a une solution facile : Karl peut ajouter l'option `[fill=white]` pour remplir l'arrière plan de la forme de texte de blanc.

Karl veut ensuite ajouter des étiquettes comme $\sin\alpha$. Celles-là il voudrait les placer « au milieu de la courbe ». Pour ce faire, au lieu de définir l'étiquette `node {$\sin\alpha$}` directement après une des extrémités de la courbe (ce qui placerait l'étiquette à cette extrémité), Karl peut placer cette étiquette directement après `--` mais avant les coordonnées. Par défaut, cela place l'étiquette au milieu de la courbe mais on peut utiliser les options `pos=` pour modifier ce comportement. On peut également utiliser les options comme `near start` et `near end` pour changer la position :




```
\begin{tikzpicture}[scale=3]
  \clip (-2,-0.2) rectangle (2,0.8);
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \filldraw[fill=green!20,draw=green!50!black] (0,0) -- (3mm,0mm) arc
  (0:30:3mm) -- cycle;
  \draw[->] (-1.5,0) -- (1.5,0) coordinate (x axis);
  \draw[->] (0,-1.5) -- (0,1.5) coordinate (y axis);
  \draw (0,0) circle (1cm);

  \draw[very thick,red]
    (30:1cm) -- node[left=1pt,fill=white] {$\sin \alpha$} (30:1cm |- x axis);
  \draw[very thick,blue]
    (30:1cm |- x axis) -- node[below=2pt,fill=white] {$\cos \alpha$} (0,0);
  \draw[very thick,orange] (1,0) -- node [right=1pt,fill=white]
    {$\displaystyle \tan \alpha \color{black}=
      \frac{{\color{red}\sin \alpha}}{\color{blue}\cos \alpha}$}
    (intersection of 0,0--30:1cm and 1,0--1,1) coordinate (t);

  \draw (0,0) -- (t);

  \foreach \x/\xtext in {-1, -0.5/-\frac{1}{2}, 1}
    \draw (\x cm,1pt) -- (\x cm,-1pt) node[anchor=north,fill=white] {$\xtext$};
  \foreach \y/\ytext in {-1, -0.5/-\frac{1}{2}, 0.5/\frac{1}{2}, 1}
    \draw (1pt,\y cm) -- (-1pt,\y cm) node[anchor=east,fill=white] {$\ytext$};
\end{tikzpicture}
```

On peut placer les étiquettes sur les courbes et, en ajoutant l'option `sloped` (*déclive, en pente*), faire qu'elles soient tournées de telle sorte qu'elles suivent la pente de la courbe. Voici un exemple :



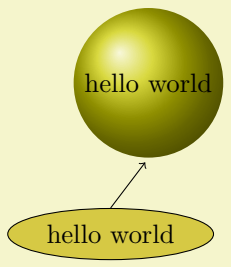

```
\begin{tikzpicture}
  \draw (0,0) .. controls (6,1) and (9,1) ..
    node[near start,sloped,above] {near start}
    node {midway}
    node[very near end,sloped,below] {very near end} (12,0);
\end{tikzpicture}
```

Il reste à écrire le texte explicatif sur la droite de la figure. La difficulté principale ici est de limiter la largeur de « l'étiquette textuelle » qui est assez longue ce qui pousse à utiliser la coupure de ligne. Heureusement, Karl peut utiliser l'option `text width=6cm` pour obtenir l'effet voulu. Ainsi, voici le code complet :

```
\begin{tikzpicture}[scale=3,cap=round]
  % Local definitions
  \def\costhirty{0.8660256}

  % Colors
  \colorlet{anglecolor}{green!50!black}
  \colorlet{sincolor}{red}
  \colorlet{tancolor}{orange!80!black}
  \colorlet{coscolor}{blue}

  % Styles
  \tikzstyle{axes}=[]
  \tikzstyle{important line}=[very thick]
  \tikzstyle{information text}=[rounded corners,fill=red!10,inner sep=1ex]

  % The graphic
  \draw[style=help lines,step=0.5cm] (-1.4,-1.4) grid (1.4,1.4);

  \draw (0,0) circle (1cm);

  \begin{scope}[style=axes]
    \draw[->] (-1.5,0) -- (1.5,0) node[right] {$x$} coordinate(x axis);
    \draw[->] (0,-1.5) -- (0,1.5) node[above] {$y$} coordinate(y axis);

    \foreach \x/\xtext in {-1, -.5/-\frac{1}{2}, 1}
      \draw[xshift=\x cm] (0pt,1pt) -- (0pt,-1pt) node[below,fill=white] {$\xtext$};

    \foreach \y/\ytext in {-1, -.5/-\frac{1}{2}, .5/\frac{1}{2}, 1}
      \draw[yshift=\y cm] (1pt,0pt) -- (-1pt,0pt) node[left,fill=white] {$\ytext$};
  \end{scope}

  \filldraw[fill=green!20,draw=anglecolor] (0,0) -- (3mm,0pt) arc(0:30:3mm);
  \draw (15:2mm) node[anglecolor] {$\alpha$};

  \draw[style=important line,sincolor]
    (30:1cm) -- node[left=1pt,fill=white] {$\sin \alpha$} (30:1cm |- x axis);

  \draw[style=important line,coscolor]
    (30:1cm |- x axis) -- node[below=2pt,fill=white] {$\cos \alpha$} (0,0);

  \draw[style=important line,tancolor] (1,0) -- node[right=1pt,fill=white] {
    $\displaystyle \tan \alpha \color{black}=
    \frac{{\color{sincolor}\sin \alpha}}{\color{coscolor}\cos \alpha}$}
    (intersection of 0,0--30:1cm and 1,0--1,1) coordinate (t);

  \draw (0,0) -- (t);

  \draw[xshift=1.85cm]
    node[right,text width=6cm,style=information text]
    {
      The {\color{anglecolor} angle $\alpha$} is $30^\circ$ in the
      example ($\pi/6$ in radians). The {\color{sincolor}sine of
        $\alpha$}, which is the height of the red line, is
      \[
      {\color{sincolor} \sin \alpha} = 1/2.
      \]
      By the Theorem of Pythagoras ...
    };
\end{tikzpicture}
```

3.21 Nœuds

Placer du texte à une position donnée n'est qu'un cas spécial d'un mécanisme sous-jacent plus général. Lorsque l'on écrit `\draw (0,0) node{texte};`, ce qui arrive en fait est que un nœud rectangulaire, ancré en son centre, est placé à la position $(0, 0)$. Par dessus le nœud rectangulaire le texte `texte` est tracé. Puisque aucune action n'est définie pour le rectangle (comme `draw` ou `fill`), le rectangle est en fait jeté et seul le texte est montré. Toutefois en ajoutant `fill` ou `draw` on peut rendre visible la forme sous-jacente. De plus, on peut *changer* la forme en utilisant par exemple `shape=circle` ou juste `circle`. Si on charge l'extension `pgflibraryshapes` on dispose également de `ellipse` :

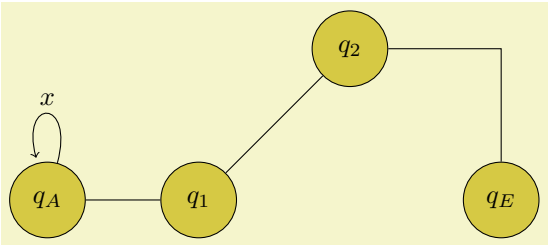



```
\begin{tikzpicture}
  \path (0,0)   node[ellipse,fill=examplefill,draw]
                  (h1) {hello world}
        (0.5,2) node[circle,shade,ball color=examplefill]
                  (h2) {hello world};
  \draw [->,shorten >=2pt] (h1.north) -- (h2.south);
\end{tikzpicture}
```

Comme le montre l'exemple ci-dessus, on peut ajouter un nom à un nœud en le plaçant entre parenthèses entre le `node` et le `{`⟨*texte*⟩`}` (on peut également utiliser l'option `name=`). Cela fera que Ti*k*Z se rappellera de nœud et de toutes ses ancres. On peut faire référence à ces ancres lorsque l'on définit des coordonnées. La syntaxe est `(`⟨*node name*⟩`.`⟨*anchor*⟩`)`. Actuellement et aussi dans un avenir proche, *cela ne marchera pas entre différentes figures puisque Ti*k*Z perd la trace des positions quand il rend le contrôle à TEX.* Avec certain pilote il est possible de faire de la magie mais une implantation portable semble impossible (pensez simplement à un possible pilote pour SVG).

L'option `shorten >` raccourcit les lignes de 2pt à la fin. De même, on peut utiliser `shorten <` pour raccourcir (ou même rallonger) les lignes à leur début. C'est possible même si on ne trace aucune flèche.

Il n'est pas toujours nécessaire de définir une ancre. Si on ne donne pas d'ancre, Ti*k*Z tentera, de lui-même, de définir une ancre raisonnable sur le bord (si Ti*k*Z échoue à en trouver quelque chose d'utile, il utilisera le centre). Voici un exemple typique :




```
\begin{tikzpicture}
  \begin{scope}[shape=circle,minimum size=1cm,fill=examplefill]
    \tikzstyle{every node}=[draw,fill]
    \node (q_A) at (0,0) {$q_A$};
    \node (q_E) at (6,0) {$q_E$};
    \node (q_1) at (2,0) {$q_1$};
    \node (q_2) at (4,2) {$q_2$};
  \end{scope}
  \draw (q_A) -- (q_1) -- (q_2) -| (q_E);
  \draw[->,shorten >=2pt] (q_A) .. controls +(75:1.4cm) and +(105:1.4cm) .. node[above] {$x$} (q_A);
\end{tikzpicture}
```

Dans l'exemple on utilise la commande `\node` qui est une abréviation de `\path node`.

4 Conseils à propos des graphiques

Cette section ne concerne pas PGF ou Ti*k*Z mais donne quelques conseils sur la création de graphiques pour des présentations, des articles ou des livres scientifiques.

Les conseils de cette section proviennent de sources différentes. Un grand nombre d'entre eux ressortissent simplement de ce que je voudrais appeler le « bon sens », d'autres reflètent mon expérience personnelle (toutefois, je l'espère, pas mes préférences personnelles), d'autres encore sont issus de livres (la bibliographie est toujours manquante, j'en suis désolé) sur la création de graphique et la typographie. Ce qui m'a le plus influencé ce sont les livres brillants de Edward Tufte. Alors même que je ne suis pas d'accord avec tout ce qu'on lit dans ces livres, je trouve que de nombreux arguments de Tufte sont si convaincant que j'ai décidé de les reprendre dans les conseils qui suivent.

4.1 Faut-il suivre ces conseils ?

La première question que l'on devrait se poser lorsque quelqu'un donne un tas de conseils est : dois-je vraiment suivre ces conseils ? C'est une question importante parce qu'il y a de bonnes raisons de ne pas suivre des conseils généraux.

- La personne qui a écrit ces conseils pourrait avoir en vu un autre objectif que le vôtre. Par exemple, un conseil pourrait être « utilisez la couleur rouge pour mettre en relief ». Ce conseil est sensé s'il s'agit, par exemple, d'une présentation faite au rétroprojecteur mais la « couleur » rouge a l'effet *opposé* à la mise en relief lorsqu'on imprime avec une imprimante noir et blanc.
  Les conseils sont presque toujours rédigés en vue d'une situation particulière. Si l'on n'est pas dans cette situation, suivre ce conseil peut faire plus de mal que de bien.
- La règle de base en typographie est : « on peut contourner une règle tant que l'on *sait* que l'on contourne une règle ». Cette règle s'applique aussi aux graphiques. Dite autrement, la règle de base est : « les seules erreurs en typographie sont les choses faites par ignorance ».
  Lorsque vous connaissez une règle et que vous décidez que la contourner permet d'obtenir l'effet voulu, contournez la.
- Certains conseils sont simplement *faux* mais tout le monde les suit par tradition ou est forcé de le faire. Mon exemple favori est la directive qu'une société de conception de logiciels pour laquelle j'ai travaillé avait incorporée à un grand projet : tous les programmeurs doivent déclarer les paramètres des fonctions dans *l'ordre croissant de taille*. Ainsi les paramètres de un octet devaient venir les premiers puis ceux de deux octets, et ainsi de suite.
  Cette directive est un non-sens total. Un conseil (discutable) plus sensé serait de « déclarer les paramètres dans l'ordre alphabétique » afin qu'ils soient plus facile à trouver. Une autre recommandation indubitablement sensée est que « les paramètres doivent être déclarés par ordre de taille décroissante » afin de réduire les absences en mémoire-cache pour raison de non-alignement sur des adresses multiples d'octets lors des accès à la pile. La recommandation qu'utilisait la société maximisait les absences en mémoire-cache et conduisait à un ordre des déclarations plus ou moins aléatoire, si bien que les programmeurs devaient constamment le réaménager. [7]

Aussi avant de suivre un conseil ou de décider de ne pas le suivre, on doit se demander :

1. Ce conseil s'applique-t-il vraiment dans ma situation ?
2. Si on fait le contraire de ce que ce conseil voudrait que l'on fasse, les avantages l'emporteront-ils sur les inconvénients que ce conseil était sensé éviter ?

4.2 Estimer le temps nécessaire à la création de graphiques

Lorsque l'on crée un article avec de nombreux graphiques, le temps nécessaire à créer ces graphiques devient un facteur important. Combien de temps devrait-on prévoir pour la création des graphiques.

En général, il faut compter qu'un graphique demande autant de temps qu'en demande un texte de la même longueur. Par exemple, lorsque j'écris un article, j'ai besoin d'une heure par page pour le premier jet. Ensuite, j'ai besoin de deux à quatre heures par page pour les relectures et corrections. Aussi je compte environ une demie heure pour créer un *premier jet* d'un graphique d'une demie page. Je m'attends à avoir besoin plus tard d'une à deux heures supplémentaires pour produire le graphique final.

Dans beaucoup de publications, même des revues de qualité, les auteurs et éditeurs ont visiblement investi beaucoup de temps sur le texte mais semblent avoir passé environ cinq minutes pour créer tous les

7. NdTdS : La traduction de ces deux dernières phrases est due à Michel OLAGNON, obligeamment venu à mon secours sur `fr.lettres.langue.anglaise`.

graphiques. Les graphiques semblent souvent avoir été ajoutés après coup ou ressemblent à une copie d'écran du logiciel de statistique qu'utilise l'auteur. Comme on le soutiendra plus loin, les graphiques produits par des programmes comme GNUPLOT sont, par défaut, médiocres.

Créer des graphiques instructif qui aident le lecteur et se marient correctement avec le texte principal est un processus long et difficile.

- Traitez les graphiques comme des citoyens de première classe de vos articles. Ils méritent autant de temps et d'énergie que le texte.
- On peut avancer que la création de graphiques mérite *même plus* de temps que l'écriture du texte principal parce qu'on sera plus attentif aux graphiques et qu'on les regarda les premiers.
- Prévoyez autant de temps pour la création et la correction d'un graphique que vous n'en prévoyez pour un texte de la même taille.
- Les graphiques difficiles contenant beaucoup d'informations peuvent même demander plus de temps.
- Des graphiques très simple demanderont moins de temps mais, très probablement, vous ne voulez pas de « graphiques très simples » dans votre article, de toute façon ; de même que vous ne voudriez pas de « texte très simple » de la même taille.

4.3 Processus de création de graphique

Lorsque l'on écrit un article (scientifique), on suit probablement le modèle suivant : on a quelques résultats ou idées qu'on voudrait publier. La création de l'article commencera d'habitude par un plan sommaire. Puis, on remplira les différentes parties avec du texte pour créer un premier brouillon. Ce brouillon sera revu de manière répétée jusqu'à ce que, souvent après d'importantes corrections, un article fini en émerge. Il n'y a en général pas une seule phrase du brouillon d'un bon article de revue qui ait survécu sans modification.

La création de graphiques suit le même modèle :

- Décider ce que le graphique doit communiquer. En faire une décision consciente c'est-à-dire répondre à la question « qu'est-ce que ce graphique est-il sensé dire au lecteur ? »
- Créer un croquis c'est-à-dire une forme générale et grossière du graphique, contenant les éléments les plus importants. Souvent il est utile de le faire avec papier et crayon.
- Placer les détails plus précis pour créer le premier brouillon.
- Reprendre ce graphique de manière répétée en même temps que le reste de l'article.

4.4 Lier le graphique avec le texte principal

On peut placer les graphiques à différents endroits dans un article. Ou bien on les place « dans le texte » c-à-d. quelque part au milieu d'un paragraphe ou entre deux paragraphes ou bien on les place à part, « en hors-texte ». Comme les imprimeurs (et les gens en général) aiment bien les pages pleines (autant pour des raisons esthétiques qu'économiques) les figures hors-texte peuvent traditionnellement être placées sur des pages très éloignées du texte principal dans lequel on y fait référence. LaTeX et TeX tendent à encourager ce rejet des graphiques pour des raisons techniques.

Quand une figure est dans le texte, elle est plus ou moins automatiquement liée au texte principal dans le sens que ses annotations seront implicitement expliquées par le texte environnant. Ainsi le texte principal explicitera en général le pourquoi de la figure et ce qu'elle montre.

Il en est tout à fait autrement pour une figure hors-texte qui sera vue lorsque le texte auquel elle est liée soit n'aura pas encore été lu soit aura été lu bien longtemps avant. Pour cela on devrait, en créant un figure hors-texte, suivre les conseils que voici :

- Les figures hors-texte devraient avoir une légende qui les rendent « compréhensibles par elles-mêmes ». Supposons, par exemple, qu'une figure montre un exemple de différentes étapes de l'algorithme de tri rapide (*quicksort*). La légende de la figure devrait, au minimum, informer le lecteur de ce que « La figure montre les différentes étapes de l'algorithme de tri *quicksort* présenté en page xyz » et non pas juste « Algorithme *quicksort* ».
- Une bonne légende donne autant de contexte que possible. Par exemple, on pourrait écrire « La figure montre les différentes étapes de l'algorithme de tri *quicksort* présenté en page xyz. Dans la première ligne, on a pris l'élément 5 pour pivot. Cela entraîne... » Bien que cette information puisse être donnée aussi dans le texte principal, en la plaçant dans la légende on garantira que le contexte est donné. N'ayez pas peur de légende de 5 lignes de long. (Votre éditeur vous haïra peut-être pour ça. Envisager de le haïr en retour.)
- On renverra à la figure, depuis le texte principal, avec quelque chose comme « Pour un exemple de *quicksort* en action, voir la Figure 2.1, page xyz. »

– La plupart des ouvrages sur la typographie et des marches d'éditeur recommandent de ne pas utiliser les abréviations comme « Fig. 2.1 » mais d'écrire, au long, « Figure 2.1 ».
L'argument principal contre les abréviations est que « le point est trop précieux pour le gaspiller dans une abréviation ». L'idée est que le point amène le lecteur à penser que la phrase s'arrête après « Fig » et qu'il doit « revenir consciemment en arrière » pour se rendre compte que la phrase ne se finissait pas là après tout.
L'argument en faveur des abréviations est qu'elles économisent de la place.
Personnellement je ne suis convaincu par aucun de ces arguments. D'un côté, je n'ai jamais vu de preuve sérieuse que les abréviations ralentissent la lecture. D'un autre, abréger tous les « Figure » en « Fig. » a peu de chance d'économiser ne fusse qu'une seule ligne dans la plupart des documents.
J'évite les abréviations.

4.5 Cohérence du texte et des figures

Peut-être que « l'erreur » la plus fréquemment faite quand on crée des graphiques (souvenez-vous que une « erreur » en conception est toujours juste de « l'ignorance ») est d'avoir une discordance entre l'aspect des graphiques et celui du texte.

Il est fréquent que les auteurs utilisent différents logiciels pour créer les graphiques de leur article. Un auteur pourra produire quelques courbes à l'aide de GNUPLOT, un diagramme avec XFIG et incorporer une image `.eps` produite par un co-auteur utilisant un logiciel inconnu. Tous ces graphiques auront, le plus vraisemblablement, différentes épaisseurs de traits, des polices différentes, des tailles différentes. De plus, les auteurs utilisent souvent, en important les graphiques, des options comme `[height=5cm]` pour les réduire à une « taille sympa ».

Si la même approche était suivie dans l'écriture du texte principal, chaque partie serait écrite avec une police et une taille différente. Dans quelques parties les théorèmes seraient soulignés, dans d'autres il seraient en capitales, dans d'autres encore écrits en rouge. De plus, les marges changeraient d'une page à l'autre.

Les lecteurs et les éditeurs ne toléreraient pas qu'un texte soit écrit de cette façon [8] mais ils tolèrent ce genre de choses pour les graphiques.

Pour assurer une cohérence entre les graphiques et le texte, suivez attentivement ces conseils :

– Ne réduisez ni n'agrandissez les graphiques.
Cela signifie que, lorsque l'on crée une figure avec un programme externe, on doit la créer « à la bonne taille ».
– Utilisez la même police dans la figure et le corps de texte.
– Utilisez la même largeur de trait dans le texte et les figures.
La « largeur de ligne » d'un texte normal est l'épaisseur du fut d'une lettre comme le T. Pour TEX, cela signifie habituellement 0,4 pt. Toutefois quelques revues n'accepteront pas de figures dont l'épaisseur du trait normal est inférieure à 0,5 pt.
– Lorsque vous utilisez des couleurs, faites le en suivant un code de couleur cohérent tant dans le texte que dans les figures. Par exemple, si le rouge est sensé attirer l'attention du lecteur sur quelque chose dans le texte principal, utilisez le dans les figures pour les parties importantes. Si le bleu est utilisé pour les éléments structurels tels que les titres de sections, utilisez aussi le bleu pour les éléments structurels des figures.
Toutefois les figures peuvent faire usage d'un code de couleur intrinsèque et logique. Par exemple, quelque soit la couleur utilisée normalement les lecteurs penseront en général, par exemple, le vert comme « positif, passez, d'accord » et le rouge comme « attention, alerte, action ».

Assurer la cohérence en utilisant différents logiciels de production de graphiques est quasi impossible. On devra donc envisager de n'en utiliser qu'un seul.

4.6 Annotations dans les figures

Presque toutes les figures contiennent des annotations c-à-d. des bouts de texte expliquant des parties du graphiques. Lorsque vous placez ces annotations, suivez ces conseils :

– Soyez cohérent en plaçant les annotations et cela sur deux plans : premièrement, soyez cohérent avec le texte principal c-à-d. utilisez la même police pour les annotations que pour le texte principal ; deuxièmement, assurez la cohérence entre annotations c-à-d. que si vous présentez certaines annotations d'une manière particulière, vous devriez présenter de la même manière toutes les annotations.

8. NdTdS : Il me semble que l'auteur fait preuve ici d'un optimisme exagéré.

– Non seulement vous devriez utilisez les mêmes polices pour le texte et pour les graphiques mais en plus les mêmes notations. Par exemple, si vous écrivez 1/2 dans votre texte principal, utilisez aussi « 1/2 » pour les annotations de vos figures et pas « 0,5 ». π est « π » et non « 3,141 ». Enfin $e^{-i\pi}$ est « $e^{-i\pi}$ », et pas « -1 » et encore moins « -1 ».
– Les annotations devraient être lisibles. Non seulement elles devraient avoir une taille raisonnable mais de plus elles ne devraient pas être recouvertes par des lignes ou d'autres textes. Cela s'applique aussi aux lignes et aux textes placés *derrière* les annotations.
– Les annotations devraient être « en place ». À chaque fois qu'il y a assez de place, l'annotation devraient être placée près de la chose qu'elle marque. On ajoutera une ligne (discrète) entre l'annotation et l'objet auquel elle se rapporte seulement si nécessaire. Essayez d'éviter les annotations qui ne font que référence à des explications situées dans des légendes extérieures. Le lecteur doit alors sauter d'avant en arrière entre les explications et l'objet décrit.
– Pensez à atténuer les annotations « accessoires » en les colorant en gris par exemple. Cela permettra de centrer l'attention sur la figure elle-même.

4.7 Courbes et diagrammes

L'espèce la plus fréquente de graphiques, spécialement dans les articles scientifiques, est celle des *courbes et diagrammes*. Elle présente de nombreuses variétés comprenant les courbes simples, les courbes paramétriques, les surfaces, les histogrammes, les diagrammes en camembert, et bien d'autres encore.

Malheureusement, il est de notoriété publique qu'il est difficile d'obtenir ces graphiques correctement. En partie, on en peut blâmer les réglages par défaut des logiciels comme GNUPLOT ou Excel car ces logiciels rendent très facile la création de mauvais diagrammes ou de mauvaises courbes.

La première question à se poser lorsque l'on crée une courbe est :

– Y-a-t'il assez de données pour mériter une courbe ?

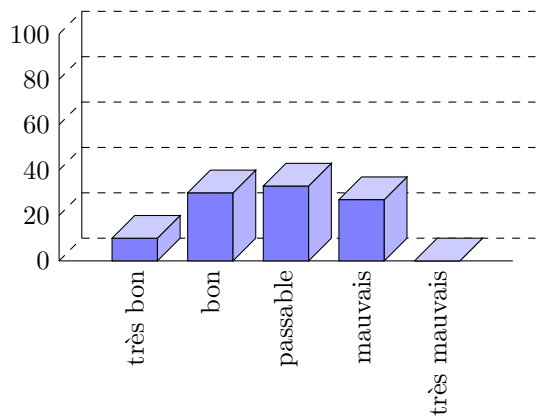
Si la réponse est « pas vraiment » utilisez une table.

Une situation typique où on n'a pas besoin d'un diagramme est lorsque des gens présentent quelques nombres avec un diagramme à barres. Voici un exemple vécu : à la fin d'un séminaire, un intervenant demande leur avis aux participants. D'après cette petite enquête, trois participants considéraient le séminaire comme « très bon », neuf comme « bon », dix comme « passable », huit comme « mauvais » et personne ne qualifiait le séminaire de « très mauvais ».

Une façon simple de résumer l'information est la table suivante :

Classement	*Nombre de participants (sur 50) pour ce classement*	*Pourcentage*
« très bon »	3	6%
« bon »	9	18%
« passable »	10	20%
« mauvais »	8	16%
« très mauvais »	0	0%
sans opinion	20	40%

Ce que fit l'intervenant fut de présenter les données à l'aide d'un diagramme à barres en 3 dimensions. Cela ressemblait à ceci :



La table et le « diagramme » ont à peu près la même taille. Si vous pensez d'abord que « le diagramme semble plus beau que la table », essayez de répondre aux questions suivantes d'après les informations données par la table ou par le graphique :

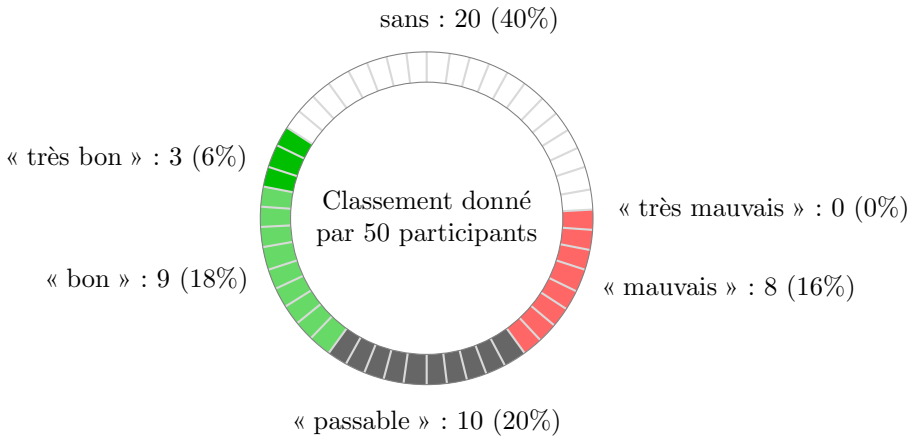
1. Combien y-avait-il de participants ?
2. Combien de participants ont-ils répondu à l'enquête ?
3. Quel pourcentage de participants ont répondu à l'enquête ?
4. Combien de participants ont-ils coché « très bon » ?
5. Quel pourcentage de tous les participants ont-ils coché « très bon » ?
6. Y-eut-il plus d'un quart des participants pour cocher « mauvais » ou « très mauvais » ?
7. Quel pourcentage de participants ont-ils rendu l'enquête en ayant coché « très bien » ?

Malheureusement le diagramme ne permet de répondre à *aucune de ces questions*. La table répond à toute directement, sauf à la dernière. Essentiellement la densité d'information du graphique est très proche de zéro. La table a une bien plus grande densité d'information en dépit du fait qu'elle utilise beaucoup d'espace blanc pour présenter quelques nombres.

Voici la liste de ce qui n'allait pas dans le diagramme 3D :

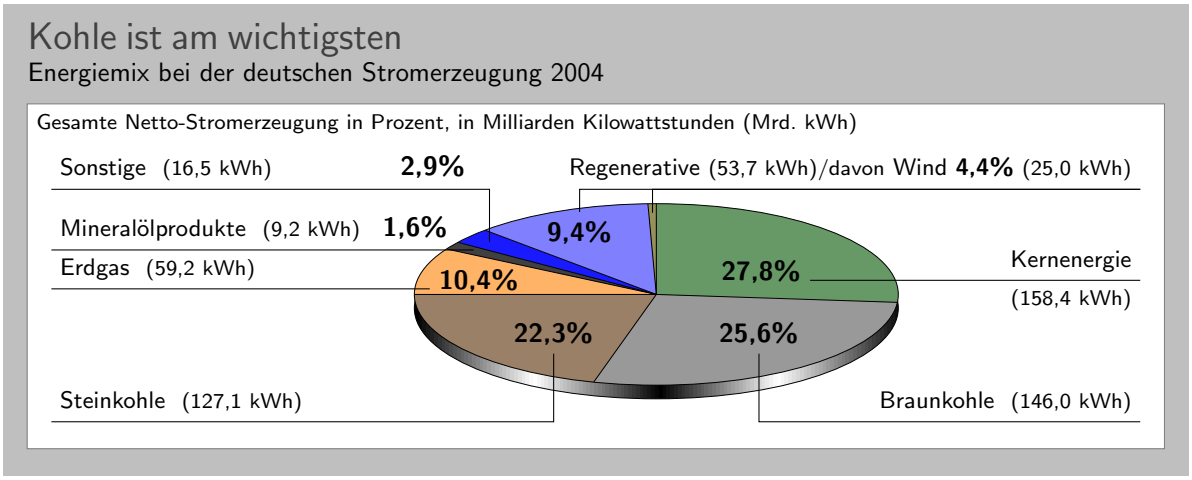
- L'ensemble du graphique est dominé par d'irritants traits en arrière plan.
- On ne comprend pas clairement ce que signifient les nombres de gauche ; vraisemblablement des pourcentages mais ce pourraient être aussi des nombres absolus de participants.
- Les annotations du bas ont subi une rotation ce qui les rend difficiles à lire.
  (Dans la présentation que j'ai vue, le texte était rendu dans une très basse résolution d'environ 10 fois 6 pixels par lettre avec de mauvaises approches, rendant ce texte presque impossible à lire.)
- La troisième dimension ajoute de la complexité au graphique sans apporter d'information.
- La présentation en trois dimension rend très difficile l'estimation de la hauteur des barres. Considérons la barre de « mauvais ». Représente-t-elle plus ou moins de 20 ? Alors que le devant de la barre est sous la ligne de 20, l'arrière (qui compte) est au-dessus.
- Il est impossible de dire quels sont les nombres représentés par les barres. Aussi les barres cachent-elles gratuitement l'information qu'elles devaient présenter.
- Que représente la somme des hauteurs des barres : 100% ou 60% ?
- La barre de « très mauvais » représente-t-elle 0 ou 1 ?
- Pourquoi les barres sont-elles bleues ?

On pourrait avancer que dans cet exemple les nombres exacts ne sont pas importants pour le diagramme. L'important est le « message » qui est qu'il y a plus de « très bon » et de « bon » que de « mauvais » et « très mauvais ». Toutefois, s'il s'agit de faire passer ce message autant utiliser une phrase qui le dit ou un graphique qui le présente plus clairement :



Le graphique ci-dessus a à peu près la même densité d'information que la table (il a à peu près la même taille et présente à peu près les mêmes nombres). De plus, on peut « voir » directement que il y a plus de bons classements que de mauvais. On peut aussi « voir » que le nombre de personne qui n'ont pas donné de classement n'est pas négligeable, ce qui est fréquent pour une telle enquête.

Un diagramme n'est pas toujours une bonne idée. Regardons un exemple que je redessine d'après un diagramme circulaire de *Die Zeit* du 4 juin 2005 :



Ce graphique a été redessiné avec Ti*k*Z, mais l'original lui ressemble beaucoup.

À première vue le graphique semble « beau et instructif » mais il y a beaucoup de choses qui ne vont pas :

- Le diagramme est en trois dimensions. Toutefois, les ombres n'apporte rien en terme d'information et, au mieux, elles perturbent.
- Dans un diagramme circulaire en 3D, les tailles relatives sont très déformées. Par exemple, l'aire de la zone grise du « Braunkohle » est plus grande que celle de la zone verte du « Kernenergie » *en dépit du fait que le pourcentage de Braukohle est plus petit que celui de Kernenergie*.
- La distortion due à l'effet 3D est encore pire pour les petites surfaces. L'aire de « Regenerative » est un peu plus grande que celle de « Erdgas ». L'aire de « Wind » est légèrement plus petite que celle de « Mineralölprodukte » *alors même que le pourcentage du Wind est presque trois fois plus grand que celui de Mineralölprodukte*.
  Dans ce dernier cas, les différences dans les tailles ne sont dues qu'en partie à la distortion. Le (ou les) dessinateur(s) du graphique originel ont fait la part du « Wind » trop petit même en tenant compte de la distortion. (Comparez simplement la taille de « Wind » et celle de « Regenerative » en général.)
- D'après sa légende, ce diagramme est sensé nous dire que le charbon est la plus importante source d'énergie en Allemagne en 2004. En laissant de côté les distortions dues à l'effet superflu et trompeur de la 3D, cela prend un temps certain pour capter ce message.
  Le charbon, comme source d'énergie, est partagée en deux secteurs : un pour le « Steinkohle » et un pour le « Braukohle » (deux sortes différentes de charbon). Lorsque l'on les ajoute, on voit que toute la partie inférieure du diagramme est consacrée au charbon.
  Les deux aires des différentes sortes de charbon ne sont pas du tout reliées visuellement. Plutôt, deux couleurs différentes sont utilisées, les annotations sont sur des côtés différents du graphique. Par comparaison, « Regenerative » et « Wind » sont très intimement reliés.
- Le code de couleur du graphique ne suit aucune structure logique. Pourquoi l'énergie nucléaire en vert ? L'énergie renouvelable est en bleu pâle, les « autres sources » en bleu. On a presque l'impression d'une plaisanterie quand on voit que la surface pour « Braunkohle » (qui se traduit littéralement par « charbon marron ») est en gris pierre alors que la surface pour « Steinkohle » (qu'on traduit littéralement par « charbon de pierre ») est marron.
- La surface qui a la couleur la plus claire est utilisée pour « Erdgas ». Cette surface est celle qui ressort le plus à cause de cette couleur plus lumineuse. Toutefois pour ce diagramme « Erdgas » n'est pas important du tout.

Edward Tufte appelle les diagrammes comme celui qui précède « diagrammes de pacotille ».

Voici quelques conseils qui pourraient vous aider à éviter de produire des diagrammes de pacotille :

- Ne pas utiliser de diagramme circulaire en 3D. C'est *mal*.
- Envisager d'utiliser une table au lieu d'un diagramme circulaire.
- Ne pas appliquer les couleurs au petit bonheur la chance ; les utiliser pour guider l'attention du lecteur et pour grouper les choses.
- Ne pas utiliser de motifs d'arrière-plan comme des hachures ou des diagonales au lieu de couleurs. Ces motifs perturbent. De tels motifs dans un diagramme instructif sont *mal*.

4.8 Attention et distraction

Prenez votre roman favori et jetez un œil sur une page typique. Vous verrez que la page est très uniforme. Rien n'est là pour perturber le lecteur dans sa lecture ; pas de gros titres, pas de texte en gras, pas de grande plage blanche. En fait, même quand l'auteur veut mettre quelque chose en évidence, on le fait avec des italiques. De tels caractères se fondent correctement dans le texte principal — de loin vous ne pourriez pas dire quelle page contient des italiques mais vous verriez immédiatement un seul mot en gras. Si les romans sont typographiés comme cela c'est pour suivre le paradigme : évitez ce qui perturbe.

Une bonne typographie (comme une bonne organisation) est quelque chose que l'on *ne remarque pas*. Le boulot de la typographie est de rendre la lecture du texte, c'est-à-dire l'absorption des informations qu'il contient, aussi aisée que possible. Pour un roman, le lecteur absorbe le contenu en lisant le texte ligne à ligne comme s'il écoutait quelqu'un lui racontant l'histoire. Dans ces circonstances tout ce qui dans la page empêche le regard d'avancer rapidement et régulièrement d'une ligne à l'autre rendra le texte plus difficile à lire.

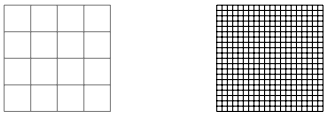
Maintenant, prenez un de vos magazines préférés ou un journal et regardez une page typique. Vous remarquez que beaucoup de choses « se passent » sur la page. Les polices sont utilisées à des tailles différentes et dans des arrangements différents, le texte est organisé en colonnes étroites, souvent entrelacées de photos. Si les magazines sont typographiés de la sorte c'est à cause du paradigme : attirez l'attention.

On ne lit pas un magazine comme on lit un roman. Au lieu de lire un magazine ligne à ligne, on utilise les titres et les résumés pour savoir si l'on veut lire ou pas un certain article. Le boulot de la typographie est d'attirer notre attention d'abord sur ces résumés et ces titres. Une fois que l'on a décidé de lire un article, toutefois, on ne tolère plus d'être distrait c'est pourquoi le texte principal de l'article est typographié de la même manière que celui d'un roman.

Les deux principes « évitez ce qui perturbe » et « attirez l'attention » s'appliquent aussi aux graphiques. Lorsque l'on crée un graphique, on devrait éliminer tout ce qui va « distraire l'œil ». En même temps, on devrait essayer d'aider activement le lecteur « à travers le graphique » en utilisant des polices/couleurs/épaisseurs de ligne qui mettent en relief les différentes parties.

Voici une liste non-exhaustive des choses qui peuvent perturber un lecteur :

– Les contrastes importants seront enregistrés les premiers par l'œil. Par exemple, considérons les deux grilles suivantes :



Bien que la grille de gauche soit la première dans l'ordre normal de lecture, la droite a la plus de chance d'être vue la première : le contraste blanc sur noir est plus grand que celui du gris au blanc. De plus, il y a plus de « places » ajoutant à l'effet global de contraste dans la grille de droite.

Les choses comme des grilles et, plus généralement, toutes les lignes d'aide ne devraient pas empoigner l'attention du lecteur et donc devraient être tracées avec un contraste faible avec l'arrière plan. Par ailleurs, une grille avec un maillage lâche perturbe moins qu'une grille au maillage serré.

– Les lignes de pointillés créent de nombreux points où apparaît un contraste du noir au blanc. Les pointillés et les tirets peuvent être très perturbants et, aussi, devraient être généralement évités.

Il ne faut pas utiliser des motifs différents de pointillés pour distinguer des courbes. De cette façon on perd des points de données et l'œil n'est pas particulièrement bon à « grouper les choses en fonction d'un motif de pointillés ». L'œil est *bien* meilleur à grouper en fonction des couleurs.

– Les motifs de fond, qui remplissent une surface avec des diagonales, des horizontales ou des verticales ou même des points, sont presque toujours perturbants et, en général, n'ont aucun but véritable.

– Les images de fond et les dégradés perturbent et il est rare qu'ils ajoutent quelque chose d'important au graphique.

– De jolis petits *cliparts* peuvent attirer l'attention loin des données.

5 Formats d'entrée et sortie

TEX a été conçu pour être flexible. C'est vrai autant pour l'*entrée* que pour la *sortie*. La présente section explique quels sont les formats d'entrée et comment PGF les prend en compte. On y explique aussi quels sont les différents formats de sortie qui peuvent être produits.

5.1 Formats d'entrée gérés

TEX ne fixe pas précisément comment doit être formatée l'entrée. Bien qu'il soit *habituel* de, par exemple, commencer un groupe dans TEX avec une accolade ce n'est absolument pas nécessaire. De la même manière, on a *l'habitude* de commencer un environnement avec `\begin` mais TEX se contrefiche radicalement du nom exact de la commande.

Bien que TEX puisse être reconfiguré, les utilisateurs, eux , non. Pour cette raison, certain *format d'entrée* définit un ensemble de commandes et de conventions pour formater l'entrée de TEX. Il y a, à ce jour, trois formats principaux : le format `plain`TEX originel de Donald Knuth ; le populaire format LATEX de Leslie Lamport et le format ConTEXt de Hans Hagen.

5.1.1 Utiliser le format LATEX

Il est facile d'utiliser PGF et Ti*k*Z avec le format LATEX. On écrit `\usepackage{pgf}` ou `\usepackage{tikz}`. Habituellement c'est tout ce que vous avez à faire. Toute la configuration sera faite automatiquement et (je l'espère) correctement.

Les fichiers de style utilisés avec le format LATEX sont situés dans le répertoire `latex/pgf/` du système PGF. Ce que font ces fichiers c'est principalement inclure des fichiers situés dans le répertoire `generic/pgf`. Par exemple, voici le contenu du fichier `latex/pgf/frontends/tikz.sty` :

```
% Copyright 2005 by Till Tantau <tantau@users.sourceforge.net>.
%
% This program can be redistributed and/or modified under the terms
% of the GNU Public License, version 2.

\RequirePackage{pgf,calc,pgffor,pgflibraryplothandlers,xkeyval}

\input{tikz.code}

\endinput
```

Le fichier du répertoire `generic/pgf` fait le vrai travail.

5.1.2 Utiliser le format Plain TEX

Lorsque l'on utilise le format plain TEX, on écrit `\input{pgf.tex}` ou `\input{tikz.tex}`. Au lieu de `\begin{pgfpicture}` et `\end{pgfpicture}`, on utilise `\pgfpicture` et `\endpicture`.

Contrairement à ce qu'il sait faire pour le format LATEX, PGF ne sait pas bien discerner la configuration adéquate pour le format plain TEX. En particulier, il ne déterminera correctement le format de sortie automatique que si vous utiliser `pdftex` ou `tex` plus `dvips`. Pour tout autre format de sortie on devra donner la valeur correcte à la macro `\pgfsysdriver`. Voir la description de l'utilisation des formats de sortie ci-dessous.

PGF fut à l'origine écrit pour une utilisation avec LATEX et cela se voit en de nombreux endroits. Toutefois, l'intégration avec plain TEX est raisonnablement bonne.

De même que les fichiers de style LATEX, les fichiers du format plain TEX comme `tikz.tex` ne font qu'inclure le fichier `tikz.code.tex` convenable.

5.1.3 Utiliser le format ConTEXt

Il n'y a pour l'heure aucune intégration avec le format ConTEXt. Ou plutôt, on doit utiliser PGF et Ti*k*Z comme dans le format plain TEX lorsque l'on utilise ConTEXt. Cela pourrait changer à l'avenir.

5.2 Formats de sortie gérés

Un format de sortie est un format dans lequel TEX produit le texte qu'il a typographié. Produire cette sortie est (conceptuellement) un processus en deux étapes :

1. TEX typographie votre texte et vos graphiques. Le résultat de cette opération est essentiellement une longue liste de paires lettre–coordonnées avec (éventuellement) quelques commandes « spéciales ». Cette longue liste de paires est écrite dans quelque chose qui s'appelle un fichier `.dvi`.
2. Un autre logiciel lit ce fichier `.dvi` est traduit les paires lettre–coordonnées en, disons, des commandes PostScript de placement de la lettre donnée aux coordonnées données.

L'exemple classique de ce processus est la combinaison de `latex` et `dvips`. Le programme `latex` (qui n'est rien d'autre que le programme `tex` avec les macros de LATEX préchargées) produit un fichier `.dvi` comme sortie. Le programme `dvips` prend cette sortie et produit un fichier `.pls` (PostScript). Éventuellement ce fichier est converti encore avec, par exemple, `ps2pdf` dont le nom est sensé signifier « PostScript vers PDF ». Un autre exemple de logiciels utilisant ce processus est la combinaison `tex dvipdfm`. Le logiciel `dvipdfm` prend un fichier `.dvi` en entrée et traduit les paires lettre–coordonnées qu'il contient en commandes PDF produisant directement un fichier `.pdf`. Enfin, `tex4ht` est aussi un logiciel qui prend un fichier `.dvi` est produit une sortie, cette fois un fichier `.html`. Les programmes `pdftex` et `pdflatex` sont particuliers : ils produisent directement un fichier `.pdf` sans passer par l'étape intermédiaire du `.dvi`. Toutefois, du point de vue du programmeur, ils se comportent exactement comme s'il y avait une étape intermédiaire.

Normalement, TEX ne produit que des paires lettre-coordonnées en sortie. Cela rend manifestement difficile le dessin d'un courbe, par exemple. Pour cela, on peut utiliser les commandes « spéciales ». Malheureusement, ces commandes spéciales ne sont pas les mêmes pour les différents logiciels qui transforment les fichiers `.dvi`. De fait, chaque logiciel qui prend un `.dvi` en entrée a une syntaxe totalement différente pour les commandes spéciales.

Une des tâches principales de PGF est de fournir une « abstraction au-dessus » de ces syntaxes des différents logiciels. Toutefois, cela signifie que la gestion de chaque logiciel doit être « programmée » ce qui est un processus long et complexe.

5.2.1 Sélectionner le pilote final

Lorsque TEX typographie votre document, il ne connaît pas quel logiciel vous allez utiliser pour transformer le fichier `.dvi`. Si votre `.dvi` ne contient aucune commande spéciale, cela ne posera pas de problème mais, de nos jours, presque tous les `.dvi` contiennent des tas de commandes spéciales. Il faut donc que l'on dise à TEX quel programme sera utilisé ensuite.

Malheureusement il n'y a pas de façon « standard » de le dire à TEX. Dans le format LATEX, il existe un mécanisme complexe à l'intérieur de l'extension `graphics` et PGF s'y raccroche. Pour les autres formats et quand on ne peut pas se raccrocher au mécanisme précédent de manière fiable, il faut dire directement à PGF quel programme on va utiliser. On le fait en redéfinissant la macro `\pgfsysdriver` avec une valeur adéquate *avant* de charger `pgf`. Si l'on utilise le programme `dvips`, on donne à la macro la valeur `pgfsys-dvips.def` ; si l'on utilise `pdftex` ou `pdflatex`, on donne la valeur `pgfsys-pdftex.def` et ainsi de suite. Dans ce qui suit, on traite des détails de la gestion des différents logiciels.

5.2.2 Produire du PDF

PGF gère trois programmes qui produisent du PDF (PDF signifie « portable document format » [*format de document portable*] et a été inventé par la société Adobe) : `dvipdfm`, `pdftex` et `vtex`. Le programme `pdflatex` est identique à `pdftex`, il utilise un format différent d'entrée mais la sortie est exactement la même.

Fichier `pgfsys-pdftex.def`

C'est le fichier de pilote à utiliser avec pdfTEX, c'est-à-dire avec la commande `pdftex` ou `pdflatex`. Il contient `pgfsys-common-pdf.def`.

Ce pilote est fonctionnellement « complet » ce qui signifie que tout ce que PGF « peut faire » est implanté dans ce pilote.

Fichier `pgfsys-dvipdfm.def`

C'est le fichier de pilote à utiliser avec (`la`)`tex` suivi de `dvipdfm`. Il contient `pgfsys-common-pdf.def`.

Ce pilote gère la plupart des fonctionnalités de PGF mais il a quelques limitations :

1. En mode LATEX il utilise `graphicx` pour l'inclusion de graphiques et ne gère pas le masquage.
2. En mode plain TEX il ne gère pas l'inclusion d'image.

Fichier `pgfsys-vtex.def`

C'est le fichier pilote à utiliser avec le programme commercial VTEX. Bien qu'il produise du PDF, il contient `pgfsys-common-postscript.def`. Remarquez que VTEX peut produire aussi bien du PostScript que du PDF. Toutefois, que l'on produise du PostScript ou du PDF ne change rien quant au pilote.

Ce pilote gère la plupart des fonctionnalités de PGF avec les limitations suivantes :

1. En mode LaTeX il utilise `graphicx` pour l'inclusion de graphiques et ne gère pas le masquage.
2. En mode plain TeX il ne gère pas l'inclusion d'image.
3. Le dégradé est totalement implanté mais aboutit à la même qualité que l'implantation pour `dvips`.
4. L'opacité n'est pas du tout implantée.

On peut également produire un fichier `.pdf` en produisant d'abord un fichier PostScript (voir ci-dessous) puis en utilisant un logiciel de conversion de PostScript vers PDF comme `ps2pdf` ou Acrobat Distiller.

5.2.3 Produire du PostScript

Fichier `pgfsys-dvips.def`

C'est le fichier pilote à utiliser avec (`la`)`tex` suivi de `dvips`. Il contient `pgfsys-common-postscript.def`.

Ce pilote aussi gère la plupart des fonctionnalités de PGF avec les limitations suivantes :

1. En mode LaTeX il utilise `graphicx` pour l'inclusion de graphiques et ne gère pas le masquage.
2. En mode plain TeX il ne gère pas l'inclusion d'image.
3. Le dégradé est totalement implanté mais le résultat n'est pas aussi bon qu'avec un pilote produisant un `.pdf`
4. L'opacité ne marche qu'avec les toutes dernières versions de GhostScript.

Fichier `pgfsys-textures.def`

C'est un fichier pilote à utiliser avec le programme TEXTURE. Il contient `pgfsys-common-postscript.def`.

Les limitations de ce pilote sont exactement les mêmes que celle du pilote pour `dvips`.

On peut également utiliser le programme `vtex` avec `pgfsys-vtex.def` pour produire du PostScript.

5.2.4 Produire du HTML / SVG

Le programme `tex4ht` convertit des fichiers `.dvi` en fichiers `.html`. Bien que l'on ne puisse pas dessiner avec le format HTML, on peut le faire avec le format SVG. Avec le pilote suivant on peut demander à PGF de produire une image SVG pour chaque graphique PGF du texte.

Fichier `pgfsys-tex4ht.def`

C'est le fichier pilote à utiliser avec le programme `text4ht`. Il contient `pgfsys-common-svg.def`.

Lorsque l'on utilise ce pilote, on doit être conscient des limitations suivantes :

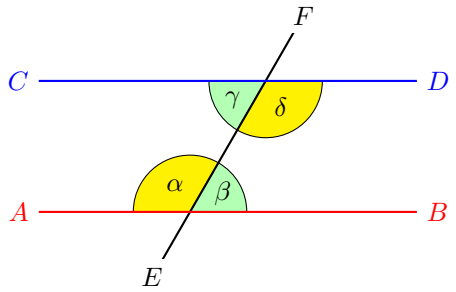
1. En mode LaTeX il utilise `graphicx` pour l'inclusion de graphiques et ne gère pas le masquage.
2. En mode plain TeX il ne gère pas l'inclusion d'image.
3. Le texte contenu dans les images `pgf` n'est pas très bien géré. Cela vient de ce que les définitions actuelles de SVG ne gère pas très bien le texte et qu'il n'est pas possible de « s'échapper » vers le HTML correctement. On peut espérer que tous ces problèmes disparaîtront à l'avenir mais, pour l'instant, seules deux sortes de texte marchent à peu près bien : d'abord le texte simple sans mode mathématique ni caractère spéciaux ni rien de spécial ; ensuite, du texte mathématique *très* simple qui contient des exposants ou des indices. Même alors les variables ne sont pas écrites joliment en italique et, en général, l'apparence du texte n'est simplement pas très bonne.
4. Si l'on utilise un texte qui contient n'importe quoi de spécial, même quelque chose d'aussi simple que `$\alpha$`, cela peut corrompre le graphique car `text4ht` ne produit pas toujours un code XML valide. Donc, une fois de plus, *il faut se limiter à du texte très simple dans un graphique.* Désolé.
5. Au contraire de ce qui se passe dans d'autres formats de sortie, la boite-cadre (*bounding box*) découpe vraiment la figure.

Le pilote fonctionne grossièrement comme suit : quand on commence une `{pgfpicture}`, les commandes `\special` idoines sont utilisées pour rediriger la sortie de `tex4ht` vers un nouveau fichier appelé `\jobname-xxx.svg` où `xxx` est un nombre incrémenté à chaque nouveau graphique. Puis, jusqu'à la fin de la figure, chaque commande (de la couche système) crée un « spécial » qui insère le texte littéral

SVG approprié dans le fichier de sortie. Les détails exacts sont un peu plus complexes puisque le modèle d'image et de processus de PostScript/PDF et ceux de SVG ne sont pas tout à fait les mêmes mais ils sont suffisamment proches pour les buts de PGF.

Deuxième partie

TikZ n'est *pas* un programme de dessin



Quand nous faisons l'hypothèse que AB et CD sont parallèles, c.-à-d. $AB \parallel CD$, alors $\alpha = \delta$ et $\beta = \gamma$.

```
\begin{tikzpicture}
  \draw[fill=yellow] (0,0) -- (60:.75cm) arc (60:180:.75cm);
  \draw(120:0.4cm) node {$\alpha$};

  \draw[fill=green!30] (0,0) -- (right:.75cm) arc (0:60:.75cm);
  \draw(30:0.5cm) node {$\beta$};

  \begin{scope}[shift={(60:2cm)}]
    \draw[fill=green!30] (0,0) -- (180:.75cm) arc (180:240:.75cm);
    \draw (30:-0.5cm) node {$\gamma$};

    \draw[fill=yellow] (0,0) -- (240:.75cm) arc (240:360:.75cm);
    \draw (-60:0.4cm) node {$\delta$};
  \end{scope}

  \begin{scope}[thick]
    \draw  (60:-1cm) node[fill=white] {$E$} -- (60:3cm) node[fill=white] {$F$};
    \draw[red]                    (-2,0) node[left] {$A$} -- (3,0) node[right]{$B$};
    \draw[blue,shift={(60:2cm)}] (-3,0) node[left] {$C$} -- (2,0) node[right]{$D$};

    \draw[shift={(60:1cm)},xshift=4cm]
    node [right,text width=6cm,rounded corners,fill=red!20,inner sep=1ex]
    {
      Quand nous faisons l'hypoth\'ese que $\color{red}AB$ et $\color{blue}CD$ sont
      parall\'eles, c.-\'a-d. ${\color{red}AB} \mathbin{\|} \color{blue}CD$,
      alors $\alpha = \delta$ et $\beta = \gamma$.
    };
  \end{scope}
\end{tikzpicture}
```

6 Principes de conception

Cette section décrit les principes de conception qui sous-tendent l'interface Ti*k*Z où Ti*k*Z signifie « Ti*k*Z ist *kein* Zeichenprogramm[9] ». Pour utiliser Ti*k*Z, un utilisateur de LaTeX écrit `\usepackage{tikz}` quelque part dans le préambule, un utilisateur de TeX écrit `\input tikz.tex`. Le boulot de Ti*k*Z est de vous faciliter la vie en fournissant une syntaxe facile à apprendre et à utiliser pour décrire des graphiques.

Les commandes et la syntaxe de Ti*k*Z ont influencées par de nombreuses sources. Le noms des commandes de base et la notion d'opérations de chemin sont pris dans METAFONT, le mécanisme d'options vient de PSTICKS, la notion de style est une réminiscence de SVG. Pour que tout cela marche ensemble, il faut quelques compromis. J'ai aussi ajouté quelques idées de mon cru, comme les méta-flèches et les transformations de coordonnées.

Les principes de conception de base suivant sous-tendent Ti*k*Z :

1. Syntaxe spéciale pour définir des points ;
2. Syntaxe spéciale pour définir des chemins ;
3. Actions sur les chemins ;
4. Syntaxe clé-valeur pour les paramètres graphiques ;
5. Syntaxe spéciale pour les nœuds ;
6. Syntaxe spéciale pour les arbres ;
7. Groupement de paramètres graphiques ;
8. Système de transformation de coordonnées.

6.1 Syntaxe spéciale pour définir des points

Ti*k*Z fournit une syntaxe spéciale pour définir des points et des coordonnées. Dans le cas le plus simple, on fournit deux dimensions TeXiques, séparées par une virgule et placées entre parenthèses comme ici `(1cm,2pt)`.

On peut aussi définir un point en coordonnées polaires en utilisant un deux-points au lieu de la virgule comme dans `(30:1cm)` qui signifie « 1 cm dans la direction 30 degrés ».

Si l'on ne fournit pas d'unité comme dans `(2,1)` on définit un point dans le système de coordonnées xy de PGF. Par défaut, le vecteur-x unité va 1 cm vers la droite et le vecteur-y unité va 1 cm vers le haut.

En donnant trois nombres comme dans `(1,1,1)` on définit un point dans le système de coordonnées xyz de PGF.

On peut aussi utiliser une ancre d'une forme définie précédemment comme dans `(first node.south)`.

On peut ajouter deux signes plus avant des coordonnées comme dans `++(1cm,0pt)`. Cela signifie « 1 cm vers la droite du dernier point utilisé ». Cela permet de définir aisément des mouvements relatifs. Par exemple, `(1,0) ++(1,0) ++(0,1)` définit les trois paires de coordonnées `(1,0)` puis `(2,0)` et `(2,1)`.

Enfin, au lieu de deux signes plus, on peut n'ajouter qu'un seul. Cela définit aussi un point de manière relative mais cela ne « change » pas le point courant utilisé dans les commandes suivantes. Par exemple, `(1,0) +(1,0) +(0,1)` définit les trois points `(1,0)` puis `(2,0)` et `(1,1)`.

6.2 Syntaxe spéciale de définitions de chemins

Quand on crée une figure avec Ti*k*Z, le travail principal est la définition de *chemins*. Un chemin est une suite de lignes droites ou courbes qui n'ont pas besoin d'être connectées. Ti*k*Z facilite la définition de chemin, en utilisant partiellement la syntaxe de METAPOST. Par exemple, pour définir un chemin triangulaire on utilise

```
(5pt,0pt) -- (0pt,0pt) -- (0pt,5pt) -- cycle
```

et l'on obtient ◺ lorsqu'on le trace.

6.3 Actions sur les chemins

Un chemin est une suite de lignes droites et courbes mais on n'a pas encore dit ce qui devait lui arriver. On peut *dessiner* (*draw*) un chemin, le *remplir* (*fill*), l'*estomper* (*shade*[10]), le *découper* (*clip*) ou faire une

9. NdTdS : Ti*k*Z n'est pas un logiciel de Dessin Assisté par Ordinateur.
10. NdTdS : Pour être exact, il faudrait traduire « shade » par « appliquer un dégradé etc. ». Même si les deux actions ne sont pas équivalentes, j'utiliserai « estomper » comme un raccourci commode.

quelconque combinaison de ces actions. Dessiner (alias *tracer* [*stroke*]) peut être vu comme l'action de prendre un crayon d'une certaine épaisseur et de le déplacer le long du chemin, par là dessiner sur la toile (*canvas*). Remplir signifie que l'intérieur du chemin est colorier de façon uniforme. Évidemment, remplir n'a de sens que si le chemin est *fermé* et un chemin est fermé automatiquement avant d'être rempli si nécessaire.

Un chemin, comme `\path (0,0) rectangle (2ex,1ex);`, étant donné, on peut le dessiner avec l'option `draw` comme dans `\path[draw] (0,0) rectangle (2ex,1ex);`, qui produit ▭. La commande `\draw` n'est qu'une abréviation pour `\path[draw]`. Pour remplir un chemin, on utilise l'option `fill` ou la commande `\fill`. On estompe avec l'option `shade` (on dispose des abréviations `\shade` et `\drawshade`) et on découpe avec l'option `clip`. Il y a aussi la commande `\clip` qui fait la même chose que `\path[clip]`, amis pas de commande comme `\drawclip`. On utilisera à la place `\draw[clip]` ou `\path[draw,clip]` par exemple.

Toutes ces commandes ne peuvent être utilisées que dans un environnement `{tikzpicture}`.

Ti*k*Z permet d'utiliser des couleurs différentes pour tracer et remplir.

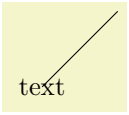
6.4 Syntaxe clé-valeur pour les paramètres graphiques

Chaque fois que Ti*k*Z dessine ou remplit un chemin, un grand nombre de paramètres graphiques influent sur le rendu. Parmi les exemples on citera les couleurs utilisées, le motif de pointillés, la surface découpée, l'épaisseur du trait et bien d'autres. Dans Ti*k*Z toutes ces options sont définies à l'aide d'une liste de paires clé-valeur comme dans `color=red`, passée comme paramètres facultatifs aux commandes de dessin et remplissage de chemin. Cet emploi rappelle celui de PSTRICKS. Par exemple, le code suivant dessinera un triangle rouge épais ;


```
\tikz \draw[line width=2pt,color=red] (1,0) -- (0,0) -- (1,0) -- cycle;
```

6.5 Syntaxe spéciale pour la définition de nœuds

Ti*k*Z introduit une syntaxe spéciale pour ajouter du texte ou, plus généralement, des nœuds à un graphique. Lorsque l'on définit un chemin, on peut ajouter des nœuds comme dans l'exemple suivant :




```
\tikz \draw (1,1) node {text} -- (2,2);
```

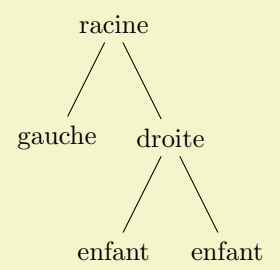
Les nœuds sont insérés à la position courante du chemin mais seulement *après* que le chemin a été rendu. Lorsque des options spéciales sont données, comme dans `\draw (1,1) node[circle,draw] {text};`, le texte n'est pas simplement placer à la position courante. Plutôt, il est entouré d'un cercle et ce cercle est « dessiné ».

On peut ajouter un nom à un nœud en vue d'une référence ultérieure soit avec l'option `name=`⟨*nom du nœud*⟩ soit en écrivant le nom du nœud entre parenthèse en dehors du texte comme dans `node[circle](nom){texte}`.

Les formes prédéfinies contiennent `rectangle`, `circle` et `ellipse` mais il est possible (quoique un peu ambitieux) de définir de nouvelles formes.

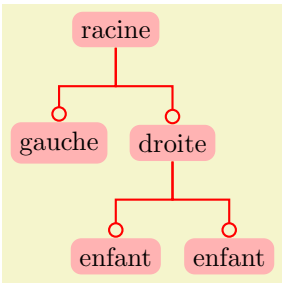
6.6 Syntaxe spéciale pour la définition des arbres

En plus de la « syntaxe pour nœuds », Ti*k*Z apporte aussi une syntaxe spéciale pour dessiner des arbres. Cette syntaxe est intégrée à la syntaxe spéciale pour les nœuds et on ne doit se souvenir que de quelques nouvelles commandes. Essentiellement, un `node` (nœud) peut être suivi d'un nombre quelconque d'enfants, chacun d'eux introduit par le mot-clé `child` (*enfant*). Les enfants sont eux-mêmes des nœuds et chacun d'eux peut avoir, à son tour, des enfants.

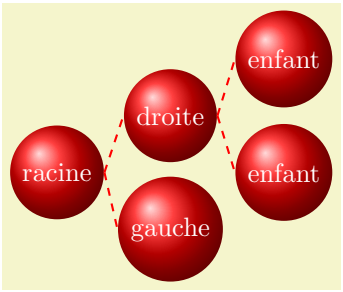



```
\begin{tikzpicture}
  \node {racine}
    child {node {gauche}}
    child {node {droite}
      child {node {enfant}}
      child {node {enfant}}
    };
\end{tikzpicture}
```

Puisque les arbres sont faits de nœuds, on peut utiliser les options pour modifier la façon dont les arbres sont dessinés. Voici deux exemples de l'arbre ci-dessus redessiné avec des options différentes :




```
\begin{tikzpicture}[edge from parent fork down]
  \tikzstyle{every node}=[fill=red!30,rounded corners]
  \tikzstyle{edge from parent}=[red,-o,thick,draw]
  \node {racine}
      child {node {gauche}}
      child {node {droite}
        child {node {enfant}}
        child {node {enfant}}
      };
\end{tikzpicture}
```




```
\begin{tikzpicture}
    [parent anchor=east,child anchor=west,grow=east]
  \tikzstyle{every node}=[ball color=red,circle,text=white]
  \tikzstyle{edge from parent}=[draw,dashed,thick,red]
  \node {racine}
      child {node {gauche}}
      child {node {droite}
        child {node {enfant}}
        child {node {enfant}}
      };
\end{tikzpicture}
```

6.7 Groupement de paramètres graphiques

Les paramètres graphiques devraient souvent s'appliquer à de nombreuses commandes de dessin ou de remplissage de chemin. Par exemple, on peut vouloir dessiner de nombreuses lignes toutes de la même épaisseur de 1 pt. Pour cela, on peut placer ces commandes dans un environnement `{scope}` qui prend, comme paramètre facultatif, les options graphiques voulues. Naturellement, les options graphiques définies ne s'appliquent qu'aux commandes de dessin et remplissage contenues dans l'environnement. De plus, des environnements `{scope}` emboîtés ou des commandes de dessin peuvent annuler ou redéfinir les options de l'environnement `{scope}` extérieur. Dans l'exemple suivant, trois lignes rouges, deux vertes et une bleue sont tracées :

```
\begin{tikzpicture}
  \begin{scope}[color=red]
    \draw (0mm,10mm) -- (10mm,10mm);
    \draw (0mm, 8mm) -- (10mm, 8mm);
    \draw (0mm, 6mm) -- (10mm, 6mm);
  \end{scope}
  \begin{scope}[color=green]
    \draw              (0mm, 4mm) -- (10mm, 4mm);
    \draw              (0mm, 2mm) -- (10mm, 2mm);
    \draw[color=blue] (0mm, 0mm) -- (10mm, 0mm);
  \end{scope}
\end{tikzpicture}
```

L'environnement `{tikzpicture}` lui-même se comporte comme un environnement `{scope}`, c'est-à-dire que l'on peut définir des paramètres graphiques avec un argument facultatif. Ces options s'appliquent à toutes les commandes de la figure.

6.8 Système de transformations des coordonnées

Ti*k*Z s'appuie entièrement sur le système de transformations des *coordonnées* de PGF pour réaliser des transformations. PGF gère également des transformations de la *toile*, un système de transformation de plus bas niveau, mais ce système n'est pas accessible depuis Ti*k*Z. Il y a cela deux raisons : premièrement, les transformations de la toile doivent être utilisées avec beaucoup de prudence et produisent souvent de « mauvais » graphiques en changeant l'épaisseur des lignes et des textes de façon inadéquate. De plus, PGF perd la trace des nœuds et des formes lorsque de telles transformations de la toile sont utilisées.

Pour plus de détails sur les différences entre les transformations de coordonnées et celles de la toile voir la section **??**, p. **??**.

7 La structure hiérarchique : Extension, environnements, portées et styles

La présente section explique comment les fichiers devraient être structurés quand on utilise Ti*k*Z. Au niveau supérieur, on doit inclure l'extension `tikz`. Dans le texte principal, chaque graphique doit être placé dans un environnement `{tikzpicture}`. Dans ces environnements, on peut utiliser des environnements `{scope}` pour créer des groupes internes. Dans ces portées (*scope*) on utilise les commandes `\path` pour dessiner effectivement quelque chose. À tous les niveaux, (sauf à celui de l'extension), on peut passer des options graphiques qui s'appliquent à tout ce qui est placé dans l'environnement.

7.1 Charger l'extension

```
\usepackage{tikz} % LaTeX
\input tikz.tex   % plain TeX
\input tikz.tex   % ConTeXt
```

Cette extension n'a pas d'option.

Cela chargera automatiquement l'extension PGF et plusieurs autres trucs dont Ti*k*Z a besoin (comme l'extension `xkeyval`).

PGF doit savoir quel pilote TEX on a l'intention d'utiliser. Dans la plupart des cas PGF est assez intelligent pour déterminer correctement le pilote tout seul ; c'est vrai en particulier si l'on utilise LATEX. À ce jour la seule situation où PGF ne peut connaître « de lui-même » le pilote est quand on utilise plain TEX ou ConTEXt avec `dvipdfm`. Dans ce cas on doit écrire `\def\pgfsysdriver{pgfsys-dvipdfm.def}` avant d'inclure `tikz.tex`.

7.2 Créer une figure

7.2.1 Créer une figure à l'aide d'un environnement

La portée la plus externe de Ti*k*Z est l'environnement `{tikzpicture}`. On ne peut utiliser les commandes de dessin que dans cet environnement, à l'extérieur (comme c'est possible dans de nombreuses autres extensions) elles produiront le chaos.

Dans Ti*k*Z, la façon dont les graphiques sont rendus est très fortement influencée par les options graphiques. Par exemple, il y a une option pour fixer la couleur des dessins, une autre pour définir la couleur utilisée pour remplir, et aussi d'autres plus obscures comme l'option pour définir le préfixe utilisé dans les noms des fichiers temporaires créés pendant la tabulation des fonctions par un programme externe. Les options graphiques sont presque toujours définies par une paire clé-valeur. (Le « presque toujours » fait référence au nom des nœuds, qui peut être défini autrement, voir6.5, p. 45.) Toutes les options graphiques sont locales à l'environnement `{tikzpicture}` auquel elles s'appliquent.

```
\begin{tikzpicture}[⟨options⟩]
  ⟨contenu de l'environnement⟩
\end{tikzpicture}
```

Toutes les commandes de Ti*k*Z devraient être données dans un tel environnement sauf la commande `\tikzstyle`. Contrairement à ce qui se passe avec d'autres extensions, on ne peut pas utiliser, par exemple, `\pgfpathmoveto` en dehors d'un tel environnement sans créer le chaos. Comme les commandes de Ti*k*Z, comme `\path`, ne sont définies qu'à l'intérieur de l'environnement, il est peu probable que l'on puisse y faire quelque chose de mal.

Quand cet environnement est rencontré, les ⟨*options*⟩ sont analysées. Toutes les options passées là s'appliqueront à toute la figure.

Ensuite, le contenu de l'environnement est traité et les commandes graphiques qu'il contient sont placées dans une boite. Le texte non-graphique est supprimé autant que se peut, mais les commandes non-PGF placées dans un environnement `{tikzpicture}` ne devraient produire aucune « sortie » car cela risquerait de brouiller le système de positionnement des pilotes terminaux. La suppression du texte normal, au fait, est faite en positionnant temporairement la police courante sur `\nullfont`. On peut, toutefois, « échapper » vers la typographie normale de TEX. Cela arrive, par exemple, quand on définit un nœud.

À la fin de l'environnement, PGF essaie de deviner correctement les dimension de la boîte-cadre (*bounding box*) du graphique et retaille la boite afin qu'elle ait ces dimensions. Pour « deviner », PGF, à chaque fois qu'il rencontre des coordonnées, met à jour les dimensions de la boite-cadre afin qu'elle contienne

tous les points dont il connaît les coordonnées. Cela donnera généralement une bonne approximation de la boite-cadre mais cela ne sera pas toujours exact. D'abord, l'épaisseur des lignes n'est pas prise en compte. De plus, les points de contrôle des courbes sont souvent loin « à l'extérieur » de la courbe et font que la boite-cadre est trop grande. Dans un tel cas, on doit utiliser l'option `[use as bounding box]`.

Les options suivantes influencent la ligne de base des figures produites :

– `baseline=`⟨*dimension*⟩ Normalement, la base de la figure est placée sur la ligne de base du texte qui l'entoure. Par exemple, quand on écrit le code `\tikz\draw(0,0)circle(.5ex);`, PGF trouvera que la base de la figure est à −.5ex et que le sommet est à .5ex. Aussi, la base sera placée sur la ligne de base, ce qui produira ceci : ○.
Avec cette option, on peut imposer à la figure d'être rehaussée ou rabaissée de telle sorte que la hauteur ⟨*dimension*⟩ soit sur la ligne de base. Par exemple, `\tikz[baseline=0pt]\draw(0,0)circle(.5ex);` produit ○ puisque, maintenant, la ligne de base est à la hauteur de l'axe des x. Si on omet les ⟨*dimensions*⟩, PGF suppose qu'elles valent `0pt`.
Cette option est souvent utile avec les graphiques « en-ligne » comme dans

$A \twoheadrightarrow B$ `$A \mathbin{\tikz[baseline] \draw[->>] (0pt,.5ex) -- (3ex,.5ex);} B$`

– `execute at begin picture=`⟨*code*⟩ On peut utiliser cette option pour préparer du code qui sera exécuté au début de la figure. On doit donner cette option dans l'argument de l'environnement `{tikzpicture}` lui-même car elle n'aura pas d'effet autrement. Après tout, par la suite, la figure a déjà commencé.
Cette option est principalement utilisée dans les styles tels que le style `every picture` pour exécuter un certain code au commencement d'une figure.

– `execute at end picture=`⟨*code*⟩ Cette option prépare du code qui sera exécuté à la fin de la figure. Si l'on utilise cette option plusieurs fois, le code sera accumulé. Cette option aussi doit être placée dans l'argument facultatif de l'environnement `{tikzpicture}`.


```
\begin{tikzpicture}[execute at end picture=%
  {
    \begin{pgfonlayer}{background}
      \path[fill=yellow,rounded corners]
        (current bounding box.south west) rectangle
        (current bounding box.north east);
    \end{pgfonlayer}
  }]
  \node at (0,0) {X};
  \node at (2,1) {Y};
\end{tikzpicture}
```

Toutes les options « finissent » à la fin de la figure. Pour définir une option « globalement » on peut utiliser le style suivant :

– `style=every picture` Ce style est placé au début de chaque figure.

```
\tikzstyle{every picture}=[semithick]
```

En plain TEX, on doit utiliser la commande suivante à la place :

`\tikzpicture[`⟨*options*⟩`]`
⟨*contenu de l'environnement*⟩
`\endtikzpicture`

7.2.2 Créer une figure à l'aide d'une commande

Les deux commandes suivantes sont utilisées pour de « petits » graphiques.

`\tikz[`⟨*options*⟩`]{`⟨*commandes*⟩`}`

Cette commande place les ⟨*commandes*⟩ à l'intérieur d'un environnement `{tikzpicture}` et ajoute un point-virgule à la fin. Ce n'est qu'une facilité.

La ⟨*commande*⟩ ne doit pas contenir de paragraphe (une ligne vide). Ce n'est qu'une précaution prise pour s'assurer que les utilisateurs ne s'en serve vraiment que pour des petits graphiques.

Exemple : `\tikz{\draw (0,0) rectangle (2ex,1ex)}` produit ▭

`\tikz[`⟨*options*⟩`]`⟨*texte*⟩ `;`

Si le ⟨*texte*⟩ ne commence pas par une parenthèse ouvrante, la fin du ⟨*texte*⟩ est le prochain point-virgule.

Exemple : `\tikz \draw (0,0) rectangle (2ex,1ex);` produit ▭

7.2.3 Ajouter un arrière-plan

Par défaut les figures n'ont pas d'arrière-plan, c'est-à-dire, qu'elles sont transparentes partout où l'on n'a pas dessiné quelque chose. On peut au contraire désirer avoir un arrière-plan coloré sous la figure, un cadre noir autour, des lignes au-dessus et au-dessous ou encore toute autre sorte de décoration.

Comme on n'a pas souvent besoin d'un arrière-plan, la définition des styles permettant d'ajouter un arrière-plan a été placée dans la bibliothèque d'extension `pgflibrarytikzbackgrounds`. Cette bibliothèque est décrite dans la Section **??**, p. **??**.

7.3 Utiliser les portées pour structurer une figure

Dans un environnement `{tikzpicture}` on peut créer des portées à l'aide de l'environnement `{scope}`. Cet environnement n'est disponible qu'à l'intérieur d'un environnement `{tikzpicture}` aussi, une fois de plus, il y a peu de chance de faire quelque chose de travers.

`\begin{scope}[`⟨*options*⟩`]`
⟨*contenu de l'environnement*⟩
`\end{scope}`

Toutes les ⟨*options*⟩ sont locales au ⟨*contenu de l'environnement*⟩. De plus, les découpages de chemins sont aussi locaux à l'environnement, c-à-d. que tout découpage fait dans l'environnement « finit » à la fin de celui-ci.

```
\begin{tikzpicture}
  \begin{scope}[red]
    \draw (0mm,0mm) -- (10mm,0mm);
    \draw (0mm,1mm) -- (10mm,1mm);
  \end{scope}
  \draw (0mm,2mm) -- (10mm,2mm);
  \begin{scope}[green]
    \draw (0mm,3mm) -- (10mm,3mm);
    \draw (0mm,4mm) -- (10mm,4mm);
    \draw[blue] (0mm,5mm) -- (10mm,5mm);
  \end{scope}
\end{tikzpicture}
```

Les styles suivants influent sur les portées :

– `style=every scope` Ce style est installé au début de chaque portée. Je ne sais pas vraiment à quoi ça peut servir mais qui sait ?

Les options suivantes sont utiles dans les portées :

– `execute at begin scope=`⟨*code*⟩ Cette option installe du code qui sera exécuté au début de la portée. Cette option doit être donnée dans l'argument de l'environnement `{scope}`.
L'effet ne s'applique qu'à la portée courante et pas aux sous-portées.

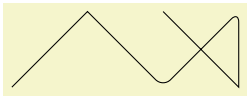
– `execute at end scope=`⟨*code*⟩ Cette option installe du code qui sera exécuté à la fin de la portée. L'utilisation répétée de cette option fera que le code sera accumulé. Cette option aussi doit être donnée dans l'argument de l'environnement `{scope}`.
Là encore, l'effet ne s'applique qu'à la portée courante et pas aux sous-portées.

En plain TEX, on utilisera les commandes suivantes en remplacement :

`\scope[`⟨*options*⟩`]`
⟨*contenu de l'environnement*⟩
`\endscope`

7.4 Utiliser les portées dans les chemins

La commande `\path`, que l'on décrira en détail dans une section à venir, prend aussi des options graphiques. Ces options sont locales au chemin. De plus, on peut créer des portées locales à l'intérieur d'un chemin simplement à l'aide d'accolades comme dans



```
\tikz \draw (0,0) -- (1,1)
          {[rounded corners] -- (2,0) -- (3,1)}
          -- (3,0) -- (2,1);
```

On remarquera que de nombreuses options s'appliquent au chemin tout entier et que l'on ne peut pas en limiter la portée de cette façon. Par exemple, on ne peut pas limiter la portée de la `color` (*couleur*) d'un chemin. Voir les explications dans la section sur les chemins pour plus de détails.

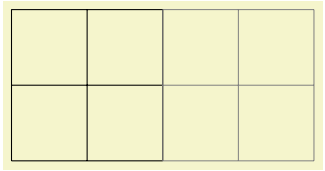
Enfin, certains éléments que l'on déclare dans l'argument de la commande `\path` prennent aussi des options locales. Par exemple, une définition de nœud prend des options. Dans ce cas, les options ne s'appliquent qu'au nœud et pas au chemin qui l'entoure.

7.5 Utiliser les styles pour gérer l'apparence des figures

Il y a une manière d'organiser des ensembles d'options graphiques « orthogonale » au mécanisme normal de portée. Par exemple, on peut vouloir que toutes les « lignes d'aide » soient tracées d'une certaine façon, disons, grises et fines (*ne pas* utiliser de pointillés cela dérange le lecteur). Pour cela on peut utiliser des *styles*.

Un style est simplement un ensemble d'options graphiques que l'on a prédéfini à un certain moment. Une fois défini, un style peut être utilisé n'importe où avec l'option `style`.

– `style=`⟨*nom de style*⟩ invoque toutes les options qui sont actuellement groupées dans le ⟨*nom de style*⟩. Un exemple de style est le style prédéfini `help lines` que l'on devrait utiliser pour les lignes d'arrière-plan comme les lignes d'un quadrillage ou les tracés de construction. On peut facilement définir de nouveau styles et modifier ceux qui existent.

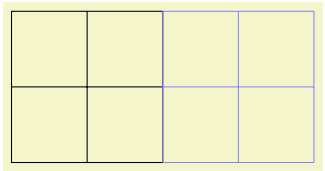


```
\begin{tikzpicture}
  \draw                    (0,0) grid +(2,2);
  \draw[style=help lines] (2,0) grid +(2,2);
\end{tikzpicture}
```

`\tikzstyle`⟨*nom de style*⟩`+=[`⟨*options*⟩`]`

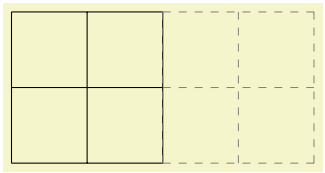
Cette commande définit le style ⟨*nom de style*⟩. Chaque fois que l'on utilisera la commande `style=`⟨*nom de style*⟩, les ⟨*options*⟩ seront appelées. Il est permis à un style d'en appeler un autre avec la commande `style=` dans les ⟨*options*⟩, ce qui permet de bâtir une hiérarchie de styles. Bien entendu, on *ne devra pas* créer de dépendances circulaires [11].

Si le style avait déjà une définition, il sera redéfini sans cérémonie ni avertissement.



```
\tikzstyle{help lines}=[blue!50,very thin]
\begin{tikzpicture}
  \draw                    (0,0) grid +(2,2);
  \draw[style=help lines] (2,0) grid +(2,2);
\end{tikzpicture}
```

Si on donne le `+` facultatif, les options sont *ajoutées* à la définition existante :

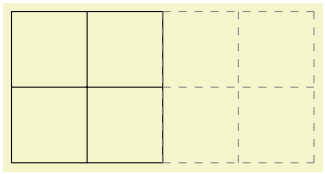


```
\tikzstyle{help lines}+=[dashed]% aaarghhh!!!
\begin{tikzpicture}
  \draw                    (0,0) grid +(2,2);
  \draw[style=help lines] (2,0) grid +(2,2);
\end{tikzpicture}
```

On peut également définir un style à l'aide d'une option :

– `set style={{`⟨*nom de style*⟩`}+=[`⟨*options*⟩`]}` Cette option a le même effet que la déclaration `\tikzstyle` devant l'argument de l'option.

11. NdTdS : Autrement dit, un style A ne peut appeler un style B qui lui même appelle un style . . . qui appelle le style A.



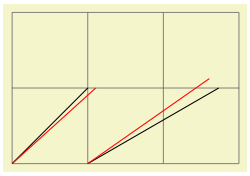
```
\begin{tikzpicture}[set style={{help lines}+=[dashed]}]
  \draw                   (0,0) grid +(2,2);
  \draw[style=help lines] (2,0) grid +(2,2);
\end{tikzpicture}
```

8 Définir des coordonnées

8.1 Coordonnées et options de coordonnées

Une *coordonnée*[12] est une position dans une figure. Ti*k*Z utilise une syntaxe spéciale pour définir les coordonnées. Les coordonnées sont toujours placées entre parenthèses. La syntaxe générale est `(`[`⟨`*options*`⟩`]`⟨`*spécification de coordonnée*`⟩)`.

On peut passer des options qui ne s'appliquent qu'à une seule coordonnée, bien que cela ne se justifie que pour des options de transformation. Pour passer des options de transformation à une seule coordonnée, on écrit ces options au début entre crochets :



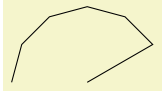
```
\begin{tikzpicture}
  \draw[style=help lines] (0,0) grid (3,2);
  \draw      (0,0) -- (1,1);
  \draw[red] (0,0) -- ([xshift=3pt] 1,1);
  \draw      (1,0) -- +(30:2cm);
  \draw[red] (1,0) -- +([shift=(135:5pt)] 30:2cm);
\end{tikzpicture}
```

8.2 Coordonnées simples

La façon la plus simple de définir des coordonnées est d'utiliser une paire de dimensions de TEX séparées par une virgule comme dans `(1cm,2pt)` ou `(2cm,\textheight)`. Comme on peut le voir, on peut mélanger différentes unités. Les coordonnées définies de cette façon signifient « 1 cm vers la droite et 2 pt vers le haut depuis l'origine de la figure ». On peut également écrire quelque chose comme `(1cm+2pt,2pt)` puisque l'extension `calc` est utilisée.

8.3 Coordonnées polaires

On peut aussi définir des coordonnées en coordonnées polaires. Dans ce cas, on définit un angle et une distance, séparée par un deux-point comme dans `(30:1cm)`. L'angle doit toujours être donné en degrés et être compris entre −360 et 720.



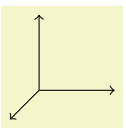
```
\tikz \draw    (0cm,0cm) -- (30:1cm) -- (60:1cm) -- (90:1cm)
            -- (120:1cm) -- (150:1cm) -- (180:1cm);
```

Au lieu de donner l'angle comme un nombre on peut aussi utiliser certains mots. Par exemple, `up` est la même chose que `90`, ainsi on peut écrire `\tikz \draw (0,0) -- (2ex,0pt) -- +(up:1ex);` et obtenir ⌴. Outre `up`, on peut aussi utiliser `down`, `left`, `right`, `north`, `south`, `west`, `east`, `north east`, `north west`, `south east`, `south west` qui ont tous leurs significations naturelles (c-à-d., en français, `haut`, `bas`, `gauche`, `droite`, `nord`, `sud`, `ouest`, `est`, `nord-est`, `nord-ouest`, `sud-est` et `sud-ouest`).

8.4 Coordonnées xy et xyz

On peut définir les coordonnées dans le système de coordonnées xy de PGF. Dans ce cas, on fournit deux nombres sans unités, séparés par une virgule comme dans `(2,-3)`. Cela signifie « ajoute deux fois le vecteur suivant x courant de PGF et soustrait trois fois le vecteur suivant y ». Par défaut, le vecteur suivant x pointe 1 cm vers la droite et le vecteur suivant y de 1 cm vers le haut mais on peut changer cela de manière arbitraire à l'aide des options graphiques `x` et `y`.

De la même manière on peut définir les coordonnées dans le système de coordonnées xyz de PGF. La seule différence avec les coordonnées xy est que l'on spécifie trois nombres séparés par des virgules comme dans `(1,2,3)`. Cela est interprété comme « une fois le vecteur x plus deux fois le vecteur y plus trois fois le vecteur z ». Par défaut le vecteur suivant z pointe vers $\left(-\frac{1}{\sqrt{2}}\text{ cm}, -\frac{1}{\sqrt{2}}\text{ cm}\right)$. Voyons l'exemple suivant :



```
\begin{tikzpicture}[->]
  \draw (0,0,0) -- (1,0,0);
  \draw (0,0,0) -- (0,1,0);
  \draw (0,0,0) -- (0,0,1);
\end{tikzpicture}
```

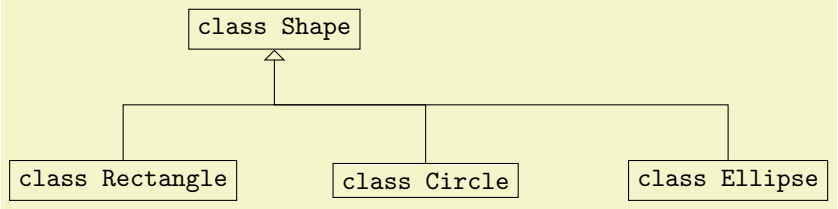
12. NdTdS : J'ai choisi de rendre le singulier anglais de l'auteur par un singulier français. Cet usage est aussi étrange en anglais qu'en français. Mais quitte à être singulier, je choisis cette traduction littérale en avançant, comme justification, l'usage technique du terme pris ici dans une acception précisée dans la définition qui suit.

8.5 Coordonnées de nœuds

Il est assez facile, dans PGF et Ti*k*Z, de définir un nœud auquel on voudrait faire référence plus loin. Une fois le nœud défini, on peut faire référence aux points de ce nœud de diverses façons.

8.5.1 Coordonnées d'ancre nommée

Une *coordonnée d'ancre* est un point d'un nœud précédemment défini avec l'opération de nœud. La syntaxe est `(`⟨*nom du nœud*⟩`.`⟨*ancre*⟩`)` où ⟨*nom du nœud*⟩ est le nom donné précédemment au nœud avec l'option `name=`⟨*nom du nœud*⟩ ou avec la syntaxe spéciale de nom de nœud (voir la section 6.5, p. 45).

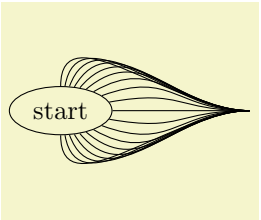



```
\begin{tikzpicture}
  \node (shape)   at (0,2)  [draw] {|class Shape|};
  \node (rect)    at (-2,0) [draw] {|class Rectangle|};
  \node (circle)  at (2,0)  [draw] {|class Circle|};
  \node (ellipse) at (6,0)  [draw] {|class Ellipse|};

  \draw (circle.north) |- (0,1);
  \draw (ellipse.north) |- (0,1);
  \draw[-open triangle 90] (rect.north) |- (0,1) -| (shape.south);
\end{tikzpicture}
```

La section 11.8, p. 95 explique quelles sont les ancres disponibles avec les formes de base.

8.5.2 Coordonnées d'angle d'ancre

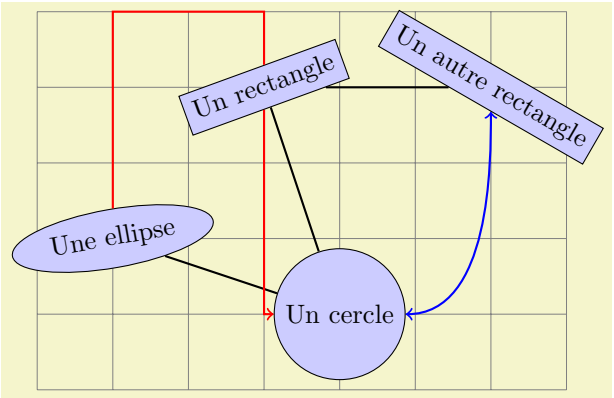
En plus des ancres nommées, on peut utiliser la syntaxe ⟨*nom de nœud*⟩`.`⟨*angle*⟩ pour nommer un point de la frontière du nœud. Ce point est la coordonnée où la demi-droite issue du centre dirigée par l'angle perce la frontière. Voici un exemple :




```
\begin{tikzpicture}
  \node (start) [draw,shape=ellipse] {start};
  \foreach \angle in {-90, -80, ..., 90}
    \draw (start.\angle) .. controls +(\angle:1cm) and +(-1,0) .. (2.5,0);
  \end{tikzpicture}
```

8.5.3 Coordonnées de nœud sans ancre

On peut aussi simplement « omettre » l'ancre et laisser Ti*k*Z calculer une position de la frontière adéquate. Voici un exemple :



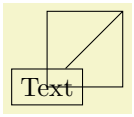
```
\begin{tikzpicture}[fill=blue!20]
  \draw[style=help lines] (-1,-2) grid (6,3);
  \path (0,0)  node(a) [ellipse,rotate=10,draw,fill]    {Une ellipse}
        (3,-1) node(b) [circle,draw,fill]               {Un cercle}
        (2,2)  node(c) [rectangle,rotate=20,draw,fill]  {Un rectangle}
        (5,2)  node(d) [rectangle,rotate=-30,draw,fill] {Un autre rectangle};
  \draw[thick] (a) -- (b) -- (c) -- (d);
  \draw[thick,red,->] (a) |- +(1,3) -| (c) |- (b);
  \draw[thick,blue,<->] (b) .. controls +(right:2cm) and +(down:1cm) .. (d);
\end{tikzpicture}
```

TikZ se montrera raisonnablement intelligent dans la détermination des points de la frontière auxquels on pense mais, naturellement, il peut échouer dans quelques situations. Si TikZ échoue à déterminer un point idoine de la frontière, il utilisera le centre à la place.

Le calcul automatique des ancres ne fonctionne qu'avec l'opération « droite jusqu'à » `--`, les versions verticale/horizontale `|-` et `-|` et avec l'opération « courbe jusqu'à » `..`. Pour toutes les autres commandes de chemin, telle que `parabola` ou `plot`, le centre sera utilisé. Si ce n'est pas ce que l'on désire, on devra utiliser une ancre nommée ou un angle d'ancre.

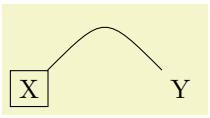
Notez que si l'on utilise des coordonnées automatique pour le début et la fin d'une « droite jusqu'à » comme dans `--(b)--`, alors *deux* coordonnées de frontière seront calculées avec un « déplacer jusqu'à » entre elles. C'est en général exactement ce que l'on veut.

Si l'on utilise des coordonnées relatives en même temps que les coordonnées automatique d'ancre, les coordonnées relatives seront toujours calculées relativement au centre du nœud, et pas relativement au point frontière. En voici un exemple :




```
\tikz \draw (0,0) node(x) [draw] {Text}
           rectangle (1,1)
           (x) -- +(1,1);
```

De même, dans l'exemple qui suit les deux points de contrôle sont $(1,1)$:




```
\tikz \draw (0,0) node(x) [draw] {X}
           (2,0) node(y) {Y}
           (x) .. controls +(1,1) and +(-1,1) .. (y);
```

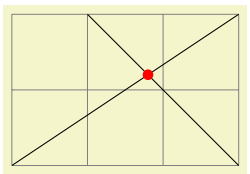
8.6 Coordonnées d'intersection

8.6.1 Intersection de deux droites

On veut souvent définir un point comme l'intersection de deux droites. La première façon de définir une telle intersection est celle-ci : on peut utiliser la syntaxe spéciale

(intersection of ⟨p_1⟩--⟨p_2⟩ and ⟨q_1⟩--⟨q_2⟩).

Cela produira le point d'intersection de la droite passant par p_1 et p_2 et de la droite passant par q_1 et q_2. Si ces deux droites ne se coupent pas où si elles sont identiques une erreur de débordement arithmétique se produira.



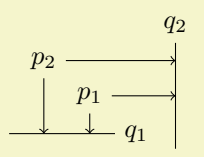
```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw (0,0) coordinate (A) -- (3,2) coordinate (B)
        (1,2)                -- (3,0);

  \fill[red] (intersection of A--B and 1,2--3,0) circle (2pt);
\end{tikzpicture}
```

8.6.2 Intersection d'une droite horizontale et d'une droite verticale

Un cas spécial fréquent d'intersection est l'intersection d'une droite verticale passant par un point p et d'une droite horizontale passant par un autre point q. Il y a, pour cette situation, une syntaxe spéciale, plus courte : on peut écrire soit (⟨p⟩ |- ⟨q⟩) soit (⟨q⟩ -| ⟨p⟩).

Par exemple, `(2,1 |- 3,4)` et `(3,4 -| 2,1)` produisent tout deux la même chose que `(2,4)` (si tant est que le système de coordonnées xy n'ait pas été modifié).

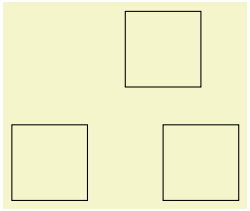
L'application la plus utile de cette syntaxe est le tracer d'un segment jusqu'à un point situé sur une verticale ou une horizontale. Voici un exemple :




```
\begin{tikzpicture}
  \path (30:1cm) node(p1) {$p_1$}   (75:1cm) node(p2) {$p_2$};

  \draw (-0.2,0) -- (1.2,0) node(xline)[right] {$q_1$};
  \draw (2,-0.2) -- (2,1.2) node(yline)[above] {$q_2$};

  \draw[->] (p1) -- (p1 |- xline);
  \draw[->] (p2) -- (p2 |- xline);
  \draw[->] (p1) -- (p1 -| yline);
  \draw[->] (p2) -- (p2 -| yline);
\end{tikzpicture}
```

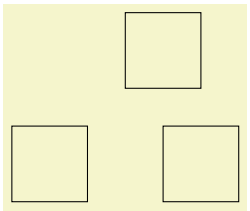
8.7 Coordonnées relatives et incrémentales

On peut préfixer les coordonnées par `++` pour les rendre « relatives ». Une coordonnée telle que `++(1cm,0pt)` signifie « 1 cm vers la droite de la position précédente ». Les coordonnées relatives sont souvent utiles dans des contextes « locaux » :



```
\begin{tikzpicture}
  \draw (0,0)     -- ++(1,0) -- ++(0,1) -- ++(-1,0) -- cycle;
  \draw (2,0)     -- ++(1,0) -- ++(0,1) -- ++(-1,0) -- cycle;
  \draw (1.5,1.5) -- ++(1,0) -- ++(0,1) -- ++(-1,0) -- cycle;
\end{tikzpicture}
```

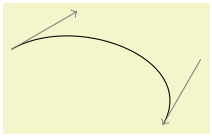
Au lieu de `++` on peut également utiliser un simple `+`. Cela définit aussi une coordonnée relative mais cela « ne met pas à jour » le point courant pour les utilisations suivantes de coordonnées relatives. Ainsi, on peut utiliser cette notation pour définir de nombreux points, tous relatifs au même point « initial ».



```
\begin{tikzpicture}
  \draw (0,0)     -- +(1,0) -- +(1,1) -- +(0,1) -- cycle;
  \draw (2,0)     -- +(1,0) -- +(1,1) -- +(0,1) -- cycle;
  \draw (1.5,1.5) -- +(1,0) -- +(1,1) -- +(0,1) -- cycle;
\end{tikzpicture}
```

Il y a une seule situation spéciale où les coordonnées relatives sont interprétées différemment. Si on utilise des coordonnées relatives comme point de contrôle d'une courbe de Bézier, la règle suivante est en vigueur : premièrement, le premier point de contrôle relatif est considéré comme relatif au début de la courbe ; deuxièmement, le deuxième point de contrôle relatif est considéré comme relatif à la fin de la courbe ; troisièmement, le point final relatif de la courbe est considéré comme relatif au point de début de la courbe.

Ce comportement spécial facilite la définition d'une courbe qui devrait « arriver ou partir suivant une certaine direction » au début ou à la fin. Dans l'exemple suivant, la courbe « quitte » à 30°et « arrive » à 60° :



```
\begin{tikzpicture}
  \draw (1,0) .. controls +(30:1cm) and +(60:1cm) .. (3,-1);
  \draw[gray,->] (1,0) -- +(30:1cm);
  \draw[gray,<-] (3,-1) -- +(60:1cm);
\end{tikzpicture}
```

9 Syntaxe pour la définition de chemin

Un *chemin* est un suite de segments de droites ou de courbes. Sa définition suit la commande `\path` et doit être conforme à une syntaxe spéciale que l'on décrit dans les différentes parties de cette section.

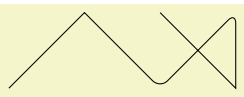
`\path`⟨*définition*⟩ ;

Cette commande n'est disponible que dans un environnement `{tikzpicture}`.

La ⟨*définition*⟩ est un long flux d'*opérations de chemin*. La plupart de ces opérations de chemin précise à Ti*k*Z la façon dont est construit le chemin. Par exemple, lorsque l'on écrit `--(0,0)`, on utilise une *opération ligne-jusqu'à* et cela signifie « continue le chemin d'où que tu sois jusqu'à l'origine ».

Partout où Ti*k*Z attend une opération de chemin, on peut également passer quelques options graphiques, c'est-à-dire une liste d'options écrites entre crochets telle que `[rounded corners]` (*coins arrondis*). Ces options peuvent avoir différents effets :

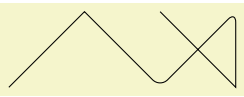
1. Quelques options prennent effet « immédiatement » et s'appliquent à toutes les opérations de chemin suivantes. Par exemple, l'option `rounded corners` arrondira tous les coins suivants mais pas ceux qui précèdent et si l'option `sharp corners` (*coins aigus*)est passée ensuite sur le chemin (dans une nouvelle paire de crochets) l'effet d'arrondi s'arrêtera.



```
\tikz \draw (0,0) -- (1,1)
          [rounded corners] -- (2,0) -- (3,1)
          [sharp corners] -- (3,0) -- (2,1);
```

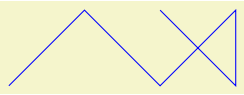
   Un autre exemple est donné par les options de transformation qui s'appliquent aussi aux coordonnées qui suivent.

2. On peut limiter la portée des options qui ont un effet immédiat en isolant une partie du chemin avec des accolades. Par exemple, on aurait pu écrire aussi l'exemple ci-dessus comme suit :



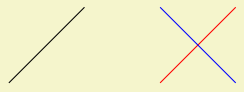
```
\tikz \draw (0,0) -- (1,1)
          {[rounded corners] -- (2,0) -- (3,1)}
          -- (3,0) -- (2,1);
```

3. Certaines options ne s'appliquent qu'à la totalité du chemin. Par exemple, l'option `color=` qui détermine la couleur utilisée pour, disons, dessiner le chemin s'applique toujours à toutes les parties du chemin. Si l'on donne plusieurs couleurs différentes à différentes parties d'un chemin, seule la dernière (de la portée la plus externe) « gagne ».



```
\tikz \draw (0,0) -- (1,1)
          [color=red] -- (2,0) -- (3,1)
          [color=blue] -- (3,0) -- (2,1);
```

   La plupart des options sont de ce type. Dans l'exemple ci-dessus, nous aurions dû « débiter » le chemin avec plusieurs commandes `\path` :



```
\tikz{\draw (0,0) -- (1,1);
     \draw [color=red] (2,0) -- (3,1);
     \draw [color=blue] (3,0) -- (2,1);}
```

Par défaut la commande `\path` ne fait « rien » avec le chemin, elle ne fait que le « jeter ». Ainsi, si l'on écrit `\path(0,0)--(1,1);`, rien n'est dessiné dans la figure. L'unique effet est que la surface occupée par la figure pourra (peut-être) être agrandie afin que le chemin tienne dedans. Pour « faire » réellement quelque chose avec le chemin, on doit donner quelque part sur le chemin une option comme `draw` ou `fill`. Une commande comme `\draw` le fait de manière implicite.

Enfin, on peut donner aussi des *définitions de nœud* sur un chemin. De telles définitions peuvent être placées à différents endroits mais elles sont toujours acceptées quand une opération normale de chemin devrait suivre. Une définition de nœud commence avec `node`. En gros, l'effet en est de typographier le texte du nœud comme du texte normale pour TEX et de le placer sur le chemin à la « position courante ». Les détails sont donnés dans la section 11, p. 86.

Remarquez toutefois que les nœuds *ne font*, en aucun cas, partie du chemin. Bien plutôt, lorsque tout a été fait avec le chemin, c-à-d. tout ce qui a été défini par les options de chemin (comme remplissage

et traçage du chemin du fait d'une option `fill` ou `draw` placée quelque part dans la ⟨*définition*⟩), les nœuds sont ajoutés dans un post-traitement.

Les styles suivants influent sur les portées :

– `style=every path` Ce style est installé au début de chaque chemin. Cela peut être utile pour ajouter (temporairement), par exemple, l'option `draw` à tout ce qui se trouve dans la portée.

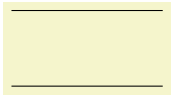
```
\begin{tikzpicture}[fill=examplefill] % ne définit que les couleurs
  \tikzstyle{every path}=[draw]       % tous les chemins sont tracés
  \fill  (0,0) rectangle +(1,1);
  \shade (2,0) rectangle +(1,1);
\end{tikzpicture}
```

9.1 L'opération déplacer-jusqu'à

L'opération peut-être la plus simple est l'opération déplacer-jusqu'à (*move-to*) qui est définie en donnant simplement une coordonnée où l'on attend une opération de chemin.

`\path` ... ⟨*coordonnée*⟩ ... ;

L'opération déplacer-jusqu'à commence normalement un chemin à un certain point. Cela n'entraîne pas la création d'un segment de droite mais définit le point initial du prochain segment. Si un chemin est déjà en construction, c-à-d. si plusieurs segments ont déjà été créés, une opération déplacer-jusqu'à commencera une nouvelle portion de chemin qui ne sera liée à aucun des segments précédents.



```
\begin{tikzpicture}
  \draw (0,0) --(2,0) (0,1) --(2,1);
\end{tikzpicture}
```

Dans la définition `(0,0) --(2,0) (0,1) --(2,1)` on spécifie deux opérations déplacer-jusqu'à : `(0,0)` et `(0,1)`. Les deux autres opérations, à savoir `--(2,0)` et `--(2,1)`, sont des opérations ligne-jusqu'à que l'on décrit ci-après.

9.2 Opération ligne-jusqu'à

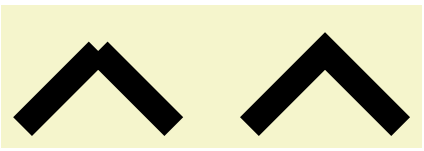
9.2.1 Droites

`\path` ... `--`⟨*coordonnée*⟩ ... ;

L'opération ligne-jusqu'à (*line-to*) étend le chemin courant du point courant en ligne droite jusqu'à la coordonnée donnée. Le « point courant » et le point final de l'opération de dessin précédente ou le point défini par une opération déplacer-jusqu'à préalable.

On utilise deux signe moins suivis par une coordonnée entre parenthèses. On peut placer des espaces avant et après le `--`.

Quand on utilise une opération ligne-jusqu'à et que un segment de chemin vient juste d'être construit, par exemple par une autre opération ligne-jusqu'à, les deux segments sont joints. Cela signifie que, s'ils sont dessinés, le point où ils se rencontrent est « joint » avec souplesse. Pour apprécier la différence, considérons les deux exemples suivants : à gauche le chemin est constitué de deux segments qui ne sont pas joints mais qui se trouvent partager un point, à droite on montre un joint souple.



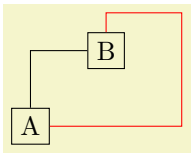
```
\begin{tikzpicture}[line width=10pt]
  \draw (0,0) --(1,1)  (1,1) --(2,0);
  \draw (3,0) -- (4,1) -- (5,0);
  \useasboundingbox (0,1.5); % agrandit la boîte-cadre
\end{tikzpicture}
```

9.2.2 Droites horizontales et verticales

Parfois on veut joindre des points par des segments de droites qui soient uniquement horizontales et verticales. Pour cela, on peut utiliser deux opérations de construction de chemin.

`\path ... -|`⟨*coordonnée*⟩ `... ;`

Cette opération signifie « d'abord horizontal puis vertical ».




```
\begin{tikzpicture}
  \draw (0,0) node(a) [draw] {A}  (1,1) node(b) [draw] {B};
  \draw (a.north) |- (b.west);
  \draw[color=red] (a.east) -| (2,1.5) -| (b.north);
\end{tikzpicture}
```

`\path ... |-`⟨*coordinate*⟩ `... ;`

Cette opération signifie « d'abord vertical puis horizontal ».

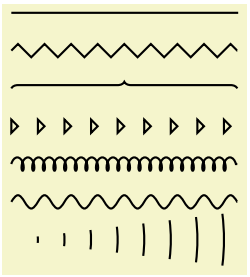
9.2.3 Lignes serpentines

On peut utiliser l'opération ligne-jusqu'à pour ajouter non seulement des lignes droites à un chemin mais encore des ligne « serpentines » (ainsi nommées car elles ressemblent un peu à un serpent vu de dessus).

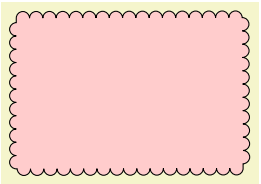
TikZ et PGF se servent d'un concept que j'ai nommé *serpents* pour ajouter de telles lignes « sinueuses ». Un serpent définit une manière d'étendre un chemin entre deux points d'une façon « fantaisiste ».

Normalement un serpent reliera simplement le point initial au point final sans commencer de nouveaux sous-chemins. Ainsi, un chemin contenant une ligne serpentine peut, néanmoins, être encore utilisé pour du remplissage. Toutefois ce n'est pas toujours le cas. Certains serpents sont constitués de nombreux segments non reliés. On ne peut pas utiliser des « lignes » formées de tels serpents comme frontière d'une surface fermée.

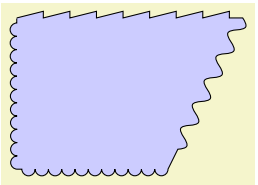
Voici quelques exemples de serpents en action :



```
\begin{tikzpicture}[thick]
  \draw                                        (0,3)   -- (3,3);
  \draw[snake=zigzag]                          (0,2.5) -- (3,2.5);
  \draw[snake=brace]                           (0,2)   -- (3,2);
  \draw[snake=triangles]                       (0,1.5) -- (3,1.5);
  \draw[snake=coil,segment length=4pt]         (0,1)   -- (3,1);
  \draw[snake=coil,segment aspect=0]           (0,.5)  -- (3,.5);
  \draw[snake=expanding waves,segment angle=7] (0,0)   -- (3,0);
\end{tikzpicture}
```



```
\begin{tikzpicture}
  \filldraw[fill=red!20,snake=bumps] (0,0) rectangle (3,2);
\end{tikzpicture}
```



```
\begin{tikzpicture}
  \filldraw[fill=blue!20]             (0,3)
  [snake=saw]                      -- (3,3)
  [snake=coil,segment aspect=0]    -- (2,1)
  [snake=bumps]                    -| (0,3);
\end{tikzpicture}
```

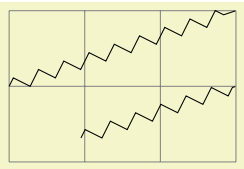
On n'a pas besoin d'opération de chemin spéciale pour utiliser un serpent. À la place on utilisera l'option suivante pour mettre le « serpentage » en route :

– `snake=`⟨*nom de serpent*⟩ Cette option fait que le serpent ⟨*nom de serpent*⟩ sera utilisé dans les opérations ligne-jusqu'à suivantes. Aussi, à chaque fois que l'on utilisera la syntaxe `--` pour spécifier qu'une ligne droite devrait être ajoutée au chemin, c'est un serpent qui sera ajouté à sa place au chemin. Les serpents seront utilisés également quand on se servira des syntaxes `-|` et `|-` et encore lorsque

l'on se servira d'une opération `rectangle`. Les serpents *ne* seront *pas* utilisés quand on utilisera une opération courbe-jusqu'à ni quand on ajoutera une autre ligne courbe au chemin.
Cette option doit être redonnée à chaque chemin. Toutefois on peut délaisser le ⟨*nom de serpent*⟩. Dans ce cas on utilise la portée ⟨*nom de serpent*⟩. Ainsi, on peut définir un nom de serpent « standard » pour la portée et écrire simplement `\draw[snake]` à chaque fois que l'on voudra vraiment utiliser ce serpent.
Le ⟨*nom de serpent*⟩ `none` est spécial. On peut l'utiliser pour arrêter le serpentage après qu'il a été mis en route sur le chemin.
Étrangement Ti*k*Z ne définit aucun ⟨*nom de serpent*⟩ valide par défaut. Pour cela, on doit charger l'extension de bibliothèque `pgflibrarysnakes`. Cette extension définit de nombreux serpents, voir la section **??**, p. **??**, pour une liste complète.

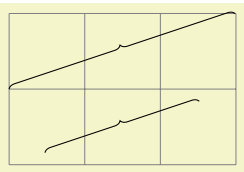
On peut configurer la plupart des serpents. Par exemple, on peut vouloir changer l'amplitude ou la fréquence du serpent qui ressemble à une sinusoïde. Il y a de nombreuses options qui influencent ces paramètres. Toutes les options ne s'appliquent pas à tous les serpents, voir la section **??**, p. **??**, une fois encore pour plus de détails.

- `gap before snakes`=⟨*dimension*⟩ Cette option permet d'ajouter un certain « écart » au serpent à son début. Le serpent ne commencera pas au point courant, au lieu de cela le point initial du serpent sera déplacé de ⟨*dimension*⟩ dans la direction de la cible.



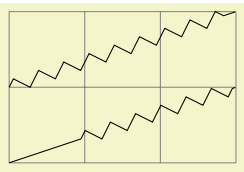
```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw[snake=zigzag]                       (0,1) -- ++(3,1);
  \draw[snake=zigzag,gap before snake=1cm] (0,0) -- ++(3,1);
\end{tikzpicture}
```

- `gap after snake`=⟨*dimension*⟩ Cette option a le même effet que `gap before snake` mais n'affecte que la fin du serpent qui « finira plus tôt ».
- `gap around snake`=⟨*dimension*⟩ Cette option fixe l'écart avant et l'écart après à ⟨*dimension*⟩.



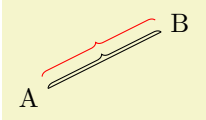
```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw[snake=brace]                       (0,1) -- ++(3,1);
  \draw[snake=brace,gap around snake=5mm] (0,0) -- ++(3,1);
\end{tikzpicture}
```

- `line before snake`=⟨*dimension*⟩ Cette option fonctionne comme `gap before snake` sauf qu'elle reliera le point courant au début du serpent par une ligne droite.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw[snake=zigzag]                       (0,1) -- ++(3,1);
  \draw[snake=zigzag,line before snake=1cm] (0,0) -- ++(3,1);
\end{tikzpicture}
```

- `line after snake`=⟨*dimension*⟩ Fonctionne comme `gap after snake` mais ajoute un segment de droite.
- `line around snake`=⟨*dimension*⟩ Fonctionne comme `gap around snake` mais en ajoutant des segments de droite.
- `raise snake`=⟨*dimension*⟩ On peut utiliser cette option avec tous les serpents. Elle déplace le serpent en le « levant » de ⟨*dimension*⟩. Une ⟨*dimension*⟩ négative baisse le serpent. Les mouvements vers le haut ou le bas sont toujours relatifs à la ligne suivant laquelle le serpent est dessiné. Voici un exemple :



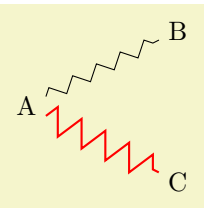

```
\begin{tikzpicture}
  \node (a) {A};
  \node (b) at (2,1) {B};
  \draw                                  (a) -- (b);
  \draw[snake=brace]                     (a) -- (b);
  \draw[snake=brace,raise snake=5pt,red] (a) -- (b);
\end{tikzpicture}
```

- `mirror snake` Cette option fait que le serpent « subi une symétrie d'axe le chemin ». On comprend mieux cela en regardant un exemple :


```
\begin{tikzpicture}
  \node (a) {A};
  \node (b) at (2,1) {B};
  \draw                                    (a) -- (b);
  \draw[snake=brace]                       (a) -- (b);
  \draw[snake=brace,mirror snake,red,thick] (a) -- (b);
\end{tikzpicture}
```

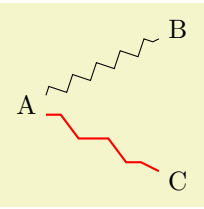
On peut utiliser cette option avec tous les serpents et la combiner avec l'option `raise snake`.

– `segment amplitude`=⟨*dimension*⟩ Cette option fixe « l'amplitude » du serpent. Pour un serpent qui est une sinusoïde se sera l'amplitude de la courbe. Pour d'autres serpents cette valeur décrit en général de combien le serpent « s'élève » ou « s'abaisse » par rapport au chemin. Pour quelques chemins cette valeur est dédaignée.

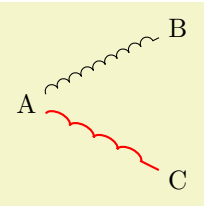



```
\begin{tikzpicture}
  \node (a) {A}   node (b) at (2,1) {B}  node (c) at (2,-1) {C};
  \draw[snake=zigzag]                                (a) -- (b);
  \draw[snake=zigzag,segment amplitude=5pt,red,thick] (a) -- (c);
\end{tikzpicture}
```

– `segment length`=⟨*dimension*⟩ Cette option fixe la longueur d'un « segment » du serpent. Pour une sinusoïde se sera la longueur d'onde, pour d'autres serpents c'est la longueur de chaque « partie répétitive » du serpent.

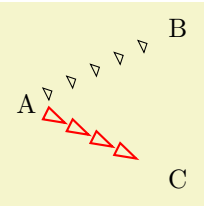



```
\begin{tikzpicture}
  \node (a) {A}   node (b) at (2,1) {B}  node (c) at (2,-1) {C};
  \draw[snake=zigzag]                                (a) -- (b);
  \draw[snake=zigzag,segment length=20pt,red,thick] (a) -- (c);
\end{tikzpicture}
```



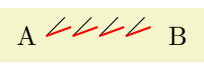

```
\begin{tikzpicture}
  \node (a) {A}   node (b) at (2,1) {B}  node (c) at (2,-1) {C};
  \draw[snake=bumps]                                (a) -- (b);
  \draw[snake=bumps,segment length=20pt,red,thick] (a) -- (c);
\end{tikzpicture}
```

– `segment object length`=⟨*dimension*⟩ Cette option fixe la longueur des objets à l'intérieur de chacun des segments du serpent. Cette option n'est utilisée qu'avec les serpents dans lesquels chaque segment contient un objet tel qu'un triangle ou une étoile.

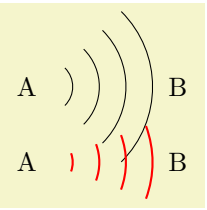



```
\begin{tikzpicture}
  \node (a) {A}   node (b) at (2,1) {B}  node (c) at (2,-1) {C};
  \draw[snake=triangles]                                (a) -- (b);
  \draw[snake=triangles,segment object length=8pt,red,thick] (a) -- (c);
\end{tikzpicture}
```

– `segment angle`=⟨*degrés*⟩ Cette option fixe un angle dont l'interprétation est spécifique aux serpents. Par exemple, les serpents `waves` (*vagues*) et `expanding waves` (*vagues grandissantes*)l'interprètent comme (la moitié) de l'angle d'ouverture de la vague (*wave*). Le serpent `border` utilise cette valeur pour l'angle des petites coches.

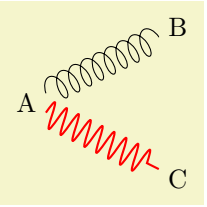



```
\begin{tikzpicture}[segment amplitude=10pt]
  \node (a) {A}   node (b) at (2,0) {B};
  \draw[snake=border]                                (a) -- (b);
  \draw[snake=border,segment angle=20,red,thick] (a) -- (b);
\end{tikzpicture}
```




```
\begin{tikzpicture}[segment amplitude=10pt]
  \node (a)            {A}   node (b)  at (2,0)  {B};
  \node (a') at (0,-1) {A}   node (b') at (2,-1) {B};
  \draw[snake=expanding waves]                          (a)  -- (b);
  \draw[snake=expanding waves,segment angle=20,red,thick] (a') -- (b');
\end{tikzpicture}
```

– `segment aspect=`⟨*rapport*⟩ Cette option fixe le rapport qui est interprété d'une façon spécifique aux serpents. Par exemple, pour le serpent `coils` (littéralement *bobine* mais penser plutôt à un ressort hélicoïdal) cela définit la « direction » depuis laquelle la bobine est vue.




```
\begin{tikzpicture}[segment amplitude=5pt,segment length=5pt]
  \node (a) {A}   node (b) at (2,1) {B}  node (c) at (2,-1) {C};
  \draw[snake=coil]                              (a) -- (b);
  \draw[snake=coil,segment aspect=0,red,thick] (a) -- (c);
\end{tikzpicture}
```

On peut définit de nouveaux serpents mais pas depuis Ti*k*Z. On doit se servir de la commande `\pgfdeclaresnake` directement dans le niveau de base, voir la section **??**, p. **??**.

Les styles suivants définissent des combinaisons de configuration de serpents que l'on peut juger utiles :

– `style=snake triangles 45` Installe un serpent consistant en de petits triangles dont l'angle au sommet est de 45°.
– `style=snake triangles 60` Installe un serpent consistant en de petits triangles dont l'angle au sommet est de 60°.
– `style=snake triangles 90` Installe un serpent consistant en de petits triangles dont l'angle au sommet est de 90°.

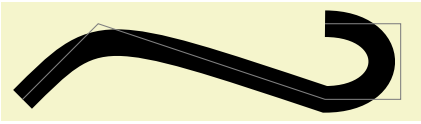
9.3 L'opération courbe-jusqu'à

L'opération courbe-jusqu'à permet d'étendre un chemin avec une courbe de Bézier.

`\path` ... `..contrôles`⟨*c*⟩`and`⟨*d*⟩`..`⟨*y*⟩ ... ;

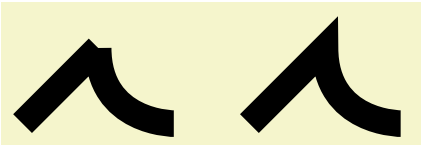
Cette opération étend le chemin courant depuis le point courant, notons le x, à l'aide d'une courbe jusqu'au point y. La courbe est une courbe de Bézier. Pour une telle courbe, y mis à part, il faut préciser également deux points de contrôle c et d. En langage mathématique, la tangente à la courbe en x passe par c. De même, la courbe se termine en y en « venant » de l'autre point de contrôle d. Plus grande est la distance entre x et c et entre d et y, plus grande est la courbe.

Si l'on ne donne pas l'expression « `and`⟨*d*⟩ », on suppose que d est égal à c.



```
\begin{tikzpicture}
  \draw[line width=10pt] (0,0) .. controls (1,1) .. (4,0)
                               .. controls (5,0) and (5,1) .. (4,1);
  \draw[color=gray] (0,0) -- (1,1) -- (4,0) -- (5,0) -- (5,1) -- (4,1);
\end{tikzpicture}
```

Comme avec l'opération ligne-jusqu'à, on n'obtient pas la même chose si les deux courbes sont reliées parce qu'elles ont été crées de deux opérations courbe-jusqu'à ou ligne-jusqu'à consécutives ou s'il se trouvent qu'elles partagent une même extrémité :



```
\begin{tikzpicture}[line width=10pt]
  \draw (0,0) -- (1,1) (1,1) .. controls (1,0) and (2,0) .. (2,0);
  \draw (3,0) -- (4,1) .. controls (4,0) and (5,0) .. (5,0);
  \useasboundingbox (0,1.5); % agrandit la boîte-cadre
\end{tikzpicture}
```

9.4 L'opération cycle

\path ... --cycle ... ;

Cette opération ajoute une ligne droite depuis le point courant jusqu'au dernier point défini pour une opération déplacer-jusqu'à. Notez que cela peut ne pas être le début du chemin. De plus l'opération cycle crée une liaison souple entre le premier segment créé après l'opération déplacer-jusqu'à et une ligne droite est ajoutée.

Considérons l'exemple suivant. À gauche on crée deux triangles avec trois segments de droite mais ils ne sont pas joints aux extrémités. À droite on utilise une opération cycle.

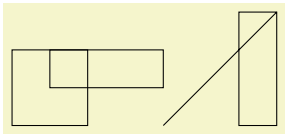
```
\begin{tikzpicture}[line width=10pt]
  \draw (0,0) -- (1,1) -- (1,0) -- (0,0) (2,0) -- (3,1) -- (3,0) -- (2,0);
  \draw (5,0) -- (6,1) -- (6,0) -- cycle (7,0) -- (8,1) -- (8,0) -- cycle;
  \useasboundingbox (0,1.5); % agrandit la boîte-cadre
\end{tikzpicture}
```

9.5 L'opération rectangle

On peut évidemment créer un rectangle avec quatre lignes droites et une opération cycle. Toutefois, comme on a besoin si souvent de rectangles, on dispose pour eux d'une syntaxe spéciale.

\path ... rectangle⟨*coin*⟩ ... ;

Quand on utilise cette opération le point courant donne un coin, l'autre coin est donné par ⟨*coin*⟩ qui devient le nouveau point courant.



```
\begin{tikzpicture}
  \draw (0,0) rectangle (1,1);
  \draw (.5,1) rectangle (2,0.5) (3,0) rectangle (3.5,1.5) -- (2,0);
\end{tikzpicture}
```

9.6 Arrondir les coins

Toutes les opérations de construction de chemin vues jusqu'ici peuvent être influencées par les options suivantes :

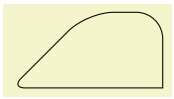
– rounded corners=⟨*insert*⟩ Quand cette option est en vigueur, tous les coins (endroits où une ligne est continuée soit par une opération ligne-jusqu'à ou courbe-jusqu'à) sont remplacés par de petits arcs de telle sorte que les coins soient lisses.

```
\tikz \draw [rounded corners] (0,0) -- (1,1)
          -- (2,0) .. controls (3,1) .. (4,0);
```

L'⟨*insert*⟩ décrit la taille du coin. Notez que ⟨*insert*⟩ n'est *pas* mis à l'échelle lorsque l'on utilise une option de mise à l'échelle comme `scale=2`.

```
\begin{tikzpicture}
  \draw[color=gray,very thin] (10pt,15pt) circle (10pt);
  \draw[rounded corners=10pt] (0,0) -- (0pt,25pt) -- (40pt,25pt);
\end{tikzpicture}
```

On peut arrondir les coins ou cesser de les arrondir « au cours du chemin » et des coins différents du même chemin peuvent avoir des rayons différents :



```
\begin{tikzpicture}
  \draw (0,0) [rounded corners=10pt] -- (1,1) -- (2,1)
                      [sharp corners] -- (2,0)
                  [rounded corners=5pt] -- cycle;
\end{tikzpicture}
```

Voici un rectangle aux coins arrondis :

```
\tikz \draw[rounded corners=1ex] (0,0) rectangle (20pt,2ex);
```

On prendra garde que cette option présente plusieurs écueils. En premier lieu, le coin arrondi ne sera jamais un arc (de cercle) que si l'angle vaut 90°. Dans les autres cas, le coin arrondi sera toujours arrondi mais « pas aussi beau ».
Deuxièmement, si le chemin contient de très petits segments, l'arrondi peut entraîner des effets inattendus. Dans de tels cas il peut être nécessaire de faire cesser l'arrondissement avec `sharp corners` (*coins aigus*).

– `sharp corners` Cette option arrête tout arrondi des coins suivants du chemin.

9.7 Les opérations circle et ellipse

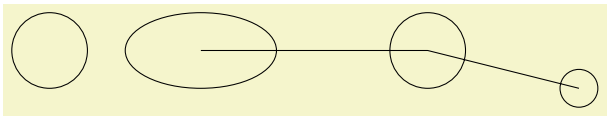
On peut bien approcher un cercle avec quatre courbes de Bézier. Toutefois le faire correctement est difficile. Pour cette raison on dispose d'une syntaxe spéciale pour ajouter une telle approximation d'un cercle au chemin courant.

`\path ... circle(⟨rayon⟩) ... ;`

Le centre du cercle est donné par le point courant. Le nouveau point courant du chemin restera le centre du cercle.

`\path ... ellipse(⟨demi-largeur⟩ and ⟨demi-hauteur⟩) ... ;`

Notez que l'on peut ajouter des espaces après `ellipse` mais que l'on *doit* en mettre autour de `and`.



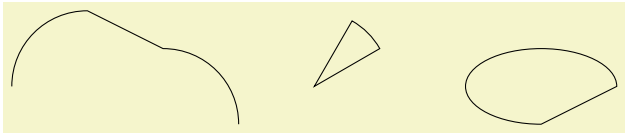
```
\begin{tikzpicture}
  \draw (1,0) circle (.5cm);
  \draw (3,0) ellipse (1cm and .5cm) -- ++(3,0) circle (.5cm)
    -- ++(2,-.5) circle (.25cm);
\end{tikzpicture}
```

9.8 L'opération arc

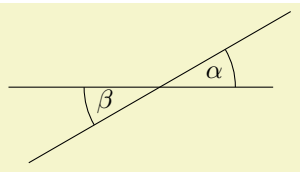
L'opération *arc* permet d'ajouter un arc au chemin courant.

`\path ... arc(⟨angle de départ⟩ :⟨angle de fin⟩ :⟨rayon⟩/⟨demi-hauteur⟩) ... ;`

L'opération arc ajoute une arc du cercle de rayon donné entre les deux angles. Cet arc commencera au point courant et finira avec la fin de l'arc.



```
\begin{tikzpicture}
  \draw (0,0) arc (180:90:1cm) -- (2,.5) arc (90:0:1cm);
  \draw (4,0) -- +(30:1cm) arc (30:60:1cm) -- cycle;
  \draw (8,0) arc (0:270:1cm/.5cm) -- cycle;
\end{tikzpicture}
```



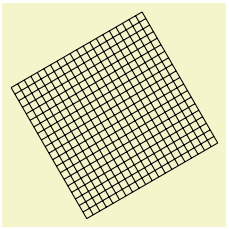

```
\begin{tikzpicture}
  \draw (-1,0) -- +(3.5,0);
  \draw (1,0) ++(210:2cm) -- +(30:4cm);
  \draw (1,0) +(0:1cm) arc (0:30:1cm);
  \draw (1,0) +(180:1cm) arc (180:210:1cm);
  \path (1,0) ++(15:.75cm) node{$\alpha$};
  \path (1,0) ++(15:-.75cm) node{$\beta$};
\end{tikzpicture}
```

9.9 L'opération grid

On peut ajouter un quadrillage au chemin courant avec l'opération de chemin `grid` (*quadrillage*).

`\path` ... `grid[`⟨*options*⟩`]`⟨*coin*⟩ ... ;

Cette opération ajoute un quadrillage remplissant un rectangle dont les coins sont donnés par ⟨*coin*⟩ et la coordonnée précédente. Ainsi, la manière courante de dessiner un quadrillage est `\draw (1,1) grid (3,3);`, qui produit un quadrillage remplissant le rectangle dont les coins sont $(1,1)$ et $(3,3)$. Toutes les transformations de coordonnées s'appliquent au quadrillage.



```
\tikz[rotate=30] \draw[step=1mm] (0,0) grid (2,2);
```

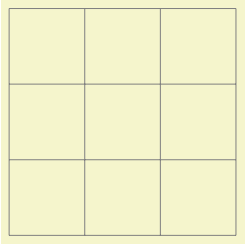
Le pas du quadrillage est soumis aux options suivantes :

- `step=`⟨*dimension*⟩ fixe le pas à la fois dans les directions x et y.
- `xstep=`⟨*dimension*⟩ fixe le pas dans la direction x.
- `ystep=`⟨*dimension*⟩ fixe le pas dans la direction y.

Il est important de noter que le quadrillage est toujours « réglé » de telle sorte qu'il contienne le point $(0,0)$ si ce point se trouve être à l'intérieur du rectangle. Ainsi, le quadrillage n'a *pas* toujours une intersection aux points des coins ; cela n'arrive que si les coins sont des multiples du pas. Notez que du fait des erreurs d'arrondi les « dernières » lignes du quadrillage peuvent manquer. Dans ce cas, on devra ajouter un *epsilon* aux points de coin. [13]

Les styles suivants sont utiles pour le dessin de quadrillages :

- `style=help lines` Cela atténue les lignes en les dessinant fines et grises. Toutefois ce style n'est pas disponible automatiquement et on devra écrire par exemple :



```
\tikz \draw[style=help lines] (0,0) grid (3,3);
```

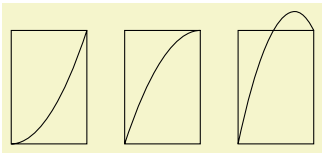
9.10 L'opération parabole

L'opération de chemin `parabola` (*parabole*) poursuit le chemin courant avec une parabole. Une parabole est une courbe (à homothétie et translation près) définie par l'équation $f(x) = x^2$ et ressemble à ceci : ∪.

`\path` ... `parabola[`⟨*options*⟩`]bend`⟨*coordonnée du sommet*⟩⟨*coordonnée*⟩ ... ;

Cette opération ajoute une parabole passant pour le point courant et la ⟨*coordonnée*⟩ donnée. Si l'option `bend` (*sommet*) est donnée, elle précise au devrait être le sommet ; on peut également utiliser ⟨*options*⟩ pour préciser au se trouve le sommet. Par défaut le sommet est à l'ancien point courant.

13. NdTdS : On rajoutera « les coordonnées de » *ad libitum*.

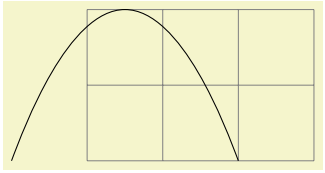


```
\begin{tikzpicture}
  \draw                (0,0) rectangle                (1,1.5)
                       (0,0) parabola                 (1,1.5);
  \draw[xshift=1.5cm] (0,0) rectangle                 (1,1.5)
                       (0,0) parabola[bend at end]    (1,1.5);
  \draw[xshift=3cm]   (0,0) rectangle                 (1,1.5)
                       (0,0) parabola bend (.75,1.75) (1,1.5);
\end{tikzpicture}
```

Les options suivantes influencent les paraboles :

– `bend`=⟨*coordoneée*⟩ A le même effet qu'écrire `bend`⟨*coordonnée*⟩ en dehors des ⟨*options*⟩. L'option précise que le sommet de la parabole devrait être à la ⟨*coordonnée*⟩ donnée. On doit soi-même prendre soin à ce que le sommet ait une position « valide » c'est-à-dire que, si aucune parabole d'équation $y = ax^2 + bx + c$ passant par le point courant ancien, le sommet donné et le nouveau point courant, le résultat ne sera pas une parabole.
⟨*coordonnée*⟩ possède une propriété spéciale : quand on donne une coordonnée relative comme `+(0,0)`, la position à quoi cette coordonnée est relative est « flexible ». Plus précisément, cette position est quelque part sur la droite joignant l'ancien point courant et le nouveau. Sa position exacte dépend de l'option suivante.

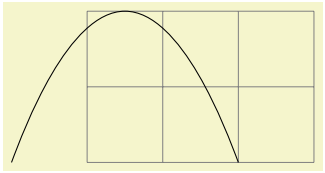
– `bend pos`=⟨*fraction*⟩ Précise où est le point « précédent » relativement auquel la position du sommet est calculée. Le point précédent sera à la ⟨*fraction*⟩ partie du segment joignant l'ancien point courant et le nouveau.
L'idée est la suivante : si l'on écrit `bend pos=0` et `bend +(0,0)`, le sommet sera à l'ancien point courant. Si l'on écrit `bend pos=1` et `bend +(0,0)`, le sommet sera au nouveau point courant. Si l'on écrit `bend pos=0.5` et `bend +(0,2cm)`, le sommet sera 2 cm au-dessus du milieu du segment joignant le point de départ et le point final. C'est surtout utile dans des situations comme celle qui suit :



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw (-1,0) parabola[bend pos=0.5] bend +(0,2) +(3,0);
\end{tikzpicture}
```

Dans l'exemple ci-dessus, `bend +(0,2)` signifie essentiellement « une parabole qui a 2 cm de hauteur » et `+(3,0)` signifie « et 3 cm de largeur ». Comme de telles situations se présentent souvent, il y a un raccourci spécial :

– `parabola height`=⟨*dimension*⟩ Cette option a le même effet que si l'on écrivait `[bend pos=0.5,bend={+(0pt,`⟨*dimen*
à la place.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw (-1,0) parabola[parabola height=2cm] +(3,0);
\end{tikzpicture}
```

Les styles suivants sont d'utiles raccourcis :

– `style=bend at start` Cela place le sommet au début de la parabole. C'est un raccourci pour les options suivantes : `bend pos=0,bend={+(0,0)}`.

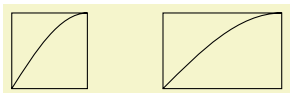
– `style=bend at end` Cela place le sommet à la fin de la parabole.

9.11 Les opérations sine et cosine

Les opérations `sin` et `cos` sont semblables à l'opération `parabola`. Elles aussi peut être utilisées pour dessiner des (parties de) sinusoïdes.

`\path` ... `sin`⟨*coordonée*⟩ ... ;

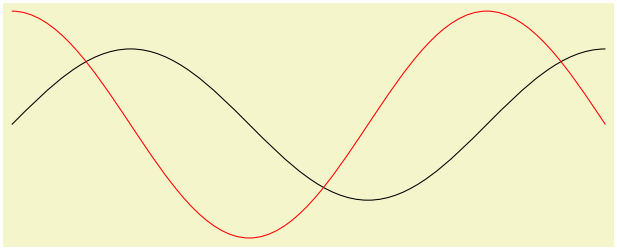
`sin` dessine une sinusoïde d'équation $y = \sin x$ avec x dans l'intervalle $[0, \pi/2]$. Cette courbe subie une homothétie et une translation pour que le point de départ de la courbe soit l'ancien point courant et que le point final de la courbe soit ⟨*coordonnée*⟩. Voici un exemple qui devrait éclaircir cela :



```
\tikz \draw (0,0) rectangle (1,1)     (0,0) sin (1,1)
            (2,0) rectangle +(1.57,1) (2,0) sin +(1.57,1);
```

`\path` ... `cos`⟨*coordonnée*⟩ ... ;

Cette opération fonctionne de manière similaire sauf que c'est la courbe d'équation $y = \cos x$ qui est dessinée pour x dans $[0, \pi/2]$. En alternant correctement `sin` et `cos` on peut créer une sinusoïde complète :



```
\begin{tikzpicture}[xscale=1.57]
  \draw (0,0) sin (1,1) cos (2,0) sin (3,-1) cos (4,0) sin (5,1);
  \draw[color=red] (0,1.5) cos (1,0) sin (2,-1.5) cos (3,0) sin (4,1.5) cos (5,0);
\end{tikzpicture}
```

On notera qu'il n'y a pas de moyen de dessiner (commodément) une partie d'une sinusoïde dont le point final n'est pas un multiple de $\pi/2$.

9.12 L'opération plot

On peut utiliser l'opération `plot` pour ajouter au chemin une courbe qui passe par un grand nombre de coordonnées. Ces coordonnées sont soit données comme une simple liste de coordonnées soit lues dans fichier.

Il y a différentes versions de la syntaxe de `plot` [14].

`\path` ... `--plot`⟨*autres arguments*⟩ ... ;

Cette opération trace la courbe qui passe par les coordonnées spécifiées dans les ⟨*autres arguments*⟩. Le (sous-)chemin courant est simplement poursuivi, c-à-d. qu'une opération ligne-jusqu'à jusqu'au premier point de la courbe est ajoutée implicitement. On explique les détails des ⟨*autres arguments*⟩ dans quelques instants.

`\path` ... `plot`⟨*autres arguments*⟩ ... ;

Cette opération trace la courbe qui passe par les coordonnées spécifiées dans les ⟨*autres arguments*⟩ en commençant par « déplacer » jusqu'à la première coordonnée de la courbe.

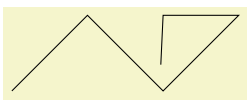
On utilise les ⟨*autres arguments*⟩ de trois façons différentes pour spécifier les coordonnées des points par lesquels doit passer la courbe.

1. `--plot[`⟨*options locales*⟩`]coordinates{`⟨*coordonnée 1*⟩⟨*coordonnée 2*⟩... ⟨*coordonnée n*⟩`}`
2. `--plot[`⟨*options locales*⟩`]file{`⟨*nom de fichier*⟩`}`
3. `--plot[`⟨*options locales*⟩`]function{`⟨*formule pour gnuplot*⟩`}`

On explique ces trois façons différentes dans ce qui suit.

9.12.1 Interpolation à partir de points donnés en ligne

Dans les deux premiers cas, les points sont donnés directement dans le fichier TEX comme dans l'exemple suivant :



```
\tikz \draw plot coordinates {(0,0) (1,1) (2,0) (3,1) (2,1) (10:2cm)};
```

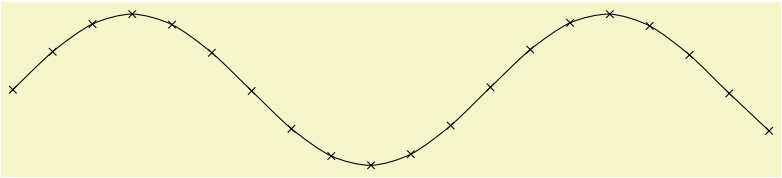
Voici un exemple montrant la différence entre `plot` et `--plot` :

14. NdTdS : Il s'agit ici d'interpolation.

```
\begin{tikzpicture}
  \draw (0,0) -- (1,1) plot coordinates {(2,0)  (4,0)};
  \draw[color=red,xshift=5cm]
        (0,0) -- (1,1) -- plot coordinates {(2,0)  (4,0)};
\end{tikzpicture}
```

9.12.2 Interpolation à partir de points lus dans un fichier externe

La deuxième façon de spécifier des points est de les placer dans un fichier externe nommé ⟨*nom de fichier*⟩. À ce jour, le seul format admis par Ti*k*Z est le suivant : chaque ligne de ⟨*nom de fichier*⟩ doit commencer avec deux nombres séparés par un espace. Tout ce qui suit les deux nombres sur la ligne est ignoré. De plus, les lignes commençant par `%` ou `#` sont ignorées tout comme les lignes vides. (C'est exactement le format produit par GNUPLOT lorsque l'on a donné la commande `set terminal table`.) Si nécessaire d'autres formats seront gérés à l'avenir mais il est généralement facile de produire un fichier contenant les données sous cette forme.



```
\tikz \draw plot[mark=x,smooth] file {plots/pgfmanual-sine.table};
```

Voici le fichier `plots/pgfmanual-sine.table` :

```
#Curve 0, 20 points
#x y type
0.00000 0.00000  i
0.52632 0.50235  i
1.05263 0.86873  i
1.57895 0.99997  i
...
9.47368 -0.04889  i
10.00000 -0.54402  i
```

On l'a créé, à l'aide de `gnuplot` à partir du source suivant :

```
set terminal table
set output "../plots/pgfmanual-sine.table"
set format "%.5f"
set samples 20
plot [x=0:10] sin(x)
```

Les ⟨*options locales*⟩ de l'opération `plot` sont locales à chaque interpolation et n'affectent pas les autres « sur le même chemin ». Par exemple, `plot[yshift=1cm]` soulèvera localement la courbe de 1 cm. Souvenez-vous, toutefois, que la plupart des options ne s'appliquent qu'à la totalité du chemin. Par exemple, `plot[red]` n'a pas pour effet de colorier la courbe en rouge. Après tout, vous essayez de colorier « localement » le chemin en rouge, ce qui n'est pas possible.

9.12.3 Tabuler et tracer une fonction

Souvent on voudra tracer le graphe d'une fonction comme $f(x) = x \sin x$. Malheureusement, TEX ne possède pas vraiment la puissance de calcul nécessaire pour calculer les coordonnées des points d'une telle courbe de manière efficace (après tout c'est un logiciel de typographie). Toutefois, si on l'y autorise, TEX peut essayer d'appeler un programme externe qui peut produire facilement ces coordonnées. À ce jour Ti*k*Z sait comment appeler GNUPLOT.

La première fois que Ti*k*Z rencontrera l'opération `plot[id=`⟨*id*⟩`] function{x*sin(x)}` il créera un fichier appelé ⟨*préfixe*⟩⟨*id*⟩`.gnuplot` où ⟨*préfixe*⟩ est `\jobname.` par défaut, c-à-d. le nom du fichier `.tex` principal. Si on ne donne aucun ⟨*id*⟩, il restera vide, ce qui est correct, mais il vaut mieux que chaque `plot` ait un ⟨*id*⟩ unique pour des raisons que l'on donnera dans un moment. Ensuite, Ti*k*Z écrit du code d'initialisation dans ce fichier qu'il fait suivre de `plot x*sin(x)`. Le code d'initialisation définit ce qu'il faut pour que l'opération `plot` écrive les coordonnées dans un autre fichier appelé ⟨*préfixe*⟩⟨*id*⟩`.table`. Enfin, ce fichier `.table` est lu comme si l'on avait donné l'instruction `plot file{`⟨*préfixe*⟩⟨*id*⟩`.table}`.

Pour que le mécanisme de tabulation fonctionne, il faut deux conditions :

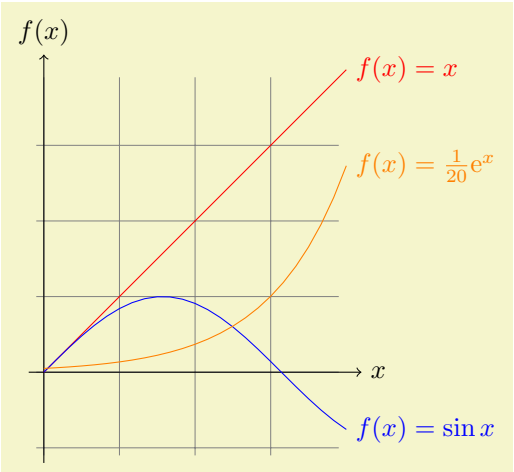
1. On doit autoriser TEX à appeler un programme externe. Cette possibilité est souvent désactivée par défaut car il y a des risques pour la sécurité (on pourrait, sans le savoir, TEXer un fichier qui appellerait toutes sortes de « mauvaises » commandes.) Pour activer cet « appel aux programmes externes » on doit passer une option de ligne de commande à TEX le programme. Habituellement, cette option est appelée `shell-escape`, `enable-write18` ou quelque chose comme ça. Par exemple, je peux donner l'option `--shell-escape` à mon `pdflatex`.
2. On doit avoir installé le logiciel `gnuplot` et TEX doit le trouver en compilant un fichier.

Malheureusement ces conditions ne sont pas toujours remplies. En particulier si l'on fournit des sources à un coauteur et que celui-ci n'a pas installé GNUPLOT, il aura du mal à compiler les fichiers.

C'est pour cette raison que Ti*k*Z se comporte différemment lorsque l'on compile le graphique pour la deuxième fois : si, en arrivant à `plot[id=`⟨*id*⟩`] function{...}`, Ti*k*Z voit que le fichier ⟨*préfixe*⟩⟨*id*⟩`.table` existe déjà *et* s'il contient ce que Ti*k*Z pense qu'il « devrait » contenir, alors le fichier `.table` est lu immédiatement sans essayer de faire appel à `gnuplot`. Cette approche offre les avantages suivants :

1. Si l'on passe un lot de ses fichiers `.tex` et tout les fichiers `.gnuplot` et `.table` à quelqu'un d'autre, il pourra les TEXer sans avoir à installer `gnuplot` ;
2. Si `\write18` est désactivé par raison de sécurité (ce qui est une bonne idée) alors, à la première compilation du fichier `.tex`, le fichier `.gnuplot` sera crée mais pas le fichier `.table`. On pourra simplement appeler `gnuplot` « à la main » pour chacun des fichiers `.gnuplot` ce qui produira tous les fichiers `.tables` requis ;
3. Si l'on change la fonction que l'on voudrait tracer ou bien son domaine, Ti*k*Z essaiera automatiquement de recréer le fichier `.table` ;
4. Si, par paresse, on ne fournit pas de `id`, le même fichier `.gnuplot` sera utilisé pour les différentes `plot` mais cela ne pose pas de problème puisque chaque fichier `.table` est automatiquement recréé à la volée pour chaque `plot`. *N. B. : Si l'on a l'intention de partager ses fichiers, on utilisera toujours un identificateur* `id` *afin que les fichiers puissent être traités sans que* GNUPLOT *soit installé.* Par ailleurs, un identifiant unique pour chaque courbe améliore la vitesse de compilation puisque aucun programme externe n'est appelé à moins que ce ne soit vraiment nécessaire.

Quand on utilise `plot function{`⟨*formule pour gnuplot*⟩`}`, la ⟨*formule pour gnuplot*⟩ doit respecter la syntaxe de `gnuplot` dont les détails sont hors de la portée de ce manuel. Voici l'essentiel ultracondensé : utiliser `x` comme variable et la syntaxe de C pour les courbes normales [15] et utiliser `t` comme variable pour les courbes paramétriques. Voici quelques exemples :




```
\begin{tikzpicture}[domain=0:4]
  \draw[very thin,color=gray] (-0.1,-1.1) grid (3.9,3.9);

  \draw[->] (-0.2,0) -- (4.2,0) node[right] {$x$};
  \draw[->] (0,-1.2) -- (0,4.2) node[above] {$f(x)$};

  \draw[color=red]    plot[id=x]   function{x}           node[right] {$f(x) =x$};
  \draw[color=blue]   plot[id=sin] function{sin(x)}      node[right] {$f(x) = \sin x$};
  \draw[color=orange] plot[id=exp] function{0.05*exp(x)} node[right] {$f(x) = \frac{1}{20} \mathrm e^x$};
\end{tikzpicture}
```

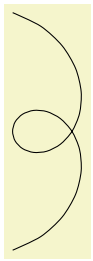
15. NdTdS : Les courbes représentatives d'une fonction d'une variable.

Les options suivantes agissent sur `plot` :

– `samples`=⟨*nombre*⟩ définit le nombre d'échantillons utilisés pour l'interpolation. La valeur par défaut est 25.

– `domain`=⟨*début*⟩`:`⟨*fin*⟩ définit le domaine dans lequel on prend les échantillons. La valeur par défaut est `-5:5`.

– `parametric`=⟨*true ou false*⟩ précise si la courbe est paramétrique. Si la valeur est *true* alors on doit utiliser `t` au lieu de `x` comme paramètre et on doit donner deux fonctions séparées par une virgule dans la ⟨*formule pour gnuplot*⟩. En voici un exemple :



```
\tikz \draw[scale=0.5,domain=-3.141:3.141,smooth]
  plot[parametric,id=parametric-example] function{t*sin(t),t*cos(t)};
```

– `id`=⟨*id*⟩ définit l'identifiant de la courbe courante. Cet identifiant devrait être unique pour chaque `plot` (bien que les choses marcheront aussi sinon mais pas aussi bien, voir les explications ci-dessus). Le ⟨*id*⟩ fera partie d'un nom de fichier, il ne devrait donc pas contenir de chose bizarre comme `*` ou `$`.

– `prefix`=⟨*préfixe*⟩ sera placé devant chaque nom de fichier `plot`. Par défaut la valeur est `\jobname.` mais si l'on a de nombreuses courbes il vaudrait peut-être mieux utiliser par exemple `plots/` pour que tous les fichiers de `plot` soient placés dans un répertoire. On devra créer soi-même le répertoire au préalable.

– `raw gnuplot` fait que la ⟨*formule pour gnuplot*⟩ est passée à GNUPLOT sans que ni l'échantillon ni l'opération `plot` soient initialisés. On pourrait ainsi écrire :

```
plot[raw gnuplot,id=raw-example] function{set samples 25; plot sin(x)}
```

Ce peut être utile pour passer des choses compliquées à GNUPLOT. Toutefois, pour des situations vraiment difficiles, on devrait créer un fichier GNUPLOT externe spécial et utiliser la syntaxe de `fichier` pour inclure la table « à la main ».

Les styles suivants influent sur `plot` :

– `style=every plot` Ce style est installé dans chaque `plot` c-à-d. comme si l'on écrivait :

```
plot[style=every plot,...]
```

C'est particulièrement utile pour définir globalement un préfixe pour tous les `plot`s avec :

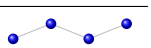
```
\tikzstyle{every plot}=[prefix=plots/]
```

9.12.4 Placer des marques sur une courbe

Comme on l'a déjà vu, on peut ajouter des *marques* sur une courbe avec l'option `mark`. Quand on utilise cette option, une copie de la marque est placée à chaque point tabulé. On notera que les marques sont placées *après* que tout le chemin a été dessiné/rempli/estompé. À cet égard elles sont traitées comme le texte des nœuds.

Dans le détail, les options suivantes précisent la façon dont les marques sont tracées :

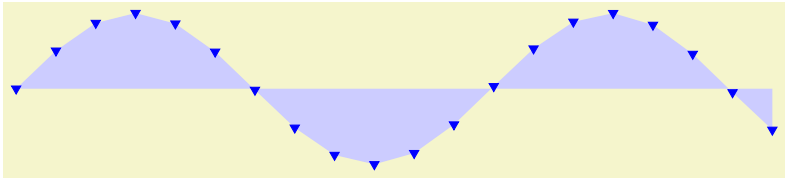
– `mark`=⟨*symbole de marque*⟩ Met en vigueur le symbole de marque précédemment défini avec `\pgfdeclareplotmark`. Par défaut, on dispose de `*`, `+` et `x` qui sont rendus par un disque, un signe plus et une croix. Bien d'autres marques sont disponibles lorsque on a chargé la bibliothèque `pgflibraryplotmarks`. La section **??**, p. **??** donne la liste des marques disponibles.
Une marque est spéciale : la marque `ball` (*ballon*) n'est disponible que dans Ti*k*Z. L'option `ball color` détermine la couleur des ballons. Il ne faut pas utiliser cette option lorsque l'on a besoin d'un grand nombre de marques puiqu'il faudra beaucoup de temps pour en faire le rendu en PostScript.

Option	Effet
`mark=ball`	

– `mark size`=⟨*dimension*⟩ Définit la taille des marques. Pour les marques circulaires, la ⟨*dimension*⟩ est le rayon, pour les autres la ⟨*dimension*⟩ devrait être environ la moitié de la hauteur et de la largeur.

Cette option n'est pas vraiment nécessaire puisque l'on peut obtenir le même effet à l'aide de l'option locale `scale=`⟨*facteur*⟩ où ⟨*facteur*⟩ est le quotient de la taille désirée par la taille par défaut. Toutefois, utiliser `mark size` est un peu plus rapide et plus naturel.

– `mark options=`⟨*options*⟩ Ces options sont appliquées aux marques quand elles sont tracées. Par exemple, on peut mettre à l'échelle (ou transformer autrement) la marque ou en définir la couleur.



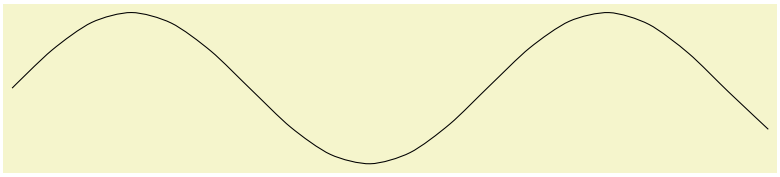
```
\tikz \fill[fill=blue!20]
  plot[mark=triangle*,mark options={color=blue,rotate=180}]
    file{plots/pgfmanual-sine.table} |- (0,0);
```

9.12.5 Aspects des courbes

L'opération `plot` peut faire différentes choses avec les points qu'elle lit depuis un fichier ou dans une liste en ligne. Par défaut, les points seront joints par des segments de droite. Toutefois on peut utiliser des options pour changer le comportement de `plot`.

– `sharp plot` C'est l'option par défaut. Les points sont reliés par des segments de droite. Cette option n'est fournie que pour permettre de la réactiver si l'on a, par exemple, passé l'option `smooth`.

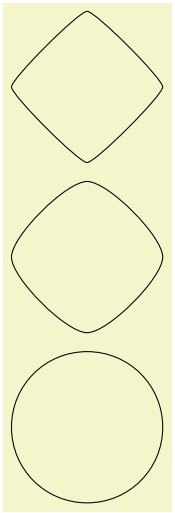
– `smooth` Cette option fait que les points sont reliés par une courbe lisse :



```
\tikz\draw plot[smooth] file{plots/pgfmanual-sine.table};
```

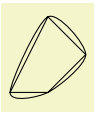
Notez que l'algorithme de lissage n'est pas très intelligent. On obtiendra les meilleurs résultats si les angles de courbure sont petits, c-à-d. moindre que 30° environ et, de manière encore plus importante, si les distances entre les points sont identiques sur tout le chemin où a lieu l'interpolation.

– `tension=`⟨*valeur*⟩ Cette option influence le degré de « tension » du lissage. Une valeur basse donne des coins plus aigus, une valeur élevée les rend plus « ronds ». Une valeur de 1 donne un cercle si l'on donne quatre points partageant un cercle en quatre. La valeur par défaut est 0,55. La valeur « correcte » dépend des détails de l'interpolation.



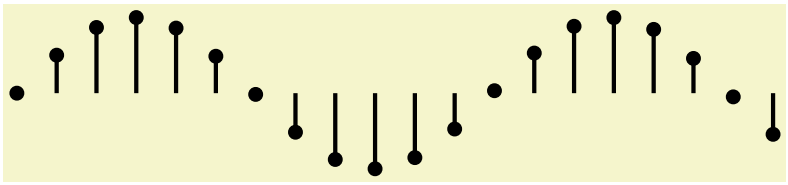
```
\begin{tikzpicture}[smooth cycle]
  \draw                  plot[tension=0.2]
    coordinates{(0,0) (1,1) (2,0) (1,-1)};
  \draw[yshift=-2.25cm] plot[tension=0.5]
    coordinates{(0,0) (1,1) (2,0) (1,-1)};
  \draw[yshift=-4.5cm]  plot[tension=1]
    coordinates{(0,0) (1,1) (2,0) (1,-1)};
\end{tikzpicture}
```

– `smooth cycle` Cette option fait que les points du chemin sont reliés par une courbe lisse fermée.

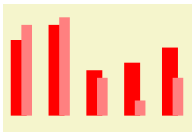


```
\tikz[scale=0.5]
  \draw plot[smooth cycle] coordinates{(0,0) (1,0) (2,1) (1,2)}
        plot               coordinates{(0,0) (1,0) (2,1) (1,2)} -- cycle;
```

– ycomb Cette option fait que l'opération `plot` interprète les points donnés différemment. Au lieu de les relier, elle ajoute au chemin un segment de droite pour chaque point donné, segment joignant verticalement le point à l'axe des x, produisant une sorte de « peigne » (*comb*) ou de « diagramme à segments ».

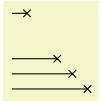


```
\tikz\draw[ultra thick] plot[ycomb,thin,mark=*] file{plots/pgfmanual-sine.table};
```



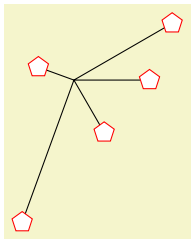
```
\begin{tikzpicture}[ycomb]
  \draw[color=red,line width=6pt]
    plot coordinates{(0,1) (.5,1.2) (1,.6) (1.5,.7) (2,.9)};
  \draw[color=red!50,line width=4pt,xshift=3pt]
    plot coordinates{(0,1.2) (.5,1.3) (1,.5) (1.5,.2) (2,.5)};
\end{tikzpicture}
```

– xcomb Cette option fonctionne comme `ycomb` sauf que les segments sont horizontaux.



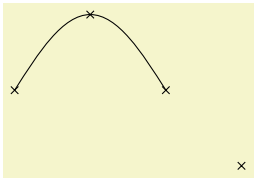
```
\tikz \draw plot[xcomb,mark=x] coordinates{(1,0) (0.8,0.2) (0.6,0.4) (0.2,1)};
```

– polar comb Cette option fait que, pour chaque point donné, un segment joignant le point et l'origine est ajouté au chemin.



```
\tikz \draw plot[polar comb,
     mark=pentagon*,mark options={fill=white,draw=red},mark size=4pt]
   coordinates {(0:1cm) (30:1.5cm) (160:.5cm) (250:2cm) (-60:.8cm)};
```

– only marks Cette option fait que seules les marques sont montrées ; aucun segment de chemin n'est ajouté au chemin réel. On peut s'en servir pour ajouter rapidement quelques marques sur un chemin.



```
\tikz \draw (0,0) sin (1,1) cos (2,0)
  plot[only marks,mark=x] coordinates{(0,0) (1,1) (2,0) (3,-1)};
```

9.13 L'opération de restriction de portée

Quand Ti*k*Z rencontre une accolade ouvrante ou fermante (`{` ou `}`) là où on attendrait une opération de chemin, il ouvrira ou fermera une portée. Toutes les options que l'on peut « localiser » verront leur action restreinte à la portée ainsi définie. Par exemple, si l'on applique une transformation comme `[xshift=1cm]` à l'intérieur d'une portée, le déplacement ne s'appliquera qu'à la portée. D'un autre côté, une option comme `color=red` n'a pas d'effet dans une portée puisqu'elle ne peut s'appliquer qu'à la totalité du chemin.

9.14 L'opération Node

On peut ajouter des nœuds à un chemin avec l'opération `node`. Puisque cette option est assez compliquée et puisque les nœuds ne font pas vraiment partie du chemin lui-même, il y a une section séparée traitant des nœuds, voir la section 11, p. 86.

10 Actions sur les chemins

Une fois un chemin construit, on peut faire différentes choses avec lui. Il peut être dessiné (ou tracé) avec un « crayon », il peut être rempli avec une couleur ou un dégradé, il peut être utilisé pour découper un dessin postérieur, il peut être utilisé pour définir l'extension de la figure — ou toute combinaison de ces actions en même temps.

Pour décider ce que l'on doit faire d'un chemin, on peut utiliser deux méthodes. D'abord, on peut utiliser une commande à utilisation spéciale comme `\draw` pour indiquer que le chemin devrait être dessiné. Toutefois, les commandes comme `\draw` et `\fill` ne sont que des abréviations pour des cas spéciaux de méthode plus générale : ici la commande `\path` est utilisée pour définir le chemin. Ensuite, on rencontre, sur le chemin, des options qui indiquent ce qu'il faut faire de ce chemin.

Par exemple, `\path (0,0) circle (1cm);` signifie « c'est un chemin constitué d'un cercle de centre l'origine. Ne fais rien avec (jette le) ». Toutefois, si l'option `draw` est rencontrée n'importe où sur le chemin, le cercle sera dessiné. « N'importe où » est n'importe quel point du chemin où une option peut être donnée autrement dit n'importe où sont autorisées des commandes comme `circle (1cm)` ou `rectangle (1,1)` ou même simplement `(0,0)`. Ainsi les commandes suivantes tracent toutes le même cercle :

```
\path [draw] (0,0) circle (1cm);
\path (0,0) [draw] circle (1cm);
\path (0,0) circle (1cm) [draw];
```

Enfin, `\draw (0,0) circle (1cm);` trace également un chemin puisque `\draw` est une abréviation de `\path [draw]` et ainsi la commande se développe en la première ligne de l'exemple ci-dessus.

Semblablement, `\fill` est une abréviation de `\path[fill]` et `\filldraw` est une abréviation de la commande `\path[fill,draw]`. Puisque les options s'ajoutent les unes aux autres, les commandes suivantes ont toutes le même effet :

```
\path [draw,fill]   (0,0) circle (1cm);
\path [draw] [fill] (0,0) circle (1cm);
\path [fill] (0,0) circle (1cm) [draw];
\draw [fill] (0,0) circle (1cm);
\fill (0,0) [draw] circle (1cm);
\filldraw (0,0) circle (1cm);
```

Dans les sections suivantes on explique les différentes actions que l'on peut accomplir avec un chemin. Les commandes suivantes sont des abréviations de certains ensembles d'actions mais il n'y a pas d'abréviation pour beaucoup de combinaisons utiles.

`\draw`

Dans `{tikzpicture}` c'est une abréviation de `\path[draw]`.

`\fill`

Dans `{tikzpicture}` c'est une abréviation de `\path[fill]`.

`\filldraw`

Dans `{tikzpicture}` c'est une abréviation de `\path[fill,draw]`.

`\shade`

Dans `{tikzpicture}` c'est une abréviation de `\path[shade]`.

`\shadedraw`

Dans `{tikzpicture}` c'est une abréviation de `\path[shade,draw]`.

`\clip`

Dans `{tikzpicture}` c'est une abréviation de `\path[clip]`.

`\useasboundingbox`

Dans `{tikzpicture}` c'est une abréviation de `\path[use as bounding box]`.

`\node`

Dans `{tikzpicture}` c'est une abréviation de `\path node`. Notez, cette fois, que `node` n'est pas une option mais une opération de chemin.

`\coordinate`

Dans `{tikzpicture}` c'est une abréviation de `\path coordinate`.

10.1 Spécification de couleur

Les options les moins spécifiques pour fixer les couleurs sont les suivantes :

– `color=`⟨*nom de couleur*⟩
Cette option fixe la couleur utilisée pour remplir, dessiner et pour le texte dans la portée courante. Toute spécification de couleur de remplissage ou de dessin est immédiatement « annulée » par cette option.
Le ⟨*nom de couleur*⟩ est le nom d'une couleur définie au préalable. Pour les utilisateurs de LaTeX c'est juste une couleur LaTeXienne normale et les extensions `xcolor` sont autorisées. Voici un exemple :


```
\tikz \fill[color=red!20] (0,0) circle (1ex);
```

On peut se passer de la partie `color=` est écrire aussi :


```
\tikz \fill[red!20] (0,0) circle (1ex);
```

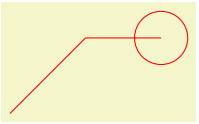
Ce qui se passe c'est que chaque option inconnue de Ti*k*Z, comme `red!20`, se voit offrir une deuxième chance comme nom de couleur.
Pour les utilisateurs de plain TeX ce n'est pas aussi facile de définir des couleurs puisque plain TeX n'a pas de mécanisme standard de nommage de couleur. De ce fait, PGF simule l'extension `xcolor`, bien que la simulation soit *extrêmement simple* (plus précisément, ce que j'ai pu bidouiller en environ deux heures). La simulation vous permet ce qui suit :
  – Définir une nouvelle couleur avec `\definecolor`. Seuls deux modèles de couleurs sont gérés : `gray` et `rgb`.
    Exemple : `\definecolor{orange}{rgb}{1,0.5,0}`
  – Utiliser `\colorlet` pour définir une nouvelle couleur à partir d'une ancienne. Ici le mécanisme `!` est géré quoique une seule fois. (Utiliser plusieurs `\colorlet` pour des couleurs plus frivoles.)
    Exemple : `\colorlet{lightgray}{black!25}`
  – Utiliser `\color{`⟨*nom de couleur*⟩`}` pour activer la couleur dans le groupe courant de TeX. Un bidouillage à coup de `\aftergroup` sert à rétablir la couleur après le groupe.

Comme signalé précédemment, l'option `color` s'applique à « tout » (sauf au dégradé), ce qui n'est pas toujours ce que l'on veut. À cause de cela, il existe plusieurs options de coloriage plus spécialisées. Par exemple, l'option `draw=` fixe la couleur du tracé mais ne modifie pas celle utilisée pour remplir. Ces options de couleur sont documentées là où sont décrites les actions de chemin sur lesquelles elles agissent.

10.2 Dessiner un chemin

On peut dessiner un chemin avec l'option suivante :

– `draw=`⟨*color*⟩ Fait que le chemin est dessiné. L'action de « dessiner » (alias « tracer ») peut être vue comme l'action de prendre un crayon et de le déplacer le long du chemin et, ce faisant, laisser de « l'encre » sur la toile.
De nombreux paramètres influent sur la manière dont une ligne est dessinée comme l'épaisseur ou le motif de pointillés. On expliquent ces options ci-dessous.
Si l'on passe l'argument facultatif ⟨*couleur*⟩, le dessin est réalisé avec cette ⟨*couleur*⟩. Cette couleur peut être différente de celle utilisée pour remplir, cela permet de dessiner et remplir un chemin avec des couleurs différentes. Si aucun argument ⟨*couleur*⟩ n'est donné, on utilise la valeur de la dernière option `color=`.
Si l'on passe le nom spécial de couleur `none` (*aucun*), cela entraîne que le dessin est « désactivé ». Cela est utile si un style a précédemment activé le dessin et qu'on veut suspendre cet effet localement.
Quoique cette option soit normalement utilisée sur des chemins pour indiquée que le chemin devrait être dessiné, cela a un sens de l'utiliser avec un environnement `{scope}` ou `{tikzpicture}`. Toutefois cela *ne* fera *pas* que tout le chemin soit dessiné. Plutôt, cela fixe simplement la ⟨*couleur*⟩ à utiliser pour dessiner les chemins à l'intérieur de l'environnement.



```
\begin{tikzpicture}
  \path[draw=red] (0,0) -- (1,1) -- (2,1) circle (10pt);
\end{tikzpicture}
```

Les sections suivantes listent les différentes options qui influent sur la manière dont un chemin est dessiné. Toutes ces options n'ont d'effet qui si l'option `draw` est passée (directement ou indirectement).

10.2.1 Paramètres graphiques : Line Width (largeur de ligne), Line Cap (extrémité de ligne), et Line Join (raccord de lignes)

– `line width`=⟨*dimension*⟩ Définit la largeur (*width*) de la ligne. Notez l'espace. Par défaut : `0.4pt`.

```
\tikz \draw[line width=5pt] (0,0) -- (1cm,1.5ex);
```

Il y a un certain nombre de styles prédéfinis qui fournissent des façons plus « naturelles » de fixer la largeur de la ligne. On peut aussi redéfinir ces styles. On se souviendra que l'on peut ne pas écrire le `style=` lorsque l'on active un style.

– `style=ultra thin` Fixe la largeur de ligne à 0,1 pt.

```
\tikz \draw[ultra thin] (0,0) -- (1cm,1.5ex);
```

– `style=very thin` Fixe la largeur de ligne à 0,2 pt.

```
\tikz \draw[very thin] (0,0) -- (1cm,1.5ex);
```

– `style=thin` Fixe la largeur de ligne à 0,4 pt.

```
\tikz \draw[thin] (0,0) -- (1cm,1.5ex);
```

– `style=semithick` Fixe la largeur de ligne à 0,6 pt.

```
\tikz \draw[semithick] (0,0) -- (1cm,1.5ex);
```

– `style=thick` Fixe la largeur de ligne à 0,8 pt.

```
\tikz \draw[thick] (0,0) -- (1cm,1.5ex);
```

– `style=very thick` Fixe la largeur de ligne à 1,2 pt.

```
\tikz \draw[very thick] (0,0) -- (1cm,1.5ex);
```

– `style=ultra thick` Fixe la largeur de ligne à 1,6 pt.

```
\tikz \draw[ultra thick] (0,0) -- (1cm,1.5ex);
```

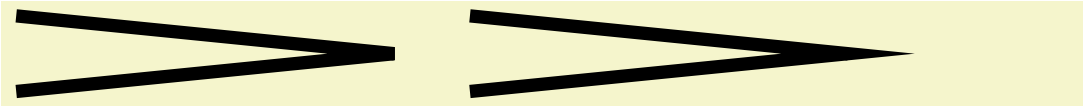
– `cap`=⟨*type*⟩ Définit comment les lignes « finissent ». Les valeurs autorisées de ⟨*type*⟩ sont `round` (*arrondi*), `rect` et `butt` (*bout*) valeur par défaut. Voici leurs effets :

```
\begin{tikzpicture}
  \begin{scope}[line width=10pt]
    \draw[cap=rect]  (0,0 ) -- (1,0);
    \draw[cap=butt]  (0,.5) -- (1,.5);
    \draw[cap=round] (0,1 ) -- (1,1);
  \end{scope}
  \draw[white,line width=1pt]
    (0,0 ) -- (1,0) (0,.5) -- (1,.5) (0,1 ) -- (1,1);
\end{tikzpicture}
```

– `join`=⟨*type*⟩ Définit la façon dont les lignes sont « jointes ». Les ⟨*type*⟩ autorisés sont `round` (*arrondi*), `bevel` (*biseau*) et `miter` (*anglet*) valeur par défaut. Voici leurs effets :

```
\begin{tikzpicture}[line width=10pt]
  \draw[join=round] (0,0) -- ++(.5,1) -- ++(.5,-1);
  \draw[join=bevel] (1.25,0) -- ++(.5,1) -- ++(.5,-1);
  \draw[join=miter] (2.5,0) -- ++(.5,1) -- ++(.5,-1);
  \useasboundingbox (0,1.5); % agrandit la boite-cadre
\end{tikzpicture}
```

– `miter limit`=⟨*facteur*⟩ Lorsque l'on utilise le raccord (*join*) en anglet `miter` et que se présente un angle très aigu (très petit), le raccord peut dépasser de beaucoup le point réel de raccord. Dans ce cas, s'il devait dépasser de plus de ⟨*facteur*⟩ fois la largeur de ligne, le raccord en anglet serait remplacer par un raccord en biseau. La valeur par défaut est `10`.



```
\begin{tikzpicture}[line width=5pt]
  \draw                 (0,0) -- ++(5,.5) -- ++(-5,.5);
  \draw[miter limit=25] (6,0) -- ++(5,.5) -- ++(-5,.5);
  \useasboundingbox (14,0); % agrandit la boite-cadre
\end{tikzpicture}
```

10.2.2 Paramètres graphiques : Dash Pattern (motif de pointillés)

– `dash pattern`=⟨*dash pattern*⟩ Définit le motif de pointillés. La syntaxe est identique à celle de METAFONT. Par exemple, `on 2pt off 3pt on 4pt off 4pt` signifie « tracer 2 pt puis lever le crayon pendant 3 pt puis retracer 4 pt puis relever le crayon pendant 4 pt, répéter ».


```
\begin{tikzpicture}[dash pattern=on 2pt off 3pt on 4pt off 4pt]
  \draw (0pt,0pt) -- (3.5cm,0pt);
\end{tikzpicture}
```

– `dash phase`=⟨*phase*⟩ Déplace le début du motif de pointillés de la valeur ⟨*phase*⟩.


```
\begin{tikzpicture}[dash pattern=on 20pt off 10pt]
  \draw[dash phase=0pt] (0pt,3pt) -- (3.5cm,3pt);
  \draw[dash phase=10pt] (0pt,0pt) -- (3.5cm,0pt);
\end{tikzpicture}
```

De même que pour la largeur de ligne, quelques styles prédéfinis permettent de spécifier commodément un motif de pointillé.

– `style=solid` Raccourci pour spécifier un trait continu comme « motif de pointillé ». C'est la valeur par défaut [16].


```
\tikz \draw[solid] (0pt,0pt) -- (50pt,0pt);
```

– `style=dotted` Raccourci pour spécifier un motif de points.


```
\tikz \draw[dotted] (0pt,0pt) -- (50pt,0pt);
```

– `style=densely dotted` Raccourci pour spécifier un motif de points serrés.


```
\tikz \draw[densely dotted] (0pt,0pt) -- (50pt,0pt);
```

– `style=loosely dotted` Raccourci pour spécifier un motif de points espacés.


```
\tikz \draw[loosely dotted] (0pt,0pt) -- (50pt,0pt);
```

– `style=dashed` Raccourci pour spécifier un motif de tirets.


```
\tikz \draw[dashed] (0pt,0pt) -- (50pt,0pt);
```

– `style=densely dashed` Raccourci pour spécifier un motif de tirets serrés.


```
\tikz \draw[densely dashed] (0pt,0pt) -- (50pt,0pt);
```

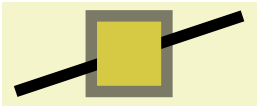
– `style=loosely dashed` Raccourci pour spécifier un motif de tirets espacés.


```
\tikz \draw[loosely dashed] (0pt,0pt) -- (50pt,0pt);
```

10.2.3 Paramètres graphiques : Draw Opacity (opacité de dessin)

Normalement, une ligne est dessiné « occulte » tout ce qui se trouve dessous comme si l'on se servait d'une encre parfaitement opaque. Il est également possible de demander à Ti*k*Z d'utiliser une encre un petit peu (ou un gros peu) transparente. Pour ce faire, utiliser l'option suivante :

– `draw opacity`=⟨*valeur*⟩ Cette option fixe le degré de transparence des lignes. La valeur `1` signifie « totalement opaque » ou « pas transparent du tout », la valeur `0` signifie « complètement transparente » ou « invisible ». La valeur `0.5` produit des lignes semitransparentes.
Notez que lorsque l'on utilise PostScript comme format de sortie, cette option ne marche qu'avec des version récentes de GhostScript.



```
\begin{tikzpicture}[line width=1ex]
  \draw (0,0) -- (3,1);
  \filldraw [fill=examplefill,draw opacity=0.5] (1,0) rectangle (2,1);
\end{tikzpicture}
```

16. NdTdS : L'auteur a déjà dit tout le mal qu'il pensait des pointillés.

Notez que l'option `draw opacity` ne fixe que l'opacité des lignes tracées. L'opacité du remplissage est fixée avec l'option `fill opacity` (documentée dans la section 10.3.2, p. 81). L'option `opacity` fixe les deux en même temps.

– opacity=⟨*valeur*⟩ Fixe tant l'opacité de dessin que de remplissage à ⟨*valeur*⟩ .
  Les styles prédéfinis suivants rendent plus facile l'utilisation de cette option :
  – style=transparent Rend tout totalement transparent, et, donc, invisible.

```
\tikz{\fill[red]             (0,0)   rectangle (1,0.5);
      \fill[transparent,red] (0.5,0) rectangle (1.5,0.25); }
```

  – style=ultra nearly transparent Rend tout, euh, ultra presque transparent.

```
\tikz{\fill[red]                       (0,0)   rectangle (1,0.5);
      \fill[ultra nearly transparent] (0.5,0) rectangle (1.5,0.25); }
```

  – style=very nearly transparent

```
\tikz{\fill[red]                      (0,0)   rectangle (1,0.5);
      \fill[very nearly transparent] (0.5,0) rectangle (1.5,0.25); }
```

  – style=nearly transparent

```
\tikz{\fill[red]                (0,0)   rectangle (1,0.5);
      \fill[nearly transparent] (0.5,0) rectangle (1.5,0.25); }
```

  – style=semitransparent

```
\tikz{\fill[red]             (0,0)   rectangle (1,0.5);
      \fill[semitransparent] (0.5,0) rectangle (1.5,0.25); }
```

  – style=nearly opaque

```
\tikz{\fill[red]           (0,0)   rectangle (1,0.5);
      \fill[nearly opaque] (0.5,0) rectangle (1.5,0.25); }
```

  – style=very nearly opaque

```
\tikz{\fill[red]                (0,0)   rectangle (1,0.5);
      \fill[very nearly opaque] (0.5,0) rectangle (1.5,0.25); }
```

  – style=ultra nearly opaque

```
\tikz{\fill[red]                 (0,0)   rectangle (1,0.5);
      \fill[ultra nearly opaque] (0.5,0) rectangle (1.5,0.25); }
```

  – style=opaque Cela produit des dessins complètement opaque, ce qui est le comportement par défaut.

```
\tikz{\fill[red]    (0,0)   rectangle (1,0.5);
      \fill[opaque] (0.5,0) rectangle (1.5,0.25); }
```

10.2.4 Paramètres graphiques : Arrow Tips (pointes de flèche)

Lorsque l'on dessine une ligne, on peut ajouter une pointe de flèche à chaque extrémité. On ne peut ajouter qu'une seule pointe de flèche au début et une seule à la fin. Si le chemin est constitué de plusieurs segments, seuls le dernier segment prend les pointes de flèche. Le comportement des chemins fermés n'est pas défini et peut changer à l'avenir.

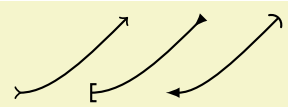
– arrows=⟨*type de flèche au début*⟩-⟨*type de flèche à la fin*⟩ Cette option fixe les pointes de flèche au début et à la fin (une valeur vide comme dans `->` indique qu'aucune pointe ne sera dessiné au début). *Note : puisque l'option* `arrow` *est si souvent utilisée, on peut se dispenser du texte* `arrows=`. Ce qui se passe est que toute option qui contient un `-` est interprétée comme une spécification de flèche.

```
\begin{tikzpicture}
  \draw[->]         (0,0)   -- (1,0);
  \draw[o-stealth] (0,0.3) -- (1,0.3);
\end{tikzpicture}
```

Les valeurs autorisées sont toutes les pointes de flèche prédéfinies mais on peut également en définir de nouvelles comme expliqué en section **??**, p. **??**. C'est souvent nécessaire pour obtenir des pointes de flèche « doubles » et des pointes de taille fixe. Comme la bibliothèque `pgflibraryarrows` est chargée par défaut, toutes les pointes décrites dans la section **??**, p. **??** sont disponibles.

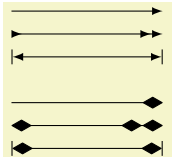
Un type de pointes de flèche est spécial : `>` (et tous les types de pointes de flèche qui contiennent `>` comme `<<` ou `>|`). Ces types de pointes ne sont pas fixés. Ou plutôt, on peut les redéfinir en utilisant l'option `>=` (voir ci-dessous).

Exemple : On peut combiner les types de pointes comme dans



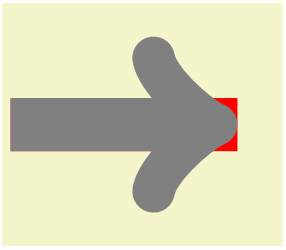
```
\begin{tikzpicture}[thick]
  \draw[to reversed-to]   (0,0) .. controls +(.5,0) and +(-.5,-.5) .. +(1.5,1);
  \draw[[-latex reversed] (1,0) .. controls +(.5,0) and +(-.5,-.5) .. +(1.5,1);
  \draw[latex-)]          (2,0) .. controls +(.5,0) and +(-.5,-.5) .. +(1.5,1);
  \useasboundingbox (-.1,-.1) rectangle (3.1,1.1); % agrandit la boite-cadre
\end{tikzpicture}
```

– `>=`⟨*type de flèche de fin*⟩ On peut utiliser cette option pour redéfinir la pointe de flèche « standard » `>`. La motivation de ce comportement est que des gens différents ont des idées différentes à propos du genre de pointe de flèche dont on devrait se servir normalement. Je préfère la pointe de la commande TEXienne `\to` (que l'on utilise dans dans choses comme $f\colon A \to B$). D'autre préféreront la flèche standard de LATEX dont l'apparence est la suivante $\rightarrow$. Comme le type de pointe `>` est certainement le plus « naturel » que l'on peut utiliser, il est conservé libre de toute signification prédéfinie. Ou plutôt, on peut le changer avec `>=to` pour définir la pointe de flèche « standard » comme la flèche TEXienne, alors que `> =latex` la définira comme la pointe de flèche LATEXienne et que l'on utilisera `>=stealth` pour obtenir une pointe de flèche à la PSTRICKS.
Outre la redéfinition du type de pointe de flèche `>` (et `<` pour le début), cette option réalise la redéfinition des types suivants : `>` et `<` comme version échangée pour ⟨*type de pointe de fin*⟩, `<<` et `>>` comme version double, `>>` et `<<` comme version double échangée et `|<` et `>|` comme pointe de flèche terminée par une barre verticale.



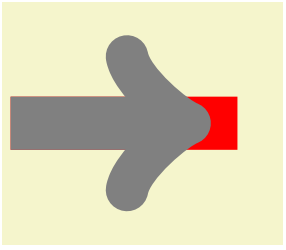
```
\begin{tikzpicture}[scale=2]
  \begin{scope}[>=latex]
    \draw[->]    (0pt,6ex) -- (1cm,6ex);
    \draw[>->>]  (0pt,5ex) -- (1cm,5ex);
    \draw[|<->|] (0pt,4ex) -- (1cm,4ex);
  \end{scope}
  \begin{scope}[>=diamond]
    \draw[->]    (0pt,2ex) -- (1cm,2ex);
    \draw[>->>]  (0pt,1ex) -- (1cm,1ex);
    \draw[|<->|] (0pt,0ex) -- (1cm,0ex);
  \end{scope}
\end{tikzpicture}
```

– `shorten >=`⟨*dimension*⟩ Cette option raccourcit la fin des lignes de la ⟨*dimension*⟩ donnée. Si l'on spécifie une pointe de flèche, les lignes sont déjà raccourcies de telle sorte que la pointe de la flèche touche l'extrémité spécifiée et ne « dépasse » pas ce point. Voici un exemple :



```
\begin{tikzpicture}[line width=20pt]
  \useasboundingbox (0,-1.5) rectangle (3.5,1.5);
  \draw[red]        (0,0) -- (3,0);
  \draw[gray,->]    (0,0) -- (3,0);
\end{tikzpicture}
```

L'option `shorten >` (*raccourcir*) permet de raccourcir la fin de la ligne *additionnellement* de la distance donnée. Cette option peut être aussi utile même si l'on n'a pas spécifié de pointe du tout.

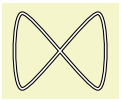


```
\begin{tikzpicture}[line width=20pt]
  \useasboundingbox (0,-1.5) rectangle (3.5,1.5);
  \draw[red]                      (0,0) -- (3,0);
  \draw[-to,shorten >=10pt,gray] (0,0) -- (3,0);
\end{tikzpicture}
```

– `shorten <=`⟨*dimension*⟩ fonctionne comme `shorten >` mais pour le début.

10.2.5 Paramètres graphiques : Double Lines (lignes doubles) et Bordered Lines (lignes longées)

– `double=`⟨*couleur de fond*⟩ Cette option fait que « deux » lignes sont tracées au lieu d'une seule. Toutefois, ce n'est pas vraiment ce qui arrive. En fait, le chemin est tracé deux fois. D'abord, avec la couleur normale de dessin puis, la deuxième fois, avec la ⟨*couleur de fond*⟩ qui est normalement `white` (*blanc*). La seconde fois, la ligne est réduite. L'effet final est qu'il semble qu'on a tracé deux lignes et cela marche bien même avec des chemins courbes et complexes.

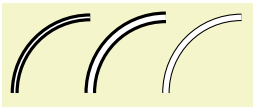


```
\tikz \draw[double]
  plot[smooth cycle] coordinates{(0,0) (1,1) (1,0) (0,1)};
```

On peut aussi utiliser l'option de doublement pour créer un effet par lequel une ligne semble être longée par une « bordure ».

```
\begin{tikzpicture}
  \draw (0,0) -- (1,1);
  \draw[draw=white,double=red,very thick] (0,1) -- (1,0);
\end{tikzpicture}
```

– `double distance=`⟨*dimension*⟩ Fixe la distance séparant les « deux » lignes (par défaut 0,6 pt). En fait, c'est l'épaisseur de la ligne utilisée pour tracé le chemin la deuxième fois. L'épaisseur de la *première* ligne avec laquelle on trace le chemin la première fois est la largeur de ligne normale plus la ⟨*dimension*⟩ donnée. Cette option a pour effet secondaire d'activer l'option `double`.

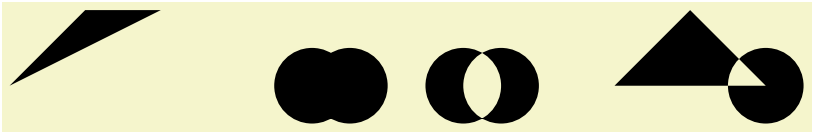


```
\begin{tikzpicture}
  \draw[very thick,double]              (0,0) arc (180:90:1cm);
  \draw[very thick,double distance=2pt] (1,0) arc (180:90:1cm);
  \draw[thin,double distance=2pt]       (2,0) arc (180:90:1cm);
\end{tikzpicture}
```

10.3 Remplir un chemin

Pour remplir un chemin, utiliser l'option suivante :

– `fill=`⟨*color*⟩ Cette option fait que le chemin est rempli. Toutes les parties ouvertes du chemin sont d'abord fermées si nécessaire. Ensuite, la surface entourée par le chemin est remplie avec la couleur de remplissage courante, couleur qui est soit la dernière couleur spécifiée à l'aide de l'option générale `color=` soit la couleur facultative ⟨*couleur*⟩. Pour ce qui est des chemins auto-sécants et ceux qui consistent en plusieurs surface closes, la « surface entourée » est quelque peu difficile à définir et deux définitions différentes existent, à savoir la règle de l'indice non-nul (*nonzero winding number*) et la règle de parité (*even odd*). Voir l'explication de ces options ci-dessous.
De la même manière que pour l'option `draw`, fixer la ⟨*couleur*⟩ à `none` (*aucun*) désactive localement le remplissage.



```
\begin{tikzpicture}
  \fill (0,0) -- (1,1) -- (2,1);
  \fill (4,0) circle (.5cm)  (4.5,0) circle (.5cm);
  \fill[even odd rule] (6,0) circle (.5cm)  (6.5,0) circle (.5cm);
  \fill (8,0) -- (9,1) -- (10,0) circle (.5cm);
\end{tikzpicture}
```

Si l'on utilise l'option `fill` avec l'option `draw` (soit que les deux options aient été données soit qu'on ait utilisé la commande `\filldraw`), le chemin est *d'abord* rempli puis, *dans deuxième temps*, il est dessiné. C'est particulièrement utile quand on sélectionne des couleurs différentes pour le dessin et le remplissage. Même si la même couleur est utilisée, il y a une différence ente cette commande et la simple option `fill` : une surface « dessinée-remplie » (*filldrawn*) sera un peu plus large que la surface simplement remplie du fait de l'épaisseur du « crayon ».

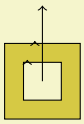
```
\begin{tikzpicture}[fill=examplefill,line width=5pt]
  \filldraw (0,0) -- (1,1) -- (2,1);
  \filldraw (4,0) circle (.5cm)  (4.5,0) circle (.5cm);
  \filldraw[even odd rule] (6,0) circle (.5cm)  (6.5,0) circle (.5cm);
  \filldraw (8,0) -- (9,1) -- (10,0) circle (.5cm);
\end{tikzpicture}
```

10.3.1 Paramètres graphiques : Interior Rules (règles pour l'intérieur)

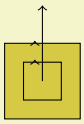
On peut se servir des deux options suivantes pour décider la façon dont seront déterminés les points intérieurs :

– `nonzero rule` Si cette règle est utilisée (ce qui est le comportement par défaut), la méthode suivante est utilisée pour déterminer si un point donné est « à l'intérieur » du chemin : on trace une demi-droite d'origine le point considéré (on choisit la direction de telle sorte qu'aucun cas limite étrange n'apparaisse). Alors la demi-droite peut couper le chemin. À chaque fois qu'elle coupe le chemin, on incrémente ou décrémente un compteur qui vaut zéro au départ. Si la demi-droite coupe le chemin quand ce dernier va « de gauche à droite » (relativement à la demi-droite) le compteur est incrémenté, sinon il est décrémenté. Alors, à la fin, on regarde si le compteur est non-nul (d'où le nom *nonzero*). Dans ce cas, le point est considéré comme « intérieur » sinon il est considéré comme « extérieur ». Cela paraît compliqué, non ? Ça l'est !

crossings : $-1+1=0$



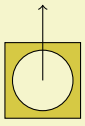
crossings : $1+1=2$



```
\begin{tikzpicture}
  \filldraw[fill=examplefill]
  % rectangle dans le sens indirect (dans le sens des aiguilles d'une montre)
  (0,0) -- (0,1) -- (1,1) -- (1,0) -- cycle
  % rectangle dans le sens direct
  (0.25,0.25) -- (0.75,0.25) -- (0.75,0.75) -- (0.25,0.75) -- cycle;

  \draw[->] (0,1) (.4,1);
  \draw[->] (0.75,0.75) (0.3,.75);

  \draw[->] (0.5,0.5) -- +(0,1) node[above] {crossings: $-1+1 = 0$};

  \begin{scope}[yshift=-3cm]
    \filldraw[fill=examplefill]
    % rectangle dans le sens indirect
    (0,0) -- (0,1) -- (1,1) -- (1,0) -- cycle
    % rectangle dans le sens indirect
    (0.25,0.25) -- (0.25,0.75) -- (0.75,0.75) -- (0.75,0.25) -- cycle;

    \draw[->] (0,1) (.4,1);
    \draw[->] (0.25,0.75) (0.4,.75);

    \draw[->] (0.5,0.5) -- +(0,1) node[above] {crossings: $1+1 = 2$};
  \end{scope}
\end{tikzpicture}
```

– `even odd rule` Cette option fait qu'une autre méthode est utilisée pour déterminer l'intérieur et l'extérieur des chemins. Quoique moins souple, elle apparaît comme plus intuitive.
Dans cette méthode, on trace aussi une demi-droite depuis le point pour lequel on se pose la question de savoir s'il se trouve à l'intérieur de la région à remplir. Toutefois, cette fois, on compte simplement le nombre d'intersections avec le chemin et un point est déclaré être « intérieur » si le nombre d'intersections est impair.
Avec cette règle de parité, on peut facilement « percer des trous » dans un chemin.

crossings : $1+1=2$

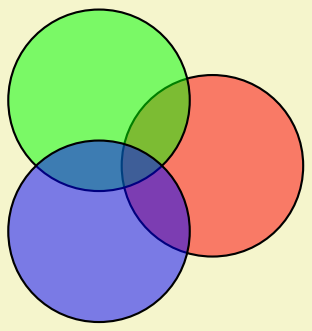


```
\begin{tikzpicture}
  \filldraw[fill=examplefill,even odd rule]
    (0,0) rectangle (1,1) (0.5,0.5) circle (0.4cm);
  \draw[->] (0.5,0.5) -- +(0,1) [above] node{crossings: $1+1 = 2$};
\end{tikzpicture}
```

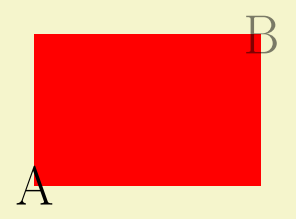
10.3.2 Paramètres graphiques : Fill Opacity (opacité de remplissage)

De la même manière qu'avec `draw opacity`, on peut fixer l'opacité du remplissage.

– `fill opacity`=⟨*valeur*⟩ Cette option fixe l'opacité des remplissages. Outre les opérations de remplissage, cette option concerne aussi le texte et les images.
Notez, une fois de plus, que si l'on utilise PostScript comme format de sortie, cette option ne marche qu'avec des versions récentes de GhostScript.



```
\begin{tikzpicture}[thick,fill opacity=0.5]
  \filldraw[fill=red]   (0:1cm)    circle (12mm);
  \filldraw[fill=green] (120:1cm)  circle (12mm);
  \filldraw[fill=blue]  (-120:1cm) circle (12mm);
\end{tikzpicture}
```




```
\begin{tikzpicture}
  \fill[red] (0,0) rectangle (3,2);

  \node                    at (0,0) {\huge A};
  \node[fill opacity=0.5] at (3,2) {\huge B};
\end{tikzpicture}
```

10.4 Appliquer un dégradé à un chemin

On peut appliquer un dégradé[17] à un chemin avec l'option `shade` (littéralement *ombre*). Un dégradé ressemble à un remplissage mais le dégradé change de couleur doucement d'une couleur à une autre.

– `shade` Fait que l'on applique au chemin le dégradé courant (plus de détails plus loin). Si cette option est utilisée avec l'option `draw`, alors on applique un dégradé au chemin puis on le dessine.
Utiliser cette option avec l'option `fill` ne crée pas d'erreur mais n'a aucun sens.


```
\tikz \shade (0,0) circle (1ex);
```

```
\tikz \shadedraw (0,0) circle (1ex);
```

On ne voit pas vraiment clairement comment certains dégradés pourront être appliqué à un chemin. Par exemple, le dégradé `ball` ressemble normalement à ceci . Comment cela doit-il être appliqué à un rectangle ? Ou à un triangle ?

Pour résoudre ce problème, les dégradés prédéfinis comme `ball` (*balle*) ou `axis` (*axes*) remplissent complètement un grand rectangle de manière judicieuse. Aussi, quand un chemin est estompé, ce qui se passe vraiment est que le chemin est utilisé temporairement pour découper et qu'ensuite le dégradé est appliqué au rectangle, puis mis à l'échelle et déplacé de telle sorte que toutes les parties du chemin soient remplies.

10.4.1 Choisir un type de dégradé

Le dégradé par défaut est une transition douce du gris au blanc appliquée de haut en bas. Toutefois, d'autres dégradés sont également possibles, par exemple, un dégradé qui étendra une couleur depuis le centre vers les coins. Pour choisir le dégradé, on peut se servir de l'option `shading=` qui invoquera automatiquement l'option `shade`. Notez que cela *ne change pas* la couleur du dégradé mais simplement la manière dont la couleur est étendue. Pour changer les couleurs, on a besoin d'autres options qui sont expliquées ci-dessous.

– `shading`=⟨*nom*⟩ Sélectionne le dégradé nommé ⟨*nom*⟩. Les dégradés suivants sont prédéfinis :

17. NdTdS : On utilisera aussi le verbe « estomper » en lieu et place de la locution « appliquer un dégradé » même si le sens n'est pas exactement le même.

– `axis` C'est le dégradé par défaut dans lequel la couleur change graduellement entre trois lignes horizontales. La ligne du haut est au point le plus haut du chemin, celle du milieu au milieu, celle du bas au point le plus bas du chemin.

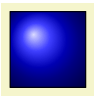
```
\tikz \shadedraw [shading=axis] (0,0) rectangle (1,1);
```

La couleur du haut par défaut est le gris, celle du bas par défaut est le blanc, celle du milieu par défaut est le « milieu » de ces deux premières.

– `radial` Ce dégradé remplit le chemin avec un changement graduel d'une certaine couleur au centre jusqu'à une autre couleur sur le bord. Si le chemin est un cercle, la couleur extérieure sera atteinte exactement à la bordure. Sinon la couleur extérieure sera utilisée encore un peu jusqu'aux coins. La couleur intérieure par défaut est le gris et la couleur extérieur le blanc.

```
\tikz \shadedraw [shading=radial] (0,0) rectangle (1,1);
```

– `ball` Ce dégradé remplit un chemin de telle façon que « ça ressemble à une balle ». La « couleur » par défaut de la balle est le bleu (sans raison particulière).



```
\tikz \shadedraw [shading=ball] (0,0) rectangle (1,1);
```

```
\tikz \shadedraw [shading=ball] (0,0) circle (.5cm);
```

– `shading angle=`⟨*degrés*⟩ Cette option fait tourner le dégradé (pas le chemin !) de l'angle donné. Par exemple, on peut transformer un dégradé axial de haut en bas pour en faire un dégradé de gauche à droite en le faisant tourner de 90°.

```
\tikz \shadedraw [shading=axis,shading angle=90] (0,0) rectangle (1,1);
```

On peut aussi définit de nouveau type de dégradé soi-même. Toutefois, pour cela, on a besoin d'utiliser directement la couche de base ce qui est, enfin, plus fondamental et plus difficile à utiliser. Les détails sur la création de dégradé applicable à des chemins sont donnés dans la section **??**, p. **??**.

10.4.2 Choisir les couleurs du dégradé

On peut se servir des options qui suivent pour changer les couleurs utilisées dans les dégradés. Quand une de ces options est passée, l'option `shade` est automatiquement sélectionnée ainsi que le dégradé « idoine ».

– `top color=`⟨*couleur*⟩ Cette option spécifie la couleur à utiliser en haut du dégradé `axis`. Quand cette option est passée, plusieurs choses se passent :

1. L'option `shade` est sélectionnée ;
2. L'option `shading=axis` est sélectionnée ;
3. La couleur centrale du dégradé axial est fixée à la moyenne de la ⟨*couleur*⟩ donnée pour le haut et de la couleur, quelle qu'elle soit, couramment sélectionnée pour le bas ;
4. L'angle de rotation du dégradé est fixé à 0.

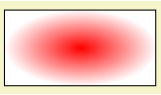
```
\tikz \draw[top color=red] (0,0) rectangle (2,1);
```

– `bottom color=`⟨*couleur*⟩ Cette option marche comme `top color` mais pour la couleur du bas.

– middle color=⟨*couleur*⟩ Cette option définit la couleur du milieu d'un dégradé axial. Elle aussi sélectionne les options `shade` et `shading=axis` mais elle ne change pas l'angle de rotation.
Note : Comme les deux options `top color` et `bottom color` changent la couleur du milieu, cette option devrait être donnée *en dernier* si toutes ces options devaient être utilisées :

```
\tikz \draw[top color=white,bottom color=black,middle color=red]
  (0,0) rectangle (2,1);
```

– left color=⟨*couleur*⟩ Cette option fait exactement la même chose que `top color` sauf qu'elle fixe, de plus, l'angle de rotation à 90°.
– right color=⟨*couleur*⟩ Fonctionne comme `left color`.
– inner color=⟨*couleur*⟩ Cette option fixe la couleur utilisée au centre d'un dégradé `radial`. Quand cette option est utilisée, les options `shade` et `shading=radial` sont sélectionnées.



```
\tikz \draw[inner color=red] (0,0) rectangle (2,1);
```

– outer color=⟨*couleur*⟩ Cette option définit la couleur utilisée sur le bord et à l'extérieur d'un dégradé `radial`.

```
\tikz \draw[outer color=red,inner color=white]
  (0,0) rectangle (2,1);
```

– ball color=⟨*couleur*⟩ Cette option fixe la couleur utilisée pour un dégradé `ball`. Elle sélectionne les options `shade` et `shading=ball`. Notez que la balle n'aura jamais « complètement » la couleur ⟨*couleur*⟩. Au point « éclairé » on ajoute une certaine quantité de blanc et sur le bord une certaine quantité de noir. De ce fait, il est tout à fait possible de choisir `ball color=white` ou `ball color=black`.

```
\begin{tikzpicture}
  \shade[ball color=white] (0,0) circle (2ex);
  \shade[ball color=red] (1,0) circle (2ex);
  \shade[ball color=black] (2,0) circle (2ex);
\end{tikzpicture}
```

10.5 Établir une boîte-cadre

PGF garde assez bien la trace de la taille de la figure et réserve juste la bonne quantité de place pour elle dans le document principal. Toutefois, dans certain cas, on voudrait pouvoir dire des choses comme « ne tiens pas compte de ça pour la taille de la figure » ou « la figure est, en fait, un peu plus grande ». Pour cela on peut se servir de l'option `use as bounding box` ou la commande `\useasboundingbox` qui n'est qu'une abréviation de `\path[use as bounding box]` (*utiliser en tant que boîte-cadre*).

– use as bounding box Normalement, quand on utilise cette option sur un chemin, la boîte-cadre du chemin courant est utilisée pour déterminer la taille de la figure et les tailles de tous les chemins *suivants* sont ignorées. Toutefois, si une opération de chemin a établir précédemment une boîte-cadre plus grande, cette opération ne la rapetissera pas.
Dans un sens, `use as bounding box` a le même effet que le découpage de tout le reste du dessin par le chemin courant — sans découper véritablement mais en faisant que PGF traite tout comme si c'était découpé.
La première application de cette option permet d'obtenir une `{tikzpicture}` qui déborde sur le texte principal :

Gauche de la figure droite de la figure.

```
Gauche de la figure\begin{tikzpicture}
  \draw[use as bounding box] (2,0) rectangle (3,1);
  \draw (1,0) -- (4,.75);
\end{tikzpicture}droite de la figure.
```

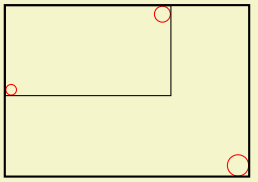
La deuxième application de cette option permet de gagner un meilleur contrôle de l'espace blanc laissé autour de la figure :

Gauche de la figure  droite de la figure.

```
Gauche de la figure
\begin{tikzpicture}
  \useasboundingbox (0,0) rectangle (3,1);
  \fill (.75,.25) circle (.5cm);
\end{tikzpicture}
droite de la figure.
```

Note : si on utilise cette option sur un chemin contenu dans un groupe TEXien (portée), l'effet ne dure que jusqu'à la fin de la portée. Là encore, c'est le comportement du découpage.

Il existe un nœud qui permet d'obtenir la taille de la boîte-cadre courante : `current bounding box` à une forme de `rectangle` et sa taille est toujours celle de la boîte-cadre courante.

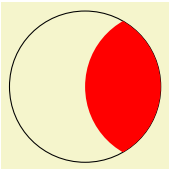


```
\begin{tikzpicture}
  \draw[red] (0,0) circle (2pt);
  \draw[red] (2,1) circle (3pt);

  \draw (current bounding box.south west) rectangle
        (current bounding box.north east);

  \draw[red] (3,-1) circle (4pt);

  \draw[thick] (current bounding box.south west) rectangle
               (current bounding box.north east);
\end{tikzpicture}
```

10.6 Utiliser un chemin pour découper

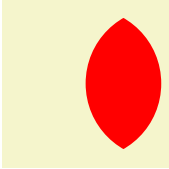
Pour utiliser un chemin pour découper, se servir de l'option `clip`.

– `clip` Cette option fait que tous les dessins suivants seront découpés le long du chemin courant et que leurs tailles ne seront pas prises en compte pour la taille de la figure. Si on découpe suivant un chemin autosécant la règle de parité ou celle de l'indice non-nul est utilisée pour déterminer si un point est intérieur ou extérieur à la région de découpage.
Le chemin de découpage est un paramètre graphique d'état aussi il sera remis à zéro à la fin de la portée courante. Des découpages multiples s'accumule, c'est-à-dire que le découpage est toujours fait suivant l'intersection de toutes les surfaces de découpage qui ont été spécifiées dans la portée courante. La seule manière d'agrandir une aire de découpage est de finir la portée `{scope}`.



```
\begin{tikzpicture}
  \draw[clip] (0,0) circle (1cm);
  \fill[red] (1,0) circle (1cm);
\end{tikzpicture}
```

C'est d'habitude une *très* bonne idée de n'appliquer l'option `clip` qu'à la première commande de chemin dans une portée.
Si l'on veut « simplement découper » sans vouloir tracer quoi que ce soit, on peut utiliser la commande `\clip` qui est une abréviation de `\path[clip]`.



```
\begin{tikzpicture}
  \clip (0,0) circle (1cm);
  \fill[red] (1,0) circle (1cm);
\end{tikzpicture}
```

Pour localiser le découpage, il faut utiliser des environnements `{scope}` comme dans l'exemple suivant :

```
\begin{tikzpicture}
  \draw (0,0) -- ( 0:1cm);
  \draw (0,0) -- (10:1cm);
  \draw (0,0) -- (20:1cm);
  \draw (0,0) -- (30:1cm);
  \begin{scope}[fill=red]
    \fill[clip] (0.2,0.2) rectangle (0.5,0.5);

    \draw (0,0) -- (40:1cm);
    \draw (0,0) -- (50:1cm);
    \draw (0,0) -- (60:1cm);
  \end{scope}
  \draw (0,0) -- (70:1cm);
  \draw (0,0) -- (80:1cm);
  \draw (0,0) -- (90:1cm);
\end{tikzpicture}
```

Il a un piège légèrement ennuyeux : on ne peut pas spécifier certaines options graphiques avec les commandes utilisées pour découper. Par exemple, dans le code ci-dessus, on n'aurait pas pu déplacer l'option `fill=red` vers la commande `\fill`. Les raisons ont à voir avec les spécifications internes de PDF. On ne cherchera pas à en connaître les détails. Le mieux est de simplement ne passer aucune option à ces commandes.

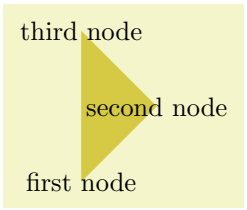
11 Nœuds

11.1 Les nœuds et leurs formes

Ti*k*Z offre une manière simple d'ajouter ce que l'on appelle des *nœuds* à une figure. Dans le cas le plus simple, un nœud est juste un peu de texte placé à une certaine coordonnée. Toutefois, un nœud peut également avoir une bordure ou un arrière-plan et un avant-plan plus compliqués. En fait, certains nœuds ne contiennent pas de texte du tout mais consiste seulement en un arrière-plan. On peut nommer les nœuds afin de pouvoir faire référence à leurs coordonnées plus tard dans la figure. Toutefois, *les nœuds ne peuvent être référencés d'une figure à une autre.*

Il n'y a pas de commandes TEXiennes spéciales pour ajouter un nœud à une figure, plutôt il y a une opération de chemin appelée `node` pour ce faire. Les nœuds sont créés à chaque fois que Ti*k*Z rencontre `node` ou `coordinate` à un point d'un chemin où il attend une opération normale de chemin (comme `-- (1,1)` ou `sin (1,1)`). On peut également donner des spécifications de nœud *à l'intérieur* de certaines opérations de chemin comme on l'explique plus loin.

L'opération de nœud est typiquement suivie par quelques options qui ne s'appliquent qu'au nœud. Puis on peut, de manière facultative, *nommer* le nœud en fournissant un nom entre parenthèses. Enfin, avec l'opération `node` on doit fournir au nœud un texte placé entre accolades alors que l'on peut ne pas en fournir avec `coordinate`. Le nœud est placé à la position courante du chemin *après que le chemin a été dessiné.* Ainsi, tous les nœuds sont dessinés « par dessus » le chemin et conservé jusqu'à ce que le chemin soit complet. S'il y a plusieurs nœuds sur un chemin, ils sont dessinés par dessus le chemin dans l'ordre dans lequel ils sont rencontrés.




```
\tikz \fill[fill=examplefill]
    (0,0) node {first node}
  -- (1,1) node {second node}
  -- (0,2) node {third node};
```

La syntaxe pour définir des nœuds est la suivante :

`\path` ... `node[`⟨*options*⟩`](`⟨*nom*⟩`)at(`⟨*coordonnée*⟩`){`⟨*texte*⟩`}` ... `;`

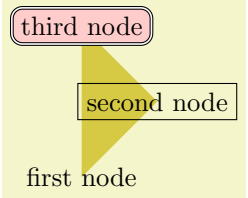
`at` a pour effet de placer le nœud à la coordonnée donnée après le `at` et non, comme c'est le cas normalement, à la dernière position. La syntaxe de `at` n'est pas disponible quand un nœud est donné à l'intérieur d'une opération de chemin (ça n'aurait aucun sens à cet endroit).

Le `(`⟨*nom*⟩`)` est un nom à utiliser pour des références futures et il est facultatif. On peut aussi ajouter l'option `name=`⟨*nom*⟩ à la liste des ⟨*options*⟩ ; l'effet est le même.

– `name=`⟨*nom de nœud*⟩ affecte un nom au nœud pour référence ultérieure. Comme c'est un nom de « haut niveau » (les pilotes n'en connaîtront jamais rien), on peut utiliser des espaces, des nombres, lettres ou tout autre signe dans le nom du nœud. Ainsi, on peut nommer un nœud simplement `1` ou peut-être `début de graphe` ou encore `y_1`. Le nom de nœud *ne doit pas* contenir de point, de virgule, de deux-point car ceux-ci servent à détecter le type de coordonnées utilisées quand on fait référence au nœud.

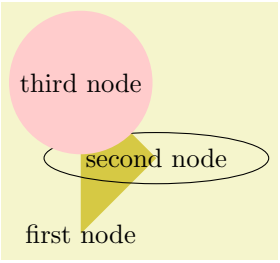
Les ⟨*options*⟩ sont données sous la forme d'une liste (facultative) d'options qui *ne s'appliquent qu'au nœud* et n'ont pas d'effet en dehors. D'un autre côté, la plupart des options « externes » s'appliquent aussi au nœud, mais pas toutes. Par exemple, la rotation « externe » ne s'applique pas au nœud (à moins que quelques options spéciales soient utilisées, [soupir]). De même, les actions de chemin externes, comme `draw` ou `fill`, ne s'appliquent jamais au nœud et doivent être données dans le nœud (à moins que certaines autres options spéciales ne soient utilisées [soupir profond]).

Comme signalé précédemment, on peut ajouter une bordure et même un arrière-plan à un nœud :




```
\tikz \fill[fill=examplefill]
      (0,0) node {first node}
   -- (1,1) node[draw] {second node}
   -- (0,2) node[fill=red!20,draw,double,rounded corners] {third node};
```

La « bordure » est en fait simplement un cas spécial d'un mécanisme beaucoup plus général. Chaque nœud a une certaine *forme* qui, par défaut, est un rectangle. Toutefois, on peut aussi demander à Ti*k*Z d'utiliser à la place une forme circulaire ou elliptique (il faut charger `pgflibraryshapes` pour cette dernière forme) :




```
\tikz \fill[fill=examplefill]
      (0,0) node{first node}
   -- (1,1) node[ellipse,draw] {second node}
   -- (0,2) node[circle,fill=red!20] {third node};
```

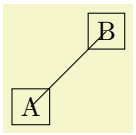
À l'avenir il se pourrait que d'autres formes bien plus compliquées soient disponibles comme, par exemple, une forme pour un résistor ou une forme pour un état d'un automate fini ou une forme pour une classe UML. Malheureusement créer de nouvelles formes et un peu embêtant et nécessite l'utilisation directe de la couche de base. La vie est une chienne !

Pour sélectionner la forme d'un nœud, on utilise l'option suivante :

– `shape=`⟨*nom de forme*⟩ sélectionne la forme soit pour le nœud courant soit, quand cette option n'est pas donnée dans un nœud mais quelque part en dehors, la forme de tous les nœuds situés dans la portée courante.
Comme cette option est souvent utilisée, on peut laisser le `shape=` de côté. Quand Ti*k*Z rencontre une option comme `circle` qu'il ne connaît pas, il essaiera, quand tout le reste aura échoué, de voir si cette option n'est pas le nom d'une forme. Si c'est le cas, il sélectionne cette forme comme si l'on avait écrit `shape=`⟨*nom de forme*⟩.
Par défaut, les formes suivantes sont disponibles : `rectangle`, `circle`, `coordinate` et, lorsque la bibliothèque `pgflibraryshapes` est chargée, `ellipse` également. Les détails sur ces formes, comme sur leurs ancres et options de taille, sont données dans la section 11.8, p. 95.

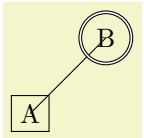
Les styles suivants influent sur le rendu des nœuds :

– `style=every node` Ce style est installé au début de chaque nœud.




```
\begin{tikzpicture}
  \tikzstyle{every node}=[draw]
  \draw (0,0) node {A} -- (1,1) node {B};
\end{tikzpicture}
```

– `style=every` ⟨*forme*⟩ `node` Ce style est installé au début de chaque nœud de la ⟨*forme*⟩ donnée. Par exemple, `every rectangle node` sert pour les nœuds rectangulaire, etc.




```
\begin{tikzpicture}
  \tikzstyle{every rectangle node}=[draw]
  \tikzstyle{every circle node}=   [draw,double]
  \draw (0,0) node[rectangle] {A} -- (1,1) node[circle] {B};
\end{tikzpicture}
```

Il y a une syntaxe spéciale pour spécifier des nœuds « poids plume » :

`\path` ... `coordinate[`⟨*options*⟩`](`⟨*nom*⟩`)at(`⟨*coordonnée*⟩`)` ... `;`

Qui a le même effet que

`node[shape=coordinate][`⟨*options*⟩`](`⟨*nom*⟩`)at(`⟨*coordonnée*⟩`){}`,

où la partie `at` peut manquer.

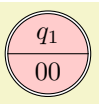
11.2 Nœuds à plusieurs parties

La plupart des nœuds n'ont qu'une simple étiquette textuelle. On peut, toutefois, créer des nœuds de formes plus complexes à partir de plusieurs *parties de nœud*. Par exemple, en théorie des automates, un état dit de Moore a un nom d'état, dessiné dans la partie supérieure du cercle d'état, et un texte de sortie, dessiné dans la partie inférieure du cercle d'état. Ces deux parties sont indépendantes. De même, une forme de classe UML aurait une partie pour le nom, une partie pour la méthode et une partie pour les attributs. Des formes moléculaires peuvent utiliser des parties pour les différents atomes à dessiner à des positions différentes, etc.

Tant PGF que Ti*k*Z gèrent de tels nœuds à plusieurs parties. Au niveau le plus bas, PGF fournit un système pour spécifier qu'une forme est constituée de plusieurs parties. Au niveau de Ti*k*Z, on définit les différentes parties de nœud à l'aide de la commande suivante :

`\nodepart{`⟨*nom de partie*⟩`}`

On ne peut utiliser cette commande que dans un argument ⟨*texte*⟩ d'une opération de chemin `node`. Elle fonctionne un peu comme une commande `\part` dans LaTeX. Elle arrête de typographier la partie du nœud qui était typographiée jusque là puis commence à placer tout le texte suivant dans une partie de nœud nommée ⟨*nom de partie*⟩ — jusqu'à une autre `\nodepart` ou jusqu'à la fin du ⟨*texte*⟩ du nœud.




```
\begin{tikzpicture}
  \node [state with output,draw,double,fill=red!20]
  {
    % Jusqu'ici, on n'a utilisé aucun \nodepart. Aussi ce qui suit est placé dans
    % la partie "texte"  de noeud par défaut
    $q_1$
    \nodepart{output} % Fini la partie "texte" et commence la partie "output"
    $00$
  }; % fin de la partie "output"
\end{tikzpicture}
```

On cherchera quelles parties sont définies par une forme.

Les styles suivants influent sur les parties de nœud :

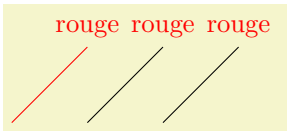
– `style=every ⟨nom de partie⟩ node part` Ce style est installé au début de chaque partie de nœud nommée ⟨*nom de partie*⟩


```
\tikzstyle{every output node part}=[red]
\tikz \node [state with output,draw] {$q_1$ \nodepart{output} $00$};
```

11.3 Options pour le texte dans les nœuds

L'option la plus simple pour le texte des nœuds concerne sa couleur. Normalement cette couleur est simplement la dernière couleur installée avec `color=`, éventuellement héritée d'une autre portée. Toutefois, on peut fixer spécialement la couleur utilisée pour le texte avec l'option suivante :

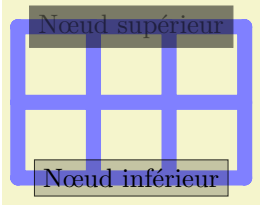
– `text=`⟨*couleur*⟩ Définit la couleur utilisée pour le texte des étiquettes. Une option `color=` annule immédiatement cette option.




```
\begin{tikzpicture}
  \draw[red]       (0,0) -- +(1,1) node[above]     {rouge};
  \draw[text=red]  (1,0) -- +(1,1) node[above]     {rouge};
  \draw            (2,0) -- +(1,1) node[above,red] {rouge};
\end{tikzpicture}
```

De même que la couleur elle-même, on peut vouloir déterminer l'opacité du seul texte. Pour cela, on utilisera l'option suivante :

– `text opacity=`⟨*valeur*⟩ Définit l'opacité du texte des étiquettes.




```
\begin{tikzpicture}
  \draw[line width=2mm,blue!50,cap=round] (0,0) grid (3,2);
  \tikzstyle{every node}=[fill,draw]

  \node[opacity=0.5] at (1.5,2) {N\oe ud supérieur};
  \node[draw opacity=0.8,fill opacity=0.2,text opacity=1]
    at (1.5,0) {N\oe ud inférieur};
\end{tikzpicture}
```

Ensuite, on peut vouloir ajuster la fonte utilisée pour le texte. On le fera avec l'option suivante :

– `font=`⟨*commandes de fontes*⟩ Définit la fonte utilisée pour le texte des étiquettes.


```
\begin{tikzpicture}
  \draw[font=\itshape] (1,0) -- +(1,1) node[above] {italique};
\end{tikzpicture}
```

Voici un exemple peut-être plus utile :


```
\tikzstyle{every text node part}=[font=\itshape]
\tikzstyle{every output node part}=[font=\footnotesize]
\tikzstyle{every state with output node}=[draw]
\tikz \node [state with output] {état \nodepart{output} sortie};
```

Normalement quand un nœud est typographié, tout le texte donné entre accolades n'est qu'une longue ligne (dans une `\hbox` précisément) et le nœud devient aussi large que nécessaire.

On peut changer ce comportement avec les options suivantes. Elles permettent de limiter la largeur du nœud (bien entendu au prix d'un changement de hauteur).

– `text width`=⟨*dimension*⟩ Cette option placera le texte du nœud dans une boîte de largeur ⟨*dimension*⟩ donnée (plus précisément, dans une `{minipage}` de cette largeur ; pour plain TEX on utilise une simulation rudimentaire de minipage).
Si le texte du nœud n'est pas aussi large que la ⟨*dimension*⟩, il sera néanmoins placé dans une boîte de cette largeur. S'il est plus large, la ligne sera coupée.
Par défaut, quand cette option est donnée, le texte est placé au fer à gauche (*ragged right*). Cela est judicieux car, généralement, ces boîtes sont étroites et la justification du texte produirait un aspect hideux.

Ceci est un texte de démonstration pour faire voir comment la coupure de ligne fonctionne.

```
\tikz \draw (0,0) node[fill=examplefill,text width=3cm]
  {Ceci est un texte de démonstration pour faire voir comment la coupure de ligne fonctionne.};
```

– `text justified` fait que le texte est justifié plutôt que placé au fer à gauche. À n'utiliser qu'avec des nœuds assez larges.

Ceci est un texte de démonstration pour faire voir comment la coupure de ligne fonctionne.

```
\tikz \draw (0,0) node[fill=examplefill,text width=3cm,text justified]
  {Ceci est un texte de démonstration pour faire voir comment la coupure de ligne fonctionne.};
```

Dans l'exemple précédent, TEX se plaint (à bon droit) de trois mauvaises coupures de lignes. (Pour ce manuel, j'ai demandé à TEX d'arrêter de se plaindre avec `\hbadness=10000`, mais c'est vraiment une action déloyale.)

– `text ragged` fait que le texte est serré au fer à gauche (*ragged right*) ; utilise la définition originelle donnée par plain TEX qui tente d'éviter les drapeaux trop flottants autant que faire se peut [18]. C'est le comportement par défaut.

This is a demonstration text for showing how line breaking works.

```
\tikz \draw (0,0) node[fill=examplefill,text width=3cm,text ragged]
  {This is a demonstration text for showing how line breaking works.};
```

– `text badly ragged` fait que le texte est serré au fer à gauche à la façon LATEXienne dans laquelle les coupures de mots sont inhibées. L'aspect en est horrible mais ce peut être utile avec des boîtes très étroites ou lorsque l'on veut éviter les coupures de mot.

This is a demonstration text for showing how line breaking works.

```
\tikz \draw (0,0) node[fill=examplefill,text width=3cm,text badly ragged]
  {This is a demonstration text for showing how line breaking works.};
```

18. NdTdS : Ce que l'on appelle « drapeau » c'est la ligne brisée, ici à droite du paragraphe, formée par les fins de lignes du dit paragraphe. Être « flottant » pour un drapeau, c'est présenter une ligne très accidentée.

– `text centered` centre le texte mais en cherchant à équilibrer les lignes.

This is a demonstration text for showing how line breaking works.

```
\tikz \draw (0,0) node[fill=examplefill,text width=3cm,text centered]
  {This is a demonstration text for showing how line breaking works.};
```

– `text badly centered` centre le texte sans chercher à équilibrer les lignes

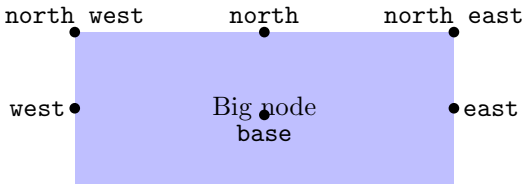
This is a demonstration text for showing how line breaking works.

```
\tikz \draw (0,0) node[fill=examplefill,text width=3cm,text badly centered]
  {This is a demonstration text for showing how line breaking works.};
```

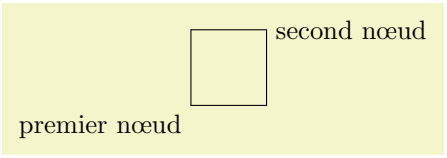
11.4 Placer des nœuds à l'aide d'ancres

Lorsque l'on place un nœud à une certaine coordonnée, celui-ci est, par défaut, centré sur cette coordonnée. Ce comportement est souvent indésirable et il vaudrait mieux que le nœud soit placé à droite ou au-dessus de la coordonnée.

PGF utilise un mécanisme dit d'ancrage pour permettre un contrôle très précis du placement. L'idée en est simple : imaginons le nœud comme une forme rectangulaire d'une certaine taille. PGF définit de nombreuses positions d'ancrage sur la forme. Par exemple, le coin supérieur droit est appelé, en fait pas « ancre haut droite » mais ancre `north east` (*nord-est*) de la forme. Au centre de la forme est placée une ancre appelée `center` (*centre*) . En voici quelques exemples (une liste complète est donnée à la section 11.8, p. 95).



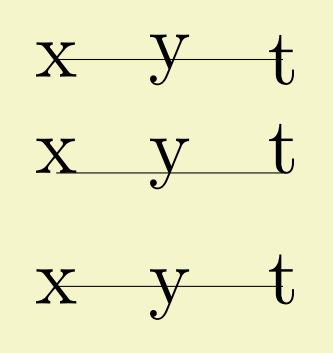
Maintenant, quand on a placé un nœud à une certaine coordonnée, on peut demander à Ti*k*Z de déplacer le nœud de telle sorte qu'une certaine ancre soit située à la coordonnée en question. Dans l'exemple suivant, on demande à Ti*k*Z de déplacer le premier nœud pour que son ancre `north east` soit à la coordonnée `(0,0)` et de placer l'ancre `west` du second nœud à la coordonnée `(1,1)`.




```
\tikz \draw          (0,0) node[anchor=north east] {premier n\oe ud}
           rectangle (1,1) node[anchor=west] {second n\oe ud};
```

Comme l'ancre par défaut est `center`, le comportement par défaut est de placer le nœud de telle façon qu'il soit centré sur la position courante.

– `anchor=`⟨*nom d'ancre*⟩ fait que le nœud est déplacé afin que l'ancre ⟨*nom d'ancre*⟩ soit placée sur la position courante.
La seule ancre présente dans tous les formes est `center`. Toutefois, la plupart des formes définissent au moins les ancres dans toutes « les directions de la boussole ». De plus, les formes de base définissent aussi une ancre `base` ainsi que `base west` et `base east` pour le placement de choses sur la ligne de base du texte.
Les formes *standard* définissent aussi une ancre `mid` (ainsi que `mid west` et `mid east`). Cette ancre est à une demie hauteur de « x » au-dessus la ligne de base. Cette ancre est utile pour centrer verticalement de multiples nœuds qui ont des hauteurs et profondeurs différentes. En voici un exemple :



```
\begin{tikzpicture}[scale=3,transform shape]
  % First, center alignment -> wobbles
  \draw[anchor=center] (0,1)  node{x} -- (0.5,1)  node{y} -- (1,1)  node{t};
  % Second, base alignment -> no wobble, but too high
  \draw[anchor=base]   (0,.5) node{x} -- (0.5,.5) node{y} -- (1,.5) node{t};
  % Third, mid alignment
  \draw[anchor=mid]    (0,0)  node{x} -- (0.5,0)  node{y} -- (1,0)  node{t};
\end{tikzpicture}
```

Malheureusement, bien que parfaitement logique, il est souvent plutôt contre-intuitif de devoir préciser l'ancre `south` (*sud*) pour placer un nœud *au-dessus* d'un point donné. Pour cette raison, il y a quelques options utiles qui permettent de sélectionner les ancres de façon plus intuitive :

- `above=`⟨*déport*⟩ fait comme `anchor=south`. Si le ⟨*déport*⟩ [19] est spécifié, le nœud est, de plus, déplacé vers le haut du ⟨*déport*⟩ donné.

above

```
\tikz \fill (0,0) circle (2pt) node[above] {above};
```

above

```
\tikz \fill (0,0) circle (2pt) node[above=2pt] {above};
```

- `above left=`⟨*déport*⟩ fait comme `anchor=south east`. Si le ⟨*déport*⟩ est spécifié, le nœud est, de plus, déplacé du ⟨*déport*⟩ vers le haut à gauche.

above left

```
\tikz \fill (0,0) circle (2pt) node[above left] {above left};
```

above left

```
\tikz \fill (0,0) circle (2pt) node[above left=2pt] {above left};
```

- `above right=`⟨*déport*⟩ fait comme `anchor=south west`.

above right

```
\tikz \fill (0,0) circle (2pt) node[above right] {above right};
```

- `left=`⟨*déport*⟩ fait comme `anchor=east`.

left

```
\tikz \fill (0,0) circle (2pt) node[left] {left};
```

- `right=`⟨*déport*⟩ fait comme `anchor=west`.
- `below=`⟨*déport*⟩ fait comme `anchor=north`.
- `below left=`⟨*déport*⟩ fait comme `anchor=north east`.
- `below right=`⟨*déport*⟩ fait comme `anchor=north west`.

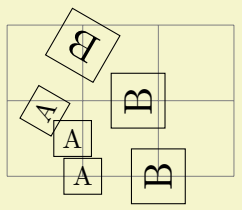
11.5 Transformations

On peut appliquer une transformation aux nœuds mais, par défaut, les transformations ne s'appliquent pas aux nœuds. La raison en est que, habituellement, on *ne veut pas* que le texte soit mis à l'échelle ou subisse une rotation même si le graphique principal est transformé. Mettre du texte à l'échelle est mal, le faire tourner un tout petit peu moins.

19. NdTdS : Déport traduit ici « offset ».

Toutefois, il arrive que l'on *veuille* transformer un nœud ; par exemple, tourner un nœud de 90 degrés est parfois sensé. On peut le faire de deux manières :

1. On peut se servir de l'option suivante :
   – `transform shape` fait que la matrice de transformation « externe » s'applique à la forme. Par exemple, si on écrit `\tikz[scale=3]` puis `node[transform shape] {X}`, on obtiendra un « énorme » X dans le graphique.
2. On peut donner l'option de transformation *à l'intérieur* de la liste d'options du nœud. *Ces* transformations-là s'appliquent toujours au nœud.



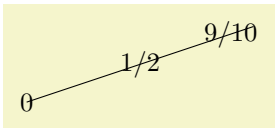

```
\begin{tikzpicture}
  \tikzstyle{every node}=[draw]
  \draw[style=help lines] (0,0) grid (3,2);
  \draw             (1,0) node{A}
                    (2,0) node[rotate=90,scale=1.5] {B};
  \draw[rotate=30] (1,0) node{A}
                    (2,0) node[rotate=90,scale=1.5] {B};
  \draw[rotate=60] (1,0) node[transform shape] {A}
                    (2,0) node[transform shape,rotate=90,scale=1.5] {B};
\end{tikzpicture}
```

11.6 Placer des nœuds sur une droite ou une courbe

Jusqu'à maintenant, on a toujours placé les nœuds sur des coordonnées écrites dans le chemin. Souvent, toutefois, on désire placer des nœuds au « milieu » d'une courbe et on ne veut pas calculer ces coordonnées « à la main ». Pour faciliter ces positionnements, Ti*k*Z permet de spécifier qu'un certain nœud doit se trouver quelque part « sur » une courbe. On peut le faire de deux manières : soit, explicitement, avec l'option `pos` soit, implicitement, en plaçant le nœud « à l'intérieur » d'une opération de chemin. On décrit ces deux façons dans ce qui suit.

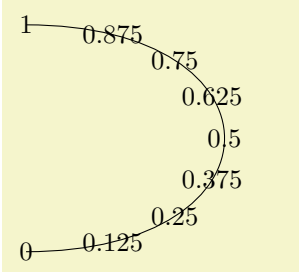
11.6.1 Utilisation explicite d'une option de position

– `pos`=⟨*fraction*⟩ Quand on donne cette option, le nœud n'est pas ancré sur la dernière coordonnée. Il est, plutôt, ancré à un point de la courbe situé quelque part par rapport a la dernière coordonnée. La ⟨*fraction*⟩ impose « l'éloignement » du point sur la courbe. Un ⟨*fraction*⟩ de 0 correspond à la coordonnée précédente, 1 au point courant, tout le reste est entre ces deux extrêmes. En particulier, 0,5 correspond au milieu.
Maintenant, qu'est-ce que la « courbe précédente » ? Cela dépend des opérations précédentes de construction de chemin.
Dans le cas le plus simple, l'opération précédente de chemin était une opération « ligne-jusqu'à », c'est-à-dire une opération `--`⟨*coordonnée*⟩ :



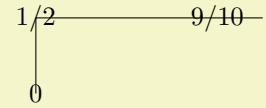

```
\tikz \draw (0,0) -- (3,1)
    node[pos=0]{0} node[pos=0.5]{1/2} node[pos=0.9]{9/10};
```

Le cas suivant est l'opération « courbe-jusqu'à » (l'opération `..`). Dans ce cas, le « milieu » de la courbe, c'est-à-dire, la position `0.5` n'est pas nécessairement un point à exactement la moitié de la distance suivant la courbe. Plutôt, c'est le lieu atteint au « temps » 0,5 par un point partant au temps 0 du début de la courbe et arrivant à la fin de la courbe au temps 1. La « vitesse » du point dépend de la norme des vecteurs supports (vecteurs qui joignent les points de début et de fin aux points de contrôle). Les mathématiques mises en jeu sont un peu complexes (suivant votre point de vue, bien entendu) ; si vous êtes intrigués, consultez un bon livre sur le dessin par ordinateur et les courbes de Bézier.

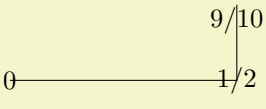



```
  \tikz \draw (0,0) .. controls +(right:3.5cm) and +(right:3.5cm) .. (0,3)
    \foreach \p in {0,0.125,...,1} {node[pos=\p]{\p}};
```

Un autre cas intéressant est fournit par les opérations « ligne horizontale/verticale jusqu'à » `|-` et `-|`. Pour elles, la position (ou temps) `0.5` est exactement le coin.



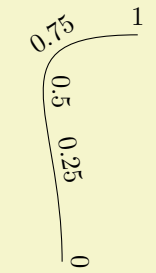

```
\tikz \draw (0,0) |- (3,1)
  node[pos=0]{0} node[pos=0.5]{1/2} node[pos=0.9]{9/10};
```



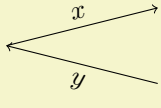

```
\tikz \draw (0,0) -| (3,1)
  node[pos=0]{0} node[pos=0.5]{1/2} node[pos=0.9]{9/10};
```

Pour toutes les autres opérations de construction de chemin, *le placement ne marche pas* à ce jour. On peut espérer que cela change à l'avenir (particulièrement pour les opérations sur les arcs).

– `sloped` Cette option fait que le nœud subi une rotation telle que l'horizontale soit placée sur la tangente à la courbe. La rotation sera toujours faite pour que le texte ne soit pas « tête en bas ». Si l'on tient vraiment à avoir du texte avec la tête en bas, on utilisera `[rotate=180]`.



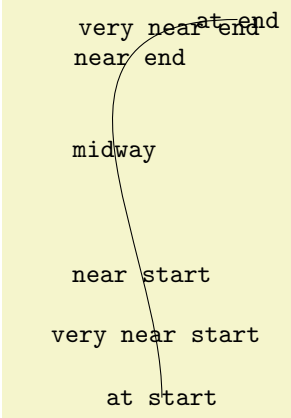

```
\tikz \draw (0,0) .. controls +(up:2cm) and +(left:2cm) .. (1,3)
    \foreach \p in {0,0.25,...,1} {node[sloped,above,pos=\p]{\p}};
```




```
\begin{tikzpicture}[->]
  \draw (0,0)   -- (2,0.5) node[midway,sloped,above] {$x$};
  \draw (2,-.5) -- (0,0)   node[midway,sloped,below] {$y$};
\end{tikzpicture}
```

Il existe des styles pour spécifier des positions d'une façon un peu moins « technique » :

– `style=midway` est défini par `pos=0.5` (*midway* signifie *à mi-chemin*).




```
\tikz \draw (0,0) .. controls +(up:2cm) and +(left:3cm) .. (1,5)
       node[at end]          {|at end|}
       node[very near end]   {|very near end|}
       node[near end]        {|near end|}
       node[midway]          {|midway|}
       node[near start]      {|near start|}
       node[very near start] {|very near start|}
       node[at start]        {|at start|};
```

– `style=near start` est défini par `pos=0.25`.
– `style=near end` est défini par `pos=0.75`.
– `style=very near start` est défini par `pos=0.125`.
– `style=very near end` est défini par `pos=0.875`.
– `style=at start` est défini par `pos=0`.
– `style=at end` est défini par `pos=1`.

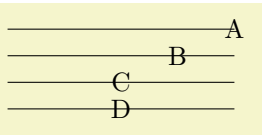
11.6.2 Utilisation implicite de l'option de position

Lorsque l'on veut placer un nœud sur la droite `(0,0) -- (1,1)`, il est naturel de ne pas spécifier de nœud après le `(1,1)` mais « quelque part au milieu ». C'est, en fait, possible et on peut écrire `(0,0) -- node{a} (1,1)` pour placer un nœud à mi-chemin entre `(0,0)` et `(1,1)`.

Voici ce qui se passe : la syntaxe de l'opération de chemin ligne-jusqu'à est en fait `--node`⟨*spécification de nœud*⟩⟨*coordonnée*⟩. (On peut même définir plusieurs nœuds de cette manière.) Lorsque l'option facultative `node` est rencontrée, la (ou les) spécification(s) est (sont) « stockées ». Puis, quand la ⟨*coordonnée*⟩ a été atteinte, la (les) spécification(s) est (sont) réinsérée(s) avec l'option `pos` activée.

On doit remarquer deux choses à ce propos : quand la spécification de nœud est « stockée », ses catcodes sont fixés. Cela signifie que l'on ne peut pas utiliser de *verbatim* excessivement complexe dans le texte. Si l'on a vraiment besoin de, par exemple, texte *verbatim*, il faudra le placer dans un nœud normal suivant la coordonnée et ajouter l'option `pos`.

Deuxièmement, quelle est la valeur de `pos` choisie pour le nœud ? La position est héritée de la portée enveloppante. Toutefois, cela n'est valide que pour les nœuds définis de cette manière implicite. Ainsi, si on ajoute l'option `[near end]` à une portée `{scope}`, cela n'entraînera pas que tous les nœuds définis dans cette portée seront placés près de la fin des courbes. Seuls les nœuds pour lesquels l'option `pos` est ajoutée implicitement seront placés près de la fin. C'est typiquement ce que l'on veut. Voici quelques exemples qui devraient clarifier tout cela :




```
\begin{tikzpicture}[near end]
  \draw (0cm,4em) -- (3cm,4em) node{A};
  \draw (0cm,3em) --           node{B}          (3cm,3em);
  \draw (0cm,2em) --           node[midway] {C} (3cm,2em);
  \draw (0cm,1em) -- (3cm,1em) node[midway] {D} ;
\end{tikzpicture}
```

De même que l'opération ligne-jusqu'à, l'opération courbe-jusqu'à `..` permet de spécifier des nœuds *dans* l'opération. On peut placer des spécifications de nœuds autant après le premier `..` que le second. Comme dans le cas de l'opération `--`, ces spécifications seront recueillies et réinsérées, l'option `pos` activée, après l'opération.

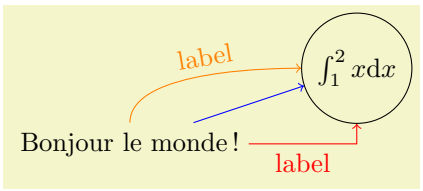
11.7 Relier des nœuds

Une fois un nœud défini et nommé, on peut en utiliser le nom pour y faire référence. Ce peut être fait de deux manières, voir également la section 8.5, p. 54. Supposons que l'on ait écrit `\path(0,0) node(x) {Bonjour le monde !}` pour définir le nœud nommé `x`.

1. Une fois le nœud `x` défini, on peut utiliser `(x.`⟨*ancre*⟩`)` partout où on pourrait utiliser une coordonnée normale. Cela produira la position à laquelle l'⟨*ancre*⟩ est située dans la figure. Remarquez que les transformations ne s'appliquent pas à cette coordonnée, c-à-d. que `(x.north)` sera l'ancre nord de `x` même si l'on écrit `scale=3` ou `xshift=4cm` par exemple. C'est en général ce que l'on attend.
2. On peut aussi utiliser simplement `x` comme une coordonnée. Dans la plupart des cas cela produira la même coordonnée que `(x.center)`. En fait, si la forme `shape` de `x` est `coordinate`, alors `(x)` et `(x.center)` ont le même effet.

   Toutefois, pour la plupart des autres formes, les opérations de construction de chemin comme `--` tentent d'être « futée » quand on leur demande de dessiner une courbe entre deux coordonnées de cette sorte. Lorsque l'on écrit `(x)--(1,1)`, l'opération de chemin `--` ne dessinera pas une droite depuis le centre de `x` mais *depuis la frontière* de `x` en allant du centre vers `(1,1)`. De même, `(1,1)--(x)` arrêtera le dessin de la droite sur la frontière de `x` en venant de `(1,1)`.

   Outre `--`, les opération de chemin `..`, `-|` et `|-` manipuleront correctement les nœuds sans ancres. Voici un exemple (voir aussi la section 8.5, p. 54) :



```
\begin{tikzpicture}
  \path (0,0) node             (x) {Bonjour le monde !}
        (3,1) node[circle,draw](y) {$\int_1^2 x \mathrm d x$};

  \draw[->,blue]   (x) -- (y);
  \draw[->,red]    (x) -| node[near start,below] {label} (y);
  \draw[->,orange] (x) .. controls +(up:1cm) and +(left:1cm) .. node[above,sloped] {label} (y);
\end{tikzpicture}
```

11.8 Formes prédéfinies

Par défaut, PGF et Ti*k*Z définissent trois formes :

- `rectangle` ;
- `circle`, et
- `coordinate`.

En chargeant des bibliothèques, on peut définir d'autres formes. À ce jour, l'extension `pgflibraryshapes` définit

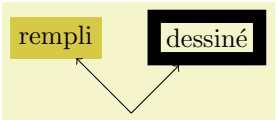
- `ellipse`.

Les comportements exacts de ces formes diffèrent. Les formes définies pour des usages plus particuliers (comme, par exemple, une forme de transistor) auront des comportements encore plus adaptés à leurs fonctions. Toutefois, un certain nombre d'options s'appliquent à la plupart des formes :

- `inner sep`=⟨*dimension*⟩ Un espace additionnel (invisible) de dimension ⟨*dimension*⟩ sera ajouté dans la forme, entre le texte et le chemin d'arrière-plan de la forme. L'effet est le même que si l'on avait ajouté des blancs horizontaux et verticaux idoines au début et à la fin du texte pour l'agrandir un peu. Par défaut, `inner sep` a la taille d'un espace normal.

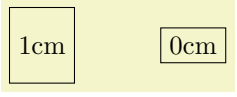

```
\begin{tikzpicture}
  \draw (0,0)     node[inner sep=0pt,draw] {serré}
        (0cm,2em) node[inner sep=5pt,draw] {relaché}
        (0cm,4em) node[fill=examplefill]   {par défaut};
\end{tikzpicture}
```

- `inner xsep`=⟨*dimension*⟩ détermine la séparation intérieure mais uniquement suivant l'axe des x.
- `inner ysep`=⟨*dimension*⟩ détermine la séparation intérieure mais uniquement suivant l'axe des y.
- `outer sep`=⟨*dimension*⟩ Cette option ajoute un espace additionnel (invisible) de dimension ⟨*dimension*⟩ à l'extérieur du chemin d'arrière-plan. L'effet principal de cette option est de déplacer un peu toutes les ancres « vers l'extérieur ».
  Par défaut, cette option a pour valeur une demie hauteur de ligne. Quand on utilise la valeur par défaut, et que le chemin d'arrière-plan est dessiné, les ancres seront placées exactement sur la « frontière externe » du chemin (pas sur le chemin lui-même). Quand la forme est remplie, mais pas dessinée, ce peut être indésirable. Dans ce cas, on devrait donner à `outer sep` la valeur 0.



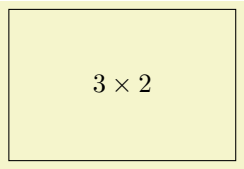

```
\begin{tikzpicture}
  \draw[line width=5pt]
    (0,0) node[outer sep=0pt,fill=examplefill]     (f) {rempli}
    (2,0) node[inner sep=.5\pgflinewidth+2pt,draw] (d) {dessiné};

  \draw[->] (1,-1) -- (f);
  \draw[->] (1,-1) -- (d);
\end{tikzpicture}
```

- `outer xsep`=⟨*dimension*⟩ Définit la séparation extérieure seulement suivant l'axe des x.
- `outer ysep`=⟨*dimension*⟩ Définit la séparation extérieure seulement suivant l'axe des y.
- `minimum height`=⟨*dimension*⟩ Cette option garantit que la hauteur de la forme (y compris la séparation intérieur mais sans la séparation extérieure) sera au moins égale à ⟨*dimension*⟩. Ainsi, si le texte plus la séparation intérieure n'est pas au moins aussi haute que ⟨*dimension*⟩, la forme sera convenablement agrandie. Toutefois, si le texte est déjà plus haut que ⟨*dimension*⟩, la forme ne sera pas réduite.



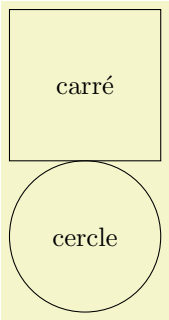

```
\begin{tikzpicture}
  \draw (0,0) node[minimum height=1cm,draw] {1cm}
        (2,0) node[minimum height=0cm,draw] {0cm};
\end{tikzpicture}
```

– `minimum width`=⟨*dimension*⟩ comme `minimum height` mais pour la largeur.



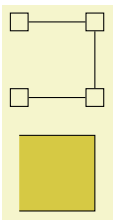

```
\begin{tikzpicture}
  \draw (0,0) node[minimum height=2cm,minimum width=3cm,draw] {$3 \times 2$};
\end{tikzpicture}
```

– `minimum size`=⟨*dimension*⟩ fixe, en une fois, les valeurs pour la hauteur et la largeur minimum.




```
\begin{tikzpicture}
  \draw (0,0)  node[minimum size=2cm,draw] {carré};
  \draw (0,-2) node[minimum size=2cm,draw,circle] {cercle};
\end{tikzpicture}
```

Ti*k*Z manipule la forme `coordinate` d'une façon spéciale. Lorsqu'un nœud `x`, dont la forme est `coordinate`, est utilisé comme coordonnée `(x)`, l'effet est le même que si l'on utilisait `(x.center)`. Aucune règle spéciale de « réduction des courbes » ne s'applique alors. Cela peut être utile puisque, normalement, la réduction des courbes fait que les chemins sont segmentés et ne peuvent, de ce fait, être rempli. Voici un exemple qui montre les différences :

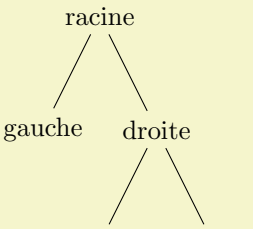


```
\begin{tikzpicture}
  \tikzstyle{every node}=[draw]
  \path[yshift=1.5cm,shape=rectangle]
    (0,0) node(a1){} (1,0) node(a2){}
    (1,1) node(a3){} (0,1) node(a4){};
  \filldraw[fill=examplefill] (a1) -- (a2) -- (a3) -- (a4);

  \path[shape=coordinate]
    (0,0) coordinate(b1) (1,0) coordinate(b2)
    (1,1) coordinate(b3) (0,1) coordinate(b4);
  \filldraw[fill=examplefill] (b1) -- (b2) -- (b3) -- (b4);
\end{tikzpicture}
```

12 Faire pousser les arbres

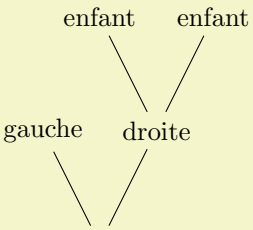
12.1 Introduction à l'opération child (enfant)

Les *arbres* fournissent une façon ordinaire de visualiser les structures hiérarchiques. Un arbre simple a l'aspect suivant :




```
\begin{tikzpicture}
  \node {racine}
    child {node {gauche}}
    child {node {droite}
      child {node {enfant}}
      child {node {enfant}}
    };
\end{tikzpicture}
```

Je reconnais que, dans la nature, les arbres ont plutôt tendance à grandir « vers le haut » et non pas vers le bas comme ci-dessus. On peut savoir si l'auteur d'un article est mathématicien ou informaticien rien qu'à regarder la direction dans laquelle ses arbres croissent. Les arbres d'un informaticien grandissent vers le bas, ceux d'un mathématicien vers le haut. Naturellement, la façon *correcte* est celle du mathématicien, que l'on peut obtenir comme suit :




```
\begin{tikzpicture}
  \node {racine} [grow'=up]
    child {node {gauche}}
    child {node {droite}
      child {node {enfant}}
      child {node {enfant}}
    };
\end{tikzpicture}
```

Dans Ti*k*Z on définit les arbres en ajoutant des *enfants* à un nœud situé sur un chemin à l'aide de l'opération `child` (*enfant*).

`\path` ... `child[`⟨*options*⟩`]foreach`⟨*variables*⟩`in{`⟨*valeurs*⟩`}{`⟨*chemin-enfant*⟩`}` ... ;

Cette opération devrait suivre directement une opération `node` complète ou une autre opération `child` quoiqu'il soit permit de faire précéder la première opération `child` d'options (ce que nous verrons plus loin).

Quand une opération `node` comme `node {X}` est suivie par `child`, Ti*k*Z commence à compter le nombre de nœuds-enfants qui suivent le nœud d'origine `node {X}`. Pour cela, il analyse l'entrée et stocke chacun des `child`s et leurs arguments jusqu'à ce qu'il rencontre une opération de chemin qui ne soit pas un `child`. Notez que cela fixe les codes de caractères (*catcode*) de tous les textes situés dans les arguments des enfants ; il en découle que, essentiellement, on ne peut pas utiliser de texte *verbatim* dans les nœuds intérieurs à un `child`. Désolé.

Quand les enfants ont été groupés et comptés, Ti*k*Z entame la création des nœuds-enfants. Pour chaque enfant d'un nœud parent, Ti*k*Z calcule une position appropriée pour placer l'enfant. Pour chaque enfant, il transforme le système de coordonnées de telle sorte que l'origine soit placée à cette position. Puis le ⟨*chemin-enfant*⟩ est dessiné. Typiquement, le chemin enfant est réduit à une spécification de nœud, ce qui entraîne le dessin du nœud à la position de l'enfant. Enfin, une arête est dessinée depuis le premier nœud du ⟨*chemin-enfant*⟩ au nœud parent.

La partie `foreach` facultative (notez qu'il n'y a pas de barre oblique inverse devant le `foreach`) permet de spécifier de multiples enfants dans une unique commande `child`. L'idée est la suivante : un énoncé `\foreach` est utilisé (en interne) pour itérer sur la liste de ⟨*valeurs*⟩. Pour chaque valeur de cette liste, on ajoute un nouveau `child` au nœud. La syntaxe pour les ⟨*variables*⟩ et les ⟨*valeurs*⟩ est la même que dans l'énoncé `\foreach`, voir la section **??**, p. **??**. Par exemple, lorsque l'on écrit :

```
node {root} child [red] foreach \name in {1,2} {node {\name}}
```

cela produit le même résultat que si l'on avait écrit

```
node {root} child[red] {node {1}} child[ref] {node {2}}
```

Quand on écrit

```
node {root} child[\pos] foreach \name/\pos in {1/left,2/right} {node[\pos] {\name}}
```

on obtiendra le même résultat qu'avec

```
node {root} child[left] {node[left] {1}} child[right] {node[right] {2}}
```

On peut emboîter les choses comme dans l'exemple suivant :

```
\begin{tikzpicture}[level distance=4mm]
  \tikzstyle{level 1}=[sibling distance=8mm]
  \tikzstyle{level 2}=[sibling distance=4mm]
  \tikzstyle{level 3}=[sibling distance=2mm]
  \coordinate
    child foreach \x in {0,1}
      {child foreach \y in {0,1}
        {child foreach \z in {0,1}}};
\end{tikzpicture}
```

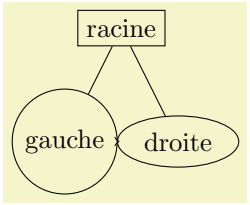
On décrit les détails et les options de cette opération dans la suite de la présente section.

12.2 Chemins-enfants et nœuds-enfants

Pour chaque `child` d'un nœud-racine, son ⟨*chemin-enfant*⟩ est inséré à une position spécifique de la figure (les règles de positionnement sont décrites dans la section 12.5, p. 100). Le premier nœud du ⟨*chemin-enfant*⟩, s'il existe, est spécial et appelé *nœud-enfant*. S'il n'y a pas de nœud dans le ⟨*chemin-enfant*⟩, c-à-d. si le ⟨*chemin-enfant*⟩ manque (y compris les accolades) ou s'il ne commence pas par `node` ou `coordinate`, alors on ajoute automatiquement un nœud-enfant vide dont la forme est `coordinate`.

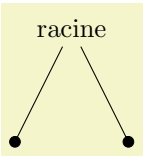
Regardons l'exemple `\node {x} child {node {y}} child;`. Le premier nœud-enfant présente un ⟨*chemin-enfant*⟩ qui possède un nœud-enfant `node {y}`. Pour le second enfant, aucun nœud-enfant n'est défini et, donc, c'est simplement `coordinate`.

De même que pour un nœud normal, on peut nommer un nœud-enfant, le déplacer ou utiliser des options pour influer sur son rendu.




```
\begin{tikzpicture}
  \node[rectangle,draw] {racine}
    child {node[circle,draw] (left node) {gauche}}
    child {node[ellipse,draw] (right node) {droite}};
  \draw[dashed,->] (left node) -- (right node);
\end{tikzpicture}
```

Dans de nombreux cas, le ⟨*chemin-enfant*⟩ ne contiendra qu'une spécification de nœud-enfant et, peut-être, des enfants de ce nœud-enfant. Toutefois, la spécification de nœud peut être suivie d'autres choses quelconques qui seront ajoutées à la figure, transformée dans le système de coordonnées de l'enfant. Pour vous faciliter la vie, une opération déplacer-jusqu'à `(0,0)` est insérée automatiquement au début du chemin. Voici un exemple :




```
\begin{tikzpicture}
  \node {racine}
    child {[fill] circle (2pt)}
    child {[fill] circle (2pt)};
\end{tikzpicture}
```

À la fin du ⟨*chemin-enfant*⟩ on peut ajouter une opération spéciale de chemin appelée `edge from parent`. Si l'on ne donne pas cette opération quelque part sur le chemin, elle sera ajoutée automatiquement à la fin. Cette option fait qu'une arête, reliant le nœud-parent et le nœud-enfant, est ajoutée au chemin. En donnant des options à cette opération on peut influer sur le rendu de cette arête. De plus, les nœuds suivants cette opération `edge from parent` seront placés sur l'arête, voir les détails dans la section 12.6, p. 103.

Pour résumer :

1. Le chemin-enfant commence avec une spécification de nœud. S'il n'y en a pas, elle est ajoutée automatiquement.

2. Le chemin-enfant se termine avec une opération `edge from parent`, peut-être suivie de nœuds à placer sur l'arête. Si on ne donne pas cette opération à la fin, elle est ajoutée automatiquement.

12.3 Nommer les nœuds-enfants

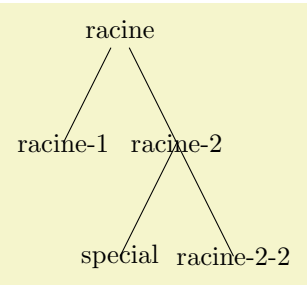
On peut nommer les nœuds-enfants comme tous les autres nœuds soit avec l'option `name` soit avec la syntaxe spéciale dans laquelle le nom du nœud est placé entre parenthèses entre l'opération `node` et le texte du nœud.

Si l'on ne nomme pas un nœud-enfant, Ti*k*Z le nommera automatiquement de la manière suivante : supposons que le nom du nœud-parent soit `parent`. (Si on ne nomme pas le parent, Ti*k*Z le fera lui même mais le nom ne sera pas accessible par l'utilisateur.) Le premier enfant de `parent` sera nommé `parent-1`, le deuxième enfant `parent-2` etc.

Cette convention de nommage fonctionne récursivement. Si le deuxième enfant `parent-2` a des enfants, alors le premier de ces enfants sera nommé `parent-2-1`, le deuxième `parent-2-2` etc.

Si l'on nomme soi-même un nœud-enfant, aucun nom n'est créé automatiquement (un nœud n'a pas deux noms). Toutefois, « le compte continue » ce qui signifie que le troisième enfant de `parent` et nommé `parent-3` indépendamment du fait que l'on ait ou pas nommé le premier ou le deuxième enfant de `parent`.

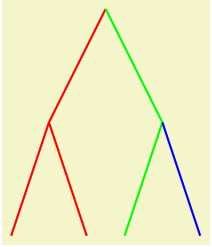
Voici un exemple :




```
\begin{tikzpicture}
  \node (root) {racine}
    child
    child {
      child {coordinate (special)}
      child
    };
  \node at (root-1) {racine-1};
  \node at (root-2) {racine-2};
  \node at (special) {special};
  \node at (root-2-2) {racine-2-2};
\end{tikzpicture}
```

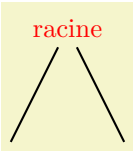
12.4 Spécifier des options pour les arbres et les enfants

Chaque `child` peut avoir ses propres ⟨*options*⟩ qui s'appliquent à « l'enfant tout entier » y compris à tous ses descendants. Voici un exemple :



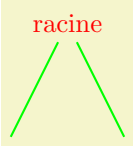
```
\begin{tikzpicture}[thick]
  \tikzstyle{level 2}=[sibling distance=10mm]
  \coordinate
    child[red]   {child child}
    child[green] {child child[blue]};
\end{tikzpicture}
```

Les options du nœud-racine n'ont aucun effet sur les enfants puisque les options d'un nœud sont toujours « locales » à ce nœud. C'est à cause de cela que les arêtes de l'arbre suivant sont noires et non rouges.




```
\begin{tikzpicture}[thick]
  \node [red] {racine}
    child
    child;
\end{tikzpicture}
```

Ce pose donc la question de passer des options à *tous* les enfants. Naturellement, on peut toujours passer les options à tout le chemin comme dans `\path [red] node {root} child child;` mais cela est ennuyeux dans certaines situations. Au lieu de cela, il est plus facile de passer le options *avant le premier enfant* comme dans ce qui suit :




```
\begin{tikzpicture}[thick]
  \node [red] {racine}
    [green] % cette option s'applique à tous les enfants
    child
    child;
\end{tikzpicture}
```

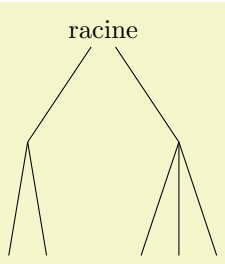
Voici l'ensemble des règles :

1. On donne les options pour tout l'arbre avant le nœud-racine.
2. On donne les options pour le nœud-racine directement à l'opération `node` de la racine.
3. On peut donner les options pour tous les enfants entre le nœud-racine et le premier enfant.
4. On donne les options s'appliquant à un nœud particulier comme options de l'opération `child`.
5. On donne les options qui s'appliquent au nœud d'un enfant mais pas à tout le chemin-enfant comme options de la commande `node` dans le ⟨*chemin-enfant*⟩.

```
\begin{tikzpicture}
  \path
    [...]             % Options pour l'arbre entier
    node[...] {racine}  % Options pour le seul n\oe ud racine
      [...]           % Options pour tous les enfants
      child[...]      % Options pour cet enfant et tous ses descendants
      {
        node[...] {}  % Options pour ce n\oe ud-enfant seulement
        ...
      }
      child[...]      % Options pour cet enfant et tous ses descendants
    ;
\end{tikzpicture}
```

Des styles supplémentaires influent sur le rendu des enfants :

- `style=every child` Ce style est utilisé au début de chaque enfant, comme si on passait les options à l'opération `child`.
- `style=every child node` Ce style est utilisé au début de chaque nœud-enfant en plus du style `every node`.
- `style=level` ⟨*nombre*⟩ Ce style est utilisé au début de chaque ensemble d'enfants pour lequel ⟨*nombre*⟩ est le niveau courant dans l'arbre courant. Par exemple, si on écrit `\node {x} child child;`, alors le style `level 1` est utilisé avant le premier `child`. Si ce premier enfant a lui-même des enfants, alors le `level 2` sera utilisé pour eux.




```
\begin{tikzpicture}
  \tikzstyle{level 1}=[sibling distance=20mm]
  \tikzstyle{level 2}=[sibling distance=5mm]
  \node {racine}
    child { child child }
    child { child child child };
\end{tikzpicture}
```

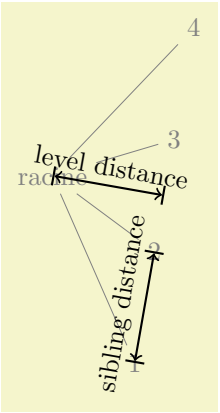
12.5 Placer les nœuds-enfants

La partie la plus difficile du dessin d'un arbre est, peut-être, le placement correct des enfants. Typiquement, les enfants ont des tailles différentes et il n'est pas facile de les placer de manière à limiter le gaspillage d'espace, de manière à ce que leurs enfants ne se recouvrent pas et de faire qu'ils soient placés régulièrement ou bien que leurs centres soient placés régulièrement. Calculer de bonnes positions est particulièrement difficile puisque la bonne position du premier enfant peut dépendre de la taille du dernier enfant.

On choisit dans Ti*k*Z une approche comparativement simple pour placer les enfants. Afin de calculer la position d'un enfant, on ne considère que le rang de l'enfant courant dans la liste des enfants et le nombre d'enfants dans cette liste. Ainsi, si un nœud a cinq enfants, le premier reçoit une position fixée, le deuxième également, etc. Ces positions *ne dépendent pas de la taille des enfants* et, donc, les enfants peuvent facilement se recouvrir. Toutefois, comme on peut utiliser des options pour déplacer un peu chaque enfant individuellement, ce problème n'est pas aussi difficile qu'il pourrait sembler.

Bien que le placement des enfants ne dépende que de leur rang dans la liste des enfants et du nombre total d'enfants, tout le reste, concernant le placement, est largement configurable. On peut changer la distance entre enfants (appelé judicieusement `sibling distance` *distance entre frères*) et la distance entre les niveaux de l'arbre. Ces distances peuvent être modifiées à chaque niveau. On peut changer la direction de croissance de l'arbre globalement et aussi pour chaque partie de l'arbre. On point même définir sa propre « fonction de croissance » pour placer les enfants sur un cercle ou le long d'une courbe quelconque.

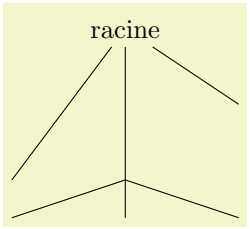
La fonction de croissance par défaut marche comme ceci : supposons donnés un nœud et cinq enfants. Ces enfants sont placés sur une droite de telle façon que leurs centres (ou, plus généralement, leurs ancres) soient séparés pour la `sibling distance` courante. La droite est perpendiculaire à la *direction de croissance* courante, direction que l'on fixe à l'aide de l'option `grow` ou `grow'` (cette dernière renverse l'ordre des enfants). La distance entre la droite du nœud-parent et celle des nœuds-enfants est donnée par `level distance`.



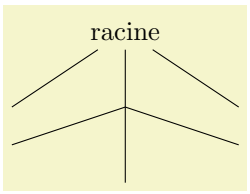

```
\begin{tikzpicture}
  \path [help lines]
    node (root) {racine}
    [grow=-10]
    child {node {1}}
    child {node {2}}
    child {node {3}}
    child {node {4}};

  \draw[|<->|,thick] (root-1.center)
    -- node[above,sloped] {sibling distance} (root-2.center);

  \draw[|<->|,thick] (root.center)
    -- node[above,sloped] {level distance} +(-10:\tikzleveldistance);
\end{tikzpicture}
```

Voici une description détaillée des options :

– `level distance`=⟨*distance*⟩ Cette option permet de modifier la distance entre différents niveaux de l'arbre, plus précisément, entre le parent et la courbe sur laquelle ses enfants sont placés. Quand l'option est passée à un seul enfant, elle ne fixe la distance que pour cet enfant-là.



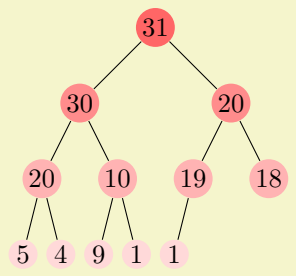

```
\begin{tikzpicture}
  \node {racine}
    [level distance=20mm]
    child
    child {
      [level distance=5mm]
      child
      child
      child
    }
    child[level distance=10mm];
\end{tikzpicture}
```




```
\begin{tikzpicture}
  \tikzstyle{level 1}=[level distance=10mm]
  \tikzstyle{level 2}=[level distance=5mm]
  \node {racine}
    child
    child {
      child
      child[level distance=10mm]
      child
    }
    child;
\end{tikzpicture}
```

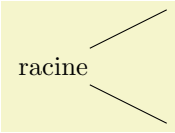
– `sibling distance`=⟨*distance*⟩ Cette option définit la distance entre les ancres des enfants d'un nœud-parent.

```
\begin{tikzpicture}[level distance=4mm]
  \tikzstyle{level 1}=[sibling distance=8mm]
  \tikzstyle{level 2}=[sibling distance=4mm]
  \tikzstyle{level 3}=[sibling distance=2mm]
  \coordinate
     child {
       child {child child}
       child {child child}
     }
     child {
       child {child child}
       child {child child}
     };
\end{tikzpicture}
```

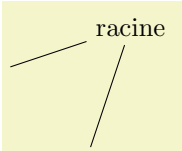



```
\begin{tikzpicture}[level distance=10mm]
  \tikzstyle{every node}=[fill=red!60,circle,inner sep=1pt]
  \tikzstyle{level 1}=[sibling distance=20mm,
    set style={{every node}+=[fill=red!45]}]
  \tikzstyle{level 2}=[sibling distance=10mm,
    set style={{every node}+=[fill=red!30]}]
  \tikzstyle{level 3}=[sibling distance=5mm,
    set style={{every node}+=[fill=red!15]}]
  \node {31}
     child {node {30}
       child {node {20}
         child {node {5}}
         child {node {4}}
       }
       child {node {10}
         child {node {9}}
         child {node {1}}
       }
     }
     child {node {20}
       child {node {19}
         child {node {1}}
         child[fill=none] {edge from parent[draw=none]}
       }
       child {node {18}}
     };
\end{tikzpicture}
```

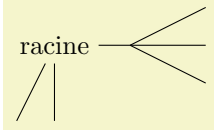
– grow=⟨*direction*⟩ On utilise cette option pour définir la ⟨*direction*⟩ dans laquelle l'arbre grandira. La ⟨*direction*⟩ peut être soit un angle en degrés soit l'une des chaînes de caractères suivantes : `down` (*bas*), `up` (*haut*), `left` (*gauche*), `right` (*droite*), `north` (*nord*), `south` (*sud*), `east` (*est*), `west` (*ouest*), `north east` (*nord-est*), `north west` (*nord-ouest*), `south east` (*sud-est*) et `south west` (*sud-ouest*). Toutes ont leur « signification évidente » et, donc, `south west` est pareil que l'angle $-135°$.
Cette option a pour effet secondaire d'installer la fonction de croissance par défaut.
En plus de fixer la direction, cette option a aussi, ce qui peut paraître étrange, pour effet de fixer la distance entre frères du niveau courant à `0pt` mais laisse inchangée la distance entre frères des niveaux suivants.
Ce comportement quelque peu étrange a un effet très souhaitable : si l'on donne cette option avant le début de la liste des enfants d'un nœud, le « niveau courant » est encore le niveau parent. Chaque enfant sera sur le niveau suivant et, donc, la distance entre frères sera celle définie à l'origine. Cela fait que les enfants sont nettement alignés sur une perpendiculaire à la ⟨*direction*⟩ donnée. Toutefois, si l'on donne l'option localement à un seul enfant, alors le « niveau courant » est le niveau de cet enfant. Comme la distance entre frère est nulle, l'enfant est placé exactement à un point distant de `level distance` dans la ⟨*direction*⟩. Toutefois, les enfants de cet enfant seront placés normalement sur une perpendiculaire à la ⟨*direction*⟩.
Quelques exemples démontrent mieux les effets de ces placements :



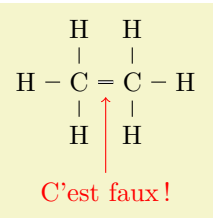

```
\tikz \node {racine} [grow=right] child child;
```



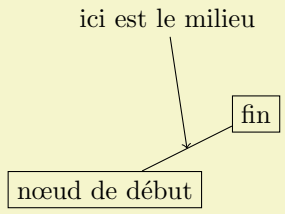

```
\tikz \node {racine} [grow=south west] child child;
```




```
\begin{tikzpicture}[level distance=10mm,sibling distance=5mm]
  \node {racine}
    [grow=down]
    child
    child
    child[grow=right] {
      child child child
    };
\end{tikzpicture}
```




```
\begin{tikzpicture}[level distance=2em]
  \node {C}
    child[grow=up]    {node {H}}
    child[grow=left]  {node {H}}
    child[grow=down]  {node {H}}
    child[grow=right] {node {C}
        child[grow=up]    {node {H}}
        child[grow=right] {node {H}}
        child[grow=down]  {node {H}}
      edge from parent[double]
        coordinate (wrong)
    };
  \draw[<-,red] ([yshift=-2mm]wrong) -- +(0,-1)
    node[below]{C'est faux !};
\end{tikzpicture}
```




```
\begin{tikzpicture}
  \node[rectangle,draw] (a) at (0,0) {n\oe ud de début};
  \node[rectangle,draw] (b) at (2,1) {fin};

  \draw (a) -- (b)
    node[coordinate,midway] {}
      child[grow=100,<-] {node[above] {ici est le milieu}};
\end{tikzpicture}
```

– grow'=⟨*direction*⟩ Cette option a le même effet que `grow` mais les enfants sont rangés dans l'ordre inverse.
– growth function=⟨*nom de macro*⟩ Cette option, plutôt de bas niveau, permet de définir une nouvelle fonction de croissance. Le ⟨*nom de macro*⟩ doit être le nom d'une macro sans paramètre. Cette macro sera appelée pour chaque enfant du nœud.
Le développement de cette macro devrait avoir l'effet suivant : transformer le système de coordonnées de telle sorte que l'origine devienne le lieu où l'enfant courant doit être ancré. Lorsque la macro est appelée, le système de coordonnées courant est fixé de sorte que l'origine soit placée à l'ancre du nœud-parent. Donc, à chaque appel, l'effet de ⟨*nom de macro*⟩ est essentiellement une translation de l'origine. Quand la macro est appelée, le compteur TEXien `\tikznumberofchildren` prend la valeur du nombre total d'enfants du nœud-parent et le compteur `\tikznumberofcurrentchild` prend la valeur du rang de l'enfant courant.
En plus de translater le système de coordonnées, la macro peut le transformer plus profondément. Elle peut, par exemple, lui faire subir une rotation ou une homothétie (mise à l'échelle).
Une bibliothèque définit des fonctions de croissances supplémentaires, voir la section **??**, p. **??**.

12.6 Arêtes issues du nœud-parent

Chaque nœud-enfant est relié à son nœud-parent par une sorte spéciale d'arête appelée `edge from parent` (*arête issue du parent*). Cette arête est ajoutée au ⟨*chemin-enfant*⟩ quand l'opération suivante de chemin est rencontrée :

`\path` ... `edge from parent[`⟨*options*⟩`]` ... ;

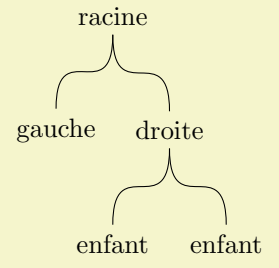
Cette opération de chemin ne peut être utilisée qu'à l'intérieur d'un ⟨*chemin-enfant*⟩ et devrait être donnée à la fin, éventuellement suivie par des spécifications de nœud (nous verrons cela plus loin). Si un ⟨*chemin-enfant*⟩ ne contient pas cette opération, elle est ajoutée automatiquement à sa fin.

Cette opération a plusieurs effets. Le plus important est qu'il insère « l'arête issue du chemin-parent » dans le chemin-enfant. L'arête issue du chemin-parent peut être modifiée à l'aide de l'option suivante :
– edge from parent path=⟨*chemin*⟩ Cette option permet de définir le nouveau chemin donnant la forme de l'arête issue du chemin-parent. Par défaut on a :

```
(\tikzparentnode\tikzparentanchor) -- (\tikzchildnode\tikzchildanchor)
```

`\tikzparentnode` est une macro qui sera développée en le nom du nœud-parent. Cela marche même quand on n'a pas nommé le nœud-parent car dans ce cas un nom interne est créé automatiquement. `\tikzchildnode` est une macro qui se développe en le nom du nœud-enfant. Donc, ce qui est inséré essentiellement n'est que le segment de chemin `(\tikzparentnode) -- (\tikzchildnode)` ; c'est exactement une arête reliant le parent à l'enfant.

On peut modifier cette arête issue du chemin-parent pour obtenir toutes sortes d'effets. Par exemple, on peut remplacer la droite par une courbe comme ceci :

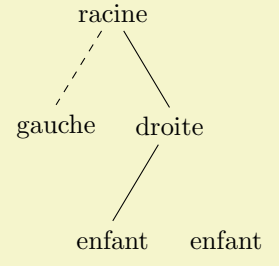



```
\begin{tikzpicture}[edge from parent path=
  {(\tikzparentnode.south) .. controls +(0,-1) and +(0,1)
                           .. (\tikzchildnode.north)}]
  \node {racine}
    child {node {gauche}}
    child {node {droite}
      child {node {enfant}}
      child {node {enfant}}
    };
\end{tikzpicture}
```

D'autres arêtes issues du chemin-parent sont définies dans la bibliothèque d'arbres, voir la section **??**, p. **??**.

Comme signalé précédemment, les ancres de l'arête issue du chemin-parent par défaut sont vides. Toutefois, on peut les définir avec les options suivantes :

– `child anchor=`⟨*ancre*⟩ Définit l'ancre où l'arête issue du chemin-parent rencontre le nœud-enfant en donnant à `\tikzchildanchor` la valeur `.`⟨*ancre*⟩.
Si on spécifie `border` (*frontière*) comme ⟨*ancre*⟩, alors la macro `\tikzchildanchor` prend pour valeur une chaîne vide. L'effet de tout cela est que l'arête issue du parent atteindra l'enfant sur la frontière, à un lieu déterminé automatiquement.

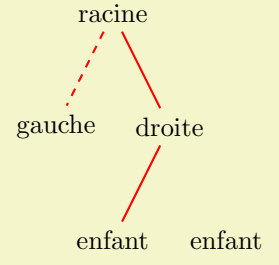



```
\begin{tikzpicture}
  \node {racine}
    [child anchor=north]
    child {node {gauche} edge from parent[dashed]}
    child {node {droite}
      child {node {enfant}}
      child {node {enfant} edge from parent[draw=none]}
    };
\end{tikzpicture}
```

– `parent anchor=`⟨*ancre*⟩ Cette option fonctionne de la même manière que `child anchor` mais, cette fois, pour le parent.

En plus de l'insertion de l'arête issue du chemin-parent, l'opération `edge from parent` a un autre effet : les ⟨*options*⟩ sont insérées directement avant l'arête issue du chemin-parent et le style suivant est installé avant l'insertion du chemin :

– `style=edge from parent` Ce style est inséré juste avant l'arête issue du chemin-parent et avant que les ⟨*options*⟩ soient insérées. Par défaut, il se contente de dessiner l'arête issue du parent mais on peut s'en servir pour modifier l'aspect de l'arête.




```
\begin{tikzpicture}
  \tikzstyle{edge from parent}=[draw,red,thick]
  \node {racine}
    child {node {gauche} edge from parent[dashed]}
    child {node {droite}
      child {node {enfant}}
      child {node {enfant} edge from parent[draw=none]}
    };
\end{tikzpicture}
```

Note : les ⟨*options*⟩ insérées avant que soit ajoutée l'arête issue du chemin-parent *s'appliquent à tout le chemin-enfant*. Aussi, on ne peut par, par exemple, dessiner un cercle en rouge comme partie d'un chemin-enfant et obtenir une arête bleue. Toutefois, comme toujours, le nœud-enfant est un nœud et peut être dessiné d'une façon complètement différente.

Enfin, l'opération `edge from parent` a encore un effet : elle force le placement de tous les nœuds *suivant* l'opération sur l'arête. Cet effet est celui que l'on aurait si l'on ajoutait l'option `pos` à tous ces nœuds, voir la section 11.6.1, p. 92.

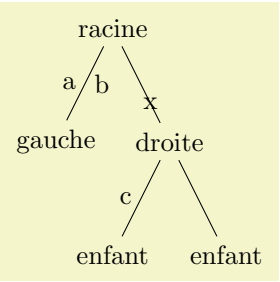
En guise d'exemple, regardons le code suivant :

```
\node (root) {} child {node (child) {} edge to parent node {label}};
```

L'opération `edge to parent` et l'opération `node` subséquente auront, ensemble, le même effet que si l'on avait écrit :

```
(root) -- (child) node [pos=0.5] {label}
```

Voici un exemple plus complexe :



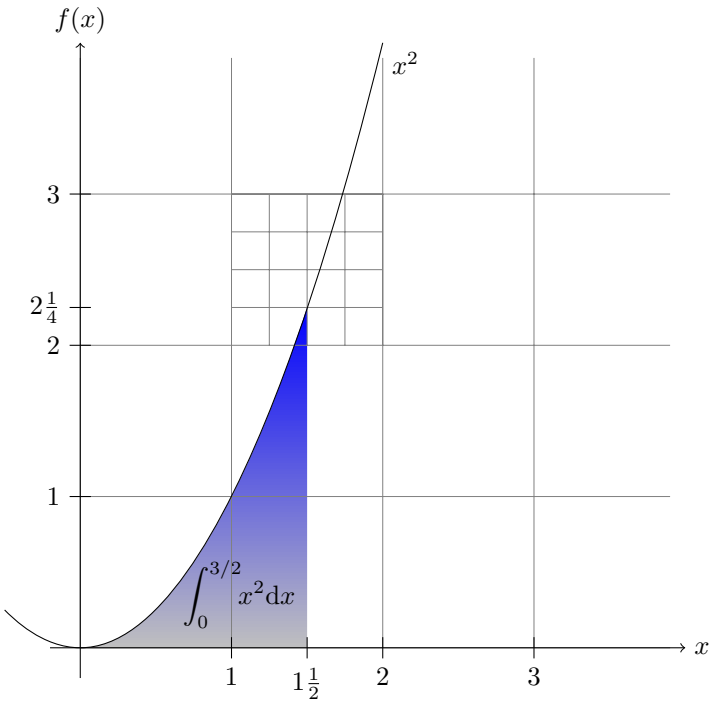

```
\begin{tikzpicture}
  \node {racine}
    child {
      node {gauche}
      edge from parent
        node[left] {a}
        node[right] {b}
    }
    child {
      node {droite}
        child {
          node {enfant}
          edge from parent
            node[left] {c}
        }
        child {node {enfant}}
      edge from parent
        node[near end] {x}
    };
\end{tikzpicture}
```

Troisième partie

Bibliothèques et Utilitaires

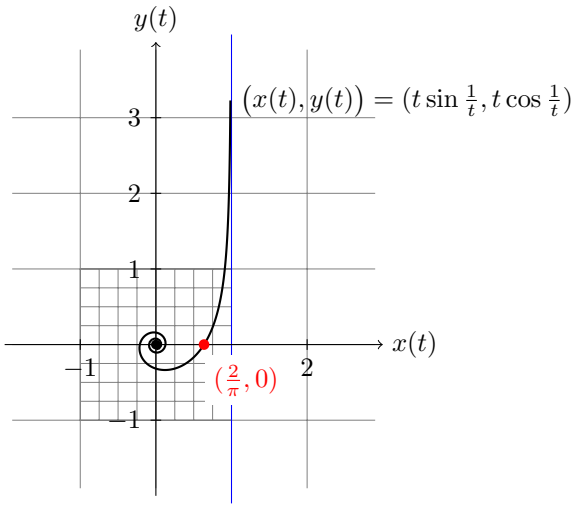
Dans cette partie on documente les extensions (*package*) bibliothèques et utilitaires. Les extensions bibliothèques fournissent des objets graphiques prédéfinis supplémentaires comme de nouvelles têtes de flèches ou de nouvelles marques pour les courbes. Elles ne sont pas chargées par défaut puisque de nombreux utilisateurs n'en ont pas besoin.

Les extensions d'utilitaires ne sont pas directement impliquées dans la création de graphiques mais vous pourriez néanmoins les trouver utiles. Toutes dépendent directement de PGF ou sont conçues pour fonctionner correctement avec PGF même si elles peuvent être utilisées seules.




```
\begin{tikzpicture}[scale=2]
  \shade[top color=blue,bottom color=gray!50] (0,0) parabola (1.5,2.25) |- (0,0);
  \draw (1.05cm,2pt) node[above] {$\displaystyle\int_0^{3/2} \!\!x^2\mathrm{d}x$};

  \draw[style=help lines] (0,0) grid (3.9,3.9)
       [step=0.25cm]      (1,2) grid +(1,1);

  \draw[->] (-0.2,0) -- (4,0) node[right] {$x$};
  \draw[->] (0,-0.2) -- (0,4) node[above] {$f(x)$};

  \foreach \x/\xtext in {1/1, 1.5/1\frac{1}{2}, 2/2, 3/3}
    \draw[shift={(\x,0)}] (0pt,2pt) -- (0pt,-2pt) node[below] {$\xtext$};

  \foreach \y/\ytext in {1/1, 2/2, 2.25/2\frac{1}{4}, 3/3}
    \draw[shift={(0,\y)}] (2pt,0pt) -- (-2pt,0pt) node[left] {$\ytext$};

  \draw (-.5,.25) parabola bend (0,0) (2,4) node[below right] {$x^2$};
\end{tikzpicture}
```

Quatrième partie
La couche de base



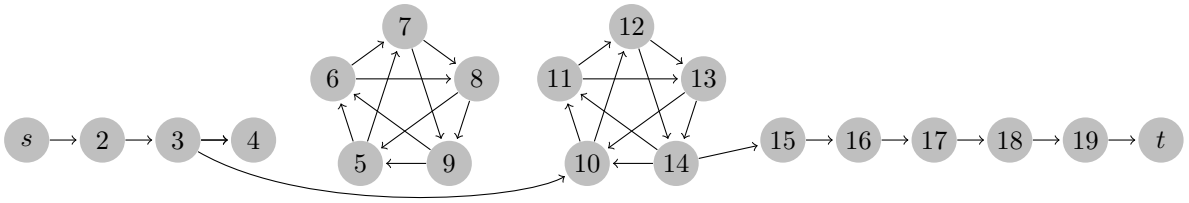

```
\begin{tikzpicture}
  \draw[gray,very thin] (-1.9,-1.9) grid (2.9,3.9)
          [step=0.25cm] (-1,-1) grid (1,1);
  \draw[blue] (1,-2.1) -- (1,4.1); % asymptote

  \draw[->] (-2,0) -- (3,0) node[right] {$x(t)$};
  \draw[->] (0,-2) -- (0,4) node[above] {$y(t)$};

  \foreach \pos in {-1,2}
    \draw[shift={(\pos,0)}] (0pt,2pt) -- (0pt,-2pt) node[below] {$\pos$};

  \foreach \pos in {-1,1,2,3}
    \draw[shift={(0,\pos)}] (2pt,0pt) -- (-2pt,0pt) node[left] {$\pos$};

  \fill (0,0) circle (0.064cm);
  \draw[thick,parametric,domain=0.4:1.5,samples=200]
    % The plot is reparameterised such that there are more samples
    % near the center.
    plot[id=asymptotic-example] function{(t*t*t)*sin(1/(t*t*t)),(t*t*t)*cos(1/(t*t*t))}
    node[right] {$\bigl(x(t),y(t)\bigr) = (t\sin \frac{1}{t}, t\cos \frac{1}{t})$};

  \fill[red] (0.63662,0) circle (2pt)
    node [below right,fill=white,yshift=-4pt] {$(\frac{2}{\pi},0)$};
\end{tikzpicture}
```

Cinquième partie
La couche système

Cette partie décrit l'interface de bas niveau de PGF, dénommée la *couche système*. Cette interface fournit une abstraction complète des mécanismes internes des pilotes sous-jacents.

À moins que vous ne vouliez créer un autre pilote pour PGF ou que vous désiriez écrire votre propre interface optimisée, vous n'avez pas besoin de lire cette partie.

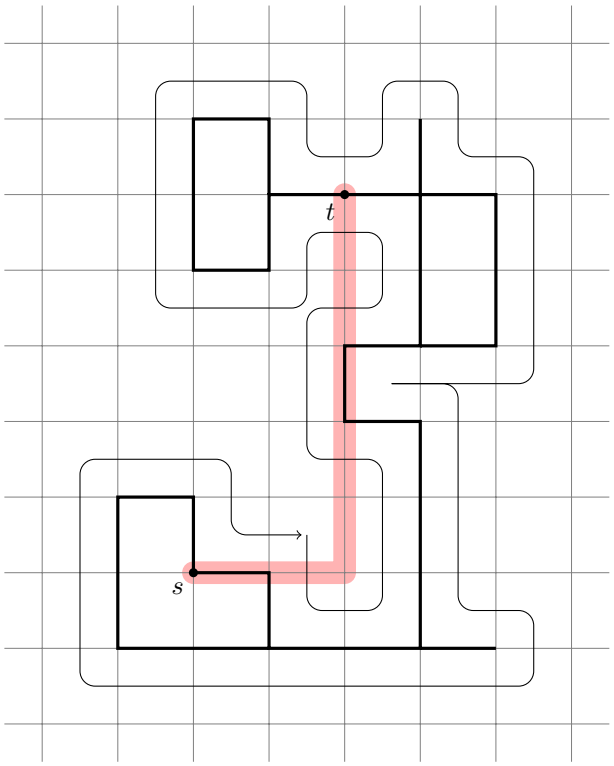
Dans la suite on considère que vous êtes familier de la manière dont fonctionne l'extension `graphics` et que vous connaissez ce que sont les pilotes TEX et comment ils travaillent.




```
\begin{tikzpicture}[shorten >=1pt,->]
  \tikzstyle{vertex}=[circle,fill=black!25,minimum size=17pt,inner sep=0pt]

  \foreach \name/\x in {s/1, 2/2, 3/3, 4/4, 15/11, 16/12, 17/13, 18/14, 19/15, t/16}
    \node[vertex] (G-\name) at (\x,0) {$\name$};

  \foreach \name/\angle/\text in {P-1/234/5, P-2/162/6, P-3/90/7, P-4/18/8, P-5/-54/9}
    \node[vertex,xshift=6cm,yshift=.5cm] (\name) at (\angle:1cm) {$\text$};

  \foreach \name/\angle/\text in {Q-1/234/10, Q-2/162/11, Q-3/90/12, Q-4/18/13, Q-5/-54/14}
    \node[vertex,xshift=9cm,yshift=.5cm] (\name) at (\angle:1cm) {$\text$};

  \foreach \from/\to in {s/2,2/3,3/4,3/4,15/16,16/17,17/18,18/19,19/t}
    \draw (G-\from) -- (G-\to);

  \foreach \from/\to in {1/2,2/3,3/4,4/5,5/1,1/3,2/4,3/5,4/1,5/2}
    { \draw (P-\from) -- (P-\to); \draw (Q-\from) -- (Q-\to); }

  \draw (G-3) .. controls +(-30:2cm) and +(-150:1cm) .. (Q-1);
  \draw (Q-5) -- (G-15);
\end{tikzpicture}
```

Sixième partie

Références et Index



```
\begin{tikzpicture}
  \draw[line width=0.3cm,color=red!30,cap=round,join=round] (0,0)--(2,0)--(2,5);
  \draw[help lines] (-2.5,-2.5) grid (5.5,7.5);
  \draw[very thick] (1,-1)--(-1,-1)--(-1,1)--(0,1)--(0,0)--
    (1,0)--(1,-1)--(3,-1)--(3,2)--(2,2)--(2,3)--(3,3)--
    (3,5)--(1,5)--(1,4)--(0,4)--(0,6)--(1,6)--(1,5)
    (3,3)--(4,3)--(4,5)--(3,5)--(3,6)
    (3,-1)--(4,-1);
  \draw[below left] (0,0) node(s){$s$};
  \draw[below left] (2,5) node(t){$t$};
  \fill (0,0) circle (0.06cm) (2,5) circle (0.06cm);
  \draw[->,rounded corners=0.2cm,shorten >=2pt]
    (1.5,0.5)-- ++(0,-1)-- ++(1,0)-- ++(0,2)-- ++(-1,0)-- ++(0,2)-- ++(1,0)--
    ++(0,1)-- ++(-1,0)-- ++(0,-1)-- ++(-2,0)-- ++(0,3)-- ++(2,0)-- ++(0,-1)--
    ++(1,0)-- ++(0,1)-- ++(1,0)-- ++(0,-1)-- ++(1,0)-- ++(0,-3)-- ++(-2,0)--
    ++(1,0)-- ++(0,-3)-- ++(1,0)-- ++(0,-1)-- ++(-6,0)-- ++(0,3)-- ++(2,0)--
    ++(0,-1)-- ++(1,0);
\end{tikzpicture}
```

Index

Cet index ne contient que des entrées crées automatiquement. Un bon index devrait contenir aussi des mots clés soigneusement choisis. Cet index n'est pas un bon index.