Formation du CIES Jussieu

Utilisation avancée de LaTeX

Manuel Pégourié-Gonnard

Printemps et été 2008

La présente version a été modifiée pour la dernière fois le 17 avril 2009(rév. 677).
La version la plus à jour est disponible, ainsi que son code source, sur www.institut.math.jussieu.fr/ mpg/latex/notes/.

Vos remarques et suggestions sont les bienvenues à mpg@elzevir.fr.

Première partie

Invitation à la programmation LaTeX

Computers are good at following instructions, but not at reading your mind.

(Donald E. Knuth)

Table des matières

Introduction

Une comparaison classique quand on souhaite expliquer LaTeX à un public plus habitué des traitement de textes du type cliquer-voir, est d'évoquer la méthode traditionnelle de préparation des documents : l'auteur prépare un manuscrit, souvent enrichi d'instructions typographiques (gras, italique, etc.) sous forme symbolique, puis un artisan typographe compose le document à partir du manuscrit, en prenant soin des détails techniques. En cas de soucis, l'auteur communique facilement avec l'éditeur, le correcteur et/ou du metteur en pages.

Aujourd'hui l'auteur est la plupart du temps chargé de composer le document lui-même, et se retrouve confronté, d'une part à des choix typographiques (qui ne sont pas notre propos ici), mais à des outils informatiques comme LaTeX, qui prennent la place (technique) du metteur en pages. L'interaction avec LaTeX est beaucoup plus difficile qu'avec un humain : il comprend pas toujours ce que l'on veut, il est difficile de lui demander, et, quand ça ne marche pas, ses messages d'erreur sont souvent incompréhensibles.

Par ailleurs, une des grandes réussites de LaTeX par rapport à Plain TeX, est de proposer un langage de haut niveau, permettant d'obtenir avec relativement peu d'instructions des résultats de bonne qualité, grâce à de bons réglages par défaut. Il n'est par contre souvent pas facile pour l'utilisateur de changer ces réglages pour personnaliser son document, et on a parfois l'impression que tous les articles écrits avec LaTeX se ressemblent.

De plus, le système sophistiqué de classes et d'extensions, ainsi que la large communauté d'utilisateurs, apporte des solutions pour une quantité considérable de problèmes sous LaTeX. Cependant, cela peut donner un sentiment d'impuissance ou d'incompréhension à l'utilisateur, qui résout tel ou tel problème en installant un peu à l'aveuglette telle extension, ou pire, en recopiant tel morceau de code lu quelque part sans savoir de quoi il retourne.

Savoir programmer LaTeX, c'est disposer d'un langage précis et puissant pour être en mesure de lui demander de faire exactement ce que vous voulez, et de comprendre vraiment ce qui se passe. C'est donc une étape très naturelle dans l'acquisition d'une meilleure maîtrise de LaTeX.

1 Commandes et environnements

1.1 Définitions de commandes

1.1.1 Sans arguments

```
\newcommand\⟨nom⟩{⟨définition⟩}
\renewcommand\⟨nom⟩{⟨définition⟩}
```

Sous LaTeX, on définit les nouvelles commandes [1] avec `\newcommand`, et on peut également redéfinir des commandes existantes avec `\renewcommand`. La syntaxe est assez simple et n'appelle pas beaucoup de commentaires. Notons qu'on peut au choix entourer `\⟨nom⟩` d'accolades ou non, comme dans l'exemple suivant.

```
\usepackage{amsmath}
\newcommand\va{\mathcal{A}}
\newcommand{\cdn}{\boldsymbol{k}}
Soit $\va$ une variété abélienne définie
sur un corps de nombres $\cdn$.
```

Soit $\mathcal{A}$ une variété abélienne définie sur un corps de nombres $\boldsymbol{k}$.

Avant de continuer, un petit guide de lecture valable tout au long de ces notes. La mise en forme ⟨*suivante*⟩ signale des éléments que vous ne devez pas saisir textuellement, mais remplacer par quelque chose d'approprié. Dans les exemples, un filet vertical en début de ligne signale que cette ligne est à placer dans le préambule. Enfin, je n'indiquerai pas systématiquement les extensions à charger quand elle me semblent assez standard : par exemple, amsmath (`amsldoc.pdf`) pour les exemples avec des maths, ou babel avec l'option frenchb (`babel.pdf`, sections 29 et 30) pour le français. Par ailleurs, les noms d'extensions seront accompagnées du nom de leur documentation chaque fois qu'il n'est pas évident (voir section 4.3.1).

Revenons aux définitions de commandes. À quoi servent des commandes sans arguments comme celles que nous venons de voir ? En général, on commence à les utiliser juste économiser sur la frappe du texte pour les notations courantes. Ce n'est pas une mauvaise idée en soit, mais il ne faut pas en rester là : on peut utiliser les définitions pour donner du *sens* à son code, et pour rendre la mise en forme plus souple et plus cohérent grâce à une couche d'abstraction supplémentaire.

Concrètement, on aurait pu appeler `\AA` la macro `\va` de l'exemple précédent. C'est aussi court à taper, et on peut faire toutes les lettres calligraphiques comme ça [2] ce qui a l'air cohérent à première vue. En fait, c'est assez mauvais, car revient à dire à LaTeX *comment* composer quelque chose, au lieu de lui dire *ce* qu'il doit écrire. En appelant `\va` une variété abélienne, on peut taper tout un texte sans risque de confusions de

1. Également appelées macros ou, par abus, séquences de contrôle.
2. Et `\Aa`, `\Bb`, etc. pour les majuscules grasses, comme on le voit bien (trop ?) souvent.

notations. Mieux, on peut changer très facilement de notation en modifiant une ligne du préambule, si par exemple on s'est aperçu qu'on utilisait déjà par mégarde le $\mathcal{A}$ pour un autre objet (ce qui est bien plus difficile si on a toujours saisi `\AA`).

Un autre usage fréquemment rencontré, et qui est techniquement mauvais (contrairement au précédent qui n'est que sous-optimal), est d'utiliser `\def` pour définir ses macros. Outre le fait que `\def` présente une syntaxe différente[3] pour les définitions plus évoluées que nous verrons bientôt, la principale différence entre `\⟨re⟩newcommand` et `\def` est que `\def` n'offre aucune sécurité : il nous laissera redéfinir un commande existante (et peut-être essentielle) sans broncher, ce qui est très dangereux[4].

Les cas d'utilisation de `\newcommand` et de `\renewcommand` sont donc complémentaires et disjoints : on utilise le premier pour définir une nouvelle commande, et le deuxième pour re-définir une commande existante. Si l'on se trompe, on obtient un message d'erreur (ce qui est *bien* !) et la (re-)définition n'est *pas* effectuée. Ces deux commandes partagent par contre la même syntaxe. Nous verrons tout à l'heure d'autres variantes.

Le lecteur attentif aura remarqué que je n'ai pas encore précisé ce qu'on pouvait utiliser comme ⟨*nom*⟩ et comme ⟨*définition*⟩. Pour la ⟨*définition*⟩, c'est simple : n'importe quel texte équilibré en accolades[5] fera l'affaire. En fait, il y a juste une restriction sur l'emploi du caractère `#` : si vous voulez vraiment l'utiliser, il vous faudra l'écrire deux fois (`##`) car, nous le verrons bientôt, il a une signification spéciale.

Le ⟨*nom*⟩ obéit à des règles plus strictes : il peut être, au choix, composé d'une suite quelconque de lettres, ou bien d'un unique caractère qui n'est pas une lettre. Pour TEX, il y a exactement 52 lettres : les 26 minuscules et les 26 majuscules. En particulier, les caractères accentués comme « é » ne sont *pas* des lettres, les caractères « a » et « A » sont différents. Ainsi, des noms de macros possibles sont `\abs`, `\+` ou même `\é`. Par contre, si vous écrivez `\ça` ou `\abs2`, TEX ne lira pas ce que vous pensez sans doute : il verra dans le premier cas `\ç` suivit de la lettre `a`, et dans le second `\abs` suivit du caractère `2`[6].

Pour les noms de macros composés de lettres, le premier caractère qui n'est pas un lettre joue donc le rôle de délimiteur. Knuth a pensé que l'espace jouerait souvent ce rôle, et a donc ajouté la règle suivante dans TEX : *les espaces suivant un nom de macro composé de lettres sont ignorés*. Cette règle, qui est souvent bonne, cause aussi souvent des surprises. On peut les éviter en utilisant une paire d'accolades[7] pour délimiter le nom. On peut aussi, lors de la définition de sa macro, placer à la fin la commande `\xspace` de l'extension éponyme, qui se charge d'ajouter de l'espace si (et seulement

3. Dans un sens moins évoluée car elle ne permet pas de gérer simplement des arguments optionnels, mais pour certains cas plus puissante, grâce à la notion d'arguments délimités qui dépasse le cadre de cet exposé.

4. Par exemple, saviez-vous que `\par` fabrique les paragraphes, et `\output` les pages ? Je vous laisse imaginer les conséquences d'une redéfinition hasardeuse…

5. C'est-à-dire, dans lequel il y a (dans le bon ordre) autant d'accolades ouvrantes que fermantes.

6. Un usage fréquent pour contourner ceci quand veut numéroter des macros, et de le faire en romain : `\macroi`, `\macroii`, etc.

7. Qui constituent en fait un groupe vide, comme on le verra tout à l'heure. D'ailleurs, le groupe vide n'est pas toujours sans effet : comparer soigneusement *effet* et *effet* (resp. `effet` et `ef{}fet`) par exemple.

si, en principe) il y a lieu.

```
\usepackage{xspace}
\newcommand\ADN{acide
  désoxyribonucléique\xspace}
Connaître le \TeX book par c\oe ur, mais
utiliser \LaTeX maladroitement, n'est-ce
pas étrange ?\par L'\ADN c'est bon,
mangez-en. Mangez de l'\ADN.
```

Connaître le TEXbook par cœur, mais utiliser LATEXmaladroitement, n'est-ce pas étrange ?
L'acide désoxyribonucléique c'est bon, mangez-en. Mangez de l'acide désoxyribonucléique.

Observez bien les espaces dans l'exemple[8] précédent : il en manque entre « LATEX » et « maladroitement ». Par contre, `\xspace` a rajouté l'espace manquant en milieu de phrase, mais pas avant le point, ce qui est bien. Je renvoie à la documentation d'xspace pour plus de détails, notamment sur la gestion de ses erreurs. Ceci est bien sûr inutile pour les macros utilisées en mode mathématique, où les espaces sont de toutes façons ignorés.

Enfin, la dernière remarque que je ferai concernant les macros en général, c'est justement qu'il faut penser au mode (mathématique ou non) dans lequel elles vont être utilisées. À ce sujet, LATEX intègre une commande `\ensuremath` qui prend un argument, et le compose en mode mathématique quelque soit le mode courant : si on était déjà en mode mathématique, `\ensuremath` ne fait rien, sinon, il encadre son argument d'une paire de dollars.

Utiliser ou non `\ensuremath` est une histoire de goût. Personnellement, je préfère ne pas l'utiliser en général : j'estime que c'est à moi de passer en mode mathématique quand il le faut, et je préfère que les commandes que je définis se comportent comme les commandes standard de LATEX, qui ne fonctionnent que dans un mode donné. Une macro un peu plus intéressante, fournie par l'extension fixltx2e, est `\TextOrMath`, qui prend deux arguments, et compose le premier quand elle est appelée en mode texte, et le second quand elle est appelée en mode mathématique.

1.1.2 Avec arguments obligatoires

```
\newcommand⟨*⟩\⟨nom⟩[⟨n⟩]{⟨définition⟩}
\renewcommand⟨*⟩\⟨nom⟩[⟨n⟩]{⟨définition⟩}
```

Pour définir une commande prenant des arguments, il suffit d'utiliser le premier argument optionnel de `\newcommand` pour spécifier le nombre d'arguments. Celui-ci doit être un entier compris entre 0 et 9 (s'il est égal à 0, on retombe dans le cas précédent). Au sein de la ⟨*définition*⟩, on accède aux arguments par leur numéro, précédé d'un `#`, par exemple `#1`, `#3`. C'est d'ailleurs la raison pour laquelle il ne peut pas y avoir plus de 9 arguments.

8. Vous remarquerez peut-être que j'utilise `\par` au lieu d'une ligne vide pour changer de paragraphe dans les exemples : c'est uniquement pour gagner de la place.

```
\newcommand*\eng[1]{%
  \foreignlanguage{english}{\emph{#1}}}
\newcommand*\coord[3]{(#1_{#2},\dots,#1_{#3})}
La NAT, ou \eng{network address translation},
consiste à \emph{traduire} des adresses réseau.
Rien à voir avec une \emph{translation}
de vecteur $\vec{x} = \coord{x}{1}{n}$ !
```

La NAT, ou *network address translation*, consiste à *traduire* des adresses réseau. Rien à voir avec une *translation* de vecteur $\vec{x} = (x_1, \dots, x_n)$!

Un piège courant est de supposer que, lors du remplacement[9] les signes comme `#1` seront considérés comme une entité unique. Ce n'est pas le cas : TEX remplace d'abord tout[10], puis examine le texte obtenu comme s'il s'agissait de quelque chose de totalement nouveau, où le remplacement de `#1` est donc éventuellement considéré comme plusieurs éléments distincts. Notez à ce sujet la précaution prise d'entourer chaque indice par des accolades ci-dessus. Cette précaution est nécessaire comme le montre l'exemple suivant.

```
\newcommand*\good[1]{(x_{#1}, y_{#1})}
\newcommand*\bad[1]{(x_#1, y_#1)}
Clairement, $\good{ij} \neq \bad{ij}$.
```

Clairement, $(x_{ij}, y_{ij}) \neq (x_ij, y_ij)$.

Vous aurez sans doute remarqué que j'utilise systématiquement la version étoilée de `\newcommand` pour définir mes commandes prenant un argument. Mais que fait donc cette étoile ? Elle informe LATEX que les arguments de la macros seront « courts », dans le sens où il ne feront pas plus d'un paragraphe. Techniquement, cela signifie qu'il n'y aura pas de ligne vide dans le source au milieu de l'argument, ou de commande `\par` (ce qui est équivalent).

Quand vous définissez une commande, à moins que vous n'envisagiez vraiment de lui passer des arguments s'étendant sur plusieurs paragraphes (ce qui est rare, on préférera généralement des environnements dans ce cas), utilisez systématiquement l'étoile. Ceci vous permet, dans le cas où vous oublieriez l'accolade fermante à la fin de l'argument, d'obtenir rapidement un message d'erreur (`Runaway argument?`) explicite : encore une fois, un message d'erreur c'est *bien*, surtout s'il n'apparaît pas trop tard.

Si vous pensez que vos arguments peuvent être longs, n'utilisez pas l'étoile. Dans toute la suite, la notation $\langle * \rangle$ signifiera : une étoile optionnelle. Enfin, s'il vous arrive de devoir passer un argument long à une macro prévue pour des arguments courts, sachez que vous pouvez utiliser `\endgraf` au lieu de `\par` ou d'une ligne vide pour changer de paragraphe : cette dernière commande ne provoquera pas d'erreur.

Enfin, je n'hésite pas à me répéter : ma remarque concernant le choix du nom des macros reste bien évidement valable ici ; on a tout intérêt à choisir des noms ayant du *sens* et pas de simple raccourcis de mise en forme.

9. Le terme technique consacré est le *développement* de la macro. Une bonne compréhension des mécanismes de développement de TEX est essentielle pour le programmer de façon poussée. Nous ne faisons que les aborder superficiellement ici.

10. En vrai, c'est un peu plus compliqué, mais pour le moment, il vaut mieux imaginer que les choses se passent comme ça.

1.1.3 Avec arguments optionnels

Les arguments optionnels servent essentiellement pour des commandes dont un argument est presque toujours le même : il est dans ce cas dommage de toujours le répéter. Pour cela, LaTeX propose la notion d'argument optionnel, avec une valeur par défaut. La syntaxe (désormais complète) de `\newcommand` qui le permet est la suivante.

```
\newcommand⟨∗⟩\⟨nom⟩[⟨n⟩][⟨défaut⟩]{⟨définition⟩}
\renewcommand⟨∗⟩\⟨nom⟩[⟨n⟩][⟨défaut⟩]{⟨définition⟩}
```

Si, après avoir indiqué le nombre d'arguments, vous utilisez le deuxième argument optionnel de `\newcommand`, cela indique à LaTeX que le premier argument de votre commande sera optionnel, et que sa valeur par défaut est ⟨*défaut*⟩. Attention, ⟨*n*⟩ désigne toujours le nombre total d'arguments, et pas juste le nombre d'arguments obligatoires. On accède toujours aux arguments par `#1` à `#9` (l'argument optionnel étant `#1`), et l'étoile optionnelle garde la même signification que précédemment.

```
\newcommand*\chevaux[2][1]{%
  (C_{#1},\dots,C_{#2})}
Considérons $n$ chevaux $\chevaux{n}$ ; par
hypothèse de récurrence, $\chevaux{n-1}$ et
$\chevaux[2]{n}$ sont de la même couleur\dots
```

Considérons n chevaux $(C_1,\dots,C_n)$; par hypothèse de récurrence, $(C_1,\dots,C_{n-1})$ et $(C_2,\dots,C_n)$ sont de la même couleur…

Toutefois, comme vous l'aurez remarqué, LaTeX impose un certain nombre de restrictions : il ne peut y avoir qu'un argument optionnel, et celui-ci devra être donné en premier (entre crochets) lors de l'appel de la macro. Si celles-ci vous dérangent, vous pouvez utiliser l'extension xargs (`xargs-fr.pdf`) qui propose une macro `\newcommandx`, avec une syntaxe légèrement différente de celle de `\newcommand`, permettant de s'en affranchir.

```
\newcommandx⟨∗⟩\⟨nom⟩[⟨n⟩][⟨liste⟩]{⟨définition⟩}
\renewcommandx⟨∗⟩\⟨nom⟩[⟨n⟩][⟨liste⟩]{⟨définition⟩}
```

Ici, au lieu de spécifier une valeur par défaut pour l'unique argument optionnel, on indique une ⟨*liste*⟩ séparée par des virgules de paires ⟨*numéro*⟩=⟨*défaut*⟩ indiquant les numéros des arguments optionnels et leur valeur par défaut. Pour plus de détails, je renvoie à la documentation d'xargs (`xargs-fr.pdf`).

```
\usepackage{xargs}
\newcommandx*\coord[3][1=1, 3=n]{%
  (#2_{#1},\ldots,#2_{#3})}
\begin{align*} \coord{x} &\quad \coord[0]{x} \\
  \coord{x}[m] &\quad \coord[0]{x}[m] \end{align*}
```

$$(x_1,\dots,x_n) \quad (x_0,\dots,x_n)$$
$$(x_1,\dots,x_m) \quad (x_0,\dots,x_m)$$

Enfin, pour des possibilités plus avancées, s'affranchissant de la limite des neuf arguments, et surtout de la syntaxe positionnelle (arguments numérotés par opposition à des arguments nommés), mais avec une méthode de définition radicalement différente, on pourra regarder du côté d'xkeyval, qui sera abordé à la section 3.3.

1.2 Environnements

Pour certains cas, une commande est mal adaptée : il s'agit essentiellement[11] des situations ou un des arguments est particulièrement long, par exemple l'énoncé d'un théorème. Pour ceci, LaTeX a introduit la notion d'environnements. Ceux-ci ne sont pas plus difficiles à définir que les commandes ; une bonne partie étant similaire à la définition des commandes, voici directement la syntaxe complète de `\newenvironment`.

```
\newenvironment⟨∗⟩{⟨nom⟩}[⟨n⟩][⟨défaut⟩]{⟨début⟩}{⟨fin⟩}
\renewenvironment⟨∗⟩{⟨nom⟩}[⟨n⟩][⟨défaut⟩]{⟨début⟩}{⟨fin⟩}
```

La différence principale avec `\newcommand` est qu'il faut désormais fournir deux définitions : une pour le ⟨*début*⟩, une pour la ⟨*fin*⟩ de l'environnement. Ces définitions peuvent écrire du texte, effectuer des changements de fontes, incrémenter des compteurs, etc. ou au contraire être vides (cas fréquent pour la ⟨*fin*⟩, comme on le verra).

L'autre différence concerne le ⟨*nom*⟩ : celui-ci est désormais obligatoirement donné entre accolades, et n'est plus précédé d'une contre-oblique. Surtout, les règles le régissant sont différentes : le ⟨*nom*⟩ peut[12] être composé d'une suite quelconque de lettres et d'étoiles (généralement, l'étoile est unique et se trouve à la fin).

Tout le reste est commun : un environnement peut ne prendre aucun argument, ou en prendre jusqu'à 9, dont le premier est éventuellement optionnel. Notons que le *contenu* de l'environnement (qui s'étend jusqu'au `\end`) n'est pas un *argument*. Il faut aussi être conscient d'une limitation : les arguments ne sont pas accessibles dans la définition de ⟨*fin*⟩. On peut tricher en « sauvegardant » le ou les arguments utilisés dans des macros, et utiliser ces macros à la fin, comme dans l'exemple suivant.

```
\newenvironment*{citetexte}[1]{%
  \newcommand\auteur{#1}\begin{quote}\itshape}{%
  \par\nointerlineskip\rule{\linewidth}{0.5pt}
  \raggedleft \bsc{\auteur}\par\end{quote}}
\begin{citetexte}{Knuth} Gentle reader, this
is an handbook about \TeX, [\ldots]\end{citetexte}
```

Gentle reader, this is an handbook about TEX, [. . .]

KNUTH

Tout se passe comme si il y avait au début de l'environnement une commande, qui prend les éventuels arguments et peut faire des choses avec, et à la fin de l'environnement, une autre commande, qui n'a bien évidement pas accès aux arguments de la première. En fait, c'est réellement le cas : un environnement `env` est implémenté[13] par les deux commandes[14] `\env` et `\endenv`.

Une pratique courante quand on définit des environnements, est de faire dériver ceux-ci d'autres environnements : par exemple, la définition du début de `enva` peut inclure un `\begin{autreenv}` et la fin `\end{autrenv}`. Dans certains cas, pour des

11. Il y a aussi le problème du verbatim, qui ne fonctionne pas dans l'argument d'une commande.
12. En pratique, on peut utiliser des noms plus compliqués, mais comme il n'est pas explicitement marqué dans le manuel qu'on peut, autant s'en tenir à ça.
13. Il faut toutefois être conscient que `\begin` et `\end` font un peu plus que d'appeler ces deux commandes ; on parlera par exemple de la notion de groupes ci-dessous.
14. Inversement, les commandes peuvent être utilisées comme des environnements, techniquement. Ce n'est en général pas recommandé, car cela peut causer des surprises, petites ou grosses.

raisons techniques, on aura intérêt à utiliser la forme `\autreenv` et `\endautreenv` : c'est par exemple le cas pour les environnements d'amsmath (`amsldoc.pdf`) et ceux de type verbatim. Dans d'autres cas, il faut au contraire utiliser `\begin` et `\end` (par exemple, `lrbox` et `multicols`). Dans la plupart des cas, c'est indifférent ; il n'y a pas de règle générale [15] pour ça, je n'en dirai donc pas plus.

Dans l'exemple précédent, vous avez peut-être remarqué que je change la forme de la police courante avec `\itshape` au début de l'environnement, et que je prends pas la peine de rétablir la forme initiale à la fin. De même, je définis `\auteur` avec `\newcommand`, mais cela ne va-t-il pas poser problème à la deuxième utilisation de l'environnement ? En fait, non, grâce au fait que les environnement définissent un *groupe* au sens de TEX. Faisons une petite parenthèse sur cette notion TEXnique générale.

Les groupes, en TEX, servent à limiter la portée des opérations qui y sont effectuées : (re-)définitions de commandes, changement de fontes, de valeur des longueurs, etc. Il est important de comprendre que la portée ne concerne pas les « variables », mais bien les opérations qui sont effectuées dessus : il n'y a pas de notion de variable locale en TEX. Avant d'illustrer ceci par un exemple, voyons quels sont les différents moyens de délimiter un groupe.

Le premier est d'utiliser `\begingroup` et `\endgroup`. C'est ce qui est implicitement fait par `\begin` et `\end` quand vous utilisez un environnement en LATEX. L'autre moyen est d'utiliser des accolades `{` et `}`, à condition qu'elles ne servent pas déjà à autre chose : les accolades qui entourent un argument de macro *ne* définissent *pas* un groupe. Par contre, c'est le cas quand vous écrivez `{\Large en gros}` par exemple. Enfin, `\bgroup` et `\egroup` sont des synonymes de `{` et `}` quand il s'agit de groupes. Leur avantage est qu'on peut placer un `\bgroup` seul dans une définition sans la rendre déséquilibrée en accolades. Signalons pour finir qu'un groupe commencé par `\begingroup` ne peut être terminé par `\egroup` ou `}` et réciproquement [16]

Voyons maintenant comment les groupes opèrent sur la portée des définitions, et des fontes, par exemple. On découvre ci-dessous la commande `\meaning`, qui sera évoquée à la section suivante.

```
\newcommand\truc{abc}
{ \scshape \meaning\truc\par
  \renewcommand\truc{zyx}
  \meaning\truc\par } \meaning\truc
```

```
\LONG MACRO:->ABC
\LONG MACRO:->ZYX
\long macro:->abc
```

Revenons enfin à nos chers environnements, après cette longue parenthèse, néanmoins essentielle, sur la notion de groupes. Ce qu'il faut en retenir, c'est que la plupart [17] des choses que vous faites à l'intérieur d'un environnement n'ont d'effet que jusqu'à la fin de l'environnement. Il faut absolument en profiter, et *ne pas* prendre la

15. Autre que d'étudier l'implémentation des environnements en question, et de repérer les comportements problématique : lire le contenu comme un (ou des) argument(s) ou en mode verbatim, fermer un groupe au début et l'ouvrir à la fin…

16. En fait, TEX distingue plusieurs type des groupes qui ont des usage différents. `\begingroup` et `\endgroup` servent à définir des groupes semi-simples, alors que les groupes délimités par `{` ou `\bgroup` et `}` ou `\egroup` sont des groupes simples. Pour notre usage, nous n'aurons pas besoin de connaître la différence entre les deux.

17. Plus précisément, toutes les assignations sauf celles précédées du préfixe `\global` ou sont définies comme globales au chapitre 24 du TEXbook.

peine de rétablir les valeurs initiales à la fin : d'une part pour économiser ses efforts, d'autre part parce que c'est une pratique bien plus robuste.

```
\newenvironment{horssujet}{%
  \sffamily\slshape}{}
Un truc bien important\par
\begin{horssujet} Une longue remarque
décalée.\par \end{horssujet}
Un autre truc tout à fait sérieux.
```

Un truc bien important
Une longue remarque décalée.
Un autre truc tout à fait sérieux.

On pourrait être tenté ici de terminer l'environnement par `\normalfont` pour revenir à la fonte par défaut : c'est une erreur, car si jamais on a, par exemple, changé la taille de la fonte à cette endroit, ce changement serait annulé par le `\normalfont` à la fin. De même, utiliser `\rmfamily\upshape` à la fin n'est pas une bonne idée non plus, car peut-être que la fonte à l'extérieur était italique… Bref, mieux vaut se reposer sur le mécanisme des groupes qui est fait pour ça.

1.3 Compléments TEXniques

Je regroupe un peu en vrac dans cette section quelques remarques plus ou moins liées à la définition de commandes sous TEX/LATEX, ou plus généralement à la façon dont TEX traite le source, soit parce qu'elles sont un peu plus techniques (ou moins souvent utiles) que les points abordés précédemment, soit qu'elles ne s'inséraient pas naturellement de le cours du texte. Elles peuvent sans doute être passées en première lecture, sauf 1.3.1 et 1.3.2.

1.3.1 Accès aux macros privées

Comme je le mentionnais ci-dessus, le nom d'une macro, s'il fait plus d'un caractère, est composé uniquement de lettres. Par défaut TEX reconnaît les 52 lettres de l'alphabet latin comme des lettres, mais il est en fait très configurable, et on peut ajouter des caractères à sa liste de lettres si on veut [18]. Ceci est utilisé par LATEX pour obtenir une notion de macro privée : ce sont les macros dont le nom contient un `@`.

`\makeatletter`
`\makeatother`

Ces macros sont en général inaccessibles à l'utilisateur, pour la raison évoquée. Si on veut y accéder, il faut utiliser la commande `\makeatletter` pour faire de `@` une lettre, manipuler à sa guise les macros privées, et enfin utiliser `\makeatother` pour refaire de `@` un « autre caractère » au sens de TEX. Ceci est très utile pour pouvoir, dans votre préambule, personnaliser certaines commandes, ou simplement pour accéder à des commandes de LATEX réservé au programmeur, dont nous verrons des exemples aux sections 1.3.4 et 3.3, ainsi que pages 3.1 et 2.1.2.

Il faut savoir que, au cours de la lecture des fichiers de classe ou d'extension (fichiers `.cls` et `.sty`), `@` est automatiquement une lettre sans avoir besoin de dire

18. Je ne vous conseille pas de jouer à ça hors du cas évoqué ici, et d'ailleurs je ne dirai pas comment on fait en général.

`\makeatletter`. C'est un point à garder en tête quand vous lisez de tels fichiers, et surtout à utiliser quand vous en écrivez. Vous pouvez aussi dans vos préambule mettre des `@` dans les noms de macros internes si vous en utilisez [19].

1.3.2 Traitement des espaces par TEX

Parlons un peu de la façon dont TEX lit le source, et plus précisément de sa gestion des espaces lors de cette lecture. Comme on l'a vu, les espaces sont ignorés après une commande dont le nom est composé de lettres. Par ailleurs, plusieurs espaces consécutifs dans le source seront toujours traités comme un seul espace. Enfin, les espaces au début d'une ligne seront toujours [20] ignorés, ce qui permet d'indenter son code sereinement à sa guise.

Au contraire, il faut se méfier de certains espaces. Une fin de ligne est équivalente à un espace pour TEX (à cette différence près que deux fins de lignes consécutives produisent un `\par` qui fini le paragraphe en cours). Vous aurez remarqué des nombreux `{%` dans mes exemples : ils servent à aller à la ligne sans produire d'espace. Si on ne prend pas cette précaution (ce qu'on appelle *commenter ses fins de lignes*), on se retrouve souvent avec des espaces non voulus, dits parasites (*spurious spaces* en anglais). Dans l'exemple suivant, ces espaces ne sont pas forcément gênantes, il s'agit juste de prendre conscience de leur présence.

```
\fbox{
  Du texte.
  }
\fbox{%
  Du texte.%
  }%
\fbox{Du texte.}
```

Du texte. Du texte.Du texte.

Il n'est bien sûr pas utile de commenter toutes les fins de lignes. Par exemple, celles qui suivent une commande dont le nom est composé de lettres : l'espace sera de toutes façons ignoré à ce niveau. Par ailleurs, si la commande est appelée en mode vertical ou mathématique, les espaces produits par les fins de lignes ne seront pas pris en compte. Toutefois, il est très utile de prendre l'habitude de commenter systématiquement [21] ses fins de lignes quand on définit des commandes : ce n'est pas seulement une question de présentation, car les espaces peuvent aussi bousiller vos alignements, voire pire [22].

19. Notez que bien sûr vous pouvez utiliser une macro qui utilise une macro en `@` même après que `@` a cessé d'être une lettre ! C'est une conséquence notable du fait qu'en TEX le texte de remplacement est « tokenisé » au moment de la définition.

20. Sauf dans des contextes très spéciaux comme un environnement verbatim, qui modifie la notion qu'a TEX de ce qui constitue une ligne.

21. Vous remarquerez sans doute que je commente parfois un peu trop, ou parfois juste le minimum selon l'humeur. Chacun son style : la présentation du source compte aussi

22. Surtout si vous faites des choses TEXniques avec des `\expandafter` ou des arguments délimités. Sans aller si loin, je vous laisse imaginer l'effet dans une boucle répétée 100 fois…

1.3.3 Différences entre macros et fonctions

Un autre point essentiel à comprendre dès qu'on veut faire des choses un peu subtiles avec les macros, est justement que ce sont des macros, et non des fonctions. La distinction est loin d'être théorique et a une grande importance en programmation TEX. Par exemple, quand on écrit $f(g(x))$, on sait tous que g va être évalué *avant* f. Au contraire en TEX, dans `\f{\g{x}}`, non seulement c'est `\f` qui est développé en premier, mais peut-être que `\g` ne le sera jamais, suivant ce que fait `\f` de son argument : il faut bien voir qu'à ce stade, l'argument de `\f` est juste une suite de symboles.

```
\newcommand\abs[1]{\lvert#1\rvert}
\showarg{\abs{x}}\par
\newcommand\fortytwo{42}
\newcommand\quarantedeux{{42}}
$2^\fortytwo \neq 2^\quarantedeux$
```

```
\abs {x}
2^4 2 ≠ 2^42
```

Dans l'exemple ci-dessus, j'ai précédemment défini[23] une commande `\showarg` dont la seule action est d'afficher son argument : on constate que comme je l'ai annoncé, ce qu'elle a reçu en argument est la suite de symboles `\abs{x}` et non pas son résultat, `\lvert x\rvert` ou autre. Par ailleurs, à la dernière ligne, on voit que le résultat du développement[24] d'une macro n'est pas toujours traité comme une unité.

Une étude complète et précise des règles de développement de TEX dépasse largement le cadre de cet exposé, et le lecteur intéressé pourra consulter le TEXbook ou [Eij91]. Il faut en tout cas retenir que tout ne se passe pas comme avec des fonctions : une macro n'est pas un truc qui renvoie un résultat qui peut être passé à d'autres macros, c'est juste une règle de remplacement.

Il est aussi utile de comprendre ce qui se passe, ou plutôt ce qui ne se passe pas, lors de la définition d'une macro : TEX ne cherche pas à évaluer le texte de définition de la macro au moment de la définition, il va juste le lire[25] comme une suite de symboles, sans se demander s'ils ont du sens, et encore moins chercher à évaluer ce sens.

```
\newcommand\un{un} \newcommand\truc{\un\deux}
\newcommand\deux{deux} \renewcommand\un{pas un : }
\newcommand\important{\textbf} \important{\truc}
```

pas un : deux

On voit clairement qu'au moment de la définition de `\truc`, le fait que `\deux` ne soit pas défini ne gêne aucunement, et que d'autre part le sens de `\un` qui a compté est celui qu'il avait au moment du *développement* de `\truc`, et pas au moment de se *définition*. Par ailleurs, il est intéressant de méditer la définition de `\important`, qui en fait en pratique une macro à un argument, même si elle est syntaxiquement définie comme une commande sans argument.

23. La définition n'est pas incluse pour ne pas perturber le lecteur : elle fait appel à des commandes non abordées ici.

24. Le lecteur attentif remarquera aussi que dans ce cas, `\fortytwo` a été développé *avant* que `_` n'agisse, ce qui montre au moins deux choses : `_` n'est pas une macro, et TEX est parfois compliqué.

25. Pour être précis, le texte va être lu comme une suite de lexèmes au sens de TEX, pas seulement comme une suite de caractères. C'est une différence subtile mais importante entre TEX et d'autres langages de macros : la « tokenisation » intervient au moment de la définition.

Ces règles de définitions sont en général très pratiques et permettent une grande souplesse dans la programmation de TEX. Il convient néanmoins de signaler un piège courant, qui apparaît par exemple quand on souhaite gérer une liste, et qu'on essaye d'ajouter un élément à la liste en faisant `\renewcommand*\liste{\liste, élément}` : après ça, le développement de `\liste` engendrera une boucle sans fin[26] due à un appel récursif. Il y a bien sûr des moyens de s'en sortir, mais ils sortent malheureusement du cadre de cet exposé.

1.3.4 Variantes étoilées

Vous connaissez sans doute des commandes ou environnements qui admettent une version étoilée : par exemple, l'environnement `align*` fait la même chose que `align` mais sans numérotation, `\section*` permet de faire un section non numérotée et n'apparaissant pas dans la table des matières. Vous pouvez vouloir définir des commandes ou environnements de façon similaire.

Si vous avez bien suivi les paragraphes sur les noms de commandes et d'environnements, vous aurez sans doute compris que c'est très facile pour les environnements : il vous suffit de définir *deux* environnements, l'un nommé `env` et l'autre `env*`. Bien sûr, ces deux environnements peuvent eux-même faire appel à un même environnement interne, pour éviter de dupliquer le code, en lui passant des arguments différents ou en réglant la valeur d'un booléen avant l'appel (voir section 3.1).

`\@ifstar{`⟨*version avec étoile*⟩`}{`⟨*version sans étoile*⟩`}`

Pour les macros, c'est plus compliqué car l'étoile ne peut faire partie du nom, et la solution retenue est la suivante : il n'y a qu'une commande, qui vérifie si elle est suive d'une étoile, et exécute une variante ou l'autre. Pour cela, on utilise `\@ifstar`, qui est une commande interne de LATEX (voir section 1.3.1). Le fonctionnement, j'espère, est suffisamment clair sur l'exemple suivant. Deux points essentiels sont à noter : d'une part, la commande « externe » ne prend pas d'arguments, il sont gérés par les commande internes. Par ailleurs, l'appel à `\@ifstar` avec ses deux arguments doit être le tout dernier élément de la définition de la macro externe.

```
\makeatletter
\newcommand*\ciel@star[1]{ciel #1 étoilé}
\newcommand*\ciel@nostar[1]{ciel #1 non étoilé}
\newcommand\ciel{\@ifstar{\ciel@star}{\ciel@nostar}}
\makeatother Un \ciel{bleu} et un \ciel*{nocturne}.
```

Un ciel bleu non étoilé et un ciel nocturne étoilé.

1.3.5 Outils rudimentaires de déboguage

Une constante universelle quand on aborde la programmation, est que les choses marchent rarement comme on veut du premier coup. On se tourne alors vers diverses techniques, que j'appellerai de déboguage, pour comprendre ce qui se passe.

26. Enfin, il y aura une fin assortie d'un `TeX capacity exceeded, sorry`, vu que la récursion n'est ici pas terminale…

`\show`	`\meaning`
`\showthe`	`\the`

La première question à se poser, c'est de vérifier comment sont définis les objets utilisés. Pour cela, on dispose des commandes `\show` (affichage sur le terminal et dans le fichier `log`) et `\meaning` (affichage dans le document) qui donnent la signification d'une séquence de contrôle [27]. C'est utile non seulement pour vérifier la définition, mais pour savoir si un objet donné est une macro ou, par exemple, une longueur dans le sens de la section 2.2 : dans se cas, son `\meaning` sera `\skip` suivi d'un nombre. On peut aussi utiliser `\showcmd` de show2e (`show2e-fr.pdf`) pour les commandes à argument optionnel.

Une commande reliée, pour afficher non plus le sens, mais plutôt la valeur d'un objet, du moment qu'il est typé (par exemple une longueur), est `\showthe` (terminal et `log`) ou tout simplement `\the`. C'est principalement utile pour les longueurs, moins pour les compteurs que LaTeX gère à sa façon. Pour les longueurs, voir aussi l'extension showdim (`showdim.sty`).

`\tracing`⟨*chose*⟩`=`⟨*n*⟩
`\tracingmacros=1`
`\tracingcommands=1`

Enfin, il y a les diverses commandes de trace, qui produisent des inscriptions dans le fichier `log` pour certains types d'opérations. Elles ont une syntaxe bizarre, de type déclaratif [28], et agissent comme des commutateurs. Ici, ⟨*n*⟩ est en général 1 (activer) ou 0 (désactiver). Des exemples utiles sont `\tracingmacros` (voir le développement des macros et leurs arguments), `\tracingcommands` (exécution des commandes, y compris écriture des caractères : utile pour repérer un espace parasite par exemple).

La sortie des commandes précédente est verbeuse et demande une certaine habitude pour être exploitée efficacement. L'extension trace fournit les commandes pratiques `\traceon` et `\tracoff` qui activent ou désactivent d'un coup toutes les fonctions de traçage, tout en les désactivant automatiquement dans certains passages particulièrement verbeux (changement de fontes notamment).

Pour en savoir plus sur les commandes de traces de TeX, une référence commode est le chapitre 34 de [Eij91]. On aura aussi intérêt à consulter le manuel d'ε-TeX [Bre98], sections 3.3 et 3.4, pour ses ajouts intéressants. Pour une discussion plus générale de la gestion des erreurs sous LaTeX, voir aussi l'annexe B [29] de [MG05].

27. Ou d'ailleurs de n'importe quel lexème.
28. En fait, c'est la syntaxe TeXienne d'une assignation
29. Disponible sur `http://www.latex-project.org/guides/lc2fr-apb.pdf`, et absente de la version papier.

2 Objets typés de LaTeX

2.1 Compteurs

2.1.1 Nom et affichage

Les compteurs sont omniprésents dans LaTeX, dont une des grandes forces (par rapport à Plain TeX par exemple) est de gérer automatiquement pour vous la numérotation des sections, les références croisées, etc. Histoire de procéder méthodiquement, disons d'abord qu'en LaTeX le nom d'un compteur est une suite de lettres (sans \ au début), et donnons la liste des compteurs définis par défaut sous LaTeX et les classes standard (extraite de `latex.pdf`[1]) :

part	paragraph	figure	enumi
chapter	subparagraph	table	enumii
section	page	footnote	enumiii
subsection	equation	mpfootnote	enumiv
subsubsection			

Disons aussi qu'à chaque environnement de type théorème, défini par exemple avec `\newtheorem`, est associé un compteur, à moins que le théorème ne soit déclaré comme associé à un compteur existant. En choisissant ce compteur, on peut par exemple s'amuser à numéroter continûment ses équations et théorèmes, propositions, etc. avec les commandes :

```
\newtheorem{thm}[equation]{Théorème}
\newtheorem{prop}[equation]{Proposition}
\begin{equation} 1 = 2 \end{equation}
\begin{thm} C'est faux.\end{thm}
\begin{equation} 1 \neq 2 \end{equation}
\begin{prop} C'est vrai.\end{prop}
```

$$1 = 2 \tag{1}$$

Théorème 2 C'est faux.

$$1 \neq 2 \tag{3}$$

Proposition 4 C'est vrai.

Dans l'exemple ci-dessus, si on avait pas spécifié `[equation]` dans la première déclaration, un nouveau compteur nommé `thm` aurait été créé, qu'on aurait pu alors utiliser à notre guise comme tout autre compteur.

\arabic	\roman	\alph
\fnsymbol	\Roman	\Alph

La chose la plus naturelle qu'on puisse vouloir faire avec un compteur, c'est d'afficher sa valeur. Pour cela, on dispose de 6 commandes correspondant à des styles

1. Si vous avez de la chance, ce fichier existe sur votre disque dur. Sinon c'est dommage, car il n'est pas sur le CTAN…

divers : en chiffres arabes `\arabic`, en chiffres romains majuscules ou minuscules (resp. `\Roman` et `\roman`), en lettres (`\alph` et `\Alph`). Notez que les deux dernières paires ont les limitations évidentes : on ne peut afficher ainsi que des compteurs positifs, et inférieurs à 26 pour les lettres. Enfin, `\fnsymbol` utilise successivement les 9 symboles suivants : ∗, †, ‡, §, ¶, ‖, ∗∗, ††, ‡‡.

`\the`⟨*compteur*⟩

Bien qu'on puisse utiliser directement les commandes précédentes si on le souhaite, il est souvent mieux d'utiliser la notion de « représentation standard » fournie par LaTeX : à chaque ⟨*compteur*⟩ est associée une commande `\the`⟨*compteur*⟩, qui détermine la façon dont il sera représenté aux endroits usuels : titre de section, références, table des matières… On peut redéfinir cette commande comme n'importe quelle autre, au moyen de `\renewcommand`, comme sur l'exemple de mauvais goût suivant.

```
\renewcommand\theequation{%
  équation \no \thesection.\Roman{equation}}
\begin{equation}\label{l} 0 = 0 \end{equation}
D'après \eqref{l}, on a \dots
```

$$0 = 0 \quad \text{(équation n° 2.1.V)}$$

D'après (équation n° 2.1.V), on a …

Certains compteurs sont particuliers car ils ont en fait une deuxième représentation standard. Il s'agit des compteurs utilisés par les énumérations `enumi` etc. (il y en plusieurs pour les énumérations emboîtées), auxquels sont associées des commandes comme `\labelenumi` qui déterminent comment ils sont composés au sein de la liste, `\theenumi` restant utilisé pour les références. Notons au passage que si on souhaite personnaliser les listes standard de LaTeX, on aura plutôt intérêt à se tourner vers les extensions enumitem ou enumerate.

2.1.2 Manipulation et références

`\newcounter{`⟨*nom*⟩`}[`⟨*dépendance*⟩`]`
`\@addtoreset{`⟨*compteur*⟩`}{`⟨*dépendance*⟩`}`
`\@removefromreset{`⟨*compteur*⟩`}{`⟨*dépendance*⟩`}`

Quand les compteurs standard ne suffisent, pas on en définit facilement de nouveaux au moyen de `\newcounter`. Cette commande prend en argument obligatoire le nom du compteur à créer, et en argument optionnel le nom d'un compteur existant, dont va *dépendre* au sens suivant : à chaque fois que ⟨*dépendance*⟩ sera incrémenté[2], « nom » sera remis à zéro. Le cas qui motive cette notion est simple : `subsection` dépend de `section` qui dépend de `chapter`, etc.

Après qu'un compteur a été défini (par exemple s'il s'agit d'un compteur standard), on peut quand même le rendre dépendant d'un autre compteur, ou au contraire supprimer sa dépendance, au moyen des commandes `\@addtoreset` (de LaTeX) et `\@removefromreset` (de l'extension remreset (`remreset.sty`, commentaires au début)). Par exemple, dans un document dont certaines sections mais pas toutes com-

2. Avec un des commandes de LaTeX prévues à cet effet.

portent des sous-sections, et où les équations sont numérotées par sous-section, on peut avoir envie de faire dépendre `equation` de `section` en plus de `subsection`.

Signalons la commande `\numberwithin` de l'extension amsmath (`amsldoc`) qui permet non seulement de gérer la dépendance du compteur `equation` mais simultanément ajuste sa représentation par `\theequation`. On a aura intérêt à toujours veiller à une certaine cohérence à ce sujet pour ses propres compteurs.

```
\setcounter{⟨compteur⟩}{⟨valeur⟩}
\addtocounter{⟨compteur⟩}{⟨valeur⟩}
\value{⟨compteur⟩}
```

On ne dispose en LaTeX pur que de peu de possibilités pour faire des opérations sur les compteurs[3]. On peut assigner une valeur à un compteur avec `\setcounter`, et lui ajouter un entier avec `\addtocounter`. Ici, ⟨*valeur*⟩ est soit un nombre écrit en chiffres, soit la valeur d'un autre compteur, obtenue avec `\value`. C'est d'ailleurs la seule occasion où sert la commande `\value`, qui a un statut assez particulier[4], est utile. Dans ce contexte, elle présente un avantage sur `\arabic` par exemple : on peut la faire précéder d'un coefficient multiplicateur. C'est le seul type d'arithmétique disponible par défaut (voir section 3.2).

```
\stepcounter{⟨compteur⟩}
\refstepcounter{⟨compteur⟩}
```

En fait, ce qu'on a le plus souvent de faire comme opération sur un compteur, c'est l'incrémenter (lui ajouter 1). Les deux commandes ci-dessus s'en chargent, et gèrent au passage la notion de dépendance : tous les compteurs dépendants de ⟨*compteur*⟩ seront remis à zéro. C'est l'unique différence entre `\stepcounter{`⟨*compteur*⟩`}` et la formulation `\addtocounter{`⟨*compteur*⟩`}{1}` qu'on pourrait croire équivalente. La commande `\refstepcounter` a une action supplémentaire : elle règle ce que j'appellerai la *référence courante*, et est donc la plus utile.

Arrêtons-nous un moment sur cette notion. À tout instant, LaTeX garde dans un coin de sa mémoire la référence courante : c'est ce qu'il utilisera comme compteur dès qu'on lui demandera de poser une étiquette avec `\label`. Par exemple, juste après une commande `\section`, c'est le compteur `section` ; dans un environnement `equation` ou théorème, c'est le compteur éponyme ; dans une énumération, c'est `enumi` ou autre selon le niveau ; juste après ces environnements, la référence courante redevient ce qu'elle était auparavant (par exemple `section`). De nombreuses erreurs avec les références sont dues au fait que la référence courante n'est pas celle qu'on croyait quand on a placé le `\label`. medbreak J'attire ici votre attention sur la portée des opérations en LaTeX : toutes les opérations sur les compteurs[5] ont un effet *global* : il n'est pas restreint par les groupes[6]. Par contre, la notion de référence courante est *locale* : la valeur précédente est restaurée à la fin du groupe courant. Enfin, le référence courante

3. Comme le reflète la dénomination, un compteur n'est pas considéré *a priori* comme une variable entière destinée à faire de l'arithmétique.

4. Précisément, elle renvoie une séquence de contrôle qui désigne le registre TeX interne où est stockée la valeur, et non pas la valeur elle-même.

5. Enfin, celles effectuées avec les opérations présentées ici : il en va différemment des opérations primitives de TeX.

6. Et heureusement, vu que les environnements définissent des groupes : on ne voudrait pas que

n'est jamais changée de façon rétroactive : il faut prendre garde de placer `\label` *après* `\refstepcoutner`.

Dans l'exemple suivant, inspiré de faits réels, on souhaite définir des macros permettant de gérer des suites de constantes numérotées automatiquement en fonction de leur ordre d'apparition. En interne, chaque constante est repérée par un label. Observez comment le groupe est utilisé pour restreindre l'action du `\refstepcounter` qui autrement empêcherait un bon référencement de l'équation en cours.

```
\newcounter{cst}
\newcommand*\newcst[1]{%
  \begingroup
  \refstepcounter{cst}\label{cst-#1}%
  \endgroup \cst{#1}}
\newcommand*\cst[1]{c_{\ref*{cst-#1}}}
Posons $\newcst{chose} = 2$ et $\newcst{truc}
= 2 \cdot \cst{chose}$. Alors $\cst{truc}$\dots
```

Posons $c_1 = 2$ et $c_2 = 2 \cdot c_1$. Alors c_2…

Notons qu'il faut alors deux compilations pour avoir une bonne numérotation de ses constantes, comme pour tout ce qui concerne les références croisées. J'en profite pour rappeler ici les commendes liées aux références croisées sous LaTeX : il y a bien sûr `\label` et `\ref`, mais aussi `\eqref` qui compose le numéro avec le même mise en forme que dans les équations (p. ex. entre parenthèses), et `\pageref` qui donne le numéro page de la référence. Je signale au passage l'extension lastpage qui place automatiquement un label sur la dernière page, permettant ainsi de numéroter ses pages avec `\thepage/\pageref{LastPage}` par exemple.

Il faut d'ailleurs noter que le mécanisme de construction des pages par LaTeX est asynchrone : au moment où il compose un texte, il ne sait pas encore sur quelle page il va le placer. Il n'est donc pas fiable (hors des en-têtes et pied de pages, qui sont gérés de manière spéciale), d'utiliser `\thepage` pour obtenir le numéro de la page en cours : on aura plutôt intérêt à utiliser le mécanisme de référence, sous le forme `\label{truc}\pageref{truc}` afficher ce numéro.

2.2 Longueurs

2.2.1 Unités, notion de ressort

En LaTeX comme en physique, une longueur est un nombre suivi d'une unité. Dans TeX, les longueurs peuvent être munies d'un signe : positif vers la droite et le bas, négatif dans l'autre sens. Un certain nombre d'unités sont utilisables pour spécifier les dimensions. Les principales sont `cm` et `mm` du système métrique, le pouce anglais `in` définit par $2,54\text{in} = 1\text{cm}$, le point typographique (anglais) `pt` définit comme $1/72,27\text{in}$ (environ 0.3515mm), le `bp` ou point PostScript, valant $1/72\text{in}$, et les unités dépendant de la fonte courante : `em` (le cadrat) et `ex` (la hauteur d'œil).

Ces deux dernières unités sont particulièrement importantes et trop peu connues. Elles sont fixés plus ou moins arbitrairement par la concepteur de la fonte, mais

le compteur `equation`, par exemple, soit remis à sa valeur précédente à la fin de l'environnement en cours.

représentent moralement la largeur d'un « M », resp. la hauteur d'un « x » dans la fonte courante, d'où leurs noms. On devrait la plupart du temps utiliser ces unités pour spécifier les espacements entre et autour des différents éléments de texte : on gagne ainsi en flexibilité pour les changements de fonte. Les unités absolues comme le point ou le cm ne devraient être utilisées que pour spécifier la mise en page au tout début, et éventuellement pour manier des graphiques externes.

Bien que la section s'appelle « longueurs », les objets dont nous allons parler sont en fait ce que j'appellerai des *ressorts*[7] : c'est une objet à trois composantes: une *longueur naturelle*, une *composante d'étirement* et une *composante de compression*. On spécifie ces trois composantes sous la forme

⟨*longueur*⟩ `plus` ⟨*étirement*⟩ `minus` ⟨*compression*⟩

Les composantes élastiques (étirement et compression) sont facultatives. Ainsi, la définition par défaut du ressort inter-paragraphes est équivalente à :

```
\setlength\parskip{0pt plus 1pt}
```

Les composantes élastiques sont essentielles pour permettre à TEX de justifier horizontalement les paragraphes, et verticalement les pages. Leur signification précise est la suivante : la composante de compressions est la dimension maximale que l'on a le droit de retrancher à la dimension naturelle du ressort : si cela ne suffit pas, on a droit à un message évoquant une `Overfull \hbox` et ça se voit : le texte déborde dans la marge. Pour la composante d'étirement, ce n'est pas un valeur limite : le ressort peut s'étirer plus que cette valeur, mais en provoquant une protestation de TEX (`Underfull` cette fois), et là aussi, ça se voit souvent, car c'est moche.

On dispose pour les composantes élastiques de toutes les unités habituelles, plus trois unités spéciales : le `fil`, le `fill` et le `filll`. Ces unités représentent des « infinis » incommensurables entre eux (par taille croissante précédemment) : si au moins un ressort d'une boîte a une composante « infinie » d'étirement, la boîte de pourra jamais être `Underfull`. On peut par contre faire de l'arithmétique[8] sur ces unités : `2fil` sont plus gros que `1fil`, mais toujours écrasés par le moindre `fill`.

Ces composantes élastiques infinies ont pour utilisation immédiate de choisir l'alignement : par exemple, pour centrer un texte, on règle les marges[9] de gauche et de droite à `0pt plus 1fil`. Traditionnellement, LATEX[10] n'utilise que des `fil` pour son usage interne, vous laissant le champ libre pour prendre la main grâce à `fill` et `filll`.

Les longueurs ont pour nom des séquences de contrôle, comme les commandes, et contrairement aux compteurs. Parmi les longueurs prédéfinies et utiles à connaître, citons `\parindent` (retrait d'alinéa en début de paragraphe), `\parskip` (espace vertical entre deux paragraphes), `\baselineskip` (espace entre les lignes de base de deux

7. En anglais, *glue* ou *skip* selon les auteurs et les circonstances.
8. Qui n'est pas celle des cardinaux infinis des matheux : raison pour laquelle j'emploie des guillemets pour « infini ». Le qualificatif « illimité serait sans doute plus adapté.
9. En fait, ce ne sont pas les marges, mais les paramètres internes `\rightskip` et `\leftskip` de TEX : on veut bien sûr centrer relativement aux marges.
10. Certaines extensions ou classes, comme beamer (`beameruserguide`), sont moins polies et utilisent déjà des `fill`.

lignes de texte successives. Il y a a aussi toutes celles liées à la mise en page, et notamment `\textwidth` et `\textheight` : voir les deux pages suivantes[11]. D'autres longueurs existent, comme celles qui contrôlent la mise en page des listes, pour lesquelles je renvoie à la figure 3.3 de [MG05].

Signalons aussi les longueurs magiques `\width`, `\height`, `\depth` et `\totalwidth`, utilisables dans les commandes de production de boîtes, comme nous le verrons à la section 2.3.

2.2.2 Commandes d'espacement

L'utilisation la plus naturelle des longueurs est pour spécifier le positionnement des éléments sur la page. Comme on ne peut pas spécifier de positionnement absolu par des coordonnées, on utilise un positionnement relatif, obtenu en ajoutant de l'espace (ou en en retirant, grâce aux longueurs négatives) vertical ou horizontal aux bons endroits sur la page.

`\hspace`⟨∗⟩`{`⟨*longueur*⟩`}`
`\vspace`⟨∗⟩`{`⟨*longueur*⟩`}`
`\stretch{`⟨*coefficient*⟩`}`

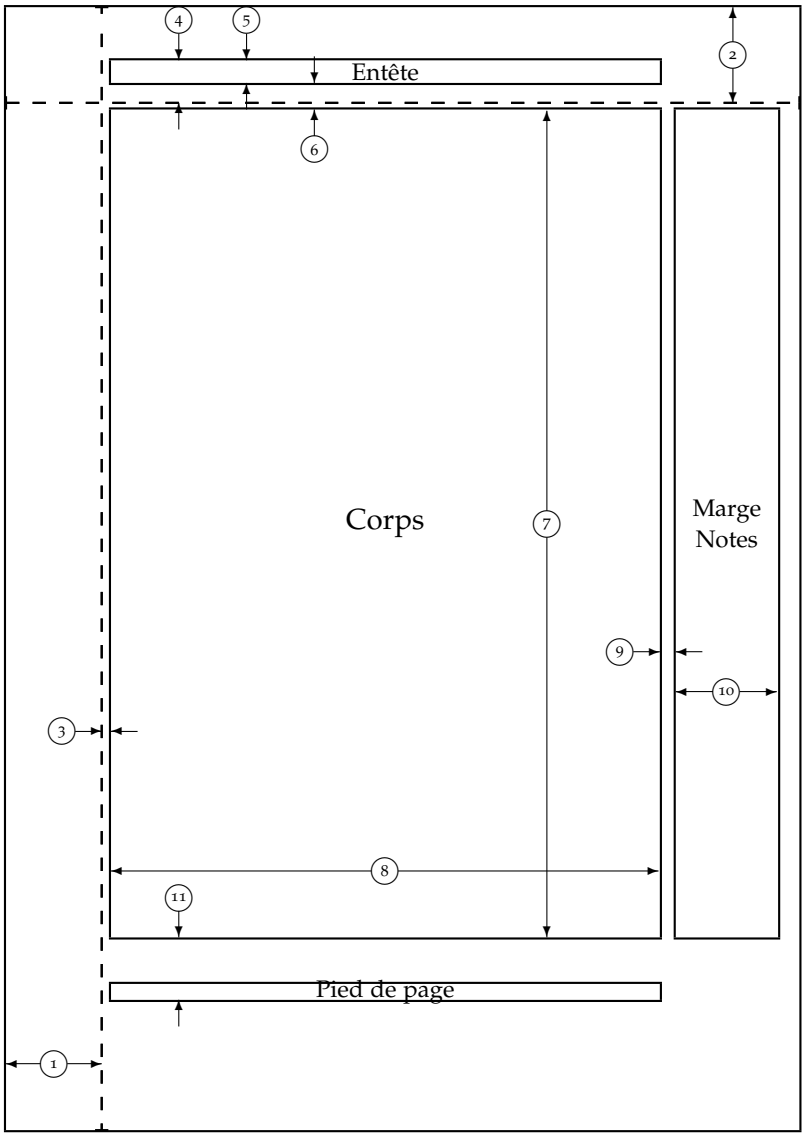
Les commandes de base pour cela sont `\hspace` et `\vspace`. L'effet de la seconde et parfois surprenant si on l'utilise au milieu d'un paragraphe : elle n'agit qu'après la fin de la ligne en cours (ici la commande était placée juste après « surprenant »). Les deux ont aussi un particularité souvent agréable : l'espace inséré disparaît automatiquement s'il se trouve en fin ou début de ligne (resp. de page). Si l'on souhaite éviter cela, il faut utiliser la forme avec un étoile.

Dans l'argument ⟨*longueur*⟩ de ces commandes, on peut bien sûr spécifier une longueur ou un ressort sous la forme vue précédemment. On dispose aussi de la commande `\stretch` qui fournit un espace élastique, d'étirement « infini » proportionnel au ⟨*coefficient*⟩, très utile par exemple pour composer une page de titre : on place un élément au tiers de la page en le faisant précéder de `\vspace*{\stretch{1}}` et suivre de `\vspace*{\stretch{2}}`.

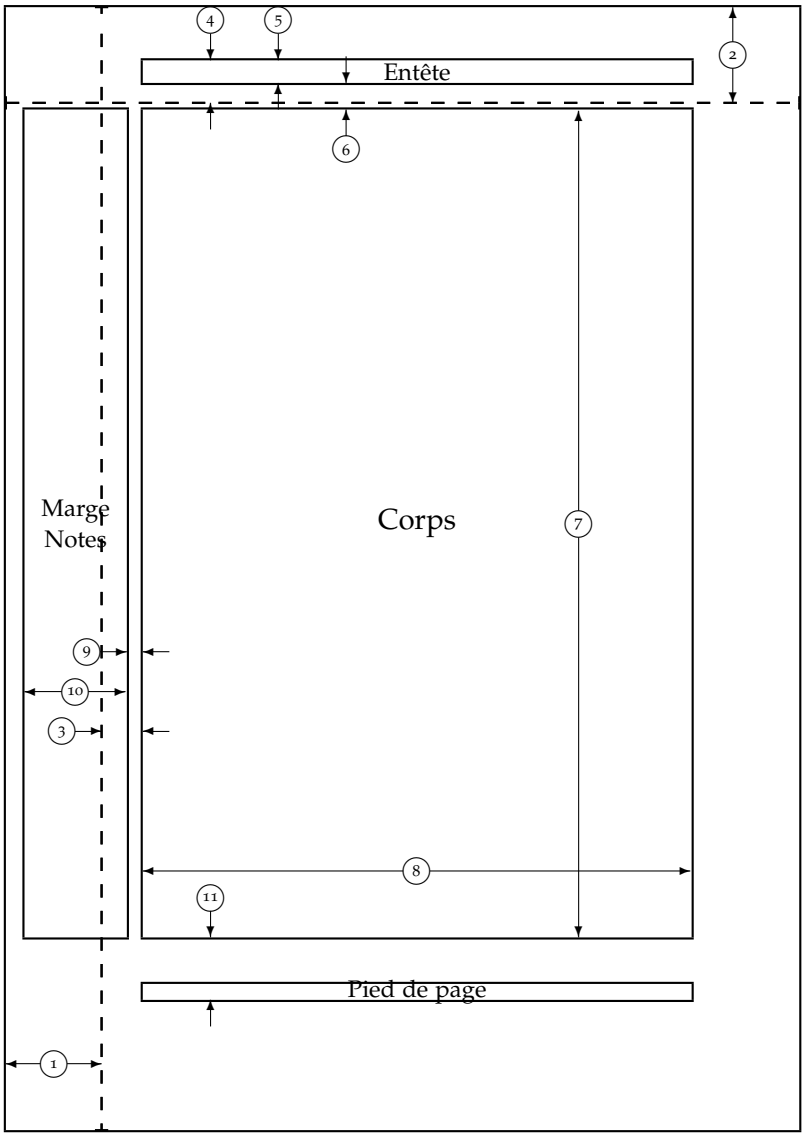
`\hfill` `\hrulefill`
`\dotfill` `\dingfill{`⟨*numéro*⟩`}`

Pour les espaces élastiques horizontaux, on dispose aussi du raccourci `\hfill`, équivalent à `\hspace{\stretch{1}}`. On a en fait d'autres commandes qui insèrent des objets élastiques tendant à remplir tout l'espace horizontal : une ligne pour `\hrulefill`, des pointillés pour `\dotfill`. Enfin, la commandes `\dingfill`, fournie par l'extension pifont (`psnfss2e.pdf`, section 7) remplit avec le symbole de ⟨*numéro*⟩ donné de la police de symboles Zapf Dingbats. J'ai terminé ce paragraphe par `\dingfill{43}` à titre d'exemple. ☞

11. Produites grâce à la commande `\layout` de l'extension éponyme, pour les curieux. Et pour dire toute la vérité, j'ai aussi utilisé la commande `\afterpage`, également fournie par l'extension de même nom.



1	un pouce + \hoffset	2	un pouce + \voffset
3	\oddsidemargin = 7pt	4	\topmargin = -32pt
5	\headheight = 17pt	6	\headsep = 20pt
7	\textheight = 623pt	8	\textwidth = 413pt
9	\marginparsep = 12pt	10	\marginparwidth = 77pt
11	\footskip = 47pt		\marginparpush = 6pt (non affiché)
	\hoffset = 0pt		\voffset = 0pt
	\paperwidth = 597pt		\paperheight = 845pt



1	un pouce + \hoffset	2	un pouce + \voffset
3	\evensidemargin = 31pt	4	\topmargin = -32pt
5	\headheight = 17pt	6	\headsep = 20pt
7	\textheight = 623pt	8	\textwidth = 413pt
9	\marginparsep = 12pt	10	\marginparwidth = 77pt
11	\footskip = 47pt		\marginparpush = 6pt (non affiché)
	\hoffset = 0pt		\voffset = 0pt
	\paperwidth = 597pt		\paperheight = 845pt

```
\\⟨*⟩[⟨longueur⟩]
\noindent
```

Terminons avec la commande \\ qui est très particulière. D'une part, car elle est souvent utilisé à tort et à travers pour obtenir des sauts de ligne, alors qu'il vaudrait souvent mieux utiliser \par pour finir le paragraphe, quitte à le faire suivre de \noindent si on ne souhait supprimer le retrait d'alinéa qui suit. D'autre part, parce que sa signification varie suivant le contexte : normal, tableau, ou environnement center. En temps normal, sa syntaxe est la suivante : l'étoile optionnelle permet d'inhiber une coupure de page au niveau de ce saut de ligne, et l'argument optionnel permet d'ajouter un espace vertical avant la ligne suivante.

2.2.3 Manipulation de longueurs

```
\newlength\⟨nom⟩
\setlength\⟨nom⟩{⟨longueur⟩}
\addtolength\⟨nom⟩{⟨longueur⟩}
```

Je passerai assez rapidement sur les commandes de définition et de réglages de longueurs, analogues à celles vues pour les compteurs. Disons seulement qu'on peut ou non entourer \⟨*nom*⟩ d'accolades selon ses goûts (comme avec \newcommand, et que la valeur d'une longueur peut être utilisée dans ⟨*longueur*⟩ sous le forme \⟨*nom*⟩, éventuellement précédé d'un coefficient multiplicateur, comme pour les compteurs, mais sans la commande \value. Par contre, les opération sur les longueurs sont locales : leur effet est limité au groupe courant (très pratique pour les environnements).

Là aussi, voir la section 3.2 pour des possibilités de calcul plus avancées, souvent très utiles car la calcul des longueurs constitue généralement une étape base dans la réalisation d'une mise en page.

```
\settowidth\⟨nom⟩{⟨matériel⟩}
\settoheight\⟨nom⟩{⟨matériel⟩}
\settodepth\⟨nom⟩{⟨matériel⟩}
```

Les trois commandes précédentes permettent de mesurer le ⟨*matériel*⟩ donné (du texte ou des choses plus compliquées) et de retenir le résultat dans une longueur. Pour l'explication des trois dimensions, voir la section 2.3.1. Une utilisation courante consiste à mesurer le plus grand item d'une liste pour créer des alignements. Une autre utilisation amusante consiste à créer des textes à trou.

```
\newlength\lgtrou
\newcommand*\trou[1]{%
  \settowidth\lgtrou{#1}%
  \hspace*{2\lgtrou}}
\setlength\baselineskip{1.2\baselineskip}
On laisse un trou \trou{plus} grand que
le mot parce qu'une \trou{écriture}
manuscrite prend plus de \trou{place}. On
augmente \trou{aussi} l'interligne\dots
```

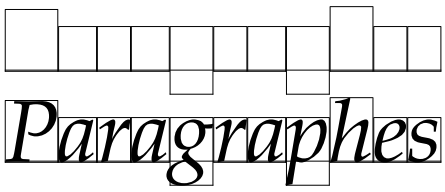
On laisse un trou grand que le mot parce qu'une manuscrite prend plus de . On augmente l'interligne…

On verra une variante de cette commande dans la section suivante, dès qu'on saura créer des boîtes et des réglures à notre guise.

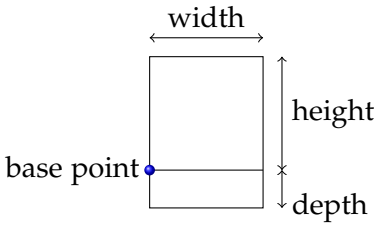
2.3 Boîtes

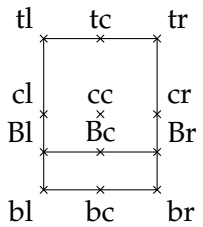
2.3.1 Modèle et types de boîtes

Dans cette section, on touche au coeur de la mise en pages avec TeX : en effet, pour TeX, tout est boîtes, des caractères individuels à la page elle-même. Composer des mots revient à aligner les boîtes-caractères. Sur l'illustration ci-dessous, TeX ne « voit » que la première ligne (remarquez au passage qu'un caractère peut déborder de sa boîte).



Comme on le voit sur cet exemple, les caractères ne sont pas alignés par le bas : il y a une notion de *ligne de base*. Pour cette raison, les boîtes de TeX ont trois dimensions : la largeur (*width*), la hauteur (*height*) mesurée à partir de la ligne de base, et la profondeur (*depth*) ; on parle aussi parfois de la hauteur totale, somme de la hauteur et de la profondeur. La figure suivante illustre ces notions, ainsi qu'une façon de repérer des points dans une boîte, souvent utilisée par les commandes « graphiques ».





Parlons maintenant des différents type de boîtes que connaît TeX. On peut les diviser en deux catégories : les boîtes qui peuvent contenir des choses, et celle dont le contenu est fixé. Dans la deuxième catégorie, on trouve les caractères et les réglures. Dans le première, on distingue les les boîtes horizontales et les boîtes verticales, selon la façon dont les boîtes placées à l'intérieur sont organisées entre elles : de gauche à droite ou de haut en bas. Pour chaque type de boîte sauf les caractères, on dispose de commandes variées pour les construire, que nous allons maintenant regarder en détails. (Pour les caractères, on verra comment jouer avec dans la partie sur le graphisme.)

2.3.2 Réglures

```
\rule[⟨décalage⟩]{⟨largeur⟩}{⟨hauteur totale⟩}
\nointerlineskip
```

Les réglures sont des boîtes pleines, c'est-à-dire noires. Comme tous les boîtes, elles sont rectangulaires, mais si l'une de leur dimensions est suffisamment petite, on peut les utiliser comme des lignes (c'est d'ailleurs le cas le plus courant). On les produit avec `\rule`, en spécifiant ses dimensions. Par défaut, la réglure est alignée sur la ligne de base.

L'argument optionnel sert à spécifier un ⟨*décalage*⟩ par rapport à la ligne de base : positif vers le haut, négatif vers le bas. S'il est négatif, la réglure produite aura une profondeur de -⟨*décalage*⟩ et une hauteur de ⟨*hauteur totale*⟩-⟨*décalage*⟩. Pour produire des réglures entre des lignes de texte, on aura intérêt à utiliser `\nointerlineskip` pour éviter que la réglure n'occupe une ligne à elle toute seule. On pourra aussi penser à utiliser `\noindent` pour éviter un retrait.

```
Un peu de texte pour commencer, puis
une coupure.\par\nointerlineskip\noindent
\rule{\textwidth}{1pt}\par
Et encore du texte après. Avec une réglure
d'apparence verticale \rule[-0.3\baselineskip]
{1pt}{\baselineskip} cette fois.
```

Un peu de texte pour commencer, puis une coupure.

Et encore du texte après. Avec une réglure d'apparence verticale | cette fois.

Ce type d'usage des réglures est le plus général. On peut les utiliser comme décoration sur une page de titre. On peut aussi les utiliser dans le texte : par exemple reprendre la commande `\trou` précédente en remplaçant l'espace par la commande `\rule{-1pt}{2\lgtrou}{1pt}` pour créer un trou souligné. Mais il existe une autre utilisation, plus subtile des réglures, qui correspond à ce que les typographes appellent des *montants*.

```
\strut
\arraystretch
\vphantom{⟨texte⟩}   \hphantom{⟨texte⟩}
```

Il s'agit de réglures de largeur nulle, donc invisibles, mais qui occupent de l'espace vertical dans la logique de TEX. C'est souvent très utile pour fixer une hauteur et/ou une profondeur minimum à certains objets. La commande `\strut` insère un montant qui a la hauteur du caractère le plus haut de la fonte courante, et la profondeur du caractère le plus profond. Elle est extrêmement utile. Elle est notamment placée automatiquement au début et à la fin de chaque case d'un tableau. Si l'on désire augmenter l'espace vertical minimum réservé dans une case de tableau, on peut redéfinir (avec `\renewcommand` le facteur multiplicatif `\arraystretch` (par défaut, 1) appliqué au `\strut` inséré). Par ailleurs, on peut utiliser en mode mathématique `\vphantom` et `\hphantom` pour obtenir des montants de la taille d'un texte donné.

```
\fbox{je} \fbox{dis} ou bien :
\fbox{je\strut} \fbox{dis\strut}.
\[ \sqrt{x} + \sqrt{y} +
\sqrt{\vphantom{y}x} \]
```

je dis ou bien : je dis.

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

2.3.3 Boîtes horizontales

```
\mbox{⟨matériel⟩}
\makebox[⟨largeur⟩][⟨lcrs⟩]{⟨matériel⟩}
```

La façon la moins sophistiquée de construire une boîte horizontale est d'utiliser `\mbox` : cela n'a pas beaucoup d'effet, à part celui de rendre le contenu insécable. C'est par exemple utilise pour composer les noms de famille, dont on ne souhaite pas qu'ils soient coupés en fin de ligne. La commande `\makebox` offre plus de possibilités : on peut choisir la largeur de la boîte créée, et l'alignement du ⟨*matériel*⟩ à l'intérieur : aligné à gauche avec `l` ou à droite avec `r`, centré avec `c`, ou enfin étiré (aligné des deux côtés) avec `s`. Pour cette dernière option, il vaut mieux que la boîte contienne des ressorts horizontaux suffisamment souples.

Une exemple intéressant d'utilisation de `\makebox` est de produire des boîtes de largeur nulle : ceci permet par exemple d'écrire dans la marge facilement, en début de paragraphe. À titre d'exemple et pour réviser un peu les compteurs aussi, une définition possible d'un environnement `exo` pour des exercices numérotés dans la marge, avec des références croisées qui marchent. Notez qu'ici il est important d'éviter les espaces parasites, qui se verraient sur l'alignement.

```
\newcounter{exo}
\newenvironment{exo}{\refstepcounter{exo}%
  \par\noindent \makebox[0pt][r]{\theexo\space}%
  \textbf{Exercice. }}{}
```

```
\fbox{⟨matériel⟩}    \framebox[⟨largeur⟩][⟨lcrs⟩]{⟨matériel⟩}
\fboxrule    \fboxsep
```

On peut aussi produire des boîtes et les encadrer avec les commandes `\fbox` et `\framebox` qui sont les analogues exacts des commandes précédentes, au niveau syntaxique. Le cadre est dessiné avec des traits d'épaisseur donné par `\fboxrule`, et une espace donné par `\fboxsep` sépare le contenu du cadre. Ces deux paramètre sont des longueurs, que l'on peut donc régler avec `\setlength`. Pour d'autres possibilités de cadres, notamment ombré, on pourra se tourner vers l'extension fancybox, ou des extensions spécialisées dans le graphisme comme PSTricks (`pst-user.pdf`) [12] ou TikZ (`pgfmanual.pdf`).

```
\raisebox{⟨décalage⟩}[⟨hauteur⟩][⟨profondeur⟩]{⟨matériel⟩}
```

Enfin, on peut élever ou abaisser une boîte par rapport à la ligne de base : là aussi, un ⟨*décalage*⟩ positif est vers le haut et un négatif vers le bas. Les paramètres optionnels permettent de mentir à TeX sur les dimension réelles de la boîte produite, afin de le forcer par exemple à ne pas en tenir compte. On peut ainsi s'amuser comme dans l'exemple suivant, directement tiré de [Oet+01].

12. Ou, selon les versions, `pstricks-doc.pdf`, plus vieux mais plus complet.

```
\raisebox{0pt}[0pt][0pt]{\Large\bfseries
  Aaaa\raisebox{-0.3ex}{a}%
  \raisebox{-0.7ex}{aa}\raisebox{-1.2ex}{r}%
  \raisebox{-2.2ex}{g}\raisebox{-4.5ex}{h}}
cria-t-il, mais la ligne suivante
ne remarqua pas qu'une chose
horrible lui était arrivée.
```

Aaaa$_{\textbf{aa}}$$_{\textbf{r}}$$_{\textbf{g}}$$_{\textbf{h}}$ cria-t-il, mais la ligne suivante ne remarqua pas qu'une chose horrible lui était arrivée.

2.3.4 Boîtes verticales

```
\begin{minipage}[⟨align.⟩][⟨hauteur⟩][⟨align. int.⟩]{⟨largeur⟩}
  ⟨texte⟩
\end{minipage}
\parbox[⟨align.⟩][⟨hauteur⟩][⟨align. int.⟩]{⟨largeur⟩}{⟨texte⟩}
```

Dès qu'on veut une boîte qui contient plusieurs lignes, il faut utiliser ce que LaTeX appelle une boîte verticale. Pour cela, on dispose de la commande `\parbox` et de l'environnement `\minipage` : pour la première, le contenu de la boîte est le dernier argument, ⟨*texte*⟩ alors que pour la seconde c'est le contenu de l'environnement. Outre la différence de présentation du source, cela a aussi pour conséquence que l'on peut utiliser des commandes comme `\verb` avec `minipage`, mais pas avec `\parbox` ; par ailleurs on peut utiliser des notes de bas de page dans une minipage et pas dans une `\parbox`. Étudions maintenant en détail la syntaxe commune, un peu compliquée, ces commandes.

Le seul argument obligatoire est ⟨*largeur*⟩, dont le sens est plutôt évident. Le premier argument optionnel, ⟨*align.*⟩, peut être une lettre parmi `t`, `c` et `b`. Il spécifie l'alignement de la boîte produite par rapport aux autres boîtes qui se trouvent sur la même ligne [13] : en termes plus techniques mais plus précis, il spécifie où se trouve la ligne de base de la boîte produite : avec `t`, elle coïncide avec la ligne de base de la première ligne du contenu [14], avec `b` elle coïncide avec celle de la dernière ligne, et avec `c` elle se trouve pile au centre de la boîte produite.

```
\newcommand\base{\rule{1em}{0.4pt}}
\newcommand\texte{A\\x\\g}
\noindent\base\parbox[t]{1em}{\texte}\base
\parbox[c]{1em}{\texte}\base
\parbox[b]{1em}{\texte}\base
```

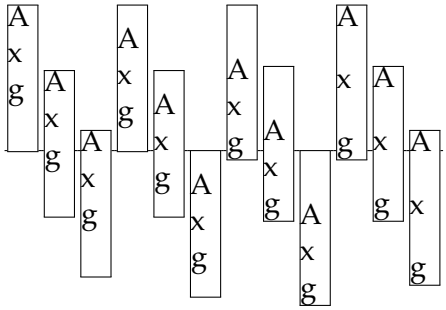
A
A x
A x g
x g
g

Par défaut, la hauteur de la boîte produite est exactement celle du texte qui est à l'intérieur. On peut toutefois en imposer une avec ⟨*hauteur*⟩, et le dernier argument prend alors un sens : ⟨*align. int*⟩ est une lettre parmi `t`, `b`, `c` et `s` qui dit si le texte à l'intérieur de la boîte doit être calé en haut, en bas, centré ou étiré pour coller en haut et en bas [15]. L'exemple suivant montre qu'on peut beaucoup s'amuser avec les

13. Il est intéressant de noter que les boîtes verticales sont en général alignées horizontalement entre elles.
14. Qui n'est pas forcément tout en haut de la boîte, si cette première ligne est relativement haute.
15. Pour être tout à fait précis, ces options insèrent un `\vss` respectivement en bas, en haut, les deux, et ni l'un ni l'autre, de façon similaire à ce qui est fait pour `\makebox`.

possibilités d'alignement ainsi offertes. On y constate notamment que l'alignement externe dépend parfois de l'alignement interne. On note aussi que la ⟨*hauteur*⟩ de la boîte n'est pas sa *hauteur totale*, mais bien la hauteur, et que la profondeur peut être nulle ou non selon l'alignement interne.

```
\newcommand\base{\rule{1pt}{0.4pt}}
\newcommand\showmp[2]{\base\fbox{%
    \parbox[#1][5em][#2]{1em}{%
      \setlength\parskip{0pt plus 1em}%
      A\par x\par g}}\base}
\setlength\fboxsep{0pt}\noindent
\showmp{b}{t}\showmp{c}{t}\showmp{t}{t}%
\showmp{b}{c}\showmp{c}{c}\showmp{t}{c}%
\showmp{b}{b}\showmp{c}{b}\showmp{t}{b}%
\showmp{b}{s}\showmp{c}{s}\showmp{t}{s}%
```



Pour finir, quelques remarques sur la production de boîtes verticales. D'une part, comme toujours quand on fait des mises en pages compliquées, attention aux espaces parasites (par exemple, penser à commenter la fin de ligne après `\end{minipage}`. D'autre part, pour l'alignement, il peut être utile de tenir compte de la profondeur, et ne pas hésiter à utiliser des `\strut` pour fixer cette dernière, ou à jouer sur des emboîtements de boîtes. Enfin, je signale l'extension varwidth (`varwidth.sty`, commentaires au début du fichier) qui fournit une variante de `minipage` de largeur adaptable automatiquement.

2.3.5 Manipulation de boîtes

```
\newsavebox\⟨nom⟩
\sbox\⟨nom⟩{⟨texte⟩}
\savebox\⟨nom⟩[⟨largeur⟩][⟨position⟩]{⟨texte⟩}
\usebox\⟨nom⟩
```

En plus de produire des boîtes, on peut aussi les mémoriser dans des « variables de boîte », et les réutiliser plus tard. Pour cela, il faut commencer par allouer une telle variable avec `\newsavebox`, puis lui affecter un contenu avec l'une des commandes `\sbox` ou `\savebox`, qui sont les analogues exacts de `\mbox` et `\makebox`. On peut ensuite ré-utiliser le contenu autant qu'on veut avec `\usebox`. Ceci est par exemple pratique [16] pour des éléments graphiques un peu compliqués que l'on veut insérer à plusieurs endroits dans le texte : cela évite à TeX d'avoir à le recalculer à chaque fois, et surtout évite les différences [17] entre les occurrences de l'élément.

```
\begin{lrbox}{\⟨nom⟩}[⟨largeur⟩][⟨position⟩]
  ⟨texte⟩
\end{lrbox}
```

16. Et surtout efficace, en termes de temps de compilation.
17. Dues, par exemple, à l'étirement des ressort : ce dernier est fixé une fois pour toutes au sein de la boîte, mais ne le serait pas en utilisant une macro.

Vous remarquerez qu'il n'y a pas de commandes de sauvegardes de boîte analogues à `minipage` ou `\parbox` : ces dernières sont tout simplement inutiles, car rien n'empêche de mettre une boîte verticale dans un boîte horizontale si on le souhaite. Par contre, il est intéressant pour diverses raisons de disposer d'un environnement permettant de mémoriser son contenu dans une boîte : c'est ce que fait `lrbox` qui est l'environnement équivalent à `\savebox`.

Pour finir, je rappelle que les assignations de boîtes faites avec l'une des commandes précédentes, comme toutes les assignations sauf celles des compteurs en LaTeX, sont locales au groupe (et donc à l'environnement) courant.

3 Extensions pour le programmeur

3.1 Tests et boucles

```
\ifthenelse{⟨test⟩}{⟨action si vrai⟩}{⟨action si faux⟩}
\isodd{⟨nombre⟩}
\lengthtest{⟨longueur a⟩⟨<=>⟩⟨longueur b⟩}
\equal{⟨chaîne a⟩}{⟨chaîne b⟩}
\AND \OR \NOT \( \)
```

L'extension standard ifthen fournit des tests divers sous une syntaxe relativement agréable. Les test est par défaut numérique, de la forme ⟨*nombre 1*⟩⟨<=>⟩⟨*nombre 2*⟩, où ⟨<=>⟩ désigne l'un des trois symboles <, = ou >. Comme ⟨*nombre*⟩, on peut utiliser un nombre écrit en chiffres ou la valeur d'un compteur avec `\value{`⟨*compteur*⟩`}`. Pour faire des tests sur les longueurs, on est obligé de les introduire avec `\lengthtest` à l'intérieur de ⟨*test*⟩. Par ailleurs, on peut faire des tests plus compliqués en utilisant les opérateurs logiques et des parenthèses. Par exemple, la macro `\settomaxwidth` suivante détermine la longueur du plus grand de deux morceaux de textes.

```
\newlength\lga \newlength\lgb \newlength\lgmax
\newcommand*\settomaxwidth[3]{\settowidth\lga{#2}\settowidth\lgb{#3}%
  \ifthenelse{\lengthtest{\lga>\lgb}}
    {\setlength{#1}{\lga}}{\setlength{#1}{\lgb}}}
% usage : \settomaxwidth\lgmax{abc}{ABC}
```

Par ailleurs, l'extension xifthen fournit quelques tests supplémentaires à utiliser avec `\ifthenelse`, comme `\isempty`, et la possibilité d'utiliser des expressions arithmétiques dans les tests, par exemple `\value{cnta}+2>\value{cntb}`. Si on la charge, elle remplace ifthen. Par contre, elle ne marchera pas sur certaines vieilles installations : elle repose sur des fonctionnalités d'ε-TEX (voir section 4.1.2).

```
\newboolean{⟨nom⟩}  \setboolean{⟨nom⟩}{⟨true/false⟩}  \boolean{⟨nom⟩}
\newif\if⟨nom⟩      \⟨nom⟩true \⟨nom⟩false
\if⟨nom⟩ ⟨texte si vrai⟩ \else ⟨texte si faux⟩ \fi
```

On peut avoir envie de mémoriser certaines options sous la forme de booléens, c'est-à-dire de variables ayant la valeur `true` ou `false`. Pour cela, on dispose de plusieurs méthodes. On peut définir et régler la valeur de variables booléennes avec ifthen, qu'on pourra alors utiliser dans le premier argument de `\ifthenelse`. On peut sinon utiliser un autre mécanisme, intégré à LaTeX, pour définir de telles variables. Le tableau ci-dessous montre les usages de ces deux solutions, peu ou prou [1] équivalentes :

1. En fait, plutôt peu si l'on veut entrer dans les détails, notamment au niveau du développement, du verrouillage des `\catcode`, de l'imbrication de différents tests, etc.

```
\newboolean{truc}                \newif\iftruc      % Allocation
\setboolean{truc}{true}          \tructrue         % Réglage
\ifthenelse{\boolean{truc}}{%    \iftruc           % Utilisation
  Truc est vrai.                   Truc est vrai.
  }{%                            \else
  Truc est faux.                   Truc est faux.
  }                              \fi
```

Parmi les autres possibilités, citons les commandes `\@ifmtarg` et `\@ifnotmtarg` de l'extension ifmtarg (`ifmtarg.sty`, commentaires à la fin du fichier), très utile pour vérifier simplement si l'argument optionnel d'une commande est vide ou non.

```
\makeatletter
\newcommand\Osheaf[2][]{%
  \mathcal{O}_{#2\@ifnotmtarg{#1}{, #1}}}
\makeatother
\[ \Osheaf{X} \hookrightarrow \Osheaf[x]{X} \]
```

$$\mathcal{O}_X \hookrightarrow \mathcal{O}_{X,x}$$

Enfin, on peut facilement faire des boucles de type « *for* » ou « *while* » à l'aide de la commande `\whiledo` de ifthen, ou de la commande `\multido` de l'extension multido.

3.2 Calculs

Comme on l'a vu aux sections sur les compteurs et les longueurs, les possibilités arithmétiques de LaTeX sont assez limitées. L'extension standard calc étend un peu ces possibilités. Notamment, elle permet d'utiliser les quatre opérations `+`, `-`, `*` et `/`, ainsi que les parenthèses, dans le deuxième argument de `\setcounter`, `\addtocounter`, ainsi que de `\setlength` et `\addtolenght`. Ceci marche aussi, par contre-coup, dans les arguments de type ⟨*longueur*⟩ des commandes de boîte : `\makebox`, `minipage`, etc. qui utilisent `\setlength` en interne.

```
\usepackage{calc}
\newcommand\cadrepage[1]{%
  \par\noindent\fbox{\parbox{%
      \linewidth -2\fboxsep -2\fboxrule}{#1}}}
\cadrepage{Si on n'avait pas prévu la place
  pour le cadre, il déborderait salement.}
```

Si on n'avait pas prévu la place pour le cadre, il déborderait salement.

Il faut toutefois être conscient du fait que les compteurs restent des entiers, lorsque l'on fait des divisions, et du fait que la multiplication des longueurs par des nombres décimaux est soumis à certaines restrictions. En fait, calc ne permet rien de plus que les opérations de bases de LaTeX, mais fournit surtout une syntaxe plus agréable et évite de passer par de nombreuses variables intermédiaires. L'exemple précédent est juste plus pénible à écrire sans calc.

`\widthof{`⟨*texte*⟩`}`	`\heightof{`⟨*texte*⟩`}`
`\depthof{`⟨*texte*⟩`}`	`\totalheightof{`⟨*texte*⟩`}`
`\maxof{`⟨*expr. 1*⟩`}{`⟨*expr. 2*⟩`}`	`\minof{`⟨*expr. 1*⟩`}{`⟨*expr. 2*⟩`}`

Par ailleurs, on peut aussi utiliser facilement dans le calcul les dimensions d'un matériel arbitraire, ainsi que le maximum ou le minimum de deux expressions de même type (compteur ou longueur). Ainsi, notre exemple précédent (voir page 3.1) de macro `\settomaxwidth` se récrit :

```
\newcommand*\settomaxwidth[3]{%
  \setlength{#1}{\maxof{\widthof{#2}}{\widthof{#3}}}}
```

Ceci illustre bien que `calc` sert surtout à simplifier les notations.

Pour faire du calcul plus poussé sur les nombres, on peut utiliser l'extension fp (`readme`)[2] qui permet du calcul « en virgule flottante », ou bien xlop (`fr-user.pdf`)[3] pour le calcul arithmétique et la présentation des calculs eux-même avec LaTeX. Il y a aussi les récentes extensions intcalc et bigintcalc qui font des calculs sur les entiers, voire sur des entiers de taille arbitraire[4]. Enfin, signalons les nouvelles primitives `\numexpr` et `\dimexpr` d'ε-TeX (section 3.5 de [Bre98]), utilisables dans les contextes où TeX attend un nombre[5] ou une dimension.

3.3 Syntaxe clé = valeur

Pour les macros ou environnement prenant plusieurs arguments, dont certains optionnels, on peut utiliser xargs (`xargs-fr.pdf`), mais il reste la limite des 9 arguments imposée par TeX, et surtout le côté pénible de la syntaxe positionnelle : on doit se souvenir de l'emplacement des arguments, plutôt que de leur nom, ce qui est plus difficile.

Pour contourner ce problème, xkeyval propose de remplacer la notion d'argument par un concept de clé, chaque clé pouvant recevoir une valeur. Il existe en outre plusieurs type de clés : générale, booléennes, à nombre fini de possibilités… On peut définir des familles de clés, afin de séparer les clés utilisées par différentes commandes. Les commandes ainsi crées prennent plusieurs arguments, dont l'un sera une liste de la forme clé=valeur, utilisé par xkeyval. Prenons l'exemple d'une commande servant à écrire un texte, éventuellement encadré, en choisissant les couleurs, et les dimensions du cadre.

```
\define@key{⟨famille⟩}{⟨nom⟩}[⟨val. def.⟩]{⟨définition⟩}
\define@boolkey{⟨famille⟩}{⟨nom⟩}[⟨val. def.⟩]{⟨définition⟩}
\setkeys{⟨famille⟩}{⟨liste⟩}
```

On définit une clé en général avec `\define@key` : on doit obligatoirement spécifier une ⟨*famille*⟩, qui sert à distinguer les clés utilisées par différentes macros, ainsi qu'un ⟨*nom*⟩ pour la clé. L'utilisateur pourra alors inclure ⟨*nom*⟩=⟨*valeur*⟩ dans sa liste d'options. La ⟨*valeur*⟩ est accessible en tant que `#1` dans la ⟨*définition*⟩ de la clé : si aucune

2. La seule documentation de cette extension est son fichier README, par exemple ici sur le CTAN, ou sur votre disque dur
3. Aussi étrange que cela puisse paraître, il n'y a qu'un seul fichier nommé comme ça sur le CTAN…
4. Ces deux dernières fournissent des macros purement développables, ce qui peut être très bien, même si expliquer pourquoi dépasse le cadre de ces notes.
5. Par exemple, `\numexpr 1+2\relax` ne marchera pas partout : on peut le faire précéder de `\number` pour l'y forcer.

valeur n'est spécifiée, c'est ⟨*val. def.*⟩ qui est utilisée à la place (comme pour une commande avec un argument optionnel). Il faut prendre garde que la valeur par défaut n'est utilisée que si la clé est spécifiée : il faut donc mieux régler les valeurs à l'avance avec `\setkeys` comme dans l'exemple ci-dessous.

Pour les clés booléennes, c'est à peu près pareil, sauf que d'une part il n'y a que deux valeurs légales : `true` et `false`, d'autre part un booléen façon LaTeX est créé (voir page 3.1), nommé `\ifKV@`⟨*famille*⟩`@`⟨*nom*⟩. Il sera automatiquement positionné à la bonne valeur, et la ⟨*définition*⟩ exécutée, au moment où l'on appellera `\setkeys`. On aura par ailleurs intérêt à enfermer l'appel à `\setkeys` dans un groupe, afin d'éviter que ses effets ne se propagent démesurément (changeant par exemple les réglages par défaut des autres macros).

Tout ceci peut avoir l'air compliqué au début, et ça l'est un peu en effet, mais j'espère que ça sera plus clair après avoir regardé un peu l'exemple[6] suivant. Pour plus de détails sur les possibilités nombreuses offertes par xkeyval, je renvoie à sa documentation.

```
\makeatletter
\define@key{magbox}{textcolor}[black]{\def\mb@tc{#1}}
\define@key{magbox}{background}[white]{\def\mb@bg{#1}}
\define@key{magbox}{framecolor}[black]{\def\mb@fc{#1}}
\define@key{magbox}{rule}[0.4pt]{\setlength\fboxrule{#1}}
\define@boolkey{magbox}{framed}[true]{}
\setkeys{magbox}{textcolor,background,framecolor,framed}
\newcommand*\magicbox[2][]{%
  \begingroup \setkeys{magbox}{#1}%
  \ifKV@magbox@framed
    \fcolorbox{\mb@fc}{\mb@bg}{\textcolor{\mb@tc}{#2}}%
  \else
    \colorbox{\mb@bg}{\textcolor{\mb@tc}{#2}}%
  \fi \endgroup}
\makeatother
\magicbox{Texte.} \magicbox[framed=false]{Texte.}
\magicbox[textcolor=red, background=gray]{Texte.}
\magicbox[framecolor=green, rule=2pt]{Texte.}
```

Texte. Texte.
Texte. Texte.

En fait, une fois passée la phase d'apprentissage, xkeyval est très pratique à utiliser : même s'il demande un peu plus de travail pour définir une commande, il offre en retour un grand confort pour l'utiliser. Son usage est donc plutôt à réserver à des commandes un peu complexes, avec beaucoup d'arguments.

6. Assez mauvais sur plusieurs points : par exemple, on peut spécifier une couleur pour le cadre même s'il n'y a pas de cadre… disons que c'est pour simplifier.

Bibliographie

[Bre98] Peter Breitenlohner. *The ε-TEX Manual*. Anglais. Version 2. 1998. URL: http://ctan.org/tex-archive/systems/e-tex/v2/doc/etex_man.pdf. P.: 18, 36.

[Eij91] Victor Eijkhout. *TEX by Topic. A TEXnician's reference*. Anglais. Wokingham: Addison Wesley, 1991. ISBN: 0-201-56882-9. URL: http://www.eijkhout.net/tbt/. P.: 16, 18.

[MG05] Frank Mittelbach et Michael Goossens. *LATEX Companion*. Français. Trad. par Jacques André et al. 2e éd. Paris: Paerson Education France, 2005. ISBN: 2-7440-7133-1. P.: 18, 24.

[Oet+01] Tobias Oetiker et al. *Une courte (?) introduction à LATEX 2ε. ou LATEX 2ε en 84 minutes*. Français. Version 3.20. 2001. URL: http://ctan.org/tex-archive/info/lshort/french/flshort-3.20.pdf. P.: 30.