Quelques aspects de TEX à travers les yeux de LATEX★

Proposé par **Sadik Boujaida**
ibouja@gmail.com

★ Il parait que TEX a des yeux, une bouche, un estomac et des boyaux

Table des matières

Avant-propos

LaTeX est pensé pour aider à s'affranchir des détails de la présentation et de la typographie pour ne se concentrer que sur l'organisation logique du contenu. Quand on utilise des commandes ou des environnements LaTeX, c'est plus pour leurs sens sémantique que pour obtenir un certain effet de style. La mise en page, les éléments de la présentation et les effets de styles peuvent être décrit dans des fichiers d'extensions séparés, de cette façon on peut complètement changer l'apparence d'un document sans toucher au contenu (dans la vie réelle, quelques ajustements seront en général nécessaires).

Pour sauvegarder la vitale et *très délicate* indépendance entre contenu et éléments de la présentation dans un contexte de collaboration professionnelle, les commandes et les environnement mis à la disposition des utilisateurs ne doivent pas se satisfaire de « fonctionner dans les cas usuels », mais aussi prévoir les situations exceptionnelles et limiter les effets non désirables. Il faut éviter d'être obligé de procéder à des ajustements dans les documents finaux pour corriger un éventuel comportement erratique d'une commande, surtout en ce qui concerne la gestion des sauts, qu'ils soient horizontaux ou verticaux. À titre d'exemple, une commande doit notamment bien agir dans le cas où son contenu est proche d'une terminaison de page. Rien n'est en effet plus néfaste pour cette indépendance qu'une rupture de page forcée quand elle n'est pas absolument nécessaire, car elle ruine la mise en page verticale au moindre changement de style ou de l'une des différentes dimensions utilisées dans le document (comme tout changement de contenu d'ailleurs). LaTeX n'offre malheureusement pas, à travers ses fichiers de classe standards, une grande latitude de personnalisation du style de certains éléments (les titres, par exemple) et incite ceux qui n'en sont pas satisfaits à créer leurs propres fichiers de classe sur la base de briques à assembler selon les besoins. Tache qui, outre la compétence requise en matière de typographie, exige de bien connaître les principes sous- jacents du fonctionnement du système. Faire l'effort de comprendre ces principes permet d'obtenir les effets désirés, mais aussi et surtout d'apprendre à éviter de gêner le système dans ce qu'il sait bien faire de lui même.

Dans toute tache de programmation, une part non négligeable du code est consacrée à la vérification du contexte d'exécution et à la gestion des erreurs. La programmation LaTeX ne fait pas exception.

Peu de fichiers de documentation adoptent le point de vue de l'utilisateur LaTeX qui veut aussi profiter des mécanismes natifs de TeX ou de ceux internes de LaTeX. C'est donc un choix délibéré dans ce document de traiter de cette facette. Y-à-il besoin de rappeler que LaTeX est un ensemble de macro ? Le langage de programmation c'est TeX qui le fournit et le code

TEX est une machine de Turing complète

interne de LaTeX est une source d'inspiration indéniable, autant qu'un vivier de commandes très utiles du point de vue de la programmation.

Pour éviter tout malentendu, quoique ce document peut sembler encourager l'usage de commandes de bas niveaux, ce n'est absolument pas en tant que commandes finales, mais comme une base pour la définition de macros LaTeX robustes et prévoyantes. On peut trouver un peu partout des commentaires qui diabolisent l'usage de commandes TeX lorsque on utilise LaTeX. Ces commentaires concernent toutefois l'utilisation dans les documents et de cette position, c'est certes une précaution vitale. Il suffit en outre d'ouvrir un quelconque fichier `.sty` ou `.cls` pour se rendre compte que les commandes de haut niveau de LaTeX (y compris `\newcommand` ou `\newenvironment`) n'y ont que très peu de place.

Ce document se base pour sa majorité sur l'ouvrage de référence « TeX by Topic », écrit par Victor Eijkhout, ainsi que sur les sources commentées du noyau LaTeX disponibles à travers le fichier `source2e.pdf`. Les deux sont librement téléchargeables sur les sites CTAN. Avec une installation complète (de TeXlive ou de MikTeX), il suffit pour les afficher de lancer en ligne de commande

```
texdoc texbytopic source2e
```

Une autre source a été utilisée pour actualiser et préciser certaines affirmations. C'est l'excellent site web de type question/réponse consacré à TeX et son écosystème : tex.stackexchange.com. Il est très courant d'y voir des réponses, et même parfois des questions, rédigées par ceux qui tiennent les rênes du dévelppement du système TeX/LaTeX et de ces extensions.

Présentation de la bête

Glossaires

Quelle est la différence entre le moteur et l'automobile ?

Le moteur TEX. C'est le programme TEX lui même, represenété par le fichier exécutable `initex`. Le propos de TEX n'est pas d'apporter des solutions finalisées, mais de définir les outils qui permettent d'en élaborer. Il propose un langage de programmation avec ses structures de données et de contrôle, des mécanismes d'assignement, de développement et d'éxécution, des tas d'algorithmes… et des commandes (primitives) qui donnent accés à ces fonctionnalités. Des évolutions du moteur original existent, celles qui sont arrivées au stade d'outils de production sont :

— eTEX, première extension de TEX qui a aboutit à un résultat utilisable. Elle a entre autre introduit des primitives qui sont devenues standards dans les extensions suivantes.

— pdfTEX moteur qui peut produire directement du **PDF**. Il est capable d'utiliser des polices **TrueType** (à condition de préparer les fichiers de métriques nécessaires) et possède un support de la micro-typographie. Il ajoute des commandes primitives qui donnent accés aux spécifications du format **PDF** comme les hyperliens, les bookmarks ou l'intégration d'éléments graphiques. Pouvant aussi produire du **DVI** (à travers l'option `-output-format=dvi`), il s'est imposé comme moteur par défaut pour la compilation des documents LATEX.

Le package `microtype` *offre une interface utilisateur avancée vers les fonctionnalités de micro-typographie du moteur pdfTEX. L'essayer c'est définitivement l'adopter.*

— xeTEX, moteur moderne supportant nativement l'**Unicode**, ce qui sert de base à la possibilité de traiter des documents multi-langues au niveau des sources. Il est en outre apte à accéder directement et

sans configuration préalable aux polices **OpenType** installées dans le système d'exploitation.

— *lua*TEX, le dernier moteur en date. Comme xeTEX il supporte l'**Unicode** et il est capable d'accéder aux polices de l'**OS**. Il possède dans sa dernière version les mêmes compétences que pdfTEX pour la microtypgraphie et il est partiellement compatible avec `microtype`.

*luaTEX embarque le langage de programmation interprété **Lua**. Il possède des primitives et une API qui permettent d'exécuter directement du code Lua à partir d'un fichier source TEX. Il ouvre ainsi la porte à la programmabilité de parties qui sont hermétiques dans les autres moteurs.*

pdfTEX est devenu le moteur par défaut pour le traitement des documents LATEX (depuis une dizaine d'années déjà), y compris pour la production des fichiers **DVI**. Pour s'en convaincre, exécuter la ligne de commande

```
ls -l $(which latex)
```

si le système utilisé est OSX ou une distribution Linux. On peut aussi lancer l'exécutable `latex` en ligne de commande (quelque soit le système utilisé) pour entrer dans le mode interactif : un message s'affiche dont un exemple est

```
This is pdfTeX, Version 3.1415926-2.5-1.40.14 (TeX Live
 2013/Debian)
```

Le moteur pdfTEX est actuellement figé dans sa dernière version (1.40) et il sera remplacé à terme par *lua*TEX. Le moteur TEX original n'est plus vraiment utilisé qu'avec les documents du format plainTEX.

Formats associés à TEX. Le moteur de lui même ne sait pas traiter des documents. Il lui faut un certain nombre de paramètres de configuration, une association entre ses fonctionnalités et les caractères qui les déclenchent (`\`, `{` ou `$` par exemple), mais surtout des macros de haut niveau utilisables dans les documents. C'est le format qui s'occupe de régler ces aspects. plainTEX est le format historique associé à TEX dès sa naissance. Les formats les plus utilisés sont LATEX et ConTEXt. Le terme TeX est souvent utilisé à tord pour designer le format plainTEX est non le moteur.

L'exécutable `tex` lance le moteur et charge le format plainTEX. L'exécutable `latex` fait de même avec le moteur par défaut est le format LATEX.

L'ellipse du titre de cette sous-section ne manque pas de pertinence. Disons que tout ce qu'il y a sous le capot c'est TEX (le moteur). LATEX offre le luxe de bien s'installer et pour conduire on n'est pas tenu de savoir ce que déclenche l'action conjointe de la pédale d'embrayage et du levier de vitesse, pas plus que n'importe quelle action sur les autres mécanismes du poste de pilotage (le format).

Le développeur LATEX est le garagiste bidouilleur.

Une commande primitive TEX est une commande du moteur TEX. Elles sont écrites dans le même langage (**web** ou **C**) que le moteur. TEX compte quelque 300 commandes primitives (plus avec les extensions eTEX et autres). Ce sont les commandes primitives qui permettent d'écrire un format.

Une commande TEX est une commande primitive du moteur ou une

Les commandes non primitives sont aussi dites des macros.

commande du format plainTEX. Ces dernières sont écrites en langage TeX, utilisant les commandes primitives ou d'autres commandes plainTEX.

Une commande LATEX est une commande du format LATEX. Elles sont écrites en utilisant les commandes TEX ou d'autres commandes LATEX.

Dans LATEX, on peut distinguer entre trois familles de commandes :

`\makeatletter` et `\makeatother` sont risquées dans un fichier `.sty`… si on le charge avec `\usepackage`.

— Les commandes internes du format (et de ses packages) et dont le nom porte un caractère `@`. Elles ne sont pas destinées à une utilisation dans les documents à moins d'encadrer le bloc de code qui les contient entre `\makeatletter` et `\makeatother`. Elles sont utilisables sans précaution dans les fichiers de style et les fichiers de classe. Elles peuvent être modifiées au fil des versions, y compris au niveau de l'interface d'utilisation. On estime toutefois que les commandes internes du format LATEX sont moins sujet à des changements et certaines font désormais partie du kit « essentiel pour développeurs ».

Usuellement, les noms de macros ne peuvent contenir que des lettres.

— Les commandes dont les noms contiennent seulement des lettres mais avec des lettres majuscules comme

`\RequirePackage`, `\InputIfFileExists`, `\AtBeginDocument`…

sont des commandes plus pérennes et destinées au développement d'extensions.

— Les commandes utilisateurs. Leurs noms contiennent seulement des lettres et jamais de lettre majuscule. Ces commandes sont presque immuables (sauf correction de bogues).

LATEX ne possède aucune méthode propre pour subdiviser une boite verticale afin d'étaler son contenu sur plusieurs page, le mécanisme des leaders n'y est pas reproduit, la gestion des « token lists » est celle par défaut de (plain)TEX…

Outre ces trois catégories, les commandes TEX sont utilisables [1] sous LATEX et sont souvent qualifiées de commandes de bas niveau. Leurs utilisation est déconseillée dans les documents, mais elles deviennent vite indispensables à l'utilisateur avancé ou au développeur d'extensions. LATEX est avant tous orienté utilisateur et a pour but de minimiser les connaissances requises pour produire des documents typographiquement correct. En revanche, il n'offre pas d'accès à certaines fonctionnalités avancées du moteur.

Fichier de classe Un fichier de classe porte l'extension `.cls` et est chargé par les documents via la commande `\documentclass`. Il définit la structure du document, cela peut aller de la fixation des dimensions de la mise en page, en passant par le style des titres et des tables des matières ou encore de la façon de traiter les éléments flottants, à la manière dont doit être traité le document par le moteur à travers divers paramètres de configurations. Outre les fichiers de classe standards `article.cls`, `report.cls` et `book.cls`, d'autre extensions sont disponibles, telles que celle du pack Koma-Script qui fournit des alternatives aux classes standards, plus adaptées à la typographie européenne (respectivement `scrartcl`, `scrreprt` et

1. le fichier `tplain.dtx` qui fait partie de toutes les distribution LATEX est en fait une reprise de toutes les commandes et de la plupart des paramètres du format plainTEX.

scrbook), le fichier memoir.cls qui est une meta-classe permettant une personnalisation très poussée des documents ou encore la classe beamer.cls spécialisée dans la préparation de documents pour des présentations numériques. L'utilisateur peut former ses propres fichiers de classes. Il peut y charger un fichier .cls existant avec la commande \LoadClass et redéfinir les structures qui ne correspondent pas à ces besoins ou en ajouter de nouvelles.

Le fichier de documentation standard clsguide.pdf*, fourni avec toutes les distributions, décrit le nécessaire pour former un fichier de classe ou un package.*

Fichier d'extension un fichier d'extension ou « package » porte l'extension .sty et on le charge dans les documents avec la commande \usepackage. Un fichier d'extension adresse en général un aspect particulier et est utilisable avec n'importe quel fichier de classe. Il peut faire appel à d'autres packages en utilisant non pas \usepackage, mais la commande développeur \RequirePackage.

TEX N'EST PAS OBSOLÈTE

À cause de l'amalgame entre le moteur et le format associé plainTEX, l'idée selon laquelle « TeX est obsolète » est assez courante. En fait TEX n'a jamais été aussi vivant que pendant ces dernières années, où on a vu se profiler des déclinaisons très excitantes en remplacement du moteur historique ainsi que certaines extensions graphiques plus au moins intégrée au système comme PGF/Ti*k*Z, Asymptote ou encore la bibliothèque mplib qui accompagne LuaTEX. La plupart des évolutions apparentes qu'on peut attribuer au dynamisme de LATEX ne sont en fait que des répercussions ou des exploitations de celles des nouveaux moteurs. Paradoxalement le progré de LATEX est au point mort vu que le projet LATEX3 peine à se finaliser, là où conTEXt, l'autre format majeur, montre un enthousiasme à toute épreuve (il repose déjà entièrement dans sa version MK.IV sur le moteur LuaTEX).

QU'EST CE QU'IL A DE PLUS CE LATEX

LATEX propose une interface d'utilisation simple vers les fonctionnalités, parfois très complexes, du moteur TEX. Par rapport au format de base plainTEX, il offre

— Une structuration plus poussée des documents et des méthodes standards et sous contrôle pour charger des extensions ou des fichiers annexes.

— Des spécifications plus ou moins précises pour créer des fichiers d'extension qui permettent d'ajouter de nouvelles fonctionnailités ou de créer des classes pour certains types de documents.

— Un mécanisme standard de gestion des polices de caractère (**NFSS**).

— L'officialisation de la notion d'environnement, une homogénéisation de la syntaxe des commandes et un certain contrôle des noms donnés aux nouvelles macros et environnements afin d'éviter d'écraser d'éventuelles définitions préexistences.
— Une gestion des titres et un mécanisme pour leur intégration dans les entêtes/pieds de page. Une génération automatisée et configurable des tables des matières et aussi, à travers d'autres programmes, d'une bibliographie ou d'un index.
— Un mécanisme pour les références croisées.
— Des environnements pour les listes et la possibilité d'en définir et d'en configurer de nouveaux. Des environnement pour produire du contenu en mode « DISPLAY » tels que `center` ou `theorem`.
— Des enveloppes plus conviviales et plus « humaines » des commandes de mise en boite TEX (`\makebox`, `\framebox`, `\parbox`, `minipage`…).
— Une bien meilleure intégration des mécanismes du moteur pour la création de tableaux (il parait que c'est la partie la plus évoluée mise en place par LATEX).
— Une meilleure exploitation des capacités du moteur pour la gestion des éléments flottants : tables, figures, notes de marge et de bas de page.
— L'environnement basique de création de graphiques `picture`.

Tout est dans la boite !

Les objets de base manipulés par TEX sont essentiellement les boites. Cela peut aller des boites contenant de simples caractères, en passant par des boites qui contiennent des paragraphes complets voire des éléments graphiques, jusqu'à la boite qui contient tout le contenu d'une page. On distingue entre les

hbox ou boites horizontales, typiquement crées avec la commande primitive TEX `\hbox` ou avec des commandes LATEX telles que `\mbox` ou `\makebox`. Le contenu est produit sur une seule ligne sans rupture possible. Les commandes au fonctionnement vertical sont au mieux ignorées, au pire produisent des erreurs dans une **hbox**.

vbox ou boites verticales , typiquement crées avec les commande primitives `\vbox` ou `\vtop` (ou `\vcenter`, réservée au mode mathématique) ou encore la commande `\parbox` ou l'environnement `minipage` sous LATEX. Le contenu y est éclaté en lignes et les lignes empilées verticalement. Une **vbox** peut contenir plusieurs paragraphes. C'est une sorte de page réduite.

Pour chaque boite TEX, **vbox** ou **hbox**, on parle de

— la largeur de la boite, qui n'est pas forcément la largeur de son contenu mais celle déclarée dans l'instruction qui l'a crée.
— la hauteur de la boite, qui est sa hauteur à partir de la ligne de base ;
— la profondeur de la boite, qui est la distance de son point le plus bas à la ligne de base.

La hauteur et la profondeur d'une boite **vbox** dépendent de sa position sur la ligne de base. Elle ne sont pas forcément égales aux dimensions visibles de ce qu'elle produit. La position d'une **vbox** par rapport à la ligne de base dépend de la commande primitive utilisée pour sa construction : `\vbox`, `\vtop` ou `\vcenter`. Incidemment, cela dépend de l'option (`b`, `t` ou `c`) de `\parbox` et de `{minipage}` sous LaTeX, ces derniers utilisant les commandes primitives TeX en interne.

Une section complète de ce document est consacrée aux boites.

Suis le mode

Au fur et à mesure du traitement d'un document, le moteur TeX se trouve dans l'un des six états suivants :

— Le mode horizontal. Mode dans lequel TeX traite un paragraphe en l'éclatant en lignes. Une liste (dite liste horizontale), maintenue pour le paragraphe en cours, contient les boites (horizontales ou verticales) du paragraphe dont les boites des caractères, les sauts horizontaux implicites ou explicites, les filets verticaux `\vrule`, les informations de césusre, les pénalités …
— Le mode horizontal restreint. Celui à l'interieur d'une **hbox**. Ce mode est désigné par le terme **LR-mode** (left to right mode) dans le jargon LaTeX. Le contenu est produit sur une seule ligne.
— Le mode vertical. Dans ce mode TeX empile sur une liste (dite liste verticale) les`\hbox` des lignes des paragraphes traités par le mode horizontal, les sauts verticaux et les penalités.
— Le mode vertical interne. Celui en vigueur à l'intérieur d'une **vbox**.
— Les modes mathématiques, qui sont deux. Le mode dit **inlinemode** pour les formules en ligne et le mode dit **displaymode** pour les formules centrées.

Les modes horizontal restreint, vertical interne et **inlinmode** sont dit les modes internes. Les listes des modes horizontal non restreint et vertical externe sont dites listes principales.

Les sauts horizontaux ou verticaux sont des instructions insérées dans la liste active explicitement par l'utilisateur ou implicitement par le moteur TEX ou les macros TEX/LATEX. Dans la plupart des situations, les sauts sont des éléments résiliables dans le sens où ils peuvent disparaître dans le processus de formation des lignes ou de la page.

Les penalités sont des instructions faisant intervenir des variables numériques entièresspécialisées et qui fixent, chacune pour un aspect particulier, le degré de flexibilité dans la rupture d'une ligne (dans le mode horizontal) ou de la page (dans le mode vertical). Les instructions de pénalité sont en général insérées implicitement à divers endroits dans la liste active, par le moteur TEX ou les macros TEX/LATEX. Les pénaliés sont résiliables aussi.

Dans le mode vertical TEX stocke dans la liste verticale tous les sauts verticaux, les pénalités verticales et les **hbox** des lignes des paragraphes traités par le mode horizontal… et c'est (presque) tout. Le contenu d'une liste horizontale principale est plus varié : TEX y intégre tout le contenu d'un paragraphe, ce qui inclut

— toutes les boites **hbox** ou **vbox** du paragraphe, y compris les boites des caractères avec leurs ligatures éventuelles ;
— les sauts horizontaux y compris les espaces inter-mots ;
— Les pénalités horizontales ;
— les informations de césures de tous les mots du paragraphe ;
— d'autres informations plus mystérieuses.

C'est après avoir formé la liste d'un paragraphe en entier que TEX cherche, selon des algorithmes variés, le meilleur choix pour les points de rupture des lignes. Le but de ces algorithmes est de mesurer la qualité typographique de chaque possibilité avant de faire un choix définitif. Cela dépend des pénalités dans la liste et de la notion de **badness** qui mesure pour chaque possibilité de ligne le rapport entre les vides naturels dans la ligne et ceux avec les espaces ajoutés pour produire un texte justifié. Certains critères esthétiques et typographiques doivent être respectés comme : éviter autant que possible de former deux lignes adjascentes avec des vides visuellement incompatibles ; deux lignes successives qui se terminent toutes les deux avec des césures ; que l'avant-dernière ligne du paragraphe ne se termine avec une césure… C'est ainsi que des petites interventions comme changer quelques mots dans le paragraphe ; y placer judicieusement des espaces insécables (avec la commande ~) ou parfois aider TEX en plaçant explicitement une césure qu'il n'a pas pris en compte (avec la commande \-)… peuvent complétement modifier la configuration des lignes dans le paragraphe. À titre d'exemple, il n'est pas rare de voir TEX produire moins

de lignes pour un paragraphe alors qu'on vient d'y ajouter quelques mots (si les vides dans le paragraphe d'origine étaient trop étirés).

Dans le mode vertical principal, TEX maintient en fait deux listes, une liste « contributions récentes » et une liste « page courante ». La liste « contributions récentes » est continuellement alimentée et à des moments cruciaux (un paragraphe traité, une équation **displaymode** achevée…), il va en prélever du matériel qu'il va ajouter à la queue de la liste « page courante » et tester s'il est possible de former convenablement une page avec le les éléments ainsi cumulés dans cette deuxième liste. Une fois ce but atteint, le contenu de la liste « page courante » est envoyé à la routine de formation de pages et la liste est vidée pour une nouvelle cession. TEX prend aussi soin, à ce moment, d'ôter tous les éléments résiliales du sommet de la liste « contributions récentes ».

Le problème de formation des pages est ***LP-Complet****.*

Les algorithmes de formation de lignes sont plus complexes que ceux de formation de pages. Les premiers prennent en compte toutes les lignes d'un paragraphe alors qu'avec les seconds, seul un contenu approximatif d'une page est analysé. Le mode vertical ne dispose en outre pas d'un équivalent de la flexibilité induite par les césures en fin de ligne. On comprend alors que dans un document TEX/LATEX assez long, on a plus de soucis avec la mise en page verticale que celle horizontale, sachant qu'en plus, la seconde influe sur la première et que le contraire n'est pas forcément vrai.

Avec `\raggedbottom`*, le texte « n'occupe sa hauteur naturelle » qu'en apparence. En fait un saut infini est placé en fin de page. Un constat similaire tient pour les lignes avec* `\raggedright` *et* `\raggedleft`*.*

Que ce soit pour la formation d'une page ou d'une ligne, une fois le point de rupture choisi tous les espaces élastiques dans la liste sont étirés pour que le contenu occupe toute la page ou toute la ligne à moins que le contexte indiqué par l'utilisateur n'impose que ces espaces prennent leurs dimensions naturelles. Pour les pages , cela peut être effectuée grâce à la commande `\raggedbottom` [2] placée quelque part au début du document, et pour les lignes des commandes telles que `\raggedright` ou `\raggedleft` ont pour conséquences que les vides dans la lignes prennent leurs dimensions naturelles. Quelque soit le mode, TEX veille à ce qu'aucun élément résiliable, saut ou pénalité, ne soit placé au sommet d'une liste.

LA FORMATION DES PAGES EST ASYNCHRONE

Avec la méthode suivie pour former des pages, TEX ne sait pas, au moment de lire du contenu dans quelle page il va finir. Ce détail pose un problème de programmation dans certains cas. Par exemple, dans un document recto-verso, LATEX place les notes de marge dans le côté intérieur ou extérieur de la page selon que son numéro est paire ou impaire, et

2. De plus en plus utilisée comme reglage par défaut dans les distributions TEX/LATEX.

parfois une note est placée sur le mauvais côté en début de page. Une technique courante pour remèdier à ce genre de problème est d'utiliser le mècanisme des références croisées de LaTeX : on place un `\label` et on vérifie ensuite son numéro de page, ce qui peut nécessiter deux compilations avant que tout prenne sa place.

Quelques bases, sans plus

Des macros pour commander

Une commande est soit une commande primitive soit une macro-commande.

Le concept de macro-commande, ou macro en plus court, est au centre de la remarquable capacité qu'a le système TEX/LATEX de se voir ajouter des extensions.

Les méhodes TEX et LATEX

Quelque soit la méthode utilisée, pour définir une macro on utilise toujours.

- un nom qui commence avec le caractère \ et qui n'est formé que de lettres, mais qui peut aussi être formé d'un seul caractère « actif » sans être forcément une lettre, comme la commande ~ ;
- zéro ou au plus neuf paramètres de position, avec une syntaxe qui diffère entre les primitives TEX et les mécansimes LATEX ;
- un texte de définition qui comprend éventuellement les références aux paramètres de position sous la forme #1, #2… texte qui doit être obligatoirement encadré par les caractères { et }.

Développer, expand en anglais, est la façon TEX de dire « évaluer ». L'« expansion » est ce qui rend le langage TEX si particulier et si difficile à réellement maîtriser.

Selon le mécanisme utilisé pour sa création, le texte de définition d'une macro est développé au moment même de la définition, ou bien au moment de son utilisation. Quand elle admet des paramètres, une macro va substituer chaque argument au paramètre de même position pour former ce qu'on appelle son texte de remplacement. Quand elle n'a pas de paramètres son texte de remplacement est le même que son texte de définition ou bien le résultat du développement de ce dernier selon la méthode utilisée pour sa création.

On parle de paramètres quand on définit une macro, mais d'arguments quand on l'utilise

> Encadrer un argument entre des accolades lorsqu'on utilise une commande n'est nécessaire que s'il contient plus qu'un caractère. L'instruction `\frac{1}{2}` est parfaitement équivalente à `\frac 12` et même à `\frac12` car après un caractère \ qui commence le nom d'une commande, le premier caractère qui n'est pas une lettre signale la fin du nom de la commande.
> En fait, une macro va piocher autant que nécessaire ses arguments dans la pile qui va suivre sachant que du texte encadré avec des accolades est pris comme un lot indivisible y compris lorsqu'il n'y a rien entre les accolades. En général, les caractères espaces entre les arguments sont ignorés dans ce processus.

Le moteur TEX dispose d'un choix varié de primitives pour définir de nouvelles macros. Il s'agit des commandes `\def`, `\gdef`, `\edef` et `\xdef`. Les quatres s'utilisent de la même façon, mais elle répondent à des besoins différents.

— Avec `\def` et `\gdef`, le texte de défintion d'une macro est conservé tel quel et n'est developpé qu'au moment où elle est utilisée, chaque paramètre étant remplacé par l'argument de même position.

Développer une macro c'est dévelpper son texte de remplacement après absorption des arguments nécessaires.

— Une macro définie avec `\edef` ou `\xdef` ne peut pas avoir des paramètres et le texte utilisé dans sa définition est complètement développé au moment même de cette définition. C'est le résultat de ce développement qui est utilisé comme texte de remplacement de la macro.

De base, un groupe TEX est un bloc de texte encadré entre deux accolades sauf dans le cas où les accolades servent à encadrer des arguments.

La différence entre `\def` et `\gdef` réside dans le fait qu'une commande définie avec `\def` au sein d'un groupe est oubliée quand on sort du groupe, alors qu'une macro définie avec `\gdef` l'est globalement dans le sens qu'elle survit même si sa définition se fait dans un groupe. De la même façon, `\xdef` est la version « globalisante » de `\edef`.

Toutes ces primitives sont parfaitement utilisables sous LATEX bien que celui-ci ajoute ses propres méthodes. La commande LATEX `\newcommand` est une version plus polie et plus sécuritaire que la primitive `\def`. Il n'y a pas de mécanisme propre à LATEX pour définir des macros dont le texte de remplacement est développé à la défintion et pour définir globalement une macro il suffit de faire précéder `\newcommand` par la commande `\global`. D'ailleurs `\gdef` est l'équivalent de `\global\def`.

Pour faire simple et sans perdre en généralité, la définition avec `\def` d'une macro `\mymac` qui prend deux paramètres ressemble à

`\def\mymac#1#2{`⟨*defintion text*⟩`}`

les paramètres #1 et #2 étant aussi utilisés (ou pas) dans ⟨*defintion text*⟩. Avec exactement le même texte de définition, \newcommand permet de définir la macro \mycom qui sera parfaitement équivalente à \mymac avec

```
\newcommand\mycom[2]{⟨defintion text⟩}
```

\newcommand vérifie si le nom \mycom n'est pas déjà utilisé par le système, auquel cas elle provoque une erreur. Les primitives de la famille \def se dispensent d'une telle vérification, ce qui rend leurs utilisation potentiellement dangereuse vu le risque d'écraser la définiton d'une commande indispensable au système. Par ailleurs \newcommand permet de rendre optionnel un, et seulement un, des paramètres de \mycom. Ce sera toujours le paramètre #1 et on indique sa valeur par défaut avec la syntaxe

```
\newcommand\mycom[2][⟨default arg⟩]{⟨defintion text⟩}
```

On utilisera ensuite \mycom[⟨*opt arg*⟩]{⟨*mand arg*⟩} pour que ⟨*opti arg*⟩ et ⟨*mand arg*⟩ se substituent respectivement à #1 et à #2 dans le texte de défintion de \mycom, ou simplement \mycom{⟨*mand arg*⟩} pour que #1 soit remplacé par la valeur par défaut ⟨*default arg*⟩ et #2 par ⟨*mand arg*⟩.

Le concept de macro-raccourci peut être poussé très loin. Il est ainsi possible de définir une macro qui ne fait qu'appeler une autre macro sachant que la seconde prend bien plus de paramètres. À l'utilisation, la macro ainsi définie consommera plus d'arguments que sa définition n'a de paramètres, ces arguments étant en fait utilisés par la commande interne.

```
\newcommand\oneandhalf{\makebox[1.5\width]}
\fbox{\oneandhalf{Texte}}
```

Texte

Commentaires : \oneandhalf se contente d'insérer \makebox[1.5\width] laissant la commande \makebox puiser le reste de ces arguments dans le flux de texte suivant. En apparence \oneandhalf a besoin d'au moins un argument alors que sa définition ne déclare aucun paramètre.

Les commandes TEX ne peuvent créer directement des macros qui ont des paramètres optionnels, mais elles disposent d'un mécanisme autrement plus primitif et très puissant : la notion de délimiteur de paramètres. Par exemple, dans la définition

```
\def\fraction#1/#2{\frac{#1}{#2}}
```

la macro \fraction utilise le caractère / pour séparer ses deux paramètres.

```
$\fraction 1/2$, $\fraction{a+b}/d$
```

$\frac{1}{2}$, $\frac{a+b}{d}$

C'est dans la partie réservée aux paramètres qu'on précise les délimiteurs désirés. N'importe quelle chaine de caractère peut servir de délimiteur, à condition qu'elle ne contienne pas le caractère { ou } (qui annoncent le début ou la fin du texte de définition), mais elle peut contenir des « noms de macros » sans que celles-ci soient forcément définies. Lors de l'utilisation d'une macro qui utilise des délimiteurs, TEX provoque une erreur s'il n'arrive pas à construire une séquence qui correspond au prototype fourni par la zone de paramètres. De plus, puisqu'un bloc de texte encadré entre accolades est un noeud indivisible, toute occurence d'un éventuel délimiteur entre { et } est ignorée. Un caractère espace est significatif dans la zone de paramètres et sert de délimiteur, mais lors de la lecture des arguments tous espace qui ne correspond pas à un délimiteur est ignoré.

Un exemple moins artificiel maintenant.

```
\def\otesmoica#1--#2--#3.{#1 #3.}
\otesmoica ou comment \^oter ces ellipses --
 parfois inutiles -- du texte final.\par
```

ou comment ôter ces ellipses du texte final.

Pas question de délimiter

Une commande LATEX lit ses arguments l'un après l'autre dans l'ordre dans lequel ils se présentent. C'est ainsi que l'emploi des délimiteurs de paramètres est contraire à l'esprit de LATEX pour ce qui est des commandes utilisateurs car ils empêchent l'harmonisation de leur syntaxe. La commande \newcommand ne permet d'ailleurs pas de le faire. Toutefois, LATEX ne se prive pas lui même de l'utilisation de ces mécanismes en interne.

Outre la commande \global, d'autres servent à préfixer les commandes de création de macros pour changer le comportement de ces dernières. L'une d'elle est \long. Par défaut une commande créée avec \def ne permet pas que l'un de ses arguments contienne des lignes vides (ou la commande \par). Préfixer \def par \long permet ensuite à la macro ainsi créée de gérer ce genre d'arguments. La commande \newcommand est équivalente en fait à \long\def, mais \newcommand* est équivalente à \def.

TP **Le @titre de cet @auteur paru à cette @date** • Les commandes \title, \author et \date permettent de renseigner respectivement le titre du document, son auteur et une date. Elles sont définies par

```
\def\title#1{\gdef\@title{#1}}
\def\@title{\@latex@error{No \noexpand\title given}\@ehc}
\def\author#1{\gdef\@author{#1}}
\def\@author{\@latex@warning@no@line{No \noexpand\author
 given}}
\def\date#1{\gdef\@date{#1}}
\gdef\@date{\today}
```

Ainsi, la commande `\title` se contente de faire de son argument le texte de remplacement de la macro interne `\@title`. Les commandes `\author` et `\date` font la même chose avec `\@author` et `\@date`. L'usage de la commande `\gdef` assure que ces macros internes persistent même si les commandes de haut niveau sont saisies dans un groupe. La commande `\maketitle` utilise ensuite les macros ainsi définies pour produire le titre du document. Les définitions par défaut des macros `\@title` et `\@author` font qu'elles produisent des message d'erreur une fois lancées par `\maketitle`, si les macros `\title` et `\author` ne sont pas utilisées avant.

Cette technique peut être utilisée par un fichier d'extension pour que des informations s'affichent dans diverses parties du document (la zone des entêtes par exemple) sachant que les dites informations ne seront disponibles que lorsque l'utilisateur les renseignera.

TP **Zapper une partie d'un document** • Comme application de la notion de délimiteur, nous allons définir la commande `\zaptext` qui permet de ne par produire une partie du texte source dans le fichier compilé. En fait c'est élémentaire

```
\long\def\zaptext#1\endzap{}
```

Ici `\endzap` sert de délimiteur à l'unique paramètre de la commande `\zaptext` et n'admet aucune autre signification. La conséquence de cette définiton est que tout le texte entre `\zaptext` et `\endzap` est remplacé par « rien ». Le préfixe `\long` permet à `\zaptext` d'agir sur du texte qui comprend éventuellement des lignes vides.

```
\long\def\zaptext#1\endzap{}
Du texte \zaptext avec une indication en moins
 \endzap amoindri.
```

Du texte amoindri.

Conventions de nomenclature

Comme l'emploi des primitives TEX de définition de nouvelle macros est inévitable dans un fichier d'extension LATEX, `.cls` ou `.sty`, il est de convention de suivre les consignes suivantes

— Choisir un nom de domaine qui va préfixer toutes les commandes de bas niveau. Par exemple dans un hypothétique fichier `foobar.sty`, utiliser des noms qui commence avec `foo@`. Cela permet de minimiser le risque de conflit avec la multitude de commandes internes des autres extensions.
— Seules les commandes utilisateurs finales devraient porter des noms sans préfixes et n'utiliser que des noms assez évocateurs composés uniquement de lettres minuscules.

Ce qui amène à se questionner sur le statut ambigu que représente ces commandes avec le caractère `@`. Dans les documents finaux, le caractère `@` n'est pas une lettre et donc la construction `\foo@truc` est lue par TEX comme étant la commande `\foo` suivie des caractères `@truc`. Par contre dans un fichier d'extension, ce caractère est mué par la magie des « **catcode** » [1] en une lettre normale et `\foo@truc` devient un nom de commande valide. Ce sont les commandes `\usepackage` et `\documentclass` (mais pas `\input` ou `\include`) qui s'occupent de cette transformation lorsqu'il lisent les fichiers correspondants. Ce mécanisme est disponible à l'usage dans les documents finaux à travers les commandes `\makeatletter` et `\makeatother`, littéralement transformer `@` en une lettre et en « autre chose ».

Copier pour faire mieux que l'original

Outre les commandes qui définissent de nouvelles macros, TEX offre à travers la primitive `\let`, la possibilité de créer une copie exacte sous un autre nom d'une macro déjà définie. Cette copie devient autonome et changer la version originale ne l'affecte pas. `\let` ne peut être utilisée pour créer une copie d'une macro définie avec `\newcommand` et ayant un paramètre optionnel.

```
\def\texte{Ancien texte}
\let\Texte=\texte \def\texte{Nouveau texte}
\texte. \Texte.
```

Nouveau texte. Ancien texte.

Commentaires : Le symbole = utilisé ici avec `\let` est optionnel.

L'exemple suivant est plus intéressant dans la mesure où il montre comment utiliser `\let` pour modifier le comportement d'une commande du système.

1. Voir le chapitre « Fondement »

```
\let\oldtextbf\textbf
\renewcommand\textbf[1]{\fbox{\oldtextbf{#1}}}
\textbf{Texte.} \oldtextbf{Texte.}
```

Texte. **Texte.**

COMMENTAIRES : Technique très utile pour modifier le comportement d'une commande en gardant le même nom. On en crée une copie avec `\let` et on utilise cette copie pour la redéfinir.

LaTeX utilise beaucoup cette dernière technique. Il sauvegarde par exemple la version originale de la primitive `\par` avec `\let\@@par\par` et modifie la signification de `\par` dans la plupart des environnements qu'il crée.

`\let` peut être aussi utilisée pour créer un synonyme d'un caractère donné. Le format plainTeX définie par exemple la commande `\bgroup` par un simple `\let\bgroup={`, la commande `\bgroup` peut ainsi être utilisée dans la plupart des situations où `{` est utilisé, pour encadrer des arguments ou délimiter un groupe par exemple.

AVOIR LA MESURE

SAUTS ET DIMENSIONS

Faisons le point sur les dimensions sous TeX. Elles sont de deux sortes, les dimensions rigides, qu'on va noter **dimen**, et celles élastiques (ou ressort) et qu'on va noter **skip**. Une dimension rigide n'a qu'une composante avec une unité de mesure. Une dimension élastique en a (deux ou) trois, toujours accompagnées d'unités de mesures. La première est la composante naturelle, décidant de la dimension à appliquer. Les autres composantes qu'on fait précéder des mots clés **plus** et/ou **minus** sont des mesures d'élasticité constituant respectivement le maximum à retrancher ou à ajouter à la dimension naturelle dans le cas où TeX a besoin de remplir une boite en étirant ou en compressant les vides qu'elles contient. Les **dimen** sont utilisées un peu partout : pour mesurer des boites ou de certaines zones du document, pour tracer des filets de dimensions fixes, pour les espaces de crénages (des sauts fixes)… Les ressorts par contre ne sont utilisés que pour effectuer des sauts élastiques, qu'ils soient verticaux ou horizontaux. Signalons que

> les espaces inter-mots, ainsi que les espaces interlignes et inter-paragraphes sont élastiques par nature.

Pour les unités de mesure, outre le centimètre (cm) et le millimètre (mm), celles qui sont les plus utilisées dans TeX sont décrites dans le tableau 2.1. TeX dispose aussi des unités de mesures « dynamiques »`ex` et `em`. Elles

point	`pt`	point typographique TEX	
inch	`in`	pouce	`1in=72.27pt=2.54cm`
pica	`pc`	pica	`1pc=12pt`
big point	`bp`	grand point, notamment utilisé par le langage **PostScript** et le format **PDF**.	`1in=72bp`

Table 2.1 Unités de mesure courantes.

sont mises à jour à chaque changement effectué sur la police de caractère : `1ex` est la hauteur du caractère `x` et `1em` est (tenue pour être) la largeur du caractère `M`. Les deux sont en général récupérée à partir des fichiers de la police elle-même. Toutes ces unités sont converties en interne dans l'unité « scaled point » propre à TEX. Sa valeur est décidée par la relation

$$1\text{pt} = 65\,536\,\text{sp} = 2^{16}\,\text{sp}$$

*Il n'est pas vraiment correct de parler de « macro » de dimension. Les entités qu'on crée afin de contenir des dimensions sont en fait des alias de ce qu'on appelle « registre de dimension ». Ces registres sont en nombre limité, 256 pour chaque catégorie, **dimen** et **skip**.*

Une variable de dimension est une « macro » qui sert à contenir des dimensions. On déclare une nouvelle variable de dimension TEX avec `\newdimen` pour les dimensions rigides et `\newskip` pour les ressorts. On affecte une valeur à une variable de dimension TEX en utilisant l'opérateur = ou simplement en faisant suivre son nom de la valeur voulue.

```
\newdimen\mydimen  \mydimen = 3em
\newskip\myskip  \myskip = 3em plus 4pt minus 2pt
```

> Si un **skip** comporte une composante **plus**, elle doit toujours figurer avant la composante **minus**. Si on affecte à une variable **dimen** la valeur explicite d'un **skip**, seule la composante naturelle est absorbée en argument et le reste est produit dans le texte final. Par contre si on lui affecte une variable ressort, la valeur reçue est toujours la composante naturelle mais rien n'est produit dans le document compilé. Si à l'inverse, on affecte à une variable **skip** la valeur d'une **dimen**, le **skip** conserve sa nature mais tout saut l'utilisant ne peut ni rétrécir ni s'étirer.

```
\newdimen\mydimen \newskip\myskip
\myskip=4pt plus 2pt minus 2pt
Une assignation explicite \og\mydimen=1cm plus
 2mm\fg et une autre \mydimen=\myskip qui ne
 laisse pas de trace mais qui ne change pas la
 nature de la dimension : \the\mydimen
```

Une assignation explicite « plus 2mm » et une autre qui ne laisse pas de trace mais qui ne change pas la nature de la dimension : 4.0pt

LaTeX a ces propres mécanismes pour déclarer et modifier les variables de dimension mais se limite aux variables de type ressort. Ces variables sont déclarées avec la commande `\newlength` qui utilise en interne `\newskip`. Comme d'habitude, LaTeX vérifie si le nom donné à la variable est disponible. Il fournit la commande `\setlength` pour affecter une valeur à une variable de dimension et `\addtolength` pour lui ajouter une autre valeur. Mais comme les variables LaTeX sont declarées en interne avec `\newskip`, rien n'empêche d'utiliser la syntaxe TeX pour les altérer.

cerbere va en souffrir

```
\newlength\mamesure
\setlength\mamesure{3em plus 4pt minus 2pt}
```

Le nom d'une variable ne peut être utilisé en lui même (comme celui d'une macro), il sert en général comme argument à d'autres commandes. Pour afficher les valeurs de variables numériques dans le document compilé, on peut utiliser les commandes primitives :

`\number` commande qui prend en argument un nombre explicite ou une variable numérique et la convertit en un nombre dans le système décimal. Utilisée avec une dimension, elle retourne une valeur sans unité (convertie en sp : 1pt = 2^{16} sp), sans aucune indication sur les composantes d'élasticité s'il s'agit d'un ressort.

TEX gère la représentation des nombres en décimal, en octal et en hexadécimal.

`\the` commande qui peut, entre autre chose, prendre en argument une variable numérique et afficher la chaîne de caractères qui la représente. Pour les dimensions elle donne une représentation plus fidèle de son type (**dimen** ou **skip**).

```
N : \number\bigskipamount \\
D : \the\bigskipamount
```

N : 786432
D : 12.0pt plus 4.0pt minus 4.0pt

> Il est important de savoir que pour une expression de la forme ⟨*facteur*⟩`\myskip`, où ⟨*facteur*⟩ est un nombre, entier ou flottant, et `\myskip` un **skip** déclaré, le résultat n'est pas un **skip** mais une **dimen**, les composantes d'élasticité de `\myskip` n'étant pas utilisées. La raison en est que dans cette situation `\myskip` est traité comme une unité de mésure. D'ailleurs toute expression de la forme ⟨*facteur*⟩`\var` est une dimension valide dès que `\var` est une variable numérique quelconque. Si c'est une variable sans unité alors l'unité **sp** est adoptée.

```
\the\year\\
\newdimen\mydimen \mydimen=100\year
\number\mydimen\quad\the\mydimen
```

2020
202000 3.08228pt

Commentaires : `\year` est une variable qui contient l'année en cours, `\mydimen` est calculé en **sp** et `\the` affiche sa valeur en **pt**, ce qui indique qu'elle bien une dimension.

```
\myskip=1\bigskipamount\relax \the\myskip
```

12.0pt

Commentaires : L'utilisation de l'expression `1\bigskipamount` implique effectivement une coercition du résultat en une **dimen** bien que `\myskip` et `\bigskipamount` soient toutes les deux des variables **skip**.

La séquence `\vskip2\bigskipamount` est ainsi différente de `\bigskip\bigskip`. La première insère un saut rigide de 24pt dans la liste verticale alors que la deuxième insère un saut élastique dont la valeur est de deux fois `\bigskipamount`.

TEX fournit les commandes primitives `\vskip` et `\hskip` pour effectuer des sauts, lesquels sont résiliables par défaut. Le format plainTEX définit les commandes `\vglue` et `\hglue` qui produisent des sauts non résiliables. Il définit aussi les commandes `\vfil`, `\vfill` et leurs versions horizontales pour effectuer des sauts de composantes naturelles nulles mais qui peuvent s'étendre à l'infini. LATEX définit diverses commandes au fonctionnement plus subtile : `\hspace` pour les espaces horizontaux, `\vspace` et `\addvspace` pour les espaces verticaux. Toutes ces commandes sont décrites avec plus de détails dans les sections suivantes.

Sauter autrement

À côté des sauts qui sont souvent élastiques, il y'a les espaces de crénage qui sont des sauts rigides et qu'on va noter **kern**. On les insère dans la liste en cours avec la commande `\kern`. Dans un contexte vertical `\kern` produit un saut vertical et dans un contexte horizontal, un saut horizontal.

> Un espace de crénage diffère d'un saut normal dans le fait qu'il n'est pas élastique et qu'une liste n'est jamais rompue au niveau d'un **kern** s'il est encadré par deux éléments non résiliables.

1*em*=18*mu*

En outre, le mode mathématique possède ses propres dimensions ainsi que des commandes spécifiques pour gérer plus finement les espaces dans les formules. Ce sont les **muskip** et les **mukern**. L'unité de mesure spécifique **mu** ne peut être utilisée en dehors du mode mathématique. Les commandes `\mskip` et `\mkern`, similaires dans leurs fonctionnement à `\hskip` et `\kern`, ne peuvent être utilisées que dans ce mode et seulement avec l'unité **mu**. Rien n'empêche par contre d'utiliser les commandes d'espacement horizontal habituelles dans le mode mathématique. Le tableau 2.2 montre comment sont définies les commandes d'espacement du mode mathématique.

Commande	Définition	Valeur par défaut	Utilité
`\,`	`\mskip\thinmuskip`	`\thinmuskip=3mu`	Espace placé implicitement après une ponctuation. Espace rigide
`\!`	`\mskip-\thinmuskip`	`\thinmuskip=3mu`	
`\:`	`\mskip\medmuskip`	`\medmuskip=4mu plus2mu minus4mu`	Espace placé implicitement autour du symbole d'un opérateur binaire. Peut s'étirer de la moitie de sa valeur ou complètement disparaître.
`\;`	`\mskip\thickmuskip`	`\thickmuskip=5mu plus5mu`	Espace placé autour d'un symbole de relation. Peut aller du simple au double sans jamais diminuer.
`\quad`	`\hskip1em`		Commande d'espacement qui fonctionne aussi en dehors du mode mathématique.
`\qquad`	`\hskip2em`		Commande d'espacement.
`\enskip`	`\hskip.5em`		Commande d'espacement.
		`\mathsurround=0pt`	**kern** placé de chaque côté d'une formule **inline-mode**, ce qui fait que du contenu placé immédiatement (sans aucun espace) avant ou après une formule **inlinmode** n'est jamais séparé de la formule.
			Si **amsmath** est chargé, toutes les commandes d'espacement du mode mathématiquer fonctionnent aussi en dehors de ce mode.

Table 2.2 Espaces horizontaux du mode mathématique

Les sauts **muskip** obéissent aux mêmes règles que les ressorts normaux, à l'exception du fait qu'à l'intérieur d'un groupe TEX, les composantes d'élasticité des **muskip** ne sont pas utilisées.

```
\hbox to 5cm {$a+b+c+d$}
\hbox to 5cm {$a+{b+c}+d$}
\hbox to 5cm {a {b c} d}
```

$a \quad + \quad b \quad + \quad c \quad + \quad d$
$a \quad + \quad b+c \quad + \quad d$
a b c d

L'exemple suivant

```
$a+b$ \qquad $a{+}b$
```

$a+b \qquad a+b$

n'est qu'en apparence similaire au précédent. La disparition des espaces autour du symbole + est ici due au fait TEX ajoute des espaces autour des opérateurs binaires seulement s'ils sont dans un contexte binaire. Dans le cas contraire, ils sont traités comme des caractères normaux.

```
$a-b$ \quad $-b$ \quad ${}-b$
```

$a-b \quad -b \quad -b$

Compter les moutons, sans s'endormir

Les compteurs représentent un autre type de variable numérique sous TEX. Une variable compteur est crée sous TEX avec la commande \newcount et on peut lui affecter un nombre, exclusivement entier et sans unité, de la même façon que pour une variable de dimension.

```
\newcount\mycounter \mycounter10
\the\mycounter
```

10

Les compteurs LATEX sont enveloppés dans une couche d'abstraction dont le but est d'établir des régles d'interdépendance et de donner une portée globale à toutes les opérations qui les concerne. On crée un compteur LATEX avec la commande \newcounter. À la différence des compteurs TEX (déclarés avec \newcount), le nom d'un compteur LATEX (déclaré avec \newcounter) ne commence pas le caractère \.

Comme pour les macros, les noms des variables numériques TEX sont des « control sequence ».

Une fois un compteur LATEX ⟨*compt*⟩ crée on peut :

— lui assigner une valeur entière avec \setcounter{⟨*compt*⟩}{⟨*val*⟩} ;
— l'incrémenter d'une unité avec \stepcounter{⟨*compt*⟩} ;
— lui ajouter une constante entière avec \addtocounter{⟨*compt*⟩}{⟨*val*⟩}.

On peut produire la valeur d'un compteur dans le texte avec différents styles en utilisant les commandes \arabic (chiffres arabes), \alph (lettres latines en petite casse), \Alph (lettres latines en grandes casse), \roman (chiffres romains en petites casse) ou \Roman (chiffres romains en grande casse). Pour chaque compteur déclaré ⟨*compt*⟩, LATEX définit systématiquement une commande \the⟨*compt*⟩ (à ne surtout par confondre avec la primitive \the) qui restitue sa valeur dans le style \arabic. On peut redéfinir cette commande pour qu'elle affiche la valeur de ⟨*compt*⟩ dans le style voulu en y incluant si voulu des références à d'autres compteurs ou bien en y ajoutant des effets.

```
\newcounter{mouton} \setcounter{mouton}{3}
\arabic{mouton} \Alph{mouton} \roman{mouton}\\
\themouton\addtocounter{mouton}{5} \themouton\\
\renewcommand\themouton{\fbox{\Alph{mouton}}}
\themouton
```

3 C iii
3 8
H

Lorsque on déclare un compteur ⟨*compt*⟩, LATEX crée en interne un compteur TEX de nom \c@⟨*compt*⟩. À cause du caractère @, ce nom n'est pas directement utilisable dans les documents, mais l'instruction \value{⟨*compt*⟩} le reconstitue et peut être utilisée sans restriction.

```
\the\value{mouton}\\
\value{mouton}=2014 \themouton
```

8
2014

La plupart des compteurs LATEX ont des noms déduits des mécanismes qui les utilise, `section` *pour la commande* `\section`*,* `equation` *pour l'environnement* `equation`*...*

LATEX utilise les compteurs surtout pour maintenir une numérotaion de certains éléments (pages, titres, items d'une liste, équations, théorèmes...), mais aussi dans les références croisées (c'est une autre histoire). Les compteurs TEX sont par contre des variables généralistes et en dehors des variables de dimensions, ils sont utilisés partout où TEX a besoin de quantifier numériquement certains aspect de son fonctionnement. Les variables de pénalités sont par exemples des compteurs.

Petite leçon d'arithmétique

On peut faire de l'arithmetique sur les variables numeriques TEX avec les commandes primitives `\advance`, `\multiply` et `\divide`. Si \⟨*var*⟩ est une variable numérique (compteur ou dimension) et ⟨*valeur*⟩ est une valeur numŕique qui peut être aussi une variable alors

`\advance`\⟨*var*⟩ ⟨*valeur*⟩ change la valeur de \⟨*var*⟩ en lui ajoutant ⟨*valeur*⟩. \⟨*var*⟩ et ⟨*valeur*⟩ doivent être de même nature, des dimensions ou des compteurs. Si \⟨*var*⟩ est un ressort l'ajout se fait respectivement sur ses trois composantes en utilisant les composantes de ⟨*valeur*⟩, celles d'élasticité de ⟨*var*⟩ étant considérées comme nulle si elle est une **dimen**.

`\multiply`\⟨*var*⟩ ⟨*valeur*⟩ et `\divide`\⟨*var*⟩ ⟨*valeur*⟩ changent la valeur de \⟨*var*⟩ respectivement en la multipliant ou en la divisant par ⟨*valeur*⟩. ⟨*valeur*⟩ doit être obligatoirement un entier. Si \⟨*var*⟩ est un ressort, chacune de ses composantes subit l'opération. Si \⟨*var*⟩ est un compteur le résultat d'une division est le quotient entier de l'opération.

Considérons les déclarations

```
\newskip\myskip \myskip\bigskipamount
\newdimen\mydimen \mydimen 100pt
\newcount\mycount \mycount\year
```

Variables qu'on va modifier en utilisant les commandes précédentes. Noter que chacun des exemples est traité dans un groupe et donc les variables retrouvent à chaque fois leurs états initiaux.

```
myskip: \the\myskip\advance\myskip\bigskipamount\\
myskip: \the\myskip
```

myskip : 12.0pt plus 4.0pt minus 4.0pt
myskip : 24.0pt plus 8.0pt minus 8.0pt

```
myskip: \the\myskip \\ mydimen: \the\mydimen \\
\advance\myskip\mydimen\relax myskip: \the\myskip
```

myskip : 12.0pt plus 4.0pt minus 4.0pt
mydimen : 100.0pt
myskip : 112.0pt plus 4.0pt minus 4.0pt

```
myskip: \the\myskip \multiply\myskip2 \\
myskip: \the\myskip \divide\myskip 16\relax\\
myskip: \the\myskip
```

myskip : 12.0pt plus 4.0pt minus 4.0pt
myskip : 24.0pt plus 8.0pt minus 8.0pt
myskip : 1.5pt plus 0.5pt minus 0.5pt

```
mycount: \the\mycount \multiply\mycount2 \\
mycount: \the\mycount \divide\mycount5 \\
mycount: \the\mycount
```

mycount : 2020
mycount : 4040
mycount : 808

Les primitives eTEX Le moteur eTEX, et donc toutes les declinaisons modernes de TEX, définit les primitives `\numexpr`, `\dimexpr` et `\glueexpr` qui permettent de faire des calculs sur les variables numériques en utilisant les notations usuelles avec les opérateurs `+`, `*` et `/`. Contrairement aux primitives déjà citées, le résultat d'un calcul n'a pas besoin d'être assigné à une variable, il est directement accessible.

Des fonctionnalités similaires sont offertes par le package LATEX `calc`.

```
\the\dimexpr\bigskipamount*2\\
\the\glueexpr\bigskipamount*2\\
\the\glueexpr\bigskipamount*
  \numexpr\year/800\relax-\medskipamount\\
```

24.0pt
24.0pt plus 8.0pt minus 8.0pt
30.0pt plus 10.0pt minus 10.0pt

Les variables intervenant dans les calculs sont sujets aux même restrictions qu'avec les primitives `\advance`, `\multiply` et `\divide`. En outre, dans une multiplication ou une division, il faut toujours commencer par la variable de type correspondant à la primitive utilisée. Les parenthèses peuvent être utilisées pour imposer une précédence aux calculs. Une expression peut contenir elle même des sous-expressions initiées par les autres primitives. Dans une telle situation il est indiqué de terminer chaque sous-expression par une commande `\relax` pour signifier qu'elle est terminée et forcer l'évaluation de son résultat avant de continuer l'analyse de l'expression globale.

> Signalons une différence notable entre la primitive `\divide` et l'opération `/` associée à `\numexpr` quand il s'agit de diviser des entiers. La première calcule la valeur tronquée du quotient, la deuxième sa valeur arrondie. Par exemple, le résultat de la division de 11 par 4 est 2 avec `\divide`, mais 3 avec `\numexpr`

Le package LATEX `calc` Indépendamment du moteur utilisé, le package LATEX offre des fonctionnalités similaires à ce que permettent les primitives eTEX. Il repose uniquement sur les primitives `\advance` et compagnie, ce qui implique que le résultat d'un calcul ne peut qu'être assigné à une variable. En fait, il redéfinit les commandes `\setlength` et

`\addtolength` pour qu'elles supportent des expressions arithmètiques selon la même syntaxe permise par la primitive `\dimexpr`. Toute commande LaTeX qui a recours à `\setlength` pour récupérer ces arguments numériques peut donc utiliser des expressions arithmétiques. Cela inclut par exemple les commandes `\makebox`, `\parbox` ou l'environnement `\minipage`. Les commandes qui accèdent directement à leurs arguments numériques ne profitent en rien du chargement de `calc`. Les commandes `\hspace` et `\vspace` en sont des exemples.

Pour l'exemple, redéfinissons la commande `\hspace` pour qu'elle gére les expressions arithmétiques (lorsque `calc` est chargé).

```
\newlength\tmpdim \let\HSpace\hspace
\renewcommand\hspace[1]{\setlength\tmpdim{#1}\HSpace{\tmpdim}}
```

Toutefois, cette nouvelle version ne gère pas la version étoilée `\hspace*`. La redéfinition suivante le permet.

```
\newlength\tmpdim \let\HSpace\hspace
\renewcommand\hspace[1]{\setlength\tmpdim{#1}%
   \@ifstar{\HSpace{\tmpdim}}{\HSpace*{\tmpdim}}}
```

Par rapport aux primitives standards TEX et les extension ajoutées par eTEX, le package `calc` offre en plus la possibilité de multiplier par un nombre flottant. Deux cas de figures sont envisagés, multiplier directement par un flottant ou bien par le rapport de deux dimensions. Les deux cas correspondent respectivement aux syntaxes

⟨*var*⟩ * \real{⟨*float*⟩}
⟨*var*⟩ * \ratio{⟨*dimone*⟩}{⟨*dimtwo*⟩}

où ⟨*var*⟩ est une valeur ou une variable numérique. Le résultat du calcul est un entier tronqué si ⟨*var*⟩ est un entier et une **dimen** que ⟨*var*⟩ soit une **dimen** ou un **skip**. Il faut absolument respecter l'ordre d'écriture dans cette syntaxe car c'est ⟨*var*⟩ qui annonce le type du résultat à calculer, entier ou **dimen**.

```
\newlength\tmps  \newcounter{tmpc}
\setlength\tmps{\bigskipamount}
 \setcounter{tmpc}{9}
\the\tmps\quad \the\value{tmpc}\\
\setlength\tmps{\tmps*\real{2.5}}
\setcounter{tmpc}{\value{tmpc}*\real{2.5}}
\the\tmps \quad \the\value{tmpc}\\
\setlength\tmps{\tmps*\ratio{10pt}{25pt}}
 \the\tmps
```

```
12.0pt plus 4.0pt minus 4.0pt    9
30.0pt    22
11.99982pt
```

Relaxes mon cher

La façon dont TEX lit les arguments de certaines commandes peut poser problème. Dans une séquence de la forme

⟨*du texte*⟩ `\hskip 12pt plus 4pt` ⟨*du texte*⟩

l'argument `12pt plus 4pt` de `\hskip` n'est pas du tout délimité. Alors comment TEX fait pour savoir que l'instruction est terminée. Il ne le sait tout simplement pas ! Dans cette situation il va lire les caractères (en fait les tokens) les uns après les autres jusqu'à ce qu'il en trouve un qui n'est plus significatif pour l'instruction et arrête alors la lecture de l'unique argument `\hskip`. Par ailleurs, le format plainTEX définit par exemple la commande `\quad` par

```
\def\quad{\hskip1em\relax}
```

L'instruction `\relax` signifie ici à TEX d'arrêter de développer l'argument de la commande `\hskip`, que l'instruction est terminée. La commande `\relax` empêche ainsi qu'un caractère qui ne fait normalement pas partie des arguments d'une commande ne soit absorbé en tant que tel. On peut imaginer ici le cas où la commande `\quad` est suivi d'un mot « plus » qui fait partie du texte, `\relax` empêche alors que ce mot ne soit absorbé parmi les arguments de la commande `\hskip` (et éventuellement provoquer une erreur s'il n'est pas suivi par une dimension valide). Observons maintenant le code suivant

```
\myskip=12pt plus 4pt
\def\aj{plus 4pt}% pour tester le d\'eveloppement
\advance\myskip 12pt \aj \the\myskip\\
\advance\myskip 12pt\relax \the\myskip
```

```
12.0pt plus 4.0pt
36.0pt plus 8.0pt
```

Commentaires : L'instruction `\the\myskip` sur la deuxième ligne survient à un moment où TEX n'as pas fini le processus de développement des arguments de la commande `\advance`, d'où la valeur qui reflète en fait l'état toujours en cours du **skip** `\myskip`. Par contre sur la dernière ligne, `\relax` arrête ce processus, et on se rend compte que même l'assignement précédent a bien été effectif.

LATEX a une approche différente et toutes ses commandes exigent que leurs arguments soient bien encadrés (et non délimités). L'instruction `\hspace2pt` provoque par exemple une erreur car TEX va lire `\vspace{2}pt` ce qui est invalide car `\vspace` n'accepte pas un nombre sans unité.

Inconsistence de la syntaxe des commandes TEX

Le probléme décrit ci-haut n'est qu'un cas particulier d'un constat plus général : l'inconsistance de syntaxe des commandes TEX. En effet, ces

commandes n'ont pas une convention commune de syntaxe. Cet aspect est accentué par la liberté que permettent les primitives de création de macros (`\def`, `\edef`…) en introduisant la notion de délimiteur de paramètres. On peut le constater sur les exemples suivants

```
\hskip 1cm plus .5cm minus .5cm
\advance\mydim by 1cm
\hbox to 3cm {⟨contenu⟩}
\lowercase{⟨texte⟩}
```

LaTeX fait un effort pour homogénéiser la syntaxe de ses commandes quelque soit leur sémantique. À de rares exceptions, elles s'utilisent toutes sous l'une des deux formes

```
\macommande[⟨arg opt⟩]{⟨arg 1⟩}···{⟨arg n⟩}
 ou {\mabascule . . . ⟨contenu⟩. . . }
```

Rappelons toutefois que certaines commandes LaTeX peuvent gérer plus qu'un paramètre optionnel.

Des dimensions dynamiques

Les unités `ex` et `em` sont dymamiquement mise à jour à chaque changement de propriété de la police de caractère. Toutefois le fait d'assigner une valeur à une variable, même avec une unité dynamique, convertit la valeur en l'unité interne `sp` (special point) et la rend donc figée, reflétant l'état lors de l'assignement. En outre lorsque TeX exécute une instruction dans laquelle il doit lire une dimension, ses mécanismes d'évaluation font que peu importe la forme sous laquelle elle se présente du moment que le développement conduise à une dimension. On peut ainsi utiliser une simple macro (définie par `\newcommand`, ou par `\def`) pour enregistrer une valeur de dimension. L'avantage est que l'évaluation ne sera effective qu'au moment de l'exécution, mais l'inconvénient est que la macro ne peut subir aucune modification avec les opérations normalement possibles sur les variables de dimensions, assignement et arithmétique avec les primitives `\advance` et consorts notamment.

```
\newlength\regdim \setlength\regdim{1em}
\def\macdim{1em}
\def\squares{%
  \rule{\regdim}{\regdim}\hskip12pt
   \rule{\macdim}{\macdim}}
\squares\\ \LARGE\squares
```

Avec les mêmes notations, l'instruction `\advance\regdim\macdim` ne pose par exemple pas de problème mais `\advance\macdim\regdim` aboutit à une

erreur puisque \macdim n'est pas une variable numérique et ne peut donc subir une mise à jour de sa valeur avec \advance.

Le même constat est valables pour les valeurs entières sans unité vis à vis des variables de type compteur.

Outre ces aspects, utiliser une macro pour retenir des valeurs numériques présente l'avantage d'économiser un registre. Si des valeurs numériques ne sont utilisées que localement dans une commande ou un environnement préférer toujours les stocker dans des macros. C'est ce que fait par exemple LaTeX avec les macros \width, \height et \depth qui retiennent respectivement la largeur, la hauteur et la profondeur du contenu lors de l'utilisation des commandes \makebox et \parbox.

GROUPER POUR RÉGNER

Un groupe est un bloc de texte encadré entre les caractères { et } (entre autres). Cela n'inclut pas le cas ou { et } servent à encadrer les arguments d'une commande TeX/LaTeX. Tout code qui n'a pas une portée globale voit son effet se limiter au groupe dans lequel il intervient. Cela inclut :

— Les commandes de changement de propriétés de la police de caractère qui agissent comme des bascules telles que \bfseries ou \large ...

— Toute altération des variables numériques dont la portée est locale comme les variables de dimensions et les compteurs TeX (crées avec la commande \newcount). LaTeX préfère gérer ses compteurs de facon globale que ce soit à la création avec \newcounter ou au moment de leurs altération avec les commandes \setcounter, \stepcounter ou \addtocounter. L'altération de la valeur d'une variable numérique peut être préfixée par la commande \global pour forcer un changement global. C'est ce que fait par exemple LaTeX dans les définitions de ces commandes de gestion des compteurs.

  Quelques rares variables numériques intimement liées au moteur sont gérées de facon globale. Elles ne sont toutefois utilisables que pour modifier en profondeur le comportement de TeX dans certaines circonstances.

— Toute définition de nouvelle macro avec \newcommand, \def, \let... Ce qui signifie que toute macro définie avec ces constructeurs à l'intérieur d'un groupe, n'est reconnue qu'au sein du groupe et qu'elle est oubliée une fois celui-ci fermé. Pour forcer une déclaration globale de nouvelles macros à l'interieur d'un groupe, on préfixe la commande de création

par la commande \global. Un raccourci existe pour \def, c'est \gdef qui est l'équivalent de \global\def.

— Comme on va le voir plus tard, les ressorts sont rigidifiés et les pénalités ne sont pas tenues en compte dans un groupe qui intervient dans le mode mathématique.

TEX propose les constructeurs de commandes \def *et sa version globale* \gdef*,* \edef *et sa version globale* \xdef*. Avec* \def *le code de définiton est évalué au moment de l'exécution de la*

```
{\bfseries gras qui} se termine avec le groupe
```

moment de sa définition.

gras qui se termine avec le groupe

```
{\bigskipamount=4pt plus 2pt\relax
 \the\bigskipamount} \\
\the\bigskipamount
```

4.0pt plus 2.0pt
12.0pt plus 4.0pt minus 4.0pt

```
\def\texte{Texte initial}
{\def\texte{Texte chang\‘e}}
\texte
```

Texte initial

Le fait que les caractères { et } soient utilisés comme délimiteurs de groupe n'est pas codé dans le moteur mais défini par le format *plain*TEX et donc LATEX . Sans entrer dans les détails pour l'instant, ces définitions sont :

```
\catcode‘\{=1
\catcode‘\}=2
```

Les deux formats définissent les commandes \bgroup et \egroup en tant que synonymes respectifs de { et } avec :

```
\let\bgroup={
\let\egroup=}
```

Un groupe délimité par \begingroup *et* \endgroup *est dit un semi-groupe.*

En outre, TEX a d'autres délimiteurs de groupes : les commandes primitives \begingroup et \endgroup qui agissent comme les autres délimiteurs à l'exception du fait que

— les commandes {, }, \bgroup et \egroup peuvent être mélangées pour délimiter un groupe, mais un groupe ouvert par \begingroup doit être fermé avec \endgroup ;

— les commandes \begingroup et \endgroup ne peuvent faire office de délimiteurs pour les arguments des commandes TEX/LATEX.

Dans la pratique, préfère toujours délimiter un groupe avec \begingroup et \endgroup, sauf lorsque leurs utilisation n'aura pas de sens (autour d'un argument de commande ou bien du contenu d'une boite \hbox ou \vbox).

Signalons finalement que les environnements LATEX encadrent implicitement leurs contenus dans un groupe. Dans l'exemple

```
\begin{center}
{\Large du texte }
\end{center}
```

les accolades autour de « `\Large du texte` » sont inutiles.

LES ENVIRONNEMENTS LATEX

LaTeX supporte les espaces dans les noms des environnements, mais pas dans les noms des commandes définies par `\newcommand`.

Comment fonctionne un environnement ? Et que fait LaTeX lors de la définition ou de l'utilisation d'un environnement ? La définiton d'un nouvel environnement revêt la forme

```
\newenvironment{foo}[⟨num⟩][⟨defaut⟩]
    {⟨code d'entrée⟩}
    {⟨code de sortie⟩}
```

où ⟨*num*⟩ est le nombre de paramètres sur lesquels agit l'environnement et où ⟨*defaut*⟩ dénote la valeur par défaut d'un éventuel paramètre optionnel (s'il n'y a pas de paramètre optionnel on omet la partie `[`⟨*default*⟩`]` dans la définition). Les paramètres d'un environnement n'ont rien à voir avec le texte qu'il va encadrer lors de son utilisation. LaTeX va en fait créer deux commandes `\foo` et `\endfoo` par ce qui est similaire à :

`\newcommand` refuse de créer toute commande dont le nom commence avec `\end`. *En interne la création d'un environnment* `truc` *implique la définition de deux commandes,* `\truc` *et* `\endtruc`. *D'où la réticence de* `\newcommand`.

```
\newcommand\foo[⟨num⟩][⟨defaut⟩]{⟨code d'entree⟩}
\def\endfoo{⟨code de sortie⟩}
```

D'ailleurs beaucoup d'environnements du noyau LaTeX sont directement définis de cette facon sans passer par `\newenvironment`

Maintenant, lors de l'utilisation d'un environnement quelconque `foo`, les opérations suivantes se succédent :

1. Avec l'instruction `\begin{foo}`, la commande `\begin`
   - initie un groupe avec `\begingroup` ;
   - exécute la commande `\foo` si elle est définie et provoque une erreur « `Environment foo undefined` » sinon ;
2. Le texte entre `\begin{foo}` et `\end{foo}` est traité en tenant compte des initialisations opérées par la commande `\foo` ;
3. Lors du traitement de l'instruction `\end{foo}`, la commande `\end`
   - exécute `\endfoo`. Si cette dernière n'est pas définie un simple `\relax` est exécuté sans provoquer une erreur ;
   - vérifie qu'elle ferme bien `foo` et non un autre environnement, sinon elle provoque une erreur de la forme

     `\begin{bar} on input line 1368 ended by \end{foo}`

     Ce qui interdit l'entrecroisement des environnements ;
   - ferme le groupe avec `\endgroup`.

Notons que

La commande `\document` *commence par fermer le groupe ouvert par* `\begin`.

— Le groupe autour du contenu est crée par les commandes `\begin` et `\end` et non par `\foo` et `\endfoo` elles mêmes. Quelques exceptions à ce constat existent comme pour l'environnement principal `document` : aucun groupe n'est crée.
— Les seuls cas où l'environnement provoque une erreur à l'exécution sont celui où la commande `\foo` n'est pas définie et celui où un croisement avec un autre environnement est survenu.
— Une commande LaTeX ne crée pas un groupe autour de son argument :

```
\newcommand\test[1]{%
 \bfseries #1}
\test{Texte gras} qui continue de l'\^etre
```

Texte gras qui continue de l'être

Une fois l'environnement `foo` défini, rien n'empêche d'utiliser une séquence de la forme

`\foo` ⟨*contenu*⟩ `\endfoo`

sans passer par `\begin` et `\end`. Dans ce cas, aucun groupe n'encapsulera le contenu et tout changement d'état effectué par la commande `\foo` ou survenu dans le contenu survivra après `\endfoo`.

```
\begin{center}\bfseries
Un Texte centr\'e en gras
\end{center}
Du texte normal
```

Un Texte centré en gras

Du texte normal

```
\center \bfseries
Un Texte centr\'e en gras
\endcenter
l'effet cumul\'e dure en dehors
```

Un Texte centré en gras

l'effet cumulé dure en dehors

Dans le sens inverse, le fait que `\end{foo}` ne déclenche pas une erreur si la commande `\endfoo` n'est pas définie rend possible l'utilisation des commandes en tant qu'environnements. Ce n'est utile (et tout à fait indiqué) qu'avec des commandes qui impliquent un changement d'état car cela dispense de recourir aux accolades pour limiter leurs effets et profite du contrôle qu'exerce LaTeX sur les environnements croisés (les messages d'erreur sont plus explicites). Il ne faut pas oublier que le texte encadré par un environnement `foo` ne fait pas office d'argument pour la commande `\foo`. Si `\foo` prend des arguments alors il sont piochés dans ce qui suit `\begin{foo}`.

```
\vbox to 4cm{%
\begin{scshape}\itshape
 Petites capitales italiques \end{scshape}
\vfil
\begin{color}{green!50!black}\bfseries
Graine \‘ecologique \end{color}
\vfil
\begin{fbox} Hello world! \end{fbox}
\vfil
\begin{colorbox}{yellow}{Hello World!}
\end{colorbox}}
```

Petites capitales italiques

Graine écologique

Hello world!

Hello World!

Commande ou environnement, le dilemme

Il est de convention de définir des commandes quand le but est d'agir sur un texte qui ne dépasse pas un paragraphe et des environnements dans le cas de texte de plusieurs paragraphes. La multiplication des accolades pour délimiter les arguments des commandes peut produire des erreurs de non équilibrage très difficiles à identifier. Un environnement a l'avantage d'éviter l'utilisation des accolades et d'intégrer un mécanisme plus explicite de contrôle des erreurs.

Il est par ailleurs conseillé de toujours utiliser `\newcommand*` pour définir de nouvelles commandes. Une commande définie par `\newcommand*` provoque une erreur si l'un de ces arguments contient plusieurs paragraphes.

Introduction aux instructions conditionnelles

Ne me dis pas que c'est pas vrai

TEX a son propre langage de programmation et comme il se doit, il dispose d'un mécanisme pour effectuer un test et exécuter une action selon le résultat de celui-ci. Le test en lui même peut être une vérification de contexte comme savoir quel est le mode actif ou connaître l'état d'une variable booléenne, ou bien une comparaison de deux entités de même nature. La syntaxe complète d'une instruction conditionnelle selon TEX a l'une des deux formes

⟨*code test*⟩ ⟨*action true*⟩ `\fi`
⟨*code test*⟩ ⟨*action true*⟩ `\else` ⟨*action false*⟩ `\fi`

où ⟨*code test*⟩ est un bloc de code qui effectue le test, ⟨*action true*⟩ est le texte qui va être développé si le test réussit et ⟨*action false*⟩ celui qui sera développé si le test est négatif.

Les commandes qui vérifient un contexte donné commencent toujours par `\if` et il est possible à partir de quelques primitives d'en créer de nouvelles. Les commandes de comparaison sont au nombre de six : `\ifdim`, `\ifnum` et `\ifodd` pour les nombres et `\if`, `\ifcat`, `\ifx` pour les tokens. Nous ne verrons pour l'instant que quelques unes de ces structures, sachant qu'un chapitre entier de ce document est consacré au sujet.

Quelques tests génériques

Voici pour l'exemple la liste des commandes primitives de test du mode actif :

`\ifhmode` teste si le mode horizontal est actif ;

`\ifvmode` teste si le mode vertical est actif ;

`\ifmmode` teste si le mode mathématique est actif ;

`\ifinner` teste si le mode actif est horizontal interne, vertical restreint ou **inlinemode**.

Quelques tests de comparaison

Les tests de comparaison numériques nous serons utiles dans l'immediat. La commande `\ifdim`, par exemple, peut être utilisée sous la forme

`\ifdim\dimone>\dimtwo` ⟨*action true*⟩ `\else` ⟨*action false*⟩ `\fi`

instruction qui va tester si la dimension `\dimone` est strictement plus grande que `\dimtwo`. Ces dimensions peuvent êtres explicites ou des variables déclarées. Le test ne compare que les composantes naturelles. Les opérateurs de comparaison utilisables sont =, > et <. La commande `\ifnum` fait de même avec les variables compteurs ou les nombres entiers. `\ifodd` teste si la variable suivante est un nombre entier impaire.

Pour comparer deux commandes, on dispose de la commande `\ifx`. L'instruction `\ifx\`⟨*macone*⟩`\`⟨*mactwo*⟩ teste si les deux macros `\`⟨*macone*⟩ et `\`⟨*mactwo*⟩ ont le même nombre de paramètres et leurs codes de définition, sans aucun développement supplémentaires, sont exactement les mêmes. C'est une simplification exagérée du fonctionnement de la primitive `\ifx` mais qui sera suffisante pour l'instant.

Une utilisation courante de `\ifx` se fait sous la forme :

```
\ifx\myarg\@empty
```

qui teste si le texte de remplacement de la macro \myarg est vide. En effet LaTeX définit la commande \@empty par

```
\def\@empty{}
```

Ce test est très utile si lors de sa définition, on veut prévoir le cas où l'un des arguments d'une macro est vide, on récupère alors le paramètre concerné comme dans \def\myarg{#1} avant d'effectuer le test. Vu la manière avec laquelle la commande \ifx fonctionne, il faut bien comprendre que le test n'est vrai que si #1 est réellement vide et non un code qui ne produit rien. Dans cette dernière situation il est courant de placer le texte en question dans un registre de boite, disons ⟨*box*⟩, et de tester s'il se développe en « rien » avec l'instruction \ifdim\ht⟨*box*⟩=0pt.

En réalité le test peut être faux même si les deux chaînes sont en apparence exactement les mêmes. Oui c'est encore « une autre histoire », mais qui sera clarifiée lorsque on parlera des **catcode**.

Une autre application utile de la commande \ifx concerne la comparaison de deux chaînes de caractères. Il suffit de mettre chacune des chaines dans une macro et de comparer les deux avec \ifx. Le résulat de la comparaison est vrai si les deux chaînes comportent exactement les mêmes caractères, y compris lorsque elle contiennent des noms de commandes (celles-ci ne sont pas développées)

DES TESTS SUR MESURE

Comme il a été dit, il est possible de créer des commandes de test pour effectuer des branchements personnalisés. L'instruction

```
\newif\if⟨cond⟩
```

Il est possible de créer des commandes de test qui vérifient réellement la réalisation d'une certaine condition comme le font les commandes \ifhmode *et similaire.*

crée la nouvelle commande de test \if⟨*cond*⟩ et les commandes \⟨*cond*⟩true et \⟨*cond*⟩false pour respectivement donner à ⟨*cond*⟩ la valeur vrai ou la valeur faux, la valeur initiale par défaut étant \⟨*cond*⟩false.

L'utilité d'une telle création n'est pas très évidente car la commande \if⟨*cond*⟩ n'est associée à rien de concret (elle ne teste rien). Mais imaginons l'application suivante : dans un document qui contient une série d'exercices avec leurs corrigés, on aimerait au moment de la compilation avoir la possibilité d'intégrer ou non les solutions au document produit. Supposons qu'on dispose déjà d'un environnement enonce pour traiter individuellement les énoncés et de même, d'un environnement solution pour les solutions. Il suffit alors de créer la commande de test \ifsol et de modifier solution de la façon suivante

```
\newenvironment{solution}
   { \ifsol ... ⟨code entrée⟩ ... }
   { ... ⟨code sortie⟩ ... \fi}
```

Il suffit ensuit de placer la commande \soltrue au début du document pour intégrer les solutions, sachant que par défaut ce n'est pas le cas. LaTeX utilise extensivement cette technique. Par exemple le test \if@minipage qui est vrai à l'entrée dans un environnement minipage est utilisé ailleurs par plusieurs commandes pour personnaliser leurs comportements dans le cas où elles sont utilisée dans une minipage.

TEX dispose des commandes primitives \iftrue qui effectue un test dont le résultat est toujours vrai et \iffalse qui aboutit toujours à « faux ». Une utilisation intéressante de ces commandes peut être d'encadrer du texte entre les commandes \iffalse et \fi pour qu'il ne soit plus produit dans le document compilé. Il suffit alors et de changer ultérieurement \iffalse par \iftrue si on tient à ce que le texte apparaisse dans le document. Plus fondamentalement, ces deux commandes sont utilisées par \newif pour définir les commandes \⟨*test*⟩true \⟨*test*⟩false quand elle crée la commande de test \if⟨*test*⟩, ces définitions reviennent simplement à

```
\def\⟨test⟩true{\let\if⟨test⟩\iftrue}
\def\⟨test⟩false{\let\if⟨test⟩\iffalse}
```

Une conséquence de ces définitions est que le changement de la valeur du test avec \⟨*test*⟩true ou \⟨*test*⟩false est local au groupe dans lequel il intervient, ce qui est notamment le cas lorsque le changement s'effectue dans un environnement. Si on veut que la modification soit globale il suffit de préfixer ces deux commandes par \global.

LaTeX dans tout cela

Toutes les commandes primitives de test ainsi que les mécanismes à base de \newif sont utilisables avec LaTeX. Le noyau LaTeX ajoute pour ces besoins internes des commandes de test créées avec \newif comme \if@twoside pour savoir si le document est en recto-verso, \if@twocolumn pour savoir si le document est traité en deux colonnes ou encore \if@minipage pour savoir si on se trouve dans une minipage. Un test en particulier peut être très utile pour certaines applications, il s'agit de \ifin@ qui est asscocié à la commande \in@. L'instruction

```
\in@{⟨str⟩}{⟨text⟩}
```

active \in@true si la chaine de caractère ⟨*str*⟩ figure au moins une fois dans le texte {⟨*text*⟩} et \in@false sinon. Les chaines ⟨*str*⟩ et ⟨*text*⟩ ne sont aucunement développées, le test est vrai si ⟨*str*⟩ figure littéralement dans ⟨*text*⟩, mais à n'importe quel position. Dans le cas où, par exemple,

on veut savoir si ⟨*str*⟩ se trouve en fin de ⟨*text*⟩ il suffit d'ajouter une caractère quelconque à la fin des deux chaines après `\in@`.

```
\def\sanspoint#1.{#1}
\makeatletter
\def\otepoint#1{\in@{.+}{#1+}
  \ifin@\sanspoint#1\else#1\fi}
\makeatother
\otepoint{Texte sans fin.}
```

Texte sans fin

Commentaires : `\sanspoint` utilise un point comme délimiteur final de son unique paramètre. Elle retourne toute la chaîne qui précède le point. La commande `\otepoint` utilise le test `\ifin@` pour savoir si son unique argument se termine par un point et recourt à `\sanspoint` pour l'enlever dans ce cas.

LaTeX n'offre pas de mécanisme utilisateur propre pour les instructions conditionnelles. Le package `ifthen` *comble ce vide.*

Par ailleurs, LaTeX ajoute sa propre convention de syntaxe, ainsi que les commandes de test qui lui sont associées comme `\@ifstar`, `\@ifnextchar`, `\@ifdefinable` ou encore `\IfFileExists`. Les noms de ces commandes particulières ne commencent jamais avec `\if` et sont en général créées pour un usage interne ou destinées aux développeurs.

La syntaxe d'une instruction conditionnelle à la manière LaTeX est de la forme

`\`⟨*testcmd*⟩`{`⟨*action true*⟩`}{`⟨*action false*⟩`}`

où `\`⟨*testcmd*⟩ est une commande de test propre à LaTeX.

Comparé au mécaninsme TeX, il n'y a donc pas de commande `\else` et `\fi` et les codes qui seront développés selon le résultat du test doivent souvent être encadrés entre accolades. En fait chaque commande de test LaTeX a deux paramètres, `{`⟨*action true*⟩`}` et `{`⟨*action false*⟩`}` jouant le rôle d'arguments.

En exemple, la commande `\@ifstar` est utilisée pour définir les commandes qui ont une version étoilée. La définition d'une telle commande ressemble à

`\def\macom{\@ifstar{`⟨*avec star*⟩`}{`⟨*sans star*⟩`}}`

Le code `{`⟨*avec star*⟩`}` sera alors développé si la macro est appélée avec `\macom*`, `{`⟨*sans star*⟩`}` si elle est appelée avec `\macom`. Pour que ce mécanisme fonctionne, la macro en question ne peut avoir aucun paramètre. En fait il ne s'agit que d'une unique commande qui adopte une action différente selon qu'elle est suivi ou non d'une `*`. Pour créer une commande qui utilise ce mécanisme mais qui possède des paramètres, il suffit de résumer

son texte de définition à `\@ifstar\comone\comtwo` et de définir ensuite les commandes `\comone` et `\comtwo` avec leurs paramètres éventuels.

```
\makeatletter
\newcommand\TheMouton{%
 \@ifstar{\roman{mouton}}{\arabic{mouton}}}
\makeatother
\TheMouton\quad \TheMouton*
```

8 viii

Commentaires : Une commande qui ne prend aucun argument.

```
\makeatletter
\newcommand\bforit{\@ifstar\textbf\textit}
\makeatother
\bforit{Du texte en italique.}
\bforit*{Un autre en gras.}
```

Du texte en italique. **Un autre en gras.**

Commentaires : Une commande qui prend un argument. La macro `\bforit` n'a aucun paramètre, elle se contente d'aiguiller vers la bonne commande à utiliser selon qu'elle suivie ou non par `*`.

La commande `\@ifnextchar` généralise `\@ifstar`. Elle a trois paramètres dont le premier est supposé être un simple caractère. À l'utilisation si le premier argument correspond au caractères spécifié alors le deuxième argument est développé, sinon c'est le troisième qui est développé. Elle est pas exemple utilisée avec le caractère `[` pour gérer le paramètre optionnel des macros définie par `\newcommand`. Pour imiter la définition d'une macro qui a deux paramètres dont l'un est optionnel en utilisant plutôt `\def` on peut utiliser la charpente suivante.

```
\def\mymac{\@ifnextchar[\macone\mactwo}
\def\macone[#1]#2{⟨code def macone⟩}
\def\mactwo#1{⟨code def mactwo⟩}
```

Dans la défintion de `\macone` les caractères `[` et `]` servent de délimiteurs de paramètres. C'est de cette manière que sont définies toutes les commandes LaTeX qui possèdent plus qu'un paramètre optionnel, comme `\makebox` et `\parbox`.

Des tests façon LaTeX

Sans recourt à des packages spéciaux, pour créer une commande de test façon LaTeX, il faut la définir comme une commande à deux paramètres qui à l'utilisation développe le premier argument ou le deuxième selon la réalisation ou non d'une certaine condition. Pour cela on peut utiliser les commandes internes `\@firstoftwo` et `\@secondoftwo`. Ces deux

commandes possèdent deux paramètres, `\@firstotwo` se développe en son premier argument et `\@secondoftwo` en son second second.

La construction \ifcase

`\ifcase` est une commande de branchement particulière. Elle s'utilise avec une instruction de la forme

```
\ifcase⟨num⟩
   ⟨action 0⟩ \or ⟨action 1⟩ . . .\or ⟨action n⟩
\fi
```

ou ⟨*num*⟩ est un nombre entier explicite ou une variable compteur dont la valeur décide de l'action ⟨*action k*⟩ à éxécuter en ignorant les autres. On peut ainsi personnaliser le comportement d'une macro LaTeX selon la valeur d'un argument numérique optionnel qu'on traitera avec `\ifcase`

```
\newcommand\smbskip[1][1]{\ifcase #1\relax%
   \or \smallskip \or \medskip \or \bigskip \fi}
```

La commande `\relax` n'a aucune utilité dans cette situation mais c'est une bonne habitude à prendre quand on utilise `\ifcase` pour éviter qu'un éventuel chiffre qui fait partie du texte de la première action ne soit absorbé avec le nombre de test. La macro `\smbskip` exécute `\smallskip`, `\medskip` ou `\bigskip` selon que son argument optionnel vaut 1 (valeur par défaut), 2 ou 3. Le test `\ifcase` utilisé dans sa définition ne prévoit pas d'action zéro, donc si l'argument 0 est utilisé la macro ne fait rien.

TP **Définir un style pour compteur LaTeX** • Il est facile de définir un nouveau style pour les compteurs LaTeX en utilisant `\ifcase`.

```
\def\ordinal#1{\ifcase\value{#1}%
 \or premier \or deuxi\‘eme \or troisi\‘eme \or quatri\‘eme
  \or cinqui\‘eme \or sixi\‘eme \or septi\‘eme \or huiti\‘eme
  \or neuvi\‘eme \fi}
```

```
\newcounter{test}
\setcounter{test}{2} \ordinal{test}.\\
\stepcounter{test} \ordinal{test}.
```

deuxième .
troisième .

Ces mystérieuses bestioles qui peuplent les dotsties

`dotsties` = *pluriel de* `.sty` *à prendre au second degré bien sûr.*

L'utilisation de certaines commandes est très récurrente dans les fichiers d'extensions `.sty` ou `.cls`.

`\expandafter` Il est parfois utile de retarder d'un cran le mécanisme de développement du moteur. Dans la séquence

```
\expandafter\cmdone\cmdtwo
```

la commande `\cmdtwo` sera développée avant `\cmdone`, le résultat de ce développement pouvant fournir tout ou une partie des arguments dont a besoin `\cmdone`. Le développement dont il s'agit ici est un développement en un seul pas. Pour une macro cela consiste en son échange contre son texte de remplacement tel quel, les paramètres étant remplacés par autant d'arguments que nécessaire. Il est très fréquent de multiplier l'usage de la commande `\expandafter` sur une même instruction. Par exemple, avec

```
\expandafter\cmdone\expandafter\cmdtwo\cmdtree
```

c'est `\cmdtree` qui est développée et le résultat est placé devant la séquence `\cmdone\cmdtwo` : le premier `\expandafter` exécute le deuxième qui développe à son tour `\cmdtree`. Selon la même logique avec

```
\expandafter\expandafter\expandafter\cmdone
   \expandafter\cmdtwo\cmdtree
```

la séquence devient après une première étape du développement

```
\expandafter\cmdone\cmdtwo⟨developpement de \cmdtree⟩
```

qui finalement aboutit à

```
\cmdone⟨developpement de \cmdtwo⟩⟨developpement de \cmdtree⟩
```

```
\def\firstoftwo#1#2{#1}
\expandafter\fbox\firstoftwo{AAAA}{le reste}
```

A AAA

Commentaires : `\expandafter` développe `\firstoftwo` qui absorbe deux arguments et place la chaîne `AAAA`, résultat de ce développement, devant `\fbox` qui n'utilise que le premier `A` comme argument.

```
\def\gfirstoftwo#1#2{{#1}}
\expandafter\fbox\gfirstoftwo{AAAA}{le reste}
```

AAAA

Commentaires : Ici `\gfirstoftwo` se développe avec ses arguments en la chaîne `{AAAA}` qui est utilisée comme argument par `\fbox`.

`\csname... \endcsname` ces deux commandes vont de paire (en fait `\endcsname` n'est qu'un délimiteur) et servent à construire le nom d'une « control sequence » à partir du texte qu'elles encadrent. Ce texte est developpé avant la construction du nom ce qui permet d'y utiliser des macros et d'avoir ainsi un moyen dynamique de construction de noms. Il est aussi possible d'y utiliser des caractères normalement non autorisés dans les noms de macros comme le caractère espace. Si le nom à former doit être lui même l'argument d'une commande (de définition de macros par exemple) on doit faire précéder la commande en question d'un `\expandafter` pour donner à `\csname` le temps de former le nom d'abord. Par exemple, l'instruction

« control sequence » est le nom qui rassemble toutes les structures dont les noms commencent avec le caractère \, les macros et les variables numériques par exemple.

```
\expandafter\def\csname cmd\romannumeral 1\endcsname{%
                                                Ne fait rien}
```

définit une nouvelle macro dont le nom est `\cmdi` et dont le texte de remplacement est « `Ne fait rien` »

```
\cmdi
```

Ne fait rien

Voici maintenant un exemple qui met en évidence la possibilité d'utiliser des caractères spéciaux dans le nom d'une macro lorsque celui-ci est construit avec `\csname`. Un tel nom n'est alors utilisable qu'à travers `\csname`.

```
\expandafter\def\csname A 1\endcsname{A et un.}
\csname A 1\endcsname
```

A et un.

Commentaires : La commande « `\A 1` » ainsi crée ne peut être appelée directement par son nom, seulement avec `\csname`.

Lorsque on utilise une macro en appelant son nom avec `\csname`, TEX ne lance aucun message erreur si la commande n'est pas définie, un simple `\relax` est utilisé à la place.

> Les noms des deux commandes qui initie et ferme un environnement LATEX sont construits avec `\csname`. C'est ce qui explique que les environnements supportent les caractères spéciaux dans leurs nom et qu'aucun message d'erreur n'est lancé si la commande de fermeture n'est pas définie.

La commande `\expandafter` peut rendre utilisable une instruction conditionnelle là où rien ne semble marcher comme il se doit. Considérons l'exemple suivant

```
\def\isodd#1{\ifodd#1\@firstoftwo\else\@secondoftwo\fi}
```

On essaye avec ce code de définir la commande \isodd qui marche comme un test LaTeX. On voudrait l'utiliser sous la forme

\isodd⟨*num*⟩{⟨*odd action*⟩}{⟨*even action*⟩}

qui va exécuter ⟨*odd action*⟩ ou ⟨*even action*⟩ selon que ⟨*num*⟩ qui est un nombre entier ou une variable compteur est impaire ou paire. Le problème est que sous cette forme elle va provoquer une erreur à chaque coup. La raison en ait que dans le cas d'un test réussi la clause \else est ignorée et \firstoftwo se retrouve avec \fi comme premier argument

```
\makeatother
\isodd 4{impaire}{paire}
```

impairepaire

\noexpand

\aftergroup

\futurelet

\afterassignment

TRAVAUX PRATIQUES

LE PREMIER DES DEUX • On peut utiliser les commandes \@firstoftwo et \@secondoftwo pour créer, de façon très simple, une commande pour exécuter des instructions conditionnelles qui utilisent des tests TeX mais adoptent la convention de syntaxe LaTeX, à la manière de ce que fait le package ifthen.

```
\makeatletter
\def\compare#1{%
 #1\relax%
    \let\next\@firstoftwo
 \else
   \let\next\@secondoftwo
\fi\next}
\makeatother
```

```
\compare{\ifnum2>3}{Vrai.}{Faux.}
```

Faux.

On peut maintenant confectionner à partir de la commande \comparer, des versions spécialisées dans certains tests.

```
\def\numcompare#1{\compare{\ifnum#1}}
\numcompare{15>13}{Vrai.}{Faux.}
```

Vrai.

et même un test qui peut gérer des expressions.

```
\def\enumcompare#1#2#3{%
 \compare{\ifnum\numexpr#1\relax#2\numexpr#3}}
\enumcompare{15+100}>{13*9}{Vrai.}{Faux.}
```

Faux.

On peut ainsi multiplier les mécanismes de test personnalisés tout en honorant l'esprit LaTeX. Pour finir, une commande qui teste si son premier argument est une sous-chaine de son deuxième et développe son troisième ou son quatrième argument selon le résultat du test, amusant sans être forcément utile (du moins sous cette forme).

```
\makeatletter
\def\incompare#1#2{%
  \in@{#1}{#2}\compare{\ifin@}}
\makeatother
\incompare{.+}{Texte au point.+}{Vrai.}{Faux.}
```

Vrai.

Calcul de pgcd • Hé oui ! Bien que ce ne soit pas son terrain de prédilection, TeX est tout à fait capable de faire de tels calculs. Dans la suite on définit la commande \pgcd qui calcule le pgcd de ses deux arguments. Pour cela, on applique un algorithme récursif, la récursivité étant un autre aspect qui entre dans les compétences de TeX.

On fait le choix d'utiliser la primitive \numexpr pour son côté pratique, mais cela oblige à traiter le cas où le résultat d'une division n'est pas la valeur tronqué du quotient. La récursivité à la sauce TeX n'est pas simple à digérer, la faute à la façon très particulière de celui-ci de « développer » le texte qu'il engloutit. Pour se donner une chance de réussir dans une telle tache, il faut que l'appel récursif soit la dernière instruction dans la définition de la commande.

```
\newcommand\pgcd[2]{
  \edef\argone{\the\numexpr#1}%
  \edef\argtwo{\the\numexpr#2}%
  \ifnum\argone>\argtwo\else
     \edef\argone{\the\numexpr#2}%
     \edef\argtwo{\the\numexpr#1}%
  \fi%
  $(\argone, \argtwo)$ % affichage interm\'ediaire
  \ifnum\argtwo=0% \ifx\argtwo0 est aussi valable ici
    \fbox{PGCD: \argone}\par% fin de la r\'ecursivit\'e
  \else
```

```
    \let\tmpone\argone \let\tmptwo\argtwo% clonage des
     variables
    \def\tmpquo{\tmpone/\tmptwo}%
    \ifnum\numexpr\tmpquo*\tmptwo>\tmpone\relax
       \def\tmpquo{\tmpone/\tmptwo-1}%
    \fi%
  \pgcd{\tmptwo}{\tmpone-(\tmpquo)*\tmptwo}%
  \fi}
```

L'utilisation de la primitive `\edef` associée à l'instruction `\the\numexpr` dans la définition de `\argone` et `\argtwo` permet de passer à `\pgcd` des expressions arithmétiques qui seront alors évaluées. Évaluation qui est de toute façon nécessaire vu que le dernier appel de `\pgcd` se fait avec des expressions et non des nombres finaux. La commande finale ainsi créée affiche tous les couples intermédiaires avant d'arriver au résultat final.

```
\begin{flushleft}
\pgcd{372}{165}
\pgcd{13+47}{14*6}
\end{flushleft}
```

(372, 165) (165, 42) (42, 39) (39, 3)
(3, 0) PGCD : 3
(84, 60) (60, 24) (24, 12) (12, 0)
PGCD : 12

Bien que le problème traité ici ne concerne pas la préparation de documents en premier lieu, on ne peut qu'imaginer les facilités que peut apporter l'embarquement du langage de programmation classique `lua` dans le moteur LuaTEX pour de telles taches. On peut même se permettre de rêver d'un moteur de calcul formet ou numérique écrit en lua est accessible depuis un fichier source TEX. Un jour ou l'autre quelqu'un le fera, nous nous contenterons de consommer sans modération.

Sauts et boites

Mise en boite, la méthode TEX

Les commandes de production de boites

Cette section propose une introduction aux boites TEX sachant qu'un chapitre entier de ce document leurs est consacré.

Pour créer des boites TEX on dispose des constructeurs primitifs `\hbox`, `\vbox` et `\vtop` (et `\vcenter` qui n'est utilisable que dans le mode mathématique).

`\hbox` ⟨*spec*⟩ **{**⟨*contenu*⟩**}**

Crée une boite horizontale dont le contenu est ⟨*contenu*⟩. ⟨*spec*⟩ est une spécification qui concerne la largeur de cette boite. Elle est de la forme `to` ⟨*dimen*⟩ ou bien `spread` ⟨*dimen*⟩ où ⟨*dimen*⟩ est une dimension valide TEX et qui peut être négative. Avec la spécification `to` une boite horizontale qui a pour largeur exacte ⟨*dimen*⟩ est crée, indépendamment de la largeur effective de ⟨*contenu*⟩. Avec `spread` la boite occupe la largeur naturelle de son contenu plus la dimension ⟨*dimen*⟩.

`\vbox` ⟨*spec*⟩ **{**⟨*contenu*⟩**}**

Crée une boite verticale dont le contenu est ⟨*contenu*⟩. La boite est alignée de telle façon que sa dernière ligne corresponde à la ligne de base des éléments environnant. ⟨*spec*⟩ a le même sens et la même syntaxe que pour `\hbox` sauf que cette fois elle concerne la hauteur de la boite.

`\vtop` ⟨*spec*⟩ **{**⟨*contenu*⟩**}**

Produit presque la même chose que `\vbox`, mais le contenu est aligné de telle façon que la première ligne corresponde à la ligne de base de l'environnement.

> Les caractères { et } autour de ⟨*contenu*⟩ sont obligatoires et sont en fait des délimiteurs de groupes et non des délimiteurs d'arguments. On peut les remplacer par les commandes \bgroup et \egroup.

Sans aucune spécification de dimension, une boite horizontale a la largeur de son contenu et une boite verticale en a la hauteur. Quelque soit la spécification, si la dimension de la boite (largeur pour une **hbox** ou hauteur pour une **vbox**) est inférieure à la dimension naturelle de son contenu, les ressorts de la liste sont comprimés (jusqu'à une certaine limite) et le contenu peut trés bien déborder de la boite. Par contre si la boite a une dimension plus grande que celle de son contenu alors les ressorts dans la liste de la boite sont étirés pour que celui-ci occupe toute la dimension spécifiée. Dans tous les cas, une boite avec une spécification de dimension occupe exactement l'espace précisé par cette dimension. On appellera hauteur totale d'une boite est la somme de sa hauteur et de sa profondeur.

```
\hbox to 4cm{A B C}
\hbox to 4cm{AB C}
```

A B C
AB C

```
A\hbox spread -2em{B}C
```

CAB

LES REGISTRES DE BOITES

TEX offre la possibilité d'utiliser ce qu'on appelle des registres de boite. Ce sont des structures capables de contenir des boites **hbox** ou **vbox** et de les reproduire ensuite ou encore de les mesurer. Ces registres sont au nombre de 256 et sont numérotés de 0 à 255. On peut affecter une boite à un registre avec la construction

\setbox⟨*num*⟩=⟨*box spec*⟩ {⟨*contenu*⟩}

où ⟨*num*⟩ est le numero de registre à utiliser, ⟨*box spec*⟩ est une spécification de boite **hbox** ou **vbox**.

Une fois cette opération effectuée on peut

— reproduire la boite et libérer le registre avec l'instruction \box⟨*num*⟩ ;
— reproduire la boite mais cette fois conserver son contenu dans le registre avec l'instruction \copy⟨*num*⟩ ;
— récuperer les dimensions de la boite avec \wd⟨*num*⟩, \ht⟨*num*⟩ et \dp⟨*num*⟩ qui designent respectivement sa largeur, sa hauteur et sa profondeur. Ces dimensions sont nulles si le registre n'a reçu aucun contenu.

Les instructions \box⟨*num*⟩ et \copy⟨*num*⟩ ont exactement le même effet que d'insérer directement ⟨*box spec*⟩ {⟨*contenu*⟩}.

La largeur d'une boite verticale est normalement celle décidée par la valeur courante de la variable \hsize, ce qui implique que pour changer cette largeur il suffit de modifier \hsize à l'intérieur de la boite. Si toutefois cette boite est formée elle même d'autres boites, **hbox** ou **vbox**, alors sa largeur est la largeur maximale de ces éléments. Sans spécification de dimension, la profondeur d'une boite \vbox est celle de son dernier élément et sa hauteur est la hauteur de cet élément plus la somme des hauteurs totales des éléments précédents, sauts verticaux inclus. À l'inverse la hauteur d'une boite \vtop et la hauteur de son premier élément et sa profondeur est la profondeur de cet élément plus la somme des hauteurs totales des éléments suivants, sauts verticaux inclus. Il est difficile de cerner la hauteur et la profondeur d'une boite verticale avec une spécification de dimension.

```
\setbox0=\vbox{A} \setbox1=\vbox{\hbox{A}}
hsize: \the\hsize \\
wd0: \the\wd0 \\ wd1: \the\wd1
```

hsize : 129.37119pt
wd0 : 129.37119pt
wd1 : 7.61023pt

```
\def\A{\hbox{A}} \setbox0=\hbox{A}
\setbox1=\vbox{\A\A\A} \setbox2=\vtop{\A\A\A}
H/P de box0\\ ht0:\the\ht0\quad dp0:\the\dp0\\
H/P de box1\\ ht1:\the\ht1\quad dp1:\the\dp1\\
H/P de box2\\ ht2:\the\ht2\quad dp2:\the\dp2\\
\newdimen\thto\newdimen\thtt
HT de box1 et box2:\\
\thto\ht1 \advance\thto\dp1 tht1:\the\thto\quad
\thtt\ht2 \advance\thtt\dp2 tht2:\the\thtt \\
```

H/P de box0
ht0 :7.1832pt dp0 :0.02736pt
H/P de box1
ht1 :34.38321pt dp1 :0.02736pt
H/P de box2
ht2 :7.1832pt dp2 :27.22737pt
HT de box1 et box2 :
tht1 :34.41057pt tht2 :34.41057pt

> Chaque ligne crée par TEX est en fait mise dans une boite \hbox de largeur la dimension \hsize courante. Le contenu d'une page est mis dans une \vbox de hauteur \vsize.

Les dimensions d'une boite peuvent être changées en utilisant les primitives \wd, \ht et \dp. Le contenu ne subit aucune modification y compris pour les vides.

```
\setbox0=\hbox to 3em{A B}
wd0 : \the\wd0\quad Box0: \copy0   \\
\wd0=0pt
wd0 : \the\wd0\quad Box0: \copy0
```

wd0 : 32.84999pt Box0 : A B
wd0 : 0.0pt Box0 : A B

On peut aussi affecter le contenu d'un registre ⟨*num1*⟩ à un registre ⟨*num2*⟩ avec les syntaxes

\setbox⟨*num2*⟩=\box⟨*num1*⟩ ou \setbox⟨*num2*⟩=\copy⟨*num1*⟩

Dans le premier cas le registre ⟨*num1*⟩ est libéré, dans le second il conserve son contenu. Dans tous les cas, ⟨*num2*⟩ contient une copie du contenu de ⟨*num1*⟩ et toute opération ultérieure sur ⟨*num1*⟩ n'a aucun effet sur ⟨*num2*⟩.

```
\setbox0=\hbox{BOX 0} \setbox1=\box0
Registre 0 : \box0 \\ Registre 1 : \box1
```

Registre 0 :
Registre 1 : BOX 0

```
\setbox0=\hbox{BOX 0} \setbox1=\box0
 \setbox0=\hbox{NEW 0}
Registre 0 : \box0 \\ Registre 1 : \box1
```

Registre 0 : NEW 0
Registre 1 : BOX 0

Dans un groupe, toute opération effectuée sur un registre de boite ne laisse aucune trace sur le registre une fois le groupe fermé. Une excéption existe pour le cas où une affectation de contenu à un registre se fait en dehors du groupe et que la production de ce même contenu se fait au sein du groupe : le registre est effectivement libéré.

```
\setbox0=\hbox{AAA}
{\setbox0=\hbox{BBB}} \box0
```

AAA

```
\setbox0=\hbox{AAA}
{\box0 \setbox0=\hbox{BBB}} \box0
```

AAA

Commentaires : Dans le premier code, le registre conserve son état initial malgré le changement effectué dans le groupe. Dans le second, le registre est libéré en reproduisant son contenu initial dans le groupe et une fois le groupe fermé le nouveau contenu est perdu.

Par ailleurs, au lieu de reproduire la boite telle quelle, on peut déballer son contenu dans la liste correspondante avec les commandes \unhbox et \unhcopy lorsque le registre a reçu une boite horizontale, ou bien \unvbox et \unvcopy lorsqu'il a reçu une boite verticale.

```
\setbox0=\hbox to 5cm{Du texte assez long pour
 bien voir l'effet.}
\itshape
Le contenu de la boite tel quel, sur une seule
 ligne : \copy0
\vskip1pc
\unhcopy0 Le contenu de la boite qui est
 mainetnant int\'egr\'e dans la liste en cours.
```

Le contenu de la boite tel quel, sur une seule ligne :
Du texte assez long pour bien voir l'effet.

Du texte assez long pour bien voir l'effet.*Le contenu de la boite qui est mainetnant intégré dans la liste en cours.*

Commentaires : Cet exemple met en évidence le fonctionnement des commande \copy et \unhcopy. Noter que l'italique en cours au moment des reproductions n'est pas appliqué au contenu de la boite. En fait ce contenu hérite des attributs actifs au moment du « remplissage » du registre.

> Le contenu du registre est développé au moment de l'assignement et non au moment de sa reproduction. Tout code contenu dans la boite est ainsi activé lors de l'utilisation de la commande \setbox. Même reproduit dans un contexte différent le contenu conserve les attributs en vigueur lors de l'assignement.

De même, \unvbox et \unvcopy intègrent le contenu d'un registre dans la liste verticale en cours.

Au moment de créer une page, son contenu est mis dans le registre 255. Les registres de 232 à 254 sont réservés par LaTeX pour le traitements des éléments flottants.

Faire référence aux registres de boites par leurs numéros peut être potentiellement dangereux. En effet, aucun mécanisme de protection n'est disponible pour empêcher d'écraser le contenu d'un registre donné. C'est pourquoi les formats mettent en place un système d'allocation dynamique. La commande \newbox définie par plainTEX – et légérement modifiée par LaTeX – permet de donner un nom au premier registre non encore alloué selon la synstaxe

\newbox\⟨*name*⟩

\⟨*name*⟩ s'utilise alors exactement comme un numéro de registre. Les registres 0 à 9, dits registres brouillons, ne sont toutefois jamais alloués par \newbox et à condition de gérer les éventuels conflits on peut les utiliser directement.

Libre n'est pas vide

Il faut distinguer entre la situation ou un registre est libre (void dans le jargon TEX)et celle ou il est vide (empty). Un registre libre est un registre qui n'a reçu aucun contenu et un registre vide est un registre qui a reçu une boite vide comme dans \setbox0=\hbox{}. Tout registre alloué par \newbox reste libre tant qu'il ne reçoit pas de contenu. Signalons que

LaTeX alloue un registre sous le nom `\voidb@x` qui est laissé libre en toute circonstance. Assigner le « contenu » d'un registre libre à un autre libère ce dernier à son tour, comme dans `\setbox0=\box\voidb@x`

On teste si un registre est libre avec la primitive `\ifvoid` suivie du numéro ou du nom du registre. Il n'y a pas de test simple pour savoir si un registre est vide. Le test `\ifdim\wd`⟨*reg*⟩`=0pt` est vrai que le registre soit libre ou ait reçu une boite vide, `\hbox` ou `\vbox`.

```
Void0:\ifvoid0 TRUE\else FALSE\fi,
wd0: \the\wd0
\setbox0=\hbox{}
Void0:\ifvoid0 TRUE\else FALSE\fi,
wd0: \the\wd0
```

Void0 :TRUE, wd0 : 0.0pt
Void0 :FALSE, wd0 : 0.0pt

En outre, il est possible de tester si un registre – qui n'est plus libre – a reçu une boite **hbox** ou **vbox** avec les commandes `\ifhbox` ou `\ifvbox`.

Aplatir, en hauteur et en profondeur • Le fait que l'on puisse modifier les dimensions d'une boite placée dans un registre admet une application qui est mise en œuvre par la commande LaTeX `\smash`. Cette commande permet de faire croire qu'un élément a une hauteur et une profondeur nulles de telle façon qu'il n'influence pas l'interligne avec les lignes adjacentes.

De base, voilà comment on peut définir une telle commande (l'implémentation LaTeX est beaucoup plus robuste)

```
\newbox\smashbox
\newcommand\mysmash[1]{%
  \setbox\smashbox=\hbox{#1}%
  \ht\smashbox=0pt \dp\smashbox=0pt%
  \box\smashbox%
}
```

La boite `\lastbox`

Quand TEX construit une liste partielle, si le dernier élément traité par le moteur est une boite, cette boite est contenue dans le registre natif `\lastbox`. Si le dernier élément traité n'est pas une boite se registre reste libre. Il est aussi libre en permanence dans le mode vertical principal.

`\lastbox` ne peut être utilisé directement. En général on récupère son contenu dans un registre quelconque avant de le réutiliser.

A

```
\hbox{A}\setbox0=\lastbox
\fbox{\box0}
```

Noter que l'instruction `\setbox0=\lastbox` fait que la dernière boite est enlevée de la liste en cours. Dans cet exemple c'est `\fbox{\box0}` qui est responsable du résultat produit et non `\hbox{A}`.

LES BOITES SELON LATEX

MISE EN BOITE AVEC LATEX

Les commandes de créations de boites de LaTeX constituent une interface plus simple et plus riche vers les commandes de bas niveaux `\hbox` et `\vbox`. Ce sont toutes des commandes horizontale, ce qui peut rendre leurs comportement plus prévisible. Par contre, on ne peut les utiliser pour affecter du contenu à un registre de boite.

`\mbox{`⟨*contenu*⟩**`}`**
est équivalente à `\leavevmode\hbox{`⟨*contenu*⟩`}`.

`\makebox[`⟨*width*⟩**`][`**⟨*pos*⟩**`]{`**⟨*contenu*⟩**`}`**
crée une **hbox** contenant ⟨*contenu*⟩. Les paramètres ⟨*width*⟩ et ⟨*pos*⟩ sont optionnels et `[`⟨*pos*⟩`]` ne peut être utilisé qu'après `[`⟨*width*⟩`]`. ⟨*width*⟩ précise la largeur voulue pour la boite. Les macros (c'est bien des macros et non des variables dimensions) `\width`, `\height`, `\depth` et `\totalheight` sont instruites par LaTeX et contiennent respectivement la largeur, la hauteur, la profondeur et la hauteur totale (hauteur+profondeur) naturelles de ⟨*contenu*⟩, mais elle ne peuvent être utilisées que dans l'option `[`⟨*width*⟩`]`. ⟨*pos*⟩ décrit la disposition du texte dans la boite. Ces valeurs possible sont

- **`c`** le texte est centré dans la boite, option par défaut.
- **`r`** le texte est disposé à droite dans la boite.
- **`l`** le texte est disposé à gauche dans la boite.
- **`s`** le texte est étalé sur toute la largeur de la boite (comme pour `\hbox`).

Sans options, `\makebox{`⟨*texte*⟩`}` revient à `\mbox{`⟨*texte*⟩`}`.

`\parbox[`⟨*pos*⟩**`][`**⟨*height*⟩**`][`**⟨*inner-pos*⟩**`]{`**⟨*width*⟩**`}{`**⟨*texte*⟩**`}`**
crée une **vbox** contenant ⟨*texte*⟩. ⟨*width*⟩ est un paramètre obligatoire et doit préciser la largeur de la boite. Les autres paramètres sont optionnels et chacun ne peut être préciseé que si ceux qui l'ont précédé le sont aussi.

⟨*pos*⟩ position verticale de la boite par rapport au texte environnant : `t` pour top (boite créée avec `\vbox`), `c` pour center (valeur par défaut, boite créée avec `\vcenter`) et `b` pour bottom (boite créée avec `\vtop`).

⟨*height*⟩ hauteur voulue pour la boite.

⟨*inner-pos*⟩ positionnement vertical de ⟨*texte*⟩ à l'intérieur de la boite, ces valeurs possibles sont `t,c,b`.

\begin{minipage}[⟨*pos*⟩][⟨*height*⟩][⟨*inner-pos*⟩]{⟨*width*⟩}
\end{minipage}
C'est la version environnement de \parbox. Les paramètres ont exactement les mêmes significations. {minipage} est toutefois plus évoluée : elle construit une sorte de page réduite avec ses propres notes de bas de page (\footnote).

\raisebox{⟨*distance*⟩}[⟨*height*⟩][⟨*depth*⟩]{⟨*box*⟩}
rehausse la boite ⟨*box*⟩ d'une distance (positive ou négative) ⟨*distance*⟩ par rapport à son positionnement naturel. Si les paramètres optionnels ⟨*height*⟩ et ⟨*depth*⟩ sont précisés, alors TEX traitera la boite ⟨*box*⟩ comme ayant la hauteur ⟨*height*⟩ et la profondeur ⟨*depth*⟩.

Toutes ces commandes sont utilisables dans le mode mathématique, leur contenu étant traité en mode texte bien sûr. D'ailleurs \mbox est souvent utilisée pour insérer du texte normal dans une formule. LATEX dispose aussi des commandes \fbox et \framebox qui s'utilisent exactement de la même façon que \mbox et \makebox respectivement, sauf qu'elle ajoutent un cadre autour du contenu. Ce cadre ne change pas la hauteur totale du contenu tel qu'il est vu par TEX. Les variables de dimension suivantes influencent la façon dont ce cadre est tracée.

\fboxrule épaisseur du filet utilisé pour tracer le cadre ;
\fboxsep espace de séparation entre le texte et le cadre.

Il n'y a pas de commandes équivalentes pour les boites verticales mais \fbox, voire \framebox, peut être utilisé pour encadrer le contenu d'une \parbox ou d'une {minipage}. Signalons que le package color (ou encore mieux, xcolor) fournit les commandes \colorbox et \fcolorbox qui sont des variantes de la commande \fbox :

\colorbox{⟨*col*⟩}{⟨*contenu*⟩}
\fcolorbox{⟨*colone*⟩}{⟨*coltwo*⟩}{⟨*contenu*⟩}

ou ⟨*col*⟩, ⟨*colone*⟩ et ⟨*coltwo*⟩ sont des noms de couleurs, ⟨*col*⟩ et ⟨*colone*⟩ servant à colorer le fond de la boite et ⟨*coltwo*⟩ le cadre autour (pour \fcolorbox seulement). \fboxsep et \fboxrule ont les mêmes significations pour ces commandes.

Alors, \vbox *ou* \parbox *?*

Quel est la différence entre les boites verticales crées avec les outils TEX et celles créés avec LATEX. À part le fait que \parbox et une commande

horizontalle et qu'elle fournit une interface unifiée vers les trois types de boites TeX, \vbox, \vtop et \vcenter, elle exécute la commande interne \@parboxrestore dans chaque boite créée. Cette commande est responsable de la restauration du l'état du système par défaut à l'intérieur de la boite, indépendamment de celui déjà en cours à l'extérieur. Des variables telles que \rightskip, \leftskip ou \parindent sont par exemple remises à zéro, ce qui fait que même si un environnement comme center est en cours, le texte sera justifié normalement dans la boite. Avec les mécanismes TeX, le texte dans une boite a les mêmes attributs qu'à l'exterieur.

REGISTRES DE BOITES AVEC LaTeX

LaTeX définit des commandes qui jouent le rôle d'interfaces vers la gestion des registres de boites TeX. Dans la suite, ⟨*nom*⟩ est un nom TeX (avec backslash).

\newsavebox{⟨*nom*⟩**}**
nomme un registre de façon dynamique. Revient à \newbox⟨*nom*⟩.

\sbox{⟨*nom*⟩**}{**⟨*contenu*⟩**}**
revient à utiliser le registre ⟨*nom*⟩ pour stocker le résultat de l'instruction \mbox{⟨*contenu*⟩}.

\savebox[⟨*width*⟩**][**⟨*pos*⟩**]{**⟨*contenu*⟩**}**
revient à utiliser le registre ⟨*nom*⟩ pour stocker le résultat de l'instruction \makebox[⟨*width*⟩][⟨*pos*⟩]{⟨*contenu*⟩}.

\begin{lrbox}{⟨*nom*⟩**}**
\end{lrbox}
environnement qui stocke son contenu dans une \hbox qui est affecté au registre ⟨*nom*⟩.

\usebox{⟨*nom*⟩**}**
Déclenche le mode horizontal avec \leavevmode et reproduit le contenu du registre ⟨*nom*⟩ en utilisant \copy.

Toutes les opérations TeX sur les registres de boites sont simultanénement utilisables avec ces commandes.

> Les primitives TeX ont un problème connu quand le contenu utilise de la couleur. Si c'est le cas, préférer l'utilisation des commandes LaTeX y compris pour l'affectation d'un contenu à un registre (avec \sbox)

MESURER DES BOITES AVEC LaTeX

LaTeX possède les commandes \settowidth, \settoheight et \settodepth qui permettent d'affecter à une variable de dimension la largeur, la hauteur

et la profondeur d'un texte. Contrairement aux primitives TEX, `\wd`, `\ht` et `\dp`, il n'est pas nécessaire de placer le texte en question dans un registre. En fait ces commandes placent elle même ce texte dans un registre `\hbox` avant de le mésurer et ensuite de le vider. Une conséquence (désagréable) de cette implémentation est que toute instruction qui est a une portée globale se trouvant dans le texte est effectivement exécutée. Si ensuite on produit le même texte, de telles instructions se verront donc exécutées plusieurs fois. Cela affecte particulièrement les compteurs LATEX : si par exemple `\stepcounter` est utilisée dans le texte alors le compteur concernée sera incrémenté à chaque utilisation de l'une de ces commandes alors qu'aucun contenu n'est produit.

> Si en écrivant une macro qui a besoin de mesurer l'un de ses arguments, on ne sait pas au préalable si cet argument va contenir des instructions dynamiques, il vaut mieux le placer dans un registre TEX et ne plus faire référence à lui par sa position (`#1` par exemple) mais seulement en tant que contenu du registre en utilisant `\box`, `\copy`, voire `\unhbox` ou `\unvbox`.

Sauter dans une boite

Parles-moi de l'infini

fill est une contraction des mots clés fil et l. D'ailleurs `fil l`*, ou même* `fil L`*, est une syntaxe valide pour désigner* `fill`.

TEX introduit les mots clès **fil**, **fill** et **filll** qui agissent comme des unités de dimension infinies et dont l'ordre va croissant de 1 à 3 (**fill** est infiniment plus grand que **fil** et **filll** l'est par rapport à **fill**). Ils sont utilisables uniquement dans les composantes d'élasticité des **skip**. Les composantes finies sont dites d'ordre nul.

> Dans une boite, **hbox** ou **vbox**, les ressorts qui ont un ordre maximal, éventuellement nul, se partagent les espaces vides dans la liste selon les facteurs utilisés et tous ceux qui sont d'ordres strictement inférieurs sont réduits à leurs composantes naturelles. Les composantes **plus** sont utilisées lorsque TEX doit étirer les ressorts de la liste et les composantes **minus** lorsqu'il doit les compresser, jamais les deux simultanément. Les composantes **minus** ne sont jamais dépassées quitte à sortir de la boite, et les composantes **plus** peuvent l'être autant que nécessaire pour occuper tout l'espace disponible.

```
\hbox to 4cm{A\hskip 1cm  plus10cm  minus1cm
B\hskip 1cm plus 1fil C
\hskip 0pt plus .0001fill D}
```

A B C D

Commentaires : Les deux premiers ressorts sont réduits à leurs composantes naturelles, cédant devant le troisième qui est d'ordre plus grand.

```
\hbox to 4cm{A \hskip 16cm B
\hskip 0pt minus 1fil C \hskip 1cm minus 1cm D}
```

A C D

Commentaires : Le caractère B est projeté très loin à l'extérieur de la boite à cause du saut incompressible qui le précède. Le deuxième ressort qui a une composante **minus** infinie, ramène donc le contenu dans la boite sans que la dernière composante minus, d'ordre inférieur, soit utilisée.

Signalons que pour éviter le recours à la syntaxe TEX, LATEX définit le **skip** `\fill` et la macro `\stretch` par

```
\newskip\fill  \fill = 0pt plus 1fill
\def\stretch#1{0pt plus #1fill\relax}
```

Une utilisation typique de ces dernières ressemble à

```
A\hspace{\fill} B \hspace{\stretch{2}} C
```

A B C

Des exemples • Voici comment sont définies les commandes plainTEX `\hfil`, `\hfill` et `\hss`.

```
\def\hfil{\hskip 0pt plus 1fil}
\def\hfill{\hskip 0pt plus 1fill}
\def\hss{\hskip 0pt minus 1fil plus 1fil}
```

`\hfil` et `\hfill` insèrent un espace horizontal qui peut être nul mais qui est infiniment étirable respectivement au premier et au second ordre. `\hss` produit un espace qui s'étend infiniment dans les deux sens. Elle est surtout utilisée pour placer le contenu dans une **hbox** (à droite, à gauche ou centré). Elle permet aussi d'absorber les espaces excédentaires dans une boite.

```
\hbox to 4cm{\hss ABC}\hrule
\hbox to 4cm{ABC\hss}\hrule
\hbox to 4cm{\hss ABC \hss}
```

ABC
ABC
ABC

La commande LATEX `\makebox` utilise cette technique pour placer le contenue dans une boite quand on spécifie une largeur et une position comme arguments optionnels. Plus précisement elle place le contenu dans un registre `\hbox`, mesure ces dimensions pour les rendre disponibles dans les macros `\width`, `\height` et `\depth`, et déballe en suite le contenu du registre dans une nouvelle `\hbox` de largeur égale à la dimension précisée.

Un exercice [1] comment expliquer que le code :

```
\hfil Un texte centr\'e \par
```

produit effectivement

Un texte centré

La une • pour disposer du texte sur une page avec des éléments espacés verticalement de façon proportionnelle et automatique, on peut faire quelque chose comme

```
\centering%
Collection  \vfill
Titre        \par\smallskip
Auteur        \vskip 0pt plus 2fill
\'edition
}}
```

Collection

Titre

Auteur

édition

Cela manque encore de précision

Comment TEX utilise-t-il exactement les composantes d'élasticité des ressorts ? Rappelons que lorsqu'il traite le contenu d'une boite, il va utiliser ces composantes pour faire en sorte que le texte occupe toute la largeur ou la hauteur disponible selon que la boite est horizontale ou verticale. qu'il ne descend jamais en deçà d'une composante **minus** mais qu'il peut dépasser les composantes **plus** autant que nécessaire.

Quand il traite une boite de dimension donnée, TEX commence par calculer les sommes des pouvoirs d'étirement et de rétrécissement de la liste concernée avec les formules

$$x_{\text{plus}} = x_0 + x_1\text{fil} + x_2\text{fill} + x_3\text{filll}$$
$$y_{\text{minus}} = y_0 + y_1\text{fil} + y_2\text{fill} + y_3\text{filll}$$

où x_i et y_i représentent respectivement la somme des composantes **plus** et celle des composantes **minus** d'ordre i dans la liste. Supposons, pour l'exemple, qu'il s'agit d'une boite horizontale de largeur ℓ et que la largeur naturelle totale des éléments dans la liste, y compris les dimensions naturelles des sauts, est L. TEX va calculer le ratio d'extensibilité r de la boite selon la méthode :

1. **Réponse** : Car \par place ce qui revient à un \hfil en fin de paragraphe.

— si $L < \ell$ (cas d'étirement), et j est l'ordre maximal des composantes **plus** dans la liste alors $r = \frac{\ell - L}{x_j}$;
— si $L > \ell$ (cas de compression), et j est l'ordre maximal des composantes **minus** dans la liste alors $r = \frac{L - \ell}{y_j}$ à l'exception du cas ou $j = 0$ et $y_0 > L - \ell$, cas pour lequel $r = 1$ et qu'on qualifiera pour l'instant de cas d'« overfull box » et qui survient lorsque le contenu déborde de la boite.

Disons qu'un saut donné de la liste a une composante naturelle a, une composante **plus** b et une composante **minus** c. Dans un cas d'étirement, TEX laisse un vide de dimension $a + rb$ si le saut a une composante **plus** d'ordre maximal et seulement de dimension a sinon. Et dans un cas de compression, il va produire un vide de dimension $a - rc$ si le saut a une composante **minus** d'ordre maximal et conserve a sinon. La définition de r dans un cas de compression pour lequel il n'y a pas de composante **minus** infinie ($j = 0$), assure que dans toutes les situations $a - rc \leqslant a - c$. De cette manière, sauf dans un cas d'« overfull box », la largeur totale finale des éléments dans la liste, y compris les vides, vaut ℓ à très peu de chose près. Une conséquence de ce mécanisme est qu'un saut associé à un ressort dont les composantes naturelles sont nulles conservera sa dimension naturelle dans toute circonstance.

Ce qui est dit ici pour une boite horizontale s'applique bien entendu à la `\hbox` d'une ligne et le même procédé est utilisé pour distribuer la différence entre hauteur naturelle et hauteur effective sur les sauts verticaux dans une boite verticale, y compris celle d'une page.

Too bad ! D'accords, mais soit tolérant

Pour chaque boite, horizontale ou verticale, TEX calcule la valeur de la variable `\badness` de la liste avec la formule

$$b = \begin{cases} 0 & \text{si } j > 0 \\ \infty & \text{si cas d'« overfull box »} \\ \min(10000, 100r^3) & \text{si } j = 0 \text{ sans « overfull box »} \end{cases}$$

où r est le ratio d'extensibilité de la liste et j l'ordre maximale des composantes d'élasticité, tous les deux décris dans la section précédente. La variable `\badness` rend compte de la « qualité » des vides dans la liste. Elle est nulle dès qu'il y a un ressort infini dans la liste, ne dépasse jamais la valeur 100 en cas de compression, sauf s'il y a « overfull box » cas pour lequel elle est infinie. Elle peut augmenter très vite en cas d'étirement, la valeur 10 000 tenant alors le rôle de limite infinie. La valeur 100 en cas de

compression signifie que TEX a utilisé les composantes **minus** en leurs intégralités. La valeur 13, très significative pour TEX, correspond à peu près à un ratio de 1/2, situation dans laquelle il utilise la moitié des composantes **plus** en cas d'étirement, ou **minus** en cas de compression. En cas d'étirement, la situation où, par exemple, TEX doit doubler les composantes **plus** correspond à une `\badness` de 800.

Maintenant, pourquoi s'encombrer de tels calculs ? surtout avec la fréquence que cela implique. La raison en est, qu'avec les pénalités, `\badness` est une composante essentielle dans les algorithmes de rupture de listes. Avant de décrire à fond ces processus, disons que `\badness` est par exemple comparée aux variables `\pretolerance` et `\tolerance` comme critère pour retenir ou rejeter une possibilité de former des lignes à partir du matériel formant la liste d'un paragraphe.

Valeurs par défaut dans plain TEX et LATEX
`\pretolerance` = 100
`\tolerance` = 200
`\hbadness` = 1000
`\hfuzz` = .1pt.

D'un autre côté, TEX utilise des variables qui n'influencent pas le résultat de la compilation mais simplement pour lancer des messages d'avertissement. Dans le cas d'une boite horizontale, le premier de ces messages, « `Underfull \hbox` », indique que les vides ont été trop étirés dans la boite. Le second, « `Overfull \hbox` », correspond au cas qu'on a qualifié jusqua'à maintenant de « overfull box ». Le premier message n'est lancé que si la `\badness` de la `\hbox` dépasse la variable `\hbadness` qui est fixée par défaut à 1000. En outre, TEX lance le message « `Overfull \hbox` » dès que le contenu dépasse la largeur de la boite de la dimension `\hfuzz` . La valeur par défaut de `\hfuzz` est très petite, elle est de 0.1pt.

Le même procédé est appliqué aux boites verticales en utilisant cette fois les hauteurs des boites et les variables `\vbadness` et `\vfuzz`. Le terme `\hbox` est remplacé par `\vbox` dans les messages d'avertissement.

Pour indiquer à TEX de ne plus afficher ces messages il suffit de mettre les instructions suivantes quelque part au début du document ou bien dans un groupe TEX pour limiter leur effet localement.

```
\hbadness=10000 \vbadness\hbadness
\hfuzz=\maxdimen \vfuzz\hfuzz
```

On peut le faire aussi au milieu d'un groupe. Pour finir, signalons que pour visualiser les boites `Overfull \hbox` dans le document compilé sans aller fouiner dans le fichier log de la compilation, on peut donner une valeur non nulle à la variable de dimension `\overfullrule`. TEX ajoute alors un trait épais de largeur `\overfullrule` devant chaque boite fautive.

Les pénalités

Penalty ! Allez, cloche-pied sur la liste

Rappelons que les pénalités sont des instructions qui influencent la rupture d'une liste, que ce soit pour former les lignes d'un paragraphe dans le mode horizontal ou bien une page dans le mode vertical. Elles sont souvent insérées implicitement par le moteur ou par les macros à divers endroits dans la liste en cours. TEX utilise pour cela la commande `\penalty` à laquelle il passe des variables numériques (de type compteur) qui fixent chacune la valeur de pénalité qui correspond à une situation particulière et dont les noms indiquent en général l'utilité. Il place par exemple des pénalités dans la liste verticale principale dans la plupart des situations courantes, après chaque ligne d'un paragraphe, avant et après une équation **displaymode**, autour des éléments flottants …. LATEX ajoute d'autres variables et place par exemple des pénalités avant et après un environnement de liste ainsi qu'entre ses items, avant un titre… Certaines variables de pénalité sont interdépendantes et contribuent selon des règles arithmétiques pour former la valeur finale à insérer. Le tableau 4.1 donne les valeurs par défaut instaurées par le format plainTEX – et reprises telles quelles par LATEX – de certaines variables de pénalité utilisées implicitement par le moteur.

Outre les pénalités implicites du système, l'utilisateur peut insérer explicitement une pénalité avec une instruction `\penalty` ⟨*valeur*⟩ ou ⟨*valeur*⟩ est un nombre entier compris entre -10 000 et 10 000. Une telle directive est empilée dans la liste correspondante au mode actif. Ce qui implique que pour placer une pénalité dans la liste verticale, il faut s'assurer que le mode vertical est actif (en placant par exemple `\par` juste avant l'instruction).

Variable de penalité	Valeur	Mode	Point d'insertion
`\binoppenalty`	700	m	après un opérateur binaire dans une formule **inlinemode**. La valeur par défaut décourage de rompre la ligne à son niveau mais la rend quand même possible.
`\relpenalty`	500	m	après le symbole d'une relation dans une formule **inlinemode**. Même remarque.
`\hyphenpenalty`	50	h	après un caractère de césure lorsqu'il est placé implicitement par le moteur.
`\exhyphenpenalty`	50	h	après un caractère de césure explicite (comme dans sous-espace).
`\interlinepenalty`	0	v	après l'insertion d'une ligne dans la liste verticale.
`\clubpenalty`	150	v	Valeur qui s'ajoute à `\interlinepenalty` pour la première ligne d'un paragraphe.
`\widowpenalty`	150	v	Valeur qui s'ajoute à `\interlinepenalty` pour la dernière ligne d'un paragraphe.
`\brokenpenalty`	100	v	Valeur qui s'ajoute à `\interlinepenalty` dans le cas où la ligne se termine par une césure.
`\predisplaypenalty`	10 000	v	avant une ligne qui correspond à une formule **displaymode**. La valeur 10 000 rend impossible la rupture d'une page à ce niveau.
`\postdisplaypenalty`	0	v	après une ligne qui correspond à une formule **displaymode**.

v : mode vertical
h : mode horizontal
m : mode mathématique

Table 4.1 Valeurs par défaut de certaines variables de pénalité du système. Les variables des mode horizontal et mathématique concernent les ruptures de lignes et celles du mode vertical, les ruptures de pages.

L'instruction `\penalty 0` *sert à autoriser une rupture de liste,* `\penalty-10000` *à la forcer et* `\penalty10000` *à la rendre impossible.*

Plus la valeur d'une pénalité est grande plus elle va décourager une rupture de la liste à son niveau. Une valeur positive a tendance à décourager une rupture de la liste et une valeur négative va plutôt l'encourager. Une valeur supérieure ou égale à 10 000 rend impossible la rupture et une valeur inférieure ou égale à −10 000 va la forcer. Une pénalité proche en valeur absolue de 1000 est considérée comme forte. Une valeur absolue de 9999 aura toutes les chances d'aboutir au résultat souhaité mais ne constitue pas toutefois un ordre de rupture ou de non rupture. On peut constater sur cet échelonnement, la similitude entre les ordres de grandeur des valeurs de pénalités et la **badness** d'une boite. Les deux sont en effet intimement liées et constituent les paramètres principaux dans les algorithmes de formation de lignes ou de pages.

Sauf avec une valeur maximale, placer une pénalité en un point donné en fait un candidat à une rupture de liste, y compris en des lieux où la rupture n'est pas normalement permise. Si deux instructions de pénalité se suivent, celle qui utilise la plus petite valeur est retenue car elle indique un meilleur point de rupture.

Pour mieux illustrer les ordres de grandeur des valeurs de pénalité de la configuration, voici les valeurs que reçoivent certaines variables internes propres au format LaTeX :

```
\@lowpenalty   51
\@medpenalty  151
\@highpenalty 301

\@beginparpenalty -\@lowpenalty
\@endparpenalty   -\@lowpenalty
\@itempenalty     -\@lowpenalty
```

Ce code fait partie du fichier `article.cls`. Les variables `\@lowpenalty`, `\@medpenalty` et `\@highpenalty` sont utilisées par les commandes LaTeX `\linebreak` et `\pagebreak` et donnent une idée de l'étalonnage des pénalités dans LaTeX. Les variables `\@beginparpenalty` et `\@endparpenalty` sont utilisées dans les environnements de liste (`enumerate`, `itemize`, …) pour placer des pénalités respectivement juste avant et juste après la liste, `\@itempenalty` étant utilisée entre les items.

Quoiqu'une telle pratique est découragé sans connaissance de cause, les variables de pénalité du système peuvent être modifiées par l'utilisateur n'importe où dans le document en cours de traitement. Elles prennent alors effet à partir de leur occurrence et la portée d'un tel changement se limite au groupe TeX dans lequel il survient. Étant des variables numériques, on

modifie leurs valeurs en utilisant le symbole = ou simplement en faisant suivre leurs noms de la valeur voulue.

Outre la commande primitive \penalty qui est utilisée pour placer directement une pénalité, d'autres commandes TEX ont un lien direct avec la gestion des pénalités.

\allowbreak insère une pénalité nulle : \def\allowbreak{\penalty0}
Elle sert à autoriser une rupture de liste là où elle est normalement impossible.

\break empêche une rupture de liste : \def\break{\penalty10000}

\nobreak force une rupture de liste : \def\nobreak{\penalty-10000}

\eject force une rupture de page : \def\eject{\par\break}

~ insère un espace insécable :

```
\catcode`\~=\active % active le caractere ~
\def~{\penalty10000\ }
```

\unpenalty enlève la dernière pénalité de la liste en cours. Cette commande primitive est ignorée dans le mode vertical principal.

À l'exception de \eject qui est une commande verticale, toutes ces macros peuvent s'exécuter dans le mode vertical ou horizontal pour influer respectivement sur la rupture de la ligne ou de la page. LaTeX définit aussi son lot de commandes en rapport avec la gestion des pénalités. Ces commandes seront exposées dans une prochaine section.

Casser avec douceur

On retiendra que pour influer sur le choix du point de rupture dans une liste, on peut insérer directement une instruction \penalty ⟨*valeur*⟩. De cette façon on dispose de plus de flexibilité qu'avec les commandes de type \break, \nobreak, voire \eject ou les commandes LaTeX \newpage ou \clearpage, qui utilisent des valeurs extrêmes. Une pénalité bien dosée permet dans certains cas d'obtenir le résultat souhaité et ne produira plus aucun effet si son point d'insertion n'est plus proche de la terminaison de la ligne ou de la page. Il faut veiller à ce que l'instruction s'exécute dans le mode voulu.

Les maths ont aussi leurs pénalités

Les seuls points ou une formule **inlinemode** peut être rompue en fin de ligne, c'est après le symbole d'un opérateur binaire (+, ∪ …) ou d'un symbole de relation (=, ≤ …). Les variables de pénalité associées à ces

deux situations sont assez fortes par défaut. Pour interdire toute rupture dans les formules **inlinemode**, il suffit de placer, quelque part au début du document (dans le préambule par exemple), les instructions
`\binoppenalty=10000` `\relpenalty=10000`.

Pour encourager une rupture en un endroit donné dans une formule **inlinemode**, on peut utiliser la commande `\allowbreak` et ce, même si la rupture n'est pas normalement autorisée pour la situation.

```
$a+b+c+d+e+f+g+hhhh+k+l+m+n+o$
\\ \rule{3cm}{.4pt} \\
$a+b+c+d+e+f+g+hh\allowbreak hh+k+l+m+n+o$
```

$$a+b+c+d+e+f+g+hhhh+ k+l+m+n+o$$

$$a+b+c+d+e+f+g+hh \\ hh+k+l+m+n+o$$

Commentaires : On constate sur la première équation que les vides autours des symboles + ne sont pas les mêmes entre la première et la deuxième ligne, TEX ne trouvant pas un point de rupture convenable pour les équilibrer. La rupture induite par la commande `\allowbreak` dans la deuxième équation aurait été impossible sans elle.

On peut ainsi interdire globalement les ruptures de lignes dans le mode **inlinemode** et utiliser `\allowbreak` pour l'autoriser ponctuellement, et même créer une commande qui le fait avec un certain effet (plus ou moins fantaisiste).

```
\newcommand\brisemath{%
 \allowbreak\rightarrow\nobreak\cdots}
```

```
Texte $ a_1+\cdots+a_r+a_{r+1}+\brisemath
+a_n+a_{n+1}$
```

Texte $a_1 + \cdots + a_r + a_{r+1}+ \rightarrow \cdots + a_n + a_{n+1}$

Une sous-formule encadrée dans un groupe TEX ne peut être rompue, même explicitement :

```
$a+b+c+d+e+f+g+{hh\allowbreak hh}+k+l+m+n+o$
```

$$a+b+c+d+e+f+g+hhhh+ k+l+m+n+o$$

Précisons qu'une formule **displaymode** n'est jamais éclaté en plusieurs lignes et que pour les environnements multi-lignes (comme `eqnarray`), chaque ligne est traitée comme s'il s'agit d'une équation **displaymode** autonome. Autant TEX prend sur lui la charge de la mise en forme d'une équation **inlinemode** (au même titre que le texte normal), autant il délègue cette tache à l'utilisateur dans le mode **displaymode**.

Paragraphes au scalpel

Finis moi ce paragraphe

Alors qu'il se trouve dans le mode vertical, TEX commence à construire un nouveau paragraphe dès qu'il rencontre

- Un caractère ordinaire quelconque.
- La commande `\leavevmode`. Elle ordonne explicitement de quitter le mode vertical et donc de commencer un nouveau paragraphe.
- Une commande `\unhbox` pour déballer le contenu d'un registre de boite `\hbox` dans la liste horizontale.
- Une commande `\indent` (pour indenter le texte) ou `\noindent`.
- Une commande quelconque du mode horizontal : une commande qui produit explicitement des caractères ordinaires, `\hskip` ou une commande qui s'y ramène, le caractère `$` de basculement dans le mode mathématique, une commande `\vrule` explicite ou implicite …

Réciproquement lorsqu'il est dans le mode horizontal, c'est-à-dire en train de traiter un paragraphe, il bascule vers le mode vertical dès qu'il rencontre :

`\hbox` n'est pas une commande horizontale, `\vbox` n'est pas non plus une commande verticale. `\vrule` est une commande horizontale et `\hrule` est une commande verticale.

- la commande `\par` qui termine le paragraphe courant ;
- une ligne vide, synonyme de `\par` (une ligne vide ferme donc le paragraphe qui la précéde) ;
- la commande `\unvbox` pour déballer le contenu d'un registre de boite verticale ;
- une commande quelconque du mode vertical : une commande `\vskip` explicite ou implicite, une commande `\hrule` …

Signalons que sous LaTeX, des environnements en apparence très différents tel que `center`, `enumerate` ou `theorem` utilisent la même structure interne : l'environnement `trivlist`. Cet environnement bascule vers le mode vertical avant que son contenu ne force le retour vers le mode horizontal. Le jargon LaTeX qualifie ce comportement de mode **DISPLAY** par référence au mode **displaymode**.

`\leavevmode` *est définie par* `\def\leavevmode{\unhbox\voidb@x}` `\voidb@x` *est un registre de boite toujours vide.*

Notons que `\leavevmode` et `\par` sont opposées. La première permet de quitter le mode vertical pour le mode horizontal et la seconde, l'inverse. Contrairement à l'impression répandue, `\par` ne crée pas un nouveau paragraphe, elle clos celui d'avant. Du point de vue de l'utilisateur cela ne fait pas de différence, mais pour le programmeur c'est un détail crucial. Par exemple `\par` n'insère d'elle même aucun espace dans la liste verticale, mais avec `\leavevmode`, TeX ajoutera par contre un saut de ligne suivi d'un saut de paragraphe avant de basculer vers le mode horizontal.

L'espace d'indentation n'est pas un saut mais une boite `\hbox` *vide de largeur* `\parindent`.

La commande `\indent` ne signifie en aucun cas « commences un nouveau paragraphe ». Elle n'active le mode horizontal que lorsque le mode vertical est en cours. Dans le mode horizontal, elle se contente de produire un espace d'indentation, y compris au milieu d'une phrase. Ceci est valable aussi pour la commande `\noindent`, sauf qu'elle ne produit rien au milieu d'une phrase. Sous LaTeX, la plupart des environnements qui amorcent ou quittent le mode vertical savent gérer l'indentation du paragraphe qui vient après et en général on n'a pas besoin de placer des commandes `\indent` ou `\noindent` manuellement. Par exemple, le texte qui vient immeédiatement après un environnement `enumerate` ou `itemize` n'est jamais indenté, à moins de laisser une ligne vide pour signifier explicitement le début d'un nouveau paragraphe.

La variable de dimension `\parskip` *est utilisée pour les sauts de paragraphe.*

Un détail crucial : TeX insère un saut vertical (`\parskip`) quand il commence un nouveau paragraphe. Même s'il est nul, la présence du saut dans la liste verticale va encourager TeX, plus que d'habitude, à déclencher une rupture de page à son niveau. Ce qui peut avoir pour conséquence de renvoyer le nouveau paragraphe sur la page suivante même s'il reste de l'espace pour contenir encore quelques lignes.

Pas de ligne vide avant une équation centrée

La commande `$`, doublée ou non, est une commande du mode horizontal (qui fait entrer dans le mode mathematique). Malgré les apparences, TeX agit comme s'il ne quite pas le mode horizontal lorsqu'il commence à traiter une formule en **displaymode**. Ce qui implique que le texte suivant l'équation n'est pas indenté (sauf ligne vide). En outre, une formule centrée n'est *jamais* séparée du texte qui la précède immédiatement s'ils

font partie du même paragraphe.

À moins d'avoir l'intention de commencer un nouveaux paragraphe, il ne faut pas laisser une ligne vide avant et/ou après une équation centrée. Dans le cas contraire des sauts verticaux supplémentaires dus à la création de nouveaux paragraphes sont insérés dans la liste verticale, ce qui en fait des points potentiels de rupture de page. On peut ainsi se retrouver, outre la rupture sémantique qui en découle, avec l'équation éjectée vers la page suivante, *alors qu'il y a assez d'espace pour la contenir sur la page courante.*

Au lieu de laisser une ligne totalement vide pour améliorer la lisibilité d'un code source alors que l'intention n'est pas de commencer un nouveau paragraphe, y insérer plutôt un ou plusieurs caractère `%`.

Terminaison de paragraphe

Par défaut, quand TEX forme une paragraphe, chaque ligne occupe toute la largeur de la zone de texte sauf la dernière ligne où le texte occupe sa largeur naturelle. C'est que lorsqu'il rencontre une fin de paragraphe, TEX exécute une séquence de commandes qui revient à : `\unskip\nobreak\hfil`

- `\unskip` va éliminer le dernier saut horizontal dans la liste ;
- `\hfil` va remplir le reste de la ligne par du vide et au même temps écraser tous les sauts élastiques finis qui s'y trouvent ;
- `\nobreak` va empêcher un éventuel retour à la ligne si la ligne est déjà pleine.

En fait au lieu de `\hfil`, TEX utilise l'instruction `\hskip\parfillskip` sachant que la valeur par défaut de `\parfillskip` est `0pt plus 1fil`.

Alors ! Verticale ou horizontale, cette commande ?

Certaines commandes (TEX, en général) ne sont affiliées à aucun mode et peuvent avoir des résultats différents selon le contexte. Les commandes primitives d'insertion de boites `\hbox`, `\vbox` et `\vtop` (ainsi que `\vcenter`) en font partie.

> Les commandes primitives `\hbox`, `\vbox` et `\vtop` insèrent leurs contenus conformément au mode actif. Peut importe la nature des boites, ces contenus sont alignés horizontalement dans le mode horizontal et empilés verticalement dans le mode vertical. Ce mode opératoire est conforme à leurs qualité de commandes primitives utilisées au plus bas niveaux par le moteur.

```
A \hbox{B} \vbox{\hbox{C}} D
```

A B C D

Commentaires : On entre dans le mode horizontal avec la lettre A, les boites suivantes sont donc placées sur la même ligne. La boite `\hbox` utilisée dans la `\vbox` sert à astreindre la largeur de celle-ci à celle du caractère *C*.

```
\hbox{Une ligne.}
\vbox{\hsize=5cm Un paragraphe sans
 interligne pour les s\'eparer.}
\hbox{Une autre ligne}
```

Une ligne.
Un paragraphe sans interligne
pour les séparer.
Une autre ligne

Commentaires : C'est le mode vertical qui est actif donc le contenu des boites est empilé verticalement. Noter que l'interligne avant la `\vbox` n'est pas « naturel ».

`\hbox` *(sans spécification de dimension) est en apparence équivalente à la commande LaTeX* `\mbox`*. Ce n'est pas totalement exact.*

Les commandes de création de boites LaTeX sont par contre toutes des commandes horizontales. Ce qui inclut `\mbox` et sa version généralisée `\makebox`, mais aussi la commande `\parbox` et l'environnement `{minipage}`. La commande `\mbox` est par exemple définie par

```
\def\mbox#1{\leavevmode\hbox{#1}}
```

```
\mbox{A}BC
```

ABC

> Placer une boite verticale directement dans le mode vertical peut avoir des résultats différents selon qu'on utilise `\vtop` ou `\vbox`. Bien que le contenu de la boite soit exactement le même, la hauteur et la profondeur sont différentes selon la primitive utilisée et donc les interlignes avec les éléments adjacents peuvent être différents.

```
\def\A{\hbox{AAA}}\def\B{\hbox{BBB}}\leavevmode
\vbox{\A\vbox{\B\B}\A} \quad
\vbox{\A\vtop{\B\B}\A}
```

AAA AAA
BBB
BBB BBB
AAA AAA

Commentaires : Une boite verticale (ne contenant que des caractères B) est placée dans une liste verticale en utilisant `\vbox` et ensuite `\vtop`. Noter les interlignes différents selon la primitive utilisée.

S'il est une commande dont l'utilité n'a aucune commune mesure avec la fréquence de son utilisation dans les documents finaux, c'est `\vadjust`.

> La commande `\vadjust`, extrêmement utile, permet d'insérer du matériel dans la liste verticale alors que le mode actif est horizontal tout en restant dans ce mode. Le matériel en question est placé au début de la ligne qui suit le point d'insertion de l'instruction.

`\vadjust` permet par exemple d'ajouter un saut à l'interligne naturel avec une instruction de la forme `\vadjust{\kern 6pt}`, ou encore d'ajouter

une pénalité à la liste verticale sans quitter le mode horizontale comme dans l'instruction `\vadjust{\nobreak}` qui empêche qu'une rupture de page ne survienne après la ligne courante.

La commande LaTeX `\\` est souvent redéfinie dans certains environnement comme `center` ou `tabular`.

La commande LaTeX `\\`, dans sa version normale, est un bon exemple d'utilisation de `\vadjust`. Elle provoque un retour à la ligne sans quitter le mode horizontal et sa version étoilée `\\*` empêche en plus qu'un saut de page ne survienne après. Elle ne produit pas du tout le même résultat que la commande `\par` car, outre le fait qu'avec `\par` un saut de paragraphe est ajouté à la liste verticale, le texte qui suit `\\` fait toujours partie du même paragraphe et donc les résultats des calculs fait par le moteur (démérite du paragraphe) pour choisir les points de ruptures de toutes les lignes ne sont pas du tout les mêmes.

Un titre • À supposer qu'on veuille écrire une commande LaTeX qui prend deux paramètres et qui va écrire chaque argument avec son propre style sur une ligne, le tout étant centré horizontalement,

```
\newcommand\partie[2]{%
 \begin{center}\textsf{#1} \\ \textbf{#2}\end{center}}
```

Seulement, si on utilise l'instruction `\partie{}{Texte}`, on est gratifié d'un message d'erreur qui indique qu'il n'y a aucune ligne à terminer avec la commande `\\`. La raison est que la commande `\\` s'exécute uniquement dans le mode horizontal et que dans cette situation le mode actif est vertical. Pour corriger ce problème on peut ajouter une instruction `\mbox{}`, voire `\leavevmode`.

```
\newcommand\partie[2]{%
 \begin{center}\mbox{}\textsf{#1} \\ \textbf{#2}\end{center}}
```

La commande `\null` définie comme `\hbox{}` a un effet similaire à `\mbox{}`, mais n'est pas une instruction horizontale.

L'instruction `\mbox{}` déclenche maintenant le mode horizontal (et ne produit rien), ce qui ne résoud que partiellement le problème, puisqu'une ligne vide inutile va être produite dans le document final. En fait, il serait plus adéquat ici de vérifier si le premier argument est vide et d'exécuter seulement l'instruction `\textbf{#2}` dans ce cas, mais c'est une autre histoire. Par ailleurs, rien n'empêche une rupture de page entre les deux lignes. Pour éviter que cela ne se produise, il suffit de remplacer `\\` par `\\*` dans la définition.

Retenir le contenu d'un environnement pour lui appliquer un effet • Parfois on a besoin d'appliquer un effet à tout le contenu d'un environnement. Par exemple créer un cadre autour de ce contenu avec `\fbox` ne marchera pas car cette commande traite son argument en **LR-mode**. L'idée est de stocker le contenu de l'environnement dans un registre de boite `\vbox` et d'appliquer ensuite l'effet à la boite.

```
\newbox\envbox
\newenvironment{fparbox}
  {\setbox\envbox=\vbox\bgroup%
     \advance\hsize-2\fboxsep \advance\hsize-2\fboxrule%
 }{\egroup\par\medskip\fbox{\box\envbox}\par\medskip}
```

Les commandes \bgroup et \egroup sont précieuses ici puisque l'usage direct des caractères { et } provoque une erreur de compilation à cause des accolades non équilibrées. Dans le code de sortie de l'environnement le premier \par fait entrer dans le mode vertical, la commande \fbox, qui est horizontale, crée un nouveau paragraphe dont le seul contenu sera la boite avec son cadre.

```
Du texte en dehors de l'environnement.
\begin{fparbox}
Texte encadr\'e convenablement.\par Il peut
 m\^eme contenir plusieurs paragraphes.
\end{fparbox}
Texte apr\`es l'environnement.
```

Du texte en dehors de l'environnement.

Texte encadré convenablement.
Il peut même contenir plusieurs paragraphes.

Texte après l'environnement.

LES PARAMÈTRES QUI INFLUENCENT LA FORMATION DE PARAGRAPHES

> La plupart des variables mentionnées dans cette section (largeur de la ligne, interligne, retraits, espace inter-mots…) ne sont lues par TEX qu'au moment ou le paragraphe est fermé. Un changement de ces variables n'est donc pris en compte au sein d'un groupe que si le moteur rencontre une commande \par à l'interieur du dit groupe, car une fois celui-ci fermé le changement n'est plus visible.

DIMENSIONS DE LA LIGNE ET ENTRE DEUX LIGNES

La gestion de la largeur des lignes ainsi que l'espace de séparation entre deux lignes successives est assez complexe. Il y a la largeur naturelle de toute la zone de texte, la largeur de la ligne dans un environnement de liste, la largeur d'une colonne dans un environnement multi-colonne ou encore la largeur d'une boite verticale ou d'une cellule dans un tableau. Pour l'espace interligne, la hauteur et la profondeur des contenus ont

une influence sur la distance entre les lignes de bases de deux lignes adjacentes.

Pour les largeurs de lignes, on dispose de

\textwidth qui est une variable de dimension LaTeX. C'est la largeur de toute la zone de texte et elle est toujours la même quelque soit la situation.

\columnwidth est une dimension LaTeX qui est calculée automatiquement en fonction du nombre de colonne et de l'espace qui les sépare dans un environnement multi-colonnes. L'espace de séparation des colonnes est déterminé par la variable `\columnsep`.

\linewidth c'est une dimension LaTeX qui est continuellement mise à jours pour refléter la largeur de ligne effective, que ce soit dans le mode horizontal externe, dans un environnement multi-colonne, dans un environnement de liste, ou dans une boite verticale LaTeX, `\parbox` ou `minipage`.

\hsize c'est une dimension native du moteur qui reflète la largeur en cours de la ligne. Elle est notamment utilisée pour fixer la largeur d'une boite verticale TeX, `\vbox` ou `\vtop`. Dans la plupart des situations où `\linewidth` change, LaTeX s'assure que `\hsize` le soit aussi.

L'altération de la dimension `\textwidth` n'a un effet que dans le préambule. C'est l'instruction `\begin{document}` qui l'utilise pour renseigner les variables internes (dont `\hsize`) qui sont effectivement utilisées dans le document. Les dimensions `\linewidth` et `\hsize` peuvent être altérée au milieu du document, mais vu qu'elles sont constamment mises à jour, les changer en dehors de tout groupe peut aboutir à des effets très aléatoires. En général on les utilise comme références pour effectuer d'autres mesures (tracer des filets horizontaux, fixer la largeur d'une boite verticale…). On préviligiera en général la dimension LaTeX `\linewidth` sauf lorsqu'on veut fixer la largeur d'une boite verticale créee avec les commandes primitives car ses dernières ne reconnaissent pas les variables LaTeX comme on le constate sur l'exemple suivant.

```
\vbox{\linewidth=2cm du texte dans une boite
 verticale}\medskip
\vbox{\hsize=2cm du texte dans une boite
 verticale}
```

du texte dans une boite verticale

du texte dans une boite verticale

Commentaires : La primitive `\vbox` ne reconnaît pas la variable `\linewidth`.

En ce qui concerne l'espace interligne on dispose de

\baselineskip c'est une variable primitive du moteur qui fixe la distance idéale entre deux lignes successives. LaTeX fait en sorte qu'elle soit mise à jour à

Le package setspace *fournit la commande* \setstretch *qui permet d'altérer l'espace interligne sans risques.*

chaque changement de taille de la police de caractère et lui donne en général une valeur de 20% supérieure à la hauteur totale maximale de celle-ci. Normalement, elle ne doit pas être alterée manuellement, sauf pour obtenir des effets non standards.

\lineskiplimit `\lineskiplimit` est la distance minimale à maintenir entre le plus bas niveau d'une ligne et le plus haut de celle qui la suit.

\lineskip `\lineskip` est l'écart à maintenir entre deux lignes successives dans la situation où, en appliquant le principe de `\baselineskip`, la distance entre elles est plus petite que `\lineskiplimit`. Cette situation se présente si la ligne courante contient un élément de profondeur plus grande que la normale et/ou la ligne suivante un élément plus haut que la normale.

\parskip `\parskip` est la dimension du saut inséré entre deux paragraphes ou plus exactement avant chaque paragraphe.

Comme leurs noms l'indiquent, ces dimensions sont des ressorts. LaTeX préfère toutefois leurs affecter des valeurs rigides. C'est ce qui explique que lorsqu'une page est rompue sans qu'il y ait encore assez de matériel pour la remplir, les espaces étirés dans la liste sont ceux entre les paragraphes, ceux entre les lignes conservant leurs dimensions naturelles. Dans ce sens, donner à `\parskip` une valeur rigide comme dans `\parskip=0pt` est apocalyptique dans certains cas puisqu'il risque de donner une **badness** infinie à toutes les pages.

Faisons une synthèse de la gestion des espaces interlignes du moteur. Disons que b est la valeur de `\baselineskip`, l celle de `\lineskiplimit`, p est la profondeur de la ligne courante et h est la hauteur de la ligne qui va suivre. Si $b - p - h \leqslant l$ alors l'espace inséré entre les deux lignes est de dimension $b - p - h$, dans le cas contraire il est de dimension `\lineskip`.

`\hrule` *est une commande verticale qui trace un filet horizontal qui parcoure toute la largeur disponible.*

Signalons qu'il y a des exceptions à cette règle. L'une d'elle concerne la commande primitive `\hrule` qui élimine tout espace interligne entre le filet qu'elle trace est les lignes adjacentes.

Les commandes plainTeX suivantes peuvent altérer l'espace interligne.

\nointerlineskip Annule l'espace interligne pour la ligne suivante. Peut être utile pour disposer verticalement du matériel graphique de façon rigoureuse.

\offinterlineskip Désactive l'espace interligne dans tout le groupe TeX qui la contient.

La commande `\strut` est utilisée pour obliger une ligne d'avoir une hauteur et une profondeur maximale. Elle ne fait qu'insérer une boite `\hbox` de largeur nulle, de hauteur `0.7\baselineslkip` et de profondeur `0.3\baselineslkip` dans la liste horizontale et veille à ce que les espaces

horizontaux autour d'elle ne subissent aucun changement. Cette boite, associée au registre \strutbox, est reconstruite à chaque changement de taille de la police de caractère. On peut arriver au même résultat en traçant directement un filet de même dimension que \strutbox avec une instruction

La hauteur totale de la boite \strutbox *est* \baselineskip.

```
\vrule width0pt height .7\baselineskip depth .3\baselineskip
```

mais cela va ajouter un espace inter-mot supplémentaire. Une forme raccourcie de cette instruction peut se faire grâce à la commande LaTeX de tracé de filet \rule

```
\rule[-.3\baselineskip]{0pt}{\baselineskip}
```

On peut généraliser cette technique en choisissant une hauteur et une profondeur plus grande si nécessaire. Cela peut s'avérer très utile pour mieux aérer sélectivement certaines lignes d'un tableau par exemple.

```
\let\h\hbox \hfil
\vtop{\h{Texte de}\h{profondeur}\h{nulle}}\hfil
\vtop{\h{Texte
 de\strut}\h{profondeur}\h{nulle}}
```

Texte de profondeur nulle Texte de profondeur nulle

Interligne fixe • Il est intéressant de voir comment la commande TEX \offinterlineskip est définie :

```
\def\offinterlineskip{\baselineskip-1000pt
  \lineskip0pt \lineskiplimit\maxdimen}
```

\baselineskip recoit une valeur négative et \lineskiplimit la dimension maximale tolérée par TEX de telle sorte que deux lignes sont toujours considérées comme trop proches, ce qui fait basculer vers l'application de \lineskip qui est nulle pour l'occasion. À l'opposé donner à \lineskiplimit une valeur négative assez grande en valeur absolue implique de toujours utiliser \baselineskip. Ceci peut inspirer des applications comme

```
{\lineskiplimit=-100pt \baselineskip=18pt
Un paragraphe avec un interligne fixe, co\^ute
 que co\^ute : $\displaystyle\int_0^1
 \frac{\mathrm dx}{\sqrt x}$. Une \'ecriture
 sur  grille, ou comment bousiller tout ce que
 \TeX{} essaye de faire.\par}
```

Un paragraphe avec un interligne fixe, coûte que coûte : $\displaystyle\int_0^1 \frac{\mathrm dx}{\sqrt x}$. Une écriture sur grille, ou comment bousiller tout ce que TEX essaye de faire.

FICHU baselineskip • Si vous n'êtes pas d'accords, observez le résultat du code suivant

```
\LARGE Ce titre  d\'erange\\
\huge par ses interlignes \\
\normalsize tout semble normal pourtant
```

Ce titre dérange
par ses interlignes
tout semble normal pourtant

Toujours pas convaincus, en voilà encore

```
\centering
\LARGE Ce titre ne d\'erange plus\\
\huge par ses interlignes \\
\normalsize presque rien n'a chang\'e pourtant
```

Ce titre ne dérange plus
par ses interlignes
presque rien n'a changé pourtant

Bizarrement, l'interligne semble plus correct dans ce deuxième exemple. Qu'est ce qui a changé ?

Dans le premier exemple, il faut se rappeler que les paramètres du paragraphes ne prennent effet qu'une fois celui-ci fermé et que la commande `\\` ne crée justement pas de nouveau paragraphe. L'interligne est donc celui imposé par la commande `\normalsize` (qui change la valeur de `\baselineskip`). Dans le deuxième exemple, la commande `\centering` change la définition de `\\`. La nouvelle définition utilise `\par` et compense le saut `\parskip` pour donner l'impression d'un saut de ligne normal. Moralité,

La commande `\centering` *est à la base de l'environnement* `center`.

> Ne pas utiliser la commande `\\` quand on change de taille de police dans un paragraphe. Traiter, autant que possible, chaque paragraphe avec un même corps de police. Si `\parskip` n'est pas nul et qu'on ne tienne pas à avoir des sauts de paragraphe, on peut annuler son effet avec une instruction
> `\par\addvspace{-\parskip}`.

LE PROBLÉME DES VBOX DANS LE MODE VERTICAL

Selon les principes de calcul de l'espace interligne, on comprend mieux la différence dans le placement d'une **vbox** dans le mode vertical selon que la primitive utilisée est `\vbox` ou `\vtop`.

```
\def\A{\hbox{A}}
\setbox0=\vbox{\A\A}
\setbox1=\vtop{\A\A}
lineskip:\the\lineskip\\
lineskiplimit:\the\lineskiplimit\\
baselinskip:\the\baselineskip\\
\footnotesize
ht0:\the\ht0\quad dp0:\the\dp0\\
ht1:\the\ht1\quad dp1:\the\dp1
```

lineskip :1.0pt
lineskiplimit :0.0pt
baselinskip :13.6pt
ht0 :20.7832pt dp0 :0.02736pt
ht1 :7.1832pt dp1 :13.62737pt

À voir ces résultats, on peut prévoir que si on insère \box0 dans une liste verticale alors \lineskip sera forcément l'interligne avec la ligne précédente et probablement \baselineskip sera utilisée pour le calcul de celui avec la ligne suivante. Avec \box1, c'est le contraire qui se produira. Une façon de corriger les espaces interlignes est d'utiliser les commandes \strut et \nointerlineskip comme dans l'exemple suivant.

```
\leavevmode
\vtop{\hsize4cm Texte avant.\strut\par\nointerlineskip
\vbox{\strut Texte dans la boite,\\ avec plusieurs lignes.}
Texte apr\'es.}
\vtop{\hsize4cm Texte avant.\par
\vtop{Texte dans la boite,\\ avec plusieurs lignes.\strut}
\nointerlineskip\strut Texte apr\'es.}
```

Texte avant.
Texte dans la boite,
avec plusieurs lignes.
Texte après.

Texte avant.
Texte dans la boite,
avec plusieurs lignes.
Texte après.

Avec les commandes \unvbox et \unvcopy, c'est une autre histoire.

À TITRE D'EXERCICE [1], expliquer pourquoi l'interligne du texte produit par le premier code de cet encadré n'est pas correct pour les trois premières lignes.

LES VARIABLES \prevdepth, \spacefactor ET COMPAGNIE

\prevdepth est une variable interne du moteur qui n'est accessible que dans le mode vertical. Elle est utilisée pour retenir la profondeur du dernier élément ajouté à la liste verticale courante. TEX lui donne la valeur -1000pt dans certaines situations pour indiquer qu'aucun espace interligne ne doit être ajouté pour l'élément suivant. C'est cette valeur qui est utilisée par exemple au tout début d'une liste verticale ou bien avant et après un filet \hrule. La commande \nointerlineskip est aussi définie par

1. **Réponse :** \\ ne ferme par le paragraphe et \footnotesize modifie la valeur de \baselineskip. C'est cette nouvelle valeur qui est appliquée à tout le paragraphe.

```
\def\nointerlineskip{\prevdepth-1000pt}
```

Maintenant, quand on utilise les commandes \unvbox ou \unvcopy pour déballer le contenu d'une boite \vbox dans la liste verticale, \prevdepth reçoit la valeur -1000pt avant la boite, ce qui signifie qu'aucun espace interligne ne sera ajouté. La valeur réelle de \prevdepth précédant l'insertion de la boite est retenue et elle est rétablie après cette insertion. Aucun espace interligne ne sera donc ajouté après le contenu de la boite si la liste verticale commence avec l'une de ces commandes.

```
\let\h\hbox
\setbox0=\vbox{\h{Dans la boite}\h{avec plusieurs lignes.}}
\leavevmode\hfil
\vtop{\h{Avant la boite} \unvcopy0 \h{Apr\'es la boite.}}
\hfil\vtop{\unvcopy0 \h{Apr\'es la boite.}}
```

Avant la boite
Dans la boite
avec plusieurs lignes.
Après la boite.

Dans la boite
avec plusieurs lignes.
Après la boite.

Il est possible de maitriser l'interligne en jonglant astucieusement avec \prevdepth lorsque on utilise \unvbox ou \unvcopy.

```
\newdimen\dimdepth \let\h\hbox
\h{juste avant la boite} \dimdepth=\prevdepth
\setbox0=\vbox{\prevdepth=\dimdepth
\h{dans la boite,}\h{avec plusieurs lignes.}}
\unvbox0\h{juste apr\'es la boite.}
```

juste avant la boite
dans la boite,
avec plusieurs lignes.
juste après la boite.

TP **INTERLIGNES SANS DOULEUR** • Formalisons la technique précédente dans un environnement LaTeX dont le but est de simplement collecter du matériel dans un registre \vbox pour ensuite déballer son contenu dans la liste verticale en cours en produisant des interlignes tout à fait normaux.

```
\newdimen\dimdepth \newbox\envbox
\newenvironment{xunvbox}
  {\par\dimdepth\prevdepth
  \setbox\envbox=\vbox\bgroup\prevdepth\dimdepth}
  {\par\global\dimdepth\prevdepth\egroup%
  \unvbox\envbox\prevdepth\dimdepth}
```

On enregistre la valeur de \prevdepth juste avant d'entrer dans la boite et juste avant d'en sortir et on rétablit opportunément ces valeurs pour corriger les interlignes. Noter l'utilisation de \global au milieu de la boite

pour que \dimdepth conserve sa valeur en dehors du groupe délimitant le contenu de la boite, mais aussi celui crée par l'environnement.

```
Texte avant l'environnement.
\begin{xunvbox}
La magie de \verb+\prevdepth+ qui op\'ere dans
 l'environnement pour corriger les interlignes
 dans toutes les situations.
\end{xunvbox}
Texte apr\'es l'environnement.
```

Texte avant l'environnement.
La magie de \prevdepth qui opère dans l'environnement pour corriger les interlignes dans toutes les situations.
Texte après l'environnement.

\spacefactor est une variable interne de type compteur du moteur qui n'est accessible que dans le mode horizontal. Elle joue le rôle tenu par \prevdepth, mais cette fois pour le mode horizontal : elle détermine la dimension du vide qui remplace un caractère espace dans le fichier compilé. \spacefactor est alimentée après chaque elément inséré dans la liste horizontale, mais elle n'est effectivement utilisée que lorsque le moteur rencontre un caractère espace. La plupart du temps sa valeur est 1000, cas dans lequel le ressort inter-mot est utilisé tel quel. La régle précise étant de multiplier la composante **plus** du saut par le ratio $\frac{\texttt{\spacefactor}}{1000}$ et de diviser sa composante **minus** par ce même ratio. Ainsi, lorsque \spacefactor est plus grand que 1000 la composante **plus** du saut est allongée et sa composante **minus** et raccourcie. En outre lorsque \spacefactor est plus grande que 2000, la composante naturelle du saut est elle aussi augmentée (de la valeur restituée par \fontdimen7\font).

Plusieurs caractères espace successifs ne comptent que pour un seul. Un caractère de fin de ligne est remplacé par un caractère espace sauf si le dernier caractère de la ligne est un caractère de commentaire %.

Pour plus de précision :

→ TEX récupère les composantes du saut inter-mot appliqué dans des conditions normales (quand \spacefactor vaut 1000) à partir des fichiers de la police de caractère utilisée.

→ Indépendamment de la police utilisée, TEX associe un code individuel, dit « space factor code » ou **sfcode**, à chaque caractère.

x	1000	Y	999	1	1000
,	1001	;	1002	:	1003
!	1004	?	1005	.	1006
)	0	(	1000		

Quelques caractères avec les **sfcode** utilisés dans ce document (et imposés par amsmath! pour les ponctuations). Le **sfcode** d'une lettre minuscule est 1000, celui d'une majuscule est 999.

Le **sfcode** d'un caractère ⟨*char*⟩ est une variable numérique de type compteur accessible en lecture et en écriture à travers la construction `\sfcode`\⟨char⟩`.

→ **Après un caractère**, `\spacefactor` reçoit la valeur du **sfcode** de ce dernier sauf dans deux situations :

- Si ce **sfcode** est nul, `\spacefactor` conserve sa valeur. C'est ce qui arrive après une parenthèse fermante ou un accent par exemple.
- Si ce **sfcode** est > 1000 mais l'ancienne valeur de `\spacefactor` est < 1000 alors ce dernier reçoit la valeur 1000. Une telle situation survient lorsque une majuscule est suivie d'une ponctuation.

→ **Aprés une boite `\hbox` ou un filet `\vrule`**, `\spacefactor` vaut 1000.

→ **Après le contenu d'un registre déballé par la primitive `\unhbox` ou `\unhcopy`**, `\spacefactor` conserve la valeur d'avant l'insertion du contenu du registre.

Après le contenu d'une boite insérée directement par `\hbox` ou bien avec `\box` ou `\copy`, un espace est transformé en un saut inter-mot normal. Après le contenu d'une boite déballé par `\unhbox` ou `\unhcopy`, un espace est transformé en un saut qui a la même amplitude que le saut qui a précédé le contenu de la boite, donc un saut nul si ce contenu commence une nouvelle liste.

TEX permet aussi de fixer le saut inter-mot à travers la variable `\spaceskip`. Dès que cette variable est non nulle, TEX l'utilise en lieu et place des informations de la police, les règles impliquant `\spacefactor` étant les mêmes. Dans le cas où `\spacefactor` ⩾ 2000, TEX utilise de la même façon la variable `\xspaceskip` dès qu'elle non nulle.

Dimensions de la police

Certaines dimensions relatives à la police de caractère courante sont accessibles à travers des instructions de la forme `\fontdimen`⟨*p*⟩`\font`, où ⟨*p*⟩ est un entier entre 1 et 7. Le tableau suivant donne la signification de ces dimensions selon la valeur de ⟨*p*⟩

1. espace ajouté avant un accent sur un caractère penché.
2. composante naturelle du saut inter-mots.
3. composante **plus** du saut inter-mots.
4. composante **minus** du saut inter-mots.
5. hauteur du caractère `x`. C'est elle qui donne l'unité `ex`.
6. largeur du caractère `M`. C'est elle qui donne l'unité `em`.
7. dimension ajoutée à la composante naturelle du saut inter-mots

quand \spacefactor ⩾ 2000.

eTEX possède en outre des primitives qui permettent de récupérer individuellement les dimensions d'un caractère. Il s'agit de \fontcharwd, \fontcharht et \fontchardp. Pour la largeur d'un caractère ⟨*char*⟩ dans la police courante on utilise une instruction de la forme

\fontcharwd\font`\⟨*char*⟩

La hauteur et la profondeur s'obtiennent en remplacant le suffixe wd par ht et par dp dans le nom de la primitive.

```
\the\fontcharht\font‘\x\par
\the\fontcharwd\font‘\M\par
\sffamily
\the\fontcharht\font‘\x\par
\the\fontcharwd\font‘\M
```

4.56067pt
9.18704pt
4.84537pt
9.70169pt

Sauter à la française • Les format plainTEX et LATEX définissent la commande \frenchspacing qui oblige les espaces après une ponctuation d'être des espaces inter-mot normaux. Elle est définie par

```
\def\frenchspacing{\sfcode‘\.\@m \sfcode‘\?\@m \sfcode‘\!\@m
\sfcode‘\:\@m \sfcode‘\;\@m \sfcode‘\,\@m}
```

\frenchspacing ne fait donc que donner la valeur 1000 à tous les **sfcode** des caractères de ponctuation. La commande \nonfrenchspacing permet de rétablir les valeurs par défaut de ces **sfcode**

```
\def\nonfrenchspacing{\sfcode‘\.3000\sfcode‘\?3000
\sfcode‘\!3000\sfcode‘\:2000\sfcode‘\;1500\sfcode‘\,1250}
```

Avec ces valeurs les espaces après certaines ponctuations sont plus grand que d'autres, eux même plus grands que l'espace inter-mot normal.

```
\frenchspacing
\hbox to 8cm{Un, deux et trois. Et 4 alors? \the\sfcode‘\?}
\nonfrenchspacing\hrule
\hbox to 8cm{Un, deux et trois. Et 4 alors? \the\sfcode‘\?}
```

Un, deux et trois. Et 4 alors? 1005
Un, deux et trois. Et 4 alors? 3000

Noter qu'une façon d'arriver au même résultat que \frenchspacing est de donner à \xspaceskip la valeur du saut inter-mot normal

```
\xspaceskip=\fontdimen2\font plus \fontdimen3\font minus
 \fontdimen4\font
```

TEST POUR L'ITALIQUE • La primitive \fontdimen est utilisé d'une façon intéressante par certaines commandes LaTeX. Le test

```
\ifdim\fontdimen1\font > 0pt
```

peut servir pour savoir si la police active est en italique. C'est ce que fait la commande \em qui permet d'écrire une portion de texte en italique dans un contexte de police droite et vice versa. Elle est définie par ce qui revient à

```
\def\em{\ifdim \fontdimen1\font >0pt
 \upshape \else \itshape \fi}
```

ESPACE INTER-MOT TROP LARGE • Certaines polices de caractères disponible à l'utilisation avec LaTeX ont un espace inter-mots trop large. C'est le cas par exemple de la police **Venturis**, une alternative plus complète au paquet disponible pour la fameuse **Utopia**. Elle peut être utilisée globalement à l'aide du package venturis ou localement avec son nom interne yvt et la commande \fontfamily.

*À part **Computer Modern**, aucune des polices fournies dans le paquet de base **NFSS** ne possède de* vraies *petites capitales.*

```
\newcommand\fntd[1]{\fontdimen#1\font}
\fontfamily{yvt}\selectfont
Du texte \'ecrit dans la police Venturis. L'espace inter-mot
 par d\'efaut est trop large. On peut le modifier gr\^ace \`a
 \verb+\spaceskip+ et \verb+\xspaceskip+.\par\medskip
\spaceskip .5\fntd2 plus .5\fntd3 minus .5\fntd4
 \xspaceskip=\spaceskip
Du texte \'ecrit dans la police Venturis. L'espace inter-mot
 par d\'efaut est trop large. On peut le modifier gr\^ace \`a
 \verb+\spaceskip+ et \verb+\xspaceskip+.\par
```

Du texte écrit dans la police Venturis. L'espace inter-mot par défaut est trop large. On peut le modifier grâce à \spaceskip et \xspaceskip.

Du texte écrit dans la police Venturis. L'espace inter-mot par défaut est trop large. On peut le modifier grâce à \spaceskip et \xspaceskip.

RETRAITS ET JUSTIFICATION

Précisons qu'en imprimerie et en typographie on appelle justification d'un texte la largeur maximale que peut avoir une ligne. Lorsque les lignes ont la même largeur on parle de justification forcée, si le texte est aligné à gauche mais les lignes ont des largeurs différentes on parle de fer à gauche et de drapeau à droite, de fer à droite et de drapeau à gauche dans le cas

contraire. Dans ce document on utilisera toutefois le vocabulaire courant (même si c'est à tord…) de texte « justifié à gauche » au lieu de « ferré à gauche ».

Ce ne sont pas `\leftskip` *et* `\rightskip` *qui sont utilisés pour obtenir les retraits dans un environnement de liste LaTeX.*

Les ressorts natifs `\rightskip` et `\leftskip` fixent les dimensions des retraits à droite et à gauche de chaque ligne. Ils sont comptabiliés parmi les sauts de la ligne et ne modifient donc pas la largeur de celle-ci. Ces ressorts sont par défaut nuls.

```
Valeur initiale de hsize : \the\hsize.\par
\leftskip=1pc\rightskip=1pc
Les retraits changent, mais pas hsize :
 \the\hsize.
```

Valeur initiale de hsize : 159.3356pt.

Les retraits changent, mais pas hsize : 159.3356pt.

En outre, TEX ajoute le saut `\parfillskip` à la liste à la fin de chaque paragraphe. Sa valeur par défaut est `0pt plus 1fil`, ce qui a pour effet de faire prendre leurs dimensions naturelles aux ressorts d'ordre fini dans la dernière ligne.

TP **UTILISER UN FER •** C'est en utilisant les variables de dimension `\leftskip` et `\rightskip` qu'on peut avoir un effet de texte justifié seulement à gauche, seulement à droite ou bien centré. Pour illustrer ces propos, regardons comment est définie la commande LaTeX `\centering`.

```
\newskip\@flushglue \@flushglue = 0pt plus 1fil
\def\centering{%
  \let\\\@centercr
  \rightskip\@flushglue\leftskip\@flushglue
  \parindent\z@\parfillskip\z@skip}
```

Comme on le voit effectivement sur cette définition, LaTeX leurs donne la valeur `0pt plus 1fill`. On peut alors deviner que `\raggedright` donne à `\rightskip` la même valeur et conserve la valeur nulle de `\leftskip`, et que `\raggedleft` fait le contraire.

```
Du texte normal, justifi\'e des deux
 c\^ot\'es. C'est le comportement par d\'efaut
 sous plain\TeX{} et \LaTeX.\par
\rightskip=0pt plus 1fil
Du texte justifi\'e \`a gauche. Il se doit
 d'\^etre suffisamment fourni
en termes longs  pour pouvoir constater la
 diff\'erence de largeur des lignes et
 l'absence de c\'esures.\par
\leftskip=0pt plus 1fil
Du texte centr\'e maintenant, en cumulant les
 deux valeurs des retraits.
```

Du texte normal, justifié des deux côtés. C'est le comportement par défaut sous plainTEX et LATEX.
Du texte justifié à gauche. Il se doit d'être suffisamment fourni en termes longs pour pouvoir constater la différence de largeur des lignes et l'absence de césures.

Du texte centré maintenant, en cumulant les deux valeurs des retraits.

Le package ragged2e *permet de justifier des paragraphes tout en tolérant les césures. Il est utilisé pour les notes de marge dans ce document.*

Noter que donner des valeurs infinies aux ressorts `\rightskip` et/ou à `\leftskip` va donner aux espaces inter-mots leurs dimension naturelle mais malheureusement, rendre toute césure en fin de ligne impossible. Une justification du cote gauche seulement est souvent utilisée lorsque la largeur de la ligne est réduite (avec plusieurs colonnes par exemple) ce qui permet d'éviter que les vides ne soient trop étirés afin de maintenir la justification à droite. Mais en empêchant les césures, les lignes vont avoir des largeurs apparentes très disparates. Ce problème rend l'utilisation de `\raggedright` inadéquate pour de grandes portions de texte.

Un effet pour titres • On voudrait maintenant obtenir une variante de l'effet de `\centering` : un paragraphe reste normalement justifié à l'exception de sa dernière ligne qui doit être centrée (pratique pour des titres longs par exemple). L'idée est de donner respectivement à `\leftskip` et `\rightskip` des composantes `plus` de `1fil` et `-1fil` de telle façcon qu'ils se compensent et ne produisent plus aucun effet. Il suffit alors de donner à `\parfillskip` une composante `plus` de `2fil` pour que les ressorts finaux des deux côtés de la dernière ligne se retrouvent également étirables.

```
\leftskip=0pt plus 1fil\relax
\rightskip=0pt plus -1fil\relax
\parfillskip=0pt plus 2fil
Un paragraphe justifi\'e normalement \`a
 l'exception de la derni\`ere ligne qui se
 retrouve centr\'ee.
```

Un paragraphe justifié normalement à l'exception de la dernière ligne qui se retrouve centrée.

Fer à gauche, drapeau à droite • Une technique qui permet de produire du texte non justifié à droite consiste en donner aux sauts inter-mots des dimensions rigides. Mais si on se limite à cela, les lignes vont avoir la plupart du temps une **badness** infinie à cause des pouvoirs

d'étirement et de rétrécissement nuls. Une astuce pour complèter la technique est de donner au saut `\rightskip` une valeur élastique (finie, pour ne pas empêcher les césures). Cette méthode est celle utilisée par la commande `\raggedright` dans sa version plainTEX. Cette version donne de bien meilleurs résultats que celle de LATEX dans le cas d'une largeur de ligne étroite car elle n'interdit pas les césures. Malheureusement elle la version LATEX est la seule disponible.

Le package `ragged2e` *utilise la technique originale de plainTEX.*

Du texte dans une boite verticale très serrée pour comparer l'annulation de la justification à droite selon les méthodes plainTEX et LATEX.

Du texte dans une boite verticale très serrée pour comparer l'annulation de la justification à droite selon les méthodes plainTEX et LATEX.

La colonne à droite est traitée avec `\raggedright` version LATEX. La colonne à gauche imite la technique plainTEX en utilisant `\spaceskip=.3em`, `\xspaceskip=.5em` et `\rightskip=0pt plus .5em`

Algorithmes et méthodes

Allons donc ! Ce texte démérite-t-il vraiment ?

Dans les sections précédentes, on a signalé que les pénalités ont un rôle primordial dans le choix des points de rupture des lignes dans un paragraphe, tout autant que la **badness** de chaque possibilité de ligne. Il est temps de voir comment exactement.

Un point de rupture dans un paragraphe ne peut survenir qu'au niveau

— d'une pénalité, sans autre condition ;
— d'un ressort, s'il est précédé de matériel non résiliable ;
— d'un espace de crénage (avec la commande `\kern`), s'il est immédiatement suivi d'un ressort.

Rappelons que les seules pénalités implicites du mode horizontal sont celles concernant les variables `\hypenpenalty` et `\exhypenpenalty` qui sont insérées après un caractère de césure implicite ou explicite et que, par

ailleurs, une formule mathématique peut être rompue en fin de ligne après un symbole d'opérateur ou de relation selon les variables `\binoppenalty` et `\relpenalty`.

Maintenant, pour chaque possibilité de ligne, TEX calcule la **badness** de la `\hbox` dont le contenu est celui de la ligne. Il rejete, autant que possible, tout choix qui donne une **badness** supérieure à la variable `\pretolerance` ou `\tolerance` selon que la ligne se termine par une césure ou non. Il y a des situations où ces conditions sont impossibles à remplir et TEX se résigne à retenir quelques possibilités avec une mauvaise **badness**. Celles par exemple où une ligne se termine par du contenu indivisible d'une largeur conséquente (une `\hbox` ou une formule mathématique par exemple) et qu'envoyer ce contenu sur la ligne suivante donne à la ligne courante une **badness** trop grande. Dans ces conditions il est plus fréquent d'obtenir des lignes `Overfull \hbox` que des lignes `Underfull \hbox`.

Outre cette étape préliminaire du tri, TEX calcule pour chaque possibilité de ligne ainsi que pour certaines combinaisons de lignes une valeur numérique appelée démérite et additionne ensuite toutes ces valeurs pour obtenir le démérite de la configuration globale du paragraphe. Parmi toutes les possibilités testées, celle qui a le démérite le plus bas est retenue.

Si on note p l'entier dont la valeur est celle de la pénalité au point candidat à la rupture si celui-ci se fait au niveau d'une pénalité et qui vaut 0 sinon, b la **badness** de la ligne éventuelle, le démérite d d'une possibilité de ligne se calcule grâce à la formule

$$d = \begin{cases} (l+b)^2 + p^2 & \text{si} \quad 0 \leqslant p < 10000 \\ (l+b)^2 - p^2 & \text{si} \quad -10000 < p < 0 \\ (l+b)^2 & \text{si} \quad p \leqslant -10000 \end{cases}$$

où l représente la variable numérique `\linepenalty` dont la valeur par défaut est 10. On constate que le cas où $p \geqslant 10000$ n'est pas pris en compte dans cette formule puisque de toute façon un tel point n'est jamais choisi pour une rupture. L'expression de d dans le cas où $p \leqslant -10\,000$ semble étrange comparée aux autres cas. Elle est en fait simplifiée pour éviter un calcul inutile puisqu'un point où $p \leqslant -10\,000$ sera toujours choisi comme point de rupture.

Le démérite supplémentaire qui s'ajoute à la somme de ceux des lignes peut être

`\adjdemerits` démérite du cas où deux lignes adjacentes ont des vides visuellement incompatibles (trop serrés pour l'une, relâchés pour l'autre par exemple), se basant sur une comparaison de leurs **badness** ;

`\doublehyphendemerits` démérite du cas où deux lignes successives se terminent par des césures ;

`\finalhyphendemerits` démérite du cas où l'avant-dernière ligne se termine par une césure.

Ces démérites doivent peser autant que les carrés des pénalités et des **badness** pour qu'il puisse avoir un effet. Leurs valeurs par défaut dans le même ordre que leur citation sont 10 000, 10 000 et 5 000. Ce qui fait que de telles combinaisons ne se produisent qu'en des situations vraiment extrêmes où, par exemple, la **badness** cumulée des lignes dans les autres possibilités est trop grande.

Mais ce n'est pas tout, TEX effectue en fait le processus de formation de lignes d'un paragraphe en trois passes

La première passe est effectuée sans aucune information de césure, la badness des lignes ne devant pas dépasser `\pretolerance`. Si le processus réussit à obtenir une configuration convenable sans `(Over|Under)full \hbox`, TEX passe au paragraphe suivant. Pour cette passe, aucune pénalité n'est implicitement insérée dans la liste horizontale (sauf celles du mode **inlinemode**). Le plus souvent donc, les ruptures de lignes s'effectuent sur des sauts. Si `\pretolerance` a une valeur négative, cette passe est sautée. À l'opposé, une situation où cette passe se termine toujours par un succés serait d'avoir une `\badness` nulle pour toute possibilité de ligne. C'est ce qui arrive quand `\leftskip` ou `\rightskip` est un ressort d'ordre infini et c'est ce qui exclut dans ce cas toute possibilté de césure en fin de ligne.

La deuxième passe est effectuée en ajoutant d'abord toutes les informations de césure dans la liste. La **badness** ne doit pas dépasser `\tolerance` cette fois. La valeur par défaut des pénalités associées à une rupture au niveau d'une césure est de 50, donc une telle rupture n'est éventuellement préviligiée que si les possibilités voisines sans césure ont des **badness** qui dépassent la sienne de 50.

La troisième passe est effectué si les deux précédentes échouent à obtenir un résutat satisfaisant. Elle implique l'utilisation de la variable de dimension `\emergencystretch`. Celle-ci est systématiquement ajoutée au pouvoir d'étirement des lignes, ce qui diminue artificiellement les valeurs des **badness** et peut ainsi rendre acceptable une possibilité auparavant non tolérée. Aucun saut n'est ajouté, ce sont ceux sont qui sont déjà présents qui peuvent s'entendre un peu plus. Si `\emergencystretch` est nul (valeur par défaut) alors cette passe n'est pas effectuée et TEX se résigne à produire le résultat de la deuxième passe même s'il n'est pas satisfaisant. La troisième passe a pour effet de minimiser les situations de lignes `Overfull \hbox`.

> Les variables \tolerance, \pretolerance et \emergencystretch déterminent l'aspect final du texte dans le document compilé contrairement à \hbadness et \hfuzz qui n'ont qu'un rôle informatif.

*Les logiciels de traitement de texte usuels adoptent une approche de formation de paragraphes ligne par ligne. Seuls les logiciels professionnels de **PAO** utilisent une approche globale à la TEX.*

Tout ce procédé, établi de façon empirique, s'avère très efficace et participe à faire de TEX une référence dans son domaine. Les algorithmes de formation de paragraphe partiellement exposés ici ont une complexité presque linéaire. Les meilleurs algorithmes connus ont une compléxité linéaire mais aucun qui soit plus efficace que ceux de TEX n'a été effectivement implémenté.

TP **FAIRE LE FÉNÉANT** • Pour savoir comment utiliser toute cette profusion de paramètres, rien de tel que de voir comment LATEX en fait usage dans certaines circonstances. Regardons les définitions des commandes \sloppy et \fussy.

```
\def\sloppy{\tolerance 9999%
\emergencystretch 3em%
\hfuzz .5\p@
\vfuzz\hfuzz}
```

La commande \sloppy est utilisé par exemple dans toutes les boites verticales LATEX. Elle indique au moteur d'être très tolérant en donnant de grandes valeurs à \tolerance et \emergencystretch et d'accepter avec moins de rigueur les boites Overfull \hbox en donnant à \hfuzz une valeur plus grande que celle par défaut. Le message Undefull \hbox n'est presque jamais lancé mais si la largeur disponible est trop étroite, le texte produit sous l'effet de la commande \sloppy a en général des vides trop relachés et il faut souvent réarranger les phrases pour lui donner un aspect moins atroce. La commande \fussy ne fait que restaurer les paramètres par défaut de LATEX.

```
\def\fussy{%
\emergencystretch\z@
\tolerance 200%
\hfuzz .1\p@
\vfuzz\hfuzz}
```

La figure 5.1, page 91, donne des exemples où on teste l'effet des paramètres \tolerance et \emergencystretch dans une boite parbox de largeur assez réduite. Les gros traits noirs en fin de certaines lignes indique les boites Overfull \hbox. Ce résultat est obtenu en donnant une valeur non nulle à la variable de dimension \overfullrule.

Ce texte est traité dans une boite parbox. LaTeX place automatiquement une commande \lossy dans ses boites verticales. Mais ici on teste différents réglages des paramètres tolerance et emergencystretch.

1 \tolerance=9999 \emergencystrtech=3em

Ce texte est traité dans une boite parbox. LaTeX place automatiquement une commande \lossy dans ses boites verticales. Mais ici on teste différents réglages des paramètres tolerance et emergencystretch.

2 \tolerance=200 \emergencystrtech=0pt

Ce texte est traité dans une boite parbox. LaTeX place automatiquement une commande \lossy dans ses boites verticales. Mais ici on teste différents réglages des paramètres tolerance et emergencystretch.

3 \tolerance=200 \emergencystrtech=3em

Ce texte est traité dans une boite parbox. LaTeX place automatiquement une commande \lossy dans ses boites verticales. Mais ici on teste différents réglages des paramètres tolerance et emergencystretch.

4 \tolerance=9999 \emergencystrtech=0pt

Ce texte est traité dans une boite parbox. LaTeX place automatiquement une commande \lossy dans ses boites verticales. Mais ici on teste différents réglages des paramètres tolerance et emergencystretch.

5 \tolerance=1400 \emergencystrtech=.5em

Ce texte est traité dans une boite parbox. LaTeX place automatiquement une commande \lossy dans ses boites verticales. Mais ici on teste différents réglages des paramètres tolerance et emergencystretch.

6 \tolerance=1400 \emergencystrtech=.5em

Figure 5.1 Influence de \tolerance et \emergencystretch. Le premier et le deuxième exemple utilisent respectivement les réglages de \lossy et \fussy, le troisième et le quatrième croisent les réglages de ces deux commandes et le cinquième essaye de trouver un compromis qui donne un meilleur résultat sans produire des Overfull \hbox. Une fois le problème identifié on modifie ces variables par petits paliers. L'exemple 2 montre que le réglage induit par \fussy est inapproprié quand la largeur de ligne est réduite. Le sixième exemple (à comparer au cinquième) est traité avec la **protrusion** et l'**expansion** activées pour rendre compte de l'efficacité de microtype.

Microtype, il sert à quoi ?

La microtypgraphie est l'art de donner au texte un aspect plus agréable à l'œil et plus facilement lisible sans qu'on puisse dire exactement ce qui en est responsable, les améliorations étant tellement fines qu'elles sont indétectables. Le moteur original TEX a déjà des compétences certaines qui vont dans ce sens mais pdfTEX va plus loin en implémentant de vraies notions de microtypographie. Ces fonctionnalités sont accessibles à travers des commandes primitives mais ne sont pas activées par défaut car le résultat qu'elles produisent n'est pas le même que celui que produit le moteur TEX. Le package `microtype` permet de le faire de façon simple est transparente. Les techniques majeures exploitables à travers lui sont :

Protrusion ou **Margin kerning**, une technique qui consiste en repousser certains (petits) caractères dans la marge pour renforcer l'impression de justification à droite du texte. Il s'agit en général des caractères de ponctuation et de celui utilisé pour les césures (en-dash). Cette technique s'active grâce à l'option `protrusion` de `microtype`.

Expansion technique qui va très légérement agrandir ou rétrécir la largeur de certains caractères pour conserver des espaces inter-mots plus homogènes. Cette technique s'active grâce à l'option `expansion` de `microtype`.

Tracking ou **Letterspacing** technique qui permet d'allonger ou de réduire uniformément les espaces entre les lettres comme dans « du `texte en letterspacing` ». Cette technique peut être activée globalement grâce à l'option `tracking` de `microtype` et s'applique alors par défaut seulement au texte écrit en PETITES CAPITALES. Les commandes `\textls` et `\lsstyle`, qui s'utilisent respectivement comme `\textit` et `\itshape`, permettent de n'appliquer l'effet qu'à une portion de texte, même si l'option n'est pas activée.

*Dans le mode **DVI**, seule la **protrusion** est activée par défaut.*

Avec le moteur pdfTEX en mode **PDF**, la **protrusion** et l'**expansion** sont activées par défaut lors du chargement de `microtype`. On peut les désactiver en passant la valeur `false` aux clés `protrusion` et `expansion` lors du chargement du package ou le faire localement (dans un groupe TEX) grâce à la commande `\microtypsetup`

```
\microtypesetup{protrusion=false,expansion=false}
```

Activer les fonctions de microtypgraphie du moteur permet d'avoir des vides distribués d'une façon très homogènes sur chaque ligne et ce même dans des situations très difficiles. Comme conséquence, les messages

Détail à prendre en compte avant d'effectuer la dernière étape qu'est la correction de la mise en page verticale dans la préparation d'un document.

(Over|Under)full \hbox deviennent beaucoup moins fréquents. Toutefois les points de rupture des lignes ne sont pas les mêmes que ceux qu'aurait produit le moteur sans ces fonctionnalités, et par suite les ruptures de pages ne sont pas forcément les mêmes. Pour indiquer à microtype de ne pas changer la configuration des lignes tout en activant la **protrusion** et l'**expansion**, il suffit de passer la valeur compatibility aux clés protrusion et expansion. On peut aussi utiliser la clé activate qui se contente de relayer les valeurs qu'on lui passe à ces deux clés. Le résultat produit de cette manière est bien sûr suboptimal.

```
\usepackage[activate={true,compatibility}]{microtype}
```

Voici maintenant, côté à côté, un même paragraphe traité d'abord avec la protrusion désactivée et l'expansion en mode compatibility et ensuite avec la **protrusion** et l'**expansion** activées de façon normale.

Du texte dans une boite parbox. Dans la boite à gauche ce sont les réglages par défaut dans une boite verticale LaTeX. Dans la boite à droite, l'expansion et la protrusion sont activées grâce au package microtype. Le tout avec une largeur de ligne assez serrée.

Du texte dans une boite parbox. Dans la boite à gauche ce sont les réglages par défaut dans une boite verticale LaTeX. Dans la boite à droite, l'expansion et la protrusion sont activées grâce au package microtype. Le tout avec une largeur de ligne assez serrée.

Le résultat est sans appel en faveur de la microtypographie. Mais noter qu'effectivement, il y a une ligne en moins dans le paragraphe à droite.

Dessiner des paragraphes

On peut changer certains aspects dans la formation de paragraphes à travers plusieurs paramètres et structures primitives du moteur TEX.

La commande \par peut être redéfinie pour obtenir des effets supplémentaires quand un paragraphe est fermé. On peut par exemple conserver la signification originale de \par en utilisant la commande \let et lui ajouter ensuite d'autres effets, technique très commune dans la programmation TEX/LaTeX, notamment dans le noyau LaTeX lui même.

```
\let\PAR\par \def\par{\dotfill\hskip0pt\PAR}
Un premier paragraphe qui se termine vite.

Un second qui dure un peu plus longtemps mais
 qui reste assez court.
```

Un premier paragraphe qui se termine vite. . . . . .
Un second qui dure un peu plus longtemps mais qui reste assez court. . . . . .

COMMENTAIRES : `\hskip0pt` sert à protéger `\dotfill` de l'action de `\unskip` exécutée à la fin du paragraphe. `\dotfill` utilise les leaders et ces derniers sont éléminés par `\unskip` au même titre que les **skip**.

La commande `\par` est exécutée à la fin de chaque paragraphe. Qu'en est-il de la possibilité d'exécuter une action au début de chaque paragraphe ? Ceci est possible à travers la « commande »`\everypar` dont le contenu est inséré au début de chaque paragraphe. `\everypar` est en fait ce qu'on appelle une « token list ».

> Une « token list » est une structure pouvant retenir du code qu'on peut ensuite insérer à volonté grâce à la commande `\the`. TEX exécute par exemple implicitement l'instruction `\the\everypar` au début de tout paragraphe. Le contenu d'une « token list » peut être redéfini par un assignement : avec l'opérateur = ou simplement en faisant suivre le nom de la liste du nouveau contenu.

```
\everypar={\textbullet\textbullet\enspace}
Un premier paragraphe.\par
Et un deuxi\‘eme qui commence de la m\^eme
 fa\c con.
```

•• Un premier paragraphe.
•• Et un deuxième qui commence de la même façon.

Seulement les choses ne sont pas aussi simples car LATEX redéfinit le contenu de `\everypar` et le sens de la commande `\par` au début d'un grand nombre de ses environnements, par exemple `itemize` et `enumerate`.

INDENTATION DE PARGARAPHES

La dimension `\parindent` fixe l'espace d'indentation au début d'un paragraphe. Mais il faut savoir que cet espace n'est pas un saut mais une boite **hbox** vide de largeur `\parindent`. Pour éliminer définitivement l'indentation de paragraphes au cours d'un certain contenu, il ne suffit pas de donner une valeur nulle à `\parindent` car certains environnements peuvent lui affecter localement une autre valeur. Une façon d'y arriver quelque soit la valeur de `\parindent` est de vider la boite d'indentation au début de chaque nouveau paragraphe. Pour cela il suffit d'utiliser une instruction de la forme `\setbox0=\lastbox` juste au début du paragraphe pour enlever la boite responsable de l'indentation de la liste en cours. Pour que le registre de boite 0 ne soit pas modifié non plus,

il suffit d'inclure cette instruction dans un groupe. Maintenant pour que l'action perdure il suffit d'utiliser la token list `\everypar`

Des paragraphes sans indentation.
Et ce quelque soit la valeur de la variable parindent.

```
\everypar={{\setbox0=\lastbox}}
\parindent=1.3em
Des paragraphes sans indentation.\par
Et ce quelque soit la valeur de la
 variable \texttt{parindent}.
```

Bien sûr, tout cela tombe à l'eau si on l'utilise au sein d'un environnement qui modifie lui même `\everypar`.

TEX dispose en outre de techniques primitives qui permettent de changer localement la façon dont sont construits les paragraphes.

`\hangindent` et `\hangafter`

ces deux variables numériques fonctionnent de concert. La variable de dimension `\hangindent` précise l'espace d'indentation à appliquer à un certain nombre de lignes, nombre qui est précisé avec la variable `\hangafter`. Les deux peuvent recevoir des valeurs positives ou négative. Si `\hangindent` est positive, l'indentation se fait à gauche, sinon elle se fait à droite. Si `\hangafter` a pour valeur absolue n alors l'effet s'applique aux n premières lignes si elle est négative et à partir de la $(n+1)$-ième ligne si elle est positive. L'effet ne dure que le temps d'un paragraphe.

```
\hangindent=2pc \hangafter=-2
Un paragraphe avec une indentation sp\'eciale.
  Effet qui peut servir de base \`a
 l'utilisation de  lettrines.
\par\medskip \hangindent=-3pc \hangafter=2
Maintenant, c'est un effet encore plus
 inhabituel. Mais on peut toujours lui trouver
 des applications int\'eressantes
```

Un paragraphe avec une indentation spéciale. Effet qui peut servir de base à l'utilisation de lettrines.

Maintenant, c'est un effet encore plus inhabituel. Mais on peut toujours lui trouver des applications intéressantes

`\parshape`

C'est une variable numérique spéciale de TEX. Si on lui affecte un nombre entier n, TEX s'attend alors à lire $2n$ dimensions qui vont préciser, par couples, une indentation et la largeur du texte sur chacune des n premières lignes d'un paragraphe. Si le paragraphe comporte plus que n lignes, l'effet de la n-ième ligne dure pour le reste des lignes. C'est une technique qui généralise celle décrite ci-dessus et là encore l'effet n'est valable que pour le paragraphe en cours.

```
\def\lw{\linewidth} \parshape=6
    0pt \lw .4\lw .2\lw .3\lw .4\lw
    .2\lw .6\lw .1\lw .8\lw 0pt \lw
Un paragraphe  avec un dessin original.
 \verb+\parshape+ peut \^etre utilis\'ee de
 mani\'ere cr\'eative. C'est toutefois une
 commande utile pour obtenir des effets
 simples. Elle est par exemple utilis\'ee par
 \LaTeX{} pour former des titres.
```

Un paragraphe avec un dessin original.
\parshape peut être utilisée de manière créative. C'est toutefois une commande utile pour obtenir des effets simples. Elle est par exemple utilisée par LaTeX pour former des titres.

```
\bfseries \newdimen\lw \lw=\hsize
\setbox0=\hbox{LABEL\enspace}
\advance\lw-\wd0
\parshape=2 0pt \linewidth \wd0 \lw
 \raggedright
\unhbox0 Un paragraphe qui peut faire office
 de titre, selon la m\'ethode utilis\'ee par
 \LaTeX{} lui m\^eme.
```

LABEL Un paragraphe qui peut faire office de titre, selon la méthode utilisée par LaTeX lui même.

\hangindent, \hangafter et \parshape fonctionnent indépendamment de l'indentation de paragraphes. Si cumuler leurs effets n'est pas désiré, il suffit d'utiliser la commande \noindent au début du paragraphe.

Faire perdurer les retraits • On voudrait créer un environnement qui puisse traiter plusieurs paragraphes et produire des retraits non nuls à gauche et à droite du texte.

```
\newdimen\linejust
\newenvironment{adjustpar}[2]
  {\par\setlength{\linejust}{\linewidth-#1-#2}
  \everypar{{\setbox0=\lastbox}%
              \parshape=1 #1 \linejust\relax}}
  {\par}
```

```
Du texte en pleine largeur de ligne.
\begin{adjustpar}{12pt}{12pt}
Texte avec retraits \'a doite et \'a
 gauche.\par
L'effet reste actif pour plusieurs paragraphes.
\end{adjustpar}
Retour aux r\'eglages par d\'efauts.
```

Du texte en pleine largeur de ligne.

Texte avec retraits à doite et à gauche.

L'effet reste actif pour plusieurs paragraphes.

Retour aux réglages par défauts.

Commentaires : On utilise ici la token list \everypar pour annuler l'indentation et établir la forme des paragraphes avec \parshape. Ce document utilise le package calc, ce qui permet à \setlength de faire des calculs.

Une autre façon (plus simple) d'arriver au même résultat est d'altérer les dimension \leftskip et \rightskip.

```
\newenvironment{paradjust}[2]
  {\par\leftskip=#1\rightskip=#2}
  {\par}
```

```
Du texte en pleine largeur de ligne.
\begin{paradjust}{12pt}{12pt}
Texte avec retraits \‘a doite et \‘a
 gauche.\par
L'effet reste actif pour plusieurs paragraphes.
\end{paradjust}
Retour aux r\'eglages par d\'efauts.
```

Du texte en pleine largeur de ligne.

Texte avec retraits à doite et à gauche.

L'effet reste actif pour plusieurs paragraphes.

Retour aux réglages par défauts.

Commentaires : Un environnement LaTeX crée un groupe autour de son contenu. L'effet de l'altération des retraits n'est donc valable que pour ce contenu. Cet environnement n'est toutefois pas très robuste. Si par exemple le contenu contient un environnement de liste alors ce dernier utilisera ces propres retraits.

Faire régurgiter à TEX le contenu de ces entrailles • Dans le but d'appliquer un certain effet à tout un paragraphe, on va récupérer toutes ses lignes et leurs appliquer ensuite l'effet individuellement. Ceci peut être réalisé grâce à la boite \lastbox et à un procédé récursif : on laisse TEX former le paragraphe ensuite on rècupère l'une après l'autre les \hbox des lignes qu'on retraite avec une macro préparée à cet effet.

```
\newbox\linebox \newbox\xlinebox \newdimen\wofline
%%% Commande qui forme la boite \xlinebox qui contient
%%% une ligne avec son effet.
\def\linewitheffect{%
  \setbox\linebox=\hbox{\strut\unhbox\linebox}%
  \ifdim\wd\linebox<.85\hsize%
    \wofline=\wd\linebox%
 \else \wofline=\hsize \fi%
 \advance\wofline2\fboxsep%
 \setbox\xlinebox=\hbox to \wofline{%
 \color{black!25}%
 \vrule width\wofline height \ht\linebox depth \dp\linebox%
 \normalcolor%
 \hskip-\wofline\hskip\fboxsep%
 \unhbox\linebox\kern\fboxsep}}
%%% commande qui sera appel\'ee \‘a la fin de chaque
 paragraphe
```

```
\def\effectpar{\setbox\linebox=\lastbox%
 \ifvoid\linebox\else% si le dernier \'el\'ement est une boite
   \unskip\unpenalty%  enl\'eve ce qui la pr\'ec\'ede
 {\effectpar}% appel recursif, le groupe assure
  l'ind\'ependance
             % des boites \linebox interm\'ediaires
 \linewitheffect% forme la boite \xlinebox : ligne avec effet
 \box\xlinebox% ins\'ere \xlinebox dans la liste verticale
 \hrule width\wofline% trace un filet de largeur \wofline
 \fi}
%%% environnement utilisateur qui \'etablit durablement
 l'effet
\newenvironment{pareffect}
  {\par\medskip% passe en mode vertical
  \let\PAR\par\def\par{\PAR\effectpar}% redefinit \par
     % pour qu'il execute \effectpar
  \vbox\bgroup\advance\hsize-2\fboxsep% initier une \vbox qui
     % va retenir tout le contenu avant de le traiter. \vbox
     % est indispensable, sinon \lastbox sera toujours vide.
  }
  {\par\egroup}
```

```
\begin{pareffect}
Du texte avec un certain effet qui peut
 s'appliquer \'a plusieurs paragraphes.

Le but est de montrer comment r\'ecup\'erer du
 contenu est le retraiter avant son rendu
 final. Cet environnement n'est pas tr\'es
 robuste mais son contenu peut s'\'etaler sur
 plusieurs pages.
\end{pareffect}
```

Du texte avec un certain effet qui peut s'appliquer à plusieurs paragraphes.
Le but est de montrer comment récupérer du contenu est le retraiter avant son rendu final. Cet environnement n'est pas très robuste mais son contenu peut s'étaler sur plusieurs pages.

Le mode displaymode

On ne peut être complet sur le traitement des paragraphes par TEX sans toucher un mot sur le mode mathématique **displaymode**.

(ACOMP)

Les pages, en profondeur

Allez ! On tourne la page

L'algorithme

En consultant la liste verticale principale, TEX ne peut rompre une page que

— au niveau d'une pénalité, sans condition ;
— au niveau d'un **vskip** s'il est précédé de matériel non résiliable ;
— au niveau d'un **kern** s'il est suivi d'un **vskip**.

c'est-à-dire, exactement au même genre de points en lesquels un paragraphe est éclaté en lignes. La différence se situe maintenant au niveau de la nature des points effectivement choisis. TEX insère des pénalités dans la plupart des situations où il doit ajouter du matériel à la liste verticale, assez souvent donc une rupture de page survient au niveau d'une pénalité.

Rappelons que TEX maintient deux listes dans le mode vertical. La liste « recent contibutions » qui contient tout les élements traités dans ce mode et la liste « current page » qui reçoit du matériel provenant de la première à certains moments dont les plus fréquents sont

— à chaque début ou fin d'un paragraphe ;
— à chaque entrée ou sortie du mode **displaymode** ;
— à chaque instruction de pénalité explicite ;

Là ou TEX utilise la notion de démérite pour former les lignes d'un paragraphe, il définit la notion de coût de la rupture lorsque il doit former une page. Ce coût est calculé pour la liste « page courante » à chaque fois que

des éléments y sont ajoutés. La formation de pages se compliquant considérablement si le document contient des éléments flottants, regardons l'expression du coût juste dans le cas où ces éléments sont absents :

$$c = \begin{cases} p & \text{si } b < \infty \text{ et } p \leqslant -10\,000 \\ b + p & \text{si } b < 10\,000 \text{ et } |p| < 10\,000 \\ 100\,000 & \text{si } b = 10\,000 \text{ et } |p| < 10\,000 \\ \infty & \text{si } b = \infty \text{ et } p < 10\,000 \end{cases}$$

où p est la pénalité en un point candidat à une rupture ($p = 0$ s'il n'y a pas de pénalité) et b est la **badness** de la boite verticale qui va contenir tous les éléments de la page éventuelle.

TEX va continuer à ajouter du matériel dans la liste « current page » tout en mémorisant à chaque fois la position du point qui a donné le coût minimal, jusqu'à ce qu'il rencontre une pénalité de valeur inférieure ou égale à $-10\,000$ (qui provoque une rupture immédiate) ou que le coût devienne infini (ce qui correspond à une **badness** infinie, c'est-à-dire à une situation d'`Overfull \vbox`). Il rompe ensuite la liste au point qui induit le coût minimal, le premier morceau est mis dans la boite `\box255` qui est passée à la routine de formation de pages et le reste est placé au somment de la liste « recent contributions ». On observe sur l'expression de c que le cas où $p \geqslant 10\,000$ n'est pas pris en compte, un tel cas correspond en effet à un point où la rupture n'est pas envisageable. En outre, le premier cas sera toujours favorisé au deuxième et celui-ci est systématiquement préviligié aux deux derniers. Dans des conditions normales, la rupture se fera donc en un point qui correspond au deuxième cas. Dans une telle situation, une page n'ayant pas assez d'éléments va avoir une grande **badness** et aura donc peu de chance d'être formée. Si on désire néanmoins provoquer une rupture de page en un tel lieu, il suffit d'y insérer une pénalité de valeur négative de plus en plus petite jusqu'à ce que la rupture devienne effective. Si jamais le contenu change et que ce lieu s'éloigne de la fin de la page, la pénalité choisie ne suffira plus pour compenser la nouvelle **badness** et la rupture se fera ailleurs.

Variables en rapport avec la formation de pages

Quelques variables de dimensions ont un rapport avec le processus de formation de pages dont certaines sont renseignées tout au cours du traitement de la liste verticale.

`\topskip` Ressort utilisé par le moteur pour que la première ligne d'une page soit toujours placée au même endroit. Le ressort effectivement placé

avant la première ligne à les mêmes composantes d'élasticité que \topskip, mais sa composante naturelle est égale à celle de \topskip moins la hauteur de la première ligne, à moins que cette différence soit négative, cas dans lequel la composante naturelle de ce premier ressort est mise à zéro.

\vsize Variable du moteur qui fixe la hauteur de la zone de texte.

\textheight Variable LaTeX qui fixe la hauteur de la zone de texte. Son changement n'a un effet que s'il se fait dans le prémabule. L'instruction \begin{document} l'utilise pour renseigner la variable \vsize ainsi que d'autres variables LaTeX internes.

\pagetotal Variable du moteur qui contient la hauteur du contenu empilé dans la liste « current page », composantes naturelles des sauts comprises. Elle est mise à ajour à chaque fois que du matériel est ajouté à la liste « current page ».

\pagegoal Variable du moteur renseignée au tout début de la formation de la liste « current page » et qui n'est plus changé ensuite. En condition normale, elle contient la hauteur de la zone de texte, sinon cette hauteur moins la hauteur totale des éléments flottants qui devraient être intégrés à la page.

\pagedepth variable du moteur qui contient la profondeur du dernier élément ajouté à la liste « current page » et qui va donner la profondeur de la boite qui va contenir le texte de la page si une rupture doit survenir.

\maxdepth variable du moteur qui fixe la profondeur maximale que peut avoir la boite d'une page.

\pagestretch variable du moteur qui contient la somme des composantes **plus** d'ordre 0 dans la liste « current page ». D'autre part, les variables \pagefilstretch, \pagefillstretch et \pagefilllstretch contiennent respectivement la somme des composantes **plus** d'ordre 1, 2 et 3.

\pageshrink variable du moteur qui contient la somme des composantes **minus** d'ordre 0 dans la liste.

La somme de \pagetotal et \pagedepth représente à tout moment la hauteur totale naturelle de la boite éventuelle formée du contenu de la liste « current page ». \pagegoal représente la hauteur encore disponible sur la page à l'instant ou une nouvelle liste « current page » est formée.

Commander la mise en page verticale

Dans cette section on donne une liste de commandes avec leurs prototypes d'utilisation ainsi que certaines variables de dimension. Elles ont toutes

un rapport avec le fonctionnement de TEX dans le mode vertical. Certaines commandes ont déjà été introduites. Elle sont reprises ici, parfois avec des descriptions étoffées, dans le but de former un petit dictionnaire rapidement consultable.

Légende :	⟨*skip*⟩	un **skip**
	⟨*dimen*⟩	une **dimen**
	⟨*num*⟩	un nombre
	⟨*cmds*⟩	un bloc de code
	⟨*pen*⟩	une valeur de pénalité

\vskip ⟨*skip*⟩

\vskip est une commande primitive verticale, ce qui signifie qu'elle ferme le paragraphe courant si le mode horizontal est actif. Le saut qu'on lui spécifie est ajouté au saut interligne et au saut de paragraphe. De ce fait l'instruction \vskip0pt ferme le paragraphe courant et provoque un saut interligne suivi d'un saut de paragraphe. En dehors du ressort déduit de \topskip et placé automatiquement par TEX au début de toutes les pages, un \vskip n'est jamais inséré au sommet de la liste verticale. Ce qui signifie que commencer une page avec un \vskip n'a aucun effet.

\vglue ⟨*skip*⟩

\vglue est une commande verticale du format plainTEX. Elle fonctionne comme \vskip, mais le saut qu'elle induit ne disparaît pas du haut de la page. De fait, elle commence par tracer un filet \hrule d'épaisseur nulle, interdit toute rupture de page en plaçant une penalité maximale et insère l'instruction \vskip adéquate.

\kern ⟨*dimen*⟩

\kern est une commande primitive TEX. Elle fonctionne dans les deux modes, vertical et horizontal. Dans le mode vertical elle a le même effet que \vskip, et dans le mode horizontal le même effet que \hskip, Elle diffère de ces deux commandes dans le fait qu'elle produit des sauts rigides et que jamais une ligne (dans le mode horizontal) ou une page (dans le mode vertical) n'est rompue au niveau d'un **kern** s'il est encadreé entre deux élements non résiliables. Dans un contexte horizontal, l'instruction \kern⟨*dimen*⟩ ne peut pour autant faire office d'espace insécable car le mot suivant sera placé sur la même ligne même s'il y a un Overfull \hbox. (le caractère actif ~ est plutôt équivalent à {\penalty10000\ }).

\vadjust{⟨*cmds*⟩**}**

\vadjust est une commande primitive TEX qui permet de faire migrer du contenu dans la liste verticale en cours alors qu'on est dans un contexte horizontal et sans en sortir. Le matériel en question est traité dans le mode

vertical interne (comme dans une \vbox) et il est inséré au début de la ligne qui suit la ligne en cours. C'est ainsi qu'un

```
\vadjust{\kern1cm}
```

va insérer une séparation verticale de 1cm pour la ligne suivante tout en restant dans le même paragraphe. Le gros point à gauche est lui inséré par l'instruction

```
\vadjust{\llap{\smash{\Huge\textbullet}}}
```

\\[⟨*skip*⟩**]**

c'est une commande LaTeX qui n'est utilisable que dans le mode horizontal et dont la définition change dans plusieurs environnement comme center ou tabular. Dans sa version normale, elle implique une rupture de ligne sans création d'un nouveau paragraphe (gâce à \vadjust) et dans la version redéfinie par les environnements center, flushleft et flushright elle utilise \par et donc crée un nouveau paragraphe, mais compense le saut de paragraphe. Quelque soit le contexte, l'argument optionnel sert à augmenter l'interligne de la valeur de celui-ci et la version étoilée * interdit qu'un saut de page ne survienne à son niveau.

la commande similaire \newline n'a ni paramètre optionnel ni version étoilée. Par contre sa signification reste toujours la même quelque soit l'environnement LaTeX utilisé.

\vspace{⟨*skip*⟩**}**

\vspace est une commande LaTeX. Elle se comporte différemment selon que le mode est vertical ou horizontal. Dans le mode vertical elle utilise \vskip pour insérer un saut de la dimension précisé. Dans le mode horizontal, elle utilise la primitive \vadjust pour insérer un espace vertical (après la fin de la ligne courante) sans quitter le mode horizontal. Dans tous les cas, le saut induit resiste à un éventuel \removelastskip (voir plus bas). Il est préférable de l'utiliser uniquement dans le mode vertical pour un comportement plus prévisible.

\vspace{⟨skip⟩} insère le saut indiqué suivi d'un saut de dimension nulle qui sera le seul vu par un éventuel \removelastskip.

Sa version étoilée \vspace* insère un espace non résiliable y compris au sommet de la page (comme \vglue).

\addvspace{⟨*skip*⟩**}**

\addvspace est une commande LaTeX dont le comportement peut être déroutant mais qui offre des fonctionnalités intéressantes. Elle ne peut agir que dans un contexte vertical et provoque une erreur sinon. Quand \addvspace suit immédiatement un saut (qui peut être implicite), seule l'espace maximal est retenu. En fait si le saut précédent est positif et plus petit que celui qu'elle veut ajouter, elle commence par annuler son effet avec \removelaskip, dans le cas contraire elle n'ajoute rien. Plusieurs environnements et commandes LaTeX l'utilisent pour gérer les espaces verticaux autour de leurs contenus, comme les commandes de titres, les en-

vironnements de listes, les environnements d'alignement comme `center`, `flushleft` ou `flushright`, les environnements pour théroèmes…. Insérer un saut vertical entre deux de ces environnement donne un résultat qui peut être inattendu. Par exemple, avec la séquence

```
\addvspace{12pt}\vskip3pt\addvspace{12pt}
```

on s'attend à ajouter un saut de `3pt` aux `12pt` résultant des deux `\addvspace`, mais on se retrouve avec un saut global de `24pt`. En effet, le premier `\addvspace` ajoute `12pt` et le second va annuler les `3pt` pour ajouter encore `12pt`. Pire si on utilise `\vspace` au lieu de `\vskip` on finit avec un saut total de `27pt` puisque le second `\addvspace` ne verra que le deuxième saut ajouté par `\vspace` et qui est nul. Une solution dans cette situation peut être de plutôt utiliser une séquence

```
\addvspace{12pt}\addvspace{15pt}\addvspace{12pt}
```

`\smallskip`

Commande définie par

```
\def\smallskip{\vspace\smallskipamount}
```

Il y a aussi les commandes `\medskip` et `\bigskip`, respectivement associées aux ressorts `\medskipamount` et `\bigskipamount`. Ces dimensions reçoivent comme valeurs par défaut :

`\smallskipamount`, `\medskipamount` *et* `\bigskipamount` *forment une progression géométrique.*

```
\smallskipamount = 3pt plus 1pt minus 1pt
  \medskipamount = 6pt plus 2pt minus 2pt
  \bigskipamount = 12pt plus 4pt minus 4pt
```

Ce sont des commandes plainTEX redéfinies par LATEX pour utiliser `\vspace` au lieu de `\vskip`. Ce qui implique qu'elles fonctionnent aussi dans le mode horizontal pour un résultat qui peut être déroutant, surtout si elles sont utilisées avec une commande au fonctionnement similaire comme `\\` ou `\vspace`.

```
Ce saut  qui se veut  vraiment  grand,\\
\bigskip n'est pas l\'a ou on l'attend.\par
Fichtre !
```

Ce saut qui se veut vraiment grand,
n'est pas là ou on l'attend.

Fichtre !

`\vfil` ou `\vfill`

Ce sont des commandes TEX. Elles insèrent des espaces nuls mais infiniment étirables, respectivement au premier et au deuxième ordre. Il n'y a pas de commande `\vfilll`. Il peut être plus souple d'utiliser une commande `\vskip` avec une composante `plus` infinie car cela autorise un facteur de multiplication qui diminue ou augmente la force d'étirement du saut.

`\vss`

C'est une commande TEX qui produit un espace nul mais infiniment extensible (au premier ordre) dans les deux sens. Elle est l'équivalent de :

```
\vskip 0pt minus 1fil plus 1fil
```

`\vss` est surtout utilisée pour ajuster le positionnement des éléments dans une boite verticale et pour absorber l'excédent de dimension dans la boite en cas d'« `Overfull \vbox` ».

`\lastskip`

c'est une variable de dimension renseignée continuellement par le moteur TEX. Si le dernier item ajouté à une liste (horizontale ou verticale) est un ressort alors sa valeur est affectée à `\lastskip`, dans le cas contraire `\lastskip` reçoit une valeur nulle. Dans le mode horizontal ou dans le mode vertical interne il est possible d'enlever le dernier **skip** de la liste avec la commande primitive `\unskip`. Ce n'est pas possible de le faire dans la plupart des cas dans le mode vertical principal.

`\removelastskip`

c'est une commande TEX. Il n'est souvent pas possible d'enlever le dernier saut dans la liste verticale principale. La commande `\removelastskip` compense en ajoutant un autre saut de valeur `-\lastskip` :

```
\def\removelastskip{\vskip-\lastskip}
```

`\removelastskip` n'annule pas un saut inséré par `\vspace`.

`\abovedisplayskip`=⟨*skip*⟩

`\abovedisplayskip` est le saut inséré au dessus d'une équation **displaymode**. Si la ligne précédant l'équation n'est pas assez longue pour chevaucher visuellement avec la formule par le haut, c'est plutôt le saut `\abovedisplayshortskip` qui est utilisé.

`\belowdisplayskip`=⟨*skip*⟩

C'est le ressort inseré au dessous d'une équation **displaymode**.

`\belowdisplayshortskip` est associée à `\abovedisplayshortskip`.

Ces quatres dimensions sont redéfinies à chaque appel des commandes de changement de corps de police `\normalsize`, `\small` ou `\footnotesize`. Ce comportement est fixé dans les fichiers de classes (article, report, book, …). Tout changement direct effectué sur ces dimensions sera perdu si l'une de ces trois commandes est appelée sans être protégée par un groupe. Sachant que la commande `\normalsize` est appelée implicitement dans beaucoup d'environnement LATEX, à commencer par l'instruction `\begin{document}`, la seule façon fiable de changer ces dimensions et de redéfinir les trois commandes.

`\allowdisplaybreaks`

Bascule `amsmath` qui rend possible la rupture en fin de page des équations multi-lignes, dans tout le groupe où elle intervient et à partir de son occurrence. Lorsque `\allowdisplaybreaks` est active, l'instruction `\\*` peut être utilisée pour empêcher une rupture de page là où elle n'est pas désirée.

\newpage
c'est une commande qui impose une rupture de page, mais après avoir placé une instruction \vfil. Ce qui fait que le contenu de la page courante occupera sa hauteur naturelle.

\clearpage
c'est une commande qui agit comme \newpage. Elle s'assure en plus que tous les éléments flottants en suspens (figures, tables, notes de pied de page et notes de marge) prennent place avant la nouvelle page.

\cleardoublepage
commande qui fonctionne comme \clearpage et ne s'en distingue que dans le cas où le document est en recto-verso (option de classe twoside). Dans ce cas elle s'assure que la page suivante est une page de droite (numéro de page impaire), quitte à laisser une page vide.

\pagebreak[⟨*num*⟩**]**
\pagebreak est une commande LaTeX. L'argument optionnel est un nombre qui doit être compris entre 0 et 4, sa valeur par défaut étant 4, auquel cas elle force une rupture de page. Elle permet d'encourager une rupture de page d'autant plus fortement que cet argument est grand. En interne, elle insère une pénalité dont la valeur est calculée en fonction de l'argument optionnel selon le schémas :

0 ⟶ 0	1 ⟶ − \@lowpenalty
2 ⟶ − \@medpenalty	3 ⟶ − \@highpenalty
4 ⟶ −10 000	

\pagebreak fonctionne dans les deux modes pour le même effet. Dans le mode horizontal, elle utilise \vadjust. Contrairement à \newpage ou \clearpage, elle n'utilise pas \vfil, son propos étant surtout d'influencer le choix d'un point de rupture en terminaison de page et non de commencer une éventuelle nouvelle partie du document comme pour ces dernières commandes.

\nopagebreak[⟨*num*⟩**]**
comme son nom l'indique, \noagebreak agit de façon opposée par rapport à \pagebreak.

\samepage
c'est une commande obsolète et très capricieuse du format LaTeX2.09. En interne elle se contente de donner une valeur maximale (10 000) à certaines variables de pénalité.

\enlargethispage{⟨*skip*⟩**}**
C'est une commande LaTeX qui permet d'étendre la hauteur de la zone de texte sur la page courante. Typiquement, elle est utilisé avec des instructions de la forme

```
\enlargethispage\baselineskip
ou \enlargethispage{2\baselineskip}
```

pour gagner respectivement une ou deux lignes sur la page. Le texte supplémentaire peut très bien empiéter sur celui d'une éventuelle note de bas de page, mais les autres éléments flottants à insérer en bas de page sont bien gérés. LaTeX 2_ε présente `\enlargethispage` comme une alternative à la commande obsolète `\samepage` de l'ancien format LaTeX2.09.

`\flushbottom`

c'est une commande LaTeX qui implique qu'après son occurrence, toute page sera entièrement remplie quitte à étirer les **vskip** disponibles dans la liste verticale.

`\raggdebottom`

c'est une commande LaTeX, inverse de `\flushbottom` : le texte occupe sa hauteur naturelle sur la page. C'est de plus en plus le comportement par défaut dans les distributions LaTeX.

Les deux bascules `\raggedbottom` et `\flushbottom` redéfinissent la commande `\@textbottom` qui est placée à la fin de la boite d'une page. Pour `\flushbottom` cette commande est rendue synonyme de `\relax`. Pour `\raggedbottom`, elle est redéfinie en l'équivalent de

```
\vskip 0pt plus .0001fil
```

Ce qui va consommer toutes les composantes d'élasticité finies dans la liste, avec de fortes de chances qu'elle céde devant celles qui sont infinies. Noter qu'il n'y a aucune différence de contenu sur la page quelque soit la commande active, `\flushbottom` ou `\raggedbottom`.

Brises-page consécutifs

Toutes les commandes de rupture de page placent en fait une pénalité de valeur négative minimale dans la liste verticale. Étant une instruction résiliable, une pénalité n'est jamais placée au sommet de la liste verticale. C'est ce qui explique que deux commandes de rupture de page qui se suivent ne produisent pas un saut de deux pages. Dans le cas où cet effet est recherché, il suffit d'utiliser une instruction de la forme

```
\newpage\null\newpage
```

.

Influencer les ruptures de pages

Empêcher un saut de page en certains lieux

La mise en page verticale devrait être normalement laissée pour la dernière phase de préparation d'un document. Provoquer quelques part une

rupture de page irréversible peut donner un résultat abominable si jamais le contenu change ou que la présentation de certains éléments est modifiée. Dans cette partie, on indique quelques techniques pour imposer qu'un bloc de texte reste toujours sur la même page. La plupart de ces techniques sont destinées à une utilisation directe dans les documents finaux.

Au milieu d'un paragraphe • Si une rupture de page survient au sein d'un paragraphe qu'on préfère laisser groupé, il suffit de donner à la variable `\interlinepenalty` une valeur positive assez forte afin de décourager la rupture de page entre ses lignes

```
{\interlinepenalty=1000 un paragraphe qu'il ne faut pas
 subdiviser sur plusieurs pages ...\par}
```

Les accolades servent ici à êmpecher que le changement de la variable `\interlinepenalty` ne soit permanent et la commande `\par` à la fin du groupe sert à rendre effective la modification. Une autre solution est de mettre le paragraphe dans une `\parbox` ou une `minipage` de largeur `\textwidth`. L'incovénient dans ce cas est que `\parbox` exécute en interne la commande `\@parboxrestore` qui peut annuler tous les effets qui sont utilisé dans le texte en dehors (justification des lignes par exemple).

Juste avant une liste • Parfois un saut de page survient juste avant un environnement de liste alors qu'on ne désire pas que la liste soit détachée de son texte d'introduction. Dans ce cas, même si on place une pénalité positive forte avant la liste, TEX s'entêtera à vouloir rompre la page au même endroit. La raison en est que LATEX place une pénalité négative de valeur `\@beginparpenalty` juste avant la liste et que cette dernière va résilier celle de l'utilisateur. Plusieurs solutions peuvent être adoptées.

— Si le texte d'introduction est assez court, il suffit de placer une pénalité négative plus forte avant lui pour qu'il accompagne la liste sur la nouvelle page.

— Si ce texte est un paragraphe assez long on peut donner à la variable `\interlinepenalty` une valeur négative (dans un groupe) pour encourager la rupture au milieu du paragraphe, ou encore donner à `\@beginparpenalty` une valeur positive à l'intérieur de l'environnement de liste (en n'oubliant pas le couple de commandes `\makeatletter` `\makeatother`). Une autre solution serait de passer à la commande `\vadjust`, une péanlité suffisante pour provoquer la rupture de la page au milieu du paragraphe qui précéde la liste.

Chacune de ces propositions est une bonne alternative à la solution brutale qui consiste en l'utilisation d'une commande `\eject` ou `\newpage`.

Que faire lorsque un élément est trop haut pour l'espace restant dans la page ? • Dans certaines situations, un élément indivisible de hauteur conséquente est renvoyé sur la page suivante en laissant un vide trop important dans la page courante. La commande `\enlargethispage` est utile dans de tels cas. Si la hauteur du dit élément dépasse de peu celle de l'espace restant dans la page, il suffit d'ajouter l'espace nécessaire sur la page courante. Dans le cas contraire, on peut ajouter des petits espaces sur les pages précédentes de telle façon à libérer plus de place sur la page problématique.

`\enlargethispage` ne modifie pas `\textheight` ou toute autre representation de la hauteur de zone de texte. Elle insère immédiatement un élément flottant vide dont la hauteur est celle indiquée et provoque ensuite un saut vertical en arrière pour revenir à la position courante.

La commande `\enlargethispage` est loin d'être triviale. Elle sait gérer des situations complexes comme lorsqu'on désire maintenir un élément flottant en bas de page. Elle possède en outre une version étoilée dont le comportement est légérement différent. Sans étoile, elle ajoute de l'espace sur la page courante sans modifier d'aucune sorte l'espacement des éléments qui ne sont pas concernés par cet ajout. La version étoilée force l'utilisation des composantes **minus** dans leurs intégralité sur toute la page. Lorsque le but est de maintenir un certain contenu sur la page courante, il suffit de passer à `\enlargethispage*` une dimension qui peut être exagérée et rompre la liste là où on veut avec une commande `\pagebreak`, voir par un dosage de pénalité.

Après un saut vertical • Dans certaines situations, on peut vouloir séparer deux éléments d'un saut vertical mais empêcher qu'une rupture de page ne survienne entre les deux. Le schéma

… ⟨*contenu*⟩ …
`\vskip\myskip \nobreak`
… ⟨*contenu*⟩ …

échouera lamentablement dans cette tache car TEX pourra toujours rompre la liste au niveau du **skip** puisqu'il est précédé d'un élément non résiliable. La bonne séquence est `\nobreak\vskip\myskip` car le moteur ne peut ainsi rompre la liste au niveau du **skip** puisqu'il est précédé d'une pénalité qui est un élément résiliable, et ne peut le faire non plus au niveau de la pénalité puisque justement celle-ci l'interdit.

Supposons qu'on veuille définir un environnement qui met en valeur verticalement son contenu (un théorème par exemple) en le faisant précéder et suivre d'un filet horizontal. La définition

```
\newenvironment{encadre}
  {\hrule\vskip3pt}
  {\vskip3pt\hrule}
```

fait très bien l'affaire moyennant quelques ajustement des espaces autour des filets. Mais rien n'empêche une rupture de page juste après le premier filet ou juste avant le second. Une version qui résiste à ces ruptures et améliore le choix des sauts autour de l'environnement maintenant

```
\newenvironment{encadre}[1]
  {\par\medskip\hrule\nobreak\vskip.3\baselineskip
      \textbf{#1: }}
  {\par\nobreak\vskip.3\baselineskip\hrule\medskip}
```

```
Texte avant.
\begin{encadre}{Constat}
\LaTeX{} est un syst\'eme de traitement de
 texte programmable largement r\'epandu dans
 les milieux scientifiques.
\end{encadre} Texte apr\`es.
```

Texte avant.

Constat : LaTeX est un système de traitement de texte programmable largement répandu dans les milieux scientifiques.

Texte après.

Brises cette page... sans lui faire mal

Cette section propose quelques astuces pour provoquer un saut de page dans certaines conditions mais qui soit assez souple pour ne plus être effectif si jamais le point choisi pour la rupture n'est plus proche de la fin d'une page.

Plus souple que \pagebreak • On voudrait écrire une commande qui puisse agir dans les deux modes et qui provoque une rupture de page tout en offrant plus de souplesse que \pagebreak. Pour cela il suffit d'utiliser son argument comme valeur de pénalité brute.

```
\newcommand\nicebreak[1][351]{%
  \ifvmode \penalty-#1\relax
  \else \ifhmode \vadjust{\penalty-#1}\fi}
```

La commande \nicebreak a un paramètre optionnel dont la valeur par défaut correspond à celle prise par \@highpenalty. Ce paramètre est utilisé pour ajouter une pénalité négative dont il est la valeur absolue dans la liste verticale, quelque soit le mode actif.

Maintenant on voudrait que cette commande ait une version étoilée qui insère en plus un saut infini en fin de page à la manière de la commande \newpage. C'est l'occasion de donner un exemple d'utilisation de la commande de test \@ifstar.

```
\def\nicebreak{\@ifstar\nicebreak@\nice@break@@}
\newcommand\nicebreak@@[1][351]{
  \ifvmode \penalty-#1\relax
  \else \ifhmode \vadjust{\penalty-#1}\fi}
\newcommand\nicebrak[1][351]{
  \vskip 0pt plus .0001fil%
  \nicebreak@@[#1]}
```

`\nicebreak` est la commande utilisateur. Si elle est suivie d'un caractère `*` elle se contente d'exécuter la commande interne `\nicebreak@`, sinon la commande `\nicebrak@@`. Ce sont ces deux dernières commandes qui font tout le travail, y compris la gestion du paramètre optionnel.

Faux Displaymode • Si on veut créer un environnement LaTeX qui ne provoque pas la sortie du mode horizontal (rester dans le même paragraphe) tout en effectuant des sauts verticaux avant et après le contenu (à la manière du mode **displaymode**), une bonne idée est d'utiliser la primitive `\vadjust`, en n'oubliant pas d'y insérer aussi une instruction `\nobreak` là on ne souhaite pas une rupture de page. On pourrait, par exemple, définir une commande

```
\newcommand\vhaddskip[1]{%
\ifvmode%
   \vskip #1\relax%
  \else%
     \unskip\hfil\nobreak% simule une fin de paragraphe
      \vadjust{\nobreak\vskip #1}%
  \fi}
```

qui sera exécuté en premier une fois l'environnement en question actif. La commande `\vhaddskip` détermine le mode actif, dans le mode vertical elle se comporte comme `\vskip` et dans le mode horizontal, elle simule une fin de paragraphe et place un **vskip** dans la liste verticale sans quitter ce mode. Pour mieux gérer les espaces de séparation dans le cas où deux occurrences de cet environnement se succèdent, on peut utiliser la commande `\addvspace` au lieu de `\vskip`.

Saut conditionnel • les variables `\pagetotal` et `\pagedepth` peuvent être utiles pour créer une commande qui provoque une rupture de page si l'espace restant sur la page est inférieur à une dimension donnée. Une telle commande peut servir pour insérer un titre, en évitant qu'il se retrouve orphelin en fin de page.

```
\newlength\pagereste
\newcommand\condbreak[1][.33\textheight]{%
```

```
  \setlength\pagereste\textheight%
   \advance\pagereste-\pagetotal%
    \advance\pagereste-\pagedepth%
   \ifdim\pagereste < #1 \pagebreak\fi}
```

\condbreak initie la variable de dimension \pagereste avec la hauteur de la zone de texte et la diminue ensuite de \pagetotal et de \pagedepth, de sorte qu'elle contienne l'espace naturel restant sur la page. Le test final provoque une rupture de page dans le cas où cet espace est plus petit que celui passé comme un argument optionnel à \condbreak (dont la valeur par défaut est le tier de la hauteur de la zone de texte). L'avantage est que \condbreak ne provoque plus un saut de page dans le cas où, suite à des modifications dans le document, plus d'espace devient disponible.

Si on veut qu'en plus la commande insère un saut vertical, on n'a qu'à la redéfinir de la manière suivante

```
\newlength\pagereste
\newcommand\condbreak[2][.33\textheight]{%
  \setlength\pagereste\textheight%
   \addtolength\pagereste{-\pagetotal}%
    \advance\pagereste-\pagedepth%
 \ifdim\pagereste < #1
      \clearpage
   \else%
        \vskip #2%
    \fi}
```

L'espace à insérer fait maintenant office de paramètre obligatoire de cette nouvelle version.

L'opération \vsplit

Les boites TEX sont indivisibles, qu'elles soient horizontales ou verticales. Pourtant, au moment de former une page, le moteur place le contenu de la liste « current page » dans la boite \box255 de hauteur \vsize. Si le contenu ne peut tenir sur la page, TEX fait appelle à la commande \vsplit pour diviser la boite en deux parties, celle qui sera retenu pour la formation de la page et l'autre qui sera renvoyée vers la liste « recent contributions ».

La commande \vsplit est disponible à l'utilisation en dehors de ce procédé et peut avoir des applications non triviales.

Quand ⟨*reg*⟩ est le numero ou le nom d'un regsitre de boite qui a recu une boite **vbox** alors l'instruction

\setbox⟨*regprime*⟩=\split⟨*reg*⟩ to ⟨*dim*⟩

va rompre la liste des éléments contenus dans ⟨*reg*⟩ en deux parties selon la même méthode qui abouti à la formation d'une page. La partie supérieure est affectée au registre ⟨*regprime*⟩ avec une hauteur de dimension ⟨*dim*⟩ et la seconde partie est conservée dans le registre original ⟨*reg*⟩. Les boites ainsi formées ont les propriétés suivantes :

— La hauteur de la boite \box⟨*regprime*⟩ est ⟨*dim*⟩ quelque soit le contenu retenu.
— La boite \box⟨*regprime*⟩ ne peut dépasser en profondeur la valeur de la variable \splitmaxdepth. Ce qui n'empêchera pas la boite d'être en situation d'« Overfull \vbox » si jamais TEX n'arrive pas à rompre la liste correctement à la hauteur précisée (à cause d'une pénalité trop forte par exemple).
— Tout élément résiliable est éliminé du début de \box⟨*reg*⟩ et le ressort \splittopskip est utilisé de la même façon que \topskip lorsqu'il s'agit de former une page.
— Quitte à déplacer verticalement le point de référence de la boite \box⟨*reg*⟩, sa profondeur vaut au maximum \splitmaxdepth.
— La fin de la boite originale est un point de rupture valable. Si la rupture s'y produit effectivement alors ⟨*reg*⟩ est libéré. Le registre ⟨*regprime*⟩ peut être vide mais il n'est libre que si le registre original est libre.

Quand il s'agit d'appliquer ce procédé à \box255, \splitmaxdepth et remplacé par la variable \maxdepth et \splittopskip par \topskip. Les formats plainTEX et LATEX donnent à \maxdepth et \splitmaxdepth la valeur de \maxdimen et à \topskip et \splittopskip la valeur 10pt.

Considérons l'affectation

```
\setbox0=\vbox{Premi\‘ere ligne
\[\int_0^{1}x\,\mathrm dx=\frac12\]\par
 Fin de la boite.}
```

et analysons les différentes manières d'éclater la boite en deux morceaux avec \vsplit et de reconstituer ensuite le texte original en concaténant le contenu des deux boites. Rappelons que le registre qui reçoit le résultat d'une opération \vsplit n'est libre que si le registre original est libre. Une instruction \vsplit to 0pt dont le but est d'isoler le première ligne de la boite a donc bien un sens à priori.

```
Avant \verb+\vsplit+~:\par
ht0: \the\ht0 \quad dp0: \the\dp0\par
tht0 : \the\dimexpr\ht0+\dp0\relax
\setbox1=\vsplit0 to 0pt\par
\setbox2=\vbox{\unvcopy1}
Apr\‘es \verb+\vsplit+~:\par
ht1: \the\ht1 \quad dp1: \the\dp1\par
ht0: \the\ht0 \quad dp0: \the\dp0\par
tht0+tht1: \the\dimexpr
 \ht1+\dp1+\ht0+\dp0\relax\par
\box1\box0\bigskip
```

Avant \vsplit :
ht0 : 68.1241pt dp0 : 0.12592pt
tht0 : 68.25002pt
Après \vsplit :
ht1 : 0.0pt dp1 : 9.76115pt
ht0 : 10.0pt dp0 : 0.12592pt
tht0+tht1 : 19.88707pt

Première ligne
Fin de la boite.

$$\int_0^1 x\,\mathrm{d}x = \frac{1}{2}$$

COMMENTAIRES : L'intention avec `\vsplit to 0pt` est de mettre juste la première ligne dans `\box1`, mais `\predisplaypenalty` interdit la rupture avant l'équation et `\box1` contient également celle-ci, se retrouvant ainsi en `Overfull \vbox`. La différence (énorme) entre la hauteur totale de la boite orginale et la somme de celles des deux boites est perdue dans la profondeur de `\box1`. Ce qui explique le chevauchement des deux boites à la fin, comme si `\box1` ne contenait que la première ligne.

```
\setbox1=\vsplit0 to 0pt
ht1: \the\ht1 \quad dp1: \the\dp1\par
\setbox1=\vbox{\unvcopy1}
ht1: \the\ht1 \quad dp1: \the\dp1\par
\box1\box0
```

ht1 : 0.0pt dp1 : 9.76115pt
ht1 : 38.8403pt dp1 : 9.76115pt
Première ligne

$$\int_0^1 x\,\mathrm{d}x = \frac{1}{2}$$

Fin de la boite.

COMMENTAIRES : On rétablit la hauteur et la profondeur du contenu de `\box1` en le déballant dans le même registre `\vbox`. La rupture a eu lieu avant le saut qui vient après l'equation et celui-ci disparaît du début de la nouvelle boite `\box0`, d'où les espaces non équilibrés.

```
\predisplaypenalty0
\setbox0=\vbox{Premi\‘ere ligne
\[\int_0^{1}x\,\mathrm dx=\frac12\]\par
 Fin de la boite.}
\setbox1=\vsplit0 to 0pt
\setbox1=\vbox{\unvbox1}
\box1\box0
```

Première ligne

$$\int_0^1 x\,\mathrm{d}x = \frac{1}{2}$$

Fin de la boite.

COMMENTAIRES : On annule `\predisplaypenalty` pour permettre la rupture avant l'équation, mais cette fois c'est le saut avant l'équation qui disparaît du début de `\box0`.

Une astuce permet de reconstituer les espaces verticaux de la liste originale. Il s'agit de calculer la différence entre la hauteur totale de la boite

originale avec la somme de celles des deux boites résultats de l'opération et d'insérer ensuite, entre les deux boites, un saut dont la dimension est cette différence.

```
\newdimen\dimperdue
\dimperdue=\dimexpr\ht0+\dp0\relax
\setbox1=\vsplit0 to 0pt
\setbox1=\vbox{\unvbox1}
\dimperdue=\dimexpr\dimperdue-\ht1-\dp1
   -\ht0-\dp0\relax
\box1\nointerlineskip\vskip\dimperdue\box0
```

Première ligne

$$\int_0^1 x\,\mathrm{d}x = \frac{1}{2}$$

Fin de la boite.

eTEX possède des primitives qui permettent d'automatiser cette technique. Il s'agit de la variable `\savingvdiscards` qui, si elle recoit une valeur non nulle, informe le moteur d'enregistrer les éléments éventuellement résiliés (sauts et pénalités) de la dernière opération `\vsplit`, la primitive `\splitdiscards` placé au bon endroit permet alors de rétablir ces éléments.

```
\savingvdiscards=1
\setbox1=\vsplit0 to 0pt
\setbox1=\vbox{\unvbox1}
\box1\nointerlineskip\splitdiscards\box0
```

Première ligne

$$\int_0^1 x\,\mathrm{d}x = \frac{1}{2}$$

Fin de la boite.

Tout cela est bien compliqué et on ne peut que se poser des questions sur l'utilité de `\vsplit` en dehors de son utilisation par le moteur pour former les pages. Le fait est que l'usage de `\vsplit` est la seule technique qui permette de produire une décoration autour d'un contenu tout en permettant que celui-ci s'étale sur plusieurs pages. Les packages LaTeX qui vont dans ce sens placent souvent le contenu qu'il traitent dans un registre `\vbox`, l'éclatent si nécessaire en plusieurs morceaux, parfois ligne par ligne, et appliquent individuellement l'effet désiré aux différents morceaux. La difficulté est alors de reconstituer fidèlement tous les éléments de la boite originale, pénalités et sauts compris. La plupart de ces packages ne savent toutefois pas gérer les situations vraiment complexes où le texte traité contient des éléments flottants.

Les packages LaTeX `framed`, `mdframed` *qui repose sur le premier et* `tcolorbox` *permettent de produire des décorations autour d'un texte sans interdire que la page soit rompue en son sein.*

Les listes selon LaTeX

Introduction

LaTeX construit toute une infrastructure pour la création de diverses sortes d'environnement de liste, `itemize`, `enumerate` voire `description` n'en étant que des applications. À la base de cette infrastructure, les deux environnements de bas niveau `trivlist` et `list` ainsi que la commande non triviale `\item`. Le propre des environnements de liste est de gérer les « marges » et les espaces de séparation verticale (en plus de la formation des labels des items), en cela ils sont plus ou moins une façon de formater les paragraphes. Alors que `list` est utilisé pour créer des environnements pour les listes à proprement parler (et encore), `trivlist` sert dans la définition d'environnements divers qui ont en commun d'opérer dans un mode que LaTeX qualifie de **DISPLAY**. Il s'agit de généraliser le comportement du mode mathématique **displaymode** : mettre en valeur verticalement du contenu tout en continuant à adopter les normes du mode horizontal. Par exemple, le « paragraphe » qui vient après un environnement à base de `trivlist` (ou de `list`) n'est pas indenté s'il suit immédiatement l'environnement, sans ligne vide entre les deux. Les environnements `center`, `flusleft`, `flushright`, `verbatim`, `tabbing` et tout environnement crée avec `newtheroem` utilisent tous `trivlist`. C'est même le cas de l'environnement pour équation centrée de LaTeX dans le cas où l'option de classe `fleqn` est active. Par exemple, la définition de `center` se résume à

*Quand l'option de classe `fleqn` est utilisée, les formules **displaymode** traitée avec le mécansime LaTeX (`\[... \]`) sont alignées à gauche avec une indentation.*

```
\def\center{\trivlist \centering\item\relax}
\def\endcenter{\endtrivlist}
```

le but de la manœuvre étant de laisser la commande `\centering` s'occuper de l'effet « texte centré » mais de confier la gestion des espaces verticaux

autour de l'environnement à trivlist.

Les environnements list et trivlist partagent en interne les mêmes conventions pour la gestion des espaces verticaux. Les différences essentielles entre les deux résident dans le fait que trivlist ignore les retraits de marge et que sa gestion des labels des items se résume en leurs restitution dans le même style que le texte normal. En outre, list contrôle le niveau d'imbrication des environnements et ne permet pas de dépasser le niveau 5 quelque soit l'incarnation sous laquelle il est utilisé. Il n'y a aucun contrôle de la sorte avec trivlist.

Les détails

Dans la suite on donne la liste des paramètres numériques et de certaines commandes qui influencent le formatage du texte dans un environnement de liste, qu'il soit basé sur list ou sur trivlist.

Les espaces verticaux (trivlist et list)

\topsep espace ajouté juste avant et juste aprés le contenu de l'environnement ;

\partopsep espace ajouté à \topsep là où il y a une commande \par (ou une ligne vide) ;

\itemsep espace entre deux items ;

\parsep espace qui remplace \parskip entre les paragraphes d'un même item.

Signalons que tous les espaces verticaux autour et au milieu d'un environnement de liste impliquant ces dimensions sont gérés par la commande \addvspace et que celle-ci est en fait liée historiquement à ces environnements. C'est ainsi que deux environnements de liste qui se suivent ne vont pas cumuler leurs espaces de séparation verticale.

Les pénalités verticales (trivlist et list)

\@beginparpenality pénalité ajoutée juste avant le contenu de l'environnement ;

\@endparpenalty pénalité ajoutée juste après le contenu de l'environnement ;

\@itempenalty pénalité ajoutée juste avant un item.

Ce sont des variables utilisées pour gérer les ruptures de pages en leurs lieux d'insertion. Elles reçoivent toutes la valeur négative −\lowpenalty (−51 dans les fichiers de classe standard). Les pénalités qui leurs sont associée ont donc tendance à encourager une rupture de page. Signalons qu'en plus de ces variables, la commande \item modifie localement \clubpenalty pour que jamais la première ligne d'un item ne se retrouve seule en fin de page.

Les espaces horizontaux (list seulement)

\leftmargin marge à gauche du texte de l'environnement par rapport au texte normal, elle est cumulable s'il y a imbrication des environnements ;

\rightmargin la même chose, mais pour la marge à droite ;

\listparindent espace qui remplace \parindent pour l'indentation des paragraphes qui ne commencent pas un item ;

\itemindent indentation juste avant le label d'un item ;

\labelsep espace entre le label de l'item et le texte qui le suit ;

\labelwidth largeur de la boite qui contient le label, mais si le label dépasse \labelwidth alors il est produit avec sa largeur naturelle.

Les dimensions \leftmargin et \rightmargin sont utilisées indépendamment de \leftskip et \rightskip. D'ailleurs, ces dernières sont systématiquement annulées à l'entrée dans un environnement de liste et à moins de de les modifier à l'intérieur de l'environnement leur effet n'est plus valable. C'est la raison pour laquelle, le contenu d'une liste n'est pas du tout centré à l'intérieur d'un environnement center par exemple et que le même constat tient pour les environnement flusleft et flushright.

```
\begin{center}\begin{itemize}
\item Ce texte n'est pas centr\'e.
\end{itemize} Contrairement \`a celui-ci.
\end{center}
```

— Ce texte n'est pas centré.
Contrairement à celui-ci.

La dimension \hsize *n'est par contre jamais altérée dans* list *et* trivlist.

Contrairement à la façon dont sont utilisés \leftskip et \rightskip, \leftmargin et \rightmargin déterminent la largeur de la ligne \linewidth : à chaque entrée dans un environnement list, \linewidth est diminué de la valeur courante de \leftmargin et de \rightmargin et la valeur de \leftmargin est ajoutée à la variable interne \@totalleftmargin. LaTeX utilise ensuite la primitive \parshape pour formater tout le contenu de l'environnement avec

```
\parshape 1 \@totalleftmargin \linewidth
```

Cette instruction n'est pas utilisé dans trivlist, c'est ce qui fait que les dimensions \leftmargin et \rightmargin n'y ont aucun rôle. Cela implique aussi qu'une fois un environnement list initié, modifier ces dimensions n'aura aucun effet non plus. Ce n'est pas le cas des dimensions \labelwidth, \labelsep et \itemindent, leur altération prend effet n'importe où dans l'environnement car elles ne sont utilisées que par la commande \item.

```
hsize: \the\hsize\\ linewidth: \the\linewidth
\begin{itemize}
\item hsize: \the\hsize\\
linewidth: \the\linewidth\\
linewidth+leftmargin:
\the\dimexpr\leftmargin+\linewidth
\end{itemize}
```

hsize : 170.71652pt
linewidth : 170.71652pt

— hsize : 170.71652pt
linewidth : 154.59163pt
linewidth+leftmargin : 170.71652pt

La syntaxe

`trivlist` s'utilise de la faqn suivante,

```
\begin{trivlist}
    \item[⟨Label Opt⟩] ⟨contenu⟩
    ...
\end{trivlist}
```

`trivlist` est plus « compliquée »

```
\begin{list}{⟨LABEL⟩}{⟨paramètres⟩}
    \item[⟨Label Opt⟩] ⟨contenu⟩
    ...
\end{list}
```

⟨*Label Opt*⟩ est un paramètre optionnel qui indique le label à produire. En cas de son absence la valeur par défaut selon l'environnement est produite. L'environnement `list` utilise deux paramètres obligatoires :

⟨*LABEL*⟩ indique le label par défaut à utiliser pour les items. Ce doit être en général une commande qui va générer le texte du label, par exemple `\the`⟨*COMPT*⟩ ou ⟨*COMPT*⟩ est un compteur LaTeX déclaré ;

⟨*paramètres*⟩ tout code qui a un sens ici : fixation des paramètres numériques de la liste ; redéfinition de la commande `\makelabel` qui se charge du formatage du label ; précision du compteur à utiliser via `\usecounter`…

Rappelons que `list` et `trivlist` servent à définir de nouveaux environnements, ils ne sont pas destinés à être utilisés directement dans les documents finaux.

Les commandes utilisables avec `list`

`\makelabel` C'est la commande qui s'occupe du formatage du label. Sa définition par défaut sous `trivlist` est simplement

```
\def\makelabel#1{#1}
```

et sous `list` elle est par défaut synonyme de la commande `\@mklab` qui est définie par

```
\def\@mklab#1{\hfil #1}
```

Dans le cas d'un environnement défini par `list`, son sens peut être redéfini dans le deuxième paramètre de `list`. Par contre en dehors de tout environnement de liste elle est définie de telle façon qu'elle produise le fameux message d'erreur

```
Lonely \item--perhaps a missing list environment
```

message qui est déclenché lorsque `\item` est utilisée en dehors d'un environnement de liste.

`\usecounter` indique un compteur LaTeX qui sera remis à zéro à l'ouverture de l'environnement puis incrémenté à chaque `\item`. C'est au premier paramètre obligatoire de l'environnement `list` de faire usage du compteur ainsi déclaré, mais la commande `\usecounter` en elle même doit être utilisée dans le deuxième paramètre.

`\@listdepth` compteur TeX qui peut être utilisé pour connaître le niveau d'imbrication de l'environnement `list` en cours. S'il est plus grand strictement que 5, `list` provoque l'erreur « `Too deeply nested` ».

`\@listctr` macro qui contient le nom du compteur LaTeX utilisé pour numéroter les items pour le niveau courant.

`@nmbrlist` variable booléenne qui est vraie si l'environnement en cours utilise des items numérotés. Elle peut être utilisée dans une instruction conditionnelle qui commence avec `\if@nmbrlist`.

La commande `\item` : son rôle ne se limite pas à produire le label de l'item, elle est responsable de beaucoup de choses dans un environnement de liste. Le label peut être optionnel, sa valeur par défaut étant vide dans `trivlist` et configurable dans les paramètres de `list`. Elle produit une erreur si elle est utilisée en dehors d'un environnement de liste ou lorsque dans un tel environnement on insère du matériel horizontal avant la première commande `\item`. Dans les faits, `\item` se contente d'ajouter le label dûment formaté avec `\makelabel` avec les espaces horizontaux nécessaires dans le registre de boite `\hbox \@labels`. Plus exactement, `\item` place un espace de dimension

`\item` programme `\everypar` pour qu'elle insère le contenu de `\box\@label` et qu'elle se vide ensuite elle même pour les paragraphes qui vont suivre.

```
-\labelwidth - \labelsep + \itemindent
```

avant le label formaté et de `\labelsep` après lui. Le contenu du registre `\@labels` n'est produit que lorsque TeX entre dans le mode horizontal ou qu'une autre commande `\item` est rencontrée. C'est ce qui explique par exemple qu'une séquence `\item\vskip1pc` va produire un espace vertical d'un pc avant l'item et son lablel ou qu'en général toute commande non horizontale qui suit immédiatemment `\item` se voit prendre effet avant l'insertion du label de l'item.

```
\begin{itemize}
\item \hbox{Du texte en premier sans
 \verb+\item+.}
ou comment mettre \‘a genoux les d\’efenses de
 l’environnement.
\end{itemize}
```

Du texte en premier sans \item.
— ou comment mettre à genoux les défenses de l’environnement.

En dehors du fait qu’elles préparent le contexte de formatage des paragraphes, les commandes \trivlist et \list n’ajoutent aucun matériel à la liste verticale en cours, c’est à peine si elles exécutent une commande \par pour fermer le paragraphe précédent. C’est à la première commande \item de l’environnement qu’incombe la tache de placer le saut vertical adéquat et la pénalité \@beginparpenalty.

Des exemples

Encore une fois, pour savoir comment utiliser les environnements list et trivlist, regardons comment LaTeX en fait usage dans quelques exemples.

TP **L’inspiration d’une description** • description est un environnement de liste dont les labels des items sont traités comme des titres.

```
\newenvironment{description}
    {\list{}{\labelwidth\z@ \itemindent-\leftmargin
                \let\makelabel\descriptionlabel}}
    {\endlist}
\newcommand*\descriptionlabel[1]{\hspace\labelsep
     \normalfont\bfseries #1}
```

Ainsi description initie un environnement list avec un label par défaut vide pour les items, le réglage des variables \labelwidth et \itemindent fait prendre à la boite contenant le label sa largeur naturelle et la place en retrait de −\leftmargin−\labelsep de la position courante. La commande \makelabel, synonyme de la commande \descriptionlabel définie à l’extérieur, place un saut de dimension \labelsep avant le label ce qui fait qu’au final, celui-ci commence exactement là où commence le texte avant l’environnement.

```
\begin{description}
\item [Un premier item] avec un titre et une
 description~;
\item [Un deuxi\‘eme item] sans plus.
\end{description}
En dehors de toute \’enum\’eration.
```

Un premier item avec un titre et une description ;

Un deuxième item sans plus.

En dehors de toute énumération.

Inspirons nous maintenant de cette définition pour créer un environnement de liste dans lequel il n'y a aucune marge et où les labels sont encadrés, occupent toujours le même espace et commencent exactement là où commence le texte.

```
\newdimen\questionwidth
\newenvironment{corrige}
    {\list{}{\leftmargin0pt\labelwidth0pt\itemindent\labelsep
     \let\makelabel\questionlabel}}
    {\endlist}
\newcommand*\questionlabel[1]{\fboxsep=2pt\strut\fbox{%
   \makebox[\questionwidth+12pt]{\bfseries#1}}}
\def\question#1{\item[#1]}
```

```
\settowidth{\questionwidth}{III.8.a}
\begin{corrige}
\question{II.1} Cette question est si
 difficile qu'elle n'aura pas de r\'eponse
 dans ce corrig\'e.
\question{II.2.a} $\mathrm e^{i\pi}=-1$.
\end{corrige}
```

II.1 Cette question est si difficile qu'elle n'aura pas de réponse dans ce corrigé.

II.2.a $e^{i\pi} = -1$.

`trivlist` pour les maths ! • Voici maintenant comment est définie la commande `\[`, qui initie le mode **displaymode** de LaTeX, quand l'option de classe `fleqn` est utilisée

```
\newdimen\mathindent
\AtEndOfClass{\mathindent\leftmargini}
\renewcommand\[{\relax
    \ifmmode\@badmath
    \else
      \begin{trivlist}%
      \@beginparpenalty\predisplaypenalty
      \@endparpenalty\postdisplaypenalty
      \item[]\leavevmode
      \hb@xt@\linewidth\bgroup $\m@th\displaystyle %$
      \hskip\mathindent\bgroup
    \fi}
```

La dimension `\leftmarginі` est la valeur de `\leftmargin` pour le premier niveau d'imbrication de l'environnement `list`. Elle est fixée par le fichier de classe, d'où l'usage de `\AtEndOfClass` pour donner cette valeur à la dimension `\mathindent` qui va servir d'indentaion à l'équation. Si le mode

mathématique est déjà actif, la macro `\@badmath` provoque un message d'erreur selon lequel `\[` ne devrait pas être utilisée dans ce mode. Dans le cas contraire un environnement `trivlist` est initié et les pénalités verticales reçoivent les valeurs normalement utilisées dans le **displaymode**. La macro `\fb@xt@` est un raccourci vers « `\hbox to` » et `\m@th` annule `\mathsurround`. La commande `\leavevmode` déclenche le mode horizontal après `\item[]`, pour préparer l'insertion de la `\hbox` suivante dans ce mode, `\hbox` qui obligera l'équation à tenir sur une seule ligne.

Question : que fait la derniere commande `\bgroup` dans cette définition ? [1]

La commande `\]` qui ferme le mode est définie par

```
\renewcommand\]{\relax
  \ifmmode
    \egroup $\hfil% $
     \egroup
    \end{trivlist}%
  \else \@badmath
\fi}
```

Elle se contente de fermer les groupes ouverts par `\]` et de declencher une erreur si jamais elle se trouve en dehors du mode mathématique.

Notons que les deux commandes négligent d'utiliser les espaces verticaux propres au mode **displaystyle** (`\abovedisplayskip` et compagnie) et se contente d'utiliser plutôt ceux de l'environnement `trivlist`

Comme application, définissons une commande qui va prendre trois arguments, une équation à centrer et deux blocs de texte qui seront insérés des deux côtés de l'équation, en ne se souciant pas d'un éventuel cheveauchement entres les trois blocs. Un style un peu original mais qui peut s'avérer très pratique dans certaines situations.

```
\newcommand\leftcentersright[3]{%
  \begin{trivlist}%
    \@beginparpenalty\predisplaypenalty
    \@endparpenalty\postdisplaypenalty
    \item[]\makebox[0pt][l]{#1}\hfill
      \mbox{$\displaystyle#2$}
     \hfill\makebox[0pt][r]{#3}
  \end{trivlist}}
```

1. **Réponse :** Elle initie un groupe qui va récolter toute l'équation. De cette façon les ressorts de la liste mathématique seront rigides et l'équation occupera sa largeur naturelle

Les deux commandes `\makebox` insèrent des boites de largeur nulle en alignant leurs contenu respectivement à droite et à gauche de leurs positions courantes, à savoir le début et la fin de la ligne. Il est inutile ici d'utiliser `\leavevmode` aprés `\item[]` puisque `\makebox` est une commande horizontale.

La commande ainsi définie respecte les conventions du mode **dsiplaymode** (mais pas ses sauts verticaux), en particulier son contenu est toujours sur la même page que le texte qui le précéde et le texte qui le suit n'est pas indenté s'il n'est pas lui même précédé d'une ligne vide. On profite aussi d'une gestion intelligente des espaces verticaux autour de ce contenu lorsque deux occurrences de `\leftcentersright` se suivent.

```
\leftcentersright{Finalement,}
{\int_0^{+\infty}t^n\mathrm e^{-t}\mathrm dt=n!}
{(E.1)}
```

Finalement, $$\int_0^{+\infty} t^n \mathrm{e}^{-t} \mathrm{d}t = n! \tag{E.1}$$